Adversarial Examples for Models of Code

Noam Yefet
Technion
snyefet@cs.technion.ac.il

Uri Alon
Technion
urialon@cs.technion.ac.il

Eran Yahav
Technion
yahave@cs.technion.ac.il

Abstract

We introduce a novel approach for attacking trained models of code with adversarial examples. The main idea is to force a given trained model to make a prediction of the adversary’s choice by introducing small perturbations that do not change program semantics. We find these perturbations by deriving the desired prediction with respect to the model’s inputs while holding the model weights constant, and following the gradients to slightly modify the input.

To defend a model against such attacks, we propose placing a defensive model in front of the downstream model. The defensive model detects unlikely mutations and masks them before feeding the input to the downstream model.

We show that our attack succeeds in changing a prediction to the adversary’s desire (“targeted attack”) up to 89% of the times, and succeeds in changing a given prediction to any incorrect prediction (“non-targeted attack”) 94% of the times. By using our proposed defense, the success rate of the attack drops drastically for both targeted and non-targeted attacks, with a minor penalty of 2% relative degradation in accuracy while not performing under attack.

1 Introduction

Neural models of code have recently achieved state-of-the-art performance on various tasks such as prediction of variable names and types [1, 5, 10, 32], code summarization [2, 3, 18], code generation [4, 11, 28], code search [13, 25, 34], and bug finding [29, 33, 35].

In other domains, such as computer vision, deep models have been shown to be vulnerable to adversarial examples [19, 38]. Adversarial examples are inputs crafted by an adversary to force a trained neural model to make a certain (incorrect) prediction. Generation of adversarial examples was demonstrated for image classification [19, 38] as well as for other domains [14]. The basic idea underlying many of the techniques for generating adversarial examples is the following: add specially-crafted noise to a correctly labeled input such that the model under attack yields a desired incorrect label when presented with the modified input (with the addition of noise).

The idea of adding noise to continuous objects such as images, in a way that is imperceptible to the human eye but changes the prediction of a model, is very natural and relatively easy to realize mathematically – changing intensity of pixel values in the image. Unfortunately, this does not carry over to the domain of programs, as a program is a discrete object that has to maintain semantic properties. In this paper, we provide a novel approach for generating adversarial examples for neural models for code. Additionally, we propose an effective defense against these attacks that results in only a minor degradation in the overall model accuracy.

Existing Techniques

To the best of our knowledge, our work is the first to investigate adversarial examples for models of code. The challenge of adversarial examples for discrete inputs has been studied in the domain of natural language processing (NLP). Ebrahimi et al. [17] present a technique for generating adversarial examples that attack a character-level neural classifier. The idea is to flip a single character in a word (i.e., produce a typo) such that the change is hardly noticeable, but still leads the model to predict the adversarial target label. Their model can also flip a whole word, as long as it is “similar” (cosine similarity under a precomputed embedding) to the original word. Alzantot et al. [7] present a technique for generating “semantically and syntactically similar adversarial examples” that attack well-trained deep models. The main idea is to replace a random word in a given sentence with a similar word (nearest neighbour) in some embedding space. In contrast to these techniques, our challenge is to find adversarial examples that preserve program semantics and are also syntactically similar to the original input. Rabin et al. [31] identified a problem of robustness in models of code, without suggesting a concrete method for producing adversarial examples or defending against them.

Our Approach

The main idea in our approach is to leverage semantic preserving perturbations to the program in a way that causes a model to make incorrect predictions. We select these perturbations by deriving the output distribution of the model with respect to the model’s input and following the gradient. We show that models of code are susceptible both to targeted attacks which force a model to make a specific (incorrect) prediction chosen by the adversary, as well as to simpler non-targeted attacks which force a model to make an incorrect prediction without a specific target prediction in mind. Our approach is a “white-box” approach, as it assumes that the attacker has access to the model under attack or to a similar model. Under this assumption, our approach is general and is applicable to any model that can be derived with respect to its inputs, namely, any neural model. We do not make any assumptions on the internal details or specific architecture of the model under attack.
The defense mechanism allows us to trade off accuracy of Code2vec sorts a given array. The technique.

In this section, we provide an informal overview of our tech-

2 Overview

In this section, we provide an informal overview of our tech-

Considering the code snippet \( f_1 \) of Figure 1. This code snippet sorts a given array. The Code2vec model [6] applied to this code snippet predicts the correct name, sort. Our goal is to find semantically equivalent snippets that will cause an underlying model to yield an incorrect target prediction. More formally:

**Goal** Given a program \( P \) labeled by correct label \( L \), find a semantically equivalent program \( P' \) such that \( P' \) is labeled with a given adversarial label \( T \) by the model under attack.

The main challenge in tackling the above goal is how to explore the vast space of programs that are semantically equivalent to \( P \), and how to find a program that is going to be labeled by \( T \) by the model under attack.

**Space of Semantically Equivalent Programs** Generally, we can define a set of semantic-preserving transformations, which in turn induce a space of semantically equivalent programs. For example, we can: (i) rename variables, and (ii) add dead code. There are clearly many other semantic preserving transformations (e.g., re-ordering independent statements), but their application required deeper analysis of the program to guarantee that they are indeed semantic preserving. In this paper we therefore focus on the above two semantic-preserving transformation which can be safely applied without any semantic analysis.

**Brute-Force Exploration of the Program Space** One naïve approach for exploring the space of equivalent program is to simply apply transformations randomly. We can apply transformations randomly to generate new programs, and use the model to predict the label of each generated program. However, the program space to be explored is huge. For example, in the code snippet \( f_1 \) of Figure 1, even considering a single kind of transformation—variable renaming—on a
single variable, leads to a space of more than $10^{11}$ programs. This makes exhaustive exploration prohibitively expensive.

**Gradient-Based Exploration of the Program Space** We need a way to guide exploration of the program space towards a target label (in a targeted attack), or away from the original label (in a non-targeted attack).

In standard SGD-based training of neural networks, the weights of the network are updated to minimize the loss function. The gradient is used to guide the update of the network weights to minimize the loss. However, what we are trying to determine is not an update to the network’s weights, but rather an “update” to the network’s inputs. A natural way to obtain such guidance is to derive the desired prediction with respect to the model’s inputs while holding the model weights constant, and following the gradient to modify the inputs.

In settings where the input is continuous (e.g., images), modifying the input can be done directly by adding a small noise value and following the direction of the gradient towards the desired target label (targeted), or away from the original label (non-targeted). A common technique used for images is the Fast Gradient Signed Method (FGSM) approach, which modifies the input using a small fixed $\epsilon$ value.

**Deriving with respect to a Discrete Input** In settings where the input is discrete, the first layer of a neural network is typically an embedding layer that embeds the discrete object into continuous space [2, 3, 22]. The question for discrete inputs is therefore, *what does it mean to derive with respect to the model’s inputs?*

One approach is to initially ignore the embedding layer and derive with respect to the embedding vector which is the result of the embedding layer. In this approach, after the gradient is obtained, we need to reflect the update of the embedding vector back to discrete-input space. This can be done by looking for nearest-neighbors of the updated embedding vector in the original embedding space, and finding a nearby vector that has a corresponding discrete input. In this approach, there is no guarantee that the nearby input is the best step for following the gradient.

In contrast, our approach is to derive with respect to a one-hot vector that represents the distribution over possible discrete values (e.g., variable names). Intuitively, this allows us to directly obtain the best discrete value for following the gradient (see details in Section 4).

**Targeted gradient-based attack** Using our gradient-based method, we explore the space of semantically equivalent programs directly towards a desired adversarial target. For example, given the code snippet $f_1$ of Figure 1 and the desired target label contains, our approach for generating adversarial examples automatically infers the snippet $f_2$ of Figure 1. Similarly, given the target label escape, our approach automatically infers the snippet $f_3$ of Figure 1.

**Figure 2.** Illustration of Gradient Descent, subscripts denote different time steps: in each step, the gradient is computed w.r.t. $\theta_i$ for calculating a new $\theta_{i+1}$, by updating $\theta_i$ towards the opposite direction of the gradient, until we reach a minimum value of $J$.

All code snippets of Figure 1 are semantically equivalent. The only difference between $f_1$ and $f_2$ is the names of variables. Specifically, these snippets only differ in the name of a single variable which is named array in $f_1$ and ttypes in $f_2$. Nevertheless, when array is renamed to ttypes, the prediction made by Code2Vec changes to the desired (adversarial) target label contains.

The difference between $f_1$ and $f_3$ is the addition of a single variable declaration int upperhexdigits, which is never used in the code snippet. Nevertheless, adding this declaration changes the prediction made by the model to the desired (adversarial) target label escape.

### 3 Background

In this section we provide a fundamental background about neural networks and adversarial examples.

#### 3.1 Training Neural Networks

A Deep Neural Network model (DNN) can be viewed as a function $f : X \rightarrow Y$ where $X$ is the input domain (image, text, code, etc.) and $Y$ is usually a finite group of labels. Assuming a perfect classifier $h^* : X \rightarrow Y$, the goal of the function $f$ is to assign the correct label $y \in Y$ (which determined by $h^*$) for each input $x \in X$. In order to accomplish that, $f$ contains a set of trainable weights (denoted by $\theta$) that can be adjusted to fit a given labeled training set $T = \{(x,y) | x \in X \subset X, y \in Y, y = h^*(x)\}$. The process of adjusting $\theta$ (i.e., training) is done by solving an optimization problem defined by a certain loss function $J(\theta, x, y)$ (usually Mean Square Error or Cross Entropy) which estimates the model’s generalization ability:

$$\theta^* = \arg\min_\theta \sum_{(x,y) \in T} J(\theta, x, y)$$

One of the most common algorithms to approximate the above problem is backpropagation [23] using Gradient Descent [15]. When backpropagation using Gradient Descent is used for training, the following update rule is applied:
repeatedly to update the model’s weights:

\[ \theta_{t+1} = \theta_t - \eta \cdot \nabla \theta J(\theta_t, x, y) \]  

(2)

where \( \eta \) is hyper parameter called learning rate. Intuitively, the Gradient Descent algorithm can be viewed as taking small steps in the steepest descent, until reaching a (possibly local) minimum. This process is illustrated in Figure 2 where \( \theta \) contains a single trainable variable and \( Y \) contains one target (hence a scalar loss function).

3.2 Adversarial Examples

Neural network models are very popular and had been applied in many domains, including computer vision [20, 24, 36, 37], natural language [16, 21, 27], and source code [1–3, 5, 8, 10, 11, 25, 26, 28, 29, 32–34].

However, although DNNs have shown astonishing results in many domains, they were found to be vulnerable to adversarial examples. Adversarial example is an input which intentionally force a given trained model to make a wrong prediction. For DNNs that are trained on images, the adversarial examples are usually achieved by applying a certain hardly perceptible perturbation on a given input image [19, 38].

Recently, attempts were made to find adversarial examples in Natural Language Processing (NLP) domain. However, while adversarial examples on images are easy to generate with impressive results, the generation of adversarial text is harder. This is due to the discrete nature of text and the difficulty to generate semantics-preserving perturbations. One of the approaches to overcome this problem is by replacing words to their synonym [7]. Other approaches insert typos into the words by replacing few characters in text [9, 17]. However, while these approaches yield an adversarial readable text that fools character level models, they don’t fit the code domain for the following reasons: (i) the code vocabulary is larger (token-level vocabulary), (ii) a typo can cause compilation errors and therefore to generate out-of-domain example, (iii) due to the nature and structure of code, many characters replacements can be easily detected and corrected (which make them no-longer “adversarial”), and (iv) many models rely mainly on the code structure (e.g. AST) rather than characters processing.

Additionally, NLP models don’t take into account unique properties of the code, such as: multiple occurrences of variables, patterns in code, the relation between different parts of the code and the readability of the entire code, etc. So, applying NLP methods on code domain makes the hard problem even harder.

4 Adversarial Examples for Models of Code

In this section we describe the process of generating adversarial examples. In Section 4.1 we define basic notations and the types of adversarial attacks. In Section 4.2 and Section 4.3 we focus on each type of attack (“targeted” and “non-targeted”).

4.1 Definitions

Suppose we are given a trained model of code. The given model can be described as a function \( f: C \rightarrow L \), where \( C \) is the set of all code snippets and \( L \) is a set of labels. We define \( \text{Var}(c) \) as the set of all variables existing in \( c \) and We define \( \text{Sym}(c) \) as the set of all identifiers and literals existing in \( c \). For brevity, in this section we focus only on generating adversarial examples by variable renaming of the original variables (\text{VarNAME}). Nonetheless, our approach can also be applied to a code snippet without changing the existing names, and instead adding a redundant variable declaration (\text{DeadCode}, see Section 6.1.1). In such case, our approach can be applied similarly by choosing an initial random name for the new redundant variable, and selecting this variable as the variable we wish to rename.

Given a code snippet \( c \in C \) that the given trained model predicts as \( y \in L \), i.e., \( f(c) = y \), and given an adversarial label \( y_{bad} \in L \), the adversary’s objective is thus to select a single variable \( v \in \text{Var}(c) \) and an alternative name \( v' \), such that renaming \( v \) to \( v' \) will make the model predict the adversarial label: \( f(c_{v \rightarrow v'}) = y_{bad} \).

The advantages of variable renaming as the form of semantic-preserving transformation are that (i) each variable appears in several places in the code, so a single variable renaming can induce multiple perturbations; (ii) the adversarial code can still be compiled and therefore stays in code domain; and (iii) some variables don’t affect the readability of the code and hence renaming them creates unnoticed adversarial examples.

We focus on two distinct types of adversarial examples:

- **Targeted attack** – forces the model to output a specific prediction (which is not the correct prediction).
- **Non-targeted attack** – forces the model to make any incorrect prediction.

In high-level, the main idea in both kinds of attacks lies in the difference between the standard approach of training a neural network using back-propagation and generating adversarial examples: while training a neural network, we derive the loss with respect to the learned parameters and update each learned parameter; in contrast, when generating adversarial examples, we derive the loss with respect to the inputs while holding the learned parameters constant, and update the inputs.

4.2 Targeted attack

In this kind of attack, our goal is to make the model predict an incorrect desired label \( y_{bad} \) by renaming a given variable \( v \) to \( v' \). Let \( \theta \) be the learned parameters of a model, \( c \) be the input code snippet to the model, \( y \) be the target label and \( J(\theta, c, y) \) be the loss function used to train the neural model.
As explained earlier, when training using Gradient Descent, the following rule in being used in order to update the model weights and minimize $J$:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_{\theta} J(\theta_t, x, y)$$

Let $y_{bad}$ be the desired target label. We can apply a Gradient Descent step with $y_{bad}$ as the desired label in the loss function, and derive with respect to any given code variable $v$:

$$v' = v - \eta \cdot \nabla_{v} J(\theta, x, y_{bad})$$

where $v$ is the one-hot vector of $v$. The above action can intuitively be viewed (in Figure 3) as taking a step toward the steepest descent, however now the direction is determined by $y_{bad}$'s loss function.

In fact, the result of the above action does not produce a desired new variable name $v'$, but instead, a distribution over all possible variable names. To concertize the name of

$\bar{v}$, we choose the argmax over the resulting distribution, as illustrated in Figure 4 and detailed in Section 4.4.

**Search** Sometimes, adversarial examples can be found by applying the adversarial step of Equation (4) once, but other times – multiple steps are needed, i.e., replacing $v$ with $v'$ and compute gradient again with respect to $v'$. We limit the number of times we apply the adversarial step by a depth hyper-parameter.

Additionally, the argmax operation of concertizing the distribution over potential names to a single name can be relaxed with “top-k” operation. Namely, instead of taking the name with the maximal gradient, we take the $k$ words with highest gradient. These define a Breadth-First Search (BFS), where the width parameter is defined by the “top-k”.

### 4.3 Non-Targeted Attack

In a non-targeted attack, our goal is to update $v$ to $v'$ in a way that will increase the loss to any direction (instead of decreasing it, as in the training process). Thus, we compute the gradient w.r.t $v$ and use Gradient Ascent:

$$v' = v + \eta \cdot \nabla_{v} J(\theta, x, y)$$

This rule can be illustrated as taking a step toward the steepest ascent in the loss function (Figure 5).

The same BFS search as in targeted attacks (Section 4.2) can be applied here as well.

### 4.4 Deriving by Integer Indices

Usually, in neural models there is no real need to use one-hot vectors at all, since all word embeddings can be stored in a matrix such that each row in the matrix corresponds to a word vector. Looking up a specific vector is then performed by simply looking up a row in the matrix using its index. Nevertheless, in the context of adversarial examples, deriving the loss with respect to a single variable name is equivalent to deriving with respect to an index, which is zero almost everywhere. Thus, instead of using indices, we have to represent variable names using one-hot vectors, because these *can* be derived. Looking up a vector in a matrix can
then be performed by taking the dot product of the embedding matrix with the one-hot vector. Deriving the loss by one-hot vectors instead of indices is thus equivalent to deriving by the (differentiable) distribution over indices, rather than deriving by the index itself.

The result of each adversarial step is thus a distribution over all variable names, in which we select the argmax (Figure 4).

5 Defense against Adversarial Examples

In this section, we describe our proposed approach for defending against name-based adversarial examples.

5.1 Definitions

To tackle adversarial attacks, we suggest to place a defensive-model $g$ before the model of code $f$, which is independent of $g$. The goal of $g$ is to fix the given input (if necessary) in a way that $f$ will predict correctly. Mathematically, the new model can be defined as composing of the two models $f \circ g$.

We assume the adversary has access to the model $f$ being attacked, but not to the defense model $g$. In the rest of this section, we denote $\text{vec}(v)$ as the embedding vector of $v$.

5.2 Defense by Outlier Detection

Since the adversary attacks the model via variable renaming or dead-code, our defense model $G$ tries to identify an outlier variable name and filter it out by replacing it with UNK, hence neutralize the adversarial effect. The main idea is that the adversarial variable name is likely to have low contextual relation to the other, existing, identifiers and literals in code.

We detect outliers by finding an outlier variable in terms of $L_2$ distance among the vectors of the existing variable names. Given a code snippet $c \in C$, we select the variable $z^*$ which is the most distant from the average of the other symbols:

$$z^* = \arg\max_{z \in \text{Var}(c)} \left\| \sum_{v \in \text{Sym}(c), v \neq z} \frac{\text{vec}(v)}{|\text{Sym}(c)|} - \text{vec}(z) \right\|_2$$ (6)

We then define a threshold $\sigma$ that determines whether $z^*$ is an outlier: if the $L_2$ distance between the vector of $z$ and the average of the rest of the symbols is greater than $\sigma - z$ is replaced with an UNK; otherwise, the code snippet is left untouched. Practically, the threshold $\sigma$ is tuned on a validation set, and determines the trade-off between the effectiveness of the defense and the accuracy while not under attack, as we evaluate and discuss in Section 6.3.4.

6 Evaluation

Our set of experiments comprises two parts: (a) evaluating the success of the adversary to change the prediction of the downstream classifier (targeted and non-targeted attacks); and (b) evaluating our proposed defense and its ability to mitigate the attacks.

6.1 Setup

To evaluate the performance of the attack and the defense, we used code2vec [6] as a downstream model. This simple and commonly used model predicts a method name conditioned on its body. The goal of the attack is to thus change the predicted method name by perturbing the method body.

6.1.1 Adversarial Strategies

While there is a variety of possible adversarial perturbations, we focus on two main adversarial strategies:

- **Variable Renaming (VarName):** choose a single variable, and iteratively change its name to change the model’s prediction using a BFS (as explained in Section 4). In general, the adversary can try all variables and choose the most successful one. For simplicity, in the following experiments we chose a variable to rename randomly.
- **Dead-Code Insertion (DeadCode):** performed by inserting a new unused variable declaration and deriving the model with respect to its name. The advantage of this strategy is that the existing code remains unchanged, which might make this attack more difficult to notice. Seemingly, this kind of attack can be mitigated by simply ignoring unused variables. Nonetheless, in the general case, detecting unreachable code is undecidable and it is thus important to evaluate this attack and the defense against it.

In all experiments, we used the adversarial step to run BFS with $\text{width} = 2$ and $\text{depth} = 2$. Increasing $\text{width}$ and $\text{depth}$ can definitely improve the adversary’s success with the cost of longer search.

6.1.2 Dataset

We evaluate our proposed attack and defense on the Java-large dataset [3]. This dataset consists of more than 16M Java methods and their labels, taken from 9500 top-starred Java projects from GitHub that were created since January 2007. It contains 9000 projects for training, distinct 200 projects for validation and distinct 300 project for test. Methods with no local variables or arguments were filtered out from evaluation, since they cannot be perturbed by variable renaming.

To evaluate the effectiveness of our attack, we focus on the examples which the model predicted correctly out of the test set of Java-large. That is, on this filtered test set, the accuracy of the original code2vec model is 100% by construction.

6.2 Attack

We focus on two main attack tasks: targeted and non-targeted attacks. For targeted attacks, we used Equation (4) as the adversarial step. For non-targeted attacks, we used Equation (5) as the adversarial step.
Table 1. Robustness of Code2vec to our adversarial attacks (targeted and non-targeted), compared to the trivial adversary (the lower robustness, the more efficient the attack).

|                  | Trivial Adversary | Our Adversary |
|------------------|-------------------|---------------|
|                  | Robustness        | Robustness    |
|                  | VarName | DeadCode | VarName | DeadCode |
| Non-targeted     | 34.10   | 54.90   | 6.00    | 21.83    |
| init             | 84.47   | 96.79   | 48.44   | 87.62    |
| merge|from    | 72.79   | 99.82   | 10.39   | 22.65   |
| size             | 99.47   | 99.97   | 78.27   | 93.96    |
| isempty          | 88.61   | 99.98   | 79.04   | 87.63    |
| clear            | 89.56   | 99.07   | 82.8    | 97.89    |
| remove           | 84.94   | 99.29   | 63.15   | 80.02    |
| value            | 99.77   | 100.0   | 76.75   | 98.33    |
| load             | 86.75   | 99.03   | 55.65   | 86.65    |
| add              | 92.88   | 99.93   | 68.6    | 93.75    |
| run              | 95.11   | 99.36   | 51.52   | 77.63    |

For the desired label of targeted attacks, we randomly sampled labels which occurred at least 10k times in the training set.

6.2.1 Metrics
We measure the robustness of each setting to each of the different attack approaches (the lower model robustness – the higher effectiveness of the attack).

In targeted attacks, the goal of the adversary is to change the prediction of the model to a label of the attacker’s desire. We thus define robustness as the percentage of examples in which the correctly predicted label was not changed to the adversary’s desired label. If the predicted label was changed to a label that is not the adversarial label, we consider the model as robust to the targeted attack.

In non-targeted attacks, the goal of the adversary is to change the prediction of the model to any label other than the correct label. We thus define robustness as the percentage of examples in which the correctly predicted label was changed to any other label than the correct label.

6.2.2 Baselines
Since our task is new, we are not aware of existing baselines. We thus compare our attack to the Trivial Adversary which takes slightly different steps in targeted and non-targeted attacks:

In targeted attacks, the trivial adversary replaces a variable with the desired adversarial label. For example, if the adversarial label is contains – the baselines renames a variable to contains as well.

In non-targeted attacks, the trivial adversary replaces the given variable with a randomly selected variable name from the vocabulary.

6.2.3 Attack - Results
Table 1 summarizes the results of our attack approach. Our adversary outperforms the baseline adversary in both targeted and non-targeted attacks.

For non-targeted attacks, our attacks are more effective than the trivial adversary: the robustness scores to our attack are 6% to VarName and 21.83% to DeadCode, while the robustness to the trivial adversary is 34.10% and 54.90% to VarName and DeadCode, respectively. Thus, our non-targeted attack is more effective.

In targeted attacks, our adversary performs better than the trivial adversary for each adversarial label. Additionally, not all targets behave the same way, and different targets were easier to achieve by adversarial examples than others.

Non-targeted attacks are easier than targeted attacks and generally yield lower model robustness. This is expected, as the goal of non-targeted attacks is to change the label to any label other than the correct one, while targeted attacks are measured as successful only if they change the prediction to the desired label.

In general, the VarName attack is more lethal than DeadCode. We hypothesize that the reason is that the inserted unused variable impacts only a small part of the code, hence leading to a small change that might not affect the computation of the model enough. In contrast, renaming a variable changes multiple occurrences in the code and thus has a wider effect.

6.3 Defense
We composed our similarity defense (as described in Section 5) before a downstream Code2vec model.
Table 2. Precision, Recall, F1 and robustness percentage of different models. The higher robustness, the more effective the defense.

|                | No Defense | Our Defense | No Vars | Train Without Vars |
|----------------|------------|-------------|---------|--------------------|
| Performance    |            |             |         |                    |
| (not under attack) | 100        | 98.18       | 78.78   | 89.74              |
| Precision      | 100        | 97.75       | 80.83   | 90.86              |
| Recall         | 100        | 97.92       | 79.98   | 90.4               |
| F1             |            |             |         |                    |
| VarName Robustness (%) | 6.0        | 74.35       | 100     | 100                |
| Targeted: "run" | 51.52     | 96.49       | 100     | 100                |
| DeadCode Robustness (%) | 22.83 | 96.73       | 100     | 100                |
| Targeted: "run" | 77.63 | 95.76       | 100     | 100                |

Table 3. Robustness of CODE2VEC to adversarial attacks with our defense, across different adversarial targets (the higher the robustness – the more effective the defense). When our defense is used, robustness rises dramatically.

|                | Our Adversary | Our Adversary + Our Defense |
|----------------|--------------|-------------------------------|
| Robustness     | VarName      | DeadCode                      | VarName      | DeadCode                      |
| init           | 48.44        | 87.62                         | 74.32        | 91.41                         |
| merge|from       | 10.39                        | 22.65        | 99.98                        |
| size           | 78.27        | 95.99                         | 99.58        | 99.94                         |
| is|empty       | 79.04                        | 87.63        | 99.22                        |
| clear          | 82.8         | 97.89                         | 98.77        | 99.56                         |
| remove         | 63.15        | 80.02                         | 94.5         | 99.33                         |
| value          | 76.75        | 98.33                         | 90.87        | 99.72                         |
| load           | 55.65        | **86.65**                     | 60.27        | 85.28                         |
| add            | 68.6         | 93.75                         | 88.97        | 97.69                         |
| run            | 51.52        | 77.63                         | 96.49        | 95.76                         |

6.3.1 Metrics and the Performance-Defense trade-off

We measure the success rate of the different approaches in preventing the adversarial attack and improving the robustness of the original model (without any defense). When evaluating alternative defenses, it is also important to measure the performance of the original model while using the defense, but not under attack: a too-defensive approach can lead to 100% robustness, at the cost of reduced prediction performance.

To tune the threshold $\sigma$ of our defense, we thus balance the following factors: (1) the robustness of the model using our defense; and (2) the F1 score the model using our defense while not under attack.

We tuned the threshold on the validation set. We chose $\sigma = 2.7$ since it leads to 75% robustness against non-targeted attack with the cost of 2% degradation in F1 score while not under attack. However, this threshold can be tuned according to the desired needs in the trade-off between performance and defense (see Section 6.3.4).

6.3.2 Baselines

We compare our defense to the following settings:

- **No Defense** - applies the attack directly on the downstream model.
- **Our Defense** - is our proposed similarity defense.
- **No Vars** - is a conservative defensive baseline which replaces all variables to an UNK symbol only at test time. This approach is 100% robust by construction, but does not leverage variable names for prediction.
- **Train Without Vars** - replaces all variables with an UNK symbol both at training and test time. This approach is also 100% robust by construction. It is expected to perform better than No Vars in terms of F1 because it is trained not to rely on variable names, and use other signals instead. However, the down side is that it requires training a model from scratch, while the other defense approaches can be applied to an already-trained model.

6.3.3 Defense - Results

The effectiveness of our defense compared to the baselines is presented in Table 2.

The main result is that **Our Defense** mitigates both targeted and non-targeted attacks with 74.35%-96.49% VarName robustness, respectively (Table 2), compared to 6.0%-51.52% of No Defense. A similar effect exists in DeadCode attacks.
The penalty of using our defense is only 2.18% degradation in F1 score compared to No Defense. On the other extreme, Train Without Vars is 100% robust to VarName and DeadCode attacks, but its F1 is degraded by 9.6% compared to No Defense. That is, Our Defense is a sweet spot in the trade-off between performance and robustness: it is almost as accurate as the original model (No Defense), and almost as robust as the model that ignores variables completely.

No Vars is 100% robust as Train Without Vars, but is about 10 F1 points worse than Train Without Vars. The only benefit of No Vars over Train Without Vars is that No Vars can be applied to a model without re-training it.

Table 3 shows the performance of the defense across various adversarial labels.

### 6.3.4 Robustness - Performance trade-off

Figure 6 shows the trade-off between robustness and performance, with respect to the similarity threshold $\sigma$. With small values of $\sigma$, the model with Our Defense is more robust, almost as robust as No Vars, but performs worse in terms of F1 score; as the value of $\sigma$ increases, the model with Our
Defense becomes less robust, but performs better while not under attack.

6.4 Examples
Figure 7 shows additional targeted attacks against the "sort" example from Figure 1: renaming the variable array to mstyleids changes the prediction to get with probability of 99.99%; renaming array to possiblematches changes the prediction to indexOf with probability of 86.99%. The predicted adversarial labels (get and indexOf) were chosen arbitrarily before finding the necessary variable name replacement. Figures 8-10 demonstrate additional attacks of this kind.

Occasionally, a dead code insertion has the same effect even in different examples. This is demonstrated in Figure 11: adding the unused variable declaration int introsorter = 0; to each of the snippets on the left changes their prediction to sort with probability of 100%. This effect reminds of the Adversarial Patch [12], that was shown to force an image classifier to predict a specific label, regardless of the input example.

7 Related Work
Szegedy et al. [38] discovered that many models, especially deep neural networks, are vulnerable to adversarial examples. They showed that by using Box-constrained L-BFGS search on image space, they could cause an image classification model to misclassify an image by applying a certain hardly perceptible perturbation. Goodfellow et al. [19] introduced an efficient way (called "fast gradient sign method") for generating adversarial examples. Their method was based on adding an imperceptibly small vector, whose elements are equal to the sign of the elements of the gradient of the cost function with respect to the input.

Other work attempted to find adversarial examples in natural language processing domain. However, while adversarial examples on images are easy to generate with impressive results, the generation of adversarial text is harder. The main factor is that images are continuous, and can thus be added hardly perceptible perturbation – natural language text is discrete, and thus cannot be easily perturbed. One of the approaches to overcome this problem is by replacing words to their synonym [7]. Other approaches inserted typos into words by replacing few characters in text [9, 17].

There are different approaches to mitigate adversarial examples. A related idea to our defense was proposed for natural language by Pruthi et al. [30]: they found that state-of-the-art NLP models perform much worse when the input contains spelling mistakes. To make these models more robust to misspellings, the authors placed a character-level word recognition model in front of the downstream model. This word recognition model fixes misspellings before they are fed into the downstream model. Our defense is similar in spirit since it uses a composition of a downstream model following an upstream defense model; the main difference is that the goal of our defensive model is more difficult: our defensive model needs to detect outlier names, which is more difficult than identifying character-level typos.

8 Conclusion
We presented the first approach for targeted attacks of models of code using adversarial examples. Our approach is a general white-box technique that can work for any model of code in which we can compute gradients. The main idea is to force a given trained model to make a prediction of the adversary’s choice by introducing small perturbations that do not change program semantics. We find these perturbations by deriving the desired prediction with respect to the model’s inputs while holding the model weights constant, and following the gradients to slightly modify the input.

We further present a defensive model that can be placed in front of the downstream model. The defensive model detects unlikely mutations and masks them before feeding the input to the downstream model.

We show that our attack succeeds in changing a prediction to the adversary’s desire ("targeted attack") up to 89% of the times, and succeeds in changing a given prediction to any incorrect prediction ("non-targeted attack") 94% of the times. By using our proposed defense, the success rate of the attack drops drastically for both targeted and non-targeted attacks, with a minor penalty of 2% relative degradation in accuracy while not performing under attack.

We believe that the principles presented in this work can help in training more accurate and more robust models of code. Furthermore, they are crucial for models of code deployed in realistic and production environments. To this end, we will make all our code and trained models publicly available.
Figure 7. A snippet classified correctly as sort by the model of code2vec.org. The same example is classified as get by renaming array to mstyleids and is classified as indexOf by renaming array to possiblematches.

Figure 8. A snippet classified correctly as contains by the model of code2vec.org. The same example is classified as escape by renaming elem to upperhexdigits and is classified as load by renaming elem to musicservice.
int f(String target, ArrayList<String> array) {
    int count = 0;
    for (String str: array) {
        if (target.equals(str)) {
            count++;
        }
    }
    return count;
}

Prediction: count (42.77%)

int count(String target, ArrayList<String> orderedlist) {
    int count = 0;
    for (String str: orderedlist) {
        if (target.equals(str)) {
            count++;
        }
    }
    return count;
}

Prediction: sort (51.55%)

int f(String thisentry, ArrayList<String> array) {
    int count = 0;
    for (String str: array) {
        if (thisentry.equals(str)) {
            count++;
        }
    }
    return count;
}

Prediction: contains (99.99%)

Figure 9. A snippet classified correctly as count by the model of code2vec.org. The same example is classified as sort by renaming array to orderedlist and is classified as contains by renaming target to thisentry.

String f(String txt) {
    txt = replace(txt, "\s", "\&amp;");
    txt = replace(txt, "\"", "\&quot;);
    txt = replace(txt, "<", "\&lt;");
    txt = replace(txt, ">", "\&gt;");
    return txt;
}

Prediction: escape (73.55%)

String f(String expres) {
    expres = replace(expres, "\s", "\&amp;");
    expres = replace(expres, "\"", "\&quote;");
    expres = replace(expres, "<", "\&lt;");
    expres = replace(expres, ">", "\&gt;");
    return expres;
}

Prediction: done (77.68%)

String f(String claimed) {
    claimed = replace(claimed, "\s", "\&amp;");
    claimed = replace(claimed, "\"", "\&quote;");
    claimed = replace(claimed, "<", "\&lt;");
    claimed = replace(claimed, ">", "\&gt;");
    return claimed;
}

Prediction: contains (94.99%)

Figure 10. A snippet classified correctly as escape by the model of code2vec.org. The same example is classified as contains by renaming txt to expres and is classified as done by renaming txt to claimed.
String[] f(final String[] array) {
    final String[] newArray = new String[array.length];
    for (int index = 0; index < array.length; index++) {
        newArray[array.length - index - 1] = array[index];
    }
    return newArray;
}

String[] f(final String[] array) {
    int introsorter = 0;
    final String[] newArray = new String[array.length];
    for (int index = 0; index < array.length; index++) {
        newArray[array.length - index - 1] = array[index];
    }
    return newArray;
}

Prediction: reverseArray (77.34%)

int f(Object target) {
    int i = 0;
    for (Object elem: this.elements) {
        if (elem.equals(target)) {
            return i;
        }
        i++;
    }
    return -1;
}

int f(Object target) {
    int introsorter = 0;
    int i = 0;
    for (Object elem: this.elements) {
        if (elem.equals(target)) {
            return i;
        }
        i++;
    }
    return -1;
}

Prediction: indexOf (86.99%)

Prediction: sort (100%)

Figure 11. Adding the dead code `int introsorter = 0;` to each of the snippets on the left changes their label to `sort` with confidence of 100%.
References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In International Conference on Learning Representations. https://openreview.net/forum?id=BJOFETxR-

[2] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. 2091–2100. http://jmlr.org/proceedings/papers/v48/allamanis16.html

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. Code2Seq: Generating Sequences from Structured Representations of Code. In International Conference on Learning Representations. https://openreview.net/forum?id=H1gKYo09tX

[4] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2019. Structural Language Models for Any-Code Generation. arXiv preprint arXiv:1910.00577 (2019).

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-based Representation for Predicting Program Properties. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018). ACM, New York, NY, USA, 404–419. https://doi.org/10.1145/3192366.3192412

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. Proc. ACM Program. Lang. 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[7] 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).

[8] Rohan Bavishi, Michael Pradel, and Koushik Sen. 2018. Context2Name: A deep learning-based approach to infer natural variable names from usage contexts. arXiv preprint arXiv:1809.05193 (2018).

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

[10] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016. 2933–2942. http://jmlr.org/proceedings/papers/v48/bielik16.html

[11] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In International Conference on Learning Representations. https://openreview.net/forum?id=Bke4KsA5FX

[12] Tom B Brown, Dandelion Mané, Aurko Roy, Martin Abadi, and Justin Gilmer. 2017. Adversarial patch. arXiv preprint arXiv:1712.09665 (2017).

[13] Jose Cambrerono, Hougyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When Deep Learning Met Code Search. arXiv preprint arXiv:1905.03813 (2019).

[14] Nicholas Carlini and David Wagner. 2018. Audio adversarial examples: Targeted attacks on speech-to-text. In 2018 IEEE Security and Privacy Workshops (SPW). IEEE, 1–7.

[15] Augustin Cauchy. [n.d.]. Méthode générale pour la résolution des systèmes d’équations simultanées. (In d.).

[16] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 (2014).

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

[18] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2019. Deep program search. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, Berlin, Germany, Volume 1: Long Papers. http://aclweb.org/anthology/P/P16/P16-1195.pdf

[19] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572 (2014).

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition. 770–778.

[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. Neural computation 9, 8 (1997), 1735–1780.

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

[23] Henry J Kelley. 1960. Gradient theory of optimal flight paths. Ars Journal 30, 10 (1960), 947–954.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[25] Jason Liu, Seohyun Kim, Vijayaraghavan Murali, Swarat Chaudhuri, and Satish Chandra. 2019. Neural Query Expansion for Code Search. In Proceedings of the 3rd ACM SIGPLAN International Workshop on
Machine Learning and Programming Languages (MAPL 2019). ACM, New York, NY, USA, 29–37. https://doi.org/10.1145/3315508.3329975

[26] Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. 2017. Data-Driven Program Completion. CoRR abs/1705.09042 (2017). arXiv:1705.09042 http://arxiv.org/abs/1705.09042

[27] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In Eleventh annual conference of the international speech communication association.

[28] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Sketch Learning for Program Synthesis. CoRR abs/1703.05698 (2017). arXiv:1703.05698 http://arxiv.org/abs/1703.05698

[29] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-based Bug Detection. Proc. ACM Program. Lang. 2, OOPSLA, Article 147 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276517

[30] Danish Pruthi, Bhuwan Dhingra, and Zachary C Lipton. 2019. Combating Adversarial Misspellings with Robust Word Recognition. ACL (2019).

[31] Md Rafiul Islam Rabin, Ke Wang, and Mohammad Amin Alipour. [n.d.]. Testing Neural Program Analyzers. ([n. d.]).

[32] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from “Big Code”. In Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’15). ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

[33] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting argument selection defects. Proceedings of the ACM on Programming Languages 1, OOPSLA (2017), 104.

[34] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. 31–41. https://doi.org/10.1145/3211346.3211353

[35] Andrew Scott, Johannes Bader, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. arXiv preprint arXiv:1902.06111 (2019).

[36] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014).

[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition. 1–9.

[38] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013).