Generating Adversarial Source Programs Using Important Tokens-based Structural Transformations

Penglong Chen, Zhen Li*, Yu Wen, and Lili Liu
School of Cyber Security and Computer, Hebei University, Baoding, China
Email: sc2014612@gmail.com, lizhenhbu@gmail.com, wenyu@mail.hbu.edu.cn, liulili2847@gmail.com

***Abstract*—Deep learning models have been widely used in source code processing tasks, such as code captioning, code summarization, code completion, and code classification. Recent studies have shown that deep learning-based source code processing models are vulnerable. Attackers can generate adversarial examples by adding perturbations to source programs. Existing attack methods perturb a source program by renaming one or multiple variables in the program. These attack methods do not take into account the perturbation of the equivalent structural transformations of the source code. We propose a set of program transformations involving identifier renaming and structural transformations, which can ensure that the perturbed program retains the original semantics but can fool the source code processing model to change the original prediction result. We propose a novel method of applying semantics-preserving structural transformations to attack the source program processing model in the white-box setting. This is the first time that semantics-preserving structural transformations are applied to generate adversarial examples of source code processing models. We first find the important tokens in the program by calculating the contribution values of each part of the program, then select the best transformation for each important token to generate semantic adversarial examples. The experimental results show that the attack success rate of our attack method can improve 8.29% on average compared with the state-of-the-art attack method; adversarial training using the adversarial examples generated by our attack method can reduce the attack success rates of source code processing models by 21.79% on average.**

***Index Terms*—software security, source code, adversarial examples, deep learning, program transformation**

I. Introduction

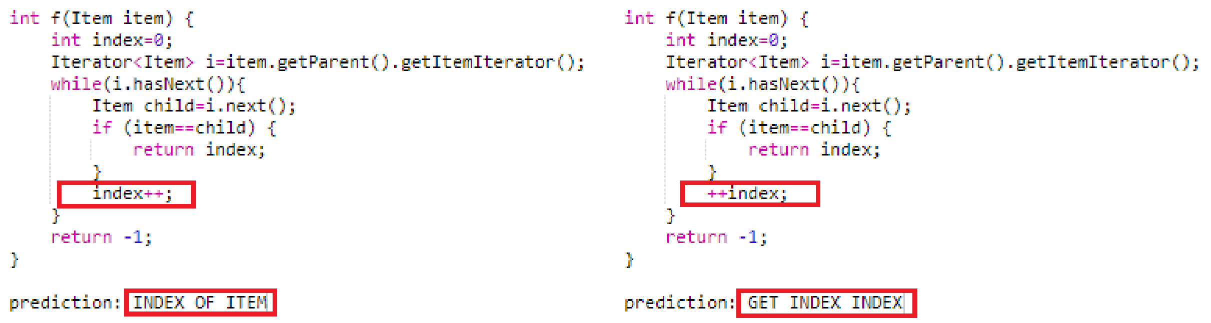
Deep learning-based source code processing tasks have attracted much attention [1], such as code captioning [2], code summarization [3], [4], code completion [5], code classification [2], [6], and vulnerability detection [7], [8]. They usually collect a large number of source code programs from websites (e.g., GitHub) and express the programs in different forms, which are fed to deep neural networks to complete various source code processing tasks.

Recent studies have shown that deep neural networks can be easily fooled by adversarial examples [9] for source code processing tasks. Source code processing models can be attacked under the black-box [10], [11] or white-box [12]–[15] settings. In the black-box setting, Zhang et al. [10] and Springer et al. [11] generated adversarial examples by renaming the variables in the program. In the white-box setting, Yefet et al. [12] generated adversarial examples with the same semantics as the source code by renaming identifiers and inserting dead codes. Ramakrishnan et al. [14] and Sriant et al. [15] expanded the perturbation method by six equivalent transformations. These existing studies generate adversarial examples by replacing or inserting new identifier names into the program. However, the perturbation methods they use fail to consider the equivalent structural transformations. The equivalent structural transformation refers to obtaining a program with the same semantics as the original program by equivalently transforming the structure of the program. Existing source code processing models are not robust to the program generated after the equivalent structural transformation [16], indicating that the equivalent structural transformation can introduce perturbations to the program. An adversarial example of a seq2seq model [15] is shown in Fig. 1. By replacing "index++" in the source code with "++index", the prediction of the seq2seq model is changed from "INDEX OF ITEM" to "GET INDEX INDEX". Therefore, equivalent structural transformations should be involved in perturbations.

Adversarial examples in the source code domain should have the same semantics as the original program. This requires that the perturbation we impose needs to satisfy the syntax and semantics of source code, which is more stringent than the constraints in the image and natural language processing domains. For this reason, we should consider the following three factors when choosing perturbations: *preserving semantics*, *minimal code changes*, and *human-readability*. In other words, the perturbed program should be semantically consistent with the original sample, and the code changes are required to be as small as possible. In addition, the added perturbation should not generate source code involving code obfuscation which is hard to read. Under these constraints, we propose a set of program transformations (see Section III for details).

After determining a set of program transformations containing structural transformations, we need to apply the program transformations to the original program. Yefet et al. [12], Ramakrishnan et al. [14], and Bielik et al. [13] only proposed transformation sets but did not involve how to apply these transformations. Srikant et al. [15] proposed an optimization algorithm for code perturbation location and perturbation selection principle. These works only involve the identifier transformation, and cannot locate the structural transformations.

*Zhen Li is the corresponding author.



```
int f(Item item) {
    int index=0;
    Iterator<Item> i=item.getParent().getItemIterator();
    while(i.hasNext()){
        Item child=i.next();
        if (item==child) {
            return index;
        }
        index++;
    }
    return -1;
}

prediction: INDEX OF ITEM
```

```
int f(Item item) {
    int index=0;
    Iterator<Item> i=item.getParent().getItemIterator();
    while(i.hasNext()){
        Item child=i.next();
        if (item==child) {
            return index;
        }
        ++index;
    }
    return -1;
}

prediction: GET INDEX INDEX
```

Fig. 1: An adversarial example of the seq2seq model [15] showing different predictions for the source programs with same semantics

In this paper, we propose a novel method to generate adversarial examples in the white-box setting by applying semantics-preserving structural transformations. It involves two steps: important tokens selection and transformations selection. For *important tokens selection*, we parse the program into a sequence of tokens, use the Jacobian matrix to calculate the contribution value of each token to the classification result, and determine the importance order of the tokens according to the contribution value. For *transformations selection*, we select the appropriate program transformations to apply to the perturbation position and select the best transformation according to the model's prediction results.

Our main contributions are summarized as follows:

- We propose a set of program transformations involving identifier renaming and structural transformations. These transformations can ensure that the perturbed program and the source program have the same semantics and are difficult to perceive.
- We propose a novel method of generating adversarial examples for source code processing models in the white-box setting. As far as we know, this is the first time that semantics-preserving structural transformations are applied to generate adversarial examples of source code processing models.
- We conduct systematic experiments to prove the effectiveness of our method. Our experimental results show that the attack success rate of our attack method can improve 8.29% on average compared with the state-of-the-art attack method; adversarial training using the adversarial examples generated by our attack method can reduce the attack success rates of source code processing models by 21.79% on average.

Paper organization. Section II reviews the related work. Section III introduces our proposed program transformations. Section IV gives the design of our adversarial examples generation method. Section V describes the experiments and results. Section VI discusses the limitations and future work. Section VII concludes the paper.

II. Related Work

A. Deep Learning-based Source Code Processing

Deep learning-based source code processing tasks can be divided into two categories: code analysis and program generation [17]. *Code analysis* tasks take source code as input and output many forms such as natural language [18] and code snippets [19]. *Program generation* tasks can take many forms as input such as code [5] and natural language [20] and take the source code as output. In this paper, we use code analysis as the research task, which is a relatively complex and important task in source code processing tasks. For example, code summarization expresses the program as a sequence of tokens, and the model expresses the function name by outputting a natural language sequence.

B. Generation of Adversarial Examples

The adversarial examples was first proposed by Szegedy et al. [9] in image classification. Their experiments show that unnoticeable perturbations to benign input images can lead to misclassification. Researchers have discovered the existence of adversarial examples in the fields of image [21]–[23], natural language processing [24]–[26], speech recognition [27]–[29], etc. However, the generation of adversarial examples in the source code domain is more challenging. Source code is discrete and adversarial examples of source code require compound code semantics. These challenges limit the perturbation added to the source code. Zhang et al. [10] treated the selection counter perturbation as a sampling problem and generated counter samples by sampling the code identifiers. Springer et al. [11] proposed a non-gradient attack method against the source code model, which generated adversarial examples by replacing local variables in the code with variable names with high scores in the code dataset. Yefet et al. [12] and Srikant et al. [15] applied identifier renaming and dead code insertion (to introduce identifiers) as code perturbation sets, which generated adversarial examples through first-order optimization based on gradients.

Our adversarial example generation method is different from existing methods in the following two aspects. First, we use semantics-preserving structural transformations to extend the set of perturbations. Existing methods [10]–[13], [15] mainly generate adversarial examples of the source code model by renaming identifiers, so these attack methods are difficult to find weaknesses in the code structure of the model. In contrast, we consider the equivalent structure transformations that preserve the semantics of the program. Second, we find important tokens in the program for transformation. Bielik et al. [13] used a greedy approach to search for the transformation space. Yefet et al. [12] and Ramakrishnan et al. [14] randomly selected variables to attack. Srikant et al. [15] formalized the adversarial example generation problem as a joint optimization problem of perturbation location selection and perturbation selection. They generated adversarial samples by a first-order optimization algorithm. The difference from their work is that we directly select important tokens in the program as the locations for transformation.

C. Defense and Robustness

In order to prevent adversarial attacks, previous studies have proposed a variety of defense methods to protect the source code model. Generally speaking, the defense methods of source code models can be divided into two categories: detection and model enhancement. For *detection*, anomaly detection is performed on the input samples, so as to protect the model from them [12]. For *model enhancement*, the main method is to train the model to enhance the robustness [9]. Improving model robustness through adversarial training has been widely used in the field of image and natural speech processing, and recently in the field of source program processing [10], [13]–[15]. In this paper, we use the adversarial examples we generated for adversarial training to improve the robustness of the source code processing model.

III. Program Transformations

In this section, we introduce a set of program transformations at the method level to generate semantics-preserving methods. In addition to common identifier renaming, this set also contains transformations that change the code structure. The purpose of the program transformation is to add perturbations to the program to generate adversarial examples, which have the same semantics as the original program, but can make the source code model predict different results from the original prediction results.

In the domains of image and natural language processing, the perturbations added in the adversarial examples are required to be as undetectable as possible by humans. However, there are different requirements for the source code domain. Specifically, we consider the following three factors to select program transformations. The *first* factor is preserving semantics. Semantic program transformations can ensure that the program has the same functionality before and after the transformation. Only the program with the same functionality will we consider the result of the source code model processing to be consistent. The *second* factor is minimum code changes. We can generate programs with the same semantics in many ways, but the adversarial samples generated by changing the code as little as possible can maintain the similarity with the original samples, just like the image domain modifies as few pixels as possible. The *third* factor is code readability. We know that program obfuscation can also generate transformed programs with the same functionality, but the result of obfuscation makes the code difficult for people to understand. Our aim is to generate readable adversarial examples. In other words, the transformations are ubiquitous during program development and the program after we apply these transformations does not look obtrusive. In short, we select program transformations based on these three factors and add these perturbations to the original program to obtain adversarial examples.

In order to introduce perturbations into the source program, we construct a set of program transformations that preserve the semantics, as shown in Table I. There are 6 different program transformations, which can be divided into two types: transformations that do not change the code structure and transformations that change the code structure.

Identifier renaming. Identifier renaming transformation completes the program transformation by renaming the identifier in the code. This method only involves the transformation of the identifier and does not affect the code structure. The original prediction result in Fig. 2(a) is "Check done". As shown in Fig. 2(b), by renaming the variable "done" in the code to the variable "object", the prediction result of the source code model is changed to "run".

Loop exchange. Loop exchange transformation is between *while* statement and *for* statement, which can change the code structure. The original prediction result in Fig. 2(a) is "Check done". As shown in Fig. 2(c), by replacing the *while* statement in line 4 with a *for* statement in the code, combined with Boolean exchange transformation (i.e., replacing $true$ in line 6 with $!false$), the prediction result of the model is changed to "clear".

Comparison operator exchange. Comparison operator exchange transformation completes the equivalent transformation of the comparison operator, which can change the code structure. For example, $a > b$ is equivalent to $b <= a$. The original prediction result in Fig. 2(a) is "Check done". As shown in Fig. 2(d), by converting the comparison expression $remaining <= 0$ in line 5 of the code to $0 >= remaining$, the prediction result of the model is changed to "skip to done".

Boolean exchange. Boolean exchange transformation applies the inversion operation to the Boolean value to complete the equivalent transformation. For example, $true$ is equivalent to $!false$, as shown in Fig. 2(c). This method changes the code structure.

Prefix and suffix exchange. Prefix and suffix exchange transformation converts the prefix and suffix operations to each other, which changes the code structure. The prediction result of the program in Fig. 3(a) is "index of item". As shown in Fig. 3(b), by replacing $index++$ in line 10 of the code with $++index$, the prediction result of the model is changed to "get index index".

```
...
void f() {
  boolean done = false;
  while (!done) {
    if (remaining() <= 0) {
      done = true;
    }
  }
}
...
```

(a) A source program p_1

```
...
void f() {
  boolean object = false;
  while (!object)
    if (remaining() <= 0) {
      object = true;
    }
  }
}
...
```

(b) A variant of p_1 obtained by applying identifier renaming

```
...
void f() {
  boolean done = false;
  for ( ; !done; )
    if (remaining() <= 0) {
      done = !false;
    }
  }
}
...
```

(c) A variant of p_1 obtained by applying loop exchange and boolean exchange

```
...
void f() {
  boolean done = false;
  while (!done) {
    if (0 >= remaining())
      done = true;
    }
  }
}
...
```

(d) A variant of p_1 obtained by applying comparison operator exchange

Fig. 2: A source program example p_1 and its variants obtained by applying program transformations of identifier renaming, loop exchange and Boolean exchange, and comparison operator exchange

TABLE I: Semantics-preserving program transformations involving identifier renaming and structural transformations

Program transformation	Description	Change structure?
Identifier renaming	Rename the identifier in the function to another name	No
Comparison operator exchange	Exchange binary comparison operation	Yes
Boolean exchange	Exchange True and False on an equivalent basis	Yes
Loop exchange	Exchange the for loop and the while loop equivalently	Yes
Compound assignment operator exchange	Exchange compound assignment operation to assignment operation	Yes
Prefix and suffix operator exchange	Exchange the prefix operation and suffix operation to each other	Yes

Compound assignment operator exchange. Compound assignment operator exchange transformation splits the assignment operator, which can change the code structure. For example, $a+=b$ is equivalent to $a=a+b$. As shown in Fig. 3(c), by replacing $index+=1$ in line 10 of the code with $index=index+1$, the model's prediction result is changed to "get index of item".

Fig. 2 and Fig. 3 illustrate the equivalent transformation operations that preserve semantics through two examples. Note that these examples are used to illustrate that the transformations can confuse the model's predictions, not the adversarial examples generated by our attack algorithm.

IV. Design

Our goal is to introduce a systematic way to perturb the program by applying the set of program transformations described above, so that the trained model incorrectly classifies the program that was classified correctly in the first place. That is to say, for a given model and an input sample, an adversarial example is generated by applying program transformations to the input sample. We focus on the white-box attack method, thus the source code processing model structure and parameters are considered to be known. The internal information of the model allows us to find adversarial examples faster, which in turn helps us determine the robustness of the model. The attack method we propose is shown in Fig. 4.

For a program p, in order to generate an adversarial example p', we have to solve two problems. First, which token position should we choose to perform the transformation? Given a program p of length N, how to determine a set of transformations of size k has the greatest impact on the model, where k is the perturbation intensity? We call this problem *important tokens selection* problem. Second, what kind of transformations should be selected for a given token? After determining a set of tokents to be transformed, it is necessary to consider what kind of transformation is performed for each token to make the prediction result of the model change the most. We call this problem the *transformation selection* problem.

For the first problem, we complete the selection of important positions in two steps. First, we represent the program as a sequence of tokens. As shown in Fig. 5(a), all identifiers except

```
...
int f(Item item) {
  int index = 0;
  Iterator<Item> i = item.getParent().
    getItemIterator();
  while (i.hasNext()) {
    Item child = i.next();
    if (item == child) {
      return index;
    }
    index ++;
  }
  return -1;
}
...
```

(a) A source program p_2

```
...
int f(Item item) {
  int index = 0;
  Iterator<Item> i = item.getParent().
    getItemIterator();
  while (i.hasNext()){
    Item child = i.next();
    if (item == child) {
      return index;
    }
    ++ index;
  }
  return -1;
}
...
```

(b) A variant of p_2 obtained by applying prefix and suffix operator exchange

```
...
int f(Item item) {
    int index = 0;
    Iterator<Item> i = item.getParent().
    getItemIterator();
    while (i.hasNext()){
        Item child = i.next();
        if (item == child) {
            return index;
        }
        index = index + 1;
    }
    return -1;
}
...
```

(c) A variant of p_2 obtained by applying compound assignment operator exchange

Fig. 3: A source program example p_2 and its variants obtained by applying program transformations of prefix and suffix operator exchange and compound assignment operator exchange

format control symbols are elements of the sequence. Second, we extract a set of candidate tokens from the sequence. As shown in Fig. 5(b), we use the red solid box to mark the tokens using identifier renaming and use the blue dashed box to mark the tokens using equivalent structural transformation. We control the size of the candidate token set by perturbing the intensity k.

For the second problem, we perform different transformation operations for two different types of tokens. For identifier renaming, we follow the operation of Yefet et al. [12] and use the idea of gradient-based optimization to find the best replacement identifier. For the equivalent structural transformation, we try all available transformations to select the best one. Fig. 5(c) shows an example of equivalent structural transformation. It uses the candidate token marked by the red double solid line box and then selects a binary operator exchange for it. The program after the transformation is shown in Fig. 5(d), and the expression involved is marked with a blue dashed box.

A. *Important Tokens Selection*

Algorithm 1 describes the process of important tokens selection. There are three substeps.

Step I.1: Parse the source program (line 2). For a benign program p, we can parse it as a sequence $\{t_1, \cdots, t_n\}$. The sequence content obtained by different program representation methods is different. The sequence may be composed of tokens, e.g., the seq2seq model parses the program into a sequence of symbols except for format control symbols as the input. It may be composed of other nodes, e.g., the code2seq model extracts the path from the abstract syntax tree to represent the source program.

Step I.2: Calculate the contribution of the tokens (lines 3-5). To evaluate the degree of influence of each token in the program representation sequence on the prediction result of the model, we can calculate the Jacobian matrix to find the important tokens . The Jacobian matrix for the classification task model can be expressed as Eq. (1), where n represents the sequence length of program p, m represents the dimensionality of the model output domain, and $F_j(\cdot)$ represents the confidence level of the model output jth category. For the case where the prediction result is multiple words, the contribution value C_{t_i} of each token can be expressed as Eq. (2), where y is the target label sequence corresponding and N is the length of the prediction result y corresponding to the input p.

$$J_{\mathcal{F}}(\boldsymbol{p}) = \frac{\partial \mathcal{F}(\boldsymbol{p})}{\partial \boldsymbol{p}} = \left[\frac{\partial \mathcal{F}_j(\boldsymbol{p})}{\partial t_i}\right]_{i \in 1..n, j \in 1..m} \tag{1}$$

$$C_{t_i} = \frac{1}{N} \cdot \sum_{j=1}^{N} J_{\mathcal{F}(i,y_j)} = \frac{1}{N} \cdot \sum_{j=1}^{N} \frac{\partial \mathcal{F}_{y_j}(\boldsymbol{p})}{\partial t_i} \tag{2}$$

Step I.3: Sort and collect candidate tokens set (lines 6-7). After getting the contribution value of each token, we reverse sort the tokens according to the contribution value. In this way, we can get a sequence of tokens sorted by decreasing importance. We select the top k tokens from the sorted sequence of tokens as the candidate token set $T_{candidate}$.

B. *Transformations Selection*

After determining the candidate important tokens set, we need to choose the best transformation for each token in the candidate set. Algorithm 2 describes the process involving four substeps.

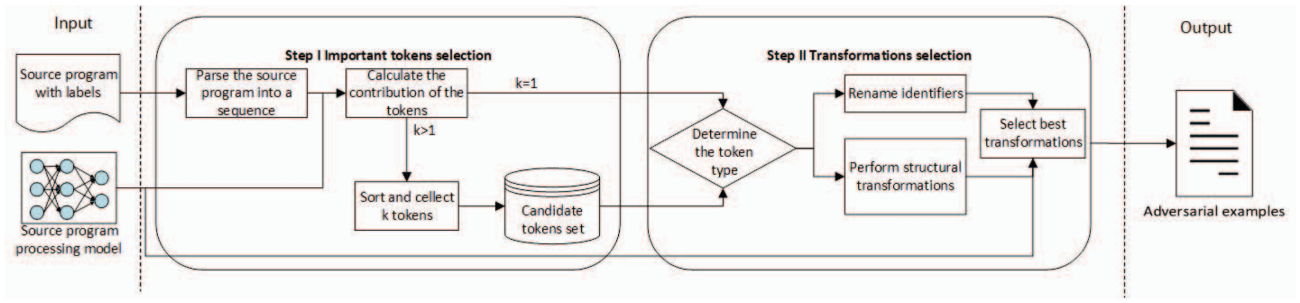
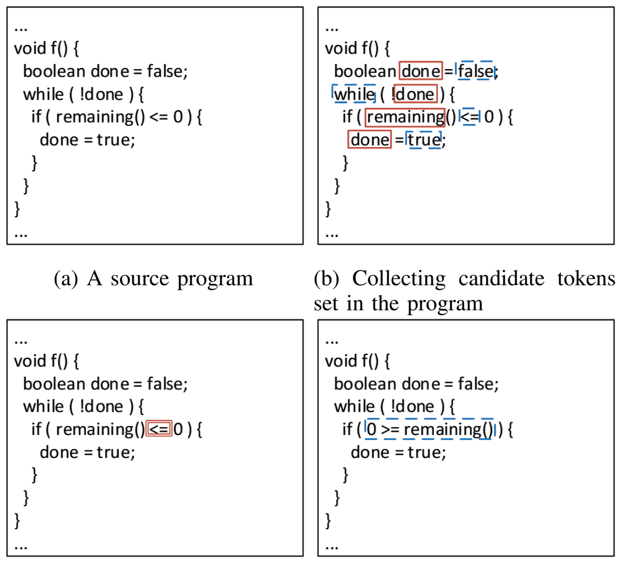

Input
Source program with labels
Source program processing model
Step I Important tokens selection
Parse the source program into a sequence
Calculate the contribution of the tokens
k=1
k>1
Sort and cellect k tokens
Candidate tokens set
Step II Transformations selection
Determine the token type
Rename identifiers
Perform structural transformations
Select best transformations
Output
Adversarial examples

Fig. 4: Overview of our attack method, involving two steps: important tokens selection and transformations selection



```
...
void f() {
  boolean done = false;
  while ( !done ) {
    if ( remaining() <= 0 ) {
      done = true;
    }
  }
}
...
```

(a) A source program

```
...
void f() {
  boolean done = false;
  while ( !done ) {
    if ( remaining() <= 0 ) {
      done = true;
    }
  }
}
...
```

(b) Collecting candidate tokens set in the program

```
...
void f() {
  boolean done = false;
  while ( !done ) {
    if ( remaining() <= 0 ) {
      done = true;
    }
  }
}
...
```

(c) Selecting important tokens in the program

```
...
void f() {
  boolean done = false;
  while ( !done ) {
    if ( 0 >= remaining() ) {
      done = true;
    }
  }
}
...
```

(d) A program variant after applying the best transformation

Fig. 5: An example showing the process of attacking the source program. Fig. 5(a) shows a source program, Fig. 5(b) shows the candidate tokens set of the program snippet. The red solid box indicates the token that can be transformed by identifier renaming, and the blue dashed box indicates the token that can be applied to the structural transformation. The red double solid line box in Fig. 5(c) represents the important token selected from the candidate tokens set, and the blue dashed box in Fig. 5(d) identifies the result of selecting the important tokens and applying the best transformation.

Algorithm 1 Important tokens selection

```
Input: $p$: source program; $F$: source program processing model; $T_{candidate}$: candidate tokens set; $k$: the number of candidate tokens;
Output: Candidate important tokens set $T_{candidate}$;
1: $T_{candidate} \leftarrow \varnothing$;
2: Parse the program $p$ into a sequence of tokens $\{t_1, \cdots, t_n\}$;
3: for each $t \in p$ do
4:     Compute contribution $C_i$ according to Eq. (2);
5: end for
6: $tmp \leftarrow Sort(t_1, \cdots, t_n)$ according to $C_i$;
7: Collect the top $k$ tokens from $tmp$ as candidate tokens to $T_{candidate}$;
8: return $T_{candidate}$;
```

Step II.1: Determine the token type (line 3). The transformation that can be performed by different tokens can be divided into two categories: identifier renaming and equivalent structural transformation. Identifier renaming does not change the program structure, but only changes the value of the variable in the program. The equivalent structural transformation changes the code structure. For these two different transformations, we adopt different transformation selection strategies.

Step II.2: Rename identifiers (lines 4-5). We use the gradient-based approach proposed by Yetel et al. [12] to select the best replacement for the identifier. In this attack, our goal is to rename the given identifier t to r so that the model predicts an incorrect expected value. We maximize the loss in the direction of gradient ascent and then get the worst token replacement. In other words, we choose the worst r in t' to replace the original t in program p to obtain the transformed program p^*.

$$t' \leftarrow t + \eta \cdot \nabla_t L(\Theta, p, y); r = \arg\max t' \tag{3}$$

$$p^* = p[t \leftarrow r], \tag{4}$$

where L is the loss function of the attacked model, Θ is the parameter learned by the model, p is the program snippet input to the model, and y is the target label.

Step II.3: Perform structural transformations (lines 7-8). To perform the transformations on tokens involved in program structure change, the first challenge to be solved is how to associate the token with the expression to be transformed. We map the token and the transformation it will perform. For example, if the candidate token is the keyword "for", we can find the for loop statement corresponding to the token and use the "$while$" loop for equivalent replacement. If a token corresponds to multiple transform methods, we use Eq. (5)

Algorithm 2 Transformation selection

```
Input: $T_{candidate}$: candidate tokens set; $p$: source program; $y$: target labels; $F$: source program processing model;
Output: The adversarial example $p_{adv}$ generated by perturbing the source program $p$
1:  $V_p^* \leftarrow \varnothing$;
2:  for each $t \in T_{candidate}$ do
3:      if $t$ is a variable name then
4:          $r = \arg\max t'$, $t'$ is calculated by Eq. (3);
5:          $p^* = p[t \leftarrow r]$;
6:      else if $t$ is a structural identifier then
7:          $V_t \leftarrow \{p' \mid p'$ is the program after the program $p$ can perform the transformation on the $t\}$;
8:          $p^* = \arg\max_{p' \in V_t} score(p) - score(p')$;
9:      end if
10:     $V_p^* = V_p^* \cup p^*$;
11: end for
12: for $p^* \in V_p^*$ do
13:     if $F(p^*) \neq y$ then
14:         $p_{adv} = p^*$;
15:         return $p_{adv}$;
16:     end if
17: end for
```

to calculate the model's predicted result score change of the transformed program to select the optimal transformation.

$$p^* = \arg\max(score(p) - score(p')) \tag{5}$$

The program p' is the transformed program obtained after the original program p executes the transformations related to the tokens. $score(p)$ is the output obtained after inputting program p to the source code processing model. In other words, p^* is one of the most significant changes caused by token-related conversions.

In this way, we can select an optimal transformation for each token in the candidate tokens set and collect the transformed programs into set V_p^*, so as to select the adversarial example from it.

Step II.4: Select best transformations (lines 12-17). We traverse V_p^* and feed the program p^* into the source code processing model. If the prediction result of the model changes, then the program p^* is an adversarial example p_{adv}.

V. Experiments and Results

Our experiments aim to answer the following four *Research Questions* (RQ):

- **RQ1**: Can the important tokens selection improve the effectiveness of adversarial attacks against source code processing models?
- **RQ2**: Are the source code processing models vulnerable to the perturbation of equivalent structural transformations?
- **RQ3**: Is our attack method more effective than existing attack methods with different perturbation intensity?
- **RQ4**: Can the adversarial examples generated by our method improve the robustness of source code processing models through adversarial training?

A. Experimental Setup

Task and datasets. We take the code summarization task as an example to evaluate our method. Code summarization uses the given method body to predict the method's name, which is a relatively complex and important task in source code processing tasks [3]. Our goal is to generate adversarial examples that can change the model's original successful prediction results. We conduct experiments on two Java datasets which are widely used for the evaluation of code summarization tasks: (i) code2seq's Java-small dataset (denoted as Java-small) [4] and (ii) CodeSearchNet Java datasets from GitHub (denoted as CSN-Java) [30].

Evaluation metrics. We use two widely-used evaluation metrics: *F1-measure* (F1) and *Attack Success Rate* (ASR) [15]. Let `TP` denote the number of positive samples that are successfully predicted as positive, `TN` denote the number of negative samples that are successfully predicted as negative, `FP` denote the number of negative samples that are falsely predicted as positive, and `FN` denote the number of positive samples that are falsely predicted as negative. The metric $F1$ measures the overall effectiveness of the source code processing model by considering both $precision$ and $recall$.

$$F1 = \frac{2 \times precision \times recall}{precision + recall}, \tag{6}$$

where $precision = \frac{\texttt{TP}}{\texttt{TP}+\texttt{FP}}$ is the ratio of true positive samples to the predicted positive samples and $recall = \frac{\texttt{TP}}{\texttt{TP}+\texttt{FN}}$ is the ratio of true positive samples among to the entire positive samples.

The metric *ASR* is the percentage of the number of perturbed samples that the source program processing model predicts incorrectly to the total number of samples. These samples are the model's original ability to predict success.

$$ASR = \frac{\sum_{i,j} \mathbb{1}\left(\mathcal{F}\left(p_i'\right) \neq y_{ij}\right)}{\sum_{i,j} \mathbb{1}\left(\mathcal{F}\left(p_i\right) = y_{ij}\right)}, \tag{7}$$

where p_i is a program, p_i' is an adversarial example of p_i, and y_{ij} is jth token in the expected output of the sample p_i. A higher $F1$ means that the source code processing model is more effective and a higher ASR means that the attack method is more effective.

Implementation. We consider two Sequence to Sequence (seq2seq) models as victim models: LSTM-based seq2seq model and GRU-based seq2seq model [14], [15]. Each seq2seq model takes a sequence of tokens representing each program as input and generates another sequence of tokens representing the method name. The LSTM-based seq2seq model uses 2-layer BiLSTM with attention mechanism and the GRU-based seq2seq model uses 2-layer BiGRU with attention mechanism. We implement equivalent structural transformations based on

TABLE II: Effectiveness of different attack methods related to important tokens selection (unit:%)

Model	Attack method	Java-small		CSN-Java	
		ASR	F1	ASR	F1
LSTM-based seq2seq	Random attack	0	100.00	0	100 .00
	Our method	39.15	56.51	38.10	48.55
GRU-based seq2seq	Random attack	0	100.00	0	100.00
	Our method	37.03	59.16	37.48	49.69

JavaParser[1] library. We conduct experiments on an Intel (R) Core (TM) i9-9820X machine with a frequency of 3.30 GHz and GeForce RTX 2080.

B. Impact of Important Tokens Selection (RQ1)

To determine whether the important tokens selection can improve the effectiveness of adversarial attacks, we compare the effectiveness of our attack method with random attack method. For our attack method, we leverage Algorithm 1 to find the important tokens in the program and perform the best transformation on the tokens to generate adversarial examples. We set the perturbation intensity $k = 1$; we will discuss the different choices of k later. For the random attack method, we randomly select tokens from the program as candidate tokens, and then choose the best transformation for it.

Table II illustrates the results of our experiment. We observe that the attack success rates (or F1s) for the two seq2seq models are similar on both Java-small dataset and CSN-Java dataset. For the random attack, the attack success rate is 0, which shows that the strategy of randomly selecting important tokens for transformation cannot change the original prediction results of the source code processing model. For our attack method, using important tokens selection can achieve a 38.63% higher average attack success rate on the LSTM-based model and a 37.26% higher average attack success rate on the GRU-based model than the random attack method. This shows that choosing important tokens to apply equivalent program transformations can introduce effective perturbations in the program.

Insight 1: Important tokens selection can effectively improve the attack success rate by guiding program transformations.

C. Impact of Equivalent Structural Transformations (RQ2)

To show the impact of structural transformations on the effectiveness of attacks against the source code processing model, we use three strategies in the experiment: using all transformations (denoted by "All"), only using identifier renaming (denoted by "IR"), and only using structural transformations (denoted by "ST"). We set the perturbation intensity $k = 1$. The experimental results are illustrated in Table III.

We observe that three attack strategies exhibit similar phenomena with respect to different datasets. The attack method with IR strategy achieves an average attack success rate of 27.30% and an average F1 of 67.02% on the LSTM-based model, and an average attack success rate of 26.39% and an average F1 of 68.16% on the GRU-based model. The attack method with ST strategy achieves an attack success rate of 23.00% and an average F1 of 74.56% on the LSTM-based model, and an average attack success rate of 21.85% and an average F1 of 75.63% on the GRU-based model. This shows that the seq2seq model is vulnerable to both identifier renaming and program structural transformations. We can see that the attack success rate and the F1 of the attack method with All (IR+ST) strategy improves by 11.10% and 14.11% on average, compared with the attack method with IR strategy. This shows that semantics-preserving structural transformation and identifier renaming together can significantly improve the effectiveness of attacks.

[1]https://github.com/javaparser/javaparser

TABLE III: Impact of identifier renaming and structural transformations on the effectiveness of attacks (unit:%)

Model	Attack method	Java-small		CSN-Java	
		ASR	F1	ASR	F1
LSTM-based seq2seq	IR	28.19	68.36	26.40	65.67
	ST	24.09	73.43	21.92	75.68
	All	39.15	56.51	38.10	48.55
GRU-based seq2seq	IR	27.7	69.03	25.07	67.28
	ST	22.90	74.60	20.79	76.67
	All	37.03	59.16	37.48	49.69

Insight 2: Source code processing models are vulnerable to the perturbation of equivalent structural transformations; identifier renaming and structural transformations together can achieve better effectiveness of attacks.

D. Comparing Attack Methods (RQ3)

To test whether our attack method is more effective than existing attack methods, we take the Java-small dataset as an example to show the experimental results for the LSTM-based model and the GRU-based model, since the CSN-Java dataset achieves similar results. We use the *Alternating Optimization with with Randomized Smoothing* (AO+RS) [15] as the baseline, which is the state-of-the-art attack method for code summarization under the white-box setting. We also discuss the impact of different choices of perturbation intensity k, i.e., $k = 1$ and $k = 5$, on the effectiveness of attacks.

The experimental results are shown in Table IV. We observe that our attack is better than AO+RS, especially when k=1, the attack success rate of our method is improved by an average of 8.29% in the two models. Specifically, for $k = 1$, our attack success rates are 7.74% (24.65% improvement) and 8.84% (31.36% improvement) respectively for the LSTM-based model and the GRU-based model; for $k = 5$, our attack success rates are 1.94% and 3.78% respectively for the LSTM-based model and the GRU-based model. This can be explained by the fact that we introduce more important transformations into the program through important tokens selection. In addition, when the perturbation intensity k increases from 1 to 5, the attack success rate of our method increases by 5.40% and the attack success rate of AO+RS method increases by 11.20% on the LSTM-based model; the attack success rate of our method increases by 4.56% and the attack success rate

TABLE IV: Comparing with existing attack methods with different perturbation intensity on the Java-small dataset (unit:%)

Attack method	k=1		k=5	
	ASR	F1	ASR	F1
LSTM-based seq2seq				
AO+RS	31.41	65.85	42.61	54.55
Our method	39.15	56.51	44.55	51.82
GRU-based seq2seq				
AO+RS	28.19	68.35	37.81	58.81
Our method	37.03	59.16	41.59	54.92

of AO+RS method increases by 10.72% on the GRU-based model. This shows that our attack method is less affected by the perturbation intensity k, which can be attributed to the effectiveness of the important tokens selection strategy.

Insight 3: Compared with the state-of-the-art attack method (AO+RS), when k=1, the performance of our attack method has increased by 24.65% and 31.36%, the attack success rate of our method is improved by an average of 8.29% in the two models, and when k=5, our attack method is still ahead by 2.88% on average; the important tokens selection strategy makes the perturbation intensity k easier to converge.

E. Improvement of Robustness (RQ4)

To verify whether the adversarial examples generated by our attack method can improve the robustness of the source code model through adversarial training, we use the adversarial examples generated by adversarial attacks to train the model. We took the Java-small data set as an example for adversarial training, where the perturbation intensity $k = 1$. We consider the following three models:

- **No-AT.** The model is obtained using a normal training method, i.e., without adversarial training.
- **AO+RS-AT.** The model uses the adversarial examples generated by the AO+RS attack method [15] to conduct adversarial training on the model.
- **Our-AT.** The model uses the adversarial examples generated by our attack method to conduct adversarial training on the model.

The experimental results are shown in Table V. We observe that the attack success rate of AO+RS-AT model for the LSTM-based seq2seq model is improved by 13.68% compared with No-AT, while the attack success rate of our method is improved by 21.74%. For the GRU-based seq2seq model, the attack success rate of the AO+RS-AT model is improved by 13.65% compared with No-AT, while the attack success rate of our method is improved by 21.84%. This shows that the adversarial examples generated by our attack method can improve the robustness of the model through adversarial training more effectively.

Insight 4: Adversarial examples generated by our attack method can reduce the attack success rate of source code processing models by 21.79% on average through adversarial training.

TABLE V: Comparing attack success rates after adversarial training (unit:%)

Model	Attack method	
	AO+RS	Our method
LSTM-based seq2seq		
No-AT	31.41	39.15
AO+RS-AT	15.53	25.47
Our-AT	13.68	17.41
GRU-based seq2seq		
No-AT	28.19	37.03
AO+RS-AT	13.16	23.38
Our-AT	11.72	15.19

VI. Limitations and Future Work

Although our experimental results show the effectiveness of our method, there are still some limitations. First, we only evaluate the code summarization task in the source code processing task. We will verify the effectiveness of our attack methods in tasks such as code completion and code classification in the future. Second, we verify our attack method on the seq2seq model, and we will evaluate the effectiveness of our method on other models, e.g., the code2seq model, in the future. Third, the equivalent structural transformations we propose are language-independent, but not each transformation can be applied to each programming language. Therefore, for the future work, we should propose targeted transformations for different programming languages. Fourth, our attack algorithm is carried out under the white-box setting, and we need to understand the structure and parameters of the model. It is an interesting work to consider applying equivalent structural transformations to attack under the black-box setting, so as to get more general attack methods.

VII. Conclusion

We propose a novel method of applying semantics-preserving structural transformations to attack the source program processing model under the white-box setting. This method decomposes the attack into two steps: important tokens selection and transformations selection. Important tokens selection is perturbed by selecting the token that contributes a large value to the result in the program, which can locate the position to be converted more effectively than the state-of-the-art method. Transformations selection adds the most effective perturbation to the program through two strategies. Therefore, our attack method can efficiently generate adversarial examples. Experimental results show the effectiveness of our attack method. The limitations discussed in Section VI provide open problems for future research.

Acknowledgment

We thank the reviewers for their constructive comments. This work was 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. Any opinions, findings, or conclusions

expressed in this work are those of the authors and do not reflect the views of the funding agencies in any sense.

References

[1] A. F. Del Carpio and L. B. Angarita, "Trends in software engineering processes using deep learning: A systematic literature review," in *Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Portorož, Slovenia*, pp. 445–454, 2020.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[3] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33rd International Conference on Machine Learning (ICML),New York, USA*, pp. 2091–2100, 2016.

[4] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proceedings of the International Conference on Learning Representations (ICLR),Vancouver,Canada*, 2019.

[5] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *Proceedings of the 37th International Conference on Machine Learning (ICML), Vienna, Austria*, pp. 245–256, 2020.

[6] 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 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montréal, QC, Canada*, pp. 783–794, 2019.

[7] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2021.

[8] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proceedings of the 33nd Advances in Neural Information Processing Systems (NeurIPS), Vancouver, Canda*, 2019.

[9] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[10] 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 AAAI Conference on Artificial Intelligence, New York, USA*, pp. 1169–1176, 2020.

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

[12] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[13] P. Bielik and M. Vechev, "Adversarial robustness for code," in *Proceedings of the 37th International Conference on Machine Learning (ICML), Vienna, Austria*, pp. 896–907, 2020.

[14] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," *arXiv preprint arXiv:2002.03043*, 2020.

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

[16] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of neural program models with respect to semantic-preserving program transformations," *Information and Software Technology*, vol. 135, p. 106552, 2021.

[17] T. H. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[18] A. Sharma, Y. Tian, and D. Lo, "Nirmal: Automatic identification of software relevant tweets leveraging language model," in *in 2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER), Montreal, QC, Canada*, pp. 449–458, 2015.

[19] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proceedings of the 15th International Conference on Mining Software Repositories,Gothenburg, Sweden*, p. 476–486, 2018.

[20] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, United Kingdom*, p. 419–428, 2014.

[21] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS & P)*, pp. 372–387, 2016.

[22] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 427–436, 2015.

[23] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, pp. 14410–14430, 2018.

[24] J. Li, S. Ji, T. Du, B. Li, and T. Wang, "Textbugger: Generating adversarial text against real-world applications," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA*, 2019.

[25] D. Jin, Z. Jin, J. T. Zhou, and P. Szolovits, "Is bert really robust? a strong baseline for natural language attack on text classification and entailment," in *Proceedings of the AAAI conference on Artificial Intelligence*, pp. 8018–8025, 2020.

[26] W. E. Zhang, Q. Z. Sheng, A. Alhazmi, and C. Li, "Adversarial attacks on deep-learning models in natural language processing: A survey," *ACM Trans. Intell. Syst. Technol.*, vol. 11, no. 3, pp. 1–41, 2020.

[27] S. Samizade, Z.-H. Tan, C. Shen, and X. Guan, "Adversarial example detection by classification for deep speech recognition," in *Proceedings of the 45th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 3102–3106, 2020.

[28] Y. Qin, N. Carlini, G. Cottrell, I. Goodfellow, and C. Raffel, "Imperceptible, robust, and targeted adversarial examples for automatic speech recognition," in *Proceedings of the 36th International Conference on Machine Learning (ICML), California, US*, pp. 5231–5240, 2019.

[29] S. Hu, X. Shang, Z. Qin, M. Li, Q. Wang, and C. Wang, "Adversarial examples for automatic speech recognition: Attacks and countermeasures," *IEEE Communications Magazine*, vol. 57, no. 10, pp. 120–126, 2019.

[30] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.