Adversarial Examples for Models of Code and Defending Against Them

Noam Yefet
Adversarial Examples for Models of Code and Defending Against Them

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Noam Yefet

Submitted to the Senate of the Technion — Israel Institute of Technology
Nisan 5781 Haifa April 2021
This research was carried out under the supervision of Prof. Eran Yahav, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:


The generous financial help of the Technion is gratefully acknowledged.
Contents

List of Figures

Abstract

1 Introduction
   1.1 Goal .................................................. 3
   1.2 Our Approach ....................................... 4
   1.3 Overview ............................................ 7
      1.3.1 Motivating Examples ........................... 7
      1.3.2 Discrete Adversarial Manipulation of Programs (DAMP) 8
   1.4 Related Work ......................................... 9

2 Preliminaries ........................................... 13
   2.1 Training Neural Networks ......................... 13
   2.2 Adversarial Examples .............................. 14

3 Adversarial Examples .................................. 17
   3.1 Adversarial Examples for Models of Code ........... 17
      3.1.1 Definitions .................................... 17
      3.1.2 Targeted Attack ................................. 18
      3.1.3 Non-targeted Attack ............................ 20
      3.1.4 Deriving by Integer Indices ................... 20
      3.1.5 Search ............................................. 21
      3.1.6 Convergence ...................................... 21
   3.2 Defense Against Adversarial Examples ............... 22
      3.2.1 Defense without Re-training .................... 22
      3.2.2 Defense with Re-training ........................ 23
   3.3 Evaluation ........................................... 25
      3.3.1 Setup .............................................. 25
      3.3.2 Attack ............................................. 27
      3.3.3 Defense ............................................ 31
      3.3.4 Additional Examples ............................ 35
4 Conclusion

A Additional Examples

Hebrew Abstract
List of Figures

1.1 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.

1.2 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.

1.3 A C# VarMisuse example which is classified correctly as Destination-Type in the method Equals by the GGNN model of [ABK18]. Given the code in Figure 1.3a and the targetSourceType, our approach renames a local variable destination in another method to the specific name sesqblhj, 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.

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

3.1 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 $y_{bad}$.

3.2 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.
3.3 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 1.2 and repeated here for clarity.

3.4 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 3.3. 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.

3.5 The Outlier Detection defense: the trade-off between robustness and performance-while-not-under-attack, with respect to the similarity threshold $\sigma$ on the validation set. The dashed vertical line at $\sigma = 2.7$ denotes the value that we chose for $\sigma$ 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.

3.6 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.

3.7 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.

3.8 A C# VarMisuse example which is classified correctly as _getterBuilder in the method Getter 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 Getter method, thus introducing a real bug in the method Getter.

3.9 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.1 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.

A.2 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.

A.3 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.
Abstract

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.
Chapter 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 [RVK15, AZLY18, BRV16, ABK18], code summarization [APS16, ABLY19, FAB19], code generation [BAGP19, MCJ17, ASLY19], code search [SLL+18, LKM+19, CLK+19], and bug finding [PS18, RAJ+17, SBC19].

In other domains such as computer vision, deep models have been shown to be vulnerable to adversarial examples [SZS+13, GSS14]. Adversarial examples are inputs 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 [SZS+13, GSS14, GPAM+14, NYC15, MO14, PMJ+16, KGB16, PMG+17, MDFF16] and for other domains [CW18, TKCV19, ERLD17, ASE+18, ABS18, BB17, PDL19]. 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 [SZS+13, GSS14]. 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 work, we present a novel approach for generating adversarial examples for neural models of code.

More formally:

1.1 Goal

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

The main challenge in tackling the above goal lies in exploring the vast space of
Correctly predicted example

Adversarial perturbations

<table>
<thead>
<tr>
<th>Target: contains</th>
<th>Target: escape</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void f1(int[] array){</strong></td>
<td><strong>void f1(int[] array){</strong></td>
</tr>
<tr>
<td>boolean swapped = true;</td>
<td>boolean swapped = true;</td>
</tr>
<tr>
<td>for (int i = 0; i &lt; array.length &amp;&amp; swapped; i++) {</td>
<td>for (int i = 0; i &lt; array.length &amp;&amp; swapped; i++) {</td>
</tr>
<tr>
<td>swapped = false;</td>
<td>swapped = false;</td>
</tr>
<tr>
<td>for (int j = 0; j &lt; array.length-1-i; j++) {</td>
<td>for (int j = 0; j &lt; array.length-1-i; j++) {</td>
</tr>
<tr>
<td>if (array[j] &gt; array[j+1]) {</td>
<td>if (array[j] &gt; array[j+1]) {</td>
</tr>
<tr>
<td>int temp = array[j];</td>
<td>int temp = array[j];</td>
</tr>
<tr>
<td>array[j] = array[j+1];</td>
<td>array[j] = array[j+1];</td>
</tr>
<tr>
<td>array[j+1] = temp;</td>
<td>array[j+1] = temp;</td>
</tr>
<tr>
<td>swapped = true;</td>
<td>swapped = true;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td><strong>}</strong></td>
<td><strong>}</strong></td>
</tr>
</tbody>
</table>

**void f2(int[] ttypes){** | **void f2(int[] ttypes){** |
| boolean swapped = true; | boolean swapped = true; |
| for (int i = 0; i < ttypes.length && swapped; i++) { | for (int i = 0; i < ttypes.length && swapped; i++) { |
| swapped = false; | swapped = false; |
| for (int j = 0; j < ttypes.length-1-i; j++) { | for (int j = 0; j < ttypes.length-1-i; j++) { |
| if (ttypes[j] > ttypes[j+1]) { | if (ttypes[j] > ttypes[j+1]) { |
| int temp = ttypes[j]; | int temp = ttypes[j]; |
| ttypes[j] = ttypes[j+1]; | ttypes[j] = ttypes[j+1]; |
| ttypes[j+1] = temp; | ttypes[j+1] = temp; |
| swapped = true; | swapped = true; |
| } | } |
| **}** | **}** |

**void f3(int[] array){** | **void f3(int[] array){** |
| boolean swapped = true; | boolean swapped = true; |
| for (int i = 0; i < array.length && swapped; i++) { | for (int i = 0; i < array.length && swapped; i++) { |
| swapped = false; | swapped = false; |
| for (int j = 0; j < array.length-1-i; j++) { | for (int j = 0; j < array.length-1-i; j++) { |
| if (array[j] > array[j+1]) { | if (array[j] > array[j+1]) { |
| int temp = array[j]; | int temp = array[j]; |
| array[j] = array[j+1]; | array[j] = array[j+1]; |
| array[j+1] = temp; | array[j+1] = temp; |
| swapped = true; | swapped = true; |
| } | } |

**int upperhexdigits;** | **int upperhexdigits;** |

Figure 1.1: 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.

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:

- rename variables and
- 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 thesis, therefore, we focus on the above two semantic-preserving transformations, which can be safely applied without any semantic analysis.

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 1.2, and detailed in Section 3.1.

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. 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 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.

As recently shown by [WSS20], 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.
Figure 1.3: A C# Varmisuse example which is classified correctly as DestinationType in the method Equals by the GGNN model of [ABK18]. Given the code in Figure 1.3a 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.

Main Contributions The contributions of this thesis 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 [AZLY19], GGNN [ABK18], and GNN-FILM [BAGP19] 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.
1.3 Overview

In this section, we provide an informal overview.

1.3.1 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 [AZLY19] model to predict a label of our choice. Consider the code snippet $f_1$ of Figure 1.1. This code snippet sorts a given array. The code2vec model [AZLY19] applied to this code snippet predicts the correct name, sort, with a probability of 98.54%.

Given the code snippet $f_1$ and the arbitrarily chosen adversarial target contains, our approach finds that renaming the original variable array to ttypes in $f_2$ 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 $f_1$ and the adversarial target escape, our approach finds that adding an unused variable (i.e., dead code) named upperhexdigits to the end of $f_1$ forces the model to predict the label escape with a probability of 100% (see $f_3$ in Figure 1.1). 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 [DAY19, SB15, WGZ+17, ASH+14, GPM+16]. In this thesis, we demonstrate the ability to bypass semantic labeling by applying our attack on the code2vec model (Figure 1.1), 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 1.3a, a GNN model trained on the VarMisuse task [ABK18, Bro19] 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 sesqblj (Figure 1.3b), 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 [PS18, RAJ+17, SBC19] can allow a malicious developer inside an organization or inside an open-source project to intentionally introduce bugs. In this thesis, we demonstrate this ability using the VarMisuse on Graph Neural Networks (GNNs) (Figure 1.3), forcing the model to choose an incor-
rect (but type-correct) variable.

In addition to the code2vec and VarMisuse tasks that we address in this thesis, we believe adversarial examples can be applied to neural code search [SLL+18, LKM+19, CLK+19]. 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.

1.3.2 Discrete Adversarial Manipulation of Programs (DAMP)

Consider the code snippet f1 of Figure 1.1 that sorts a given array. The code2vec model [AZLY19] 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) [GSS14] approach, which modifies the input using a small fixed $\epsilon$ value.

Deriving with Respect to a Discrete Input

In settings where the input is discrete, the first layer of a neural network is typically an embedding layer that embeds discrete objects, such as names and tokens, into a continuous space [ABLY19, APS16, IKCZ16]. 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 $f_1$ of Figure 1.1 and the desired target label *contains*, our approach for generating adversarial examples automatically infers the snippet $f_2$ of Figure 1.1. Similarly, given the target label *escape*, our approach automatically infers the snippet $f_3$ of Figure 1.1.

All code snippets of Figure 1.1 are semantically equivalent. The only difference between $f_1$ and $f_2$ is the name of the variables. Specifically, these snippets differ only in the name of a single variable, which is named array in $f_1$ and ttypes in $f_2$. Nevertheless, when array is renamed to ttypes, the prediction made by code2vec changes to the desired (adversarial) target label contains. The difference between $f_1$ and $f_3$ is the addition of a single variable declaration int upperhexdigits, which is never used in the code snippet. Nevertheless, adding this declaration changes the prediction made by the model to the desired (adversarial) target label *escape*.

### 1.4 Related Work

**Adversarial Examples of Images** [SZS+13] 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. [GSS14] 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 [ERLD17] 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. [ASE+18] presented a technique for generating “semantically and syntactically similar adversarial examples” that attack trained models. The main idea is to replace a random
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”. [KBAR+18], [SCJ19] and [KDB+18] 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, [KDB+18] 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 1.3.2. In contrast, we derive the loss by the distribution over indices (Section 3.1.4), which directly optimizes towards the target label; this allows us to perform targeted attacks in multi-class models. [RSRE18] addressed a very different scenario and modified malicious programs to mimic benign calls to APIs at runtime. [YKXG17] 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. [RWA19] identified 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. [RHW+20] 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. [BV20] 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.

[PDL19] 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 3.2.1) 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 [PDL19] detects character-level typos.
Chapter 2

Preliminaries

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

2.1 Training Neural Networks

A Neural Network (NN) model can be viewed as a function \( f_\theta : X \to Y \), where \( X \) is the input domain (image, text, code, etc.) and \( Y \) is usually a finite set of labels. Assuming a perfect classifier \( h^* : X \to Y \), the goal of the function \( f_\theta \) 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, \( f_\theta \) contains a set of trainable weights, denoted by \( \theta \), 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 \( \theta \) (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:

\[
\theta^* = \arg\min_{\theta} \sum_{(x, y) \in T} J(\theta, x, y) \tag{2.1}
\]

One of the most common algorithms to approximate the above problem is gradient descent [Cau] using backpropagation [Kel60]. When gradient descent is used for training, the following update rule is applied repeatedly to update the model’s weights:

\[
\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta J(\theta_t, x, y) \tag{2.2}
\]

where \( \eta \in \mathbb{R} \) 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 2.1, where \( \theta \) contains a single trainable variable \( \theta \in \mathbb{R} \).
2.2 Adversarial Examples

Neural network models are very popular and have been applied in many domains, including computer vision [KSH12, SZ14, SLJ+15, HZRS16], natural language [MKB+10, HS97, CVMG+14], and source code [RVK15, BRV16, BPS18, AZLY18, APS16, ABLY19, BAGP19, MCJ17, LCJM17, SLL+18, LKM+19, PS18, RAJ+17, ABK18].

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 [SZS+13, GSS14, CW18]. 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 [ASE+18]. Another approach is to insert typos into words by replacing a few characters in the text [BB17, ERLD17]. 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.
Chapter 3

Adversarial Examples

3.1 Adversarial Examples for Models of Code

In this section we describe the process of generating adversarial examples using our DAMP approach.

3.1.1 Definitions

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

Given a code snippet $c \in \mathcal{C}$ 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 [BV20]: 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 $\delta : \mathcal{C} \rightarrow \mathcal{C}$ to $c$, such that $\delta \in \Delta(c)$. For instance, if $c$ is the code snippet in Figure 1.3a, then Figure 1.3b shows $\delta_{\text{destination} \rightarrow \text{scsqbhj}}(c)$ – the same code snippet where $\delta_{\text{destination} \rightarrow \text{scsqbhj}}$ is the modification that renames the variable destination to scsqbhj. As a result, the prediction of the model changes from $y=\text{DestinationType}$ to $y_{bad}=\text{SourceType}$, and $y = f_\theta(c) \neq f_\theta(\delta_{\text{destination} \rightarrow \text{scsqbhj}}(c)) = y_{bad}$.

In this thesis, 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 ($\text{VarName}$). Thus, we define the possible perturbations that we consider as $\Delta(c) = \{\delta_{v \rightarrow v'} \mid \forall c \in \mathcal{C} : \delta_{v \rightarrow v'}(c) = c_{v \rightarrow v'}\}$, 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 ($\text{DeadCode}$ insertion, see Section 3.3.1). 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 $\text{Var}(c)$ as the set of all local variables existing in $c$. The adversary’s objective is to thus select a single variable $v \in \text{Var}(c)$ 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 $f_\theta (\delta_{v \rightarrow v'} (c)) = y_{bad}$.

**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 [SZS$^+$13, CW18], 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 thesis, 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:

1. each variable appears in several places in the code, so a single renaming can induce multiple perturbations;
2. the adversarial code can still be compiled and therefore stays in the code domain; and
3. 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 backpropagation 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.

**3.1.2 Targeted Attack**

In this kind of attack, our goal is to make the model predict an incorrect desired label $y_{bad}$ by renaming a given variable $v$ to $v'$. Let $\theta$ be the learned parameters of a model, $c$ be the input code snippet to the model, $y$ be the target label, and $J(\theta, c, y)$ be the loss function used to train the neural model. As explained in Section 2.1, when training
using gradient descent, the following rule is used to update the model parameters and minimize $J$:

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

(3.1)

We can apply a gradient descent step with $y_{\text{bad}}$ as the desired label in the loss function, and derive with respect to any given variable $v$:

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

(3.2)

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 $y_{\text{bad}}$ as the correct label. This is illustrated in Figure 3.1.

The loss $J(\theta, c, y_{\text{bad}})$ in Equation (3.2) 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 (2.2)) is that Equation (3.2) 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 ($\nabla_{\theta}$) in Equation (2.2).

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 3.3 and detailed in Section 3.1.4.
Figure 3.3: 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 1.2 and repeated here for clarity.

3.1.3 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:

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

This rule can be illustrated as taking a step toward the steepest ascent in the loss function (Figure 3.2).

**Targeted vs. Non-targeted Attacks** Notice the difference between Equation (3.2) and Equation (3.3). In Equation (3.2), 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 (3.3), 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.

3.1.4 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 3.3).

3.1.5 Search

Sometimes, adversarial examples can be found by applying the adversarial step of Equation (3.2) 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.

3.1.6 Convergence

In general, gradient descent is guaranteed to converge to a global minimum only when optimizing convex functions [Nes13]. 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 3.3) when we restrict the perturbation to a single variable.

As discussed in Section 3.1.5, 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 [SGPV19, BBS+19], 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 [SGPV19] and 45K parameters [BBS+19]; 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.

### 3.2 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.

#### 3.2.1 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 $f_\theta$, which is independent of $g$. The goal of $g$ is to fix the given input, if necessary, in a way that will allow $f_\theta$ 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 $f_\theta$ 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:

$$z^* = \arg\max_{z \in Var(c)} \left\| \frac{\sum_{v \in Sym(c), v \neq z} vec(v)}{|Sym(c)|} - vec(z) \right\|_2$$ (3.4)

We then define a threshold $\sigma$ that determines whether $z^*$ is an outlier. If the $L_2$ distance between the vector of $z$ and the average of the rest of the symbols is greater than $\sigma$, then $z$ is replaced with an UNK symbol; otherwise, the code snippet is left untouched. Practically, the threshold $\sigma$ 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 3.3.3.

### 3.2.2 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 [GSS14] 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 3.1.3). 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 [HXJP17] 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**, $|\text{vocab}|=\{10k, 50k, 100k\}$ – 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.
3.3 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 [RHW+20] and [BV20]. 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.

3.3.1 Setup

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 [AZLY19] 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 [LTBZ16] as an extension to Graph Neural Networks (GNNs) [SGT+08]. 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 [CVMG+14] cell. GGNNs were later adapted to source code tasks by [ABK18], 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 [Bro19]. 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.

**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 3.1). 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 3.3.2. 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.

Dataset

code2vec - Java  We evaluated our proposed attack and defense on code2vec using the Java-large dataset [ABLY19]. 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 [ABK18]. 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.

3.3.2 Attack

We focus on two main attack tasks: targeted and non-targeted attacks. For targeted attacks, we used Equation (3.2) as the adversarial step. For non-targeted attacks, we used Equation (3.3) as the adversarial step. For the desired adversarial labels, we randomly sampled labels that occurred at least 10K times in the training set.

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.

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 $y_{bad}$: $\text{TFIDF}(y_{bad}) = \arg\max_v \frac{\#(y_{bad}, v)}{\#(v)}$.

**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.

**Attack - Results**

Table 3.1 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.
Table 3.1: 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.

<table>
<thead>
<tr>
<th>VarName (robustness %)</th>
<th>DeadCode (robustness %)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Non-targeted</strong></td>
<td></td>
</tr>
<tr>
<td>RandomVar</td>
<td>34.10</td>
</tr>
<tr>
<td>TFIDF</td>
<td>53.53</td>
</tr>
<tr>
<td>DAMP</td>
<td><strong>6.00</strong></td>
</tr>
<tr>
<td>RandomVar</td>
<td>54.90</td>
</tr>
<tr>
<td>TFIDF</td>
<td>84.00</td>
</tr>
<tr>
<td>DAMP</td>
<td><strong>21.83</strong></td>
</tr>
<tr>
<td><strong>Targeted:</strong></td>
<td></td>
</tr>
<tr>
<td>CopyTarget</td>
<td></td>
</tr>
<tr>
<td>init</td>
<td>84.47</td>
</tr>
<tr>
<td>mergeFrom</td>
<td>72.79</td>
</tr>
<tr>
<td>size</td>
<td>99.47</td>
</tr>
<tr>
<td>isEmpty</td>
<td>88.61</td>
</tr>
<tr>
<td>clear</td>
<td>89.56</td>
</tr>
<tr>
<td>remove</td>
<td>84.94</td>
</tr>
<tr>
<td>value</td>
<td>99.77</td>
</tr>
<tr>
<td>load</td>
<td>86.75</td>
</tr>
<tr>
<td>add</td>
<td>92.88</td>
</tr>
<tr>
<td>run</td>
<td>95.11</td>
</tr>
<tr>
<td>CopyTarget</td>
<td></td>
</tr>
<tr>
<td>init</td>
<td>74.33</td>
</tr>
<tr>
<td>mergeFrom</td>
<td><strong>9.01</strong></td>
</tr>
<tr>
<td>size</td>
<td><strong>77.91</strong></td>
</tr>
<tr>
<td>isEmpty</td>
<td>98.11</td>
</tr>
<tr>
<td>clear</td>
<td>89.00</td>
</tr>
<tr>
<td>remove</td>
<td>84.83</td>
</tr>
<tr>
<td>value</td>
<td><strong>71.81</strong></td>
</tr>
<tr>
<td>load</td>
<td>85.46</td>
</tr>
<tr>
<td>add</td>
<td>86.72</td>
</tr>
<tr>
<td>run</td>
<td><strong>40.92</strong></td>
</tr>
</tbody>
</table>

Count best: 0 4 6 0 2 8

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 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 3.2 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
Table 3.2: 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.

<table>
<thead>
<tr>
<th></th>
<th>GGNN Robustness</th>
<th>GNN-FiLM Robustness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CharBruteForce</td>
<td>DAMP (this work)</td>
</tr>
<tr>
<td>Non-targeted</td>
<td>96.24</td>
<td>57.99</td>
</tr>
<tr>
<td>targeted</td>
<td>97.84</td>
<td>69.00</td>
</tr>
</tbody>
</table>

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.

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 [Bro19] 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.
3.3.3 Defense

We experimented with all defense techniques as described in Section 3.2, 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.

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 $\sigma$ 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 3.3.3).

Defense - Results

The effectiveness of the different defense techniques is presented in Table 3.3. 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.
Table 3.3: 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 3.3.1). 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.

<table>
<thead>
<tr>
<th>Performance (not under attack)</th>
<th>VARNAME Robustness (%)</th>
<th>DEADCODE Robustness (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
</tr>
<tr>
<td>No Vars</td>
<td>78.78</td>
<td>80.83</td>
</tr>
<tr>
<td>Outlier Detection</td>
<td>98.18</td>
<td>97.75</td>
</tr>
<tr>
<td>Train Without Vars</td>
<td>89.74</td>
<td>90.86</td>
</tr>
<tr>
<td>Adversarial Training</td>
<td>98.09</td>
<td>92.96</td>
</tr>
<tr>
<td>Adversarial Fine-Tuning</td>
<td>85.51</td>
<td>84.86</td>
</tr>
<tr>
<td>No Vars</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[vocab</td>
<td>]=10k</td>
<td>90.50</td>
</tr>
<tr>
<td>[vocab</td>
<td>]=50k</td>
<td>94.51</td>
</tr>
<tr>
<td>[vocab</td>
<td>]=100k</td>
<td><strong>96.84</strong></td>
</tr>
</tbody>
</table>

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 3.4.

Table 3.4 shows the performance of the Outlier Detection and the Adversarial Training defenses across randomly sampled adversarial labels.

**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 3.5 shows the trade-off with respect to the similarity threshold.

Technion - Computer Science Department - M.Sc. Thesis MSC-2021-30 - 2021
Figure 3.4: 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 3.3. 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 3.4: 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 3.1.

<table>
<thead>
<tr>
<th>VARNAME (robustness %)</th>
<th>DeadCode (robustness %)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DAMP† + Outlier Detection</td>
</tr>
<tr>
<td>init</td>
<td>48.44</td>
</tr>
<tr>
<td>mergeFrom</td>
<td>10.39</td>
</tr>
<tr>
<td>size</td>
<td>78.27</td>
</tr>
<tr>
<td>isEmpty</td>
<td>79.04</td>
</tr>
<tr>
<td>clear</td>
<td>82.8</td>
</tr>
<tr>
<td>remove</td>
<td>63.15</td>
</tr>
<tr>
<td>value</td>
<td>76.75</td>
</tr>
<tr>
<td>load</td>
<td>55.65</td>
</tr>
<tr>
<td>add</td>
<td>68.6</td>
</tr>
<tr>
<td>run</td>
<td>51.52</td>
</tr>
</tbody>
</table>
Figure 3.5: The Outlier Detection defense: the trade-off between robustness and performance-while-not-under-attack, with respect to the similarity threshold $\sigma$ on the validation set. The dashed vertical line at $\sigma = 2.7$ denotes the value that we chose for $\sigma$ 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 \texttt{VARNAME} attacks.
Figure 3.6: 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.

\( \sigma \). Small values of \( \sigma \) make the model with Outlier Detection defense more robust. It becomes almost as robust as No Vars, but perform worse in terms of F1 score. As the value of \( \sigma \) increases, the model with Outlier Detection defense becomes less robust, but performs better while not under attack.

3.3.4 Additional Examples

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

Figure 3.6 shows additional targeted attacks against the “sort” example from Figure 1.1. 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 3.7, 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 [BMR17], 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 3.8 shows A C# **VAR_MISUSE** 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 3.9 shows a similar GNN targeted adversarial example.

Other **CODE2VEC** and GNNs examples are shown in Appendix A.
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);
    }
}
```

Adversarial perturbation

```
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);
    }
}
```

(a)

Figure 3.8: 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;
}
```

Adversarial perturbation

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

(a)

(b)

Figure 3.9: 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.
Chapter 4

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 retraining.

We believe the principles presented in this thesis 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.
Appendix A

Additional Examples

```java
String f(String txt) {
    txt = replace(txt, "&", "&amp;");
    txt = replace(txt, ",", ",&quote;");
    txt = replace(txt, "\", ",lt;");
    txt = replace(txt, ">", ",gt;");
    return txt;
}
```

Prediction: escape

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

Prediction: contains (94.99%)

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

Prediction: done (77.68%)

Figure A.1: 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.
Figure A.2: 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.
Figure A.3: A C# VAR Misuse 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.
Bibliography


[RVK15] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual


- Adversarial Training - מחברת את стрטגיות ההגנה עם אימון מחודש Frauenfelder)
- outlier detection - מחברת את стрטגיות ההגנה ללא אימון מחודש Frauenfelder)
The thesis title is "Modeling the Code as a Defense Against Attacks".

The thesis is about using code representation to improve the robustness of models in defense against attacks.

The first part of the thesis introduces a new model called "Model Guardian" which is a model that can be used to detect outliers in the code.

The model is trained on a dataset that contains examples of outliers such as unusual identifiers, literals, and embeddings.

The model is trained in a semi-supervised manner, where the input is first passed through the model, and then the output is used to calculate the loss.

The advantage of this approach is that it allows for the improvement of each model independently.

The thesis also describes an approach to identifying unusual identifiers by removing them from the code.

The approach uses a hyperparameter, \( \sigma \), to determine whether a code token is an outlier or not.

The thesis concludes that the optimal value for \( \sigma \) is 2.7, and that this value results in the best performance on the validation set.

The thesis also discusses the importance of code representation and the role of code2vec in this process.

The thesis is a joint work between the Computer Science Department of Technion and M.Sc. Thesis MSC-2021-30-2021.
robustness.

cases, we can evaluate the model's performance on adversarial examples. The main goal of adversarial training is to make the model robust against such examples. However, this process can also make the model less efficient for regular examples. Therefore, it is necessary to find a balance between robustness and efficiency.

In conclusion, we have demonstrated that adversarial training is effective in enhancing the model's robustness against adversarial examples. The experiments conducted in this thesis confirmed the results of previous studies and provided new insights into the use of adversarial training in various applications.

References


An Open Framework

Discrete Adversarial Manipulation of Programs

can be formulated as a minimization problem over the gradient descent DAMP. The attack is formulated as a gradient descent over the loss function, which is minimized using a gradient descent optimizer.

The function minimizes the loss function with respect to the model parameters and updates the model parameters using the gradient descent optimizer.

non-targeted attack:

1. Choose a program and formulate the attack function as a minimization problem over the loss function.
2. Minimize the attack function with respect to the model parameters.
3. Update the model parameters using the gradient descent optimizer.

targeted attack:

1. Choose a program and formulate the targeted attack function as a minimization problem over the loss function.
2. Minimize the targeted attack function with respect to the model parameters.
3. Update the model parameters using the gradient descent optimizer.

Results

The attack framework was evaluated on different types of models and datasets, including Java code2vec and C#.

Technion - Computer Science Department - M.Sc. Thesis MSC-2021-30 - 2021
תקציר

מודלים רשתות עצביות מראים מבנים רבים בתוכניות ישנים כדי להכין צעדים++, סיכום, כנף. לפניAAF התחים של התחום נראה, מודלים אלו ראה תוכנית ב commence progeny של ההוכחהagination תהליך של קטעי 신פה של התוכנית. 

ויתור על קטעי תהליך של תוכנית יוחאי בגנים.

כדי lượtיניא בציור, מודלים אלו תעכבם תהליך של קבעה על הקצבה במודח את למעוז את התוכנית . ראשתו,

מודדת מתוכנית הפרוספקציה יוצק צעדים של התוכנית. לאחר מכת, בניגה ולאנוגריפת descent המתחבצ חונים של התוכנית הפסקולות של התוכנית.

adversarial examples אנומצאנוכיםמודלים אלו י ACK לשリアת ייעודים כנוגדו אנטונומית ירוב שאז, דגamedi יירבה לא תראה לפי מרחביון ומכים פוספיים את החוב.

דוגמות יירוב במודלים של קוד

בנוגו התוכניות יסיס בכת הקול של האップ (כおよג כתות התוכנית), בור לעיריה דגמה יירבה מתכאת על ידי 디 שיניום הפיקסלים של התוכנית כתות, פעיל ואצי אפיבולית בתוכנית מאמץ ואות משתנה

בידי. לאש כל, Şiינר שיפקטר דגמה יירבה תדהאמ ב-דיפיידי שיניים שמי משלוטת סמלים תכנית

cכל התאמור. לאס פספור, בחרו את שיניים הביאים:

• כומ-כ, צוור תוקפ את המודל והתוכנית בין מטרות: VarName
• בחירת מעדכון קיים בתוכניתشيי של לארוג מציאת דגמה יירבה. (dead code)
• תוספת של קוד מת: DeadCode
• מעתה של קיים בתוכניתלש מציאת דגמה יירבה. את שמי מערפה המצלת התוכנית והורה

ולמענה בתוכנית שיש שביי וירבד:

• VarName
• כומ-כ, צוור תוקפ את המודל והתוכנית בין מטרות: non-targeted attack
• תוצאת אלعنا הכלל.
• מציאת דגמה יירבה שלתוכה לא להודול. כלומר, המודל יירדה
• targetted attack
• מציאת דגמה יירבה שלתוכה להודולตนשמות. כלומר, המודל

יתוהי ת gözüにくい (לזנכיה) שקבעה על ידי התוכן.
דוגמאות יריבות במודלים של קוד
והגה הפגד

יובהר על מחקר

לשם مليים חלקיים של הדרישת להבלט התוכן
מוניטורי למדעי החיים המחשב

נעם יפת

wayne על הפק

הנהלreasonable לתוכני – מכן טכנולוגיה לישראל
נשחתפש'א חיפה אפריל 2021
דוגמאות יריבות במודלים של קוד
ואגף בגדוד

נעם יפת