Generating Adversarial Examples for Holding Robustness of Source Code Processing Models

Huangzhao Zhang,[1] Zhuo Li,[1] Ge Li,[1]* Lei Ma,[2] Yang Liu,[3] Zhi Jin[1]*
[1]Key Lab of High Confidence Software Technologies (Peking University, China), Ministry of Education
[2]Kyushu University, Japan, [3]Nanyang Technology University, Singapore
{zhang_hz, lizhmq, lige, zhijin}@pku.edu.cn, malei@ait.kyushu-u.ac.jp, yangliu@ntu.edu.sg

Abstract

Automated processing, analysis, and generation of source code are among the key activities in software and system lifecycle. To this end, while deep learning (DL) exhibits a certain level of capability in handling these tasks, the current state-of-the-art DL models still suffer from non-robust issues and can be easily fooled by adversarial attacks.

Different from adversarial attacks for image, audio, and natural languages, the structured nature of programming languages brings new challenges. In this paper, we propose a Metropolis-Hastings sampling-based identifier renaming technique, named Metropolis-Hastings Modifier (MHM), which generates adversarial examples for DL models specialized for source code processing. Our in-depth evaluation on a functionality classification benchmark demonstrates the effectiveness of MHM in generating adversarial examples of source code. The higher robustness and performance enhanced through our adversarial training with MHM further confirms the usefulness of DL models-based method for future fully automated source code processing.

Introduction

Automated processing, analysis, and generation of program source code can greatly reduce the cost in software and system development, testing, operation and maintenance, with the currently urgent industrial demands. It is not until several years ago, researchers started to propose deep learning (DL) methods for source code processing. Up to present, a lot of successes have been achieved in many tasks, such as functionality classification (Mou et al. 2016; Zhang et al. 2019b), code clone detection (Wei and Li 2017), code completion (Li et al. 2017), method naming (Allamanis, Peng, and Sutton 2016), and code comment generation (Hu et al. 2018), *etc*.

However, there is a major hazard lying under the state-of-the-art DL models – they lack of adversarial robustness. Adversarial examples can be generated from original inputs by carefully designed minor perturbations. The adversarial examples that have little or no difference from their original counterparts by the human perceptions, can result in severe erroneous behavior of DL models.

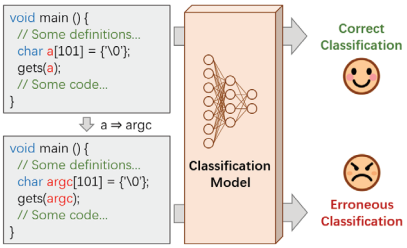

Figure 1: A simple example of adversarial attack on a source code classifier by performing identifier renaming.

Although some adversarial example generation approaches were recently proposed for natural language processing (NLP), such as GeneticAttack (Alzantot et al. 2018) and HotFlip (Ebrahimi et al. 2018), which performs word or character replacements to perturb the sentences, they are not suitable for source code. Unlike most natural languages, the source code of programming languages must strictly follow the rigid lexical, grammatical and syntactical constraints. In other words, the adversarial perturbations must satisfy all these constraints, without following which, the generated source code would encounter compilation error and would be automatically rejected for further analysis. The gradient-based perturbations, which are widely applied in image processing (Goodfellow, Shlens, and Szegedy 2015) are also difficult under this scenario since the source code space is discrete. Figure 1 gives a simple example of adversarial attacks on source code processing tasks, in which the classifier is attacked by the simple renaming of variable "*a*" to "*argc*". The renaming operation fully preserves the program semantics yet may mislead the classification model into erroneous decisions. Such hazards could potentially be reduced with adversarial training.

To address the aforementioned non-robust hazard, in this paper, we propose the Metropolis-Hastings Modifier algorithm (MHM). MHM generates adversarial examples of source code by performing iterative identifier renaming, which is based on Metropolis-Hastings sampling. Our

*Corresponding authors.

method confines the generated examples to satisfy the constraints of programming languages. The evaluation on the Open Judge (OJ) C/C++ source code functionality classification dataset (Mou et al. 2016) demonstrates that MHM is capable of attacking the LSTM model and the state-of-the-art AST-based Neural Network (ASTNN) model (Zhang et al. 2019b). Meanwhile, our further in-depth evaluation reveals the effectiveness of adversarial training with MHM in improving the classification performance. In particular, the models after adversarial training also gain resilience against the attackers. Our tool and data are open-sourced and publicly available[1]. The contributions of this paper are summarized as follows:

- We identify and confirm that the non-robust problem also occurs to DL models utilized for source code processing.
- We propose a conceptually simple but effective algorithm for adversarial example generation for source code, namely MHM.
- We demonstrate the effectiveness of MHM in adversarial attacking DL models for source code processing.
- We further demonstrate the usefulness of MHM for adversarial training to enhance robustness for DL models.

Preliminary

In this section, we give the formal definitions of the source code classification task and corresponding adversarial examples. Then, we introduce the processes of adversarial attack and adversarial training. We further briefly illustrate the robust and non-robust features.

Source Code Classification

Source code classification is a simple but important task in the field of source code processing and analysis, which is the subject task of this paper. Generation tasks such as code completion (Li et al. 2017), comment generation (Hu et al. 2018) and method naming (Allamanis, Peng, and Sutton 2016) are series classification under first-order Markov hypothesis.

Dataset. A well-labeled source code dataset $\mathcal{D}$ is defined as:

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N} \tag{1}$$

where x_i is a source code snippet and y_i is the corresponding one-hot vectorized label. The source code snippets are pre-processed into the form of character sequences, token sequences, abstract parsing trees (ASTs), or control flow graphs (CFGs), *etc*, according to the model requirements for input format. We denote the datasets for training, validation and testing by $\mathcal{D}^{(t)}$, $\mathcal{D}^{(v)}$ and $\mathcal{D}^{(e)}$, respectively.

Feature. Features are sets of mappings from inputs to scalars, denoted as $f : \mathcal{X} \to \mathcal{R}$.

Classification model. The classification model $\mathcal{C}$ serves as the subject model. It takes x as input, and outputs the predicted probabilities upon all candidate classes, denoted as $\mathcal{C}(x)$. A deep classification model first extracts features by performing non-linear transformations, and then performs classification based on the features. The overall classification can be then deducted as:

$$\mathcal{C}(x) = \int(f_1(x), f_2(x), \cdots, f_k(x)) \tag{2}$$

where $\int$ is the softmax classification function. The model is retrieved by minimizing the loss ($\mathcal{J}$) on $\mathcal{D}^{(t)}$:

$$\mathcal{J}(\mathcal{D}^{(t)}|\mathcal{C}) = -\frac{1}{N^{(t)}} \sum_{i=1}^{N^{(t)}} \sum y_i \circ \log \mathcal{C}(x_i) \tag{3}$$

where $\circ$ is the element-wise multiplication. Although $\mathcal{C}$ may obtain good performance on $\mathcal{D}^{(e)}$ after training, it still suffers from adversarial attacks on tasks like image processing.

Adversarial Examples

Adversarial example. Given a well-trained subject model $\mathcal{C}$ and a labeled data pair $((x, y))$, where $\mathcal{C}$ correctly classifies x to y, the adversarial example set $\mathcal{A}$ against x is defined as:

$$\begin{aligned}\mathcal{A}(x) = \{\hat{x} | \hat{x} \in \mathcal{E} \ \wedge \ \forall i \in \mathcal{I}, E(i|\hat{x}) = E(i|x) \\ \wedge \ \arg\max y \neq \arg\max \mathcal{C}(\hat{x})\}\end{aligned} \tag{4}$$

where $\mathcal{E}$ is the full set satisfying all lexical, grammatical and syntactical rules, $\mathcal{I}$ is the full set of legal input cases, and $E(i|x)$ is the output produced by the executable program of x on input case $i \in \mathcal{I}$.

Adversarial attack. Given an input $(x, y) \in \mathcal{D}$, adversarial attack aims to find $\hat{x}_1, \cdots, \hat{x}_n \in \mathcal{A}(x)$. In general, there are three ways for adversarial attack. One is to transform the adversarial attack as a maximization optimization problem, such as CW (Carlini and Wagner 2017). The objective could be defined as:

$$\begin{aligned}\max_{\hat{x}} \sum y \circ \log \frac{1}{C(x)}, \\ \text{s.t. } \hat{x} \in \mathcal{E} \ \wedge \ \forall i \in \mathcal{I}, E(i|\hat{x}) = E(i|x)\end{aligned} \tag{5}$$

Another way performs gradient ascent to generate adversarial examples, such as FGSM (Goodfellow, Shlens, and Szegedy 2015) and BIM (Kurakin, Goodfellow, and Bengio 2017). These approaches perturb examples based on gradient information with very limited iterations. The basic idea of one iteration could be formulated as:

$$x_{t+1} = x_t + \alpha \cdot \text{sign}(\nabla_x \sum y \circ \log \frac{1}{C(x)}) \tag{6}$$

In addition, a further approach of adversarial attack is often handled as a sampling problem. For instance, GeneticAttack by Alzantot et al. 2018 in NLP performs sampling via genetic algorithm. Examples are drawn with probability given by the target distribution. The target distribution ($\pi(x)$) is defined as:

$$\pi(x) \propto (1 - \mathcal{C}(x)[y]) \cdot \mathcal{X}_1 \cdots \mathcal{X}_k \tag{7}$$

where $\mathcal{C}(x)[y]$ is the probability of class y predicted by $\mathcal{C}$, and $\mathcal{X}_1, \cdots, \mathcal{X}_k$ are indicator functions of the lexical, grammatical and syntactical constraints.

In this paper, we adopt the sampling-based approach, performing M-H sampling for adversarial attack. Sampling algorithms, such as Markov chain Monte Carlo (MCMC), generate a sequence of examples, which may be more appropriate under the definition of adversarial examples in Equation

[1] https://github.com/Metropolis-Hastings-Modifier/MHM

4. In addition, utilizing gradient information is difficult in the source code scenario due to the discrete nature of source code space, which makes optimization-based and gradient-based approaches hard.

Adversarial attack includes two categories: 1) black-box attack that only allows the attackers to have access to model outputs, and 2) white-box attack (Ma et al. 2018; Xie et al. 2019a; 2019b) that allows full access to the subject model, including outputs, gradients and (hyper-) parameters. In this paper, we focus on the black-box setting.

Adversarial training. Adversarial training is an effective approach to improve adversarial robustness and model performance. Similar to data augmentation, adversarial training first updates the training set with adversarial examples generated from $\mathcal{D}^{(t)}$, forming $\mathcal{D}^{(t)}_{adv}$. Then a model having the same architecture and hyper-parameters with the subject model is trained from scratch on $\mathcal{D}^{(t)}_{adv}$.

In general, adversarially trained models gain better robustness to some extent, i.e. they are capable of resisting the same adversarial attack approach. Even though, it is still not known whether such a conclusion still applies in the context of source code processing.

Robustness

We formulate robust and non-robust features following the work of Ilyas et al. 2019.

Useful feature. ρ-useful features are strongly correlated with the classification label. In binary classification, useful features are defined as:

$$E_{(x,y)\sim\mathcal{D}}[y \cdot f(x)] \geq \rho \tag{8}$$

where $\rho > 0$. ρ measures the correlation between feature $f(x)$ and the classification label y. The greater ρ is, the stronger the correlation is.

Robust feature. γ-robust features are useful features, and are still useful after any allowable perturbations. If the classifier makes prediction upon robust features, it gains resistance towards adversarial attacks. In binary classification, robust features are defined as:

$$E_{(x,y)\sim\mathcal{D}}[\inf_{x'\in\Delta(x)} y \cdot f(x')] \geq \gamma \tag{9}$$

where $\Delta(x)$ is the full set of perturbed x, all elements in which satisfy the lexical, grammatical and syntactical constraints of programming languages.

Non-robust feature. Non-robust features are useful features, but not γ-robust for any $\gamma > 0$. Adversarial attack mainly perturbs the non-robust features, since any perturbations upon x may cause great change to non-robust $f(x)$.

Metropolis-Hastings Modifier

In this section, we first give the overview and workflow of our proposed technique MHM, and then elaborate each key steps in details.

Overview of MHM Technique

As aforementioned, we regard adversarial example generation as a sampling problem. Adversarial examples are supposed to be drawn from $\mathcal{A}(x)$ defined in Equation 4 with probability given by $\pi(x)$ defined in Equation 7. Existing approaches that generate adversarial examples in NLP mostly perform word-level (GeneticAttack by Alzantot et al. 2018) or character-level (HotFlip by Ebrahimi et al. 2018) replacement or substitution. Perturbations by these approaches are often not acceptable in the context of programming source code, because the lexical, grammatical and syntactical rules can be easily broken.

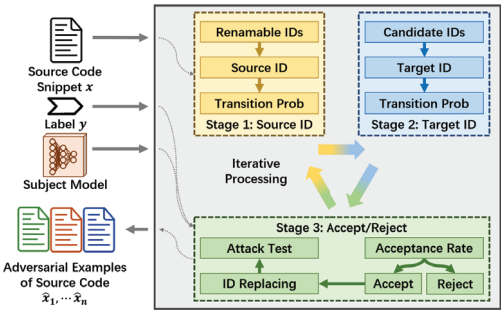

Figure 2: Workflow of Metropolis-Hastings Modifier.

In order to perturb the source code snippets within the constraints, we propose Metropolis-Hastings Modifier (MHM) method, which performs iterative identifier renaming based on Metropolis-Hastings (M-H) sampling (Metropolis et al. 1953; HASTINGS 1970; Chib and Greenberg 1995). MHM takes the source code classifier under attack ($\mathcal{C}$) and a pair of correctly classified data ($(x, y) \in \mathcal{D}$) as input, and outputs a sequence of adversarial examples $(\hat{x}_1, \hat{x}_2, \cdots)$. We expect the adversarial examples to satisfy three requirements: 1) enable to mislead the subject model, 2) free from compilation errors and enable to be executable, and 3) enable to produce the same execution results given the same input instances as the original examples.

In particular, the M-H algorithm is a classical Markov chain Monte Carlo sampling approach. Given the stationary distribution ($\pi(x)$) and transition proposal, M-H is able to generate desirable examples from $\pi(x)$. To be specific, at each iteration, a proposal to jump from x to x' is made based on the transition distribution ($Q(x'|x)$). The proposal is accepted with probability given by the acceptance rate:

$$\alpha(x'|x) = \min\{1, \alpha^*\} = \min\{1, \frac{\pi(x')Q(x|x')}{\pi(x)Q(x'|x)}\} \tag{10}$$

Upon acceptance, the algorithm jumps to x' and samples x'. Otherwise, it stays at x and does not perform sampling.

Inspired by M-H sampling, MHM consists of three stages in a single iteration: 1) selecting the source identifier, which is the identifier to be renamed, 2) choosing the target identifier, which is the identifier to rename to, and 3) determining to accept or reject the transition proposal. Stage 1 and 2 generate a transition proposal, and stage 3 accepts or rejects the proposal with probability given by the acceptance rate. Since the operation of identifier renaming is invertible, the transition in MHM is aperiodic and ergodic, making MHM eventually converge. In order to avoid unnecessary time cost, we provide to support maximum iteration threshold configuration for MHM.

Algorithm 1 Metropolis-Hastings Modifier algorithm.

Inputs:
Source code classification model $\mathcal{C}$;
Data pair (x, y), s.t. $\arg\max y = \arg\max \mathcal{C}(x)$;
Max iteration m;
Required adversarial example number n;
Candidate identifier number c.
Outputs:
n adversarial examples corresponding to (x, y).

1: Initialize $x_0 \leftarrow x$, $t \leftarrow 1$, $count \leftarrow 0$;
2: Initialize $\{\hat{x}_1, \cdots, \hat{x}_n\} \leftarrow \{\text{None}, \cdots, \text{None}\}$.
3: $\mathcal{T}_{all} \leftarrow \{w | w \text{ is an identifier} \wedge w \in \mathcal{V}\}$.
4: **while** $t < m$ **do**
5: $\quad \mathcal{S} \leftarrow \{w | w \text{ is an identifier} \wedge w \text{ is defined in } x\}$;
6: $\quad$ Sample s from $\mathcal{S}$;
7: $\quad \mathcal{T} \leftarrow \{w_i | w_i \notin \mathcal{S} \wedge i = 1, 2, \cdots, c\}$;
8: $\quad$ Sample t from $\mathcal{T}$.
9: $\quad x' \leftarrow x_{t-1}$ with all s replaced with t;
10: $\quad \alpha^* \leftarrow \frac{1-\mathcal{C}(x')[y]}{1-\mathcal{C}(x)[y]}$;
11: $\quad$ Sample $u \sim U(0, 1)$;
12: $\quad$ **if** $u < \alpha^*$ **then**
13: $\quad\quad$ Accept transition $x_{t-1} \Rightarrow x'$;
14: $\quad\quad x_t \leftarrow x'$.
15: $\quad\quad$ **if** $\arg\max y \neq \arg\max \mathcal{C}(x_t)$ **then**
16: $\quad\quad\quad count \leftarrow count + 1$;
17: $\quad\quad\quad \hat{x}_{count} \leftarrow x_t$.
18: $\quad\quad$ **end if**
19: $\quad$ **else**
20: $\quad\quad$ Reject transition $x_{t-1} \Rightarrow x'$;
21: $\quad\quad x_t \leftarrow x_{t-1}$.
22: $\quad$ **end if**
23: $\quad t \leftarrow t + 1$.
24: **end while**
25: Return $\{\hat{x}_1, \hat{x}_2, \cdots, \hat{x}_n\}$.

Figure 2 gives the overall workflow of one single iteration in MHM, while Algorithm 1 shows the corresponding detailed operations of MHM. We would illustrate the details of each stage in the rest of this section.

Transition Proposal

The transition proposal of MHM is made in stage 1 and 2. Stage 1 analyzes and collects all candidate identifiers that can be renamed and selects the source identifier from them. Stage 2 generates candidate identifiers, from which the target identifier is selected.

Most programming languages (e.g., C/C++ and Python) have similar lexical rules for identifiers. For example, the naming rule in the form of regular expression is often defined as "[a-zA-Z_][0-9a-zA-Z_]*". Identifiers that can be renamed are those defined or declared within the source code snippet. We collect all definitions and declarations of variables, and functions within the source code snippet x, forming the set $\mathcal{S}(x)$. The source identifier (s) is then drawn from $\mathcal{S}(x)$ with equal probability (see Line 5-6 in Algorithm 1).

The candidate identifier set ($\mathcal{T}(x)$) is generated from the overall vocabulary set ($\mathcal{V}$). Elements in $\mathcal{T}(x)$ must satisfy the aforementioned lexical rules for identifiers, and must not appear in $\mathcal{S}(x)$. These two constraints guarantee that the snippet after identifier renaming still satisfies the requirements defined in Equation 4. The candidate size is a hyperparameter of MHM. We do not attempt to rename identifiers with similar names (e.g., renaming the variable "count" to "cnt"), because all possible identifiers are treated equally in MHM. Therefore, the target identifier (t) is drawn from $\mathcal{T}(x)$ with equal probability (see Line 7-8 in Algorithm 1).

The transition proposal is to rename s in x to t, forming x'. The transition probability is then defined as:

$$Q(x'|x) \propto \mathcal{I}\{s \in \mathcal{S}(x) \wedge t \in \mathcal{T}(x)\} \cdot P_{\mathcal{S}(x)}(s) \cdot P_{\mathcal{T}(x)}(t), \quad \text{where } \mathcal{S}(x) \cap \mathcal{T}(x) = \emptyset \tag{11}$$

where $P_{\mathcal{S}(x)}(s)$ and $P_{\mathcal{T}(x)}(t)$ are uniform distributions to draw s and t from $\mathcal{S}(x)$ and $\mathcal{T}(x)$, respectively, and the constraint requires the generated $\mathcal{T}(x)$ not intersecting with $\mathcal{S}(x)$. The inverted transition probability of renaming t back to s is similarly defined.

Acceptance Rate

The transition proposal made in stage 1 and 2 is accepted with probability given by the acceptance rate computed in stage 3. The acceptance rate (α) is computed based on the transition probability and the inverted transition probability. If the transition proposal is accepted, s in x is then renamed to t, otherwise the transition proposal is rejected. The source code snippet after renaming will be tested by $\mathcal{C}$ to verify if it is an adversarial example (see Line 9-22 Algorithm 1).

The unnormalized acceptance rate α^* is deducted as:

$$\begin{aligned} \alpha^*(x'|x) &= \frac{\pi(x')Q(x|x')}{\pi(x)Q(x'|x)} \\ &= \frac{(1-\mathcal{C}(x')[y]) \cdot P_{\mathcal{S}(x')}(t) \cdot P_{\mathcal{T}(x')}(s)}{(1-\mathcal{C}(x)[y]) \cdot P_{\mathcal{S}(x)}(s) \cdot P_{\mathcal{T}(x)}(t)} \\ &\approx \frac{1-\mathcal{C}(x')[y]}{1-\mathcal{C}(x)[y]} \end{aligned} \tag{12}$$

where x' is the snippet after renaming s to t, and the constraints in $\pi(\cdot)$ and $Q(\cdot)$ are all satisfied and therefore neglected. We make an approximate assumption that the inverted distribution generates the same $\mathcal{T}(x')$ with $\mathcal{T}(x)$, and approximate α^* in Equation 12, which is rather convenient to compute. When $\alpha^* > 1$, the proposal will be accepted for sure (100% probability).

Experiments

In this section, we perform in-depth evaluation to demonstrate the usefulness of our proposed technique. We first introduce our experimental setups, and then present the experimental results of MHM on adversarial attack and adversarial training, respectively.

Experiment Setup

Dataset. We choose the Open Judge (OJ) dataset, the benchmark dataset in source code classification, as the study subject, which is proposed by Mou et al. 2016. OJ is collected

Hyper-parameters	LSTM	ASTNN
Vocabulary size	5,000	8,878
Embedding size	512	128
Hidden size	600	100
Layers	2	1
Dropout	0.5	0.2
Batch size	32	64
Optimizer	Adam	AdaMax
Learning rate	0.003(exp decay)	0.002

Table 1: Hyper-parameters of the subject models.

Model	Acc (%)	CE
RNN (Mou et al. 2016)	84.8	–
LSTM (Zhang et al. 2019b)	88.0	–
LSTM	**92.9**	**0.23**
TBCNN (Mou et al. 2016)	94.0	–
ASTNN (Zhang et al. 2019b)	98.2	–
ASTNN	**98.2**	**0.06**

Table 2: Performances of the subject classifiers.

from an open judge platform, consisting of 52,000 C/C++ compilable and correctly executable code files with problem number labels. There are 104 problems (i.e. 104 classes) in OJ, and each class contains 500 code files.

We first filter OJ by a C++ (ver.11) parser and discard those against the rules of grammar. Macros, such as "#include", are removed from the source code during the filtration, and so are the comments. Then, we substitute all strings, integer constants and floating point constants with "⟨STR⟩", "⟨INT⟩" and "⟨FP⟩", respectively. Finally, we split the filtered dataset $(4:1)$, resulting in a training set with the size of 38,924 and a test set with the size of 9,718, where each data pair represents a compilable source code file and its functionality label. During training, we randomly extract 20% code files from the training set, forming the validation set. The overall training set and the test set remain the same during our experiments, while the validation sets are different under each experiment trial.

Subject models. Subject models are target models under adversarial attack. We adopt LSTM and ASTNN (Zhang et al. 2019b) architectures as subject models in our experiments. We follow the instructions in the original work and choose the hyper-parameters of the subject models listed in Table 1. The obtained performances of the subject models and some other reported results are listed in Table 2.

The LSTM architecture is one of the most representative model in sequence processing, and is widely used in the field of source code processing (Zhang et al. 2019b; Li et al. 2017). In this paper, we apply bidirectional LSTM as one of the subject models. The input source code snippets are token sequences, which are retrieved by traversal upon the syntactical parsing trees. The LSTM subject model consists of two layers of bidirectional LSTMs. The prediction is generated by a softmax layer, based on the mean vector of the last layer hidden states. The hyper-parameters are listed in Table 1. Our LSTM outperforms LSTM by Zhang et al. 2019b since ours have much more parameters. After training, the LSTM subject model gives 92.9% accuracy, which is a competitive and acceptable result.

ASTNN (Zhang et al. 2019b) is the state-of-the-art architecture on the OJ benchmark. It is an AST-based model and we reuse the code open-sourced by the authors. The model takes ASTs as inputs, which are generated by the pycparser tool[2]. All hyper-parameters are the same as reported in the original paper. The ASTNN subject model gives 98.2% accuracy, which is the state-of-the-art result on OJ.

Attacker	Subject	Attack(%)	Valid(%)	Succ(%)
GA	LSTM	92.5	11.1	10.3
w/o LM	LSTM	92.0	9.7	8.9
MHM	LSTM	71.3	100	**71.3**
MHM	ASTNN	92.1	100	**92.1**

Table 3: Results of adversarial attack.

Baseline algorithm. We take GeneticAttack (GA) (Alzantot et al. 2018) as our baseline, which is the state-of-the-art attacking approach on SNLI (a natural language inference dataset) under black-box setting. GA uses a black-box population-based genetic algorithm to generate adversarial examples for natural languages. Intuitively, it maintains a population of sequences, and performs perturbation by word-level replacement according to the embedding distances (independently retrieved from the subject model) under black-box setting. Then, the intermediate sentences are filtered by the subject classifier and a language model, which leads to the next generation. Unlike MHM, GA verifies the perturbations with a language model, which cannot always guarantee the aforementioned constraints.

In our experiment, we apply a bidirectional LSTM language model with the hidden size of 600 trained on OJ, to the GA algorithm. Embedding distances are calculated on pre-trained Word2Vec embeddings obtained on OJ, which is independent of the subject model. We set the population size to 32, and the maximum iteration threshold to 30.

Adversarial Attack

To validate the attacking capability, we randomly sample 1,000 correctly classified examples from the test set of OJ for LSTM and ASTNN respectively, on which we apply the GA baseline and MHM for adversarial attack on subject models. Lexical, grammatical and syntactical constraints are neglected during generation for GA. In this paper, we use attack rate, validity rate and final success rate as the evaluation criteria. In particular, attack rate is the proportion of samples that are capable to mislead the subject model. Validity rate is the proportion of valid source code snippets among those which are able to attack the subject model. Success rate is the proportion of valid adversarial examples, which

[2]https://github.com/eliben/pycparser

Original examples

```
void main () {
  int i = 0, len, w = 0;
  char a[101] = {'\0'};
  gets(a);
  len = strlen(a);
  while(i < len) {
    if(w != 1 || a[i] != ' ')
      printf("%c", a[i]);
    if(a[i] == ' ' )
      w = 1;
    else w = 0;
    i++;
  }
}
```

```
int sushu[168] = {2, 3, 5, 7,
  /* more magic numbers ... */};
int main() {
  int i, j, m;
  cin >> m;
  for (i = 0; i < 168; ++i)
    for (j = i; j < 168; ++j) {
      if (m == sushu[i] + sushu[j])
        cout << sushu[i] << " " <<
sushu[j] << endl;
    }
    return 0;
}
```

Adversarial examples

```
void main () {
  int i = 0, len, w = 0;
  char argc[101] = {'\0'};
  gets(argc);
  len = strlen(argc);
  while(i < len) {
    if(w != 1 || argc[i] != ' ')
      printf("%c", argc[i]);
    if(argc[i] == ' ' )
      w = 1;
    else w = 0;
    i++;
  }
}
```

```
int maxindex[168] = {2, 3, 5, 7,
  /* more magic numbers ... */};
int main() {
  int i, j, m;
  cin >> m;
  for (i = 0; i < 168; ++i)
    for (j = i; j < 168; ++j) {
      if (m==maxindex[i]+maxindex[j])
        cout << maxindex[i] << " " <<
maxindex[j] << endl;
    }
    return 0;
}
```

Figure 3: Adversarial examples generated by MHM on OJ. The renamed identifiers are marked in red.

is the product of attack rate and validity rate. The results are shown in Table 3.

Although GA with or without (i.e., the row w/o LM in Table 3) the language model gives high attack rate, only very little portion of the generated examples are valid adversarial examples since most are unable to pass the compilation checking. GA also cannot attack ASTNN because the subject ASTNN performs front-end compilation to obtain ASTs. MHM is able to attack LSTM and ASTNN subject models effectively, and the adversarial example generation rate upon ASTNN even reaches above 90%. All examples generated by MHM are valid adversarial examples, which satisfy Equation 4. After restoring the macros, integer and floating point constants, and strings, all adversarial examples generated by MHM are compilable and executable. The results of execution are identical to the original examples given the test cases. This is obvious since MHM ensures any modification towards source code is error-free and synonymous. In addition, the GA experiments take several days, while MHM only takes about several hours.

Figure 3 shows some of the adversarial examples generated by MHM. Identifier renaming modifications, such as "*a*" to "*argc*" and "*sushu*" to "*maxindex*", will lead the subject models to erroneous predictions.

The results indicate that identifiers may be non-robust features. The identifier "sushu", which is the Chinese Pinyin of "prime number" (in the original example in case 2), is strongly correlated with the functionality. MHM discovers this non-robust feature, and perturbs it to "maxindex". The perturbation from "sushu" to "maxindex" breaks the original correlation and erroneously correlates to another label.

Model	Acc(%)	CE	**Attack**(%)
LSTM	92.9	0.23	71.3
+ adv train	**94.0**	**0.20**	**46.4**
ASTNN	98.2	0.061	92.1
+ adv train	98.1	**0.057**	**54.7**

Table 4: Results of adversarial training with MHM.

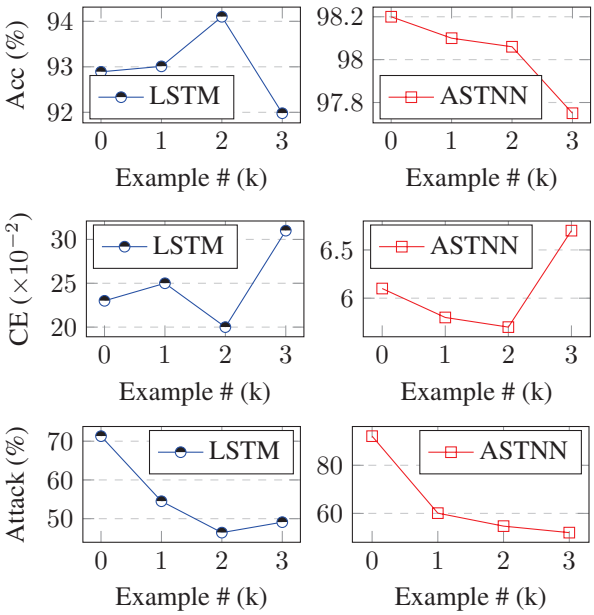

Figure 4: The Impact on number of adversarial examples mixed into the training set during adversarial training.

Adversarial Training

In order to validate whether adversarial training is still useful for improving the adversarial robustness or classification accuracy of the subject model. New models with identical hyper-parameters in Table 1 are trained from scratch on the adversarially augmented training set with generated adversarial examples included.

In particular, we generate 2,000 adversarial examples from the training set with MHM in the same procedure as for the LSTM and ASTNN of last subsection. In the next step, the adversarial examples are mixed into the training set, forming an augmented training set, based on which, we train new models. In the next step, the models are tested against the test set and adversarial attacks by MHM again. The results are shown in Table 4.

We can see that both LSTM and ASTNN gain the ability to resist adversarial attack from MHM after adversarial training without sacrificing much performance. The attacking rates on adversarially trained LSTM and ASTNN decrease greatly (about 30%-40%). Moreover, after adversarial training, LSTM even gains 1.1% accuracy improvement.

To understand the impact caused by the quantity of adversarial examples in adversarial training, we further per-

form more experiments where the number of adversarial examples mixed into the training set is treated as a hyper-parameter. We mix 1,000-3,000 examples generated by MHM into the training set, and evaluate accuracy, cross-entropy loss and attacking rate of the adversarially trained models by MHM, respectively. Figure 4 shows the impact of different number of examples on adversarial training.

After reaching the threshold, as the number of adversarial examples increases, the adversarially trained models tend to lose performance, while they tend to be more robust. It then becomes a trade-off to pick an appropriate example number. In this case, mixing 2,000 adversarial examples into the training set may be the appropriate choice.

As explained in the last subsection, identifier names may be the useful non-robust feature. Variable or function names are strongly correlated to the functionality label in OJ (such as "sushu" in case 2 in Figure 3). Therefore, a robust classification model should not make prediction based on these features. After mixing adversarial examples into the training set, the correlation between identifier names and functionality labels are broken and this non-robust feature is no longer useful for classification. As a consequence, adversarially trained models gain much better robustness.

Related Works

In this section, we briefly introduce the state-of-the-art source code processing, and discuss the current research progress in adversarial attacks most relevant to ours.

Source Code Processing

Automated program source code processing with DL techniques has achieved much progress in recent years, such as pointer mixture networks (Li et al. 2017) in code completion, convolutional attantion networks (Allamanis, Peng, and Sutton 2016) in method naming, and DeepCom (Hu et al. 2018) in comment generation, *etc.*

Source code functionality classification, along with the OJ benchmark, is proposed by Mou et al. 2016. Tasks such as code clone detection (Wei and Li 2017; Liuqing et al. 2017; Hao et al. 2019), code defect prediction (Wang, Liu, and Tan 2016; Dam et al. 2018; Gong et al. 2019), and type inference (Hellendoorn et al. 2018; Malik, Patra, and Pradel 2019), *etc.*, are related to functionality classification. TBCNN (Mou et al. 2016) and state-of-the-art ASTNN (Zhang et al. 2019b) perform functionality classification upon ASTs.

In this paper, we adopt functionality classification as the subject task, because it is a simple but important task in source code processing, and it is closely related to the aforementioned tasks. The LSTM and ASTNN architectures are employed as subject models, since LSTM is the most representative and widely-used model for token sequence processing (Zhang et al. 2019b; Li et al. 2017), while ASTNN is the state-of-art model for AST classification.

Adversarial Examples

Adversarial learning, including adversarial example, adversarial attack, and adversarial training, *etc.*, draws more and more attention in the recent years. The vulnerability of DL models were first studied in image classification (Szegedy et al. 2013). Afterwards, gradient-based perturbations are intesively studied, e.g., FGSM (Goodfellow, Shlens, and Szegedy 2015), BIM (Kurakin, Goodfellow, and Bengio 2017) and JSMA (Papernot et al. 2016), *etc.* It is not until recently, the existence of adversarial examples are also revealed in speech recognition (Carlini and Wagner 2018; Sun et al. 2018; Qin et al. 2019) and natural language processing (NLP) (Jia and Liang 2017; Alzantot et al. 2018; Ebrahimi et al. 2018; Du et al. 2019).

In particular, in the field of NLP, adversarial example generation is rather difficult due to the discrete property of the language space. Jia and Liang 2017 add distracting sentences into the input document, in order to force the subject model to produce misclassifications. GA (Alzantot et al. 2018) employs the population-based genetic algorithm to perform black-box attack toward binary sentimental classifiers and natural language inference models. Further works introduce gradient information into the generation process of adversarial examples for natural languages. HotFlip (Ebrahimi et al. 2018) iteratively flips character towards the gradient direction to attack character-level classifiers. Zhang et al. 2019a introduce gradient information into the transition proposal for white-box adversarial example generation. Recently, Cheng, Jiang, and Macherey 2019 propose doubly adversarial input examples for enhancing the robustness of neural machine translation models.

Although source code processing is similar to NLP, there is a major difference that programming languages are strictly bounded by lexical, grammatical and syntactical rules, while the constraints for natural languages are rather loose. Therefore, we embed strict identifier rule checking into our proposed MHM, which differs from existing approaches.

Researchers have made great efforts to understand adversarial examples in the field of CV. Goodfellow, Shlens, and Szegedy 2015 investigate that adversarial examples are originated from the data distribution. Lately, Ilyas et al. 2019 demonstrate that adversarial examples come from the non-robust features lying under the dataset. By erasing the non-robust features and keeping the robust ones in the dataset, the models gain higher robustness. We suggest adversary in source code processing also has similar traits. We confirm the issue of non-robustness in source code processing, and improve the robustness by adversarial training with MHM. Furthermore, we explain the robust and non-robust features in source code by empirical case study.

Conclusion

In this paper, we identify and confirm the non-robust problem of DL models in source code processing tasks. To cater to the special characteristics of source code, we propose MHM that generates adversarial examples by iteratively identifier renaming based on M-H sampling algorithm. MHM strictly confines the perturbations on the examples to satisfy the rigid constraints of programming languages. The experimental results demonstrate that our proposed MHM could effectively generate adversarial examples to attack the subject models with high attack rate and validity rate. The obtained adversarial examples from MHM are

able to improve classification performance and adversarial robustness of the subject models after adversarial training.

In future work, more advanced synonymous parsing and tree-based perturbations could be included into MHM, such as replacing "for" sub-tree with "while" sub-tree, *etc.* Structural modification without changing the execution path may also produce high-quality adversarial examples. The renaming operation applied in MHM already confirms its effectiveness in uncovering the non-robust issues. We expect the inclusion of more diverse synonymous parsing tree modifiers would further improve the attack capability.

Acknowledgement

This research is supported by the National Key R&D Program under Grant No.2018YFB1003904, the National Natural Science Foundation of China under Grant No.61832009, the JSPS KAKENHI Grant No.19K24348, 19H04086, and Qdai-jump Research Program No.01277.

References

Allamanis, M.; Peng, H.; and Sutton, C. 2016. A convolutional attention network for extreme summarization of source code. In *ICML 2016*, 2091–2100.

Alzantot, M.; Sharma, Y.; Elgohary, A.; Ho, B.-J.; Srivastava, M.; and Chang, K.-W. 2018. Generating natural language adversarial examples. In *EMNLP 2018*, 2890–2896.

Carlini, N., and Wagner, D. A. 2017. Towards evaluating the robustness of neural networks. In *SP 2017*, 39–57.

Carlini, N., and Wagner, D. 2018. Audio adversarial examples: Targeted attacks on speech-to-text. In *SPW 2018*, 1–7. IEEE.

Cheng, Y.; Jiang, L.; and Macherey, W. 2019. Robust neural machine translation with doubly adversarial inputs. *arXiv preprint arXiv:1906.02443*.

Chib, S., and Greenberg, E. 1995. Understanding the metropolis-hastings algorithm. *The american statistician* 49(4):327–335.

Dam, H. K.; Pham, T.; Ng, S. W.; Tran, T.; Grundy, J.; Ghose, A.; Kim, T.; and Kim, C.-J. 2018. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*.

Du, X.; Xie, X.; Li, Y.; Ma, L.; Liu, Y.; and Zhao, J. 2019. Deepstellar: Model-based quantitative analysis of stateful deep learning systems. In *FSE*, 477–487.

Ebrahimi, J.; Rao, A.; Lowd, D.; and Dou, D. 2018. Hotflip: White-box adversarial examples for text classification. In *ACL 2018*, 31–36.

Gong, L.; Jiang, S.; Yu, Q.; and Jiang, L. 2019. Unsupervised deep domain adaptation for heterogeneous defect prediction. *IEICE TRANSACTIONS on Information and Systems* 102(3):537–549.

Goodfellow, I. J.; Shlens, J.; and Szegedy, C. 2015. Explaining and harnessing adversarial examples. In *ICLR 2015*.

Hao, Y.; Wing, L.; Long, C.; Ge, L.; Tao, X.; and Qianxiang, W. 2019. Neural detection of semantic code clones via tree-based convolution. In *ICPC 2019*, 70–80. IEEE Press.

HASTINGS, W. 1970. Monte carlo sampling methods using markov chains and their applications. *Biometrika* 57(1):97–109.

Hellendoorn, V. J.; Bird, C.; Barr, E. T.; and Allamanis, M. 2018. Deep learning type inference. In *ESEC/FSE 2018*, 152–162. ACM.

Hu, X.; Li, G.; Xia, X.; Lo, D.; and Jin, Z. 2018. Deep code comment generation. In *ICPC 2018*, 200–210. ACM.

Ilyas, A.; Santurkar, S.; Tsipras, D.; Engstrom, L.; Tran, B.; and Madry, A. 2019. Adversarial examples are not bugs, they are features. In *ArXiv preprint arXiv:1905.02175*.

Jia, R., and Liang, P. 2017. Adversarial examples for evaluating reading comprehension systems. *arXiv preprint arXiv:1707.07328*.

Kurakin, A.; Goodfellow, I. J.; and Bengio, S. 2017. Adversarial examples in the physical world. In *ICLR 2017*.

Li, J.; Wang, Y.; Lyu, M. R.; and King, I. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*.

Liuqing, L.; Feng, H.; Wenjie, Z.; Meng, N.; and Barbara, R. 2017. Cclearner: A deep learning-based clone detection approach. In *ICSME 2017*.

Ma, L.; Juefei-Xu, F.; Sun, J.; Chen, C.; Su, T.; Zhang, F.; Xue, M.; Li, B.; Li, L.; Liu, Y.; et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *ASE*, 120–131.

Malik, R. S.; Patra, J.; and Pradel, M. 2019. Nl2type: inferring javascript function types from natural language information. In *ICSE 2019*, 304–315. IEEE Press.

Metropolis, N.; Rosenbluth, A. W.; Rosenbluth, M. N.; Teller, A. H.; and Teller, E. 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics* 21(6):1087–1092.

Mou, L.; Ge, L.; Zhi, J.; Lu, Z.; and Tao, W. 2016. Convolutional neural network over tree structures for programming language processing. In *AAAI 2016*.

Papernot, N.; McDaniel, P. D.; Jha, S.; Fredrikson, M.; Celik, Z. B.; and Swami, A. 2016. The limitations of deep learning in adversarial settings. In *EuroS&P 2016*, 372–387.

Qin, Y.; Carlini, N.; Cottrell, G. W.; Goodfellow, I. J.; and Raffel, C. 2019. Imperceptible, robust, and targeted adversarial examples for automatic speech recognition. In *ICML 2019*, 5231–5240.

Sun, S.; Yeh, C.; Ostendorf, M.; Hwang, M.; and Xie, L. 2018. Training augmentation with adversarial examples for robust speech recognition. In *Interspeech 2018*, 2404–2408.

Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; Erhan, D.; Goodfellow, I.; and Fergus, R. 2013. Intriguing properties of neural networks. In *ArXiv preprint arXiv:1312.6199*.

Wang, S.; Liu, T.; and Tan, L. 2016. Automatically learning semantic features for defect prediction. In *ICSE 2016*, 297–308.

Wei, H., and Li, M. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI 2017*, 3034–3040.

Xie, X.; Ma, L.; Juefei-Xu, F.; Xue, M.; Chen, H.; Liu, Y.; Zhao, J.; Li, B.; Yin, J.; and See, S. 2019a. Deephunter: A coverage-guided fuzz testing framework for deep neural networks. In *ISSTA*, 146–157.

Xie, X.; Ma, L.; Wang, H.; Li, Y.; Liu, Y.; and Li, X. 2019b. Diffchaser: Detecting disagreements for deep neural networks. In *IJCAI*, 5772–5778.

Zhang, H.; Zhou, H.; Miao, N.; and Li, L. 2019a. Generating fluent adversarial examples for natural languages. In *ACL 2019*, 5564–5569.

Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; and Liu, X. 2019b. A novel neural source code representation based on abstract syntax tree. In *ICSE 2019*, 783–794. IEEE Press.