API Misuse Detection Based on Stacked LSTM

Shuyin OuYang[1], Fan Ge[1], Li Kuang[1(✉)], and Yuyu Yin[2]

[1] School of Computer Science and Engineering, Central South University, Changsha 410075, HN, China
kuangli@csu.edu.cn

[2] Hangzhou Dianzi University, Hangzhou 310027, ZJ, China

Abstract. In modern software engineering, API (Application Programming Interface) is widely used to develop applications rapidly by reusing data structure, frameworks, class libs, and etc. However, due to the considerable number of interfaces, lack of documents and timely maintenance and updates, APIs are often used in a wrong way. Therefore, it has become an important problem to detect API misuse in an automatic way. Many existing automatic API detecting methods do not make full use of APIs' potential semantic information and independent integrity of each API. In this paper, we employ Stacked LSTM to learn the API usage specification to detect the API misuse defects. Specifically, first, we obtain ACSG (API Call Syntax Graph) through the static analysis of source code. And then, based on ACSG, we generate API sequences, and transform the sequences into <precious API sequence, next API> for training. Third, in order to represent the APIs in a semantic way, we apply word2vec as a pre-training model to embed features of each API. Though the stacked LSTM model, we regard embedding precious API sequence as the input to model the API use specifications and discover the potential API misuse defects by judging whether the next API is in the output (API probability list) or not. We design experiments to evaluate the effectiveness our method with Java Cryptography APIs and their used code, and the results show the advancement of our proposed method.

Keywords: API misuse detection · Static analysis · Pre-training model · Semantic representation · LSTM

1 Introduction

Nowadays, API (Application Programming Interface) plays an important role in software developing. The developers can save a lot of time to access libraries and frameworks via APIs. According to the ProgrammableWeb's reports, the number of APIs recently included in its website has exceeded 23,403. Since 2015, an average of more than 2,000 APIs has been added per year. As for the first six month of 2019, the API directory has seen over 1,320 new APIs added, which has an increase of over 30% over the previous years. More and more enterprises choose reusing existing data structure, frameworks and class libs to complete the rapid development of new software projects.

H. Gao et al. (Eds.): CollaborateCom 2020, LNICST 349, pp. 421–438, 2021.
https://doi.org/10.1007/978-3-030-67537-0_26

Although API reduces projects' development effort by accessing underlying services, it brings the challenges for ensuring the correctness of programs. API misuse often occurs in the projects, which includes using redundant APIs, using wrong APIs, missing key APIs, ignoring the handling of exceptions that may be thrown in some APIs, which may cause defects or even serious security problems. API misuse is an inevitable problem while developers are using APIs to develop projects. Generally speaking, API misuse is caused by the following reasons: (1) There are lots of APIs and usually an application involves various API interactions. For example, Java Encache API is used to create a cache between distributed nodes, and the interaction between distributed nodes requires the support of Java RMI (remote call) API. (2) The quality of API documentation is not good enough, and the low-level documentation is the main obstacle to API learning and using. (3) API need to be continuously upgraded and maintained, but the corresponding API documentation is not updated synchronously in time.

To address the issues of API misuse detection, the existing methods can be grouped as two kinds. The first kind is utilizing API's semantic information. Inspired by Natural Language Processing, software engineering starts to take "naturalness" of code into consideration. Many studies apply the powerful models in Natural Language Processing area into code analysis, code completion and bug detection. They discover the potential semantic features of underlying codes and take advantage of these features to make contributions for research on source code. For example, some studies finished their work based on n-gram statistical language model to solve code completion problem [26–28]. However, due to the fact that the syntax of the code is the rule of communicating with machines, there are also some unique attributes among code structures. Thus, the second kind is discovering code's grammatical structure. There are many studies to learn the usage specifications of code through structure and pattern mining [6–12, 19–25]. Applications of above methods are common in software repositories mining, documentations, API patterns, summarization and anomaly detection. However, there are following challenges existed in these methods: While analyzing source code structure, people often tend to mine the connection relation between APIs and use pattern matching to find misuse of code structure. The API's semantic information is often ignored.

In this paper, we employ Stacked LSTM to learn the API usage specification by feeding API sequences with semantic representation, so as to detect the API misuse defects. First, we perform a static analysis of Java code files which aims to obtain ACSG (API Call Syntax Graph) of each file. Then, we design a mining algorithm to get API sequences on ACSGs and progressively transform the sequences generated into previous API sequences and corresponding next API, which are regarded as the input and output of our model proposed later respectively. In order to utilize the properties of the API itself, we apply word2vec to acquire the semantic representation of each API by mapping the relation between APIs and API sequences into the relation between words and sentences. Next, we learn API usage specification by Stacked LSTM, which generates the next API probability list as output. We determine API misuse by checking whether the actual API is contained in the prediction list or not. Finally, we verify our method by compared with the existing method. Two experiments are conducted, using the precision, recall rate, and F1 score as the criteria. The results show that our proposed model is the best among the experimental models.

This paper is organized as follows. Section 2 introduces the related works about Service adaption and API misuse detection. Section 3 presents the overall process and its implementations. Section 4 introduces the design and analysis of the experiment designed with two schemes of model. Finally, in Sect. 5 we summarize and prospect this paper.

2 Related Work

In this section we review some of related work in the area of API misuse detection. We simply divide the existing studies into two parts, namely language models and specification mining and defect detection.

Utilizing Semantic Information. According to "The naturalness hypothesis" [1], software corpuses have similar statistical properties to natural language corpuses. Codes in software can be regarded as sequences of token, Thus, the model applied in Natural Language Processing can be also used for solving problems in API misuse detection area.

Recently, language models have been used successfully in tasks including code completion fault localization [2, 4] and code conventions checking [3]. Pu et al. [13] treated the program statement as a sequence of tokens, and they propose purely syntactic learning, through which the candidate missing or fault program statement can be generated as one token at a time. Via modifying and training the seq2seq neural network model, they achieve the goal of finding and recovering the defects in MOOCs. Raychev et al. [15] designed a scalable analysis to extract sequence of method calls. They abstract the sentence with holes from the partial program with hole at event-level, where event is the semantics representation of source codes. In order to discover the completions for the code misuses, they employ statistical language model to find the highest possible bug-occurred sentence. Ray et al. [5] proposed a cache language model to extend the traditional language models by applying an addiction cache to capture the regularities in local context. Then, they apply the bug-finder to detect the lines of code where are more likely to detect defects. Tu et al. [18] proposed "source code is localized", which shows that source code tends to take semantic features in local contexts. In order to achieve the goal of code suggest, they recommend the next token based on the current context. The cache language model they applied are capable of this task.

We can see from these studies above that it is very useful to applying Natural Language Processing models to API misuse detection. On the other hand, the "localness" in source code, which refers to the complexity of code structures, still is the chronic and main challenge for API misuse detection when applying language models.

Exploring Code's Structure. Code can be considered as a kind of language to communicate with each other through a specific compiler environment. Specification mining and defect detection aims to find a finite set of patterns from source code, which consists of part of human-interpretable behaviors. And it can present the mined patterns to software engineers without annotation or supervision.

Many algorithms and techniques have been proposed for programming rule mining and misuse detection. [6–12, 14, 19–25] Oh et al. [6] presented a method for building an adaptive static analyzer. Via Bayesian optimization, they learn a good sophisticated parameterized strategy for discovering the specifications from the real-world C programs code structures. In order to solve the problem of assessing final students' code, Piech et al. [7] introduced a neural network to model students' programs as linear maps of their code structures. Then, the feedback algorithm makes use of these maps to find the misuses for Stanford University's CS1 course's students' code assignments. Wang et al. [12] designed Bugram for bug detection. Based on the assumption on specification that the API call token is only related to the n tokens before it, the occurrence probability of the API call token sequence appearing in the software project is calculated. Though linking the occurrence of the token sequence with API misuse, Bugram achieves the goal of automatic defect detection. Wang et al. [25] set up the recurrent neural network to learning API use specification. Their study makes a context-based prediction on the API code, and finds out the potential API misuse by comparing the prediction results with the actual code.

The existing specification-based code defects detection methods show great performance on analyzing structural information on real-world codes. Therefore, learning API usage specification is helpful for API misuse detection.

3 Method

We design a new API-misuse detect process which is shown in Fig. 1, that makes full use of the strengths of previous study of API-misuse and figures out the shortcomings summarized above. As shown in Fig. 1, the specific steps are listed as follows:

- Static Analysis. We design API Call Syntax Graph (ACSG), a presentation of API usage which can capture the order between API calls and data interactions which can distinguish misuses from correct usages.
- Data Generation. We design a new API call sequences mining algorithm, which can generate all the API call sequences into <precious API sequence, next API>. Through learning the API usage specification, we apply Word2Vec as a pre-training model to achieve representation for each API which can make use of API semantic features among API sequences.
- Model Training and Prediction. We design Stacked LSTM model for next API call prediction on the basis of previous API call sequences.

3.1 Static Analysis

Abstract syntax tree (AST) is a power tool to map the Java code into a Tree data structure. We use Javaparser to parse out the structure of the source Java code. Amann et al. [32] found the AUG (API-Usage Graph) as a presentation of API usages, which is a direct, connected multi graph with named nodes and edges. However, due to the fact that AUG contains too much details about API calls, so we simplify the AUG and propose our

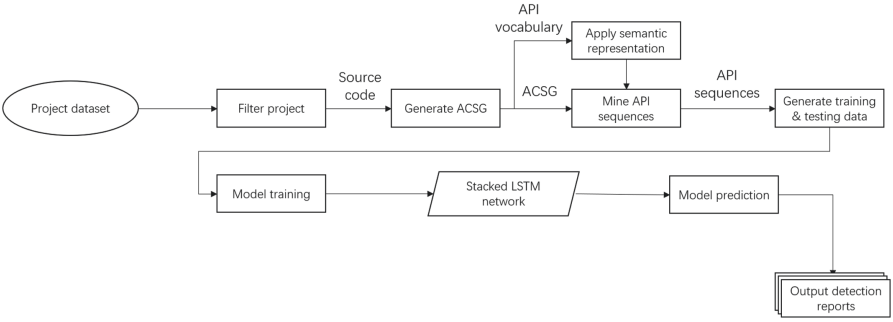

Fig. 1. The overview of our method

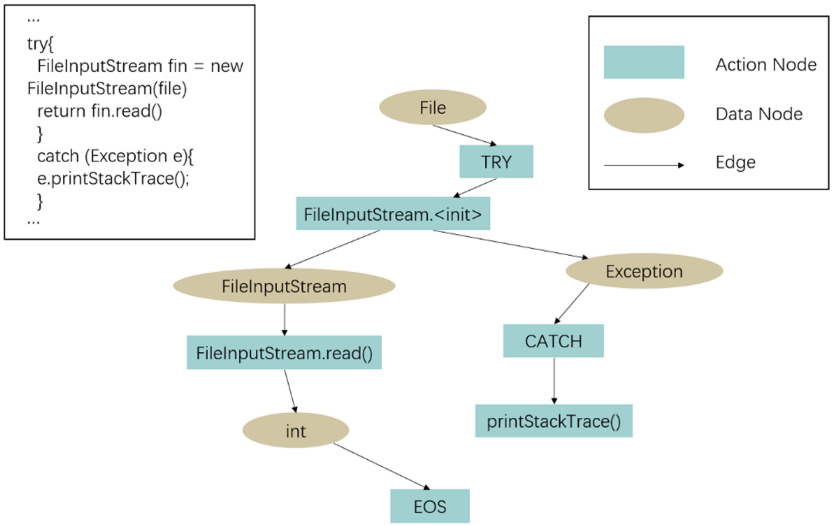

Fig. 2. Example for ACSG

API Call Syntax Graph (ACSG) for further API sequences mining. Figure 2 shows an example of ACSG in this article.

The following definitions are involved in the construction of ACSG:

a. Node: Node represents a node in ACSG, which includes Action Node and Data Node. Action Node contains API calls, method calls and control statements. Data Node presents the objects and values that appears in the source code. The Node with 0 in-degree can be regarded as root node, and the Node with 0 out-degree can be regarded as leave node.
b. Edge: Edge represent the edge which connects the parent node to the child node. In this paper, we use Edge to show the order relation between adjacent nodes.
c. Graph: Graph represents the ACSG. Graph can be composed by subgraphs. And the root of Graph represents the start of ACSG. By adding the edge from the subgraph

1's leave node to subgraph 2's root node, we can finish the compositions between subgraphs.

3.2 Data Generation

API Sequences Mining and Generate Training Data. First, we need to build a vocabulary, which stores all the APIs' information. In our research work, we found and referenced many API mining algorithms [29–31]. Due to the fact that our ACSG is still a graph, we design an API sequences mining algorithm to squeeze the API call sequences among ACSGs. The basic principle of this mining algorithm is to exploits the API sequences which start with in-degree of 0 nodes and end with out-degree of 0 nodes. Table 1 shows our API sequences mining algorithm:

The algorithm follows 2 key ideas:

- Exhaustion-based Mining: This algorithm follows the general idea of an exhaustion algorithm for API call sequences mining. Due to the fact that APIs are called in order from top to bottom, so the API call sequences are supposed to start with one of the initial nodes. Aiming at generating more data, it mines API call sequences by starting nodes with in-degree of 0 and ending nodes with out-degree of 0. The key idea here is that traverse all possible paths that may satisfy the condition mentioned above.
- Generate data recursively: When squeezing API call sequences, the algorithm first converts API call sequences to API call index sequence *s* based on vocabulary. Subsequently, the algorithm reads API call index sequence *s* in order. The training data is divided into 2 parts: one is previous API call sequence, the other is the next API call. The algorithm starts from the first node of *s*. By determining whether the next node exists by reading *s* 's API call, it generates the training data in form <API call index sequence *si*, the next API call $ci + 1$>, and persists to the local training data file. For example, an API sequence is $[API_1, API_2, API_3, API_4]$. And the final training data generated is $[<[API_1], API_2>, <[API_1, API_2], API_3>, <[API_1, API_2, API_3], API_4>]$.

After generating the API call sequences and counting the frequency of API calls in the generated API sequences, a vocabulary with API call indexes corresponding to the API calls is established and persisted to the local vocabulary file.

API Embedding. Word2vec is a kind of word embedding algorithm that provides state-of-the-art outcomes on various linguistic tasks. Due to Distributional Hypothesis [11], which proposes that words that appear in the same context tend to have close relations. As a pre-training model, Word2vec can represent words as d-dimensional vectors, so that words which have close relations with others can have similar vector representation [33, 34]. We treat the API call sequences generated above as sentence in the text, such that adjacent APIs can be semantically embedded into similar vectors. API embeddings is a solution to the problem of numerically representing APIs.

Let $\varphi = \{a_i : i = 1, 2, \ldots, V\}$ be the ordered set of APIs, where V is the size of the API vocabulary and a_i is the i-th API. Usually, the APIs in the API sequence are represented as one-hot vectors,

$$\beta_1 = (1, 0, \ldots, 0), \ldots, \beta_V = (0, 0, \ldots, 1) \tag{1}$$

Table 1. API sequences mining algorithm

```
def API_sequences_mine(acsg: ACSG)
sequences = ∅
sequences = find_all_sequence(acsg)
training_data = generate_training_data(sequences)
return training_data

def find_all_sequence(acsg: ACSG)
sequences = ∅
Start = {nodes are with 0 in-degree in A}
End = {nodes are with 0 out-degree in A}
for sequence_in_acsg in all_sequences_in_acsg: // Ex-
haustion-based sequences mining
  if sequence's start node in Start && sequence's end
node in End:
    sequences = sequences∪sequence_in_acsg
return sequences

def generate_training_data(sequences)
training_data = ∅
for sequence in sequences:   //Generate training data
recursively
  sequence -> <previous API sequence, next API>
  training_data = training_data∪sequence
return training_data
```

where β_i is the hot-vector representation of the API a_i. In this work, we define $A(a_i) \rightarrow \alpha_i$, where α_i is the embedded d-dimensional representation of the API a_i. The way we construct A is through Skip-gram model [16], which could predict a target word given a set of words called context words. We define context APIs as the APIs that appears in the context of target API. The context APIs of an API are defined as the set of APIs are at a distance less than or equal to c from each occurrence of the target API in the API call sequences, where c is a constant which is defined by us. For example, if one wants to let the neural network to show the representational vector of target API a_t, where the context APIs are $\{a_{t-c}, a_{t-c+1}, \ldots, a_{t+c-1}, a_{t+c}\}$. The neural network can be showed as in Fig. 3, where the input layer is the target API a_t, the projection layer can predict the context APIs and finally return the $Context(t)$ in the form if a vector to represent API a_t.

Through word2vec, we can put all the API sequences as input for training, and get a model of semantic representation to the API by analyzing the semantic information of the context APIs. Via this model, we can represent representation of each API that appears in the vocabulary file.

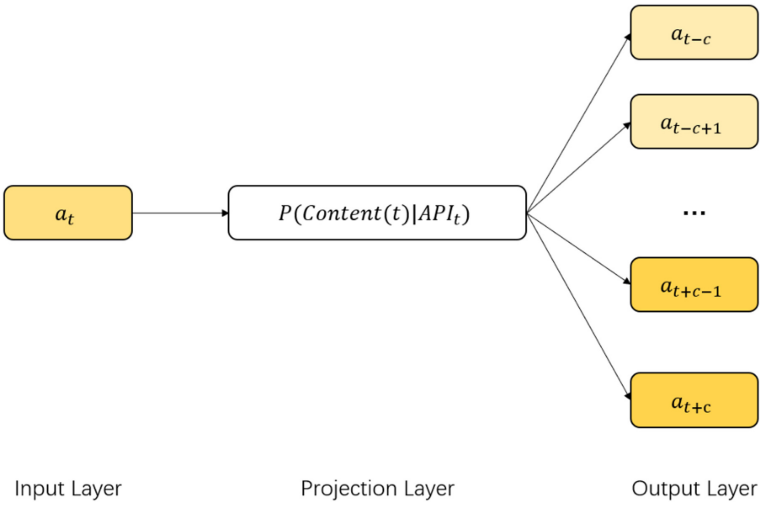

Fig. 3. Word2Vec model

3.3 Model Training and Prediction

During this step, based on TensorFlow framework, we use Python3 to build a deep learning model aiming to learn the training data obtained from the original code fragments. After achieving the model training, we use the model to predict the next API calls located after a certain API call sequence. By compared the predicted API call with target API call, we can judge whether the target API call is suitable or not, so this stage is mainly divided into two parts: model training and model prediction.

Since being regarded as a research hotspot in the field of artificial intelligence, Neural networks have already been used in regression and classification issues. Due to the fact that Recurrent Neural Networks (RNNs) have the structure that can memorize the previous information and apply it to the calculation of the current output, RNNs have the capacity of modeling sequential data. As one of the improved algorithms of RNN, it is worth noting that long short-term memory (LSTM) is designed to solve the long-term dependencies problem for modeling sequential data [16, 17, 35], such as time series, which shows great potential in modeling API call prediction.

Review on LSTM. In addition to the input layer and output layer, the standard deep LSTM model consists of a number of hidden layers which include LSTM layers and fully connected layers. As a basic layer of LSTM architecture, LSTM layers have a group of LSTM cells which can map the output sequences depending on the given input sequence represented by $(x_1, \ldots, x_T)$. Each LSTM cell (shown as Fig. 4) owns its independent weights and biases, which is similar to ANN's neural node. Different from other structures, LSTM cell structure can delete or add information to the cell state through a unit called gate. Every LSTM cell's internal information we call it cell state. There are three gates in LSTM cell, including a forget gate, an input gate and an output gate. The forget gate determine what information needs to be discarded in the cell state. The input gate updates the cell state by the flow of input activation. The output

gate controls the flow of output activation into the next LSTM cell. Here comes the denotation of LSTM operations. In the LSTM network layer, the forget gate shows as f_t, the input gate shows as i_t, the output gate shows as o_t, the cell state shows as c_t, and the hidden layer output shows as h_t. And at the previous time step, we represent the cell memory as c_{t-1}, and the hidden layer output as h_{t-1}. The specific operation in LSTM cell are described as follows:

$$f_t = \sigma\left(W_f \cdot \left[h_{t-1}, x_t\right] + b_f\right) \tag{2}$$

$$i_t = \sigma\left(W_i \cdot \left[h_{t-1}, x_t\right] + b_i\right) \tag{3}$$

$$o_t = \sigma\left(W_o \cdot \left[h_{t-1}, x_t\right] + b_o\right) \tag{4}$$

$$\tilde{C}_t = \tanh(W_C \cdot \left[h_{t-1}, x_t\right] + b_C) \tag{5}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{6}$$

$$h_t = o_t * \tanh(C_t) \tag{7}$$

where W_α (here $\alpha = \{f, i, o, C\}$) represents the weight matrices corresponding to different gates (e.g. forget gate, input gate, output gate or tanh layer as shown in Fig. 4), and b_α denotes the corresponding bias vector. In Fig. 4, $\tilde{C}_t$ represent the candidate information of cell state created by the tanh layer; σ is the logistic sigmoid unit; tanh is the hyperbolic tangent unit.

When it comes to the workflow of this LSTM cell, the input of this structure contains 3 parts: C_{t-1}, h_{t-1}, x_t, while the output includes 2 parts: C_t, h_t(y_t equals to h_t). The relationship between x_t and y_t is that x_t equals to the last layer's y_t. Back propagation of LSTM is utilized to update the weights during the training process. Subsequently, the LSTM last output will be used to predict a specific API call followed by a fully connected layer. Eventually, the next API call can be obtained by the previous API call sequences.

The standard deep LSTM network used for sequential data modeling is composed of plenty pf LSTM layers and fully connected layers (dense layers). The Fully Connected layers are located between LSTM layers and output layers, which are used to connected these two layers. Fully Connected Layers have full connections with previous layer's activation nodes. The embedding layer also can be used in this structure, which can transfer sparse matrix into higher-dimensional dense matrix. In addition, dropout layer which is not shown in Fig. 5, can be added after each layer to prevent overfitting of models. The direct effect of dropout layer is to reduce the number of intermediate features, thereby reducing redundancy, which means increasing the orthogonality between each feature of each layer.

The Full Deep LSTM Network (F-LSTM). The construction of this model is based on the deep learning framework TensorFlow 1.15.0 as the implementation framework. It is noted that, to implement this LSTM architecture, the main calculation of this model

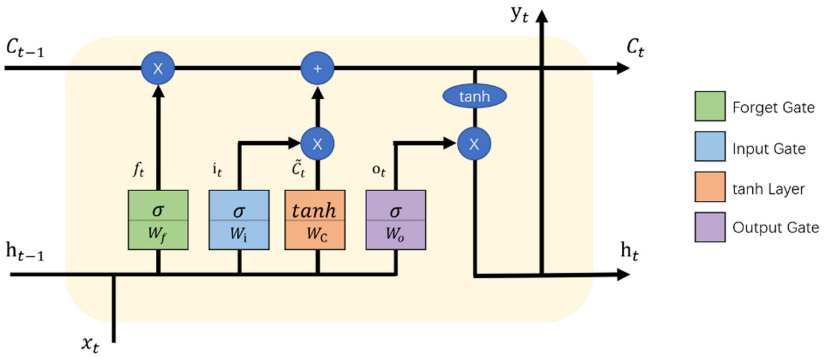

Fig. 4. LSTM cell

is shown as follows: (1) Through the embedding layer, we use the word2vec model to generate the corresponding vector to represent the API's semantic information. (2) The input vector converted by the embedding layer passes through the dropout layer, through which it discards part of the information, aiming to increase the model's robustness. (3) Send the processed data to the LSTM processing unit provided by TensorFlow for training. (4) The output of the model needs to go through the dense layer, which can convert the output into desired format, and we define the output of dense layer as logits. Subsequently, logits undergo softmax processing and cross-entropy calculation to obtain the probability distribution and loss value of API calls respectively. Besides, TensorFlow will optimize the parameters based on the given optimization function.

In this content of API prediction modeling, let us denote the input front API sequences as $x = \{x_1, x_2, \ldots, x_n\}^T \in R^{n \times p}$ and the output API as y, where p represent the number of input features and y is represented as one-hot vector. x is vector whose matrices whose rows stand for time steps and columns for features (the dimension of API). The full deep LSTM network takes the target API's preceding API sequence (x) as input and map it to the output of the model, target predicted API (y), where each cell within a LSTM layer is connected with its two neighbors via $\{t-1, t, t+1\}$. At each time step, all input features at the current time step are fed into the deep LSTM network. The process keeps sending input features to the network via repeated LSTM cells across the entire temporal space, which could build a chain-like structure to keep the long-short term time dependencies. Figure 5 illustrates the basic structure of F-LSTM which aims to model the target API (y) given its preceding API sequence (x).

In conclusion, the F-LSTM network models the corresponding input & output relation along the temporal space. But a main disadvantage of F-LSTM is that it has a massive demand for training efforts especially for long API sequences that require a large amount of computation memory. To solve this problem, we modify and propose the Stacked Deep LSTM network.

The Stacked Deep LSTM Network (S-LSTM). We present the S-LSTM model which takes the stacked input of a certain number of time steps, $\{X_1, X_2, \ldots, X_t\}^T$, to predict the output y. Assuming that we have only one feature, we can illustrate the concept of S-LSTM as Fig. 6. Via embedding layer, the original sequences of input X is

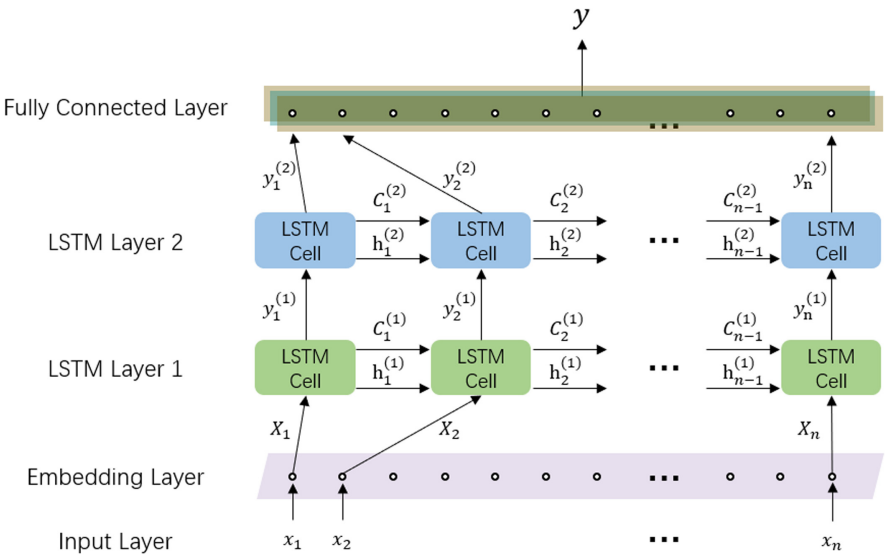

Fig. 5. The structure of full deep LSTM

divided into multiple stacks, which forms the new input $\tilde{X} = \left\{\tilde{X}_1, \tilde{X}_2, \ldots, \tilde{X}_s\right\}$, where s is the number of the stacks. Therefore, each stack we created can be considered as one new time step and reduce the temporal space the model costs. Note that $sw \leq n$ (if $sw < n$, we apply zero-padding to meet the disparity). Every input stack consists of a fixed length of original API sequence, which is considered as the new input features to fed the LSTM cells.

S-LSTM not only reduces the temporal dimension but also represent the semantic meaning of each API in the previous API sequence. It is worth noting that, w is an empirical number and depends on the specific embedding result as well as the dominant mode of the dynamic model response. In addition, S-LSTM is observed to provide better training and prediction performance as compared to LSTM-f and other models as showed in Sect. 4.

3.4 Model Prediction

In this stage, on the basis of the model trained in deep learning model training, we use the previous sequence of API calls for prediction as input, predict the probability distribution of the API call at the next position, and sort the probability distribution to obtain the API call probability list at the next position. This API call probability list will be used as an important part of defect detection.

4 Experiment

This section presents the implementation of applying LSTM model to predict the target API. In our work, the experimental evaluation is divided into 2 parts: (1) The model

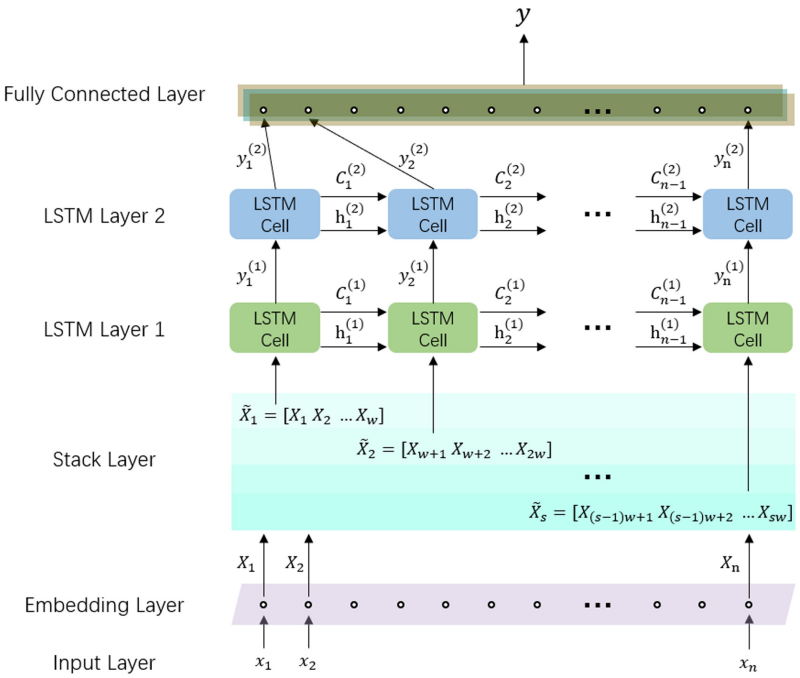

Fig. 6. The structure of stacked deep LSTM

training experiment. The purpose of this part is to optimize the accuracy and reliability of the prediction of the API call sequence through model. (2) The code defect detection experiment. The purpose of this part is to detect the model's effectiveness and usability in API misuse defect detection.

4.1 Experimental Object

Java cryptography extension (JCE) is a package provided by JDK, which [9] can provide the implementation of cryptographic primitive, including block ciphers and message authenticate codes (MACs). Java cryptography APIs provided by JCE are under the package named javax.crypto. According to related research [9], by separating the implementation details for the users, developer can use it conveniently to achieve the encryption and decryption functions. Also, these APIs provide multiple modes and configuration setting options. However, to use and combine these APIs correctly can be challenges for developers. In the following parts, we will focus on the experimental object of Java cryptography APIs to illustrate the ability of our models in defect detection.

4.2 Model Training Experiment

The Dataset of this article was obtained from Git, a free and open source distributed version control system, through which we could manage no matter what kinds of projects.

GitHub is a large-scale open source code hosting platform and version control system based on Git. The open source code on GitHub is a massive data source [20, 21] for mining projects which contain Java cryptography APIs. Then, we generate our original data by using "javax.crypto" as the key word from the project we search via GitHub.

In order to achieve comparative experiments, we use the parameter configuration proposed in Wang's work [8]. In our work, we compared the training effects of three different models, which include Bugram (3-gram and 4-gram are selected), the deep learning LSTM model proposed by Wang's work [8] (we will call it D-LSTM later), F-LSTM and S-LSTM. The essential difference between our models and D-LSTM is whether use word2vec in the padding layer or not. The specific parameter configuration is HIDDEN_SIZE = 250, NUM_LAYER = 2, LR = 0.002, NUM_EPOCH = 20. As shown in Fig. 7, we show the loss of 3 different models, where the horizontal axis is the number of epochs, and the vertical axis is the loss value.

In this configuration, the model's effect is shown in Fig. 8. The horizontal axis is the number of epochs, and the vertical axis is the accuracy.

Obviously, we can tell that F-LSTM and S-LSTM perform well in both accuracy and loss value. We can intuitively see from the Fig. that the adding Word2Vec has a great influence on the training of the model. Although model training capabilities shown by the F-LSTM and S-LSTM are very similar, S-LSTM still has about 1% better in accuracy and 0.05 lower in loss value than F-LSTM. Finally, the D-LSTM achieve 80.3% accuracy and 0.772 loss value; the F-LSTM achieve 83.2% accuracy and 0. 622 loss value; the S-LSTM achieve 84.2% accuracy and 0.567 loss value.

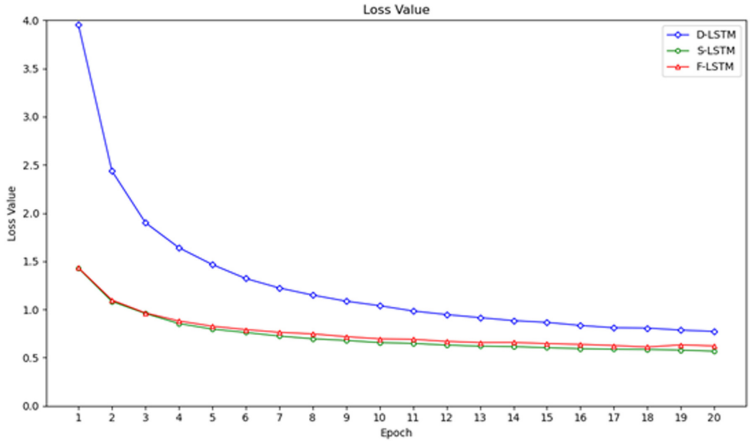

Fig. 7. Loss value of models

4.3 Code Defect Detection Experiment

Based on the models we have proposed above, the code defect detection experiment is designed to show the ability of code defect detection on specific projects. First, we define the evaluation standard for this experiment as follows:

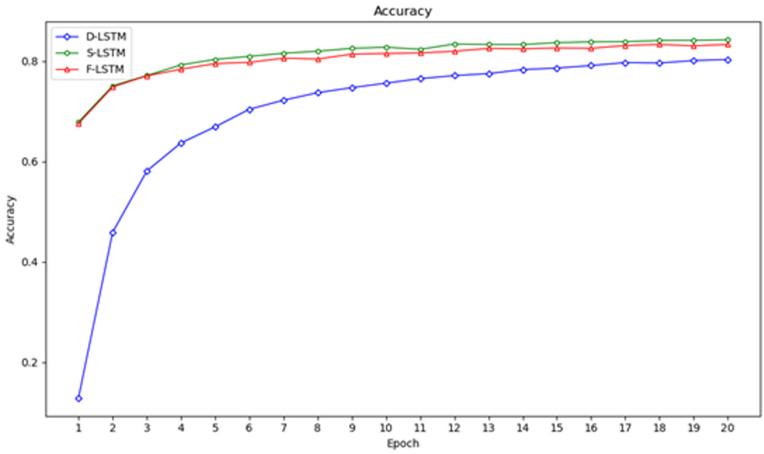

Fig. 8. Accuracy of models

1. We define the precision as follow:

$$Precision = \frac{TP}{TP + FP} \tag{8}$$

2. We define the recall as follow:

$$Recall = \frac{TP}{TP + FN} \tag{9}$$

3. We define the F1 score as follow:

$$F1 = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{10}$$

where we define TP as the number of reports that the target API without API misuse can be correctly predicted in the target API call probability list; FP as the number of reports that the target API with API misuse can be correctly predicted in the target API call probability list; FN as the number of reports that the target API with API misuse cannot be correctly predicted in the target API call probability list. Due to the fact that the score of precision and recall cannot reflect the result of experiment comprehensively, we decide to use F1 score which is the harmonic mean of precision and recall, as the final standard for the evaluation of our experimental results.

In our experiment, according to the actual API misuse of Java Cryptography API in our survey results, we select 8 API misuse codes which are collected from high-quality

projects from open code platform, such as GitHub and SourceForge. And we use these codes as the test set for our models' evaluation. The code defect detection experimental results are shown as Figs. 9, 10, 11. The lines in these three figures represent different experimental models respectively, and the x-axis represents the value of the acceptable threshold (top-k). Figure 9 shows the F1 score value of each model; Fig. 10 shows the precision value of each model; Fig. 11 shows the recall rate value of each model.

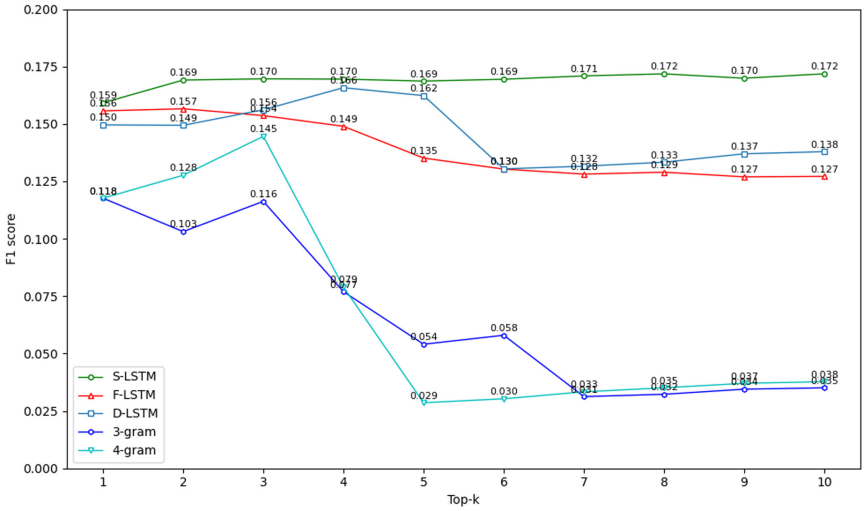

Fig. 9. The performance of F1 score

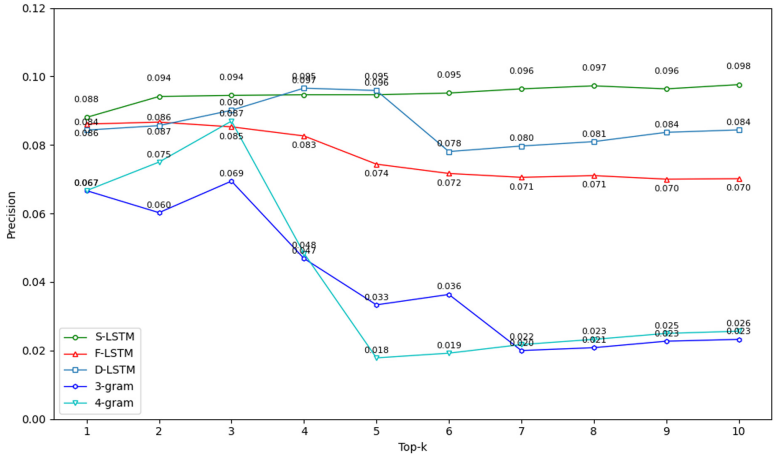

Fig. 10. The performance of precision

It can be seen from the results that S-LSTM' precision is always higher than F-LSTM's. The recall rates of both models are very low when they are at top-1, but as the

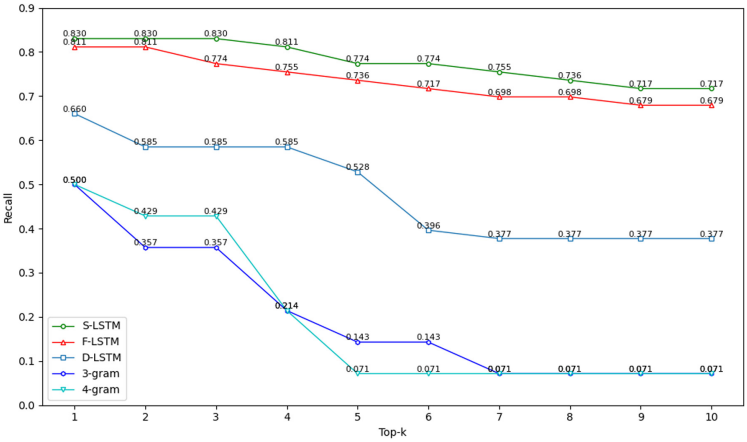

Fig. 11. The performance of recall rate

top-k increases, their recall rates also increase, and overall S-LSTM' precision is always higher than F-LSTM's. Since the model's recall rate is greater than 50, the model has defect detection capabilities. When the top-k is taken to be top-10, S-LSTM's defect detection works best.

5 Conclusion and Future Work

This paper proposes a deep learning approach, based on a long short-term memory (LSTM) recurrent neural network, for API usage specification learning and API misuse defect detection. We design a static analysis method called API Call Syntax Graph (ACSG) for presenting API usage. Different from existing models, our proposed architecture adds an embedding layer which applies the pre-training model Word2Vec to achieve the semantic representation of each API in the sequence. The two models are developed, namely, F-LSTM and S-LSTM with different input formats. The capabilities of these two models are illustrated via the two experiments mentioned above. Finally, we compare the performance of the models we proposed, the experimental results turn out that S-LSTM has a better performance, with the F1 score 0.172 where the threshold is among top-10.

In the future work, we plan to verify our method in various types of APIs in order to demonstrate a higher applicability of our approach. Due to the "naturalness" of the source code, models in NLP area with great performances can be applied into either semantic representation or API use specification learning process.

Acknowledgement. This work has been supported by the Foundation item: National Key R&D Program of China (2018YFB1402800); National Natural Science Foundation of China (61772560).

References

1. Allamanis, M., et al.: A survey of machine learning for big code and naturalness. ACM Comput. Surv. (CSUR) **51**(4), 1–37 (2018). Author, F., Author, S.: Title of a proceedings paper. In: Editor, F., Editor, S. (eds.) CONFERENCE 2016, LNCS, vol. 9999, pp. 1–13. Springer, Heidelberg (2016)
2. Nguyen, S., et al.: Combining program analysis and statistical language model for code statement completion. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE (2019)
3. Allamanis, M., et al.: Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (2014)
4. White, C., Vendome, M., Linares-Vásquez, M., Poshyvanyk, D.: Toward deep learning software repositories. In: MSR 2015, pp. 334–345 (2015)
5. Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., Devanbu, P.: On the naturalness of buggy code. In: Proceedings of the International Conference on Software Engineering (ICSE) (2016)
6. Oh, H., Yang, H., Yi, K.: Learning a strategy for adapting a program analysis via Bayesian optimisation. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) (2015)
7. Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M., Guibas, L.J.: Learning program embeddings to propagate feedback on student code. In: Proceedings of the International Conference on Machine Learning (ICML) (2015)
8. Pradel, M., Sen, K.: Deep learning to find bugs (2017)
9. Proksch, S., Lerch, J., Mezini, M.: Intelligent code completion with Bayesian networks. ACM Trans. Softw. Eng. Methodol. (TOSEM) **25**, 1–31 (2015)
10. Rabinovich, M., Stern, M., Klein, D.: Abstract syntax networks for code generation and semantic parsing. In: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) (2017)
11. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from "big code". In: Proceedings of the Symposium on Principles of Programming Languages (POPL) (2015)
12. Wang, S., Chollak, D., Movshovitz-Attias, D., Tan, L.: Bugram: bug detection with n-gram language models. In: Proceedings of the International Conference on Automated Software Engineering (ASE) (2016a)
13. Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R.: sk_p: a neural program corrector for MOOCs. In: Proceedings of the Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH) (2016)
14. Xue, Y., Wang, J., Liu, Y., Xiao, H., Sun, J., Chandramohan, M.: Detection and classification of malicious javascript via attack behavior modelling. In: ISSTA 2015, pp. 48–59 (2015)
15. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: Proceedings of the Symposium on Programming Language Design and Implementation (PLDI) (2014)
16. Xie, K., Wen, Y.: LSTM-MA: a LSTM method with multi-modality and adjacency constraint for brain image segmentation. In: 2019 IEEE International Conference on Image Processing (ICIP). IEEE (2019)
17. Xue, H., Huynh, D.Q., Reynolds, M.: SS-LSTM: a hierarchical LSTM model for pedestrian trajectory prediction. In: 2018 IEEE Winter Conference on Applications of Computer Vision (WACV). IEEE (2018)
18. Tu, Z., Su, Z., Devanbu, P.: On the localness of software. In: Proceedings of the International Symposium on Foundations of Software Engineering (FSE) (2014)

19. Kersten, M., Murphy, G.C.: Using task context to improve programmer productivity. In: FSE, pp. 1–11. ACM (2006)
20. Kuhn, A., Ducasse, S., Gírba, T.: Semantic clustering: identifying topics in source code. Inf. Softw. Technol. **49**(3), 230–243 (2007)
21. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Suggesting accurate method and class names. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE) (2015)
22. Allamanis, M., Tarlow, D., Gordon, A., Wei, Y.: Bimodal modelling of source code and natural language. In: Proceedings of the International Conference on Machine Learning (ICML) (2015)
23. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: Proceedings of the International Conference on Machine Learning (ICML) (2016)
24. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: Proceedings of the International Conference on Learning Representations (ICLR) (2018)
25. Wang, X., Chen, C., Zhao, Y.F., Peng, X., Zhao, W.Y.: API misuse bug detection based on deep learning. Ruan Jian Xue Bao/J. Softw. **30**(5), 1342–1358 (2019). (in Chinese). http://www.jos.org.cn/1000-9825/5722.htm
26. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: Proceedings of the International Conference on Software Engineering (ICSE) (2012)
27. Nguyen, A.T., Nguyen, T.N.: Graph-based statistical language model for code. In: Proceedings of the International Conference on Software Engineering (ICSE) (2015)
28. Nguyen, T.T., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: A statistical semantic language model for source code. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE) (2013)
29. Fowkes, J., Sutton, C.: Parameter-free probabilistic API mining across GitHub. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (2016)
30. Nguyen, P.T., et al.: FOCUS: a recommender system for mining API function calls and usage patterns. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE (2019)
31. Chen, C., et al.: Mining likely analogical apis across third-party libraries via large-scale unsupervised API semantics embedding. IEEE Trans. Softw. Eng. (2019)
32. Sven, A., Nguyen, H.A., Nadi, S., et al.: Investigating next steps in static API-misuse detection. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 265–275. IEEE (2019)
33. Lilleberg, J., Zhu, Y., Zhang, Y.: Support vector machines and word2vec for text classification with semantic features. In: 2015 IEEE 14th International Conference on Cognitive Informatics & Cognitive Computing (ICCI* CC). IEEE (2015)
34. Zhang, D., et al.: Chinese comments sentiment classification based on word2vec and SVMperf. Expert Syst. Appl. **42**(4), 1857–1863 (2015)
35. Zhang, R., et al.: Deep long short-term memory networks for nonlinear structural seismic response prediction. Comput. Struct. **220**, 55–68 (2019)