API Misuse Detection Method Based on Transformer

Jingbo Yang^{1,*}, Jian Ren^{1,*}, and Wenjun Wu^{2,*}
^{1}School of Computer Science and Engineering Beihang University, Beijing, China
^{2}Institute of Artificial Intelligence Beihang University, Beijing, China
872206096@qq.com, renjian@buaa.edu.cn, wwj09315@buaa.edu.cn
*corresponding author

***Abstract*—Software developers need to take advantage of a variety of APIs (application programming interface) in their programs to implement specific functions. The problem of API misuses often arises when developers have incorrect understandings about the new APIs without carefully reading API documents. In order to avoid software defects caused by API misuse, researchers have explored multiple methods, including using AI(artificial intelligence) technology.**

As a kind of neural network in AI, Transformer has a good sequence processing ability, and the self attention mechanism used by Transformer can better catch the relation in a sequence or between different sequences. Besides it has a good model interpretability. From the perspective of combining API misuse detection with AI , this paper implements a standard Transformer model and a target-combination Transformer model to the learning of API usage information in a named API call sequence extracted from API usage program code. Then we present in the paper the way that our models use API usage information to detect if an API is misused in code. We use F1, precision and recall to evaluate the detection ability and show the advantages of our models in these three indexes. Besides, our models based on Transformer both have a better convergence. Finally, this paper explains why the models based on Transformer has a better performance by showing attention weight among different elements in code.

Keywords*—*API misuse, AI, Transformer, model interpretability

I. INTRODUCTION

When developers adopt application programming interface (API), they can reuse or expand the existing software frameworks and class library functions, so as to effectively improve the efficiency of software development. For example, a Java project relies on an average of 14 different libraries[1], and the Maven library has indexed 8.77 million third-party libraries [2]. Due to the complexity of the API itself [3][4][5][6] and potential inconsistency between implementations of APIs and their documents [7][8][9], developers may have misunderstandings about these APIs and mistakenly use the API functions in their code. There are various situations of API misuses [10], such as mismatched API call parameters, absence of exception handling and lack of condition check. These API misuses can introduce defects such as potential functional and safety hazards as well as performance problems in implementation of software systems[11][12][13][14][15][16][17]. For example, in the code of Android, if an API call makes reference to the null object without checking the validity of the object, the program often causes the problem of a null pointer exception. Therefore, detecting API misuse plays an increasingly important role in ensuring the quality of software, and a variety of static analysis tools have been proposed to address the problem [18][19][20][21][22][23].

To avoid API misuse, the existing methods can be grouped into two kinds: The first kind method is designed to extract API documentation information, which mainly uses Natural Language Processing(NLP) to extract specifications from an API documentation and detect API misuses in code. The second kind of methods is designed to mine API usage information from code and extract latent information to detect API misuses. Such possible information can be represented by machine learning(ML) models [24].

By focusing on the second kind method, in this paper, we implement two models based on a famous AI model-Transformer[25][26] to detect API misuse, one is a standard Transformer model and the other is a target-combination Transformer model. We utilize the method proposed by[24] to generate API usage data and train our Transformer models to represent the latent dependencies among different code elements. The models can run as a sequence predictor to find API misuses. The experimental results confirm that the models developed in this paper outperform other baseline models.

The rest of this paper is organized as follows: Section 2 describes the state-of-the-art API misuse detection algorithms and elaborates the research motivation of the paper. Section 3 introduces the API misuse detection method based on Transformer and explains the design details. Section 4 describes experiment designs for comparing API misuse detection models and presents experimental results. Section 5 summarizes this paper and gives the conclusion.

II. RELATED WORK

A. API misuse detection method

a) Extract API documentation information: The doc2seq method proposed by Zhong[27] utilizes NLP technology to analyze the natural language in API documents, and deduces the specification about API usage from their documents. The authors extracted API usage specification from five library documents and evaluated the performance of doc2seq. The experimental results show that this method can infer different specification with high accuracy, and find the defects that have not been discovered in open source projects. It is obvious that the accuracy of such methods solely depends upon the quality of the extracted specification. Without correct descriptions of the major relations among logic entities of APIs, such methods can not deliver high quality detection results.

The study [28] developed open information extraction methods to construct API-constraint knowledge graph from API reference documentation. The knowledge graph empowers the detection of three types of frequent API misuses-missing calls, missing condition checking and missing exception handling, while existing detectors mostly focus on only missing calls. The evaluation confirms their method has a better detection result.

Unfortunately, many of such constraints are insufficiently specified by APIs' documentations. As a result, developers also refer to informal references, such as Stack Overflow, to understand the usages of an API[29] .

b) Mine API usage information from code: The other way of API misuse detection commonly attempts to capture frequent API usage patterns and formalize the problem as an anomaly detection. Many methods have been proposed from different data mining and machine learning frameworks, where they choose different ways to represent API usages and their frequency.

Li et al. studied the API usage specification mining technology based on frequent itemsets[30], and developed a mining and detection tool PR miner. The main functions of PR miner include automatically extracting implicit API usage constraints and detecting violations of these constraints. Its high-level idea is to find some frequently used elements from the source code, including functions, variables and data types, and to identify the correlation among these elements. From the source code repository of Linux and PostgreSQL, PR miner can extract 32000 programming rules in 1-42 seconds and confirm 23 reported defects. However, such specification of an API may not contain all the necessary usage constraints.

Wen[31] believes that the usage of API mutation is correct based on the usage analysis of API mutation ; Secondly, the usage of these changes can verify whether it is API misuse by executing test cases and analyzing execution information. From the verification results, learn how the API is misused, and use the misuse mode to detect API misuse defects. The author conducted experiments on 73 popular Java APIs in 16 projects. This method found that the accuracy rate of API misuse was as high as 0.78, and the recall rate was 0.49 on mubench data set. However, if we use an mutation API which may pass through the tests but not recommended by the API provider, that may cause a hidden danger.

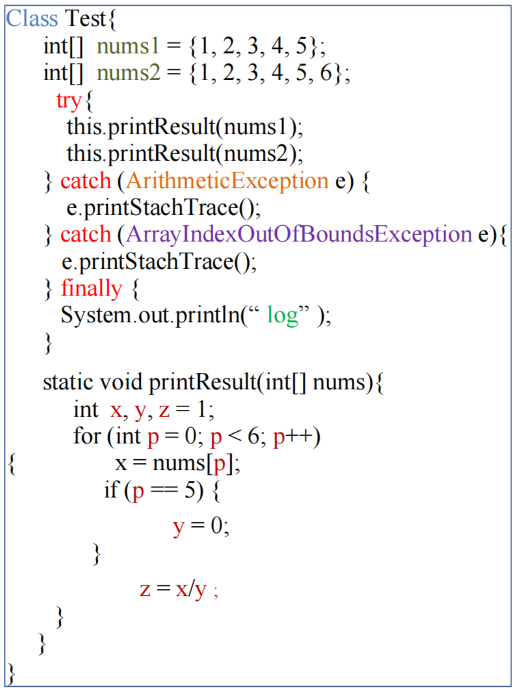
By leveraging a N-gram language model, Wang et al. introduced a defect detection tool bugram[32]. The authors assume that a program element is only relevant with its previous n-1 elements in a API call sequence. Therefore, bugram calculates the probability of all the sequences in a dataset and classifies the sequence with the lowest probability as a suspected defect. The authors detected 59 bugs in 16 new versions of Java projects. The main limitation of bugram is rooted in the N-gram language model, which can only capture the relevance between the current token and its n-1 predecessors. The actual API usage often involves more complex dependence patterns than this simple one.

Ouyang et al. [24] proposed a Stacked LSTM method to detect API misuse in code. The authors utilize JavaParser to convert code into a graph structure, then generate API call sequences from the graph as training data samples. Although The LSTM models employed in their research can represent the adjacent dependence among consequent API tokens, they are not sufficient for capturing long-stride associations of the tokens, which reflect both structural and semantic information in Java code.

We can see from these studies above that it is very useful to apply AI technologies, like 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.

B. Research motivation

When using one API, there may be different reasons for misuse. Any different element in code may cause API misuse differently. That is, when we use an API, we should pay attention to different weights of element to avoid misuse. Here we show two examples for this explanation.



```
Class Test{
    int[]  nums1 = {1, 2, 3, 4, 5};
    int[]  nums2 = {1, 2, 3, 4, 5, 6};
     try{
       this.printResult(nums1);
       this.printResult(nums2);
    } catch (ArithmeticException e) {
       e.printStachTrace();
    } catch (ArrayIndexOutOfBoundsException e){
      e.printStachTrace();
    } finally {
      System.out.println(" log" );
    }
    static void printResult(int[] nums){
        int  x, y, z = 1;
        for (int p = 0; p < 6; p++)
{             x = nums[p];
            if (p == 5) {
                    y = 0;
            }
                z = x/y ;
      }
    }
}
```

Figure 1. Example of try-catch code block.

In Figure 1, we can see that there are two different exceptions in the code. When array $nums1$ is executed, an out of bounds exception occurs while array $nums2$ is executed, an arithmetic calculation exception occurs. We can conclude that when the code executes $printResult()$, different params $nums1$ and $nums2$ have different weight contribution to API misuse and may cause different type of API misuse.

In Figure 2, there are multiple APIs in this line of code, including $getWidgetByName()$, $substring()$ and $setText()$. Resulting in the misuse of $setText()$ should be

$getWidgetByName()$ instead of $substring()$. If The API $getWidgetByName()$ can not get the object by its param or the object doesn't exist, the $setText()$ API happens error. Thus the element $getWidgetByName()$ has more contribution to lead $setText()$ misuse than $substring()$. That is to say different element at different position in the code may have different contributions to API misuse.

```
public void setWidgetName(){
  getWidgetByName("Text".substring(0, 10)).setText("hello")
;}
```

Figure 2. Example of Widget code.

Based on above analysis, we can see that any different element in code may cause API misuse and may cause different type of API misuse, so we should take care of different weight to different element in detecting API misuse, but as far as we learn, no researchers analysis and do this before from this point of view. Besides, as program code can be converted into execution sequences[32], and neural network can extract sequence characteristics and mine the relationships among sequences. We decide to use neural network to learn the relationship among different elements in code. Transformer, as a famous kind of neural network, can do well in mining the relationship among different elements in sequence. So in this paper, we propose two AI models based on Transformer to better learn API usage by catching the relation among different elements in using API code. Then we use the models to detect API misuse in test dataset to better find API misuse bugs.

III. Research Method

Our Transformer-based method is mainly divided into the following steps:

- Data generation. In this paper, we use API Call Syntax Graph (ACSG) to represent the code structure, then we use the exhaustion-based mining algorithm to generate API call sequences of ACSG[24]. According to the algorithm, the origin code data is transformed into API call sequence, and then generated to our dataset.
- Model training and prediction. This paper designs a standard Transformer model and a target-combination Transformer model to predict the API call sequence and detect whether the API is misused.

The flow chart of this method is shown in Figure 3.

A. Data generation

To better understand, the process of generating ACSG and API call sequence involves the following definitions:

a) Node: There are two kinds of nodes including action node and data node. An action node includes API calls, method calls, control statements and method in class. A data node includes object, values such as parameters including API call parameter appearing in the code. In Figure 4, the upper right rectangle including $Action$ and $Data$ are node examples for the upper left code.

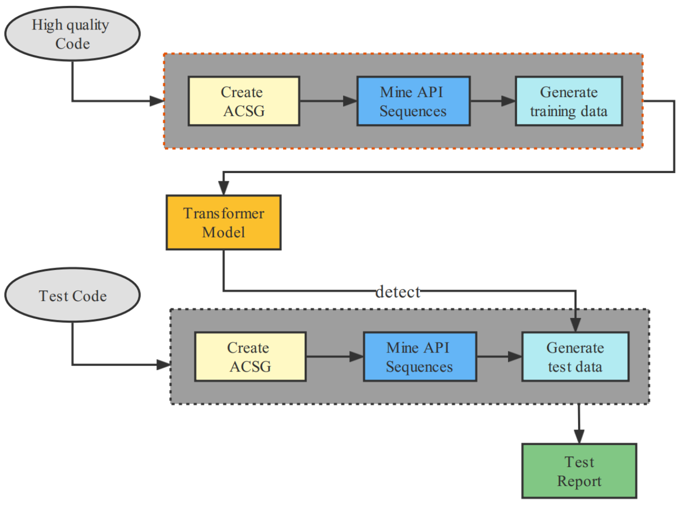


Figure 3. the workflow of based on Transformer method.

b) Edge: Nodes are connected by directed edges, pointing from one node to another, which represents the execution order in a program. The black arrow lines in Figure 4 are examples of edge.

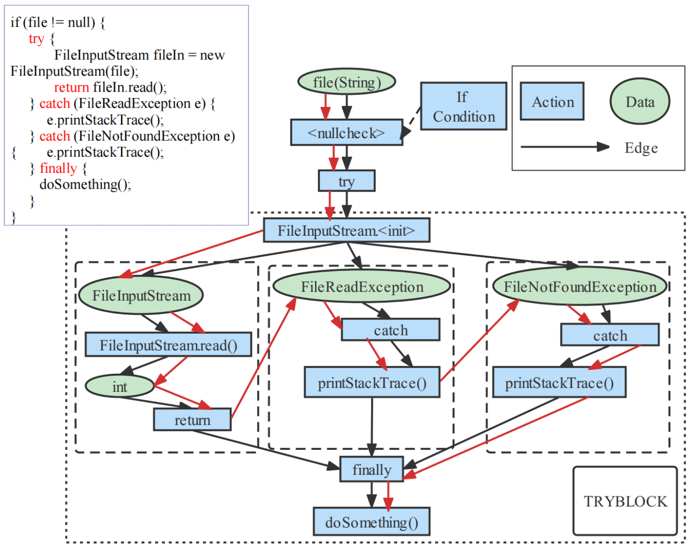
c) API call syntax graph (ACSG): An ACSG is composed of nodes and edges. ACSG starts with the root node and ends with the leaf node. There can be other nodes and edges between the root node and the leaf nodes. Multiple ACSGs can be combined into a single ACSG by adding the edge from one ACSG's leave node to another ACSG's root node. In Figure 4, all the black arrow lines and all nodes construct an ACSG example for the upper left code.

d) API call sequence: We have a special explanation, a node in an API call sequence can be any type of node defined before, such as a concrete API or a method or a parameter and so on. We use API_n to represent a node uniformly. For example, a sequence like $[getInstance(), param, While]$ can be represented as $[API_1, API_2, API_3]$. The order from left to right in an API call sequence represents an execution flow of code. The red arrow lines and all nodes in Figure 4 show an ACSG converts to an API call sequence.

Figure 4 shows an example of ACSG and one API call sequence representing an execution branch of source code. ACSG can well represent the code structure. In this way, we construct an ACSG dataset from all the usage code in a method for further training.

In the process of constructing ACSG, we run JavaParser to effectively parse the source code[33] and obtain the abstract syntax tree (AST) representing the source code structure. From the AST representation, we can further construct an ACSG from any java code segment.

With the ACSG of a program segment, we need to convert the ACSG into an API call sequence and then use the sequence to generate training data for our Transformer models. We use the API sequence mining algorithm proposed by [24] to get API call sequences. An API call sequence consists of



```
if (file != null) {
    try {
          FileInputStream fileIn = new
FileInputStream(file);
          return fileIn.read();
    } catch (FileReadException e) {
         e.printStackTrace();
    } catch (FileNotFoundException e)
{        e.printStackTrace();
    } finally {
       doSomething();
    }
}
```

Figure 4. An example of ACSG and API call sequence. The upper left rectangle shows code example while upper right rectangle shows element representation of ACSG, containing Data Node, Action Node and Edge. All kinds of nodes and all the black edges construct the ACSG while all kinds of nodes and all the red edges can represent one API call sequence. One ACSG may be extracted more than one API call sequences as control node can yield different execution branches. ACSG can well stand for the code structure while API call sequences can well represent the execution flow of source code

multiple nodes in the form of $[API_1, API_2, API_3, ...API_n]$. The position of the n-th API node can be considered to be predicted by the previous (n-1) API calls. So, the training data can be divided into 2 parts from an API call sequence. The first part is$[API_1, API_2, ..., API_{n-1}]$, while the second part is $[API_n]$. We put the two parts in the form of $([API_1, API_2, ..., API_{n-1}], API_n)$. For example, if an API call sequence is $[API_1, API_2, API_3]$, the final training data can be generated as $([API_1]$,API$_2$),($[API_1, API_2]$, API$_3$).

B. Model Design and Implementation

Based on the pytorch framework, we implement two Transformer models, a standard Transformer model and a target-combination Transformer model to learn the sequential patterns of API usages from the training dataset and run the models to detect whether an API is misused in test code. The details of implementation are described as follows:

a) Standard Transformer model: The architecture of the standard Transformer model is shown as Figure 5.

The model structure is mainly composed of encoder and decoder. The Encoder is composed of one or multiple layers with the same structure, which is illustrated on the left panel in Figure 5. Each layer consists of two sub_layers with a Multi-Head-Attention mechanism and fully connected feed forward network. Moreover, each sub_layer also contains a residual connection and normalization. The output of each sub_layer can be expressed as:

$$sub\_layer\_output = LayerNorm(x + Network(x)) \quad (1)$$

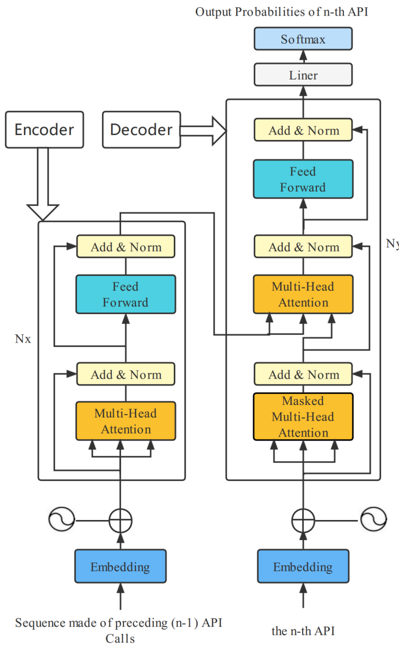


Figure 5. Our Standard Transformer model for Representing sequential patterns of API usages.

So the first sub_layer_output is:

$$LayerNorm(Z + MultiHeadAttention(Z))$$

and the second sub_layer_output is:

$$LayerNorm(Z + FeedForward(Z))$$

Where Z represents the input of the MultiHeadAttention or FeedForward, and Add refers to the residual connection.

The FeedForward layer is a two-level fully connected MLP. The first layer uses Relu function while the second layer does not use activation function. In this model design, it is assumed that the input of Multi-Head-Attention layer is $Q \in R^{N \times T_q \times d_{model}}, K \in R^{N \times T_k \times d_{model}}, V \in R^{N \times T_k \times d_{model}}$. Where N is batch_size, T_q, T_k are the max length of Q and K respectively. d_{model} a is the vector length of the original API embedding.

Assuming that h is the number of heads of Multi-Head-Attention, then h times linear Transformation is carried out, and the i-th linear transformation is:

$$M_i = \begin{cases} Q_i = Q \times W_i^Q, & W_i^Q \in R^{(d_{model}, d_k)} \\ K_i = K \times W_i^K, & W_i^K \in R^{(d_{model}, d_k)} \\ V_i = V \times W_i^V, & W_i^V \in R^{(d_{model}, d_v)} \end{cases} \quad (2)$$

Next, we do as follows:

$$temp = \frac{Q_i(K_i)^T}{\sqrt{d_k}} \quad (3)$$

$$head_i = attention(Q_i, K_i, V_i) = softmax(temp)V \quad (4)$$

$$MultiHead(Q,K,V) = Concat(head_1, head_2 ..., head_h)W^o \tag{5}$$

Among them $W^o \in R^{(N,hd_v,d_{model})}$, through the Multi-Head-Attention operation, the final matrix has the same dimension as the matrix Q of inputs.

In the model, the structure of decoder and encoder is similar, but there is a sub layer of attention. Besides the API sequence also adopts positional embedding (PE):

The dimension of PE is the same as that of API embedding. The coding equation of PE is as follows:

$$M_1 = \begin{cases} PE_{(pos,2i)} = sin(pos/10000^{2i/d}) \\ PE_{(pos,2i+1)} = cos(pos/10000^{2i/d}) \end{cases} \tag{6}$$

Where, pos represents the position of the word in the sentence, d represents the dimension of PE, $2i$ represents the dimension of even number, $(2i+1)$ represents the dimension of odd number, and the addition of API embedding and position embedding represents the representation vector of an API.

We choose positional embedding to encode the sequences of API call tokens in order to pass the semantic information of the API calls into the Transformer model. Since this model is defined based on encoder-decoder architecture, we take the preceding tokens in an API call sequence as the input of the encoder component and the left token as the input of the decoder component.

The encoded input sequence $([API_1, API_2, ..., API_{n-1}])$ is passed through the Multi-Head-Attention layer where it calculates Multi-Head-Attention as described in Eq(2), in order to represent the dependent weight relationship between API calls in the sequence. Then, the AddNorm layer adds Multi-Head-Attn $([API_1, API_2, ..., API_{n-1})$ and the input embedding $([API_1, API_2, ..., API_{n-1})$ together, and performs a Normalization operation on the adding result to accelerate the convergence speed during the train phase. The normalization result is passed through the feed-forward neural network and Add-Norm layer to deliver the encoder output, which is denoted as encoder-output $([API_1, API_2, ..., API_{n-1}])$.

At the side of the decoder, the last token of the API call sequence is the embedded and used as the input for the decoder's first stage, which is similar to the encoder. we denote the result of this stage as output$[API_n]$. The next step is to associate the preceding tokens with the last token of the API call sequence. The encoder-output $([API_1, API_2, ..., API_{n-1}])$ is used as the K and V of the decoder, and the output$[API_n]$ as Q. The Multi-Head-Attention operation is performed to represent the weight relationship between the encoder output corresponding to the Q . At the last step, the softmax layer defined in Figure 5 is responsible for calculating the probability of the occurrence of the last token given the preceding sequence. This output probability can determine whether the API node is a potential misuse or not. This is the first stage of AI model and can better find API call order or redundant call misuse in sequence. By leveraging the relation weight of different element learnt by the AI model, we can build a dictionary to include each concrete API's high relation weight condition. For example, $(API_x : [CATCH, CONDITION])$, where API_x is a concrete API, $CATCH$ appears means that API_x needs to handle exception, $CONDITION$ appears means that API_x needs to do some condition checking, like value state checking or parameter checking. If a dictionary is like $(API_q : NULL)$, this means that the concrete API_q does not need any condition checking or value checking or exception handling.

Then for the second step, our AI model uses dictionary above and travels from beginning node to end node in API call sequence to find each concrete API node's, such as $API'_x s$ high relation weight control or data node API_y whether exists after API_x in the sequence. If node API_y does not appear which means that node API_x mismatches in the sequence, the AI model determines the node API_x is misused, otherwise API_x is right. This step can deal misuse about API, such as exception.

Similarly to step 2, next for the third step, our AI model travels from end node to begining node in API call sequence to find a concrete API node is condition or value state misuse.

By combing the result of above 3 steps, our AI model outputs the final misuse result.

b) Target-Combination Transformer Model: In this section, we implement a Target-Combination Transformer Model to detect API misuse. The model is shown as Figure 6.

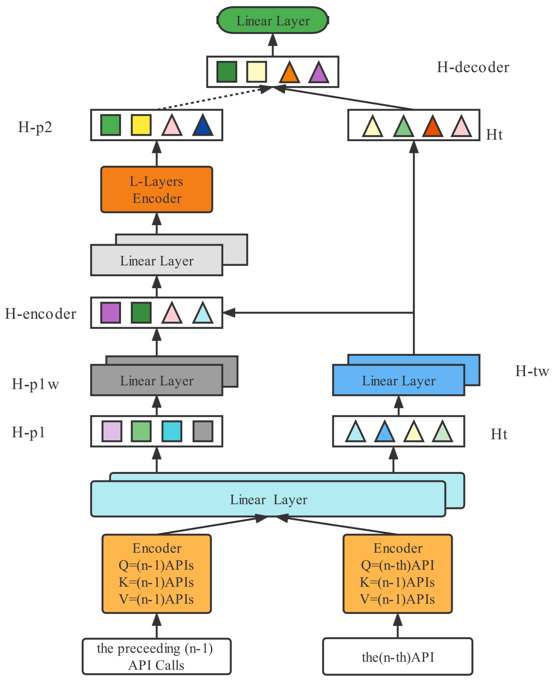


Figure 6. Target-Combination Transformer Model.

Target-Combination Transformer Model is mainly based on three components: (1) a preceeding-sequence-combining target encoder to extract relation between target API node and preceeding APIs as shown in the right of Figure 6 , (2) a preceeding-sequence self encoder to extract self relation as shown in the left of Figure 6, (3) a target-combination linear layer as shown in the top of Figure 6. A target API is the (n-th)API node as shown in the right encoder.

In Figure 6, the attention among different APIs is calculated as Eq(4) and Eq(5) and then fed into the feed-forward and layer normalization layers in the same way as the standard Transformer model. Then the two Encoder output are combined into linear layer. From the output of the linear layer, we get preceeding-sequence-combining target encoder $H_t \in R^{(N,d_{model})}$. H_t learns the target information under the context of the preceeding (n-1)API calls.

Calculated as above, the sequential representations of the preceeding (n-1) API call sequences is obtained by setting Q, K, V. The output from linear layer is calculated as $H_{p1} \in R^{(N,d_{model})}$.

Next, in order to extract the weight attention from the above two representations, we use a feed-forward network layer, namely, $H_{encoder} = [H_{tw}; H_{p1w}]W$, where $W \in R^{(2d_{model},d_{model})}$, $H_{encoder} \in R^{(N,d_{model})}$. After getting $H_{encoder}$, we feed it into several linear layers and Transformer encoder layers to further extract the target API representation.

When coming to the final linear layer, the preceeding (n-1) API calls and target API are combined at the early stage and composed by Transformer encoder layers. To distinguish these two information flows and encourage the final linear layer to obtain the target API message, the L-Layers Encoder in Figure 6 encoder layers are only applied to the (n-1)API calls to obtain a deeper representation H_{p2}. During decoding, H_{p2} and H_t are combined to generate key and value matrics, namely $H_{decoder} = [H_t, H_{p2}]W$, where $H_{decoder} \in R^{(N,d_{model})}$.

In the end of Linear layer, we obtain the output. As the standard Transformer model's second and third step, the target-combination Transformer model also follows the steps to output final misuse result.

IV. Experiment

In this section, we present the experimental plan including the dataset, the evaluation criteria and the final results.

A. Dataset

In our training experiment for our AI models, we use dataset from[24]. The dataset concludes 14422 java files all about JCE(Java cryptography extension) . All the java files are in stable software releases, thus the files provide high quality code and can be generated API call sequence training dataset with high quality for different AI models. The total size of the files is about 50MB. JCE is a package provided by JDK, which can provide the implementation of cryptographic primitive, including block ciphers and MACs(message authenticate codes). Java cryptography APIs provided by JCE are under the package named javax.crypto. 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.

In our practical API misuse detection experiment, we use real API misuse dataset from the state-of-the-art MuBench[10]. MuBench is still actively maintained. It is widely accepted by API misuse detection studies[24][36][37] and contains instances of cryptographic API misuses collected from 62 Java programs. These programs include 6 Android apps and 56 non-Android applications. We managed to use the method from [37] to compile the Java applications into JAR files, which correspond to 149 labeled instances of JCE API misuses. Therefore, we use these 149 instances as ground truth in our evaluation.

As MuBench is public and has great industrial impacts, which enables our AI model to compare different API misuse detection tools, we choose MuBench as our test dataset.

B. Experimental design

a) Comparative experiment: In order to verify the effectiveness of this AI method, this paper compares our models with other state-of-the-art methods which can be used in Java, including using AI method and non AI method, like MuDetect[33], n-gram[32], and 2 other using LSTM methods containing S-LSTM (stacked long short term memory network)[24] and D-LSTM (deep long short term memory network)[36]. MuDetect is not an AI method but a famous method and the detector has a good performance in MuBench. N-gram provided by the author proposes that introducing control condition node from code can catch high level semantic to improve detection ability and n-gram is a famous language tool in bug detection concluding API misuse. Choosing S-LSTM and D-LSTM models is because that the based LSTM models are all about using AI neural network models to detect API misuse and LSTM is a famous model in handling sequence.

We use Precision, Recall and F1 to evaluate the methods. TP is the number of defect reports with the correct location of misuse defects in the source code file under test. FP is the number of defect reports with the wrong locations in the source code file under test. FN is the number of defect reports that the model fails to identify. With the above three summation values, the Precision metric is defined as:

$$Precision = \frac{TP}{TP + FP} \tag{7}$$

Similarly, the Recall metric is defined as:

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

The harmonic mean (F1) of precision rate and recall rate is defined as:

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{9}$$

b) Experiment parameter design: Assuming that the acceptable level of defect detection model in the experiment is set to k. When a node in an API call sequence is in the output node probability list and the probability is between top 1 and top k, it is considered that the predicted node is at the acceptable level, otherwise, it is a potential API misuse. Then our models follow their second and third step to finally determine whether it is a misuse. In the API misuse detection models based on Transformer, the hyper parameter design is as follows: HIDDEN_SIZE= 250, NUM_LAYER=4, NUM_EPOCH=20, BATCH_SIZE = 64, DROP_OUT= 0.1, LEARNING_RATE = 0.005,EMBEDDING_SIZE = 512, Multi-Head-Attention NUM_HEADS= 4. When training our models, we divide the total dataset from[24] mentioned in section $Dataset$ into 2 parts, the training set is 80%, and the verification set is 20%.

C. Model training

The training results are shown in Figure 7 and Figure 8.

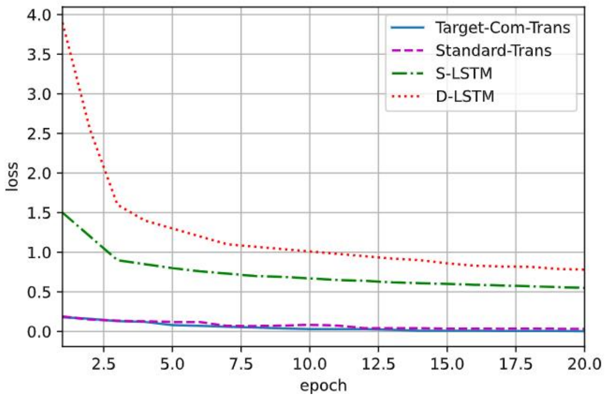


Figure 7. Comparison diagram of model loss value.

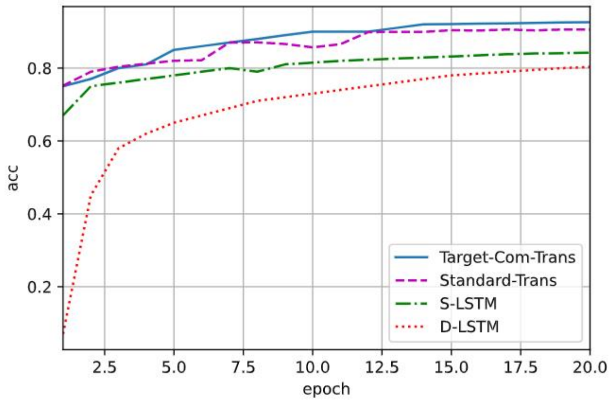


Figure 8. Comparison diagram of model accuracy.

Figure 7 shows the comparison of the loss values of 4 models. The abscissa represents the number of iterations and the ordinate represents the loss value; Figure 8 shows the accuracy change trend during model training. The abscissa represents the number of iterations and the ordinate represents the accuracy.

It can be seen from Figure 7 and Figure 8 that our Transformer models designed in this paper are both better than D-LSTM and S-LSTM methods in terms of loss value and accuracy. At 20 iterations, the loss of standard Transformer model is 0.019, and the loss of target-combination Transformer model is 0.017, which are both lower than the loss of 0.567 of S-LSTM and 0.772 of D-LSTM. At the same time, the accuracy of standard Transformer model reaches 0.908 and target-combination Transformer reaches 0.925, which both are higher than that of S-LSTM and D-LSTM. This is because the attention mechanism in Transformer processes the whole sentence sequence, which is not disturbed by the long-term dependence problem, and there is no risk of losing past information. At the same time, different elements in code establish different weight relationships according to the degree of interdependence. Figure 7 and Figure 8 also show that with the increase of the number of iterations, the values of the loss function and accuracy tend to converge. At the same time, the convergence speed of the models in this paper are faster than that of the comparative AI models. This is because the Transformer models in this paper avoid the disadvantage of processing sequences in the LSTM model, and the processing of sequences can be calculated in parallel.

D. Model interpretability based on Transformer

In order to understand how our models can make good prediction about API misuses, we develop a visualization to demonstrate the inner mechanism of the self-attention in our models. Here we select standard-Transformer. We choose the API call sequence mined from the Java code snippet in Figure 4 as the example. Its encoding string is listed as follows:

IF-CONDITION-TRY-TRYBLOCK-
java.io.FileInputStream.new(java.io.File)-
java.io.FileInputStream.read()-CATCH-
java.io.IOException.printStackTrace()-CATCH-
java.io.FileNotFoundException.printStackTrace()-Finally

From the API call sequence, our algorithm can establish the dictionary table with API strings and their positions as illustrated in Table I

TABLE I
API SEQUENCE DICTIONARY

Position	API
0	IF
1	CONDITION
2	TRY
3	TRYBLOCK
4	jave.io.FileInputStream.new(java.io.File)
5	java.io.FileInputStream.read()
6	CATCH
7	java.io.IOException.printStackTrace()
8	CATCH
9	java.io.FileNotFoundException.printStackTrace()
10	FINALLY

The misuse inference of the code sequence is executed in the four layers of our Transformer model. The top layer represents the outermost layer of the encoder, and the bottom layer represents the input layer close to the encoder. There are four heads in each layer, and the weight relationship is shown in Figure 9. The heads of different layers represent the weights of the keys and queries in the process of sequence prediction. The vertical axis of each head represents the word

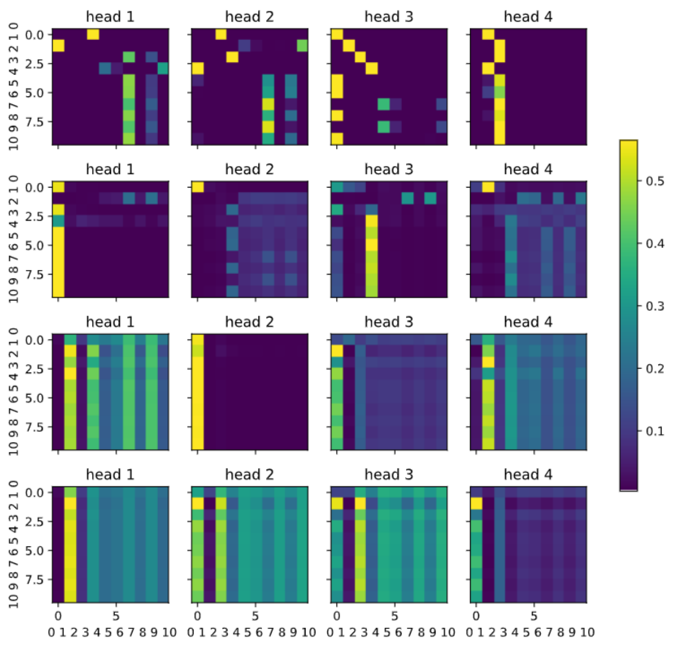


Figure 9. Visualization of the attention mechanism in our Transform model when it infers API misuses from a Java code snippet

at the query position of the sequence, and the horizontal axis represents the sequence key. After passing through the four layers of encoder, it is in the outermost layer of the encoder.

1) From head 1, one can see the catch word corresponding to the key positions 6 and 8 is associated and the query positions 5, 7 and 9 , namely java.io.FileInputStream.read(), java.io.IOException.printStackTrace(), java.io.FileNotFoundException.printStackTrace(). The key and query pairs clearly exhibit a high weight value, which indicates that the catch statements are closely related to specific exceptions.
2) From head 2, one can see that the weight value is high between the try statement with the key position 2 and the tryblock statement with the key position 3. This observation indicates that the occurrence of the try statement is closely associated with the tryblock structure with a high possibility.
3) From head 3, one can also see that the key position 4 namely, java.io.FileInputStream.new(java.io.File) has a high weight with the query positions 6 and 8, which indicates that the model pays more attention to the exceptions caused by creation of Java.io.File.
4) From head 4, one can see that there is almost a highlighted column of the try statement corresponding to the key position 2, indicating that almost all the API elements behind the try statement in the sequence have a high weight with it.

Based the above observations, we can conclude that the self-attention mechanism of the Transformer model is very good at capturing the structural and semantic associations among different API call elements in the code. Such an advantage enables the Transformer based models to make more accurate predictions about the violations of API usage in the code snippets in our test dataset.

E. Model effect in API misuse detection

This section describes an API misuse detection experiment using test dataset mentioned above.

Figure 10, Figure 11 and Figure 12 displays the F1 value, the precision and recall values of the methods respectively. The abscissa represents the acceptable threshold (Top-k) for predicting API probability.

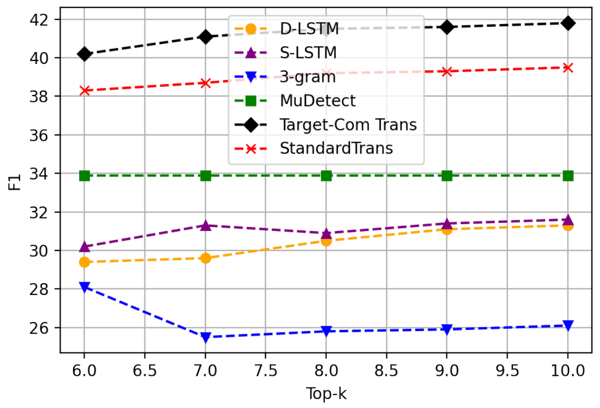


Figure 10. the F1 Performance Comparison of the API misuse detection models

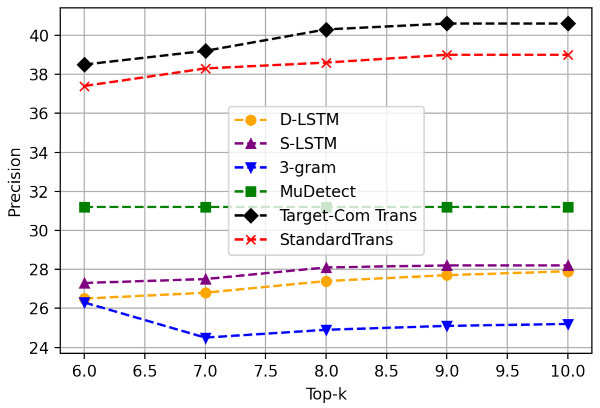


Figure 11. the Precision Performance Comparison of the API misuse detection models

It can be seen from Figure 10, Figure 11 and Figure 12 that among the three metrics defined above, the Transformer models in this paper show a better result. Since the recall value of our models in this paper exceed the famous MuDetect, it has a better detection performance; When the value of Top-k is top-10, the effect of our models are the best.

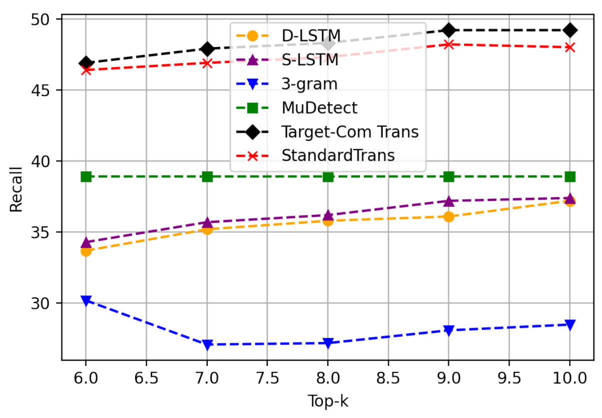


Figure 12. the Recall Performance Comparison of the API misuse detection models

Figure 10, Figure 11 and Figure 12 also show that under different Top-k values, the performance of API misuse detection method based on Transformer is relatively stable, and other methods like n-gram may have certain mutations in varying degrees. Though all methods use the same MuBench dataset, our proposed models show a better performance. Based on Transformers models can well catch the attention weight in API call sequence, thus leads to our models' better ability to detect API misuse.

F. Case study

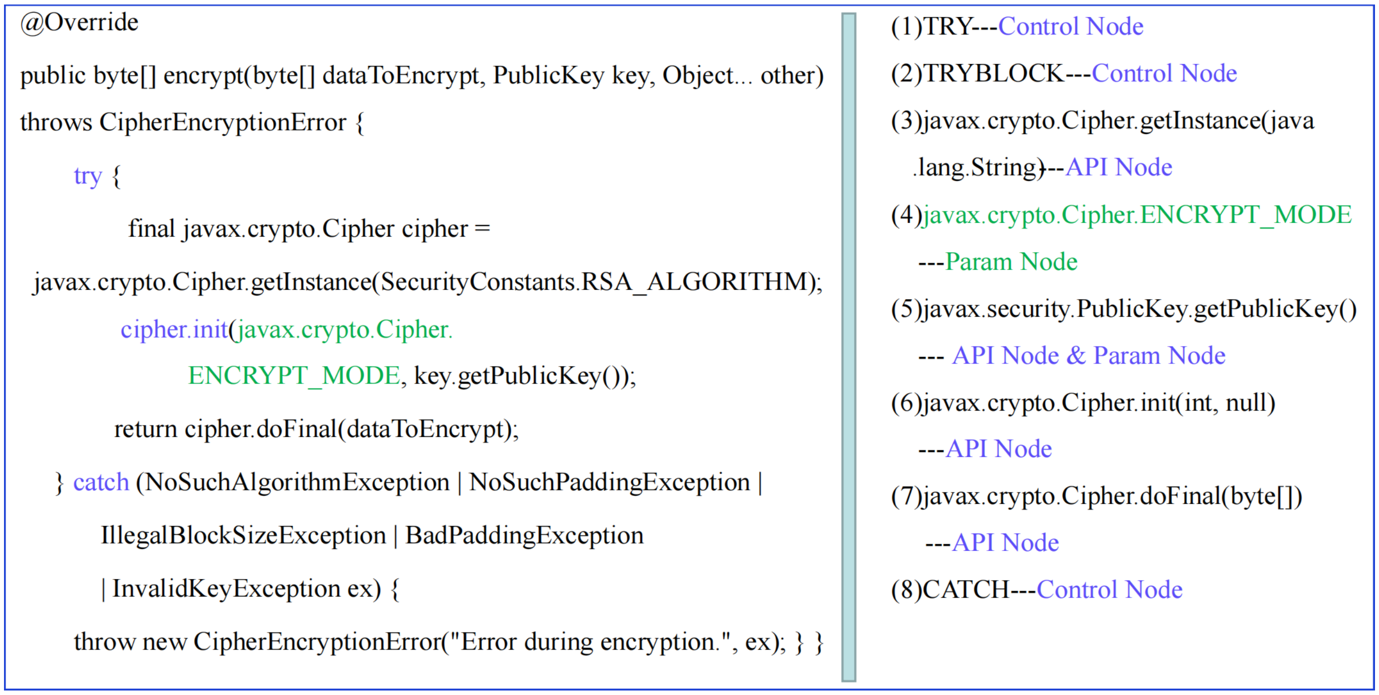
In this section, we use two typical examples to explain how our AI models work in detecting real API misuse. We still select standard Transformer model.

a) Exception misuse detection: Figure 13 and Figure 14 from our training dataset are right cases to use $Cipher.init()$, while Figure 15 is a real misuse in test dataset. When using $Cipher.init()$ to implement different functions, one should notice that this may happen exception. We convert the right usage of API code into ACSG and then extract the API call sequence which represents the execution of code. Next we generate dataset from API call sequence as described in section III to train our AI models. After that, the models has learnt the usage of API. For example, in Figure 13, when a method is used to encrypt data, TRY as control node, $Cipher.ENCRYPT\_MODE$ as param node often appear in the front of code snippet, after the mode param appears, the $Cipher.init()$ and $doFinal()$ method are called. Then $Catch$ different exception statement appears to catch the possible exception. Sometimes $Cypher.init()$ and $doFinal()$ do not always appear in the same time and $Catch$ statement sometimes follows nearest $Cypher.init()$ as shown in Figure 14. Besides, other complete cases may happen, for example, using class $KeyFactory$ to get $privateKey$ before $Cipher.init()$ as shown in Figure 15. Our AI model can well learn the relation weight among different nodes as explained in subsection model interpretablity and can predict each position in API call sequence with accepted possibility. For example, when an API call sequence begins with TRY and $TRYBLOCK$, the model predicts the next node should be $Cipher.getInstance(null)$ with possibility 50% or other nodes like $Cipher.init()$ or $SecretKeyFactory.getInstance()$ with different accepted possibilities. As different nodes may appear, we select Top-k to represent the possible node count. If the predicted node possibility is very low or out of Top-k, that means AI model thinks the node can hardly appear, once such node appears, it may be a potential misuse node. We also build a dictionary to store each concrete API node's condition limit or value limit or exception limit by finding each API's high relation weight control or data node as described in Section III. This is the first step and can well find API call order misuse. But as complete cases may appear as we analysis, our AI model executes second and third step to deeply determine the detection result.

The second step is that, our AI model uses the dictionary above and travels from beginning node to end node in API call sequence to find each concrete API node's, such as $A's$ high relation weight control or data node, such as B whether exists after A in the sequence. The high relation weight threshold can be set 0.3 or other accepted values. If node B does not appear which means that node A mismatches in the sequence, the AI model determines the node A is misused, otherwise AI model determines A is right. This step can find exception misuse.

Just like step 2, our AI model proceeds step 3 from end node to beginning node to find whether an API node mismatches. This step can find condition or value state misuse. By combing the result of above 3 steps, our AI models output the final misuse result. For example, $Cipher.init()$ is a Node in Figure 13, and AI model has learnt that it is with high relation value with $Catch$ and builds a dictionary $(Cipher.init() : [CATCH])$. When the AI model travels the API call sequence from the API node's current position to end to find whether an over threshold relation weight control node $Catch$ exists. If the satisfied relation weight node exists, then the API node $Cipher.init()$is not misuse, otherwise, the $Cipher.init()$ is misuse. In Figure 15, by using our above steps, our model thinks $Catch$ statement as a high relation weight control node to $Cipher.init()$ should appear in code. However, the $Catch$ statement does not appear, then our models think the $Cipher.init()$ is a misuse. By counting all the misuses nodes in code for a file, we then use the Eq(7), Eq(8), Eq(9) to calculate Precision, Recall and F1.

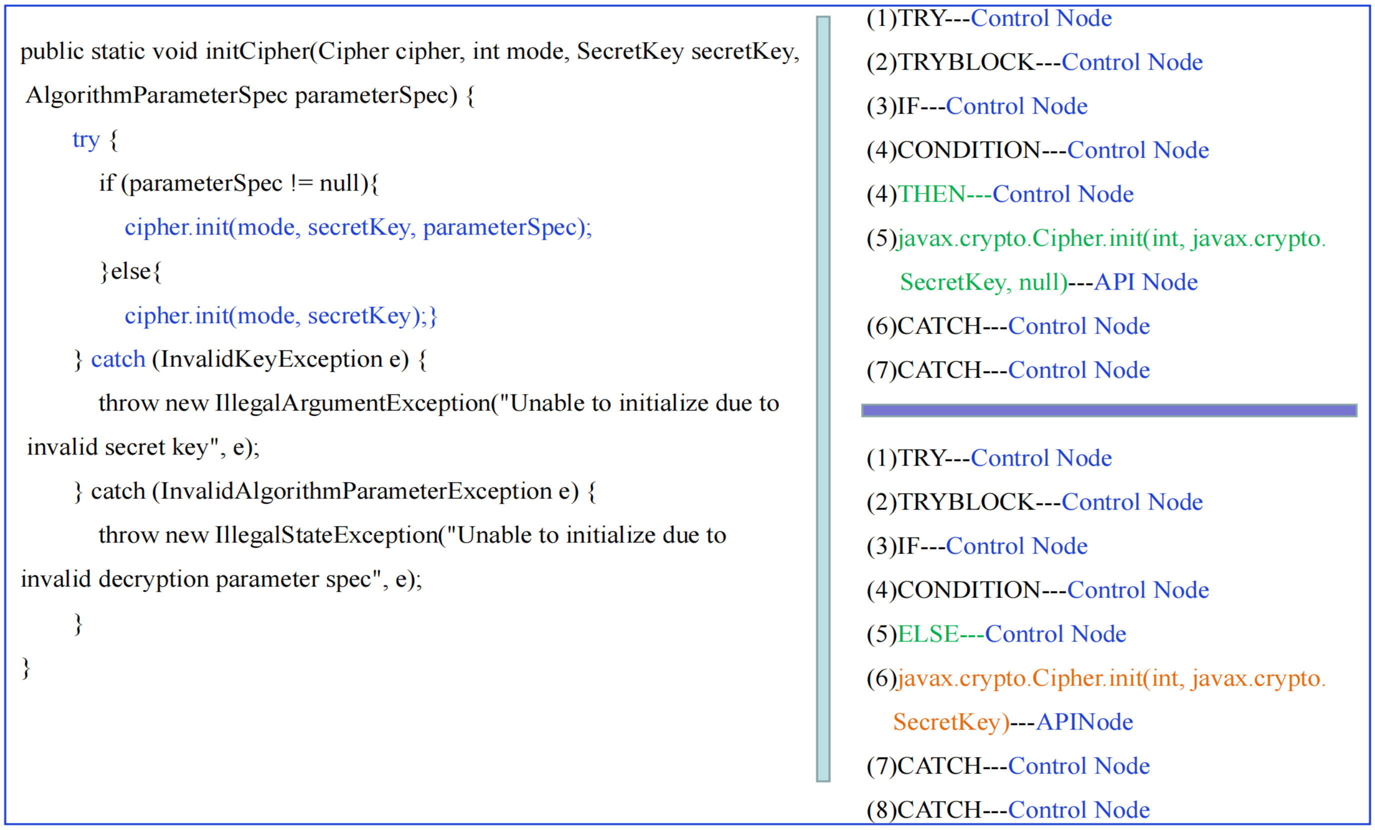
b) Parameter misuse detection: Figure 16 is a parameter type misuse about $Cipher.doFinal()$ in real test dataset. In right usage about the API, the parameter type should be encoding $byte[]$. The example shows that the string is not converted into specifying encoding and then passed to $Cypher.doFinal()$, this causes a parameter type misuse. As shown in Figure 15, $cipher.doFinal(plainText.getBytes("UTF-8"))$ is a right usage about $Cipher.doFinal()$. Like the exception misuse type detecting, our AI model first predicts each position API



```
@Override
public byte[] encrypt(byte[] dataToEncrypt, PublicKey key, Object... other)
throws CipherEncryptionError {
      try {
            final javax.crypto.Cipher cipher =
 javax.crypto.Cipher.getInstance(SecurityConstants.RSA_ALGORITHM);
           cipher.init(javax.crypto.Cipher.
                  ENCRYPT_MODE, key.getPublicKey());
          return cipher.doFinal(dataToEncrypt);
   } catch (NoSuchAlgorithmException | NoSuchPaddingException |
        IllegalBlockSizeException | BadPaddingException
        | InvalidKeyException ex) {
      throw new CipherEncryptionError("Error during encryption.", ex); } }
```

```
(1)TRY---Control Node
(2)TRYBLOCK---Control Node
(3)javax.crypto.Cipher.getInstance(java
  .lang.String)---API Node
(4)javax.crypto.Cipher.ENCRYPT_MODE
   ---Param Node
(5)javax.security.PublicKey.getPublicKey()
   --- API Node & Param Node
(6)javax.crypto.Cipher.init(int, null)
   ---API Node
(7)javax.crypto.Cipher.doFinal(byte[])
    ---API Node
(8)CATCH---Control Node
```

Figure 13. A good example for using $Cipher.init()$. The left part shows code, while the right part shows the API call sequence from up down which represents the execution of the code. API call sequence contains API Node and Param Node and Control Node. One API node can be as another's Param Node, such as node $key.getPublicKey()$.

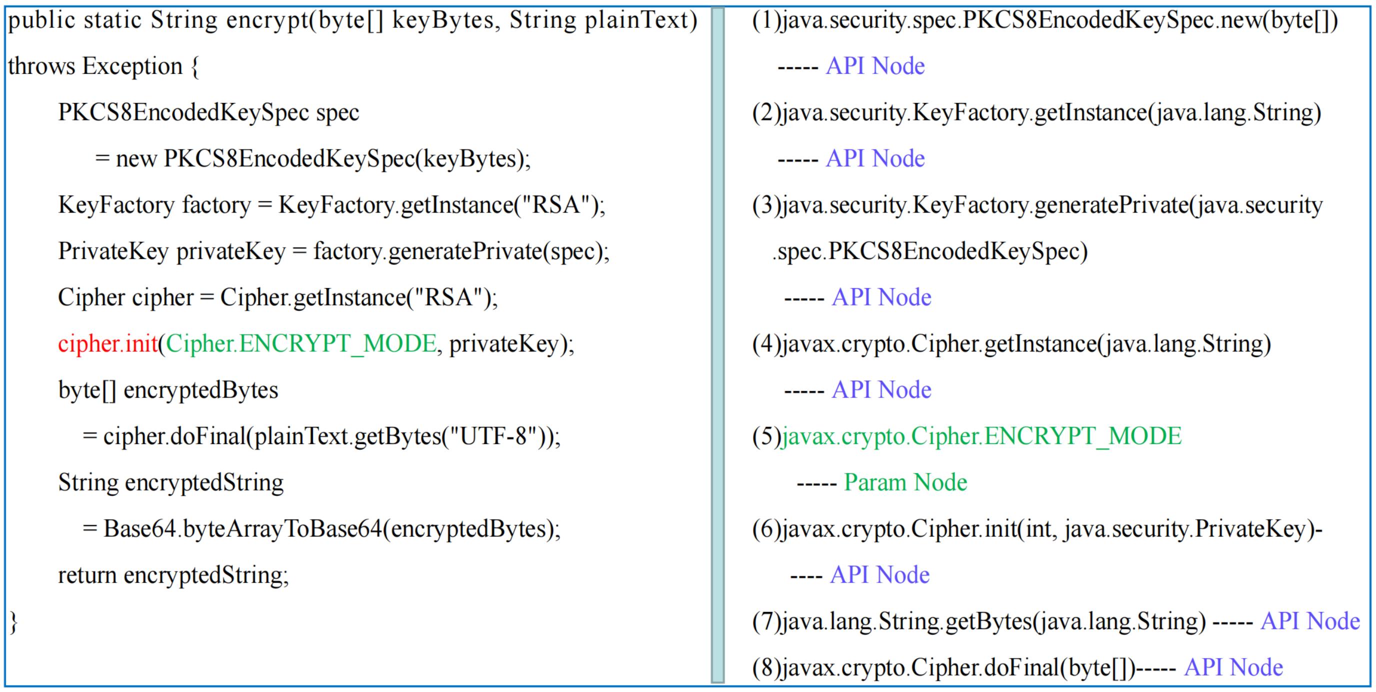


```
public static void initCipher(Cipher cipher, int mode, SecretKey secretKey,
 AlgorithmParameterSpec parameterSpec) {
      try {
          if (parameterSpec != null){
             cipher.init(mode, secretKey, parameterSpec);
          }else{
             cipher.init(mode, secretKey);}
      } catch (InvalidKeyException e) {
          throw new IllegalArgumentException("Unable to initialize due to
 invalid secret key", e);
      } catch (InvalidAlgorithmParameterException e) {
          throw new IllegalStateException("Unable to initialize due to
invalid decryption parameter spec", e);
      }
}
```

```
(1)TRY---Control Node
(2)TRYBLOCK---Control Node
(3)IF---Control Node
(4)CONDITION---Control Node
(4)THEN---Control Node
(5)javax.crypto.Cipher.init(int, javax.crypto.
   SecretKey, null)---API Node
(6)CATCH---Control Node
(7)CATCH---Control Node

(1)TRY---Control Node
(2)TRYBLOCK---Control Node
(3)IF---Control Node
(4)CONDITION---Control Node
(5)ELSE---Control Node
(6)javax.crypto.Cipher.init(int, javax.crypto.
   SecretKey)---APINode
(7)CATCH---Control Node
(8)CATCH---Control Node
```

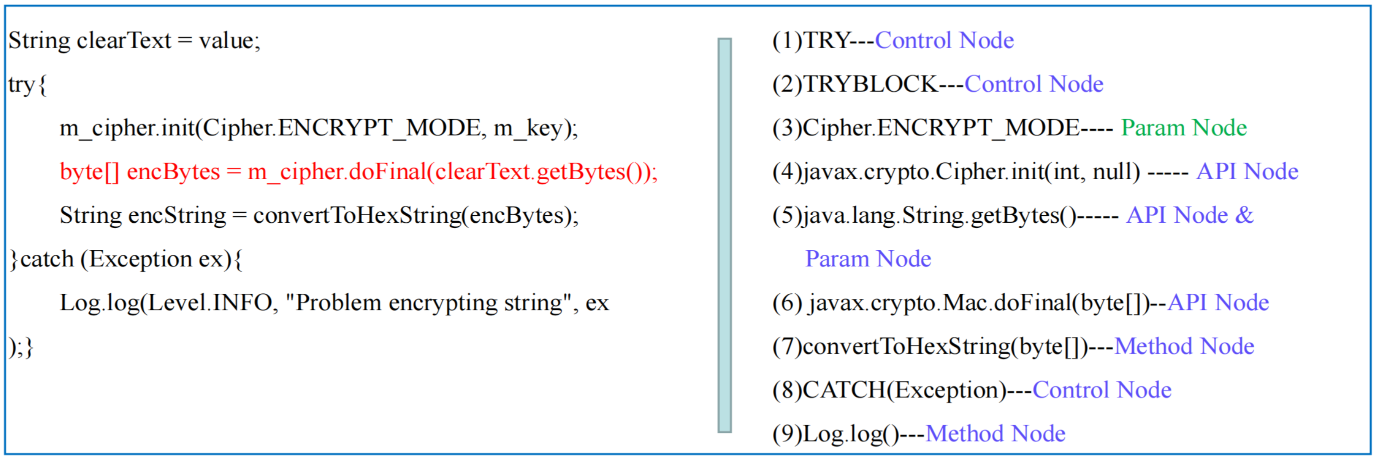
Figure 14. A good example with execution branch for using $Cipher.init()$. The left part shows code, while the right part shows two API call sequences from up down which represents the execution flow of the code. Since if is in the code, the execution flow may yield different branch. API call sequences contains API Node and Control Node.



```
public static String encrypt(byte[] keyBytes, String plainText)
throws Exception {
    PKCS8EncodedKeySpec spec
        = new PKCS8EncodedKeySpec(keyBytes);
    KeyFactory factory = KeyFactory.getInstance("RSA");
    PrivateKey privateKey = factory.generatePrivate(spec);
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.ENCRYPT_MODE, privateKey);
    byte[] encryptedBytes
       = cipher.doFinal(plainText.getBytes("UTF-8"));
    String encryptedString
       = Base64.byteArrayToBase64(encryptedBytes);
    return encryptedString;
}
```

```
(1)java.security.spec.PKCS8EncodedKeySpec.new(byte[])
  ----- API Node
(2)java.security.KeyFactory.getInstance(java.lang.String)
  ----- API Node
(3)java.security.KeyFactory.generatePrivate(java.security
  .spec.PKCS8EncodedKeySpec)
  ----- API Node
(4)javax.crypto.Cipher.getInstance(java.lang.String)
  ----- API Node
(5)javax.crypto.Cipher.ENCRYPT_MODE
   ----- Param Node
(6)javax.crypto.Cipher.init(int, java.security.PrivateKey)-
   ---- API Node
(7)java.lang.String.getBytes(java.lang.String) ----- API Node
(8)javax.crypto.Cipher.doFinal(byte[])----- API Node
```

Figure 15. A real misuse case about $Cipher.init()$ in dataset. When using $Cipher.init()$, one should notice that this may happen exception. The left part shows code, while the right part shows the API call sequence from up down which represents the execution flow of the code. API call sequences contains API Node and Param Node.



```
String clearText = value;
try{
    m_cipher.init(Cipher.ENCRYPT_MODE, m_key);
    byte[] encBytes = m_cipher.doFinal(clearText.getBytes());
    String encString = convertToHexString(encBytes);
}catch (Exception ex){
    Log.log(Level.INFO, "Problem encrypting string", ex
);}
```

```
(1)TRY---Control Node
(2)TRYBLOCK---Control Node
(3)Cipher.ENCRYPT_MODE---- Param Node
(4)javax.crypto.Cipher.init(int, null) ----- API Node
(5)java.lang.String.getBytes()----- API Node &
   Param Node
(6) javax.crypto.Mac.doFinal(byte[])--API Node
(7)convertToHexString(byte[])---Method Node
(8)CATCH(Exception)---Control Node
(9)Log.log()---Method Node
```

Figure 16. A real misuse case about $Cipher.doFinal()$ in dataset. When using $Cypher.doFinal()$, the parameter type should be $encodingbyte[]$. The example shows that the string is not converted into specifying encoding and then passed to $Cypher.doFinal()$.The left part shows code, while the right part shows the API call sequence from up down which represents the execution flow of the code. API call sequences contains API Node and Param Node.

node possibility in API call sequence. If the possibility is very low or the node is out of Top-k, then the Node may cause API misuse. Then, our model travels the API call sequence from end to beginning node to find whether $doFinal()'s$ high relation weight value node $getBytes(java.lang.String)$, not $getBytes()$ exists. If $getBytes(java.lang.String)$ exists, then the AI model thinks the $doFinal()$ is used right, otherwise, $doFinal()$ is a misuse. In the final result, our model determines Figure 16 is a misuse.

From the above examples, we can see that, our AI models executes 3 steps to find API misuse. As our models do well in catching different element's relation, the detecting API misuse ability can be good at catching different kinds of API misuse, like exception misuse, condition misuse, value misuse and so on.

V. Conclusion

In this paper, we propose the Transformer based models to learn the representations of API usages and perform API misuse detection. Experimental results confirm that both the standard Transformer model and target-combination Transformer model developed in our work achieve a higher predicative performance than the other methods in terms of precision, recall and F1. Besides, we visualize the attention weights of our models when making API misuse detection with Java code examples. The visualization explains how the Transformer models represent the implicit API usages in the Java code, which provide a good way for software developers to understand the model's predictive behaviors. In the end, we use 2 typical examlpes to show how our models work in detecting real API misuse cases. In future work, we still need other experiments to assess the efficiency of the proposed method, For example, we can use different dataset.

Acknowledgment

This paper is supported by National Key R&D Program of China (Funding No. 2018YFB1402800).

References

[1] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang et.al, "Do the dependency conflicts in my project matter?" in Proceedings of the 2018 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1–12, 2018.

[2] Maven repository, https://maven.apache.org/, February 2018.

[3] Z. Li, J. Z. Wu, M. S. Li. "Study on key issues about API usage". Ruan Jian Xue Bao/Journal of Software,vol. 29, pp. 1716-1738, 2018.

[4] Y. Zhou, R. H. Gu, T. L. Chen, Z. Q. Huang, P. Sebastiano, G. Harald. "Analyzing APIs documentation and code to detect directive defects". In: Proc. of the 39th Int'l Conf. on Software Engineering, pp. 27-37, 2017.

[5] L. Shuang, G. Bai, J. Sun, S. D. Jin. "Towards using concurrent Java API correctly". In: Proc. of the 21st Int'l Conf. on Engineering of Complex Computer Systems, pp. 219-222, 2016.

[6] P. Sacramento, B. Cabral, P. Marques . "Unchecked exceptions: Can the programmer be trusted to document exceptions". In: Proc. of the 2nd Int'l Conf. on Innovative Views of NET Technologies, 2006.

[7] U. Dekel , J. Herbsleb, "Improving api documentation usability with knowledge pushing," in Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, pp. 320–330, 2009.

[8] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online qa forum reliable?: a study of api misuse on stack overflow," in Proceedings of the 40th International Conference on Software Engineering, pp. 886–896, 2018.

[9] L. Seonah, R. Wu, S. C. Cheung, and S. Kang, "Automatic detection and update suggestion for outdated api names in documentation," IEEE Transactions on Software Engineering, 2019.

[10] S. Amann, S. Nadi, H.A. Nguyen, "MUBench: A benchmark for API-misuse detectors". In: Proc. of the 13th Int'l Conf. on Mining Software Repositories, pp. 464-467, 2016.

[11] Q. Gao, H. Zhang, J. Wang, "Fixing recurring crash bugs via analyzing qa sites (T)". In: Proc. of the 2015 30th IEEE/ACM Int'l Conf. on Automated Software Engineering, pp. 307-318, 2015.

[12] M. Monperrus , M. Mezini, "Detecting missing method calls as violations of the majority rule," ACM Trans. Softw. Eng. Methodology, vol. 22, pp. 1–25, 2013.

[13] J. Sushine, J. D. Herbsleb, and J. Aldrich, "Searching the state space: A qualitative study of API protocol usability," in Proc. 23rd IEEE Int. Conf. Program Comprehension, pp. 82–93, 2015.

[14] S. Fahl, M. Harbach, T. Muders, L. Baumg€artner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in)security," in Proc. 19th ACM Conf. Comput. Commun. Secur., pp. 50–61, 2012.

[15] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in Proc. Conf. Comput. Commun. Secur., pp. 73–84, 2013.

[16] S. Nadi, S. Kruger, M. Mezini, and E. Bodden, ""Jumping through hoops": Why do developers struggle with cryptography APIs?" in Proc. 38th Int. Conf. Softw. Eng., pp. 935–946, 2016.

[17] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating SSL certificates in non-browser software," in Proc. 19th ACM Conf.Comput. Commun. Secur., pp. 38–49, 2012.

[18] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in ACM SIGSOFT Software Engineering Notes, vol. 30, pp. 306–315, 2005.

[19] C. Lindig, "Mining patterns and violations using concept analysis," in The Art and Science of Analyzing Software Data. Elsevier, pp.17–38, 2016.

[20] A. Wasylkowski, A. Zeller, C. Lindig, "Detecting object usage anomalies," in Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM, pp. 35–44, 2007.

[21] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in ACM SIGPLAN Notices, vol. 42, pp. 123–134, 2007

[22] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N.Nguyen, "Graph-based mining of multiple object usage patterns," in Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 383–392, 2009.

[23] M. Acharya and T. Xie, "Mining api error-handling specifications from source code," in International Conference on Fundamental Approaches to Software Engineering, Springer, pp. 370-384, 2009.

[24] S. Y. OuYang , F. Ge , L. Kuang , Y. Y. Yin. "API Misuse Detection based on Stacked LSTM". CollaborateCom, 2020.

[25] Vaswani, Ashish, et al, "Attention is all you need."Advances in Neural Information Processing Systems, 2017.

[26] Y. Yang, J. M. Hao, C. J. Li, Z. L. Wang, J. G. Wang, F. Z. Zhang et al, "Query-aware Tip Generation for Vertical Search". Conference on Information and Knowledge Management 2020.

[27] H. Zhong , L. Zhang , T. Xie , H. Mei . "Inferring resource specifications from natural language API documentation". In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering, pp. 307-318, 2009.

[28] X. Xiao, Z. H. Xing, X. Xia, X. W. Xi, L. M. Zhu ,"API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph." 2020 35th IEEE/ACM International Conference on Automated Software Engineering ,2020

[29] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, pp. 320–330, 2009

[30] Z. Li, Y. Zhou, "PR-miner: Automatically extracting implicit programming rules and detecting violations in large software code". SIGSOFT Software Engineering Notes, vol.30, pp. 306-315, 2005.

[31] M. Wen, Y. P. Liu, R. X. Wu, X. Xie, S. C. Cheung , Z. D. Su. 2019. "Exposing Library API Misuses via Mutation Analysis". In Proceedings of the 41th International Conference on Software Engineering, 2019.

[32] S. Wang , D. Chollak, D. Movshovitz-Attias, L. Tan . "Bugram: Bug detection with n-gram language models". In: Proc. of the 2016 31st IEEE/ACM Int'l Conf. on Automated Software Engineering, pp. 708-719, 2016.

[33] S. Amann. "Investigating next steps in static API-misuse detection." 2019 IEEE/ACM 16th International Conference on Mining Software Repositories, 2019.

[34] A. Wasylkowski, A. Zeller. "Mining Temporal Specifications from Object Usage". Automated Software Engineering, vol. 18, pp. 263–292, 2011.

[35] N. Smith , D.V. Bruggen, F. Tomassetti . "Java Parser: Visited analyse, transform and generate your Java code base", 2017.

[36] X. Wang, C. Chen, Y.F. Zhao, X. Peng, W.Y. Zhao. "API misuse bug detection based on deep learning". Ruan Jian Xue Bao/Journal of Software, vol.30, pp. 1342-1358, 2019.

[37] Y. Zhang, M. Kabir, Y. Xiao, D.F. Yao, N. Meng."Automatic Detection of Java Cryptographic API Misuses: Are We There Yet?". Transactions on Software Engineering, 2021