Adversarial Examples for Models of Code

NOAM YEFET, Technion, Israel
URI ALON, Technion, Israel
ERAN YAHAV, Technion, Israel

Neural models of code have shown impressive results when performing tasks such as predicting method names and identifying certain kinds of bugs. We show that these models are vulnerable to *adversarial examples*, and introduce a novel approach for *attacking* trained models of code using adversarial examples. The main idea of our approach is to force a given trained model to make an incorrect prediction, as specified by the adversary, by introducing small perturbations that do not change the program's semantics, thereby creating an adversarial example. To find such perturbations, we present a new technique for Discrete Adversarial Manipulation of Programs (DAMP). DAMP works 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 code.

We show that our DAMP attack is effective across three neural architectures: CODE2VEC, GGNN, and GNN-FILM, in both Java and C#. Our evaluations demonstrate that DAMP has up to 89% success rate in changing a prediction to the adversary's choice (a targeted attack) and a success rate of up to 94% in changing a given prediction to any incorrect prediction (a non-targeted attack). To defend a model against such attacks, we empirically examine a variety of possible defenses and discuss their trade-offs. We show that some of these defenses can dramatically drop the success rate of the attacker, with a minor penalty of 2% relative degradation in accuracy when they are not performing under attack.

Our code, data, and trained models are available at https://github.com/tech-srl/adversarial-examples .

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Computing methodologies** → **Neural networks**.

Additional Key Words and Phrases: Adversarial Attacks, Targeted Attacks, Neural Models of Code

ACM Reference Format:
Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial Examples for Models of Code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 162 (November 2020), 30 pages. https://doi.org/10.1145/3428230

1 INTRODUCTION

Neural models of code have achieved state-of-the-art performance on various tasks such as the prediction of variable names and types [Allamanis et al. 2018; Alon et al. 2018; Bielik et al. 2016; Raychev et al. 2015], code summarization [Allamanis et al. 2016; Alon et al. 2019a; Fernandes et al. 2019], code generation [Alon et al. 2019b; Brockschmidt et al. 2019; Murali et al. 2017], code search [Cambronero et al. 2019; Liu et al. 2019; Sachdev et al. 2018], and bug finding [Pradel and Sen 2018; Rice et al. 2017; Scott et al. 2019].

In other domains such as computer vision, deep models have been shown to be vulnerable to *adversarial examples* [Goodfellow et al. 2014b; Szegedy et al. 2013]. Adversarial examples are inputs

Authors' addresses: Noam Yefet, Technion, Israel, snyefet@cs.technion.ac.il; Uri Alon, Technion, Israel, urialon@cs.technion.ac.il; Eran Yahav, Technion, Israel, yahave@cs.technion.ac.il.

2475-1421/2020/11-ART162
https://doi.org/10.1145/3428230

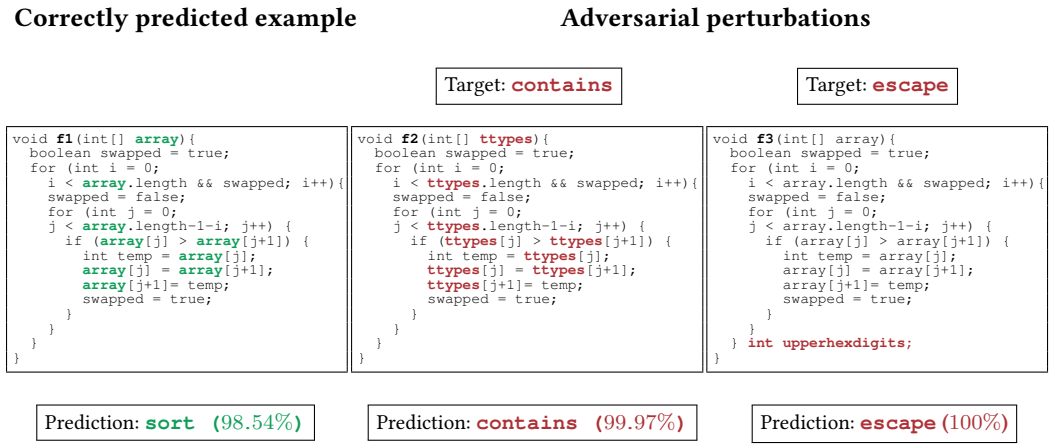

Fig. 1. A Java snippet `f1` is classified correctly as `sort` by the model of code2vec.org. Given `f1` and the target `contains`, our approach generates `f2` by renaming `array` to `ttypes`. Given the target `escape`, our approach generates `f3` by adding an unused variable declaration of `int upperhexdigits`. Additional examples can be found in Appendix A.

crafted by an adversary to force a trained neural model to make a certain (incorrect) prediction. The generation of adversarial examples was demonstrated for image classification [Goodfellow et al. 2014a,b; Kurakin et al. 2016; Mirza and Osindero 2014; Moosavi-Dezfooli et al. 2016; Nguyen et al. 2015; Papernot et al. 2017, 2016; Szegedy et al. 2013] and for other domains [Alzantot et al. 2018a,b; Belinkov and Bisk 2017; Carlini and Wagner 2018; Ebrahimi et al. 2017; Pruthi et al. 2019; Taori et al. 2019]. The basic idea underlying many of the techniques is to 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 (i.e., with the addition of noise). Adding noise to a *continuous object* to change the prediction of a model is relatively easy to achieve mathematically. For example, for an image, this can be achieved by changing the intensity of pixel values [Goodfellow et al. 2014b; Szegedy et al. 2013]. Unfortunately, this does not carry over to the domain of programs, since a program is a *discrete object* that must maintain semantic properties.

In this paper, we present a novel approach for generating adversarial examples for neural models of code. More formally:

1.1 Goal

Given a program $\mathcal{P}$ and a correct prediction y made by a model $\mathcal{M}$, such that: $\mathcal{M}(\mathcal{P}) = y$, our goal is to find a semantically equivalent program $\mathcal{P}'$ such that $\mathcal{M}$ makes a given adversarial prediction y_{bad} of the adversary's choice: $\mathcal{M}(\mathcal{P}') = y_{bad}$.

The main challenge in tackling the above goal lies in exploring the vast space of programs that are semantically equivalent to $\mathcal{P}$, and finding a program for which $\mathcal{M}$ will predict y_{bad}.

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 would require a deeper analysis of the program to guarantee that they are indeed semantic preserving. In this paper, therefore, we focus on the above two semantic-preserving transformations, which can be safely applied without any semantic analysis.

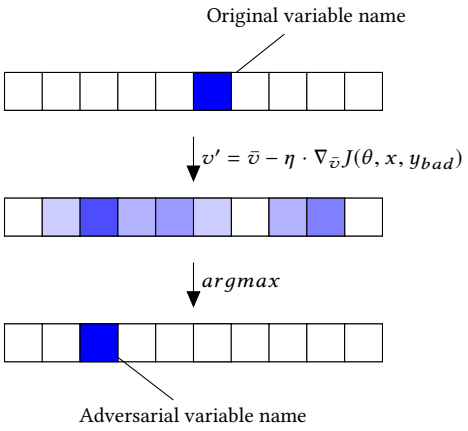

Fig. 2. Perturbing a variable name: the original variable name is represented as a one-hot vector over the variable-name vocabulary. After perturbation, the vector is no longer one-hot. We apply argmax to find the most likely adversarial name, resulting with another one-hot vector over the variable-name vocabulary.

One naïve approach for exploring the space of equivalent programs is to randomly apply transformations using brute-force. We can apply transformations randomly to generate new programs and use the model to make a prediction for each generated program. However, the program space to be explored is exponentially large, making exhaustive exploration prohibitively expensive.

1.2 Our Approach

We present a new technique called Discrete Adversarial Manipulation of Programs (DAMP). The main idea in DAMP is to select semantic preserving perturbations by deriving the output distribution of the model with respect to the model's input and following the gradient to modify the input, while keeping the model weights constant. Given a desired adversarial label y_{bad} and an existing variable name, we derive the loss of the model with y_{bad} as the correct label, with respect to the one-hot vector of the input variable. We then take the argmax of the resulting gradient to select an alternative variable name, rename the original variable to the alternative name, check whether this modification changes the output label to the desired adversarial label, and continue iteratively. This process is illustrated in Figure 2, and detailed in Section 4.

This iterative process allows DAMP to modify the program in a way that preserves its semantics but will cause a model to make adversarial predictions. We show that models of code are susceptible to *targeted attacks* that force a model to make a specific incorrect prediction chosen by the adversary, as well as to simpler *non-targeted attacks* that force a model to make *any incorrect prediction* without a specific target prediction in mind. Our approach is a "white-box" approach, since it assumes the attacker has access to either the model under attack or to a similar model. [1] Under this assumption, our approach is general and *is applicable to any model that can be derived with respect to its inputs* i.e., any neural model. We do not make any assumptions about the internal details or specific architecture of the model under attack.

To mitigate these attacks, we evaluate and compare a variety of *defensive* approaches. Some of these defenses work by re-training the model using another loss function or a modified version of

[1] As recently shown by Wallace et al. [2020], this is a reasonable assumption. An attacker can imitate the model under attack by: training an imitation model using labels achieved by querying the original model; crafting adversarial examples using the imitation model; and transferring these adversarial examples back to the original model.

Correctly predicted example

```
struct TypePair : IEquatable<TypePair>
{
  public static TypePair Create<TSource,
      TDestination>(TSource source,
        TDestination destination, ...)
  {
    ...
  }

  ...

  public Type SourceType { get; }
  public Type DestinationType { get; }
  public bool Equals(TypePair other) =>
    SourceType == other.SourceType
    && DestinationType
        == other.DestinationType;
}
```

(a)

Adversarial perturbation

Target: `SourceType`

```
struct TypePair : IEquatable<TypePair>
{
  public static TypePair Create<TSource,
      TDestination>(TSource source,
        TDestination scsqbhj, ...)
  {
    ...
  }

  ...

  public Type SourceType { get; }
  public Type DestinationType { get; }
  public bool Equals(TypePair other) =>
    SourceType == other.SourceType
    && DestinationType
        == other.SourceType;
}
```

(b)

Fig. 3. A C# VarMisuse example which is classified correctly as `DestinationType` in the method `Equals` by the GGNN model of Allamanis et al. [2018]. Given the code in Figure 3a and the target `SourceType`, our approach renames a local variable `destination` in *another method* to the specific name `scsqbhj`, making the model predict the wrong variable in the method `Equals`, thus ("maliciously") introducing a real bug in the method `Equals`. Additional examples are shown in Appendix A.

the same dataset. Other defensive approaches are "modular", in the sense that they can be placed in front of an already-trained model, identify perturbations in the input, and feed a masked version of the input into the vulnerable model. These defense mechanisms allow us to trade off the accuracy of the original model for improved robustness.

Main Contributions The contributions of this paper are:

- The first technique for generating *targeted* adversarial examples for models of code. Our technique, called Discrete Adversarial Manipulation of Programs (DAMP), is general and only requires that the attacker is able to compute gradients in the model under attack (or in a similar model). DAMP is effective in generating both targeted and non-targeted attacks.
- An experimental evaluation of attacks on three neural architectures: code2vec [Alon et al. 2019c], GGNN [Allamanis et al. 2018], and GNN-FiLM [Brockschmidt et al. 2019] in two languages: Java and C#. Our evaluation shows that our adversarial technique can change a prediction according to the adversary's will ("targeted attack") up to 89% of the time, and is successful in changing a given prediction to an incorrect prediction ("non-targeted attack") 94% of the time.
- A thorough evaluation of techniques for defending models of code against attacks that perturb names, and an analysis of their trade-offs. When some of these defenses are used, the success rate of the attack drops drastically for both targeted and non-targeted attacks, with a minor penalty of 2% in accuracy.

2 OVERVIEW

In this section, we provide an informal overview.

2.1 Motivating Examples

We begin by demonstrating our technique on two examples, which address two different tasks, using two different neural models, and in two programming languages (Java and C#).

Bypass Semantic Labeling (CODE2VEC - Java) We demonstrate how our approach can force the CODE2VEC [Alon et al. 2019c] model to predict a label of our choice. Consider the code snippet `f1` of Figure 1. This code snippet sorts a given array. The CODE2VEC model [Alon et al. 2019c] applied to this code snippet predicts the correct name, `sort`, with a probability of 98.54%.

Given the code snippet `f1` and the arbitrarily chosen adversarial target `contains`, our approach finds that renaming the original variable `array` to `ttypes` in `f2` forces the model to predict the label `contains` with an even higher probability, although the functionality remained the same. We denote this as a VARNAME attack.

Given the code snippet `f1` and the adversarial target `escape`, our approach finds that adding an unused variable (i.e., dead code) named `upperhexdigits` to the end of `f1` forces the model to predict the label `escape` with a probability of 100% (see `f3` in Figure 1). We denote this attack as a DEADCODE targeted attack.

In general, bypassing semantic labeling models can allow a variety of malicious behaviors. For example, a malicious developer can spread malicious code to users by having the code classified as "benign" by a malware detection model [Arp et al. 2014; David et al. 2019; Grosse et al. 2016; Saxe and Berlin 2015; Wang et al. 2017]. In this paper, we demonstrate the ability to bypass semantic labeling by applying our attack on the CODE2VEC model (Figure 1), forcing the model to predict a label of our choice.

Bypass Bug Detection (VARMISUSE - C#) As another example, we demonstrate how our approach can force a Graph Neural Network (GNN) model to choose an obvious bug as the correct completion. In Figure 3a, a GNN model trained on the VARMISUSE task [Allamanis et al. 2018; Brockschmidt 2019] in C# correctly chooses to "fill the blank" using the field `DestinationType` inside the method `Equals`. By renaming a local variable called `destination` in *another method* to the specific name `scsqbhj` (Figure 3b), the model chooses the incorrect field `SourceType` in the method `Equals`. The fields `DestinationType` (correct) and `SourceType` (incorrect) both have the same type; thus, the code still compiles and the attack causes a real bug in `Equals`.

More generally, bypassing a bug detection model [Pradel and Sen 2018; Rice et al. 2017; Scott et al. 2019] can allow a malicious developer inside an organization or inside an open-source project to intentionally introduce bugs. In this paper, we demonstrate this ability using the VARMISUSE on Graph Neural Networks (GNNs) (Figure 16), forcing the model to choose an incorrect (but type-correct) variable.

In addition to the CODE2VEC and VARMISUSE tasks that we address in this paper, we believe adversarial examples can be applied to neural code search [Cambronero et al. 2019; Liu et al. 2019; Sachdev et al. 2018]. A developer can attract users to a specific library or an open-source project by introducing code that will be disproportionately highly ranked by a neural code search model.

2.2 Discrete Adversarial Manipulation of Programs (DAMP)

Consider the code snippet `f1` of Figure 1 that sorts a given array. The CODE2VEC model [Alon et al. 2019c] 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 of our choice.

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

In standard stochastic gradient descent (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 of the network's weights, but rather an "update" of 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 follow 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) [Goodfellow et al. 2014b] approach, which modifies the input using a small fixed ϵ 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 discrete objects, such as names and tokens, into a continuous space [Allamanis et al. 2016; Alon et al. 2019a; Iyer et al. 2016]. The input is the index of the symbol, which is used to look up its embedding in the embedding matrix. The question for discrete inputs is therefore: *what does it mean to derive with respect to the model's inputs?*

One approach is to 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 the 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 following the gradient is the best step.

In contrast, our Discrete Adversarial Manipulation of Programs (DAMP) approach derives with respect to a one-hot vector that represents the *distribution* over discrete values (e.g., over variable names). Instead of deriving by the input itself, the gradient is taken with respect to the *distribution* over the inputs. Intuitively, this allows us to directly obtain the best discrete value for following the gradient.

Targeted Gradient-based Attack Using our gradient-based method, we explore the space of semantically equivalent programs *directly toward* a desired adversarial target. For example, given the code snippet `f1` of Figure 1 and the desired target label `contains`, our approach for generating adversarial examples automatically infers the snippet `f2` of Figure 1. Similarly, given the target label `escape`, our approach automatically infers the snippet `f3` of Figure 1.

All code snippets of Figure 1 are semantically equivalent. The only difference between `f1` and `f2` is the name of the variables. Specifically, these snippets differ only in the name of a single variable, which is named `array` in `f1` and `ttypes` in `f2`. Nevertheless, when `array` is renamed to `ttypes`, the prediction made by CODE2VEC changes to the desired (adversarial) target label `contains`. The difference between `f1` and `f3` 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 fundamental background on neural networks and adversarial examples.

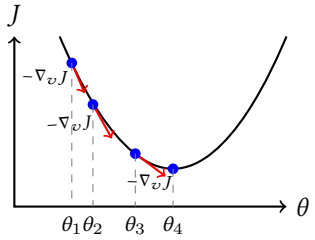

Fig. 4. Illustration of gradient descent, subscripts denote different time steps: in each step, the gradient is computed w.r.t. θ_t for calculating a new θ_{t+1} by updating θ_t towards the opposite direction of the gradient, until we reach a (possibly local) minimum value of J.

3.1 Training Neural Networks

A Neural Network (NN) model can be viewed as a function $f_\theta : X \rightarrow Y$, where X is the input domain (image, text, code, etc.) and Y is usually a finite set of labels. Assuming a perfect classifier $h^* : X \rightarrow Y$, the goal of the function f_θ is to assign the correct label $y \in Y$ (which is determined by h^*) for each input $x \in X$. In order to accomplish that, f_θ contains a set of *trainable weights*, denoted by θ, which can be adjusted to fit a given labeled training set $T = \{(x, y) | x \in \bar{X} \subset X, y \in Y, y = h^*(x)\}$. The process of adjusting θ (i.e., *training*) is done by solving an optimization problem defined by a certain loss function $J(\theta, x, y)$; this is usually mean squared error (MSE) or cross entropy and is used to estimate the model's generalization ability:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{(x,y)\in T} J(\theta, x, y) \tag{1}$$

One of the most common algorithms to approximate the above problem is *gradient descent* [Cauchy [n. d.]] using *backpropagation* [Kelley 1960]. When gradient descent is used for training, the following update rule is applied repeatedly to update the model's weights:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta_t, x, y) \tag{2}$$

where $\eta \in \mathcal{R}$ is a small scalar hyperparameter called *learning rate*, for example, 0.001. Intuitively, the gradient descent algorithm can be viewed as taking small steps in the direction of the steepest descent, until a (possibly local) minimum is reached. This process is illustrated in Figure 4, where θ contains a single trainable variable $\theta \in \mathcal{R}$.

3.2 Adversarial Examples

Neural network models are very popular and have been applied in many domains, including computer vision [He et al. 2016; Krizhevsky et al. 2012; Simonyan and Zisserman 2014; Szegedy et al. 2015], natural language [Cho et al. 2014; Hochreiter and Schmidhuber 1997; Mikolov et al. 2010], and source code [Allamanis et al. 2018, 2016; Alon et al. 2019a, 2018; Bavishi et al. 2018; Bielik et al. 2016; Brockschmidt et al. 2019; Liu et al. 2019; Lu et al. 2017; Murali et al. 2017; Pradel and Sen 2018; Raychev et al. 2015; Rice et al. 2017; Sachdev et al. 2018].

Although neural networks have shown astonishing results in many domains, they were found to be vulnerable to adversarial examples. An adversarial example is an input which intentionally forces a given trained model to make an incorrect prediction. For neural networks that are trained on continuous objects like images and audio, the adversarial examples are usually achieved by applying a small perturbation on a given input image [Carlini and Wagner 2018; Goodfellow et al.

2014b; Szegedy et al. 2013]. This perturbation is found using gradient-based methods: usually by deriving the desired loss with respect to the neural network's inputs.

In *discrete* problem domains such as natural language or programs, generating adversarial examples is markedly different. Specifically, the existing techniques and metrics do not hold, because discrete symbols cannot be perturbed with imperceptible adversarial noise. Additionally, in discrete domains, deriving the loss with respect to its discrete input results in a gradient of zeros.

Recently, attempts were made to find adversarial examples in the domain of Natural Language Processing (NLP). Although adversarial examples on images are easy to generate, the generation of adversarial text is harder. This is due to the discrete nature of text and the difficulty of generating semantic-preserving perturbations. One of the approaches to overcome this problem is to replace a word with a synonym [Alzantot et al. 2018b]. Another approach is to insert typos into words by replacing a few characters in the text [Belinkov and Bisk 2017; Ebrahimi et al. 2017]. However, these NLP approaches allow only *non-targeted* attacks.

Additionally, NLP approaches cannot be directly applied to code. NLP models don't take into account the unique properties of code, such as: multiple occurrences of variables, semantic and syntactic patterns in code, the relation between different parts of the code, the readability of the entire code, and whether or not the code still compiles after the adversarial mutation. Therefore, applying NLP methods on the 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 using our DAMP approach.

4.1 Definitions

Suppose we are given a trained model of code. The given model can be described as a function $f_\theta : C \rightarrow \mathcal{Y}$, where C is the set of all code snippets and $\mathcal{Y}$ is a set of labels.

Given a code snippet $c \in C$ that the given trained model predicts as $y \in \mathcal{Y}$, i.e., $f_\theta(c) = y$, we denote by $y_{bad} \in \mathcal{Y}$ the *adversarial label* chosen by the attacker. Following the definitions of Bielik and Vechev [2020]: let $\Delta(c)$ be a set of valid modifications of the code snippet c, and let $\delta(c)$ be a new input obtained by applying a modification $\delta : C \rightarrow C$ to c, such that $\delta \in \Delta(c)$. For instance, if c is the code snippet in Figure 3a, then Figure 3b shows $\delta_{\texttt{destination}\rightarrow\texttt{scsqbhj}}(c)$ – the same code snippet where $\delta_{\texttt{destination}\rightarrow\texttt{scsqbhj}}$ is the modification that renames the variable `destination` to `scsqbhj`. As a result, the prediction of the model changes from y=`DestinationType` to y_{bad}=`SourceType`, and $y = f_\theta(c) \neq f_\theta\left(\delta_{\texttt{destination}\rightarrow\texttt{scsqbhj}}(c)\right) = y_{bad}$.

In this paper, we focus on perturbations that rename local variables or add dead code. For brevity, in this section we focus only on generating adversarial examples by renaming of the original variables (VarName). Thus, we define the possible perturbations that we consider as $\Delta(c) = \{\delta_{v\rightarrow v'} \mid \forall c \in C : \delta_{v\rightarrow v'}(c) = c_{v\rightarrow v'}\}$, where $c_{v\rightarrow v'}$ is the code snippet c in which the variable v was renamed to v'.

Our approach can also be applied to a code snippet without changing the existing names, and instead adding a redundant variable declaration (DeadCode insertion, see Section 6.1.2). In such a case, our approach can be similarly applied by choosing an initial random name for the new redundant variable, and selecting this variable as the variable we wish to rename.

We define $Var(c)$ as the set of all local variables existing in c. The adversary's objective is to thus select a single variable $v \in Var(c)$ and an alternative name v', such that renaming v to v' will make the model predict the adversarial label. The most challenging step is to find the right alternative name v' such that $f_\theta(\delta_{v\rightarrow v'}(c)) = y_{bad}$.

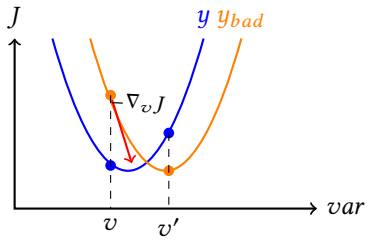

Fig. 5. Gradient is computed for y_{bad} loss function w.r.t. v. By moving toward the opposite direction of the gradient (and replacing v with v'), we decrease the loss of y_{bad}.

Minimality of Perturbation. In addition to semantic equivalence, we also require that the programs "before" and "after" the perturbation be as similar as possible, i.e., $c \approx \delta(c)$. While this constraint is well-defined formally and intuitively in continuous domains such as images and audio [Carlini and Wagner 2018; Szegedy et al. 2013], in discrete domains (e.g., programs) the existing definitions do not hold. The main reason is that every change is perceptible in code. It is possible to compose a series of perturbations $\delta_1, \delta_2, ..., \delta_k$ and apply them to a given code snippet: $\delta_1 \circ \delta_2 ... \circ \delta_k(c)$. In this paper, to satisfy the similarity requirement, we focus on selecting and applying only a single perturbation.

Semantic-preserving Transformations. Using variable renaming as the form of semantic-preserving transformation has the following advantages: (i) each variable appears in several places in the code, so a single renaming can induce multiple perturbations; (ii) the adversarial code can still be compiled and therefore stays in the code domain; and (iii) some variables do not affect the readability of the code and hence, renaming them creates unnoticed unobserved 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.

From a high-level perspective, 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 updating 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 θ 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 in Section 3.1, when training using gradient descent, the following rule is used to update the model parameters and minimize J :

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta_t, c, y) \tag{3}$$

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 variable v:

$$v' = \bar{v} - \eta \cdot \nabla_{\bar{v}} J(\theta, c, y_{bad}) \tag{4}$$

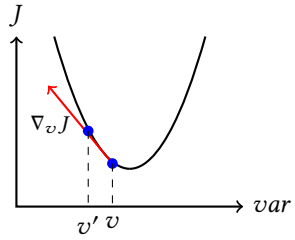

Fig. 6. Gradient ascent illustration: Gradient is computed with respect to v. By moving towards the gradient's direction (and replacing v with v'), we increase the loss.

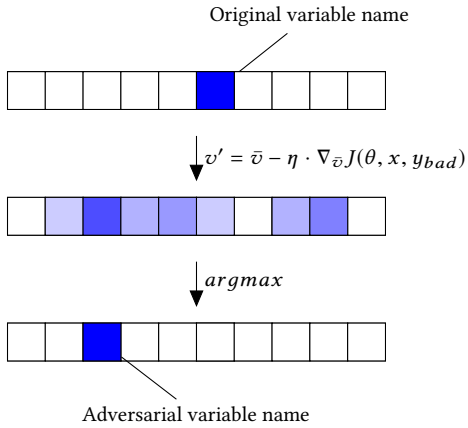

Fig. 7. Perturbing a variable name: the original variable name is represented as a one-hot vector over the variable-name vocabulary. After perturbation, the vector is no longer one-hot. We apply argmax to find the most likely adversarial name, resulting with another one-hot vector over the variable-name vocabulary. This figure is identical to Figure 2 and repeated here for clarity.

where $\bar{v}$ is the one-hot vector representing v. The above action can be viewed intuitively as taking a step toward the steepest descent, where the direction is determined by the loss function that considers y_{bad} as the correct label. This is illustrated in Figure 5

The loss $J(\theta, c, y_{bad})$ in Equation (4) is differentiable with respect to $\bar{v}$ because J is the same loss function that the model was originally trained with, e.g., cross entropy loss. The main difference from the standard gradient descent training (Equation (2)) is that Equation (4) takes the gradient with respect to the one-hot representation of the input variable ($\nabla_{\bar{v}}$), rather than taking the gradient with respect to the learnable weights (∇_θ) in Equation (2).

In fact, the result of the above action does not produce a desired new variable name v'. Instead it produces a distribution over all possible variable names. To concretize the name of v', we choose the argmax over the resulting distribution, as illustrated in Figure 7 and detailed in Section 4.4.

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 in *any* direction, instead of decreasing it, as in the training process. Thus, we compute the gradient with respect to v and use Gradient *Ascent*:

$$v' = \bar{v} + \eta \cdot \nabla_{\bar{v}} J(\theta, x, y) \tag{5}$$

This rule can be illustrated as taking a step toward the steepest *ascent* in the loss function (Figure 6).

Targeted vs. Non-targeted Attacks. Notice the difference between Equation (4) and Equation (5). In Equation (4), the goal of the targeted attack is to find the direction of the *lowest* loss with respect to the *adversarial label*. So, we take a step toward the *negative* gradient.

In Equation (5), our goal for the non-targeted attack is to find the direction of the *higher* loss with respect to the *original* label. This is exactly what the gradient gives us.

These equations differ on two cardinal axes: the target label (original or adversarial), and direction of progress (towards or away from), which are the main axes of the equations.

4.4 Deriving by Integer Indices

The operation of looking-up a row vector in an embedding matrix using its index is simple. Some common guides and tutorials describe this as taking the dot product of the embedding matrix and a *one-hot* vector representing the index. In contrast with these guides, when implementing neural networks, there is usually no real need to use one-hot vectors *at all*. 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 a one-hot vector instead of an index 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 7).

4.5 Search

Sometimes, adversarial examples can be found by applying the *adversarial step* of Equation (4) once. At other times, multiple steps are needed, i.e., replacing v with v' and computing the gradient again with respect to v'. We limit the number of times we apply the *adversarial step* by a *depth* hyperparameter. Additionally, instead of taking the *argmax* from the distribution over candidate names, we can try all "top-k" candidates. These define a Breadth-First Search (BFS), where the *width* parameter is defined by the "top-k". Checking whether a choice of variable name results in the desired output is low cost; it just requires computing the output of the model given the modified input. Thus, it is conveniently feasible to perform multiple adversarial steps and check multiple top-k candidates.

The *depth* and *width* hyperparameters fine tune the trade-off between the effectiveness of the attack and its computation cost. Small values will define a search that is computationally inexpensive but may lead to early termination without any result. Higher values can increase the probability of finding an adversarial example at the cost of a longer search.

4.6 Convergence

In general, gradient descent is guaranteed to converge to a global minimum only when optimizing *convex* functions [Nesterov 2013]. The loss functions of most neural networks are not convex, and thus, our gradient descent process of multiple adversarial steps is *not* guaranteed to succeed. Furthermore, even if we could hypothetically find a global minimum to the loss function $J(\theta, c, y_{bad})$, it is possible that for a given adversarial target y_{bad} there is no perturbation that can force the model to predict y_{bad}. In fact, it is rather intriguing that the success rate for our targeted attack is so high (as show in Section 6) when we restrict the perturbation to a single variable.

As discussed in Section 4.5, we limit the number of steps by a *depth* hyperparameter. Empirically, in most cases this process ends successfully, by finding a new variable name v' that forces the adversarial label y_{bad}. If no adversarial v' was found after *depth* steps, we consider this attack to have failed.

While recent work provides proofs and guarantees for continuous domains [Balunovic et al. 2019; Singh et al. 2019], we are not aware of any work that provides guarantees for discrete domains. Even in continuous domains, guarantees have only been provided for small networks with just 88K learnable parameters [Singh et al. 2019] and 45K parameters [Balunovic et al. 2019]; these networks often restrict the use of non-linearities. In contrast, the models we experiment with are state-of-the-art, realistic, models with several orders of magnitude more parameters.

5 DEFENSE AGAINST ADVERSARIAL EXAMPLES

To defend against adversarial attacks, we consider two broad classes of defense techniques. The first class contains techniques that serve as a "gatekeeper" for incoming input and can be plugged in on top of existing trained models, without having to re-train them. The second class contains techniques that require the model to be re-trained, possibly using a modified loss function or a modified version of the original training set.

5.1 Defense without Re-training

Techniques that do *not* require re-training are appealing because they allow us to separate the optimization of the model from the optimization of the defense, and easily tune the balance between them. Moreover, training neural models is generally computationally expensive; thus, these approaches enable us to perform the expensive training step only once.

Approaches that do not require retraining can be generalized as placing a defensive-model g before the model of code f_θ, which is independent of g. The goal of g is to fix the given input, if necessary, in a way that will allow f_θ to predict correctly. Mathematically, the new model can be defined as being composed of two models: $f_\theta \circ g$. We assume that the adversary has access to the model f_θ being attacked, but not to the defense model g.

We evaluated the following approaches that do not require re-training:

No Vars - a conservative defensive approach that replaces *all* variables to an `UNK` ("unknown") symbol, only at test time. This approach is 100% robust by construction, but does not leverage variable names for prediction.

Outlier Detection – tries to identify an outlier variable name and filter it out by replacing only this variable with `UNK`. The main idea is that the adversarial variable name is likely to have a 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 define $Sym(c)$ as the set of all identifiers and literals existing in c.

We select the variable z^*, which is the most distant from the average of the other symbols:

$$z^* = \underset{z \in Var(c)}{\operatorname{argmax}} \left\| \frac{\sum_{v \in Sym(c), v \neq z} vec\,(v)}{|Sym\,(c)|} - vec\,(z) \right\|_2 \tag{6}$$

We then define a threshold σ 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 σ, then z is replaced with an UNK symbol; otherwise, the code snippet is left untouched. Practically, the threshold σ is tuned on a validation set. It determines the trade-off between the effectiveness of the defense and the accuracy while not under attack, as we evaluate and discuss in Section 6.3.3.

5.2 Defense with Re-training

Ideally, re-training the original model allows us to train the model to be robust while, at the same time, training the model to perform as well as the non-defensive model. Re-training allows the model to be less vulnerable from the beginning, rather than patching a vulnerable model using a separate defense.

We evaluated the following techniques that do require re-training:

Train Without Vars – replaces all variables with an UNK symbol *both at training and test time*. This approach is also 100% robust by construction, as *No Vars* does not require re-training. It is expected to perform better than *No Vars* in terms of F1 because it is trained not to rely on variable names, and to use other signals instead. The downside is that it requires training a model from scratch, while *No Vars* can be applied to an already-trained model.

Adversarial Training – follows Goodfellow et al. [2014b] and trains a model on the original training set, while learning to perform the original predictions *and* training on adversarial examples *at the same time*. Instead of minimizing only the expected loss of the original distribution, every example from the training set $(x, y) \in T$ contributes both to the original loss $J(\theta, x, y)$ *and* to an adversarial loss $J_{adv}(\theta, x', y)$. Here, x' is a perturbed version of x, which was created using a single BFS step of our non-targeted attack (Section 4.3). During training, we minimized $J + J_{adv}$ simultaneously. Note, this method doesn't change the model complexity but it does increase the training time, making it about three times slower.

Adversarial Fine-Tuning – follows Hosseini et al. [2017] and trains a model for several epochs with the original examples from the training set using the original loss $J(\theta, x, y)$. Once the model is trained for optimal performance, it is fine-tuned on adversarial examples. During fine-tuning, the model is trained for a single iteration over the training set, and only on adversarial versions of each training example, using the adversarial loss $J_{adv}(\theta, x', y)$. The expectation in this approach is to establish the model's high performance first, and then ensure the model's robustness to adversarial examples because of the recent fine-tuning.

No Defense, $|\text{vocab}|=\{10\text{k}, 50\text{k}, 100\text{k}\}$ – trains the original model with a smaller vocabulary. Limiting the vocabulary size has the potential to improve robustness by ignoring rare variable names. Names that are observed in the training or test data but are outside the limited vocabulary are replaced with a special UNK symbol. This way, the model is expected to consider only frequent names. Because of their frequency, these names will be observed enough times during training such that their vector representation is more stable.

6 EVALUATION

Our set of experiments comprises two parts: (a) evaluating the ability of DAMP to change the prediction of the downstream classifier for targeted and non-targeted attacks; and (b) evaluating a variety of defense techniques and their ability to mitigate the attacks.

We note that our goal is *not* to perform a robustness evaluation of the attacked models themselves. For more on evaluating the robustness for models of code, we refer the reader to Ramakrishnan et al. [2020] and Bielik and Vechev [2020]. Instead, the goal of this section is to evaluate the effectiveness of our proposed targeted and non-targeted attacks, and to evaluate the robustness that different defense techniques provide to a given model.

Our code, data, and trained models are available at https://github.com/tech-srl/adversarial-examples .

6.1 Setup

6.1.1 Downstream Models. We evaluated our DAMP attack using three popular architectures as downstream models. We obtained the trained models from their original authors, who trained the models themselves.

CODE2VEC was introduced by Alon et al. [2019c] as a model that predicts a label for a code snippet. The main idea is to decompose the code snippet into AST paths, and represent the entire snippet as a set of its paths. Using large scale training, the model was demonstrated to predict a method name conditioned on the method body. The goal of the attack is to thus change the predicted method name by perturbing the method body. This model takes Java methods as its inputs, and represents variable names using a vocabulary of learned embeddings.

Gated Graph Neural Networks (GGNNs) were introduced by Li et al. [2016] as an extension to Graph Neural Networks (GNNs) [Scarselli et al. 2008]. Their aim is to represent the problem as a graph, and to aggregate the incoming messages to each vertex with the current state of the vertex using a GRU [Cho et al. 2014] cell. GGNNs were later adapted to source code tasks by Allamanis et al. [2018], who applied them to the VARMISUSE task of predicting the correct variable in a given blank slot among all variables in a certain scope.

GNN-FILM is a GNN architecture that was recently introduced by Brockschmidt [2019]. It differs from prior GNN models in its message passing functions, which compute the "neural message" based on both the source and target of each graph edge, rather than just the source as done in previous architectures. GNN-FILM was shown to perform well for both the VARMISUSE task as well as other graph-based tasks such as protein-protein interaction and quantum chemistry molecule property prediction. The goal of the attack here and in GGNNs is to make the model predict the incorrect output variable name; this is done by changing an unrelated input variable name, which is neither the correct variable nor the adversarial variable.

Vocabulary. In CODE2VEC, we use the original trained model's own vocabulary to search for a new adversarial variable name. We derive the adversarial loss with respect to the distribution over all variable names. Thus, the chosen adversarial variable name can be any name from the vocabulary that was used to train the original CODE2VEC model. The GNN-FILM and the GGNN models take C# methods as their inputs and represent variable names using a *character-level* convolution. Thus, in our DAMP attack we derive the loss with respect to the distribution over all *characters* in a given name; the chosen adversarial input name can be any combination of characters.

6.1.2 Adversarial Strategies. While there are 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 until the model's prediction is changed, using a BFS (as explained in Section 4). Eventually, the "distance" between the original code and the adversarial code is *a single variable name at most*.
- **Dead-Code Insertion (DeadCode)**: insert a new unused variable declaration and derive 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 removing unused variables before feeding the code into the trained model. Nonetheless, in the general case, detecting an unreachable code is undecidable. In all cases, we arbitrarily placed the dead code at the end of the input method, and used our attack to find a new name for the new (unused) declared variable. In our preliminary experiments we observed that placing the dead code *anywhere* else works similarly. In this attack strategy, *a single variable declaration* is inserted. Thus, the "distance" between the original code and the adversarial code is *a single variable declaration statement*.

Other semantic-preserving transformations such as statement swapping and operator swapping, are *not differentiable* and thus do not enable *targeted attacks*.

When renaming variables or introducing new variables, we verified that the new variable name does not collide with an existing variable that has the same name. In the code2vec experiments, we used the *adversarial step* to run a BFS with $width = 2$ and $depth = 2$. In GGNN we used $width = 1$ and $depth = 3$, and GNN-FiLM required $depth = 10$ to achieve attack success that is close to that of the other models. We discuss these differences in Section 6.2.3. Increasing the $width$ and $depth$ can definitely improve the adversary's success but at the cost of a longer search, although the entire BFS takes a few seconds at most.

6.1.3 Dataset. **code2vec - Java** We evaluated our proposed attack and defense on code2vec using the Java-large dataset [Alon et al. 2019a]. This dataset contains more than 16M Java methods and their labels, taken from 9500 top-starred Java projects in GitHub that have been created since January 2007. It contains 9000 projects for training, 200 distinct projects for validation, and 300 distinct projects for test. We filtered out methods with no local variables or arguments, since they cannot be perturbed by variable renaming. To evaluate the effectiveness of our DAMP attack, we focus on the examples that the model predicted correctly out of the test set of Java-large.

That is, from the original test set, we used a subset that the original code2vec predicted accurately. On this filtered test set, the accuracy of the original code2vec model is 100% by construction.

GGNN and GNN-FiLM - C# We evaluated DAMP with the GGNN and GNN-FiLM C# models on the dataset of Allamanis et al. [2018]. This dataset consists of about 220,000 graphs (examples) from 29 top-starred C# projects on GitHub. For all examples, there is at least one type-correct replacement variable other than the correct variable, and a maximum of up to 5 candidates. DAMP always attacks by modifying an unrelated variable, not the correct variable or the adversarial target. In targeted attacks, we randomly pick one out of the five candidates as the target for attack. In non-targeted attacks, our attacker tries to make the model predict any incorrect candidate. Similar to the Java dataset, we focus on the examples that each original model predicts correctly out of the test set.

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*. For the desired adversarial labels, we randomly sampled labels that occurred at least 10K times in the training set.

6.2.1 Metrics. We measured the robustness of each setting for 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 not changed to any label other than the correct label.

6.2.2 Baselines. Since our task is new, we are not aware of existing baselines. We thus compare DAMP to different approaches in targeted and non-targeted attacks for code2vec and GNN models.

TFIDF is a statistical baseline for attacking code2vec: for every pair of a label and a variable name it computes the number of times the variable appears in the training set under this label, divided by the total number of occurrences of the variable in the training set. Then, given a desired adversarial label y_{bad}, TFIDF outputs the variable name v that has the highest score with y_{bad}: $\text{TFIDF}(y_{bad}) = argmax_v \frac{\#(y_{bad}, v)}{\#(v)}$.

CopyTarget attacks code2vec with *targeted attacks*. *CopyTarget* replaces a variable with the desired adversarial label. For example, if the adversarial label is `badPrediction`, *CopyTarget* renames a variable to `badPrediction`.

CharBruteForce attacks GGNN and GNN-FiLM, which address the VarMisuse task. *CharBruteForce* changes the name of the attacked variable by randomly changing every character iteratively up to a limited number of iterations.

RandomVar is used in *non-targeted attacks* on code2vec: *RandomVar* replaces the given variable with a randomly selected variable name from the training set vocabulary.

In all experiments, the baselines were given the same number of trials as our attack. In *RandomVar*, we randomly sampled the same number of times. In *TFIDF*, we used the top-k TFIDF candidates as additional trials. In *CopyTarget*, we took the target as the new variable name and its k-nearest neighbors in the embedding space.

6.2.3 Attack - Results. **code2vec** Table 1 summarizes the results of DAMP on code2vec. The main result is that DAMP outperforms the baselines by a large margin in both targeted and non-targeted attacks.

In non-targeted attacks, DAMP attacks are much more effective than the baseline: for VarName, code2vec is 6% robust to DAMP, 34.10% robust to *RandomVar* and 53.53% robust to *TFIDF*. For DeadCode, code2vec is 21.83% robust to DAMP, 54.90% robust to *RandomVar*, and 84.00% robust to *TFIDF*. Thus, DAMP's attack is more effective than that of the baselines.

In targeted attacks, DAMP performs better than *CopyTarget* for each of the randomly sampled adversarial labels. For example, code2vec is only 10.39% robust to DAMP attacks that change the prediction to the label `mergeFrom`, and 72.79% robust to the *CopyTarget* attack of the same target adversarial label.

However, in some of the randomly sampled targets *TFIDF* was more effective than DAMP. In VarName, DAMP was the most effective in 6 out of the 10 randomly sampled adversarial targets, while *TFIDF* was the most effective in the remaining 4. In DeadCode attacks, DAMP was the most effective in 8 out of 10 randomly sampled adversarial targets, while *TFIDF* was the most effective in the remaining 2. Although *TFIDF* performed better than DAMP for a few of the targets, the

Table 1. Robustness of code2vec to our adversarial attacks, targeted and non-targeted, using VarName and DeadCode, compared to the baselines (the lower robustness, the more effective the attack). DAMP is more effective than the baselines in 6 out of 10 VarName randomly sampled targets and in 8 out of 10 DeadCode targets.

	VarName (robustness %)			DeadCode (robustness %)		
	RandomVar	TFIDF	**DAMP**	*RandomVar*	TFIDF	**DAMP**
Non-targeted	34.10	53.53	**6.00**	54.90	84.00	**21.83**
Targeted:	*CopyTarget*			*CopyTarget*		
init	84.47	74.33	**48.44**	96.79	96.67	**87.62**
mergeFrom	72.79	**9.01**	10.39	99.82	29.34	**22.65**
size	99.47	**77.91**	78.27	99.97	98.98	**95.96**
isEmpty	88.61	98.11	**79.04**	99.98	99.72	**87.63**
clear	89.56	89.00	**82.80**	99.07	98.55	**97.89**
remove	84.94	84.83	**63.15**	99.29	99.24	**80.02**
value	99.77	**71.81**	76.75	100.0	**98.16**	98.33
load	86.75	85.46	**55.65**	99.03	97.22	**86.65**
add	92.88	86.72	**68.60**	99.93	95.29	**93.75**
run	95.11	**40.92**	51.52	99.36	**62.87**	77.63
Count best:	0	4	**6**	0	2	**8**

Table 2. Robustness of GGNN and GNN-FiLM to our adversarial attacks (targeted and non-targeted), compared to *CharBruteForce* (the lower the robustness, the more efficient the attack). Our attack is more effective than brute-force given an equal number of trials. Every successful attack results in a type-safe VarMisuse bug.

	GGNN Robustness		GNN-FiLM Robustness	
	CharBruteForce	DAMP (this work)	*CharBruteForce*	DAMP (this work)
Non-targeted	96.24	**57.99**	95.56	**83.55**
targeted	97.84	**69.00**	96.19	**87.62**

differences were usually small. Overall, DAMP's targeted attack is more effective than that of the baselines.

Non-targeted attacks generally yield lower model robustness than targeted attacks. This is expected, since non-targeted attacks try to change the label to *any* label other than the correct one, while targeted attacks count as successful only if they change the prediction to the desired label.

In general, the VarName attack is more effective than DeadCode. We hypothesize that this is because the inserted unused variable impacts only a small part of the code. Hence, it may have a smaller numerical effect on the computation of the model. In contrast, renaming an existing variable changes multiple occurrences in the code and thus has a wider effect.

GGNN and GNN-FiLM Table 2 summarizes the results of the DAMP attack and the baseline on the GGNN and GNN-FiLM models. The main result is that DAMP is much more effective than the *CharBruteForce* baseline. The GGNN model is 69.00% robust to the DAMP targeted attack, and 98.84% robust to the baseline attack. The GNN-FiLM is 87.62% robust to the DAMP attack and 96.19% robust to the *CharBruteForce* attack.

Table 3. Precision, recall, F1, and robustness percentage of different models. The higher the robustness, the more effective the defense. Scores that are above 95% are marked in **bold**. Precision, recall, and F1 are measured on the subset of the test set that the vanilla model predicts accurately (as explained in Section 6.1.3). *Outlier Detection* and *Adversarial Training* are sweet-spots: they perform almost as good as *No Defense* in terms of precision, recall, and F1, and they are almost as robust as the extreme *No Vars*.

	Performance (not under attack)			VarName Robustness (%)		DeadCode Robustness (%)	
	Prec	Rec	F1	Non-targeted	Target: "run"	Non-targeted	Target: "run"
No Vars	78.78	80.83	79.98	**100**	**100**	**100**	**100**
Outlier Detection	**98.18**	**97.75**	**97.92**	74.35	**96.49**	**96.73**	**95.76**
Train Without Vars	89.74	90.86	90.40	**100**	**100**	**100**	**100**
Adversarial Training	**98.09**	92.96	94.98	68.80	94.48	**99.33**	**99.98**
Adversarial Fine-Tuning	85.51	84.86	85.12	13.59	68.94	31.07	56.64
No Defense	**100**	**100**	**100**	6.0	51.52	22.83	77.63
$\|vocab\|$=10k	90.50	94.26	92.66	6.56	45.78	31.13	94.55
$\|vocab\|$=50k	94.51	**100**	**98.00**	3.18	40.48	20.01	94.04
$\|vocab\|$=100k	**96.84**	**99.97**	**98.99**	2.61	32.48	19.64	68.58

In addition, we see that DAMP is more effective on code2vec than on both GNN architectures. code2vec is 6% robust to non-targeted VarName attacks and GGNNs is 57.99% robust to non-targeted attacks. There are several reasons for this difference between the attacked models:

(1) code2vec is simpler and more "neurally shallow", while GGNN uses 6 layers of message propagation steps, and GNN-FiLM uses 10 layers.
(2) The models' tasks are very different: code2vec classifies a given code snippet to 1 out of about 200,000 possible target method names, while both GNN architectures address the VarMisuse task and need to choose 1 out of only 2 to 5 possible variables.
(3) code2vec has orders of magnitude more trainable parameters (about 350M) than GGNN (about 1.6M) and GGNN (about 11.5M), making code2vec more sparse, its loss hyperspace more complex, and thus more vulnerable.

Finally, we see that the GNN-FiLM model is more robust than GGNN. Even though we used a more aggressive BFS with 10 gradient steps for GNN-FiLM and only 3 gradient steps for GGNN, the GNN-FiLM robustness to DAMP targeted attack is 87.62%, while the GGNN robustness is 69.00%. We hypothesize that this is primarily because the message passing function in GNN-FiLM computes the sent message between nodes based on both the source *and target* of each graph edge, rather than just the source as in the GGNN. This makes it more robust to an attack on a single one of these nodes. The higher results of GNN-FiLM over GGNN in other graph-based benchmarks [Brockschmidt 2019] hint that GNN-FiLM may be using the graph topology, i.e., the program structure, to a larger degree; the GGNN focuses mainly on the names and is thus more vulnerable to name-based attacks. We leave further investigation of the differences between different GNN architectures for future work.

6.3 Defense

We experimented with all defense techniques as described in Section 5, in a code2vec model. *No Defense* is the vanilla, unmodified, trained model that was trained by its original authors. *No Defense,* $|vocab|=\{10k, 50k, 100k\}$ are vanilla code2vec models that we trained using the authors' code and the default settings, without any special defense or modification, except for limiting the

vocabulary sizes (where vocabulary size is mentioned); this allowed us to examine whether the limited vocabulary size can serve as some sort of defense.

6.3.1 Metrics. We measured the success rate of the different defense approaches in preventing the adversarial attack and increasing robustness. When evaluating alternative defenses, it is also important to measure the performance of the original model while using the defense, *but not under attack*. An overly defensive approach can lead to 100% robustness at the cost of reduced prediction performance.

To tune the threshold σ of the *Outlier Detection* defense, we balanced the following factors: (1) the robustness of the model using this defense; and (2) the F1 score of the model using this defense, while not under attack. We tuned the threshold on the validation set, and chose $\sigma = 2.7$ since it led to 75% robustness against non-targeted attack at 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 6.3.3).

6.3.2 Defense - Results. The effectiveness of the different defense techniques is presented in Table 3. The main results are as follows: *Outlier Detection* provides the best performance and highest robustness among the techniques that do not require re-training; it achieves an F1 of 97.02 and above 75.35% robustness for targeted and non-targeted attacks. Among the techniques that do require re-training, *Adversarial Training* achieves the highest performance and robustness; *Adversarial Training* achieves an F1 score of 94.98 and above 68.80% robustness for targeted and non-targeted attacks.

The penalty of using *Outlier Detection* is only 2.18% degradation in F1 score compared to *No Defense*. At 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, *Outlier Detection* is a sweet spot in the trade-off between performance and robustness: it is 98% as accurate as the original model (*No Defense*) and $74 - 96\%$ as robust as the model that ignores variables completely.

Adversarial Training provides a sweet-spot among the techniques that do require re-training: its penalty is 5.02% degradation in F1 score compared to *No Defense*, and it achieves over 99% robustness to DeadCode and 94.48% robustness to targeted VarName. Adversarial training allows the model *to leverage* variable names to achieve high performance while not under attack, but also to not overly rely on variable names; it is thus robust to adversarial examples. *Adversarial Fine-Tuning* performs surprisingly worse than *Adversarial Training*, both in F1 score and in its robustness.

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.

Reducing the vocabulary size to 100K and 50K, results in a negligible decrease in performance while not under attack, compared to not defending at all. However, it also results in roughly the same poor robustness as no defense at all. Reducing the vocabulary size to 10K hurts performance while not under attack and does not provide much robustness.

These results are visualized in Figure 8.

Table 4 shows the performance of the *Outlier Detection* and the *Adversarial Training* defenses across randomly sampled adversarial labels.

6.3.3 Robustness - Performance Trade-off. One of the advantages of the *Outlier Detection* defense, which we found to be one of the most effective defense techniques, is the ability to tune its trade-off between robustness and performance. Figure 9 shows the trade-off with respect to the similarity threshold σ. Small values of σ make the model with *Outlier Detection* defense more robust. It

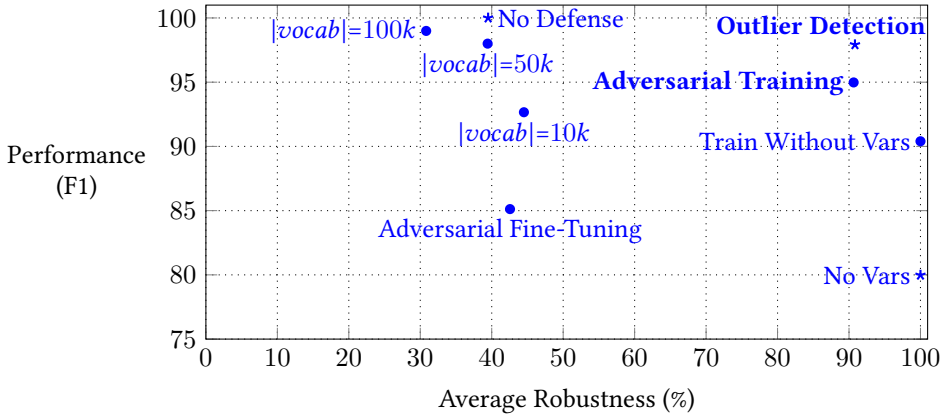

Fig. 8. The performance of each defense technique compared to its average robustness. Average robustness is calculated as the average of the four robustness scores in Table 3. Points marked with a bullet (•) require re-training; points marked with a star (⋆) do not require re-training. *Outlier Detection* provides the highest performance and robustness among the techniques that do *not* require re-training; *Adversarial Training* provides the highest performance and robustness among the techniques that *do* require re-training.

Table 4. Robustness of CODE2VEC to adversarial attacks with the *Outlier Detection* and the *Adversarial Training* defenses, across different adversarial targets (the higher the robustness — the more effective the defense). DAMP† results are the same as in Table 1.

	VARNAME (robustness %)			DEADCODE (robustness %)		
	DAMP†	DAMP + *Outlier Detection*	DAMP + *Adversarial Training*	DAMP†	DAMP + *Outlier Detection*	DAMP + *Adversarial Training*
init	48.44	74.32	**78.17**	87.62	91.41	**99.97**
mergeFrom	10.39	**99.98**	98.91	22.65	**99.99**	**100.00**
size	78.27	**99.58**	99.39	95.96	99.94	**99.99**
isEmpty	79.04	**99.22**	98.91	87.63	97.03	**99.99**
clear	82.8	**98.77**	85.09	97.89	99.56	**99.99**
remove	63.15	**94.50**	89.29	80.02	99.33	**99.99**
value	76.75	90.87	**99.47**	98.33	99.72	**100.00**
load	55.65	60.27	**88.29**	86.65	85.28	100.00
add	68.6	88.97	**95.90**	93.75	97.69	**100.00**
run	51.52	**96.49**	94.48	77.63	95.76	**99.98**

becomes almost as robust as *No Vars*, but perform worse in terms of F1 score. As the value of σ increases, the model with *Outlier Detection* defense becomes less robust, but performs better while not under attack.

6.4 Additional Examples

All examples that are shown in this paper and in appendix A can be experimented with on their original models at http://code2vec.org and https://github.com/microsoft/tf-gnn-samples.

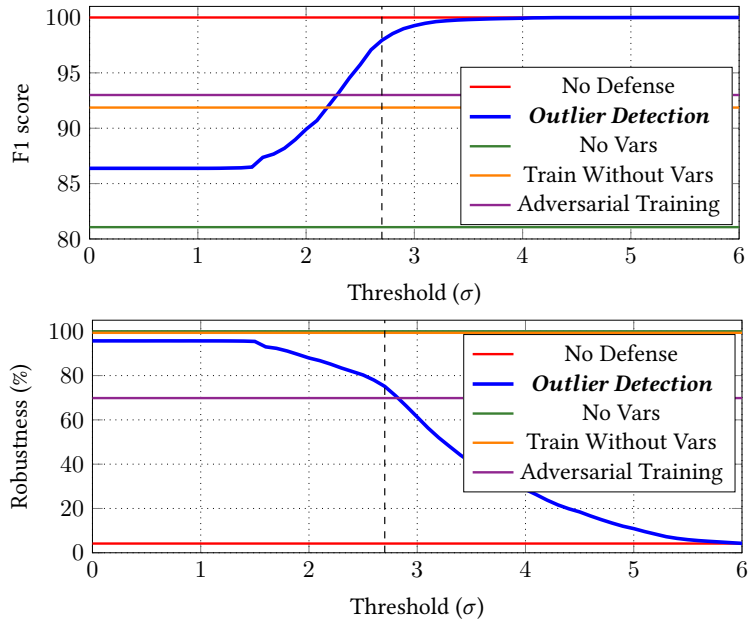

Fig. 9. The *Outlier Detection* defense: the trade-off between robustness and performance-while-not-under-attack, with respect to the similarity threshold σ on the validation set. The dashed vertical line at $\sigma = 2.7$ denotes the value that we chose for σ according to the validation set. A lower threshold leads to perfect robustness and lower performance; a higher threshold leads to performance that is equal to the original model's, but the model is also as vulnerable as the original model. The robustness score is the robustness against non-targeted VarName attacks.

Figure 10 shows additional targeted attacks against the "sort" example from Figure 1. Renaming the variable `array` to `mstyleids` changes the prediction to `get` with a probability of 99.99%; renaming `array` to `possiblematches` changes the prediction to `indexOf` with a probability of 86.99%. The predicted adversarial labels (`get` and `indexOf`) were chosen arbitrarily before finding the variable name replacements.

Transferability Occasionally, a dead code attack is *transferable* across examples, and has the same effect even in different examples. This is demonstrated in Figure 11, where adding the unused variable declaration `int introsorter = 0;` to each of the multiple snippets changes their prediction to `sort` with probability of 100%. This effect is reminiscent of the Adversarial Patch [Brown et al. 2017], that was shown to force an image classifier to predict a specific label, regardless of the input example. However, except for a few cases, we found that adversarial examples *generally do not transfer* across examples. We also did not find significant evidence that adversarial examples transfer across models that were trained on the same dataset, e.g., from GNN-FiLM to GGNN. The question of whether adversarial examples are transferable in discrete domains such as code remains open.

GNN Examples Figure 12 shows A C# VarMisuse example where the missing variable is classified correctly as `_getterBuilder` in the method `GetGetter` by the GGNN model. Given the code and the target `_setterBuilder`, our approach renames a local variable `setteril` in *another method*. This makes the model predict the wrong variable in the `GetGetter` method, thus introducing a real bug in the method `GetGetter`. Figure 13 shows a similar GNN targeted adversarial example.

```
void f(int[] array){
  boolean swapped = true;
  for (int i = 0; i < array.length && swapped; i++){
    swapped = false;
    for (int j = 0; j < array.length-1-i; j++) {
      if (array[j] > array[j+1]) {
        int temp = array[j];
        array[j] = array[j+1];
        array[j+1]= temp;
        swapped = true;
      }
    }
  }
}
```

Prediction: **sort** (98.54%)

```
void f(int[] mstyleids){
 boolean swapped = true;
 for (int i = 0;
      i < mstyleids.length && swapped; i++){
  swapped = false;
  for (int j = 0; j < mstyleids.length-1-i; j++){
   if (mstyleids[j] > mstyleids[j+1]) {
    int temp = mstyleids[j];
    mstyleids[j] = mstyleids[j+1];
    mstyleids[j+1]= temp;
    swapped = true;
   }
  }
 }
}
```

Prediction: **get** (99.99%)

```
void f(int[] possiblematches){
 boolean swapped = true;
 for (int i = 0;
      i < possiblematches.length && swapped; i++){
  swapped = false;
  for (int j = 0; j < possiblematches.length-1-i; j++){
   if (possiblematches[j] > possiblematches[j+1]){
    int temp = possiblematches[j];
    possiblematches[j] = possiblematches[j+1];
    possiblematches[j+1]= temp;
    swapped = true;
   }
  }
 }
}
```

Prediction: **indexOf** (86.99%)

Fig. 10. 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`.

Other code2vec and GNNs examples are shown in Appendix A.

7 RELATED WORK

Adversarial Examples of Images Szegedy et al. [2013] found that deep neural networks are vulnerable to adversarial examples. They showed that they could cause an image classification model to misclassify an image by applying a certain barely perceptible perturbation. Goodfellow et al. [2014b] introduced an efficient method called "fast gradient sign method" to generate 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. Generating adversarial examples in images is probably easier than in discrete domains such as code and natural language. Images are continuous objects, and thus can be perturbed with small undetected noise; in contrast, our problem domain is *discrete*.

Adversarial Examples in NLP The challenge of adversarial examples for *discrete inputs* has been studied in the domain of NLP. While adversarial examples on images are easy to generate, the generation of adversarial text is more difficult. Images are continuous and can thus have some noise added; natural language text is discrete and thus cannot be easily perturbed. HotFlip [Ebrahimi et al. 2017] is a technique for generating adversarial examples that attack a character-level neural classifier. The idea is to flip a single character in a word, producing a typo such that the change is hardly noticeable. Alzantot et al. [2018b] presented a technique for generating "semantically and syntactically similar adversarial examples" that attack trained models. The main idea is to replace a random word in a given sentence with a similar word (nearest neighbor) in some embedding

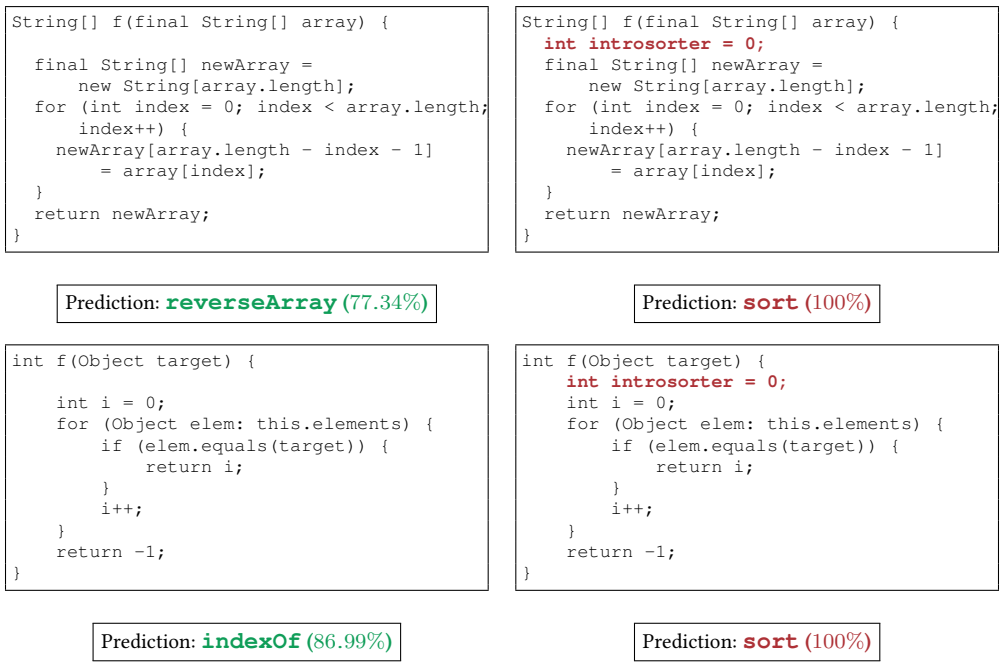

Fig. 11. Adding the dead code `int introsorter = 0;` to each of the snippets on the left changes their label to `sort` with confidence of 100%. This is an example of how the same dead adversarial "patch" can be applied across different examples.

space. However, these techniques allow only *non-targeted* attacks, i.e., impair a correct prediction, without aiming for a specific adversarial label.

Adversarial Examples in Malware A few works explored adversarial examples in malware detection that have the ability to perturb a malicious binary such that a learning model will classify it as "benign". Kreuk et al. [2018], Suciu et al. [2019] and Kolosnjaji et al. [2018] addressed binary classifiers (whether or not the program is malicious) of binary code, by adding noisy bytes to the original file's raw bytes. None of these works performed *targeted* attacks as does our work. In most cases, hiding an adversarial dead-code payload inside a binary may be easier than generating adversarial examples in high-level languages like Java or C#. For example, Kolosnjaji et al. [2018] reported that they injected at least 10,000 "padding bytes" to each malware sample; because a binary file is usually much larger, injecting 10,000 bytes can go unnoticed. In contrast, in all our attacks we renamed a *single* variable or added a *single* variable declaration. Another difference from our approach is that all these works derive the loss based on the embedding vector itself, as we discuss in Section 2.2. In contrast, we derive the loss by the distribution over indices (Section 4.4), which directly optimizes towards the target label; this allows us to perform targeted attacks in multi-class models. Rosenberg et al. [2018] addressed a very different scenario and modified malicious programs to mimic benign calls to APIs *at runtime*. Yang et al. [2017] presented a black-box approach to attack non-neural models. Their approach is not gradient-based and thus cannot perform targeted attacks.

Adversarial Examples in High-Level Programs To the best of our knowledge, our work is the first to investigate adversarial attacks for models of high-level code. Rabin et al. [2019] identified

Correctly predicted example

```
public class PropertyEmitter
{
  ...

  public PropertyEmitter( ... )
    {
    ...

    ILGenerator setteril =
      _setterBuilder.GetILGenerator();
    setteril.Emit(OpCodes.Ldarg_0);
    setteril.Emit(OpCodes.Ldarg_1);
    setteril.Emit(OpCodes.Stfld, _fieldBuilder);
    if (propertyChangedField != null)
    {
      setteril.Emit(OpCodes.Ldarg_0);
      setteril.Emit(OpCodes.Dup);
      setteril.Emit(
        OpCodes.Ldfld,
        propertyChangedField);
      setteril.Emit(OpCodes.Ldstr, name);
      setteril.Emit(
        OpCodes.Call,
        ProxyBaseNotifyPropertyChanged);
    }
    setteril.Emit(OpCodes.Ret);
    _propertyBuilder.SetSetMethod(_setterBuilder);
  }

  ...

  public MethodBuilder GetGetter(Type requiredType)
    => !requiredType.IsAssignableFrom(PropertyType)
    ? throw new InvalidOperationException(
        "Types are not compatible")
    : _getterBuilder;
}
```

(a)

Adversarial perturbation

Target: `_setterBuilder`

```
public class PropertyEmitter
{
  ...

  public PropertyEmitter( ... )
    {
    ...

    ILGenerator va =
      _setterBuilder.GetILGenerator();
    va.Emit(OpCodes.Ldarg_0);
    va.Emit(OpCodes.Ldarg_1);
    va.Emit(OpCodes.Stfld, _fieldBuilder);
    if (propertyChangedField != null)
    {
      va.Emit(OpCodes.Ldarg_0);
      va.Emit(OpCodes.Dup);
      va.Emit(
        OpCodes.Ldfld,
        propertyChangedField);
      va.Emit(OpCodes.Ldstr, name);
      va.Emit(
        OpCodes.Call,
        ProxyBaseNotifyPropertyChanged);
    }
    va.Emit(OpCodes.Ret);
    _propertyBuilder.SetSetMethod(_setterBuilder);
  }

  ...

  public MethodBuilder GetGetter(Type requiredType)
    => !requiredType.IsAssignableFrom(PropertyType)
    ? throw new InvalidOperationException(
        "Types are not compatible")
    : _setterBuilder;
}
```

(b)

Fig. 12. A C# VarMisuse example which is classified correctly as `_getterBuilder` in the method `GetGetter` by the GGNN model. Given the code and the target `_setterBuilder`, our approach renames a local variable `setteril` in *another method*. This makes the model predict the wrong variable in the `GetGetter` method, thus introducing a real bug in the method `GetGetter`.

Correctly predicted example

```
public MapRequest(TypePair requestedTypes,
    TypePair runtimeTypes,
    IMemberMap memberMap = null)
{
  RequestedTypes = requestedTypes;
  RuntimeTypes = runtimeTypes;
  MemberMap = memberMap;
}
```

(a)

Adversarial perturbation

Target: `RequestedTypes`

```
public MapRequest(TypePair chzuzb,
    TypePair runtimeTypes,
    IMemberMap memberMap = null)
{
  RequestedTypes = chzuzb;
  RequestedTypes = runtimeTypes;
  MemberMap = memberMap;
}
```

(b)

Fig. 13. A C# VarMisuse example that is classified correctly by the GGNN model. Given the code and the target `RequestedTypes`, our approach renames a local variable `requestedTypes`. This makes the model predict the wrong variable, thus introducing a real bug.

a problem of robustness in models of code, but did not suggest a concrete method for producing adversarial examples or a method for defending against them.

Defending Against Adversarial Examples A few concurrent work with ours addressed the problem of training models of code to be more robust. Ramakrishnan et al. [2020] performed semantic-preserving transformations such as renaming, and trained neural models on the modified code. However, their transformations are applied in the data preprocessing step, and their approach does not use gradients or targeted attacks. Bielik and Vechev [2020] focused on training robust models as well, with an iterative approach for adversarial training. These works only discussed approaches for adversarial training, but none presented *targeted attacks*, as our work.

Pruthi et al. [2019] found that 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 repairs misspellings before they are fed into the downstream model. The *Outlier Detection* defense that we examined (Section 5.1) is similar in spirit, since it uses a composition of an upstream defense model followed by a downstream model. The main difference between these two defenses resides in the goals of the upstream defense models: the *Outlier Detection* model detects outlier names, while the model of Pruthi et al. [2019] detects character-level typos.

8 CONCLUSION

We presented DAMP, the first approach for generating *targeted* attacks on 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.* We demonstrated DAMP on popular neural architectures for code: CODE2VEC, GGNN, and GNN-FILM, in Java and C#. Moreover, we showed that DAMP succeeds in both targeted and non-targeted attacks, by renaming variables and by adding dead code.

We further experimented with a variety of possible defense techniques and discuss their trade-offs across performance, robustness, and whether or not they require re-training.

We believe the principles presented in this paper can be the basis for a wide range of adversarial attacks and defenses. This is the first work to perform targeted attacks for models of code; thus, using our attack in adversarial training contributes to more robust models. In realistic production environments, the defense techniques that we examine can further strengthen robustness. To this end, we make all our code, data, and trained models publicly available at https://github.com/tech-srl/adversarial-examples .

ACKNOWLEDGEMENTS

We would like to thank Marc Brockschmidt and Miltiadis Allamanis for their guidance in using their GNN models and code and useful discussions about the differences in robustness between GNN types.

The research leading to these results has received funding from the Israel Ministry of Science and Technology, grant no. 3-9779.

A APPENDIX - ADDITIONAL EXAMPLES

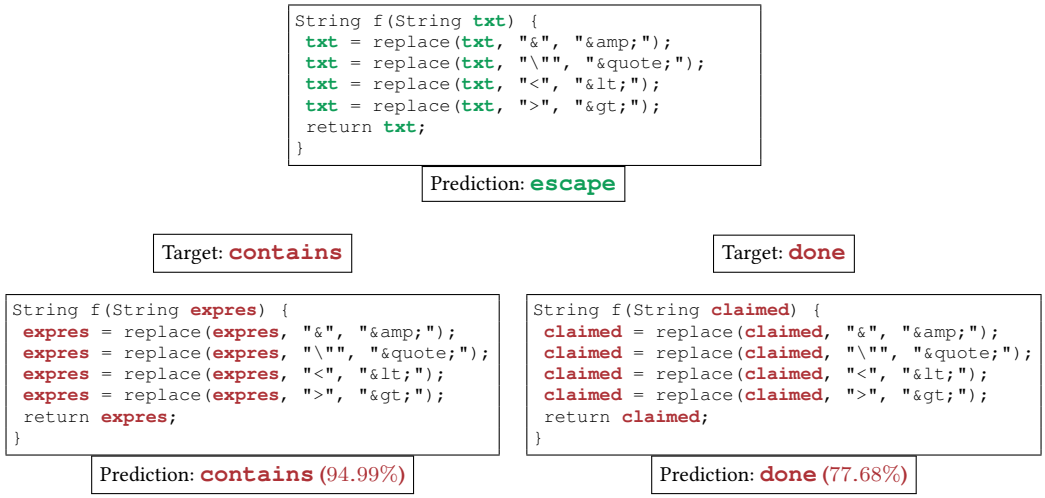

Fig. 14. A snippet classified correctly as `escape` by the model of code2vec.org. The same example is classified as `contains` by renaming `txt` to `expres` and is classified as `done` by renaming `txt` to `claimed`. These targets (`contains` and `done`) were chosen arbitrarily in advance, and our DAMP method had found the new required variable names `expres` and `claimed`.

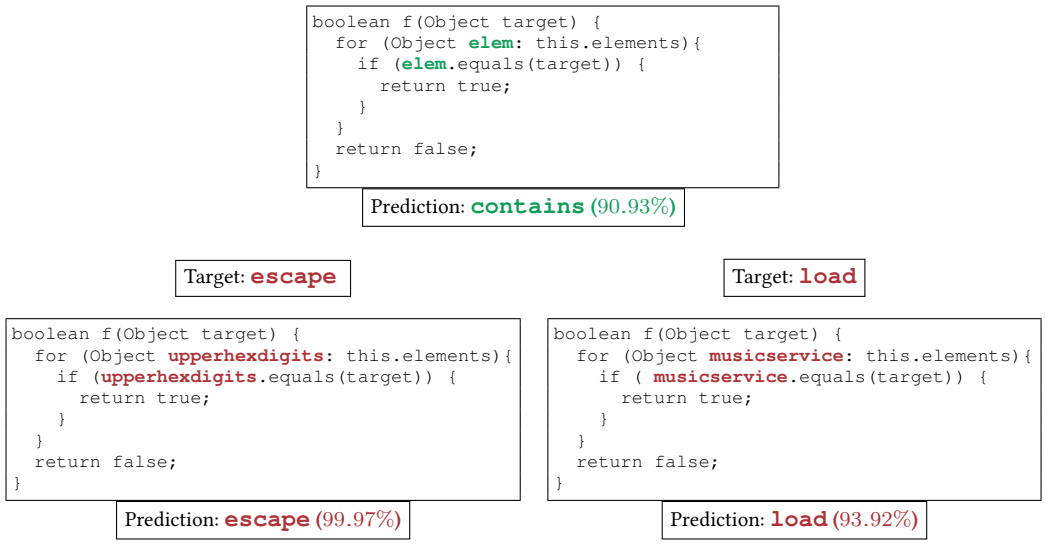

Fig. 15. A snippet classified correctly as `contains` by the model of code2vec.org. The same example is classified as `escape` by renaming `elem` to `upperhexdigits` and is classified as `load` by renaming `elem` to `musicservice`. These targets (`escape` and `load`) were chosen arbitrarily in advance, and our DAMP method had found the new required variable names `upperhexdigits` and `musicservice`.

Correctly predicted example

```
public static class ProxyGenerator
{
    ...

  private static ModuleBuilder CreateProxyModule()
  {
    AssemblyName name = new AssemblyName(
      "AutoMapper.Proxies");
    name.SetPublicKey(privateKey);
    name.SetPublicKeyToken(privateKeyToken);

    AssemblyBuilder builder =
      AssemblyBuilder.DefineDynamicAssembly(
        name, AssemblyBuilderAccess.Run);

    return builder.DefineDynamicModule(
      "AutoMapper.Proxies.emit");
  }

  private static Type EmitProxy(
    TypeDescription typeDescription)
  {
    var interfaceType = typeDescription.Type;

    ...

    if(typeof(INotifyPropertyChanged)
        .IsAssignableFrom(interfaceType))
    {
      ...
    }

    ...

    foreach(var property in propertiesToImplement)
    {
      if(fieldBuilders.TryGetValue(property.Name,
        out var propertyEmitter))
      {
        if((propertyEmitter.PropertyType
          != property.Type) && (
            (property.CanWrite)
            || (!property.Type.IsAssignableFrom(
                  propertyEmitter.PropertyType))))
        {
          throw new ArgumentException(
            $"The interface has a conflicting
              property {property.Name}",

            nameof( interfaceType ));
        }
      }
      else
      {
        fieldBuilders.Add(property.Name,
          new PropertyEmitter(
            typeBuilder,
            property, propertyChangedField));
      }
    }
    return typeBuilder.CreateType();
  }
}
```

(a)

Adversarial perturbation

Target: **typeBuilder**

```
public static class ProxyGenerator
{
    ...

  private static ModuleBuilder CreateProxyModule()
  {
    AssemblyName name = new AssemblyName(
      "AutoMapper.Proxies");
    name.SetPublicKey(privateKey);
    name.SetPublicKeyToken(privateKeyToken);

    AssemblyBuilder bzodhi =
      AssemblyBuilder.DefineDynamicAssembly(
        name, AssemblyBuilderAccess.Run);

    return bzodhi.DefineDynamicModule(
      "AutoMapper.Proxies.emit");
  }

  private static Type EmitProxy(
    TypeDescription typeDescription)
  {
    var interfaceType = typeDescription.Type;

    ...

    if(typeof(INotifyPropertyChanged)
        .IsAssignableFrom(interfaceType))
    {
      ...
    }

    ...

    foreach(var property in propertiesToImplement)
    {
      if(fieldBuilders.TryGetValue(property.Name,
        out var propertyEmitter))
      {
        if((propertyEmitter.PropertyType
          != property.Type) && (
            (property.CanWrite)
            || (!property.Type.IsAssignableFrom(
                  propertyEmitter.PropertyType))))
        {
          throw new ArgumentException(
            $"The interface has a conflicting
              property {property.Name}",

            nameof( typeBuilder ));
        }
      }
      else
      {
        fieldBuilders.Add(property.Name,
          new PropertyEmitter(
            typeBuilder,
            property, propertyChangedField));
      }
    }
    return typeBuilder.CreateType();
  }
}
```

(b)

Fig. 16. A C# VarMisuse example which is classified correctly as `interfaceType` in the method `EmitProxy` by the GGNN model. Given the code and the target `typeBuilder`, our approach renames a local variable `builder` in *another method*, making the model predict the wrong variable in the `EmitProxy` method, thus introducing a real bug in the method `EmitProxy`.

REFERENCES

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-

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

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1gKYo09tX

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2019b. Structural Language of Code. *arXiv preprint arXiv:1910.00577* (2019).

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

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

Moustafa Alzantot, Bharathan Balaji, and Mani Srivastava. 2018a. Did you hear that? adversarial examples against automatic speech recognition. *arXiv preprint arXiv:1801.00554* (2018).

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

Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. (2014).

Mislav Balunovic, Maximilian Baader, Gagandeep Singh, Timon Gehr, and Martin Vechev. 2019. Certifying geometric robustness of neural networks. In *Advances in Neural Information Processing Systems*. 15313–15323.

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).

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

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

Pavol Bielik and Martin T. Vechev. 2020. Adversarial Robustness for Code. *ArXiv* abs/2002.04694 (2020).

Marc Brockschmidt. 2019. GNN-FiLM: Graph neural networks with feature-wise linear modulation. *arXiv preprint arXiv:1906.12192* (2019).

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

Tom B Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. 2017. Adversarial patch. *arXiv preprint arXiv:1712.09665* (2017).

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When Deep Learning Met Code Search. *arXiv preprint arXiv:1905.03813* (2019).

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.

Augustin Cauchy. [n. d.]. Méthode générale pour la résolution des systemes d'équations simultanées. ([n. d.]).

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).

Yaniv David, Uri Alon, and Eran Yahav. 2019. Neural Reverse Engineering of Stripped Binaries. *arXiv preprint arXiv:1902.09122* (2019).

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

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1ersoRqtm

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014a. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

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

Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2016. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435* (2016).

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.

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

Hossein Hosseini, Baicen Xiao, Mayoore Jaiswal, and Radha Poovendran. 2017. On the limitation of convolutional neural networks in recognizing negative images. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 352–358.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. http://aclweb.org/anthology/P/P16/P16-1195.pdf

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

Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*. IEEE, 533–537.

Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. 2018. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv preprint arXiv:1802.04528* (2018).

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

Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533* (2016).

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated graph sequence neural networks. In *ICLR*.

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

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

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

Mehdi Mirza and Simon Osindero. 2014. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784* (2014).

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

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

Yurii Nesterov. 2013. *Introductory lectures on convex optimization: A basic course*. Vol. 87. Springer Science & Business Media.

Anh Nguyen, Jason Yosinski, and Jeff Clune. 2015. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 427–436.

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

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

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

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

Md Rafiqul Islam Rabin, Ke Wang, and Mohammad Amin Alipour. 2019. Testing Neural Program Analyzers. *ASE - Late Breaking Results* (2019).

Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. 2020. Semantic Robustness of Models of Source Code. *arXiv preprint arXiv:2002.03043* (2020).

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

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.

Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2018. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 490–510.

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

Joshua Saxe and Konstantin Berlin. 2015. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 11–20.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2008), 61–80.

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

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

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

Octavian Suciu, Scott E Coull, and Jeffrey Johns. 2019. Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 8–14.

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.

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).

Rohan Taori, Amog Kamsetty, Brenton Chu, and Nikita Vemuri. 2019. Targeted adversarial examples for black box audio systems. In *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 15–20.

Eric Wallace, Mitchell Stern, and Dawn Song. 2020. Imitation Attacks and Defenses for Black-box Machine Translation Systems. *arXiv preprint arXiv:2004.15015* (2020).

Qinglong Wang, Wenbo Guo, Kaixuan Zhang, Alexander G Ororbia II, Xinyu Xing, Xue Liu, and C Lee Giles. 2017. Adversary resistant deep neural networks with an application to malware detection. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1145–1153.

Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. 2017. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 288–302.