Adversarial Robustness of Deep Code Comment Generation

YU ZHOU, XIAOQING ZHANG, and JUANJUAN SHEN, Nanjing University of Aeronautics and Astronautics, China
TINGTING HAN and TAOLUE CHEN*, Birkbeck, University of London, UK
HARALD GALL, University of Zurich, Switzerland

Deep neural networks (DNNs) have shown remarkable performance in a variety of domains such as computer vision, speech recognition, and natural language processing. Recently they also have been applied to various software engineering tasks, typically involving processing source code. DNNs are well-known to be vulnerable to adversarial examples, i.e., fabricated inputs that could lead to various misbehaviors of the DNN model while being perceived as benign by humans. In this paper, we focus on the code comment generation task in software engineering and study the robustness issue of the DNNs when they are applied to this task. We propose ACCENT (**A**dversarial **C**ode **C**omment g**EN**era**T**or), an identifier substitution approach to craft adversarial code snippets, which are syntactically correct and semantically close to the original code snippet, but may mislead the DNNs to produce completely irrelevant code comments. In order to improve the robustness, ACCENT also incorporates a novel training method, which can be applied to existing code comment generation models. We conduct comprehensive experiments to evaluate our approach by attacking the mainstream encoder-decoder architectures on two large-scale publicly available datasets. The results show that ACCENT efficiently produces stable attacks with functionality-preserving adversarial examples, and the generated examples have better transferability compared with the baselines. We also confirm, via experiments, the effectiveness in improving model robustness with our training method.

CCS Concepts: • **Software and its engineering**; • **Computing methodologies** → **Artificial intelligence**;

Additional Key Words and Phrases: Code Comment Generation, Adversarial Attack, Deep Learning, Robustness

1 INTRODUCTION

Code comment generation aims to generate readable natural language descriptions of source code snippets, which plays an important role in facilitating program comprehension. Encouraged by the great success of deep learning methods in typical application areas such as computer vision and natural language processing, researchers have proposed deep neural network (DNN) based approaches for the code comment generation task [1, 14, 52], aiming to improve the quality of the generated comments.

It is well-recognized that DNNs are not robust. In particular, adversarial examples, which can be crafted by adding small perturbations to benign inputs of the model, may easily fool DNNs [12, 29], or at least elicit large changes in the model output. This would greatly impede the usability of DNN models [33], since ideally the model should generate indistinguishable comments for similar code snippets. In other words, minor semantic-preserving

*Corresponding author.

Authors' addresses: Yu Zhou, zhouyu@nuaa.edu.cn; Xiaoqing Zhang, zhangxq@nuaa.edu.cn; Juanjuan Shen, shenjuanjuan@nuaa.edu.cn, Nanjing University of Aeronautics and Astronautics, Nanjing, China; Tingting Han, t.han@bbk.ac.uk; Taolue Chen, t.chen@bbk.ac.uk, Birkbeck, University of London, London, UK; Harald Gall, gall@ifi.uzh.ch, University of Zurich, Zurich, Switzerland.

1049-331X/2022/1-ART $15.00
https://doi.org/10.1145/3501256

perturbation of the code snippets should have a minimum side-effect on the generated comments. As a result, when neural networks are adopted, improving their robustness has become indispensable.

Adversarial examples have been shown to be an effective way of assessing and improving the robustness of neural networks. In light of this, recently the deep learning community has seen a wide variety of methods to generate adversarial examples, especially for image classification [8, 12] and some NLP tasks [18, 31, 53]. Likewise, when applying DNNs to programming and software engineering tasks, it is also vital to improve the robustness of the model, which demands effective and efficient ways to generate adversarial examples. However, this is considerably more challenging for the source code of programming languages. One of the reasons is that it must satisfy various syntactic and semantic constraints, which are more stringent than the image or NLP cases. For instance, the syntactic constraint stipulates that the adversarial code snippet must be compilable and executable, whereas the semantic constraint stipulates that it must preserve the "meaning" of the original code. Nonetheless, the perturbations reveal the weakness of the model only if they do not change the input so significantly but can legitimately result in changes in the expected output. Another source of difficulty lies in that, comparatively speaking, generating adversarial images is usually much easier because, fundamentally it is a continuous optimization problem where powerful, gradient-based techniques can be utilized. In contrast, the adversarial program is of discrete nature. Notice that, when applying deep learning methods to source code, the program snippet is usually embedded into a vector space, giving rise to a continuous representation. However, in general, there is no correspondence between the perturbed representation and the valid tokens in the code snippet, which rules out a straightforward adaptation of the current approaches in image classification to the domain of programs.

The current task is more akin to the NLP domain as both are dealing with discrete texts. The key difference is that in the case of programs, one has to consider the rigid grammar imposed on programs; fundamentally a programming language is an abstract language. In other words, the adversarial perturbed code must be compilable and semantically equivalent for which natural languages (such as English) are much more liberal and easier to achieve. In contrast, it becomes harder to synthesize adversarial examples for source code when applying NLP methods directly.

In this paper, we propose a novel approach ACCENT (**A**dversarial **C**ode **C**omment g**EN**era**T**or) to generate adversarial examples and improve the robustness of neural networks for the code comment generation task. In a nutshell, we identify the importance of different identifiers appearing in the code snippet and rename them iteratively without breaking the syntactic structure and semantic of the code snippet. Furthermore, we adopt a new training method to improve the robustness of code comment generation models.

Figure 1 exemplifies an adversarial example, where 'in' and 'out' refer to the input code snippet and the generated comment by a comment generator based on the Transformer architecture [1]. This example substitutes the function name 'remove' with 'delete' and 'index' with 'index1', which are syntactically correct and clearly does not change the semantic or the functionality of the code (cf. adv-in), so should have very similar comments. However, rather surprisingly, for this seemingly innocent new code snippet, the comment "deletes a refresh from the specified name (does not exist)." is generated, which is completely irrelevant and indeed very distant from the reference comment.

We carry out evaluations to assess the effectiveness of our approach. For the dataset, we use the publicly available Java source code dataset [14] which was extracted from GitHub,[1] and Python dataset which was extracted from [41]. We consider five sequence-to-sequence (seq2seq) models for comment generation for which representative work of various architectures is selected. The experimental results show that ACCENT is capable of attacking all different models and is beneficial to improve the adversarial robustness without jeopardizing the performance (i.e., the quality of the generated comments).

[1]https://github.com/

Original-in:

```
public JSONObject remove(String name ) {
        if ( name == null ) {
                throw new NullPointerException ( STRING );
        }
        int index = indexOf ( name );
        if ( index != - _NUM ) {
                table.remove ( index );
                names.remove ( index );
                values.remove ( index ) ;
        }
        return this;
}
```

Out:

removes a member with the specified name from this object .

Adv-in:

```
public JSONObject delete(String name ) {
        if ( name == null ) {
                throw new NullPointerException ( STRING );
        }
        int index1 = indexOf ( name );
        if ( index1 != - _NUM ) {
                table.remove ( index1 );
                names.remove ( index1 );
                values.remove ( index1 ) ;
        }
        return this;
}
```

Out:

deletes a refresh from the specified name (does not exist).

Reference:

removes a member with the specified name from this object .

Fig. 1. An adversarial attack example for a comment generation model

Our main contributions are summarized as follows.

- We propose a novel approach to assess and improve the robustness of neural source code models for the comment generation task, including both adversarial examples generation and novel training methods. To the best of our knowledge, it represents one of the first work addressing the model robustness of such a task.
- We conduct comprehensive experiments to demonstrate the effectiveness of our approach, which also confirms the transferability of the generated adversarial examples, crucial for black-box attacks.

- We make the implementation of our approach, as well as the datasets publicly available,[2] which not only can facilitate the replication of our work, but also provides potential usage for related software engineering research and practice.

Structure of the paper. The remainder of the paper is organized as follows. Section 2 introduces the background. Section 3 describes the technical details of our approach, and Section 4 presents the experimental results. Discussions are given in Section 5, followed by a discussion of the related work in Section 6. We conclude the paper and outline future research plans in Section 7.

2 BACKGROUND

2.1 Source Code Comment Generation

Code comment generation is a typical software engineering task. Here both the code and the generated comment are regarded as sequences of tokens that can be represented by vectors, for which sequence-to-sequence (seq2seq) models are suitable and commonly adopted. In a nutshell, the seq2seq model turns one sequence into another one utilizing a recurrent neural network (RNN) or variants thereof, such as long short-term memory (LSTM) or gated recurrent unit (GRU) models, to avoid the problem of vanishing gradient. Typically, the model is based on the encoder-decoder architecture where both encoder and decoder are neural networks; the former turns each item into a corresponding hidden vector containing the item and its context, and the latter reverses the process, turning the vector into an output item, using the previous output as the input context. In addition to the classic RNN-based approaches [5], recent developments include various attention mechanisms [40] which allow the decoder to look at the input sequence selectively rather than generate a single vector which stores the entire context. A typical of example of the models with attentions is Transformer and BERT.

2.2 Adversarial Attacks

Adversarial attacks can be described as the process that, given the original input x, finds an adversarial perturbation δ such that $x + \delta$ can dramatically degrade the model's performance. Adversarial attacks can be conducted in both white-box and black-box manners depending on the attacker's knowledge on the model. For white-box attacks [31, 36], attackers have full access to the target model, e.g., the architecture and parameters. For black-box attacks [11], they have no or little knowledge about the target model. From another perspective, according to the purpose of the attacker, there are targeted or non-targeted adversarial attacks. Take the classification model as an example, attackers purposefully mislead the model to a selected label in the targeted attack, while they only aim at fooling the model in the non-targeted attack.

One can adapt the adversarial attacks to the code comment generation setting in a rather straightforward way. Given a (well-trained) code comment generation model M, an adversarial example can be generated by identifying a perturbation δ that maximizes the model degradation L_{adv}. Formally, $x^* = x + \delta$, where

$$\delta := \underset{||\delta||_p \leq \epsilon}{\arg\max} \{L_{adv}(x + \delta) - \lambda C(\delta)\}$$

Here, $C(\delta)$ captures the semantic and syntactic constraints; λ is the regularization penalty; $||\delta||_p$ represents the constraint on the perturbation δ. Note this seemingly simple formulation does not lend itself to efficient solutions; it merely provides a conceptual framework.

Transferability of adversarial examples. The transferability of adversarial examples has been widely exploited in adversarial attacks, which refers to the phenomenon that examples generated on one model can also be used to attack other models for similar tasks [12, 28]. Transferability is an important property reflecting the generalizability

[2]https://github.com/zhangxq-1/ACCENT-repository

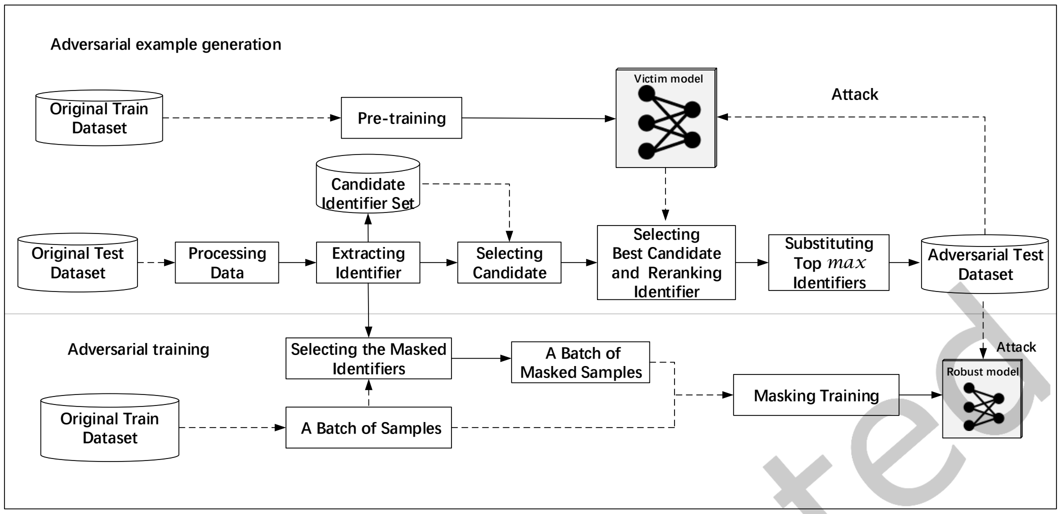


Fig. 2. The workflow of ACCENT

of the attack method, i.e., the higher the transferability of adversarial examples, the better the generalization ability of the attack method.

2.3 Defense and Robustness

To thwart adversarial attacks, various defense methods have been proposed to protect DNN models. In general, defense methods can be classified into two categories: detection and model enhancement [42]. For the former, defenders try to detect adversarial examples so can shield the model from them. For the latter, the main task is to train the model to enhance its robustness. Among others, adversarial training [12] is a widely adopted model enhancement approach, which has been successfully applied to image processing [15] and NLP [4, 11, 22] domains. In a nutshell, it mixes adversarial examples with the original dataset to synthesize a new dataset, which is used to re-train the model.

Note that in literature there are a number of variants of adversarial training, which is usually used as an umbrella term to refer to a family of training methods that utilize adversarial examples to improve the robustness of deep learning models. For instance, Madry et al. [24] formulate the adversarial training as a min-max optimization problem, which is challenging to solve. In this paper, we instead pursue a lightweight adversarial training method in Section 3.2.

The robustness of the model has a far-reaching influence on deep learning in, for example, representation learning and model interpretability. Ilyas et al. [16] claim that adversarial vulnerability is caused by non-robust features. DNN models are vulnerable to attacks because of the well generalizing non-robust features in the data. Although robust features and non-robust features are both useful, a robust model should learn the robust features, rather than non-robust ones. Our work contributes to the understanding of the model robustness for a new application domain, i.e., software engineering.

3 OUR APPROACH

The overview of ACCENT is given in Figure 2. There are mainly two parts in ACCENT, i.e., adversarial examples generation and adversarial training. For the former, source code in the original test data-set goes through a series of processing steps to generate the best candidate identifiers to substitute as adversarial examples. For the latter, the original training data and the masked data are used together for the adversarial training. The details of these two parts are described in Section 3.1 and Section 3.2 respectively.

Algorithm 1: Adversarial Example Generation Algorithm

```
Input: Code Comment Generation Model $M$;
Code Comment Generation DataSet $D$, where $(p, com) \in D$, $p$ is the original program snippet and $com$ is the comment;
Max Substitute Number max;
Candidate Identifier Number $K$;
Output: Adversarial DataSet $D_{adv}$;
1  Initialize: Candidate Identifier Set $V \leftarrow \emptyset$, Adversarial DataSet $D_{adv} \leftarrow \emptyset$;
2  for each $(p, com) \in D$ do
3      $V \leftarrow V \cup \{w \mid w$ is an identifier and $w$ is defined in $p\}$;
4  end
5  Training Identifier Embedding $Embed$;
6  for each $(p, com) \in D$ do
7      Extract the identifier set $V_p$ for $p$ by $V_p \leftarrow \{w \mid w$ is an identifier and $w$ is defined in $p\}$;
8      for each $w \in V_p$ do
9          Select $K$ candidate substitute identifiers $L_w \subseteq V - V_p$ for the identifier $w$ based on the cosine similarity;
10         $w^* \leftarrow \arg\max_{w' \in L_w} \{score(p) - score(p[w \leftarrow w'])\}$;
11         Extract the embedding of identifier $w$ from $Embed$ and the embedding of $p$ from encoder;
12         Calculate the identifier saliency $S(p, w)$;
13         Calculate $H(p, p^*, w)$;
14     end
15     For $w \in V_p$, reorder $w$ according to $H(p, p^*, w)$ in descending order;
16     for $index \leftarrow 1$ to max do
17         Generate $p_{adv}$ by replacing $w$ with $w^*$;
18     end
19     $D_{adv} \leftarrow D_{adv} \cup \{(p_{adv}, com)\}$
20 end
21 return $D_{adv}$;
```

3.1 Adversarial Attack

For the adversarial attack, we mainly consider two types of programmer-defined identifiers, i.e., single-letter and non-single-letter identifiers. For the former, we simply change it to a different letter randomly. For the latter, we adopt a black-box, non-target search-based method to generate adversarial examples. We first extract identifiers from all the program in the dataset to build up a candidate identifier set. For each identifier in the program, we select the nearest K identifiers from the candidate set according to the cosine similarity to form a sub-candidate set, from which the best candidate is identified based on its effect on the generated code comment. We then rank these candidate identifiers based on their contextual relation to the program. Finally, we generate adversarial examples by replacing the identifier with its best candidate according to the order determined in the ranking. In the sequel, we elaborate these steps.

Step 1: Identifier Extraction ("Extracting Identifier" in Figure 2). The first step is to extract identifiers from program snippets and build a candidate identifier set (cf. Line 2-3 in Algorithm 1). Since the functionality of a program snippet does not depend on the programmer-defined identifiers, changing them should preserve the execution of the program, which is more likely to preserve the semantics of the program. As a result, we choose these identifiers such as method names and variable names as our target identifiers to be substituted.

To facilitate the extraction, we exploit abstract syntax trees (ASTs). We use Javalang[3] to obtain ASTs for Java code, and the ast[4] lib for Python code. The identifiers are then extracted based on the node types in the ASTs; afterwards, they are put into an identifier candidate set V.

Step 2: Candidate Selection ("Selecting Candidate" in Figure 2). The size of the extracted identifier set is usually extremely large. To speed up the search for the optimal substitution identifier, for each identifier w in the program p, we construct a subset $L_w \subseteq V$, which contains K identifiers that have the shortest distance to w (cf. Line 9 in Algorithm 1). Note that here K is a hyper-parameter. (In our experiment we set $K = 5$.) Each identifier in L_w is then considered to be a candidate for the substitution of w. To obtain L_w, we train embeddings using word2vec [26] with the skip-gram algorithm (cf. Line 5 in Algorithm 1). The skip-gram algorithm is to construct word representations (i.e., word embedding) that are useful for predicting the surrounding words in a given corpus. Given a sequence of training words $w_1, \cdots, w_n$, the objective of the skip-gram is to maximize the average logarithm of the probability:

$$\frac{1}{n}\sum_{t=1}^{n}\sum_{-c\le j\le c, j\ne 0}\log p(w_{t+j}|w_t),$$

where c is the training context. Note that we use the tokens split from the program snippet rather than identifiers solely as the training corpus, and then extract the embeddings of the identifier set obtained in the previous step. For each identifier w, we select the K nearest identifiers according to the cosine distance, viz.,

$$L_w = top_K(cos(w, V'))$$

Here, V' is the set of identifiers obtained by deleting the identifiers and formal parameters that appeared in V, so we can make sure that the program after substitution is compilable. Each identifier in L_w is then considered to be a candidate for the substitution of w.

Importantly, we adopt the cosine similarity in selecting the candidate replacement. The reason is, when the identifier in the original program is substituted by one in the candidate set L_w to generate the adversarial examples, the program semantics should not be changed significantly (which implies that the generated comments should be similar for a robust model).

The following example shows that a naive approach would not serve the purpose. In this example, the original program is

```
  float avg_velocity(float distance, float time) {return distance/time;}
```

When we replace identifiers, possibly the method name "avg_velocity" is replaced by "density", and the arguments "distance" and "time" are replaced by "mass" and "volume" respectively. Namely, we obtain

```
  float density(float mass, float volume) {return mass/volume;}
```

As one can argue easily, the resulting program is quite different from the original program in semantics and thus should have a different comment. In other words, it should *not* be considered as an adversarial example. To rule out these cases, we adopt a *constrained substitution* approach. Namely, we utilize the word embedding method

[3]https://github.com/c2nes/javalang
[4]https://docs.python.org/3/library/ast.html

(word2vec in our implementation) and cosine similarity to only allow those identifies which are semantically related to the original identifies to be replaced. In this way, the obtained code snippet would be close to the original one in semantics and would be functionality preserving, and, if its comment deviates from the original comment significantly, it should be regarded as a valid adversarial example.

Step 3: Best Candidate Selection and Identifier Reranking ("Selecting Best Candidate and Reranking Identifier" in Figure 2). For each identifier w extracted from the program, we have obtained a candidate set L_w that contains K identifiers. Then we replace w with each w' in L_w and calculate the score change of the generated comment after substitution (cf. Line 9 in Algorithm 1). We define

$$w^* = \underset{w' \in L_w}{\arg\max}\{score(p) - score(p[w \leftarrow w'])\}$$

where $p[w \leftarrow w']$ is the new program obtained by replacing w with the candidate identifier $w' \in L_w$. In other words, w^* is the one which causes the most significant change and is replaced by w^* to generate a new program p^*. $score(p)$ is the output of the original deep code comment generation model by feeding the input p. For the code comment generation task, we use the BLEU score as the metric for the generated comment in natural language.

The change on the result between p and p^* represents the best attack effect that can be achieved after replacing w, i.e., $\Delta score^*_w = score(p) - score(p^*)$. For each identifier w, we iterate all candidate identifiers w^* and calculate the corresponding $\Delta score^*_w$.

A program snippet usually contains multiple identifiers, and each identifier may have different levels of contextual relation to the original program. We then adopt identifier saliency to quantify the degree of the contextual relation between the identifiers and the original program, which will be used to determine the identifier substitution order. The saliency of an identifier w with respect to a program p, i.e., $S(p, w)$, is computed as $\cos(vec(w), vec(p))$ where

$$cos(vec(w), vec(p)) = \frac{vec(w) \cdot vec(p)}{||vec(w)|| \cdot ||vec(p)||},$$

$vec(w)$ is the embedding of w, and $vec(p)$ is the contextual encoder of the program p. Here, we train an independent encoder-decoder model based on a single-layer LSTM using the two publicly available datasets, and extract the output of the encoder as the embedding of p (cf. Line 10 in Algorithm 1).

For all w extracted from p, we calculate the identifier saliency $S(p, w)$ to obtain a saliency vector $S(p)$ (cf. Line 11 in Algorithm 1). Then, for each identifier, we consider the change after substitution $\Delta score^*_w$ and the identifier saliency $S(p, w)$ to determine the order of substitution. We define a score function $H(p, p^*, w)$ to score each identifier and sort all the identifiers in p in descending order based on $H(p, p^*, w)$ (cf. Line 12 in Algorithm 1). The score function $H(p, p^*, w_i)$ is defined as

$$H(p, p^*, w) = \begin{cases} S(p, w) \cdot \Delta score^*_w & S(p, w) \neq 0, \Delta score^*_w \neq 0 \\ S(p, w) \cdot \beta & S(p, w) \neq 0, \Delta score^*_w = 0 \\ \Delta score^*_w \cdot \alpha & S(p, w) = 0, \Delta score^*_w \neq 0 \\ 0 & o.w. \end{cases}$$

where α and $\beta \in [0, 1]$ are the constant parameters.

The definition of the score function H considers both the change of the model output after identifier substitution and the importance of the substituted identifier to the original program snippet. In particular, $S(p, w)$ focuses on describing the impact of the identifier w on the original program snippet, while $\Delta score^*_w$ focuses on the impact on the model. In order to reduce the interference of the two metrics (i.e., to avoid the weighted score function vanishes when one of them vanishes), we simply take one of them when the other vanishes.

Step 4: Adversarial Example Generation ("Substituting Top max Identifiers" in Figure 2). We reorder all identifiers according to $H(p, p^*, w)$ and select the top *max* identifiers to replace (cf. Line 15-18 in Algorithm 1). To ensure that the program is compilable, we replace all occurrences where the identifier has appeared in the program. For example, if we replace 'A' with 'B' in "`void f() {int A=1; A++;}`", the new program becomes "`void f() {int B=1; B=B++;}`".

3.2 Robustness Improvement

Adversarial training aims to improve the robustness of deep learning models intrinsically. In the last few years, a variety of adversarial training methods have been proposed. In the sequel, we propose masked training, which considered to be a lightweight adversarial training method tailored to the code comment generation setting.

Algorithm 2: Masked Training Algorithm

```
Input: Code Comment Generation DataSet $D$;
The number of Masked Identifiers $Count_{masked}$;
Hyperparameter $\lambda$;
Output: Trained model $M_{masked}$;
1  Initializing the model parameters $M_{masked}$ according to the original deep code comment generation model
   training method ;
2  for batch $d$ of data $\in D$ do
3      for $(p, com) \in d$ do
4          Randomly mask $Count_{masked}$ identifiers;
5      end
6      Calculate origin loss: $L_{origin}(p, com)$;
7      Calculate masked loss: $L_{masked}(p', com)$;
8      Train the model $M_{masked}$ according to
       $\theta^\star \leftarrow \arg\min_\theta \left(\lambda * L_{origin}(p, com) + (1 - \lambda) * L_{masked}((p', com)\right)$
9  end
10 return $M_{masked}$;
```

As mentioned in Section 2.3, the low degree of robustness may be caused by the reliance on the so called non-robust features. As a result, the general idea of masked training is to reduce the dependence of the model on the non-robust features since any perturbations upon these features may cause great change on the output. Algorithm 2 illustrates the workflow of the method. Given source code p, we generate the corresponding masked code p' (cf. Line 3-5 in Algorithm 2), which is constructed by randomly replacing k identifiers in p by $< unk >$. The general objective function for the masked training is defined as

$$\theta^\star = \arg\min_\theta L(p, com),$$

where $L(p, com)$ is the negative log-likelihood

$$L(p, com) = -\frac{1}{m}\sum_{t=1}^{m} logP(com_t|com_{<t}, p)$$

In particular, we employ two objective functions to improve the robustness of the model (cf. Line 6-8 in Algorithm 2). Namely, $L_{origin}(p, com)$ which can guarantee good performance while keeping the stability of the model and $L_{masked}(p', com)$, which can guide the model to generate the output *com* according to the masked input p', making the output of the model independent of the identifiers.

Formally, given a model and the training corpus, the masked training objective is

$$\theta^{\star} = \arg\min_{\theta} \left(\lambda \cdot L_{origin}(p, com) + (1 - \lambda) \cdot L_{masked}(p', com)\right),$$

where λ is a hyperparameter.

4 EVALUATION

4.1 Experiment setup

We conduct comprehensive experiments to demonstrate the effectiveness of the proposed approach on the Java source code dataset [14] and the Python source code dataset [41], which are widely adopted benchmarks for the code comment generation task. The statistics of the two datasets are shown in Table 1. For the Java dataset, we follow the original work [14] which divided the examples into train dataset, validation dataset and test dataset in the ratio of 8:1:1. For the Python dataset, we also replicate the processing method in the original work [41] to extract the train dataset, the validation dataset and the test dataset. As a result, we obtain 50,400 examples for the train dataset, 13,248 for the validation dataset and 13,216 for the test dataset.

Table 1. Statistics of datasets

Dataset	Java	Python
Train	69,708	50,400
Validation	8,714	13,248
Test	8,714	13,216

For the Java dataset, the first summary sentence of the Javadoc annotations is usually used as the comment, which describes the functionality of the Java method. To be consistent with the original work [14], we reuse these extracted comments included in the public dataset. For the Python dataset, we use the comment provided by the source code. Data instances of these datasets are in the form of ⟨*p*, *comment*⟩ pair, where *p* is the source code snippet and *comment* is the reference comment. We pre-process the dataset by the Javalang [3] parser for the Java dataset and the ast[4] library for the Python dataset, and discard those syntactically incorrect programs. Finally, we follow the processing steps [1] which splits camelCase and snake_case tokens into their corresponding sub-tokens.

Victim models. The victim models (i.e., the target models under adversarial attacks) in our experiments are based on LSTM, Transformer, GNN, a dual model (CSCG), and a retrieval-based neural source code summarization model named Rencos.

- LSTM-based seq2seq model. A LSTM-based seq2seq model [1] contains 2-layers BiLSTM for encoder and decoder with attention mechanism, encoding the source code to an intermediate representation and translating it to natural language, i.e., comment.
- Transformer-based seq2seq model. Ahmad et al. [1] designed the Transformer-based seq2seq model for code comment generation by introducing multi-head attention as encoder and decoder. To the best of our knowledge, this model represents the state-of-the-art result on the Java dataset.
- GNN-based seq2seq model. LeClair et al. [2] employed two encoders, one is the GNN-based encoder to model structural information and the other is the GRU-based encoder to model textual information and GRU-based decoder to generate natural language comment.
- CSCG Dual model. Wei et al. [45] designed a dual learning framework to train a code summary i.e comment generation and a code generation model simultaneously using the LSTM-based seq2seq model.
- Retrieval-based model (Rencos). Zhang et al. [51] leverage both neural and retrieval-based techniques to enhance the neural model with the most similar code snippets at the syntax-level and the semantics-level.

We largely follow the settings of the respective original work; in particular, the hyperparameters of the victim models are listed in Table 2. All models were trained and evaluated on a server running Ubuntu 20.04 LTS OS with 2 Intel Xeon 4216 2.10GHz Silver CPUs , and 4 RTX2080Ti GPUs.

Table 2. Hyperparameters in our experiments

Hyperparameters	**LSTM**	**Transformer**	**GNN**	**CSCG**	**Rencos**
n_layers	2	6	1	3	1
n_head	–	8	–	–	–
d_k, d_v	–	64	–	–	–
d_ff	–	2048	–	–	–
embed_size	512	512	256	512	256
hidden_size	512	–	256	512	512
optimizer	adam	adam	adam	adam	adam
learning rate	0.002	0.0001	0.001	0.002	0.001
batch size	32	32	32	32	32

Baseline approaches for adversarial attack. Since we are the first to consider adversarial examples for code comment generation tasks, the literature is short of algorithms for direct comparison. To demonstrate the effectiveness of our approach, we adopt two algorithms as the baseline, i.e., the random substitute algorithm and the algorithm based on Metropolis-Hastings sampling [49].

Random substitution. The random substitute algorithm is a naïve algorithm where both the substituted identifiers and candidate identifiers are randomly sampled.

Metropolis-Hastings algorithm. The Metropolis-Hastings sampling based algorithm was recently used to generate adversarial examples for attacking source code classifiers [49]. Recall that the Metropolis-Hastings algorithm is a classical Markov Chain Monte Carlo sampling approach, which can generate desirable examples given the targeted stationary distribution and the transition proposal. We adapt the algorithm [9] to our code comment generation task.

In general, we want the adversarial examples to be as close to the original example as possible. For this purpose, we set *max*, the maximum number of identifiers that can be substituted. (In the current experiments we set *max* to be 2 or 3.)

Metrics. As the generated comments are in natural language, we adopt the standard metrics from neural machine translation, i.e., BLEU, METEOR, ROUGE-L to measure the quality of the generated comments. The lower these values are after attack, the higher the degradation of comment generation models is, i.e., the less robust these models are. Moreover, we introduce three additional metrics to evaluate the performance of different adversarial example generation algorithms.

- Relative degradation. We follow Michel et al.'s work [25] to measure the (relative) degradation of the model under attack. Formally,

$$r_d = \frac{\text{BLEU}(y, \text{refs}) - \text{BLEU}(y', \text{refs})}{\text{BLEU}(y, \text{refs})},$$

where refs denotes the reference comment, y is the original output, and y' is output of the perturbed program.

- Valid rate, which is defined as the percentage of generated adversarial examples which can pass the compilation. Formally,
$$v_r = \frac{Count_{valid}}{Count_{all}}.$$
This metric is used to assess the quality of the generated adversarial examples, as well as the efficiency of the generation process.
- Success rate, which is defined as the product of the relative degradation and the valid rate, providing a comprehensive indicator of attack efficiency and example quality. Formally,
$$s_r = r_d * v_r.$$
Essentially a higher success rate indicates the corresponding method can generate valid adversarial examples with better attack capability, hence entails a more effective attack method.

4.2 Research questions and results

In our experiments, we primarily investigate the following four research questions (RQs).

RQ1. Are existing code comment generation models vulnerable to our adversarial attacks?
RQ2. How effective is our adversarial attack method, i.e., how successful can it achieve to attack code comment generation models over the baseline methods?
RQ3. Do adversarial samples generated by our adversarial attack method have better transferability than the baseline methods?
RQ4. How efficient is the masked training method in improving robustness?

RQ1. Are existing code comment generation models vulnerable to our adversarial attacks?
To answer this research question, we generate adversarial examples on the test dataset using ACCENT to attack four different models. The performance of different models before and after the attack is listed in Table 3.

We can observe that all code comment generation models are vulnerable to our adversarial attack. When modifying maximum 2 or 3 identifiers in the source code, the performance of models degrades sharply in general, although the impact of the adversarial attack differs among these models. The CSCG Dual model has the worst performance on the two datasets. When we test the CSCG Dual model with $max = 2$, the BLEU value is only 8.85 on the Java dataset and 11.90 on the Python dataset which means that the model's output is almost meaningless and of little help to program comprehension. The retrieval-based model Rencos performs better under the adversarial attack. From Table 3 we can also see that models which are with the structural information (GNN-based seq2seq model) or with the help of most similar code snippets (Rencos) are more robust than models with only contextual information (LSTM-based, Transformer-based and CSCG Dual models).

To summarize, existing code comment generation models are of poor robustness under adversarial attacks, especially the seq2seq models with only contextual information.

RQ2. How effective is our adversarial attack method, i.e., how successful can it achieve to attack code comment generation models over the baseline methods?
For this research question, we analyze the effectiveness of different algorithms on the five models across two datasets with $max = 2$ and $max = 3$. The results are given in Table 4, Table 5, and Table 6. As the input of the GNN-based model and the retrieval-based model need to be compiled to generate AST, only a small part of the samples generated by the random substitution algorithm are valid samples (i.e., can be compiled), hence only the MH-based method and the ACCENT attack method are compared in the two models. In other models, baseline algorithms contain random substitution algorithm, MH-based algorithm, and our ACCENT attack method. Taking the original models' BLEU as the standard performance metric, the ACCENT attack method can reduce the performance by 63.12% for LSTM, 70.32% for Transformer, 58.28% for GNN, 79.12% for CSCG and 7.55% for Rencos

Table 3. Results of adversarial attack on different models ('*max*' means the maximum substitution identifier number used in different methods; 'original' is the result on the clean test set.)

		Java Dataset			Python Dataset		
		BLEU	**METEOR**	**ROUGE-L**	**BLEU**	**METEOR**	**ROUGE-L**
LSTM	original	35.47	19.72	47.57	30.83	17.06	41.77
	attack max=2	13.08	6.83	21.75	18.30	8.62	27.24
	attack max=3	13.07	6.75	21.52	17.92	8.27	26.64
Transformer	original	44.58	26.43	54.76	33.15	18.96	44.50
	attack max=2	13.23	8.08	22.79	18.87	8.99	27.91
	attack max=3	13.14	7.90	22.42	18.54	8.57	27.29
GNN	original	39.41	23.32	46.65	31.24	15.77	38.28
	attack max=2	16.44	7.77	21.36	19.38	7.36	24.07
	attack max=3	16.14	7.42	20.55	18.65	6.59	22.72
CSCG	original	42.39	25.77	53.61	30.82	17.67	48.14
	attack max=2	8.85	5.10	23.50	11.90	6.23	32.94
	attack max=3	8.95	5.02	23.44	12.08	6.15	32.83
Rencos	original	44.0	25.73	54.02	33.34	18.65	43.37
	attack max=2	40.68	23.09	49.17	31.02	15.78	38.84
	attack max=3	40.23	22.69	48.46	30.55	15.19	37.90

on the Java dataset, and 40.64% for LSTM, 43.08% for Transformer, 37.80% for GNN, 37.77% for CSCG and 6.96% for Rencos with $max = 2$, which are considerably better than the baselines. When max is 3, our attacking method can degrade the model performance even further. The effectiveness of the adversarial samples generated by the random substitution algorithm is extremely low, while the adversarial samples generated by the ACCENT attack method and the MH-based algorithm can guarantee 100% effectiveness. That means they are all correct code snippets in grammar, and the ACCENT attack method can achieve a higher success rate.

To further investigate the effectiveness of our approach, we apply the Mann-Whitney U test. Particularly, we compare ACCENT attack method with MH, and test whether the effectiveness of the former is significantly better than the latter. We focus on the r_d values of ACCENT and MH. For each run, we randomly sample 100 Java

Table 4. Evaluation of different adversarial examples generation algorithms

		Java Dataset			**Python Dataset**		
		max=2					
		r_d(%)	v_r(%)	s_r(%)	r_d(%)	v_r(%)	s_r(%)
LSTM	Random	31.38	30.82	9.67	23.39	27.48	6.43
	MH	42.09	100	42.09	40.29	100	40.29
	ACCENT	63.12	100	63.12	40.64	100	40.64
Transformer	Random	38.58	30.82	11.89	22.29	27.48	6.13
	MH	64.72	100	64.72	41.45	100	41.45
	ACCENT	70.32	100	70.32	43.08	100	43.08
GNN	Random	–	–	–	–	–	–
	MH	57.14	100	57.14	34.41	100	34.41
	ACCENT	58.28	100	58.28	37.80	100	37.80
CSCG	Random	44.44	30.82	13.69	23.56	27.48	6.47
	MH	68.25	100	68.25	37.77	100	37.77
	ACCENT	79.12	100	79.12	61.39	100	61.39
Rencos	Random	–	–	–	–	–	–
	MH	6.84	100	6.84	7.11	100	7.11
	ACCENT	7.55	100	7.55	6.96	100	6.96
		max=3					
		r_d(%)	v_r(%)	s_r(%)	r_d(%)	v_r(%)	s_r(%)
LSTM	Random	32.03	29.7	9.51	21.89	27.81	6.08
	MH	44.60	100	44.60	38.63	100	38.63
	ACCENT	63.15	100	63.15	41.87	100	41.87
Transformer	Random	45.76	29.7	13.59	25.52	27.81	7.09
	MH	66.42	100	66.42	42.29	100	42.29
	ACCENT	70.52	100	70.52	44.07	100	44.07
GNN	Random	–	–	–	–	–	–
	MH	58.51	100	58.51	36.33	100	36.33
	ACCENT	59.05	100	59.05	40.30	100	40.30
CSCG	Random	46.36	29.7	13.76	25.44	27.81	7.07
	MH	68.88	100	68.88	39.00	100	39.00
	ACCENT	78.89	100	78.89	60.80	100	60.80
Rencos	Random	–	–	–	–	–	–
	MH	7.61	100	7.61	8.34	100	8.34
	ACCENT	8.57	100	8.57	8.37	100	8.37

code snippets, and 100 Python code snippets from the two datasets, and calculate the average r_d values of the generated comments by the two attack methods on the five base models as outcomes. The experiment is repeated 5 times with max=2 and max=3 respectively. As a result, there are in total of 20 experiments (i.e., max=2 or 3 for five based models and for Java and Python datasets). For each one of them, we obtain two samples of size 5. In the hypothesis test, we follow the convention to set $\alpha = 0.05$. For the Mann-Whitney U test, a majority of p-values (15 out of 20) are less than 0.05 (typically 0.005), which indicates that the improvements are statistically significant at

Table 5. Results of adversarial attack using Random substitution algorithm and MH-based algorithm on different models with *max* = 2

		Java Dataset			**Python Dataset**		
		BLEU	**METEOR**	**ROUGE-L**	**BLEU**	**METEOR**	**ROUGE-L**
LSTM	Random	24.34	13.45	35.43	23.62	11.69	32.94
	MH	20.54	10.81	30.3	18.41	8.77	27.85
Transformer	Random	27.38	15.61	37.26	25.76	13.35	35.81
	MH	15.73	9.68	26.74	19.41	9.98	30.19
GNN	Random	–	–	–	–	–	–
	MH	16.89	8.38	22.13	20.49	8.10	25.21
CSCG	Random	23.55	13.50	39.65	23.56	12.08	40.23
	MH	13.46	7.72	29.65	19.18	9.79	37.01
Rencos	Random	–	–	–	–	–	–
	MH	40.99	23.44	49.84	30.97	15.48	38.27

Table 6. Results of adversarial attack using Random substitution algorithm and MH-based algorithm on different models with *max* = 3

		Java Dataset			**Python Dataset**		
		BLEU	**METEOR**	**ROUGE-L**	**BLEU**	**METEOR**	**ROUGE-L**
LSTM	Random	24.11	13.24	35.15	24.08	12.09	33.52
	MH	19.65	10.2	28.99	18.92	9.23	28.64
Transformer	Random	24.18	19.72	42.13	24.69	12.60	34.76
	MH	14.97	9.17	25.59	18.80	9.27	29.04
GNN	Random	–	–	–	–	–	–
	MH	16.35	7.70	20.95	19.89	7.37	24.17
CSCG	Random	22.74	12.96	38.99	22.98	11.54	39.49
	MH	13.19	7.28	28.83	18.80	9.35	36.30
Rencos	Random	–	–	–	–	–	–
	MH	40.65	23.19	49.42	30.56	14.99	37.57

the confidence level of 95%.[5] To conclude, the adversarial samples generated by ACCENT are effective, and our attack method is superior to the baseline methods.

RQ3. Do adversarial samples generated by our adversarial attack method have better transferability than the baseline methods?

Adversarial example generated for a certain model is considered to be transferable if it can successfully attack other DNN models. To answer this research question, we tested the transferability of the adversarial examples generated by our ACCENT attack method and compared them with the MH-based algorithm. The experiment uses a cross-testing method, that is, among the five models, we use the adversarial samples generated from one model

[5]The details of the samples can be retrieved in our replication package.

Table 7. BLEU scores of different algorithms for transferability on Java dataset (max = 2).

		CSCG	LSTM	Transformer	Rencos
Adversairal examples generated for GNN	MH	17.69	18.91	22.26	43.37
	ACCENT	16.22	18.10	21.59	43.39
		GNN	LSTM	Transformer	Rencos
Adversairal examples generated for CSCG	MH	15.78	17.37	20.50	43.43
	ACCENT	15.31	15.84	19.06	43.32
		GNN	CSCG	Transformer	Rencos
Adversairal examples generated for LSTM	MH	19.39	17.91	23.89	43.44
	ACCENT	16.05	14.48	19.10	43.39
		GNN	CSCG	LSTM	Rencos
Adversairal examples generated for Transformer	MH	16.11	15.07	16.59	43.43
	ACCENT	15.92	14.42	15.83	43.35
		GNN	CSCG	LSTM	Transformer
Adversairal examples generated for Rencos	MH	17.69	19.95	21.28	24.46
	ACCENT	17.50	18.85	20.26	24.12

Table 8. BLEU scores of different algorithms for transferability on Java dataset (max = 3).

		CSCG	LSTM	Transformer	Rencos
Adversairal examples generated for GNN	MH	16.33	18.39	20.76	43.37
	ACCENT	15.30	17.38	20.45	43.45
		GNN	LSTM	Transformer	Rencos
Adversairal examples generated for CSCG	MH	15.58	16.75	29.12	43.36
	ACCENT	15.25	15.35	18.27	43.27
		GNN	CSCG	Transformer	Rencos
Adversairal examples generated for LSTM	MH	19.68	19.52	22.28	43.42
	ACCENT	15.86	13.93	18.27	43.40
		GNN	CSCG	LSTM	Rencos
Adversairal examples generated for Transformer	MH	15.81	14.71	16.00	43.40
	ACCENT	15.87	14.10	15.32	43.37
		GNN	CSCG	LSTM	Transformer
Adversairal examples generated for Rencos	MH	16.82	17.87	19.26	22.24
	ACCENT	15.19	16.71	18.48	21.70

to attack the other four models. For example, the adversarial examples generated from the Transformer-based model are used to attack the LSTM-based, GNN-based, CSCGDual models and Rencos. The BLEU scores on the Java and the Python datasets are shown in Figure 3–6 and Table 7–10.

It can be observed from Figure 3–Figure 7 that, except for the Rencos model, the r_d values of the other four models after attacks are decreased by 50% for the Java dataset, and 37% for the Python dataset, which means that the performance of the model has dropped greatly, that is, the adversarial samples generated by our ACCENT attack method can be successfully transferred to other models. At the same time, we can see that, compared with the MH-based algorithm, the adversarial samples generated by the ACCENT attack method have better transferability, as the r_d of the ACCENT attack method is greater than the r_d of the MH-based algorithm on

Table 9. BLEU scores of different algorithms for transferability on Python dataset (max = 2).

		CSCG	LSTM	Transformer	Rencos
Adversairal examples generated for GNN	MH	22.65	22.63	24.17	32.76
	ACCENT	21.49	22.79	24.32	32.54
		GNN	LSTM	Transformer	Rencos
Adversairal examples generated for CSCG	MH	20.15	22.38	23.49	32.88
	ACCENT	16.92	22.96	21.35	30.44
		GNN	CSCG	Transformer	Rencos
Adversairal examples generated for LSTM	MH	20.18	21.74	22.81	32.75
	ACCENT	19.04	20.17	22.36	32.28
		GNN	CSCG	LSTM	Rencos
Adversairal examples generated for Transformer	MH	20.27	21.99	21.39	32.82
	ACCENT	18.89	20.10	20.91	32.28
		GNN	CSCG	LSTM	Transformer
Adversairal examples generated for Rencos	MH	20.58	22.67	22.85	23.95
	ACCENT	19.69	22.18	22.92	24.71

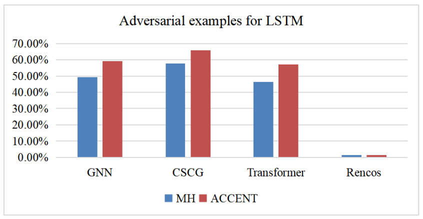
Table 10. BLEU scores of different algorithms for transferability on Python dataset (max = 3).

		CSCG	LSTM	Transformer	Rencos
Adversairal examples generated for GNN	MH	21.86	21.81	23.06	32.64
	ACCENT	20.46	21.66	22.90	32.46
		GNN	LSTM	Transformer	Rencos
Adversairal examples generated for CSCG	MH	19.88	21.49	22.60	32.79
	ACCENT	16.61	20.57	21.94	30.46
		GNN	CSCG	Transformer	Rencos
Adversairal examples generated for LSTM	MH	20.06	19.78	21.80	32.68
	ACCENT	18.65	19.42	21.32	32.25
		GNN	CSCG	LSTM	Rencos
Adversairal examples generated for Transformer	MH	20.19	19.92	21.49	32.69
	ACCENT	18.46	19.40	20.50	32.17
		GNN	CSCG	LSTM	Transformer
Adversairal examples generated for Rencos	MH	19.81	21.95	21.92	22.98
	ACCENT	18.98	21.05	21.66	23.07

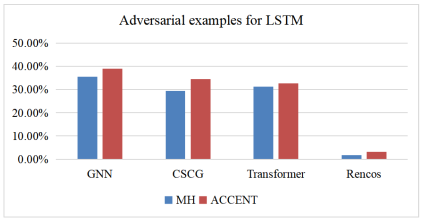
two datasets for all models. This demonstrates that our method can successfully find those identifiers that are important and effective across different models.

RQ4. How efficient is the masked training method in improving robustness?

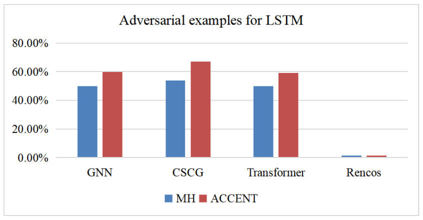
We evaluate the effectiveness of our masked training method in improving robustness, which can be evaluated by the changes in performance metrics of DNNs. For each model, we report these metrics (i.e., BLEU, METEOR, and ROGUE-L) over the original test dataset without any perturbations (i.e., 'Clean') and the adversarial examples generated by our ACCENT method with max = 2 and max = 3 (i.e., 'Adv'). We compare the performance of our masked training method with data augmentation, which is a commonly adopted robustness improvement method. In a nutshell, data augmentation improves the robustness by re-training the model with the mixed adversarial



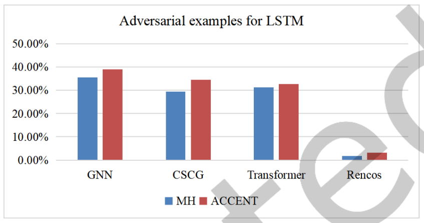
(a) Java Dataset with $max = 2$.



(b) Python Dataset with $max = 2$.

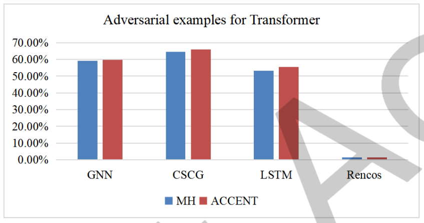


(c) Java Dataset with $max = 3$.

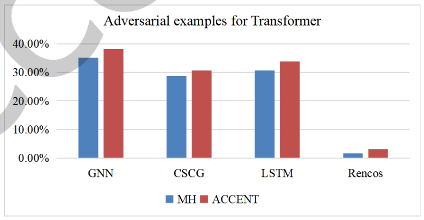


(d) Python Dataset with $max = 3$.

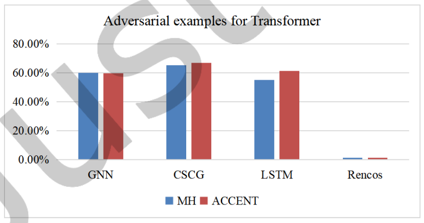
Fig. 3. The transferability of adversarial examples generated by different algorithms on LSTM model: the values r_d are tested by attacking GNN, CSCG, Transformer and Rencos model.



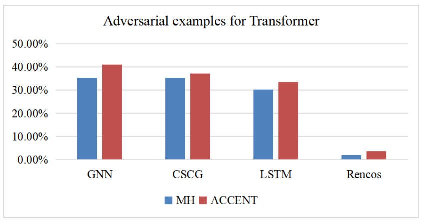
(a) Java Dataset with $max = 2$.



(b) Python Dataset with $max = 2$.



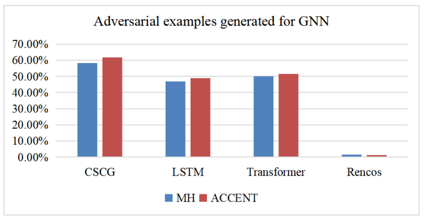
(c) Java Dataset with $max = 3$.



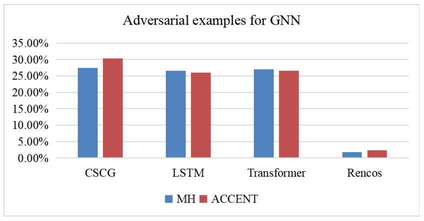
(d) Python Dataset with $max = 3$.

Fig. 4. The transferability of adversarial examples generated by different algorithms on Transformer model: the values r_d are tested by attacking GNN, CSCG, LSTM and Rencos model.

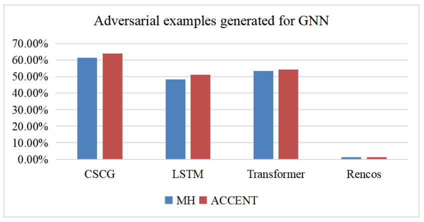
dataset which combines the original training dataset with adversarial examples. Table 11 compares the results of our method and the baseline. 'Normal' represents the model through the standard training process; 'Aug'



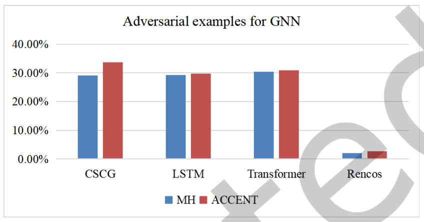
(a) Java Dataset with $max = 2$.



(b) Python Dataset with $max = 2$.

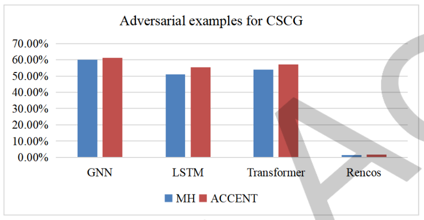


(c) Java Dataset with $max = 3$.

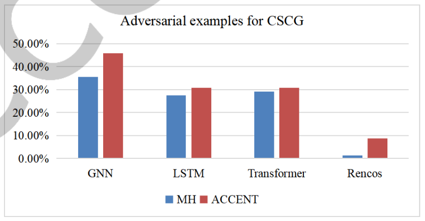


(d) Python Dataset with $max = 3$.

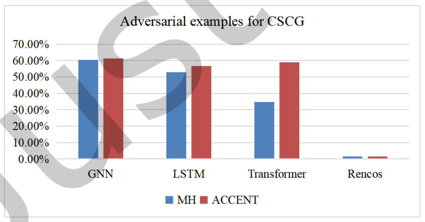
Fig. 5. The transferability of adversarial examples generated by different algorithms on GNN model: the values r_d are tested by attacking CSCG, LSTM, Transformer and Rencos model.



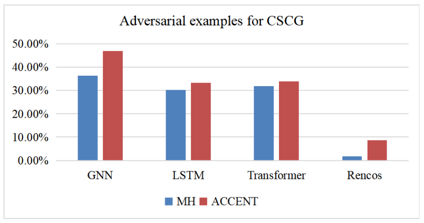
(a) Java Dataset with $max = 2$.



(b) Python Dataset with $max = 2$.



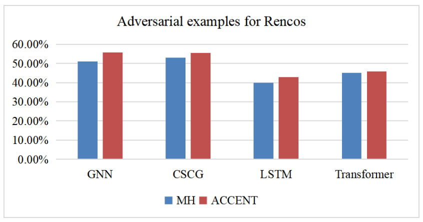
(c) Java Dataset with $max = 3$.



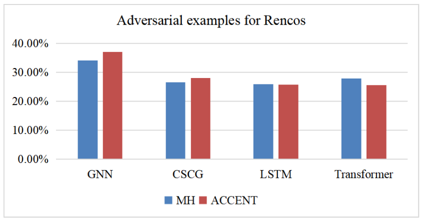
(d) Python Dataset with $max = 3$.

Fig. 6. The transferability of adversarial examples generated by different algorithms on CSCG model: the values r_d are tested by attacking GNN, LSTM, Transformer and Rencos model.

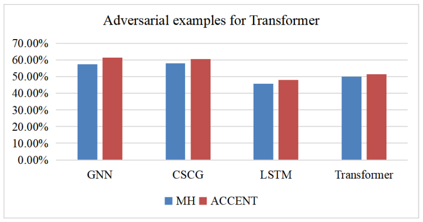
represents the model trained by data augmentation and 'Maksed' represents the model trained by our masked training method.



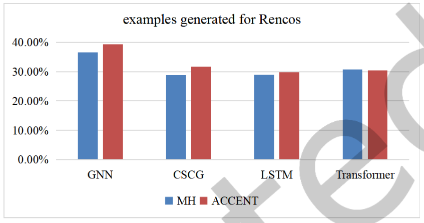
(a) Java Dataset with $max = 2$.



(b) Python Dataset with $max = 2$.



(c) Java Dataset with $max = 3$.



(d) Python Dataset with $max = 3$.

Fig. 7. The transferability of adversarial examples generated by different algorithms on Rencos model: the values r_d are tested by attacking GNN, LSTM CSCG and Transformer model.

Improving robustness may sacrifice the accuracy of the models on the clean dataset [6, 11, 23, 39]. From Table 11 we can observe that, as the robustness of the model increases, the original accuracy of the model does decrease. However, our masked training method has less impact on accuracy, where the data augmentation method may suffer from a significant drop. Furthermore, our training method can increase the accuracy of some models on the clean datasets (4 out of 10). While the accuracy of some models on the clean dataset may slightly decrease, the accuracy on the adversarial examples is improved through our masked training. For example, on the Java dataset, the performance of the Transformer-based model after the masked training has increased to 40.10 and 39.24 on the adversarial examples with $max = 2$ and $max = 3$ respectively, while the data augmentation method improves the performance to 18.10 and 17.82 respectively.

From Table 11, we can conclude that the masked training method can significantly boost the robustness across different models at the same time maintain fairly good performance on the test dataset.

4.3 Human evaluation

To complement the above objective metrics, we also conduct a human evaluation to further assess the quality of the comments generated by the masked training method, data augmentation and normal training method. Generally, we follow the evaluation settings from the previous work [17, 46]. Particularly, the comments are examined from three aspects, i.e., similarity, naturalness, and informativeness [46]. Similarity refers to how similar the generated comment is to the reference comment; naturalness measures the grammaticality and fluency; informativeness focuses on the content delivery from code snippet to the generated comments. For each of the five base models, we randomly select 20 Java code snippets and 20 Python code snippets respectively, and use the two adversarial training methods to generate comments. We obtain 600 generated comments and 200 references in total. To facilitate comparison, for each code snippet we construct a tuple consisting of a reference and three generated comments; we obtain 200 tuples accordingly.

We ask six graduate students studying in the Software Engineering programme to participate in the evaluation, all of whom have at least three years of programming experience in both Java and Python, and are professionally proficient in English. The subjects are evenly divided into two groups each of which has 3 students. The 200 comment triples are also evenly divided to two parts and assigned to the two groups randomly, with each group of 100 triples. Participants manually inspect the 100 generated comment triples as well as the code snippets, and rate them independently, which means that each comment triple is examined by three individuals. The grades are given in the Likert scale ranging from 1 to 5, corresponding to 'very poor', 'poor', 'neutral', 'good', 'very good' respectively where a higher value indicates a better quality. To be fair, the labels of the generator information in the triples are removed. Table 12 and Table 13 show the statistics of the collected results. We can observe that, on both datasets and for all the three aspects, the average scores of the comments generated by the masked training methods are consistently higher than those generated by the data augmentation method and normal training method. Moreover, a majority of comments generated by the masked training method, receive scores above 3.

4.4 Examples

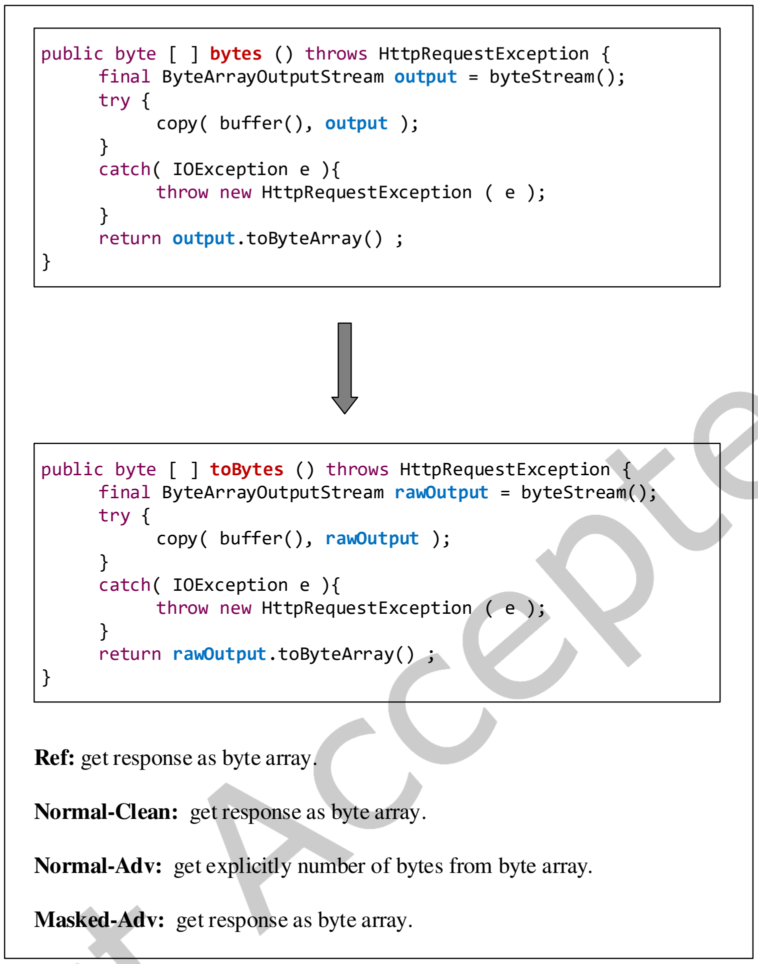
For qualitative analysis, Figure 8–Figure 11 show some examples where 'Ref' refers to the reference comment, 'Normal-Clean' refers to the result of the clean example on the standard training model, 'Nor-Adv' refers to the result of the adversarial example on the standard training model, and 'Masked-Adv' refers to the result of the adversarial example on the masked training model.

We can see that, the adversarial examples generated by ACCENT are very similar to the original code snippet, indicating that our approach can generate high-quality adversarial examples preserving original syntax, semantics and functionality. We also find that, although the standard training model ('Normal-Clean') performed well on original examples, the quality of the generated comments on the adversarial examples are poor ('Normal-AdV'). On the other hand, the masked training method can effectively defense against attacks and generate the closest comment ('Masked-Adv') to the reference ('Ref').

5 THREATS TO VALIDITY

Threats to internal validity are related to internal factors that could have influenced the results. One threat that may make the results statistically unstable is the randomness from Step 2 in the adversarial attack method. In this step, we randomly select K candidates for each identifier. To mitigate this, we randomly sample the K candidates several times and have confirmed that our method outperforms the baselines consistently. Another threat is related to the errors introduced in the implementation. To minimize these, we have double-checked and peer-reviewed our code and repeatedly conducted the baseline methods to ensure the fairness of the results.

External validity concerns the generalizability of the results on the datasets other than the ones used in the experiments [10]. Indeed, in our approach, we only focus on whether code comment generation models are vulnerable to adversarial examples and how to improve the robustness of different models for Java and Python methods. However, our approach is essentially independent of specific programming languages. Note that in the adversarial attack method we intend to find the most important tokens with respect to the model, and in the masked training, we only mask the programmer-defined identifiers in the method. Both of them can be easily applied to other datasets. Another threat originates from replacing the programmer-defined identifiers with meaningless labels as done in CODENN [17], which could invalidate ACCENT. However, in general, most of the work includes the programmer-defined identifiers as part of the input to the deep code comment generation model.



```
public byte [ ] bytes () throws HttpRequestException {
      final ByteArrayOutputStream output = byteStream();
      try {
            copy( buffer(), output );
      }
      catch( IOException e ){
            throw new HttpRequestException ( e );
      }
      return output.toByteArray() ;
}
```

```
public byte [ ] toBytes () throws HttpRequestException {
      final ByteArrayOutputStream rawOutput = byteStream();
      try {
            copy( buffer(), rawOutput );
      }
      catch( IOException e ){
            throw new HttpRequestException ( e );
      }
      return rawOutput.toByteArray() ;
}
```

Ref: get response as byte array.

Normal-Clean: get response as byte array.

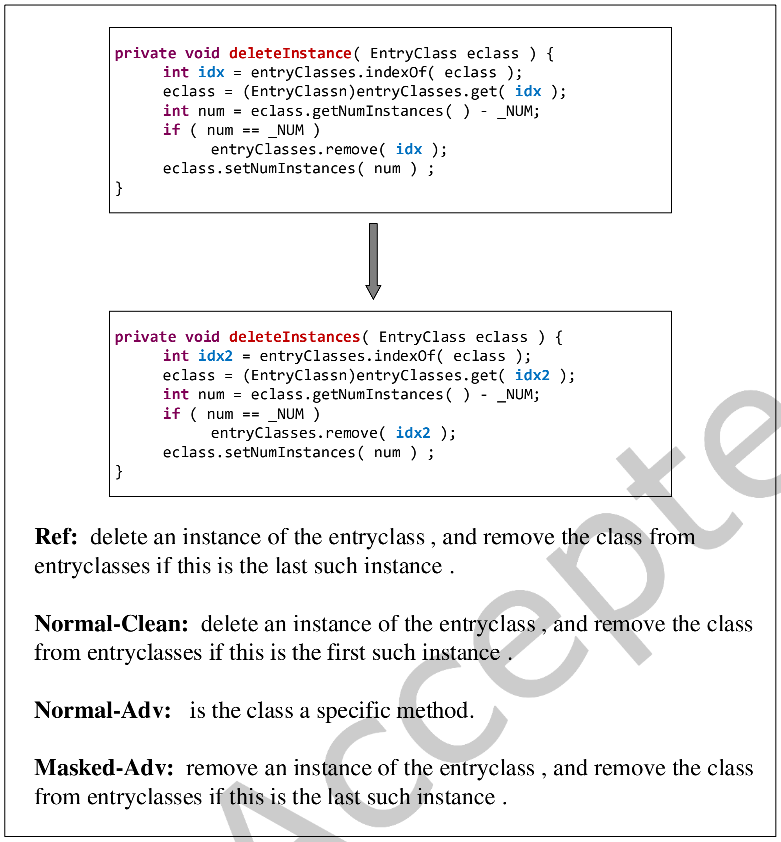
Normal-Adv: get explicitly number of bytes from byte array.

Masked-Adv: get response as byte array.

Fig. 8. Examples and corresponding adversarial examples generated by ACCENT, where 'Ref' is the reference comment, 'Normal-Clean' is the result of the clean example on the standard training model, 'Nor-Adv' is the result of the adversarial example on the standard training model and 'Masked-Adv' is the result of the adversarial example on the masked training model.

6 RELATED WORK

Code Comment Generation. Code comment generation is an essential part of the software development cycle and has attracted significant attention. Neural network based approaches have been applied to this task, which is the main focus of the discussion. Alllamanis et al. [3] adopted convolutional attention neural network to generate short and name-like comments. More recent work casts it as a seq2seq generation task and employs the encoder-decoder model as the basic architecture. Based on different code representations, these approaches can be divided into two categories, i.e., token sequence based and tree structural based approaches.



```
private void deleteInstance( EntryClass eclass ) {
     int idx = entryClasses.indexOf( eclass );
     eclass = (EntryClassn)entryClasses.get( idx );
     int num = eclass.getNumInstances( ) - _NUM;
     if ( num == _NUM )
          entryClasses.remove( idx );
     eclass.setNumInstances( num ) ;
}
```

```
private void deleteInstances( EntryClass eclass ) {
     int idx2 = entryClasses.indexOf( eclass );
     eclass = (EntryClassn)entryClasses.get( idx2 );
     int num = eclass.getNumInstances( ) - _NUM;
     if ( num == _NUM )
          entryClasses.remove( idx2 );
     eclass.setNumInstances( num ) ;
}
```

Ref: delete an instance of the entryclass , and remove the class from entryclasses if this is the last such instance .

Normal-Clean: delete an instance of the entryclass , and remove the class from entryclasses if this is the first such instance .

Normal-Adv: is the class a specific method.

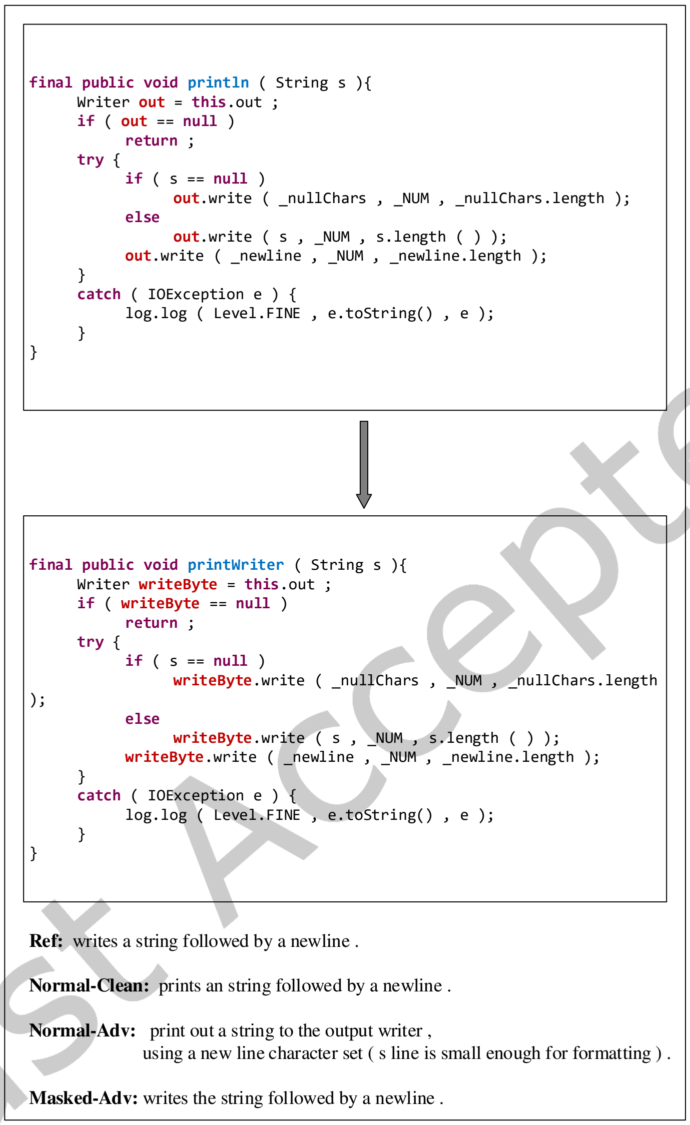
Masked-Adv: remove an instance of the entryclass , and remove the class from entryclasses if this is the last such instance .

Fig. 9. An example and corresponding adversarial example generated by ACCENT, where 'Ref' is the reference comment, 'Normal-Clean' is the result of the clean example on the standard training model, 'Nor-Adv' is the result of the adversarial example on the standard training model and 'Masked-Adv' is the result of the adversarial example on the masked training model.

For token sequence based approaches, Iyer et al. [17] presented an end-to-end neural attention model using LSTMs to generate comments for C# and SQL language. Hu et al. [14] adopted a transfer learning method utilizing API information to comment generation. Wei et al. [45] utilized dual learning to train a code comment generation model and code generation model simultaneously. Ahmad et al. [1] adopted Transformer with absolute position encoding to comment generation.

For tree structural based approaches, the general methodology is to encode the code as (variants of) ASTs which are input to purpose-designed neural networks. Hu et al. [13] proposed a structure based traversal method to flatten the AST. LeClair et al. [21] aimed to combine words from code with code structure from AST. Furthermore, techniques have been put forward to enhance the performance, e.g., approaches based on reinforcement learning [41] or aided with contextual information [52].

Adversarial Examples Generation. Adversarial examples were first proposed by Szegedy et al. in image classification [38]. Their experiment shows that an imperceptible perturbation of the benign input image could



```
final public void println ( String s ){
      Writer out = this.out ;
      if ( out == null )
            return ;
      try {
            if ( s == null )
                  out.write ( _nullChars , _NUM , _nullChars.length );
            else
                  out.write ( s , _NUM , s.length ( ) );
            out.write ( _newline , _NUM , _newline.length );
      }
      catch ( IOException e ) {
            log.log ( Level.FINE , e.toString() , e );
      }
}
```

```
final public void printWriter ( String s ){
      Writer writeByte = this.out ;
      if ( writeByte == null )
            return ;
      try {
            if ( s == null )
                  writeByte.write ( _nullChars , _NUM , _nullChars.length
);
            else
                  writeByte.write ( s , _NUM , s.length ( ) );
            writeByte.write ( _newline , _NUM , _newline.length );
      }
      catch ( IOException e ) {
            log.log ( Level.FINE , e.toString() , e );
      }
}
```

Ref: writes a string followed by a newline .

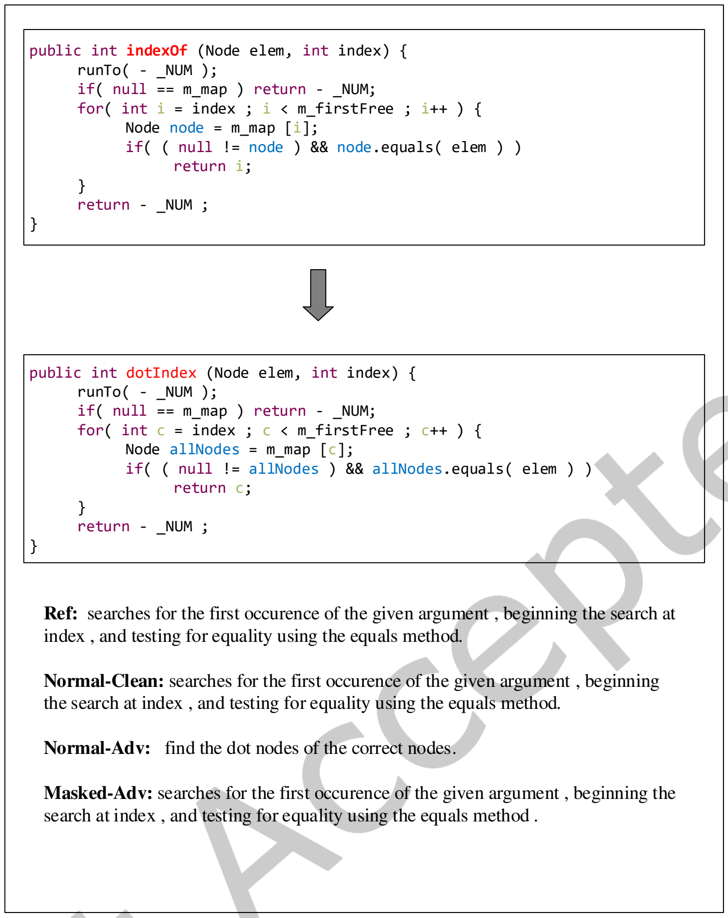
Normal-Clean: prints an string followed by a newline .

Normal-Adv: print out a string to the output writer ,
using a new line character set (s line is small enough for formatting) .

Masked-Adv: writes the string followed by a newline .

Fig. 10. An example and corresponding adversarial example generated by ACCENT, where 'Ref' is the reference comment, 'Normal-Clean' is the result of the clean example on the standard training model, 'Nor-Adv' is the result of the adversarial example on the standard training model and 'Masked-Adv' is the result of the adversarial example on the masked training model.

cause misclassification. A plethora of generation methods have been studied for image classification, a thorough survey of which is clearly out of the scope of the current paper. Here we only mention some representational work such as FGSM [12], Deepfool [27], BIM [20], JSMA [30], and the C&W method [8].



```
public int indexOf (Node elem, int index) {
      runTo( - _NUM );
      if( null == m_map ) return - _NUM;
      for( int i = index ; i < m_firstFree ; i++ ) {
            Node node = m_map [i];
            if( ( null != node ) && node.equals( elem ) )
                  return i;
      }
      return - _NUM ;
}
```

```
public int dotIndex (Node elem, int index) {
      runTo( - _NUM );
      if( null == m_map ) return - _NUM;
      for( int c = index ; c < m_firstFree ; c++ ) {
            Node allNodes = m_map [c];
            if( ( null != allNodes ) && allNodes.equals( elem ) )
                  return c;
      }
      return - _NUM ;
}
```

Ref: searches for the first occurence of the given argument , beginning the search at index , and testing for equality using the equals method.

Normal-Clean: searches for the first occurence of the given argument , beginning the search at index , and testing for equality using the equals method.

Normal-Adv: find the dot nodes of the correct nodes.

Masked-Adv: searches for the first occurence of the given argument , beginning the search at index , and testing for equality using the equals method .

Fig. 11. Examples and corresponding adversarial examples generated by ACCENT, where 'Ref' is the reference comment, 'Normal-Clean' is the result of the clean example on the standard training model, 'Nor-Adv' is the result of the adversarial example on the standard training model and 'Masked-Adv' is the result of the adversarial example on the masked training model.

Our work is more related to adversarial example generation in the NLP area which turns out to be more challenging although the underlying principles are somewhat similar. Natural language texts are discrete and are more difficult to be perturbed in a meaningful way. Papernot et al. [31] first studied the problems of adversarial examples in text by adopting FGSM. Semanta et al. [36] combined FGSM and importance of word to select the top-k words with highest importance to attack the text classification model. Similarly, Ren et al. [35] proposed PWWS which based on word saliency to attack the text classification model. Jia et al. [19] added sentences to the ends of paragraphs using crowdsourcing to fool reading comprehension system. Belinkov et al. [6] devised adversarial examples depending on natural and synthetic language errors which can fool Neural Machine Translation (NMT) system.

Comparing to the large body of work on adversarial examples for image and NLP, the corresponding work for source code processing is in its infancy; this is especially the case for comment generation.

Bielik et al. [7] improved the adversarial robustness of models for the task of type inference by learning to abstain if uncertain. Zhang et al. [49] studied the problem for the code classification tasks where they proposed a sampling based method to generate adversarial examples. Note that this work is still of classification nature whereas our work focuses on comment generation, which is of language *translation* or *generation* nature. Yefet et al. [47] generated adversairal example based on gradient for CODE2VEC. Ramakrishnan et al. [33] and Ravichandar et al. [34] both performed adversarial attacks and adversarial training on CODE2SEQ. They all concentrate on method name prediction instead of generating long comment that help programmers understand.

Adversarial Defense. There have some relatively effective methods against adversarial attacks in NLP, which can roughly be classified as detection and model enhancement methods. Li et al. [22] proposed to use a context-aware spelling check service to detect spell errors in adversarial examples. Pruthi et al. [32] proposed a method to combat adversarial spelling mistakes by placing a word recognition model in front of the downstream DNNs. Wang et al. [44] proposed an adversarial defense method SEM, which inserts an encoder network before the original model and trains it to eliminate adversarial perturbations.

In addition to the detection-based defense, adversarial training as a typical model enhancement method, is also widely adopted. Javid et al. [18] used adversarial training to improve the robustness of text classification model. Wang et al. [43] augmented original training dataset with adversarial examples generated by AddSentDiverse to enhance the robustness of reading comprehension models. In order to improve the robustness of text classification, Ren et al. [35] randomly selected clean examples from the training set to generate adversarial examples using PWWS and mixed them with the training dataset to conduct adversarial training. Moreover, other work such as [37, 48–50] adopted adversarial training to improve the robustness of DNN models.

7 CONCLUSION

In this paper, we have presented a novel approach ACCENT to address the adversarial robustness problem of DNN models for code comment generation tasks, and demonstrated that the current mainstream code comment generation architectures are of poor robustness. Simply replacing identifiers which results in functionality-persevering and syntactically correct code snippets can degrade the performance of these representative models greatly. Experiment results show that our method can generate more effective adversarial examples on two public datasets across five mainstream code comment generation architectures. In addition, we demonstrated that the adversarial examples generated by our method had better transferability. To improve robustness, we have also proposed a novel training method. Our experimental results showed that this training method can achieve better performance in the code comment generation setting compared to the data augmentation method which has widely been used to improve robustness.

In the future, we plan to extend the existing framework and include more sophisticated, structure-rewriting based adversarial example generation techniques. More generally, we plan to explore the robustness issues of machine learning models for other software engineering tasks.

ACKNOWLEDGMENTS

This work was partially supported by the National Natural Science Foundation of China (NSFC, No. 61972197), the Natural Science Foundation of Jiangsu Province (No. BK20201292), the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Qing Lan Project. T. Chen is partially supported by Birkbeck BEI School Project (ARTEFACT), NSFC grant (No. 61872340 and No. 62072309), UK EPSRC grant (EP/P00430X/1), Guangdong Science and Technology Department grant (No. 2018B010107004) and an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2018A16).

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. *arXiv preprint arXiv:2005.00653* (2020).

[2] Lingfei Wu Collin McMillan Alex LeClair, Sakib Haque. 2020. Improved Code Summarization via a Graph Neural Network. In *2020 IEEE/ACM International Conference on Program Comprehension*. https://doi.org/10.1145/3387904.3389268

[3] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International conference on machine learning*. 2123–2132.

[4] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. 2018. Generating natural language adversarial examples. *arXiv preprint arXiv:1804.07998* (2018).

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[6] Yonatan Belinkov and Yonatan Bisk. 2017. Synthetic and natural noise both break neural machine translation. *arXiv preprint arXiv:1711.02173* (2017).

[7] Pavol Bielik and Martin T. Vechev. 2020. Adversarial Robustness for Code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. 896–907.

[8] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy, SP 2017*. 39–57.

[9] Chib, Siddhartha, Greenberg, and Edward. 1995. Understanding the Metropolis-Hastings Algorithm. *American Statistician* (1995).

[10] Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research-An Initial Survey.. In *SEKE*. 374–379.

[11] Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. 2018. Black-box generation of adversarial text sequences to evade deep learning classifiers. In *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 50–56.

[12] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *ICML*.

[13] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[14] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Twenty-Seventh International Joint Conference on Artificial Intelligence IJCAI-18*.

[15] Ruitong Huang, Bing Xu, Dale Schuurmans, and Csaba Szepesvari. 2015. Learning with a Strong Adversary. *Computer ence* (2015).

[16] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. 2019. Adversarial examples are not bugs, they are features. In *Advances in Neural Information Processing Systems*. 125–136.

[17] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

[18] Ebrahimi Javid, Anyi Rao, Daniel Lowd, and Dejing Dou. 2017. Hotflip: White-box adversarial examples for text classification. *arXiv preprint arXiv:1712.06751* (2017).

[19] Robin Jia and Percy Liang. 2017. Adversarial Examples for Evaluating Reading Comprehension Systems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*. 2021–2031.

[20] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *5th International Conference on Learning Representations, ICLR 2017,Toulon, France, April 24-26, 2017, Workshop Track Proceedings*.

[21] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 795–806. https://doi.org/10.1109/ICSE.2019.00087

[22] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. 2018. Textbugger: Generating adversarial text against real-world applications. *arXiv preprint arXiv:1812.05271* (2018).

[23] Pengcheng Li, Jinfeng Yi, Bowen Zhou, and Lijun Zhang. 2019. Improving the Robustness of Deep Neural Networks via Adversarial Training with Triplet Loss. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. 2909–2915.

[24] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJzIBfZAb

[25] Paul Michel, Xian Li, Graham Neubig, and Juan Miguel Pino. 2019. On evaluation of adversarial perturbations for sequence-to-sequence models. *arXiv preprint arXiv:1903.06620* (2019).

[26] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. (2013), 3111–3119.

[27] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2574–2582.

[28] Muzammal Naseer, Salman Hameed Khan, Shafin Rahman, and Fatih Porikli. 2018. Distorting neural representations to generate highly transferable adversarial examples. *arXiv preprint arXiv:1811.09020* (2018).

[29] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. 2017. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. 506–519.

[30] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *2016 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 372–387.

[31] Nicolas Papernot, Patrick Mcdaniel, Ananthram Swami, and Richard Harang. 2016. Crafting Adversarial Input Sequences for Recurrent Neural Networks. In *Military Communications Conference*.

[32] Danish Pruthi, Bhuwan Dhingra, and Zachary C Lipton. 2019. Combating adversarial misspellings with robust word recognition. *arXiv preprint arXiv:1905.11268* (2019).

[33] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2020. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043* (2020).

[34] Harish Ravichandar, Kenneth Shaw, and Sonia Chernova. 2020. STRATA: unified framework for task assignments in large teams of heterogeneous agents. *Auton. Agents Multi Agent Syst.* 34, 2 (2020), 38. https://doi.org/10.1007/s10458-020-09461-y

[35] Shuhuai Ren, Yihe Deng, Kun He, and Wanxiang Che. 2019. Generating Natural Language Adversarial Examples through Probability Weighted Word Saliency. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL 2019)*.

[36] Suranjana Samanta and Sameep Mehta. 2017. Towards crafting text adversarial samples. *arXiv preprint arXiv:1707.02812* (2017).

[37] Motoki Sato, Jun Suzuki, Hiroyuki Shindo, and Yuji Matsumoto. 2018. Interpretable adversarial perturbation in input embedding space for text. *arXiv preprint arXiv:1805.02917* (2018).

[38] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv: Computer Vision and Pattern Recognition* (2013).

[39] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian J. Goodfellow, Dan Boneh, and Patrick D. McDaniel. 2018. Ensemble Adversarial Training: Attacks and Defenses. (2018).

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[41] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[42] W Wang, L Wang, B Tang, R Wang, and A Ye. 2019. Towards a robust deep neural network in text domain a survey. *arXiv preprint arXiv:1902.07285* (2019).

[43] Yicheng Wang and Mohit Bansal. 2018. Robust machine comprehension models via adversarial training. *arXiv preprint arXiv:1804.06473* (2018).

[44] Zhaoyang Wang and Hongtao Wang. 2020. Defense of Word-level Adversarial Attacks via Random Substitution Encoding. *arXiv preprint arXiv:2005.00446* (2020).

[45] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *Advances in Neural Information Processing Systems 32*. 6563–6573.

[46] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 349–360.

[47] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 162:1–162:30. https://doi.org/10.1145/3428230

[48] Yuan Zang, Fanchao Qi, Chenghao Yang, Zhiyuan Liu, Meng Zhang, Qun Liu, and Maosong Sun. 2020. Word-level Textual Adversarial Attacking as Combinatorial Optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 6066–6080.

[49] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1169–1176.

[50] Huangzhao Zhang, Hao Zhou, Ning Miao, and Lei Li. 2019. Generating Fluent Adversarial Examples for Natural Languages. *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL 2019)* (2019).

[51] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. 1385–1397.

[52] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. 2019. Augmenting Java method comments generation with context information based on neural networks. *Journal of Systems and Software* 156 (2019), 328–340.

[53] Wei Zou, Shujian Huang, Jun Xie, Xinyu Dai, and Jiajun Chen. 2020. A Reinforced Generation of Adversarial Samples for Neural Machine Translation. *arXiv preprint arXiv:1911.03677* (2020).

Table 11. Results of different methods for improving robustness

		Java Dataset			Python Dataset		
		BLEU	METEOR	ROUGE-L	BLEU	METEOR	ROUGE-L
LSTM	Normal-Clean	35.47	19.72	47.57	30.83	17.06	41.77
	Normal-Adv(max=2)	13.08	6.83	21.75	18.30	8.62	27.24
	Normal-Adv(max=3)	13.07	6.75	21.52	17.92	8.27	26.64
	Aug-Clean	38.14	20.96	49.22	29.36	15.58	39.71
	Aug-Adv(max=2)	22.62	11.98	32.73	23.45	10.93	31.67
	Aug-Adv(max=3)	21.44	11.31	31.60	23.06	10.60	31.10
	Masked-Clean	39.60	23.24	39.60	30.64	16.70	40.54
	Masked-Adv(max=2)	31.88	18.23	41.48	27.26	13.92	36.28
	Masked-Adv(max=3)	31.31	17.84	40.85	26.81	13.56	35.72
Transformer	Normal-Clean	44.58	26.43	54.76	33.15	18.96	44.50
	Normal-Adv(max=2)	13.23	8.08	22.79	18.87	8.99	27.91
	Normal-Adv(max=3)	13.14	7.90	22.42	18.54	8.57	27.29
	Aug-Clean	34.14	17.36	45.77	32.97	18.76	44.02
	Aug-Adv(max=2)	18.10	9.10	27.89	24.71	12.34	33.82
	Aug-Adv(max=3)	17.82	8.95	27.50	24.06	11.77	32.97
	Masked-Clean	44.84	27.16	53.48	32.88	18.34	43.19
	Masked-Adv(max=2)	40.10	24.09	48.72	28.65	14.80	37.84
	Masked-Adv(max=3)	39.24	23.46	47.88	28.02	14.21	37.04
GNN	Normal-Clean	39.41	23.32	46.65	31.24	15.77	38.28
	Normal-Adv(max=2)	16.44	7.71	21.36	19.38	7.36	24.07
	Normal-Adv(max=3)	16.14	7.42	20.55	18.65	6.59	22.72
	Aug-Clean	34.28	20.72	43.15	31.27	15.66	38.02
	Aug-Adv(max=2)	17.32	9.13	23.69	22.65	9.26	27.71
	Aug-Adv(max=3)	16.93	8.73	22.94	22.21	8.61	26.65
	Masked-Clean	36.55	21.03	45.11	31.37	15.13	37.54
	Masked-Adv(max=2)	19.56	10.16	26.43	23.48	9.96	29.63
	Masked-Adv(max=3)	18.94	9.64	25.42	24.44	10.70	30.73
CSCG	Normal-Clean	42.39	25.77	53.61	30.82	17.67	48.14
	Normal-Adv(max=2)	8.85	5.10	23.50	11.90	6.23	32.94
	Normal-Adv(max=3)	8.95	5.02	23.44	12.08	6.15	32.83
	Aug-Clean	35.37	20.22	49.65	27.99	15.72	46.01
	Aug-Adv(max=2)	15.93	8.44	31.51	19.86	10.02	38.25
	Aug-Adv(max=3)	15.52	8.01	30.83	19.55	9.75	37.87
	Masked-Clean	35.39	20.82	51.42	29.18	16.39	46.53
	Masked-Adv(max=2)	18.37	10.38	34.44	16.61	8.04	35.04
	Masked-Adv(max=3)	17.75	9.96	33.77	16.23	7.67	34.52
Rencos	Normal-Clean	44.0	25.73	54.02	33.34	18.65	43.37
	Normal-Adv(max=2)	40.68	23.09	29.17	31.02	15.78	38.84
	Normal-Adv(max=3)	40.23	22.69	48.46	30.55	15.19	37.90
	Aug-Clean	41.47	24.18	51.59	15.58	3.83	17.53
	Aug-Adv(max=2)	40.33	23.02	49.10	15.84	3.93	17.57
	Aug-Adv(max=3)	40.27	22.93	48.84	15.93	3.98	17.68
	Masked-Clean	43.73	25.26	52.53	33.05	18.25	42.58
	Masked-Adv(max=2)	43.51	24.88	51.69	32.46	17.39	41.09
	Masked-Adv(max=3)	43.48	24.86	51.62	32.32	17.21	40.77

Table 12. The evaluation results of the generated comments

	Score	Java			Python		
		Similarity	Naturalness	Informativeness	Similarity	Naturalness	Informativeness
Normal training	5	11(3.67%)	35(11.67%)	12(4%)	25(8.33%)	63(21%)	20(6.67%)
	4	34(11.33%)	95(31.67%)	40(13.33%)	58(19.33%)	103(34.33%)	45(15%)
	3	67(22.33%)	83(27.67%)	58(19.33%)	87(29%)	95(31.67%)	64(21.33%)
	2	115(38.33%)	59(19.67%)	52(17.33%)	68(22.67%)	41(13.67%)	41(13.67%)
	1	73(24.33%)	28(9.33%)	138(46%)	62(20.67%)	8(2.67%)	130(43.33 %)
Data augmentation	5	34(11.33%)	178(59.33%)	33(11%)	43(14.33%)	204(68%)	48(16%)
	4	89(29.67%)	56(18.67%)	41(13.67%)	58(19.33%)	34(11.33%)	32(10.67%)
	3	52(17.33%)	33(11%)	67(22.33%)	28(9.33%)	12(4%)	17(5.67%)
	2	58(19.33%)	22(7.33%)	55(18.33%)	70(23.33%)	20(6.67%)	40(13.33%)
	1	67(22.33%)	11(3.67%)	104(34.67%)	101(33.67%)	30(10%)	163(54.33%)
Masked training	5	42(14%)	213(71%)	42(14%)	51(17%)	210(70%)	57(19%)
	4	191(63.67%)	43(14.33%)	112(37.33%)	167(55.67%)	52(17.33%)	118(39.33%)
	3	48(16%)	32(10.67%)	83(27.67%)	52(17.33%)	15(5%)	74(24.67%)
	2	14(4.67%)	10(3.33%)	35(11.67%)	20(6.67%)	13(4.33%)	34(11.33%)
	1	5(1.67%)	2(0.67%)	28(9.33%)	10(3.33%)	10(3.33%)	17(5.67%)

Table 13. The average results of the generated comments

	Java			Python		
	Similarity	Naturalness	Informative	Similarity	Naturalness	Informative
Normal training	2.32	3.17	2.12	2.72	3.67	2.28
Data augmentation	2.88	4.23	2.48	2.57	4.21	2.21
Masked training	3.84	4.52	3.35	3.76	4.46	3.55