Generating Adversarial Examples of Source Code Classification Models via Q-Learning-Based Markov Decision Process

Junfeng Tian
Hebei University
Baoding, China
tjf@hbu.edu.cn

Chenxin Wang
Hebei University
Baoding, China
wchenxin33@163.com

Zhen Li
Hebei University
Baoding, China
lizhenhbu@gmail.com

Yu Wen
Hebei University
Baoding, China
wenyu@hbu.edu.cn

2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-5813-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/QRS54544.2021.00090

***Abstract*—Adversarial robustness becomes an essential concern in Deep Learning (DL)-based source code processing, as DL models are vulnerable to the deception by attackers. To address a new challenge posed by the discrete and structural nature of source code to generate adversarial examples for DL models, and the insufficient focus of existing methods on code structural features, we propose a *Q-Learning-based Markov decision process* (QMDP) performing semantically equivalent transformations on the source code structure. Two key issues are mainly addressed: (i) how to perform attacks on source code structural information and (ii) what transformations to perform when and where in the source code. We demonstrate that effectively tackling these two issues is crucial for generating adversarial examples for source code. By evaluating C/C++ programs working on the source code classification task, we verified that QMDP can effectively generate adversarial examples and improve the robustness of DL models over 44%.**

***Index Terms*—adversarial examples, adversarial training, robustness, source code, deep learning**

I. Introduction

In the field of software engineering, Deep Learning (DL) methods have been successfully applied to many source code classification tasks, such as code functionality classification [1], code clone detection [2], [3], defect prediction [4], [5], and vulnerability detection [6], [7]. Automatic learning of code features using DL can effectively implement source code classification and reduce the cost of software development, testing, and maintenance. In addition, efficient source code classification can also helps software developers to understand and maintain program code, and helps subsequent software development.

However, DL models for source code classification are vulnerable to malicious attackers, the adversarial robustness becomes a critical issue. By adding elaborate perturbations to the original inputs, an attacker can often generate adversarial examples that are similar from the human side, but may cause the DL models to produce incorrect results. This risk is currently addressed mainly by adversarial learning [8]. It generates adversarial examples by performing adversarial attacks on the original program files from the attack perspective, and uses them to deceive deep neural networks. From the defense perspective, it mixes adversarial examples into the training set for adversarial training to improve the robustness of the attacked DL models. Nevertheless, adversarial learning in the source code processing domain is extremely difficult, unlike the image processing [6], [9] and the natural language processing (NLP) [10], [11]. The source code has a highly structured nature while possessing the discrete feature, which needs to strictly follow the constraints of the programming language coding rules in terms of semantics and syntax. Therefore, adversarial learning needs to ensure the integrity of the code syntax and semantic structure, which makes it more complicated to generate adversarial examples on DL-based source code classification models.

In recent years, many approaches have been used to generate adversarial examples by making modifications to the original inputs of DL models, such as variable renaming [12]–[15], equivocal expression substitution [16], and dead code insertion [16]. However, these approaches do not take into account the significance of code structure features. Normally program code structures contain more semantic and syntactic information [1], [17]. In addition, the latest DL models [2], [6], [18] for source code classification are different from the traditional text- and token-based [19], [20] approaches, which are mainly based on capturing more syntactic structure information for source code processing. However,if we add some modification to the code structure, for example, replacing the $while$ statement with a semantically equivalent for statement, the program is functionally identical from a human side after such a simple transformation, but it may mislead the DL models to produce the incorrect outputs. Therefore, we propose the hypothesis that the structural features of the source code are non-robust and adversarial examples can be generated by making modifications to the structural information of source code.

Meanwhile, due to the structured nature of the source code, performing different transformations at the same location in the code or performing the same transformation at different locations may generate different code snippets after the adversarial attack. When attacks are performed on code structured features, it is necessary to pay attention to the ensuing combination explosion of the code space that

2693-9177/21/$31.00 ©2021 IEEE
DOI 10.1109/QRS54544.2021.00090

may occur. Therefore, we believe that the current adversarial learning of DL models for source code processing has the two main challenges: (1) Perform adversarial attacks on source code structural features effectively as well as ensure that the functionality of the code remains consistent; (2) Solve the problem of combinatorial explosion of the code space brought about by applying multiple transformation attacks at multiple locations during adversarial attacks.

Our contributions. In this paper, we propose a *Q-learning-based Markov Decision Process* (QMDP) approach to implement adversarial attacks on source code structure information to generate adversarial examples.

To address challenge (1), we define a set of transformation operations to generate adversarial examples by modifying the structural features of source code. This means that we iteratively perform structural transformation attacks on the inputs of the DL models to generate code segments that are semantically identical but syntactically different from the original code. The functionalities of the generated new code segments remain constant but can successfully mislead the DL models.

To address challenge (2), we use a strategic sampling approach which introduces the idea of reinforcement learning, transforms the combinatorial problem of the code transformation space into a combinatorial alignment problem of the sequence of transformation operations. It will allow for faster and more effective attacks. Specifically, we treat the adversarial examples generation process as a Markov decision process and introduce Q-learning to solve the control problem throughout the attack process. During the iterative attacks, the attack values corresponding to each transformation operation are continuously learned and searched by the significance of the transformation operations to guide the attacks in order to obtain the maximum cumulative value, i.e., the degree to which the classifier deviates from the correct classification.

We conducted an evaluation on a code functionality classification task applied to the Open Judge (OJ) dataset. It is verified that the source code structural features are non-robust and changing them can affect the classification results of the classifier for program functionalities; the effectiveness of QMDP for generating adversarial examples is also verified and using the adversarial examples generated by QMDP to perform adversarial training on the source code classification models LSTM and ASTNN applied on the OJ dataset, the adversarial robustness of the models can be improved by 44.0% and 44.8%, respectively.

II. Related Work

A. DL-based Source code Classification

Automatic classification of program source code according to some criteria like functionality, programming language or presence of bugs [1], [21] can significantly reduce the cost of software development, testing and maintenance. For instance, vulnerability detection [6], [7], source code functionality classification [1], [2], [18], function naming [22], all of which belong to the DL-based source code classification. During software development and maintenance, automatic processing using the DL model effectively stores program source code by category, which can help developers to extract and understand the program and contribute to the efficiency of subsequent software development and maintenance.

The traditional text- and token-based approach [19], [20], [23] treats source code as a natural language sequence and tokenizes its serialization. The source code classification is performed by extracting tokenized information to train the model.

More recently DL-based source code classification methods have aimed to extract more information about the syntactic structure of source code to identify source code functionalities. CDLH [3] combines Tree-LSTM to represent the functionality and semantics of source code. Mou et al. [1] proposed a Tree-Based Convolutional Neural Network (TBCNN) approach based on program's Abstract Syntax Trees (ASTs) in learning to understand source code and accomplish the task of categorizing source code functionalities. However, TBCNN cannot capture the essential semantic dependencies between code elements accurately. Accordingly, Bui et al. [18] proposed a Tree-Based Capsule Networks (TreeCaps) approach to capture syntactic structure Information and dependencies in code by processing them automatically. Zhang et al. [2] proposed an AST-based Neural Network (ASTNN) approach for source code classification, which splits a large AST of a code snippet into a set of small trees at the statement level. The large AST tree is partitioned into a short sequence of small statement trees, using a Recurrent Neural Network (RNN) to encode the statements and the sequential dependencies between statements into vectors to serve as the input to the neural network.

In this paper, we use the widely used serialization processing-based classifier LSTM and the most classical AST-based classifier ASTNN for the attacked DL models in our experiments, which are used to verify the attack effectiveness of QMDP.

B. Adversarial Learning

Adversarial learning, includes adversarial examples, adversarial attacks, adversarial training, etc. Initially adversarial learning was mostly used in the image domain, more gradient-based perturbation methods were proposed, such as FGSM [10], BIM [11], and JSMA [24]. They can perform effective attacks by changing the values of pixel points or fixing the nodes in the graph and then by adding or removing edges on the graph structure [11], [25], [26]. Since the discrete nature of the language space, adversarial learning applied to the NLP domain mostly uses word-level substitution or character-level flipping [27], [28] to generate the adversarial examples. The adversarial learning in the source code processing domain that we focus on is more similar to NLP. However, since source code has discrete nature along with strong structural nature, the source code, i.e., the programming language, is required to follow strict lexical, syntactic, and grammatical rules constraints. However, as source code has a strong structural nature along with its discrete nature. The source code, i.e. the programming

language, is required to follow strict lexical, syntactic and grammatical rules constraints.

In recent years, adversarial learning in the field of source code processing has also been gradually proposed, Bielik et al. [16] transforming the adversarial examples generation into an optimization problem and Yefet et al. [14] modified the model input by deriving the output distribution of the model relative to the model input, following the gradient. However, due to the structured and discrete nature of the code, the gradient-based and optimization-based approaches are more difficult to implement. Zhang et al. [12] proposed the Metropolis-Hastings Modifier Algorithm (MHM) and Jacob et al. [13] efficiently constructed the set of candidate identifiers by calculating L2 distance to generate adversarial examples by performing identifier substitution on code segments. These methods performed attacks only on code identifiers for substitution and dead code insertion and equivocal expressions, did not consider the importance of code structure features, and were implemented in a stochastic manner.

Ramakrishnan et al. [29] defined k-transformation robustness for source code tasks: adversarial attacks that allow the selection of transformations of length k for the input program by combining enumeration and gradient-based optimization for k transformations. Shashank et al. [30] proposed the use of optimized fuzzy processing to generate adversarial examples, similar to the approach proposed by Ramakrishnan et al. [29]. These two approaches still do not address the adversarial attack on the code structure information and cannot find an optimal set of the transformations. Quiring et al. [31] proposed to use the MCTS algorithm for adversarial to misleading the code authorship attribution model. However, it cannot complete the attack in a limited time when the code space is large enough.

The adversarial robustness of the DL models can be improved by erasing the non-robust features in the examples and retaining the robustness features [8], [12], we believe that such non-robustness features also exist in the source code structure information and the DL model's recognition and classification of source code functionality can be affected by changing the structure space of the source code. We propose the QMDP method. To address the inadequate consideration of code structure information in existing attack methods, QMDP combines the advantages of Monte Carlo and Dynamic Programming to successfully achieve an adversarial attack on the code structure within a limited iteration. To address the shortage of optimal attack operation combinations in existing methods, QMDP guides adversarial attacks by the significance of attack operations, providing a faster and more effective strategy for combining code transformations. In addition, robustness of the target classification models can be improved by adversarial training.

III. Structural Transformation Attack

QMDP is designed to perform a set of effective attacks on the structural information of a source code example and generate adversarial examples by changing the structural features. In this section, we will elaborate on all the structure transformation attack operations designed to implement QMDP. The attack operation defined in this paper specifies that it needs to satisfy the following constraints: (i) The generated new code segments need to satisfy the requirements of the grammar, syntax and coding rules of the programming language while completing the attacks; (ii) The new examples generated after the attack can be successfully compiled; (iii) The output of the new code segments before and after the attack remains unchanged with the same input. To ensure that the semantics of the program remains unchanged before and after the attack, the operations defined for structural conversion are divided into the following three levels: token-level transformations, statement- and expression-level transformations, and block-level transformations. There are 14 transformations in total.

A. Token-level Transformations

Such attacks perform adversarial attacks on the code at the token level, mainly by rewriting the declaration and initialization information of variables. There are 3 types in this group. For example, if multiple variables with the same type are declared simultaneously, they can be split into multiple lines of statements and declared separately. Fig. 1 shows that the declaration and initialization of the three variables "i, k, sum" of the int type are completed in the same statement. By parsing the xml file of the source code, we can get the type of each variable. First, we perform action ① to split one declaration statement into multiple statements, and define "j, k, sum" separately. Then, execute action ② to split the initialization and assignment of the variable "sum". With this simple attack, the content of the code declaration statement remains the same before and after the attack, but resolves the DL model's identification of the specific declaration format in the code file. Note that the types of declared variables should be consistent before and after the declaration statements are split.

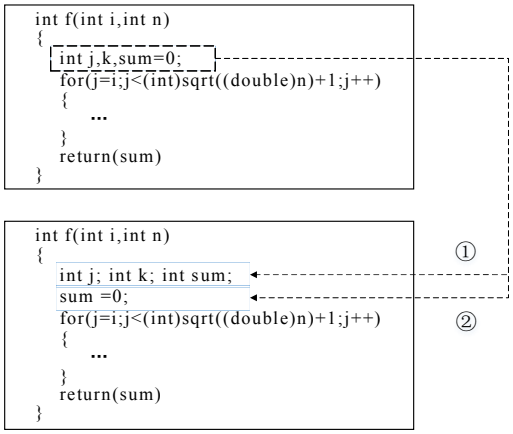

Fig. 1. An example of token-level transformations. ① We split the declaration statement when multiple variables are declared at the same time; ② We split the initialization and assignment when variable declaration and initialization are executed simultaneously.

The 3 transformation operations at this level are defined as follows:

- Declaration statement splitting: Splits the declaration statement into multiple statements when several variables with the same type are declared together in the same statement, and define the variables separately ($e.g.,\ int\ i, j; \rightarrow int\ i; int\ j;$).
- Initialization assignment splitting: Splits the variable initialization statement into a variable declaration statement and an assignment statement ($e.g.,\ int\ a\ =\ 5; \rightarrow int\ a; a = 5;$).
- Declaration initialization syncing: Performs variable declaration and initialization synchronously ($e.g.,\ int\ a; a = 5; \rightarrow int\ a = 5;$).

B. Statement- and Expression-level Transformations

This category of attacks is implemented at the statement and expression levels of program files, and there are six main types of such attacks. For example, the "$scanf$" input statement using the C program API can be modified to use the C++ program API "cin"; when multiple variables are assigned in the same statement, they can be split into multiple statements and assigned separately. As shown in Fig. 2, the variables "div" and "m" on the left side of the figure and the variables "$flag\,[i]$" and "$num1$" on the right side of the figure all complete the assignment in the same statement. We perform the multi-variable assignment splitting operation in this category, and use two separate assignment statements to replace the original statement. First, we determine the priority of the self-increment operation and the expression operation by checking whether the operator "++" is a prefix increment operator or a postfix increment operator. Such an attack adds small perturbations to the code at statement-level and expression-level to confuse the classifier's reliance on a particular statement- and expression-level coding format.

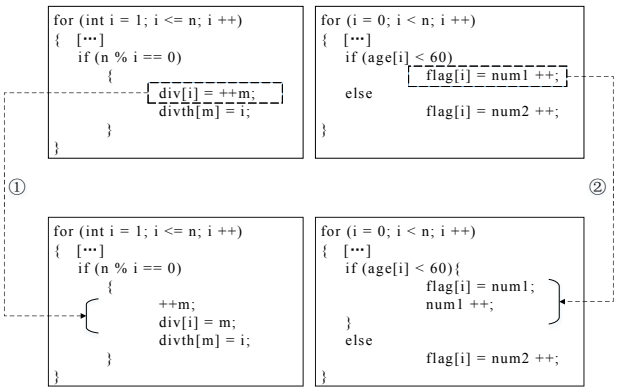

Fig. 2. Examples of statement- and expression-level transformations. ① operator "++" is a prefix increment operator, self-increasing first, followed by expression operation; ② operator "++" is a postfix increment operator, which executes expression operation first, followed by self-increment operation on the variable it is attached to.

The 6 transformation operations at this level are defined as follows:

- Multi-variable assignment splitting: Splits the assignment statement when multiple variables are assigned in the same statement($e.g.,\ temp = ++i; \rightarrow ++i; temp = i;$).
- Multi-variable assignment merge: Combines assignment statements for multiple variables into one ($e.g.,\ temp = i; i++; \rightarrow temp = i++;$).
- Input API transformation: Swaps the C API for reading input ($scanf$) with the C++ API (cin) ($e.g.,\ int\ a; scanf(\text{“}\%d\text{”}, \&n); \rightarrow int\ n; cin>>n;$).
- Output API transformation: Swaps the C API for reading output ($printf$) with the C++ API ($cout$) ($e.g.,\ int\,a; printf(\text{“}\%d\backslash n\text{”}, a); \rightarrow int\,a; cout<<n<<endl;$).
  hello
  `hello`
- Self-increasing/self-decreasing expressions unfolding: Writes self-increasing and self-subtracting expressions as expansions ($e.g.,\ i++; \rightarrow i+=1;$).
- Prefix and suffix operator swapping: Swaps the prefix and suffix operators in single operator expressions ($e.g.,\ i++; \rightarrow ++i;$).

C. Block-level Transformations

Such attacks are implemented by overwriting the block contents of the code file. There are 5 major types of implementation. For example, completing the interconversion of for statements and $while$ statements; replacing $switch\_case$ statements with if_else statements equivalently, etc. Although blocks of code for the same function can be written using different formats, it may affect the recognition of their functionalities by the DL model. Fig. 3 shows an example of using an if_else statement to rewrite a $switch\_case$ statement. The case condition in the $switch$ statement is scanned and replaced with the condition in multiple if statements. The contents of the execution in each $case$ statement are delivered to the corresponding if statement. In this attack, the implemented functionality of the code block is maintained before and after the attack, but the structural features of the block are made to change by rewriting the control statements.

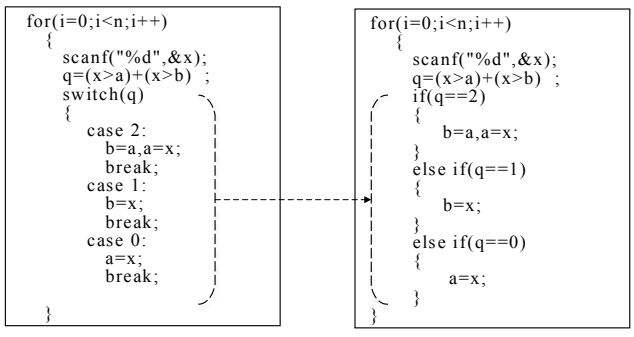

Fig. 3. An example of block-level transformations. if_else statement replaces $switch\_case$ statement equivalently.

The 5 transformation operations at this level are defined as follows:

- For/while statement transformation: Replaces the while statement with a semantically equivalent for statement or replaces the for statement with a semantically equivalent $while$ statement.
- If statement cascading: Combines multiple if statements of the same level using operators ($e.g.,\ \&\&, ||$) ($e.g.,\ if(a == 1)\{c = 2;\} if(b == 1)\{c = 2;\} \rightarrow if(a == 1||b == 1)\{c = 2;\}$).
- If statement splitting: Splits if statements that contain operators ($e.g.,\ \&\&, ||$) in the conditional statement ($e.g.,\ if(a == 1\&\&b == 2)\{c = 2;\} \rightarrow if(a == 1)\{if(b == 2)\}\{c = 2;\}$).
- Switch/if transformation: Replaces the $switch\_case$ statement with a semantically equivalent if_else statement, or replaces the if_else statement with a semantically equivalent $switch\_case$ statement (the example can be seen in Fig. 3).
- Ternary/if transformation: Replaces the wrapped ternary operator expressions with equivalent if statements ($e.g.,\ b = a > b?a : b; \rightarrow if(a > b)\{b = a;\}else\{b = b;\}$).

IV. Design of QMDP

In this section, we first give an overview and workflow of the proposed method QMDP, and then elaborate on each key step.

A. Overview

Considering the discrete and structured nature of source code, QMDP takes the adversarial example generation problem as a referenced sampling process. Existing adversarial examples generation methods in the field of source code classification are mainly implemented by performing source code identifier replacement or dead code insertion. However, research results [1], [17] demonstrate that more semantic information is contained in the source code structure compared to the identifiers, which is why QMDP is mainly implemented to generate adversarial examples for structural features of the source code. Since the results of executing different categories of structural transformation attacks at the same location of the source code or the same category of structural transformation attacks at different locations are not unique, QMDP converts the problem of combining transformations in the code space into the problem of combining structural transformation attack operations.

QMDP is based on a set of interaction objects, i.e., a target classification model M and a pair of examples ($\langle x, y\rangle \in D$) in dataset D that are correctly classified by M. Here x denotes the source code and y is the correct functional label corresponding to x. The adversarial examples ($\langle x', y\rangle \in D$) are generated by executing attacks strategically on the source code. The workflow of QMDP is shown in Fig. 4, which is theoretically based on Markov Chain. Therefore, QMDP can also be considered as a Markov model considering actions. It consists of six elements: State, Action, Policy, Reward, Decay factor, and Action value table. Q-leaning algorithm which is a non-model-based method for solving the reinforcement problem is used to solve the control problem for the whole experiment. The major advantage is the use of the Temporal-Difference (TD) Learning, which incorporates the strengths of Monte Carlo and dynamic programming, allowing for off-policy TD control to solve the optimal policy for the entire decision process. Inspired by Markov Chain, the process of a single iteration of QMDP can be organized into:

- Step 1: Executing code structural transformations. Strategically select out candidate transformation actions to execute adversarial attacks against DL models inputs;
- Step 2: Attack testing. Testing the success of adversarial attacks, i.e., testing whether the DL models can be successfully fooled;
- Step 3: Q-Learning phase. Learning the values corresponding to each transformation attack action and updating the action value table Q to guide the iterative process of adversarial attacks.

B. Method Elements

State (S): In the proposed method QMDP, S is denoted as the set of all states with respect to an example $x \in D$. Each element S_t in the set S is used to represent a new example $x^{'}$ generated after a certain transformation of the example x at moment t. Markov property is followed between state transfers, which means that the current state is only associated with the state and action at the previous moment and the states and actions at other moments are mutually independent;

Action ($Action$): $Action$ is a set used to store the structure transformation attack operations defined in this paper, as described in Section III. Three levels of tokens, code blocks, statements and expressions are defined, with a total of 14 types;

Policy (π): The policy π is the method of selecting the next action to be executed, based on the action performed and the state returned by the previous state;

Reward (R): The reward R is the reward value obtained by the current state after the execution of each attack action. In this paper, we only need to acquire the probability value of the classifier target label. The reward value can be acquired by calculating the change in the probability of the target label;

Decay factor (γ): γ indicates the discount of reward value over time;

Action value Table (Q): Q is a state-action value table during the iterative attack. It is used to store the value of each action under each state in the set of states S. Each value in Q is associated with the action that has been executed in the current state. Since the experimental process limits the number of iterations, the states stored in the state set S of Q are mutually independent and limited in number.

C. Executing code structural transformations

This step is completed in three parts, i.e. extracting actions, selecting candidate actions and executing actions.

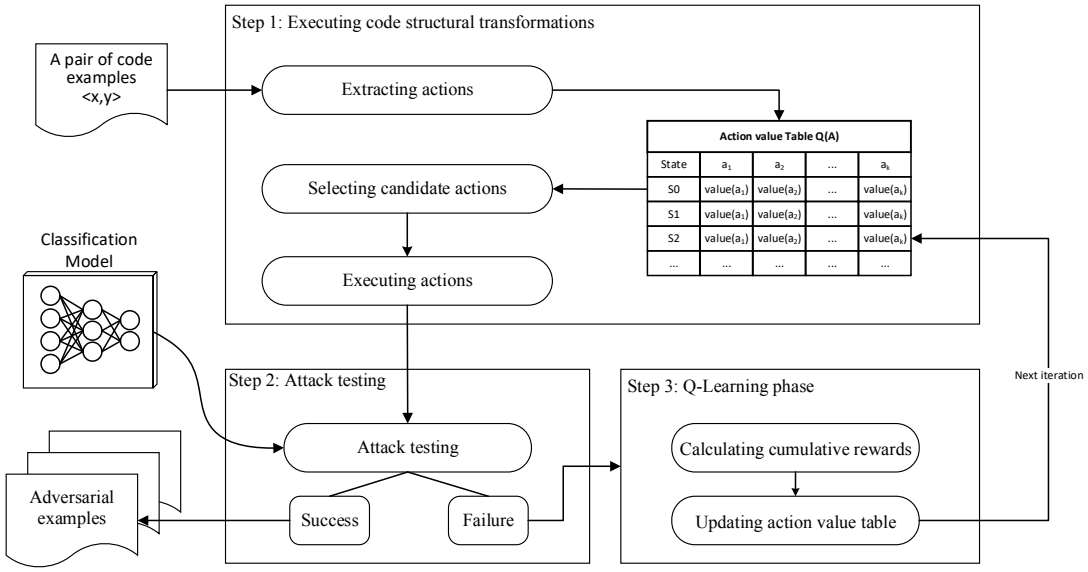

Fig. 4. Workflow of QMDP. The input is a pair of labeled example pairs and a classification model, and the outputs are the adversarial examples generated by QMDP after iterative attacks corresponding to the input.

First, we extract the actions that can be performed with the adversarial transformation based on the input set of examples $\langle x, y\rangle$. Marking the initial state of example x as S_0. Firstly, we extract the code structure features and obtain the executable attack action sequence $A\,(A \subseteq Action)$ corresponding to x, which limits the executable actions of the current sample during the whole iteration. Then build the action value table Q corresponding to A, which we denote by $Q(A)$. Each item in the "$State$" of $Q(A)$ represents a code space combination, which is a unique example state identified by a transformation operation and the code position corresponding to the execution of the operation. Meanwhile, using $value$ to record the value corresponding to each executable action $a_i\,(a_i \in A, 1 \leq i \leq 14)$ in each state.

Second, we select the candidate actions to be executed from the executable transformation sequence A extracted in the last Step 3. Referring to the action value table $Q(A)$, we strategically select the actions used to execute the attack on the examples. Since different types of actions are executed at the same position of the code or the same action is executed at different positions, the new code snippets generated after the attack will not be unique. To address this concern, QMDP converts the code-space combination problem into an action combination problem. The candidate actions are selected by the policy $\pi(x)$, and a random number $rand$ between 0 and 1 will be generated when each selection is made (see Algorithm 1, lines 5-9). The policy $\pi(x)$ is shown in (1).

$$\pi\left(x\right) = \begin{cases} random\left(a\right) & ,if\ \ rand < \epsilon \\ \arg\max_a Q\left(x\right) & ,if\ \ rand \geq \epsilon \end{cases} \quad , \qquad (1)$$

where ϵ is the threshold value set. For instance, setting ϵ to 0.6, if the random number $rand < 0.6$, we randomly select an operation from the current sequence A as the candidate action; if $rand \geq 0.6$, we select the operation corresponding to the maximum value from $Q(A)$ to be the candidate.

Algorithm 1 *Selecting candidate actions*

Input: Data pair $\langle x, y\rangle \in D$;
Max iteration m;
Executable action set A corresponding to $\langle x, y\rangle$;
Threshold parameter ϵ;
Action value table Q.
Output: A candidate action a_t.
1: Initialize $S_0 \leftarrow x$;
2: Initialize $t \leftarrow 1$;
3: Initialize $a_1, a_2, ..., a_m \leftarrow \{None, None, \ldots, None\}$;
4: **while** $t \leq m$ **do**
5: $A \leftarrow \{a | a\ is\ in\ A \wedge a\ is\ in\ Action\}$;
6: **if** $rand() < \epsilon$ or $Q(A)$ all is 0 **then**
7: $a_t \leftarrow random(a)$;
8: **else**
9: $a_t \leftarrow argmax_a Q(A)$;
10: **end if**
11: **return** a_t;
12: **end while**

Third, we apply the candidate action a selected in the previous step to x. We choose a random location in x where a can be executed. We can obtain a new state S by a and the chosen code location. Next, we determine whether S already exists by checking $Q(A)$, if it does then we just need to update the value corresponding to each action in S by the subsequent steps. Instead, we store S into $Q(A)$ and calculate the value corresponding to each operation in S by subsequent steps.

Algorithm 2 *Attack testing*

Input: Data pair $\langle x, y\rangle \in D$;
Classification model M;
Max iteration m.
Output: An adversarial examples set $E(x)$ on $\langle x, y\rangle$.

```
Initialize t ← 1;
Initialize E(x) is null;
Initialize n ← 0;
while t ≤ m do
    x_t ← x';
    if argmaxM(x') ≠ argmaxM(x) then
        add x_t to E(x);
        n ← n + 1;
        break;
    end if
end while
return E(x);
```

D. Attack Testing

In this step, the selected action is executed to perform an adversarial attack on x, generating a new example x'. Feed x' into the classification model M to verify whether x' is an adversarial example. If the classification model M misclassifies x', the attack is successful and ends the current iteration. Add x' to the current set of adversarial examples $E(x)$ (see Algorithm 2, lines 5-9). Otherwise, continue the iteration. The adversarial examples set $E(x)$ is defined as:

$$E(x) = \{x' | x' = x + \Delta x \wedge output(x) = output(x') \wedge \arg\max(M(x)) = \arg\max y \neq \arg\max(M(x'))\} \quad , \tag{2}$$

where Δx is a set of legal perturbations to the source code that satisfy the program coding rules after executing a set of adversarial operations. $output(x)$ represents the output of the executable example x after execution and $argmaxM(x)$ denotes the result of the model M for the classification of x.

E. Q-Learning phase

This step has two components. First, we calculate the cumulative reward G of the current example state from the immediate reward R_t which the example state changes. Second, we update the states sequences and actions values in the $Q(A)$ by R_t.

First, calculating cumulative rewards. After an attack test is executed in Step 2, it will generate an immediate reward value R_t. R_t is calculated as follows:

$$\begin{aligned} R_t &= prob(x_{t-1}, y) - prob(x_t, y) \\ &= P[M_y(x_{t-1})] - P[M_y(x_t)] \end{aligned} \tag{3}$$

With regard to a set of data $\langle x, y\rangle$, $P[M_y(x_t)]$ is denoted as the probability that the example x_t in the state at moment t is successfully classified as y by the classifier M.

Next, we determine whether the state S_t of the sample x_t at moment t needs to be stored into $Q(A)$ by R_t and $Q(A)$ (see Algorithm 3, lines 6-8). The cumulative reward value G of the current state, i.e., the example cumulative reward value $R(S_t)$ at the time of execution to state S_t, is calculated as follows(see Algorithm 3, lines 5-9):

$$G = R(S_t) = R(S_{t-1}) + \gamma^t R_t \quad , \tag{4}$$

Second, updating action value table $Q(A)$. By calculating the immediate reward value R_t, we can obtain the action cumulative reward value $Q(a_t)$ (see Algorithm 3, lines 11-12) at time t. QMDP continuously updates the action value table $Q(a_t)$ and searches by action importance to guide the adversarial example generation by obtaining the maximum $R(S_t)$. a_t is the action performed by the example from time $t-1$ to t. The action cumulative reward value $Q(a_t)$ can be calculated as follows:

$$Q(a_t) = \frac{Q(a_t) * count(a_t) + R_t}{count(a_t) + 1} \quad , \tag{5}$$

where γ^t denotes the discount to the reward value over time. As the iteration grows, the more discounted the contribution to the reward value becomes. $count(a_t)$ is the times that the action a_t was executed from the original state S_0 to the time t.

Algorithm 3 *Q-learning*

Input: Data pair $\langle x, y\rangle \in D$;
Classification model M;
Max iteration m;
Action value table Q;
Decay factor γ.
Output: Updated action value table Q.

```
1:  Initialize S_0 ← x, t ←1, count ←0;
2:  Initialize Cumulative reward G ← 0;
3:  while t ≤ m do
4:      x_t ← x';
5:      R_t ← P[M_y(S_count)] − P[M_y(x_t)];
6:      if R_t > 0 and x_t not in Q then
7:          count ← count + 1;
8:          S_count ← x_t;
9:          Calculate Cumulative reward G ← R(S_count);
10:     end if
11:     Calculate action a_t cumulative reward value Q(a_t);
12:     count(a_t) ← count(a_t) + 1;
13: end while
14: return Q;
```

V. EXPERIMENTS & RESULTS

In this section, we will perform an in-depth evaluation to verify the effectiveness of our approach. The experimental configuration will be described in detail, as well as the results of the QMDP adversarial attack and the adversarial training will be evaluated. Our experiments mainly revolve around the following three Research Questions (RQs):

RQ1: What is the performance of using QMDP to implement adversarial attacks?

TABLE I
HYPER-PARAMETERS OF THE CLASSIFICATION MODEL

Hyper-parameters	LSTM	ASTNN
Vocabulary size	5000	8878
Hidden size	600	100
Embedding size	512	128
Layers	2	1
Dropout	0.5	0.2
Batch size	32	64
Optimizer	Adam	AdaMax

RQ2: Can the robustness of classification models be improved by performing adversarial training using the adversarial examples generated by QMDP?

RQ3: What about the attack effect that can be achieved by each of the three levels of attack operations defined in this paper?

A. DataSet

We choose Open Judge (OJ), an open benchmark dataset proposed by Mou et al. [1] for source code classification, as the dataset for our experiments. OJ is collected from an open judge platform and consists of 52,000 C/C++ compilable and correctly executable code files with problem labels. There are 104 categories of functionality labels, each containing 500 code files. We used the srcML tool to construct our transformations in the compiler frontend and the pycparser tool to verify the validity the samples we generated. srcML is able to describe programs developed by different languages and development environments in a uniform and extensible way, which can easily parse and recover C/C++ programs in our experiments.

We first use the srcML tool to pre-process the files, parsing the C/C++ source program into a single XML file. Then we can easily wrap the dataset using Pandas. The packaged data consists of three items, which are represented by the column indexes "code", "xml" and "label". The code stores the source code file, the xml stores the xml file generated by parsing the source code, and the label stores the corresponding function label. Lastly, we randomly divide the processed dataset into a training set and a test set according to the ratio of <4:1>. In the training process, 20% of the training set is randomly selected as a validation set. The training set and test set are kept constant throughout the experiment.

B. Classification Model

In our experiments, we choose to use an AST-based classifier ASTNN and a token-based classifier LSTM. The former we use is the most classical tree-based classifier ASTNN [2] applied to the OJ dataset, which captures more syntactic information during training. It uses the ASTs generated by the pycparser tool as inputs. We follow the same hyper-parameters as the original model. The latter LSTM is one of the most representative models in sequence processing which is widely used in the field of source code processing. The input is a tokenized sequence of source code snippets. We reuse the classifier LSTM applied to the OJ dataset by Zhang et al. [12] and keep the parameters consistent with it. Table I shows the hyper-parameter information of the model configuration.

C. Evaluation Metrics

In order to better demonstrate the experimental results of the proposed method in this paper, we measure the effectiveness of QMDP attacks using "ASR", "Valid" and "Succ", and the performance of target classification models before and after adversarial training is measured by "ACC" and "CE".

- Attack success rate (**ASR**): In the experiments, ASR is used to represent the success rate of using QMDP to attack the DL modelS such that the models misclassifie the originally correctly classified examples.

$$ASR = \frac{\sum_{i}^{n} \{\exists\, x_i' \mid x_i' = x_i + \Delta x, y_i' \neq y_i\}}{\sum_{i}^{n} \{x_i \mid < x_i, y_i > \in D\}} \quad , \tag{6}$$

  where n is the total number of examples tested and Δx is the set of perturbations added to the examples during the adversarial attacks.
- Validity (**Valid**): QMDP specifies that the new code snippets generated after the attack can be successfully compiled with the syntax and semantics remaining intact. Valid represents the probability that the generated adversarial example is valid.
- Success rate (**Succ**): This term is used to statistically measure the probability that the attack is successful while the adversarial examples are valid, obtained by calculating the product of ASR and Vaild.
- Accuracy (**Acc**): Since the classification model used for the experiments in this paper is a multi-classification model, we used the ACC to indicate the percentage of correct classifications in the test set.
- CrossEntropy Loss (**CE**): In our experiments, we use CrossEntropy as the loss function of the neural network. The calculation equation is as follows:

$$CE = -\frac{1}{n} \sum\nolimits_{i} \sum\nolimits_{c=1}^{m} y_{ic} \log(p_{ic}) \quad , \tag{7}$$

  where m is the number of category labels; y_{ic} is the sign function (0 or 1), which takes 1 when the true label of example i is equal to c and 0 otherwise; P_{ic} is the probability that test example i belongs to category c; n is the number of examples.

D. Experiment on the effectiveness of QMDP attack (RQ1)

To validate the effectiveness of the QMDP attack on the code structure information, we extracted 1,000 correctly classified examples from the test set and mainly performed the attack on these 1000 examples. QMDP is implemented on the xml file generated after example x has been processed using srcML. By parsing the xml file first, the structural features of the code can be extracted. The sequences of attack actions

Original example	Adversarial example

int main (int argc , char * argv [ ])
{ char a [6] ;
char b [6] ;
int i ;
int j ;
scanf ("%s" , &a) ;
for (i = 0 ; (* (a + i)) != '\0' ; i ++)
{
* (b + i) = * (a + i) ;
}
for (j = i - 1 ; j >= 0 ; j --)
{
printf ("%c" , b [j]) ;
}
return 0 ;
}
	int main (int argc , char * argv [ ])
	{ char a [6] ;
	char b [6] ;
	int i ;
	int j ;
	cin>>a ;
	for (i = 0 ; (* (a + i)) != '\0' ; i ++)
	{
	* (b + i) = * (a + i) ;
	}
	for (j = i - 1 ; j >= 0 ; j --)
	{
	printf ("%c" , b [j]) ;
	}
	return 0 ;
	}

Fig. 5. An adversarial example generated by QMDP.

that QMDP can execute during the whole iteration of x can be obtained, and they are used to execute the adversarial attacks on x. Also control the number of iterations in the experiment to 30.

In our experiments, we take the Metropolis-Hastings Modifier (MHM) algorithm proposed by Zhang et al. [12] as a baseline. The MHM algorithm is a state-of-the-art attack method applied to the OJ dataset in the same task scenario as the scheme proposed in this paper. It is used to generate adversarial examples based on a Metropolis-Hastings algorithm to execute attacks on source code. More intuitively, MHM iteratively renames identifiers using random sampling in a black-box scenario. Unlike QMDP, MHM performs the renaming attack only at the identifier level, with no consideration of the structural information of the code. Furthermore, a completely random approach is used to iterate, which is simpler to operate, but has a certain degree of waste of time and resources.

The experimental results are shown in Table II, where we compare the QMDP attack on code structure information with the MHM using the identifier renaming method, it can be seen that the adversarial examples generated by QMDP can be guaranteed to be 100% valid. Since the training set and test set are randomly divided, and the 1,000 correctly classified attacked examples used in the attack process are also randomly drawn from the test set. The experimental results in Table II are the averages obtained from multiple training repetitions in our experimental setting.

According to the experimental results, we found that QMDP has a higher success rate than MHM when performing an adversarial attack on the LSTM model. It is because that the LSTM is a token sequence based classification model, and its input is the code tokenization sequence. When QMDP is used to attack the example, it can maximize the code tokenization sequence modification while minimizing the change of code structure features, thus misleading the LSTM. Since most of the identifiers in the OJ dataset are named randomly and meaninglessly, such as i, j, k. Thus, the effectiveness of the attack achieved by renaming only the identifiers within a limited number of iterations is unsatisfactory. However, ASTNN is a DL model trained based on AST. Its aim in the training process is that more syntactic information can be obtained. Even though QMDP is slightly less effective than MHM when performing attacks on ASTNN, as we can see in Table III, the attack success rate of the model after using the adversarial examples generated by QMDP to participate in adversarial training can be reduced by 7.4% more than that of using MHM. The small loss in attack success rate in exchange for the higher adversarial robustness of the DL models is extremely worthwhile.

A simple attack example is shown in Fig. 5. It merely replaces the $scanf$ input API in the code segment with cin. The content executed by the code remains the same before and after the attack, but it causes the classifier to produce wrong results. The result shows that the structural information in the source code is non-robust. QMDP implements an adversarial attack on the structural information of the source code, breaking the association between the functionality label and the coding format specific to the code. Resulting in misclassification by the classifier.

Insight 1. *The experimental results show that the structural features of source code are non-robust. By breaking the association between the code specific writing format and the original functionality label, the DL models can be misled to misclassify the code to another label.*

E. Experiment on the effect of the models' performance (RQ2)

In order to verify whether adversarial training contributes to the adversarial robustness of the attacked DL model, we reduce the dependence of the classification model on non-robust features (F_{nr}) by adversarial training of the DL models using the method of increasing the training set. We mix the adversarial examples generated using the QMDP method into the original training set to retrain the model. And ensure that the hyper-parameters of the new model are consistent with the original model.

Specifically, we randomly select 2,000 examples from the generated adversarial examples and mix them into the original training set as the enhanced training set for the new model, and the test set remains unchanged. After that, using the updated dataset to retrain the classification model. Finally, we tested the new model on the test set and performed the adversarial attack again with QMDP. The results are shown in Table III. In Table III, it can be seen that both LSTM and ASTNN gain the ability to resist QMDP by adversarial training, and the performance of the models also improves. The attack success rates of QMDP against trained LSTM and ASTNN models were reduced by 44.0% and 44.8%, respectively. Meanwhile, the performance of the two models increased by 4.9% and 0.3%, respectively.

The experimental results indicate that using QMDP to attack on source code structure information and using the same amount of adversarial examples as MHM for adversarial training of the models, the adversarial robustness of the new models can both be improved by about 10.0%, and the

TABLE II
RESULTS OF THE ADVERSARIAL ATTACK

Method	Model	ASR (%)	Valid (%)	Succ (%)
MHM	LSTM	71.6	100	71.6
MHM	ASTNN	89.8	100	89.8
QMDP	LSTM	89.3	100	89.3
QMDP	ASTNN	87.9	100	87.9

TABLE III
RESULTS OF ADVERSARIAL TRAINING WITH QMDP

Model	ACC (%)	ASR (%)	CE (%)
LSTM (MHM)	93.0	71.6	0.210
LSTM+MHM	94.2 (+1.2)	46.4 (-25.2)	0.190
LSTM (QMDP)	93.0	89.3	0.210
LSTM+QMDP	97.9 (+4.9)	45.3 (-44.0)	0.080
ASTNN (MHM)	98.1	89.3	0.061
ASTNN+MHM	98.0 (-0.1)	51.9 (-37.4)	0.057
ASTNN (QMDP)	98.1	87.9	0.061
ASTNN+QMDP	98.4 (+0.3)	43.1 (-44.8)	0.056

TABLE IV
THE IMPACT OF ATTACK OPERATIONS ON THE SUCCESS OF THE ADVERSARIAL ATTACK

Model	Attack category	ASR (%)	Ratio (%)
LSTM	Token-level	54.3	24.1
	Statement - and Expression-level	70.0	27.7
	Block-level	67.2	48.2
	Total	89.3	—
ASTNN	Token-level	47.5	20.0
	Statement - and Expression-level	59.3	36.0
	Block-level	73.0	44.0
	Total	87.9	—

classification performance of the new models obtained is also enhanced.

Insight 2. *Using the adversarial examples generated by the QMDP method to participate in adversarial training can improve the adversarial robustness of DL models to a greater extent than the state-of-the-art attack methods.*

F. Experiment on the effects of split level attacks (RQ3)

For obtaining the attack effect that can be achieved by each of the three levels of QMDP adversarial operation, we perform the adversarial attacks on the LSTM and ASTNN before adversarial training using these three levels of attack operations in turn. The results of the attacks are shown in Table IV. The "Ratio" represents the percentage contribution of each of the three levels of attack operations to the success of the attack when all attack operations are executed together. The iteration is still set to 30 during the experiment.

The results in Table IV show that "Total" refers to the three levels of attack operations executed together. It can be found that the ASR of all three levels of the transformation attack operation when executed separately is lower than the ASR of the "Total" state. Since code statements are interconnected and interdependent, when attacking the code structure by using QMDP to combine transformations on the code, the multi-level functional dependencies in the code structure can be disrupted and code perturbations can be minimized to maximize the interference and deception to the DL model.

Insight 3. *Using the three levels of attack operations defined in this paper to jointly modify the code structure information, we can maximize the possibility of misleading the DL classification model.*

As described in this section, code structure features are potentially useful but non-robust features for classification model classification. Therefore, when the code structure is re-encoded, the association between the encoding format and the functionality labels is broken. By mixing the adversarial examples into the training set for adversarial training, the dependence of model classification on F_{nr} can be reduced, so that the adversarial robustness can be obtained.

VI. CONCLUSION

In this paper, we discover and verify the non-robustness of source code structural features. For the discrete and structured nature of the source code, we propose QMDP, which iteratively performs three levels of adversarial attacks on the source code through a Q-learning-based Markov decision process. It can correctly and efficiently attack the source code to generate adversarial examples. The generated adversarial examples still satisfy the constraints of source code syntax, semantics and encoding rules. Moreover, the output of the adversarial examples can be consistent with the original sample with fixed input before and after the adversarial attack. We experimentally demonstrate that the classification performance and adversarial robustness of the classification model can be improved after

adversarial training using the adversarial examples generated by QMDP.

The approach in this paper also has some limitations. First, the proposed method of adversarial attacks on source code structure information in this paper is a bit complicated compared to the general application in identifier replacement methods. Nevertheless, the code can be easily parsed and refactored using the srcML tool, and the conversion time between the C/C++ program and the xml file can be essentially negligible. Second, since the dataset OJ used for the experiments in this paper is a collection of executable files collected from the competition. Compared to the real-world software programs, the number of the code lines is much smaller. But the experimental validation in this paper can conclude that performing adversarial training with adversarial examples generated by the QMDP method is feasible and effective for improving the adversarial robustness of classification models.

In future work, we expect to apply QMDP to more real software programs. QMDP is an adversarial examples generation method based on a black-box environment, which is simple to operate and has high portability. It has been experimentally proven that QMDP is effective for DL models to generate adversarial examples. We can try to adapt QMDP to other source code tasks based on DL, such as vulnerability detection, function name prediction and code clone detection, it is believed to be interesting as well.

ACKNOWLEDGMENT

This paper is supported by the National Natural Science Foundation of China under Grant No. 61802106 and the Natural Science Foundation of Hebei Province under Grant No. F2020201016.

REFERENCES

[1] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the 30th AAAI Conference on Artificial Intelligence, February 12-17, Phoenix, Arizona, USA*, 2016, pp. 1287–1293.

[2] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering, Montreal, QC, Canada, May 25-31*, 2019, pp. 783–794.

[3] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence, Melbourne, Australia, August 19-25*, 2017, pp. 3034–3040.

[4] S. Herbold, "Comments on scottknottesd in response to 'an empirical comparison of model validation techniques for defect prediction models'," *IEEE Trans. Software Eng.*, vol. 43, no. 11, pp. 1091–1094, 2017.

[5] J. Lin and L. Lu, "Semantic feature learning via dual sequences for defect prediction," *IEEE Access*, vol. 9, pp. 13 112–13 124, 2021.

[6] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium, San Diego, California, USA, February 18-21*, 2018.

[7] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "VulSniper: Focus your attention to shoot fine-grained vulnerabilities," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence, Macao, China, August 10-16*, 2019, pp. 4665–4671.

[8] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *Proceedings of the 6th International Conference on Learning Representations, Vancouver, BC, Canada, April 30 - May 3*, 2018.

[9] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: An automatic categorization system for open source repositories," *J. Syst. Softw.*, vol. 79, no. 7, pp. 939–953, 2006.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *Proceedings of the 3rd International Conference on Learning Representations, San Diego, CA, USA, May 7-9*, 2015.

[11] T. Miyato, A. M. Dai, and I. J. Goodfellow, "Adversarial training methods for semi-supervised text classification," in *Proceedings of the 5th International Conference on Learning Representations, Toulon, France, April 24-26*, 2017.

[12] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proceedings of the 34th AAAI Conference on Artificial Intelligence, New York, NY, USA, February 7-12*, 2020, pp. 1169–1176.

[13] J. M. Springer, B. M. Reinstadler, and U. O'Reilly, "STRATA: Building robustness with a simple method for generating black-box adversarial attacks for models of code," *CoRR*, vol. abs/2009.13562, 2020.

[14] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 162:1–162:30, 2020.

[15] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. C. Gall, "Adversarial robustness of deep code comment generation," *CoRR*, vol. abs/2108.00213, 2021.

[16] P. Bielik and M. T. Vechev, "Adversarial robustness for code," in *Proceedings of the 37th International Conference on Machine Learning, 13-18 July, Virtual Event*, 2020, pp. 896–907.

[17] J. F. Pane, C. A. Ratanamahatana, and B. A. Myers, "Studying the language and structure in non-programmers' solutions to programming problems," *Int. J. Hum. Comput. Stud.*, vol. 54, no. 2, pp. 237–264, 2001.

[18] N. D. Q. Bui, Y. Yu, and L. Jiang, "TreeCaps: Tree-based capsule networks for source code processing," in *Proceedings of the 35th AAAI Conference on Artificial Intelligence, Virtual Event, February 2-9*, 2021, pp. 30–38.

[19] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[20] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, May 14-22*, 2016, pp. 1157–1168.

[21] E. O. Kiyak, A. B. Cengiz, K. U. Birant, and D. Birant, "Comparison of image-based and text-based source code classification using deep learning," *SN Comput. Sci.*, vol. 1, no. 5, p. 266, 2020.

[22] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning, New York City, NY, USA, June 19-24*, 2016, pp. 2091–2100.

[23] W. Zaremba and I. Sutskever, "Learning to execute," *CoRR*, vol. abs/1410.4615, 2014.

[24] J. Gao, J. Lanchantin, M. L. Soffa, and Y. Qi, "Black-box generation of adversarial text sequences to evade deep learning classifiers," in *Proceedings of the IEEE Security and Privacy Workshops, San Francisco, CA, USA, May 24*, 2018, pp. 50–56.

[25] H. Dai, H. Li, T. Tian, X. Huang, L. Wang, J. Zhu, and L. Song, "Adversarial attack on graph structured data," in *Proceedings of the 35th International Conference on Machine Learning, Stockholmsmässan, Stockholm, Sweden, July 10-15*, 2018, pp. 1123–1132.

[26] D. Zügner and S. Günnemann, "Adversarial attacks on graph neural networks via meta learning," in *Proceedings of the 7th International Conference on Learning Representations, New Orleans, LA, USA, May 6-9*, 2019.

[27] M. Alzantot, Y. Sharma, A. Elgohary, B. Ho, M. B. Srivastava, and K. Chang, "Generating natural language adversarial examples," in *the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4*, 2018, pp. 2890–2896.

[28] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "HotFlip: White-box adversarial examples for text classification," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, Melbourne, Australia, July 15-20, Volume 2: Short Papers*. Association for Computational Linguistics, 2018, pp. 31–36.

[29] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. W. Reps, "Semantic robustness of models of source code," *CoRR*, vol. abs/2002.03043, 2020.

[30] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," in *Proceedings of the 9th International Conference on Learning Representations, Virtual Event, Austria, May 3-7*, 2021.

[31] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *Proceedings of the 28th USENIX Security Symposium, Santa Clara, CA, USA, August 14-16* 2019, pp. 479–496.

,