Semantic Robustness of Models of Source Code

Jordan Henkel*†, Goutham Ramakrishnan*‡, Zi Wang†, Aws Albarghouthi†, Somesh Jha†, and Thomas Reps†
†University of Wisconsin–Madison, Madison, WI, USA
{jjhenkel,zw,aws,jha,reps}@cs.wisc.edu
‡Health at Scale Corporation, San Jose, CA, USA
goutham7r@gmail.com

***Abstract*—Deep neural networks are vulnerable to adversarial examples—small input perturbations that result in incorrect predictions. We study this problem for models of source code, where we want the neural network to be robust to source-code modifications that preserve code functionality. To facilitate training robust models, we define a powerful and generic adversary that can employ sequences of parametric, semantics-preserving program transformations. We then explore how, with such an adversary, one can train models that are robust to adversarial program transformations.**

We conduct a thorough evaluation of our approach and find several surprising facts: we find robust training to beat dataset augmentation in every evaluation we performed; we find that a state-of-the-art architecture (code2seq) for models of code is harder to make robust than a simpler baseline; additionally, we find code2seq to have surprising weaknesses not present in our simpler baseline model; finally, we find that robust models perform better against unseen data from different sources (as one might hope)—however, we also find that robust models are not clearly better in the cross-language transfer task. To the best of our knowledge, we are the first to study the interplay between robustness of models of code and the domain-adaptation and cross-language-transfer tasks.

I. Introduction

While deep neural networks have been widely adopted in many areas of computing, it has been repeatedly shown that they are vulnerable to *adversarial examples* [1, 2, 3, 4]: small, seemingly innocuous perturbations to the input that lead to incorrect predictions. For instance, making a small imperceptible modification to pixels of an image may cause a neural network to change its prediction. Adversarial examples raise safety and security concerns, for example, in computer-vision models used in autonomous vehicles [5, 6] or for user authentication [7]. Significant progress has recently been made in identifying adversarial examples and training models that are robust to such examples. However, the majority of the research has targeted computer-vision tasks [8, 9, 1], a *continuous* domain. (See Kolter and Madry [10] for a comprehensive overview.)

In this paper, we study the problem of robustness to adversarial examples in the *discrete* domain of deep neural networks for source code With the growing adoption of neural models for programming tasks [11, 12, 13, 14, 15, 16, 17, 18, 19, 20], robustness is becoming an important property. Why do we want robust models of code? There are many answers, ranging

```
int □(Object target) {
  System.out.println("Begin search");
  int i = 0;
  for (Object elem: this.elements) {
      if (elem.equals(target)) {
          System.out.println("Found");
          return i;
      }
      i++;
  }
  return -1;
}
```

Fig. 1: code2seq [21] correctly predicts the function name "`indexOfTarget`." After the highlighted logging statements are added, it predicts "`search`."

from usability to security. Consider, for instance, a model that explains in English what a piece of code is doing—the *code-captioning* task. A developer using such a model to navigate a new code base should not receive completely different explanations for similar pieces of code. For a concrete example, consider the behavior of the state-of-the-art code2seq model [21] on the Java code in Fig. 1, where the prediction changes after logging print statements are added. Such behavior (changing output based on irrelevant detail) is the result of an over-sensitive (and under-robust) model.

The Problem. Many machine-learning-on-code models are *not robust*. Furthermore, designing robust training methods, in the space of *discrete* programs, is a difficult task—existing *continuous* methods cannot be directly applied.

With images, adversarial examples involve *small* changes that are imperceptible to a human. With code, there is no analogous notion of a change imperceptible to a human. Consequently, we consider attacks based on *semantics-preserving transformations*. Because the original program's semantics is preserved, the program that results from the attack must have the same behavior as the original. Using the idea of adversarial examples generated by *semantics-preserving transformations*, we set out to meet the following goal:

Our Goal. Find a way to train robust models of code and, in doing so, build a framework that enables experimentation with different program transforms, models, datasets and programming languages.

Through our efforts to meet our goal of training robust models, we make the following contributions:

*Equal contributions. Work completed at the University of Wisconsin–Madison.

Our Contributions. We provide a novel and generic adversary, methods for training robust models of source code, and a framework that has (already) allowed others to push the boundaries of adversaries in the domain of models of code. We perform an extensive evaluation against code-summarization models, but our framework is generic and capable of supporting any model that receives source code as input.

A. *Our Approach to Semantic Robustness*

We present a novel and generic approach for defining an adversary that manipulates a program to fool a neural network. In particular, we structure an adversary in terms of two operations: *semantics-preserving transformations* and *resolvers*. Transformations construct a program *sketch* [22]—an incomplete program with holes—and resolvers fill the holes to produce a program that fools the neural network. This insight allows us to represent a wide range of adversaries, including adversaries formulated in concurrent and follow-on work. Furthermore, our approach to adversaries (and the overall framework we built) has already enabled others to produce new state-of-the-art attacks. We take this as strong validation of our approach, and we hope that others will continue to build on our work to push the boundaries of adversaries in the domain of models of code.

We also demonstrate how to train models of source code that are robust to such adversaries, using *robust-optimization* ideas that are prevalent in image recognition [9]. Aside from contributing a generic adversary and a robust training approach, we also contribute a framework, AVERLOC, for producing both the adversaries and the robust-training pipelines required to carry out extensive evaluations.

B. *The AVERLOC Framework*

AVERLOC provides all of the components necessary to meet our goal of training robust models of code and, as we will describe, the AVERLOC framework is even able to support the methodology used in recent (and concurrent) work by others on robust training for models of code. AVERLOC requires a user to provide data, a model, and a loss function. Given those components, AVERLOC will produce an adversary, a data augmentor, and a surrogate to an adversarial-loss function. Optionally, users can leverage our pre-existing datasets and (code-summarization) models. Note that, although we evaluate models trained to perform the code-summarization task, AVERLOC supports any model taking code as input. For more details on the AVERLOC framework, see Sec. III.

C. *Evaluation of Semantic Robustness*

Our approach to generic adversaries and our framework (which embodies this approach) allows us to provide an extensive evaluation. In our evaluation, we answer the following five research questions (and provide concrete and actionable data to inform practitioners in their use of models of code):

(RQ1)*: How effective are the individual transforms we provide when used as attacks?* One of our key contributions is a generic adversary for models of code built on a library of *semantics-preserving* transforms. We implement eight such transforms, and, in this research question, measure the strength of each transform in isolation against two separate model architectures.

(RQ2)*: How effective is robust training in defending against our attacks? Are there any simpler baselines that perform well?* After understanding our attacks in isolation, we move to evaluating several pipelines for defense.

(RQ3)*: Does training with a weak adversary help with defending against a* strong *adversary?* Can we train on weaker adversaries and still retain (some) robustness when we test against much stronger adversaries?

(RQ4)*: What is the effect of robust training on the performance of models for the domain-adaptation task?* Domain adaptation asks if models trained on data from one domain can be applied to data from another different (yet similar) domain while retaining (some of) the model's original performance. To the best of our knowledge, we are the first to investigate the interplay between robustness and *domain adaptation* for models of code. If robust models perform well on unseen data (taken from sufficiently different data sources) then robust training may be desirable not only for defense against attacks, but also for increased performance in the face of unseen data.

(RQ5)*: What is the effect of robust training on the performance of models for the cross-language-transfer task?* Again, to the best of our knowledge, we are the first to provide preliminary investigations into the effect of robust training on the cross-language-transfer task. If models can work across languages (like simple seq2seq models can), then robust training can increase their cross-language performance.

In our evaluation, we find several surprising facts: although vanilla code2seq is more robust than a simpler seq2seq baseline (before applying any kind of robust training—likely due to code2seq's use of *program structure*), it is up to 1.5x more vulnerable to some of our attacks; furthermore, to our surprise, we find that it is *harder to make code2seq robust*, which results in robustly trained seq2seq models having the best overall performance; additionally, we find that robust training beats dataset augmentation in every evaluation we performed; finally, we find that robust models perform better against unseen data from different sources (as one might hope)—however, we also find that robust models are not clearly better in the cross-language-transfer task. *In summary, we train over 32 models, perform hundreds of individual evaluations, summarize our data, and answer each of our five research questions. With these extensive results, we hope that researchers and practitioners alike can gain a better understanding of robustness in the space of models on code.*

II. Semantic Robustness

In this section, we describe (1) our novel adversarial attack techniques, and (2) how to train semantically robust models for source-code tasks.

A. Semantic Adversaries

Adversaries by example: Throughout this section, we imagine a fixed deep neural network N over source code: given a piece of code P, it returns a prediction y, e.g., a textual description of what P does. The goal of an adversary is to transform P into a semantically equivalent P' that fools the neural network into making a wrong prediction. Formally, we denote an adversary as a function $A(N, P)$; the adversary attempts to produce a program P' that is equivalent to P and makes N produce a wrong prediction.

An adversary is equipped with a set of *semantics-preserving transforms*, e.g., adding dead code or print statements. Most transforms are parametric, e.g., if one adds a print statement, one has to also decide on the text to print. Therefore, we think of a transform as producing a program *sketch* [22]—a program with holes. For example, consider the following program:

```
public void incrementWeight(double weight) {
    this.weight += weight;
}
```

Applying the *insert print statement* transform produces the following sketch, where $\bullet_1$ is a hole that the adversary needs to fill with text.

```
public void incrementWeight(double weight) {
    this.weight += weight;
    System.out.println("●1");
}
```

An adversary may decide to apply multiple transforms, for example, the one we describe above plus a transform that changes the name of function arguments. Continuing our example, this compound transform produces the following sketch with two holes $\bullet_1$ and $\bullet_2$. (There are two occurrences of hole $\bullet_2$.)

```
public void incrementWeight(double ●2) {
    this.weight += ●2;
    System.out.println("●1");
}
```

After applying a number of transforms, the adversary needs to fill in the holes of the resulting sketch to produce a complete program that fools the neural network into changing its prediction. Our adversaries apply multiple transforms in a random order. However, it is possible to extend our adversaries such that transforms are applied according to user-supplied heuristics.

Adversary spectrum: We now describe how one designs an adversary algorithmically, assuming a fixed set of transforms at the adversary's disposal.

In our illustration above, we notice that an adversary has to make two choices:

1) **Transform**: Choose a sequence of transforms to apply to a program, resulting in a sketch.
2) **Resolve:** Choose values for the holes in a sketch.

The strength of an adversary depends upon how it makes these two choices. The *weakest possible adversary*, and computationally cheapest to implement, is the one that randomly chooses a sequence of transforms as well as values for sketches. The *strongest possible adversary* exhaustively tries all possible sequences of transforms and values for filling the holes in sketches, but it is intractable at best.

Our strong adversary: Our strongest adversary randomly chooses a sequence of transforms of a fixed length and then uses a *gradient-based* (i.e., *targeted*, as opposed to random) approach to fill the holes with tokens that are most *adversarial* to model performance. Specifically, we use an approach inspired by natural-language-processing techniques [23, 24].

Using a differentiable embedding layer, we take a *gradient-ascent* step in the direction that maximizes model loss. In other words, for each distinct hole in the sketch, we pick the replacement to be the token with the maximum value (in the one-hot encoding) after the gradient-ascent step. We also impose additional semantic constraints, e.g., in sketches with multiple holes, we enforce that each hole receives distinct token replacements.

B. Training Semantically Robust Models

Given an adversary, how can we train models robust to adversarial transformations?

In standard neural-network training, given a dataset of programs and labels, $(P_1, y_1), \ldots, (P_n, y_n)$, one solves an optimization objective that looks for a neural network that minimizes average prediction *loss* on the entire dataset, where the loss function $L(P, y, N)$ is a numerical measure of how bad the neural network N's prediction is on program P with label y. Formally, we solve the following problem:

$$\underset{N}{\operatorname{argmin}} \sum_i L(P_i, y_i, N) \tag{1}$$

To train robust networks, we adopt a *robust-optimization* objective [9], where we look for a neural network that minimizes average *adversarial loss*. For a program P_i, adversarial loss is the loss with respect to the semantically equivalent program P_i' produced by an adversary. In other words, the adversary is modeled in the optimization objective, forcing us to consider its behavior: whenever we compute the loss for a program P_i, we instead compute that of P_i'. Formally:

$$\underset{N}{\operatorname{argmin}} \sum_i L(P_i', y_i, N), \quad \text{where } P_i' = A(P, N) \tag{2}$$

Robust optimization has been shown to work well in image recognition and natural-language processing, and, as we shall see, results in semantically robust models for source code.

III. Framework

In this section, we explore the AVERLOC framework in greater detail. We describe (1) the transforms and resolvers AVERLOC implements, (2) the training strategies it supplies, and (3) the practical challenges of robust training on source code. We also discuss related concepts like obfuscation and non-semantics-preserving transforms (mutations).

A. Adversaries in Detail

The Transforms Library: In our framework, we provide a library of transforms on which our adversaries are built. This library consists of eight transforms and two separate implementations of these eight transforms: one implementation targeting Java programs, based on Spoon [25], and one implementation for Python, based on Astor [26]. We will use the following Java program to demonstrate our transforms:

```
public int gcd(int a, int b) {
  while (b > 0) { int c = a % b; a = b; b = c; }
  if (this.log == true) { System.out.println(a,b); }
  return a;
}
```

T1: `AddDeadCode`: A dead-code statement of the form `if (false) int ●1 = 0;`, is appended to the beginning or end of the target program. The insertion location (beginning or end) is chosen at random. Applying `AddDeadCode` to our example yields:

```
public int gcd(int a, int b) {
  if (false) { int ●1 = 0; }
  while (b > 0) { int c = a % b; a = b; b = c; }
  if (this.log == true) { System.out.println(a,b); }
  return a;
}
```

T2: `RenameLocalVariables`: A single, randomly selected, local variable declared in the target program has its name replaced by a hole. Applying `RenameLocalVariables` to our example yields:

```
public int gcd(int a, int b) {
  while (b > 0) { int ●1 = a % b; a = b; b = ●1; }
  if (this.log == true) { System.out.println(a,b); }
  return a;
}
```

T3: `RenameParameters`: A single, randomly selected, formal parameter in the target program has its name replaced by a hole. Applying `RenameParameters` to our example yields:

```
public int gcd(int ●1, int b) {
  while (b > 0) { int c = ●1 % b; ●1 = b; b = c; }
  if (this.log == true) {
    System.out.println(●1,b);
  }
  return ●1;
}
```

T4: `RenameFields`: A single, randomly selected, referenced field (`this.field` in Java, or `self.field` in Python) has its name replaced by a hole. Applying `RenameFields` to our example yields:

```
public int gcd(int a, int b) {
  while (b > 0) { int c = a % b; a = b; b = c; }
  if (this.●1 == true) { System.out.println(a,b); }
  return a;
}
```

T5: `ReplaceTrueFalse`: A single, randomly selected, Boolean literal is replaced by an equivalent expression containing a single hole. (One example: "`(●1 == ●1)`" replaces `true`.) Applying `ReplaceTrueFalse` to our example yields:

```
public int gcd(int a, int b) {
  while (b > 0) { int c = a % b; a = b; b = c; }
  if (this.log == (●1 == ●1)) {
    System.out.println(a,b);
  }
  return a;
}
```

T6: `UnrollWhiles`: A single, randomly selected, `while` loop in the target program has its loop body unrolled exactly one step. No holes are created by this transform. Applying `UnrollWhiles` to our example yields:

```
public int gcd(int a, int b) {
  while (b > 0) {
    int c = a % b; a = b; b = c;
    while (b > 0) {
      int c = a % b; a = b; b = c;
    }
    break;
  }
  if (this.log == true) { System.out.println(a,b); }
  return a;
}
```

T7: `WrapTryCatch`: The target program is wrapped by a single `try { ... } catch (...) { ... }` statement. The catch statement passes along the caught exception. A hole is used in the place of the name of the caught exception variable (e.g., `catch (Exception ●1)`). Applying `WrapTryCatch` to our example yields:

```
public int gcd(int a, int b) {
  try {
  while (b > 0) { int c = a % b; a = b; b = c; }
  if (this.log == true) { System.out.println(a,b); }
  return a;
  } catch (Exception ●1) { }
}
```

T8: `InsertPrintStatements`: A single print statement `System.out.println("●1")`, in Java, or `print('●1')` in Python, is appended to the beginning or end of the target program. The insertion location (beginning or end) is chosen at random. Applying `InsertPrintStatements` to our example yields:

```
public int gcd(int a, int b) {
  System.out.println("●1");
  while (b > 0) { int c = a % b; a = b; b = c; }
  if (this.log == true) { System.out.println(a,b); }
  return a;
}
```

Resolvers: AVERLOC provides two distinct resolution strategies (resolvers). Recall that a resolver in our framework is a method that, given a program sketch, resolves the input sketch into a complete program. For our evaluation, we implemented two resolution strategies. First, we implemented a random resolver which, given a program sketch, fills all holes in the sketch with a random token generated by selecting and concatenating a random number of sub-tokens from a (provided) fixed vocabulary. Second, we implemented the gradient-based search described in Sec. II.

B. (Robust) Optimization Objectives

Our framework enables definition of different optimization objectives for training. First, our framework can perform normal training, where the goal is to minimize a standard loss function (Eq. (1)). Second, our framework allows robust-optimization objectives (Eq. (2)) to model the adversary within the training loop.

Additionally, our framework allows for *data augmentation*, which is a standard technique where one enriches a dataset by adding random transformations of the data (e.g., rotating images using a random angle). In our framework, we can perform standard training with data augmentation by defining a completely random adversary (random choice of transforms and random resolvers) and solving, for example, the following optimization objective:

$$\underset{N}{\operatorname{argmin}} \sum_i L(P_i, y_i, N) + L(P_i', y_i, N) \quad \text{where } P_i' = A_R(P) \quad (3)$$

where A_R is a random adversary, and therefore does not take the neural network as an input. Note that the transformed set $\{P_i'\}$ is presampled before training.

Practical Challenges of Robust Optimization: Solving a robust-optimization objective is particularly challenging in the realm of source code. Practically, in every epoch of training, for every program P_i, we need to apply the adversary to compute a transformed program P_i'. This approach is wildly inefficient during training due to the *mismatch* between program formats for transformation and for training: a program P_i is an abstract syntax tree (AST) and the adversary's transformations are defined over ASTs, but the neural network expects as input a different representation—for example, sequences of (sub)tokens, or as in a recent popular line of work [27, 21], a sampled set of paths from one leaf of the AST to another.

Therefore, during training, we have to translate back and forth between ASTs and their neural representation. This approach is expensive to employ during training: in every training step, we have to apply transformations using an external program-analysis tool and convert the transformed AST to its neural representation.

We address this challenge as follows: To avoid calling program-analysis tools within the training loop, we pre-generate all possible program sketches considered by the adversary. That is, for every program P_i in the training set, before training, we generate the set of program sketches that can be produced by applying sequences of transforms of a bounded length.

C. Related Concepts

Obfuscation: One might wonder how source-code obfuscation is related to our adversaries and transforms. In general, we view obfuscation techniques as off-the-shelf adversaries that one could use in our framework. However, for the sake of a precise evaluation, we wished to build adversaries into our framework that are *tunable*, for which we can create a scoring scheme suitable for use in experiments. For our adversaries, a user can select both the resolution strategy for filling holes in program sketches and the number of transforms that may be applied, in sequence, to attack input programs. Obfuscation techniques are, in a sense, "maximal" in their efforts to mask code. For our evaluation and for ease-of-use, we focus on adversaries that can be "throttled" so that we can compare the robustness of models trained under various pipelines against progressively stronger adversaries. In summary, code obfuscators make great adversaries, and we encourage their use in our framework, but, for the sake of measured evaluation, we focus not on off-the-shelf obfuscation, but on a simpler and more configurable adversary to *control for the degree of adversarial effort* when evaluating different training pipelines.

Mutations: Given our focus on transforms that do not change the meaning of the code they target, it is natural to ask about *mutations* or non-semantics-preserving transforms. In short, mutations are *too strong*, in some sense—that is, when using mutations we have no way to *control for the degree of adversarial effort*. In fact, mutations may change the underlying meaning (semantics) of a program and, as such, the use of mutations calls into question the objective of robustness for models of code: models given equivalent (but, syntactically diverse) programs should produce equivalent outputs.

Nonetheless, one could still ask a related question: if we mutate the code, should models not be expected to change their outputs? This is an interesting question and one we hope to explore further in follow-on work. In summary, if we focus on transforms that change the meaning of code then we must change our expectations from robustness to something closer to a "mutation adequacy" score [28] for models of code. Such issues are interesting, but lie outside the scope of our investigations on robustness.

IV. Examples of existing work

In this section, we demonstrate how two recent works on adversarial robustness over code models, [24] and [29], can be implemented using AVERLOC.[1] Note that these works consider a subset of the described adversaries in Sec. III-A, but with more fine-tuned attack and defense methods. AVERLOC's adversaries are built by combining *transformations* and *resolvers*. Given transformations and resolvers, AVERLOC allows for the construction of robust training pipelines to provide *defense* against adversaries. We show how two recent works fit into the AVERLOC framework by describing their transformations, resolvers, and the defenses they provide.

[1]We also note that a preprint of our work appeared on arXiv earlier than [24] and [29].

A. DAMP

Yefet et al. [24] present a technique, Discrete Adversarial Manipulation of Programs (DAMP), to explore the adversarial robustness of code models. DAMP differs from AVERLOC in the following ways: (1) For the attack, similar to AVERLOC, DAMP uses gradient-targeted resolution to resolve program sketches. Instead of applying greedy search to find a single resolution, DAMP uses multiple trials to perform a search for the best sketch resolution. (2) For the defense, besides adversarial (robust) training, Yefet et al. [24] use outlier detection, which replaces an outlier variable name with `UNK`.

Transformation: DAMP uses transforms similar to the `RenameLocalVariables` and `AddDeadCode` transformations already provided by AVERLOC.

Resolver: To provide DAMP's multiple-trial search, after computing the gradients, instead of returning only one closest token, one would return the top-k replacement tokens, and then run the network with each token. Repeating a few such gradient-and-run steps gives us the same search procedure as in DAMP.

Defense: Besides the robust training that has already been provided by AVERLOC, one also needs to support DAMP's outlier detector. Outlier detection is an independent feature that processes and transforms the input source code. One only needs to write a function that takes in the word embeddings of the target code-snippet and computes the distances of each variable, as shown in Eq. (6) in [24]. The most distant variable would be replaced with `UNK` if the distance is above some predefined threshold.

B. Site-Selection-Perturbation (SSP)

A new formulation of an adversarial attack on code models was proposed by Srikant et al. [29]. In addition to identifying a replacement token that leads to the adversarial prediction (using a new *resolver*), they also consider *where* in the target program they should apply their resolver. (This approach is different from our current library of transforms; our transforms produce program sketches with mostly arbitrary holes—SSP optimizes *where* transforms should produce holes in the resulting program sketch.) The source code is formulated as a sequence of tokens, and the attack is formulated as an optimization problem to select which token is to be replaced/inserted with what new token. Because of the combinatorial and structural constraints of the optimization problem, SSP proposes several optimization techniques to solve the optimization problem. Because SSP does not consider defense against adversarial attacks, to implement SSP using AVERLOC, one only needs to work on the transformation and resolution components.

Transformation: SSP uses transforms similar to the following transforms from AVERLOC: `AddDeadCode`, `RenameLocalVariables`, `RenameParameters`, `RenameFields`, `ReplaceTrueFalse`, and `InsertPrintStatements`. However, SSP "upgrades" these transforms to produce a program sketch that has *optimized holes*—that is, SSP optimizes *where* transforms change the input program.

Resolver: SSP implements a resolver that is similar to AVERLOC's gradient-targeted resolver. However, SSP performs a joint (alternating) optimization to find both optimal sites for applying transforms and optimal resolutions for the sketches generated by transforms. To phrase this using the terminology of AVERLOC: SSP iterates their transforms and resolver as part of an alternating optimization routine.

V. Evaluation

In this section, we provide answers to five research questions, with the goal of understanding the adversaries our framework allows, their attack strength, defenses one can build via our framework, and downstream consequences for practitioners (such as performance of both normal and robust models on the domain-adaptation and cross-language-transfer tasks). To conduct our experiments, we utilize four datasets in two different languages (Java and Python): c2s/java-small, csn/java, csn/python and sri/py150. Each dataset contains around 0.5M data points (method-body/method-name pairs). Running extensive experiments across the 4 datasets and 2 (code-summarization) models (seq2seq, code2seq) was computationally intractable, both in terms of time and space. Thus, we randomly subsample the four datasets to have train/validation/test sets of sizes 150k/10k/20k each. The datasets remain large, and we find that subsampling has a minimal effect on model performance. Furthermore, the reduced dataset sizes allowed us to perform over 160 evaluations of over 32 distinct trained models, which, even with subsampled datasets, required hundreds of hours of GPU compute time and was expensive to perform. Unless otherwise noted, we measure (changes) in our models' F1 scores. F1 is a metric that computes the harmonic mean of a model's precision and recall.

A. RQ1: Attack Strength

AVERLOC's generic adversary is built on a library of transformations and resolvers; how effective are individual transforms under both resolution strategies (random/gradient)?

Rationale: One of the reasons we use a generic adversary based on a library of transformations and resolvers is to *control for the degree of adversarial effort* and, in doing so, allow for precise experimentation. To do this effectively, we must understand how each of our eight transforms and two resolution strategies perform against our models. Furthermore, in this evaluation we also get our first insights into the effect of having a model mostly trained on *syntax* (seq2seq) versus a model trained primarily on *structure* (code2seq).

Metrics: We measured the *drop in F1 score* of each of our models under each combination of transformation and resolver. This drop in F1 was computed by assessing the baseline performance of a given model on its original test set and then measuring that same model against an *attacked* version of its test set. To simplify the presentation, we show data from both model architectures evaluated on a single dataset.

TABLE I: Decreases in F1 induced by each of our eight semantics-preserving transformations paired with either random (R) or gradient-based (G) resolution strategies (measured against a normally trained baseline model on the c2s/java-small dataset). (Larger numbers indicate stronger attacks.)

Transform	seq2seq	code2seq
	$-\Delta$F1 (R / G)	$-\Delta$F1 (R / G)
`AddDeadCode`	4.0 / 7.7	1.4 / 2.9
`RenameParameter`	0.3 / 3.0	0.3 / 4.7
`InsertPrintStatement`	2.7 / 6.1	3.8 / 10.2
`ReplaceTrueFalse`	0.0 / 0.7	0.2 / 0.5
`RenameField`	2.3 / 5.4	2.0 / 2.0
`UnrollWhile`	0.0 / 0.0	0.4 / 0.4
`RenameLocalVariable`	0.3 / 2.2	0.0 / 2.5
`WrapTryCatch`	2.5 / 9.4	1.4 / 7.8

Results: Table I shows a breakdown of our results. Note that, across model architectures, the `AddDeadCode`, `InsertPrintStatement`, and `WrapTryCatch` transforms are particularly effective. It is also of interest to note that *gradient resolution is strictly better than random resolution* (except for `UnrollWhiles`, which produces a program sketch with no holes—thus, resolution has no effect). Finally, note that code2seq is, in many cases, more robust, but also, in notable cases, more susceptible to attack. In particular, we find that the `InsertPrintStatement` transform (with gradient-based resolution) is over 1.5 times as effective on code2seq as on seq2seq.

RQ1 Summary. We find that our individual attacks are effective and, between random and gradient-based resolution, find gradient-based resolution to be strictly better. Furthermore, we find a surprising fact: although code2seq is a naturally more robust architecture, it has some surprising weakness—allowing for up to 1.5x more effective attacks than a simpler seq2seq baseline, in some cases.

B. RQ2: Robust Training versus Baselines

How effective is robust training in *defending* against the (single-step) attacks we just examined? Are there any alternative approaches (like dataset augmentation) that perform well?

Rationale: Finding ways to train robust models of code is our primary goal; therefore, in this question, we seek to understand exactly which pieces of our framework are most useful in our quest to train robust models. As part of this evaluation, we test increasingly complex (and costly) training pipelines, with the hope that more sophisticated instantiations of our framework create more robust models.

Metrics: We trained models using three different training pipelines: (1) training with dataset augmentation, (2) robust training with an adversary configured to use any of our eight transforms and random resolution, and (3) robust training with an adversary configured to use any of our eight transforms and gradient-targeted resolution. We measured the *change in F1 score* of each of these models when attacked by an adversary using any of our eight transforms and gradient-targeted resolution. (This change is relative to a normally-trained baseline model attacked by the same adversary.)

TABLE II: Raw F1 and change in F1 (in square brackets) for models trained using three different training pipelines in AVERLOC. The first four rows show results for seq2seq while the last four rows show results for code2seq. (Larger numbers are better.)

Training	c2s/java-small	csn/java	csn/python	sri/py150
Normal	23.3	17.2	16.2	22.0
Augmented	27.8 [+4.4]	21.4 [+4.3]	20.9 [+4.7]	24.0 [+2.0]
Robust (R)	27.5 [+4.2]	30.1 [+12.9]	29.8 [+13.5]	33.4 [+11.4]
Robust (G)	**32.0 [+8.7]**	**32.8 [+15.6]**	**32.2 [+16.0]**	**37.1 [+15.1]**
Normal	24.6	19.6	21.0	23.7
Augmented	23.4 [-1.2]	19.7 [+0.1]	20.8 [-0.2]	24.4 [+0.7]
Robust (R)	28.1 [+3.5]	23.5 [+3.9]	22.5 [+1.6]	26.6 [+2.9]
Robust (G)	**31.6 [+7.0]**	**27.5 [+7.9]**	**23.8 [+2.8]**	**29.5 [+5.8]**

Results: Table II shows the results. In general, we find that robust training using an adversary with gradient-targeted resolution is, by far, the best defense we can provide. Furthermore, we find that dataset augmentation pales in comparison to true robust training. Of particular interest is the relationship between seq2seq, code2seq, and robustness. On average, seq2seq (Normal) fares worse than code2seq (Normal) under our attack (see the first and fifth rows of Table II). But, this story changes when robust training is applied: *it is harder to make code2seq robust*. After robust training, we find that "seq2seq (Robust (G))" ends up performing better under our attack (see the fourth and eighth rows of Table II). This result was quite surprising and may be worth further study: models that are better in normal circumstances may (1) have surprising weaknesses, and (2) be harder to make robust.

RQ2 Summary. We find that either form of robust training was better than dataset augmentation. Furthermore, in all tested configurations, across all models and datasets, robust training with respect to an adversary using gradient-targeted resolution gave the best defense. Finally, we find that, to our great surprise, code2seq is *harder to make robust* than a seq2seq model.

C. RQ3: Stronger Adversaries

Does training with a "single-step" adversary improve robustness against stronger adversaries? In particular, if we train with an adversary that picks just a single (random) transform and uses gradient-targeted resolution, how well does the model perform against an adversary that is allowed to apply a *sequence of five* random transforms (also using the stronger, gradient-targeted, resolution)?

Rationale: In this question, we seek to understand if training with a weak adversary is "good enough"—if this is the case, then practitioners may save effort by performing robust training against weaker (and less computationally expensive) adversaries while still retaining robustness against stronger threats.

Metrics: We compared the *decrease in F1* of a normally trained model and a model trained with robust-training (using an adversary that may select any single transform from our library and resolve it via gradient-targeted resolution). Here the drop in F1 was measured against a sequence of progressively stronger attacks: Nor (normal: no attack), R1 ("single-step" adversary

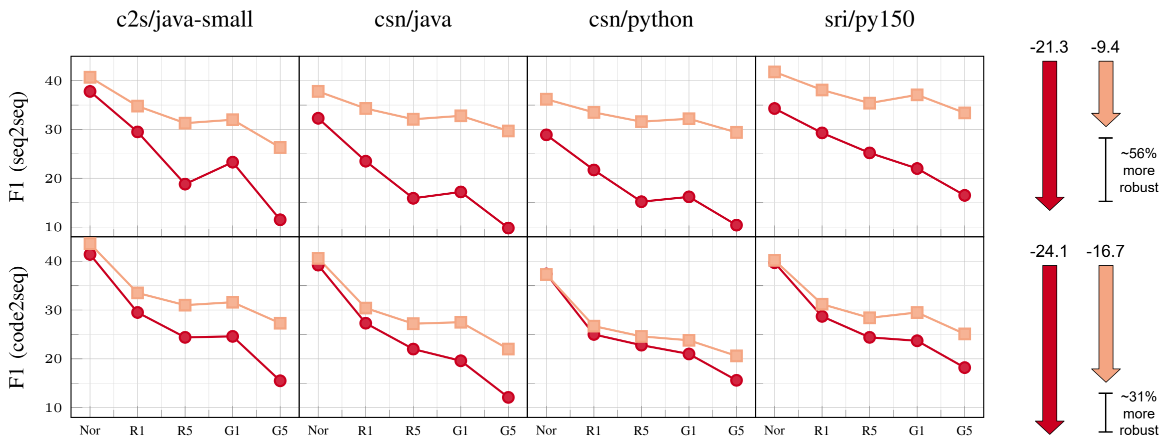

Fig. 2: A comparison of a normally trained (Normal, ●) model and a robustly trained model against a "single-step" gradient-targeted adversary (Robust, ■). These plots show F1 scores across each of four datasets and two model architectures (seq2seq and code2seq) under a sequence of progressively stronger attacks. To the right, average decrease in F1 is shown for both Normal/Robust models in both seq2seq and code2seq architectures. (Higher F1 scores are better.)

with random resolution), R5 (an adversary allowed sequences of five random transforms, still with random resolution), G1 ("single-step" adversary with gradient-targeted resolution), and G5 (an adversary allowed sequences of five random transforms using gradient-targeted resolution). We plotted the raw F1 scores of both our normal and robust models against these progressively stronger attacks to give a visual depiction of the robustness of each model.

Results: Figure 2 presents F1 scores from 80 distinct evaluations we performed across our four datasets, two model architectures, two training methods, and four (progressively stronger) adversaries.

There are several interesting things to learn from Fig. 2; first, notice that, across models, languages, and training methodology, stronger attacks induce larger drops in F1; however, as one might hope, robustly trained models (Robust: square markers) lose less F1 to a stronger adversary compared with normally trained models (Normal: circular markers). On the right side of Fig. 2, we make this explicit by noting that, on average, the robustly trained models retain more of their original F1 on progressively stronger attacks. Specifically, for seq2seq, the robustly trained model retains 56% more of its original performance than the normal model. Similarly, for code2seq, the robustly trained model retains 31% more of its original performance. These results are encouraging because we are attacking with a *much stronger adversary* than the one we used as part of robust training. Furthermore, the ability to carry out these measured evaluations is one of the key contributions of our framework: we can precisely control for the power of our adversaries by tuning the transforms, the allowed sequence length, and the resolution algorithm.

Finally, we note one last surprising result (that echoes what we observe in RQ2): code2seq is *more difficult to make robust*. This phenomenon is clearly visible in the fact that robust training has less of an effect on code2seq (31% increase in robustness, on average, under the strongest attack) compared to seq2seq (56% increase in robustness, on average, under the strongest attack). This result continues to surprise us, and is an important takeaway for practitioners of ML-on-code: we still have much to learn about how increasingly sophisticated models fare against adversarial attacks and, in our data, we find that more sophisticated models both have (1) surprising weaknesses and (2) naturally better robustness but, paradoxically, are less amenable to techniques for increasing robustness.

We also performed an *additional* 80 evaluations on models generated via training with dataset augmentation and robust training with a (weaker still) single-step adversary using random resolution (the same training pipelines we evaluated in RQ2). As one would expect, these sit directly between the normal model (least robust) and the robustly trained model (most robust).

RQ3 Summary. We find that training against weaker attacks is sufficient to provide a defense against increasingly stronger attacks. Furthermore, through a series of 160 evaluations, we find confirmation of our earlier results, including the surprising fact that code2seq is less amenable to robust training than seq2seq model. Finally, we find confirmation that robust training, aside from the nuances discussed, is an effective technique across model architectures, programming languages, and datasets.

D. *RQ4: Domain Adaptation*

What is the effect of robust training on the performance of models for the domain-adaptation task? For example, imagine you train a model on the code of one large company and, later, you wish to use that same model on code from another organization—will the model retain its original performance? What about a robustly trained model, will it perform better or worse?

Rationale: In this question, we seek to understand how both normal models of code and robustly-trained models of code *adapt* to unseen data. This data is different than simple "test-set" data because we have gone to great lengths to collect data, in both Python and Java, from different original sources that used different collection methodology. Therefore, when we apply a model trained on one Java dataset to our other Java dataset, we are getting a glimpse into how that model may perform on code that is "different" than what it has already seen. We are not the first to study robustness and domain adaptation

TABLE III: F1 across both normally trained and robustly trained models on *out-of-distribution* test sets (sourced from different original data sources using differing collection methodologies). (Higher F1 scores are better.)

Model	Trained On	Tested On	F1 (seq2seq)	F1 (code2seq)
Normal	c2s/java-small	csn/java	29.35	33.31
Robust	c2s/java-small	csn/java	**33.05**	**35.76**
Normal	csn/java	c2s/java-small	34.85	39.74
Robust	csn/java	c2s/java-small	**38.40**	**41.12**
Normal	sri/py150	csn/python	19.97	34.40
Robust	sri/py150	csn/python	**31.93**	**35.00**
Normal	csn/python	sri/py150	25.15	**23.31**
Robust	csn/python	sri/py150	**27.74**	23.20

[30], however, to the best of our knowledge, we are the first to present such results in the space of models on code.

Metrics: Again, we measured F1 scores for our models. This time, we compared models trained on one of our datasets using either normal or robust training and their performance on a second dataset *from a different original source*.

Results: Table III presents results for both model architectures under both normal and robust training pipelines. Each row shows a single model (trained with either normal or robust training), the dataset it was trained on, the dataset it was tested on (originating from a source distinct from the training data), and the F1 scores produced by both of the seq2seq and code2seq model architectures. In general, we find confirmation that *robust training improves performance on the domain-adaptation task*. This result is a useful fact for practitioners: not only does robust training strengthen your model against attack, it also provides benefits in terms of generalization. Similar to our previous research questions, we again see that code2seq benefits *less* from robust training than seq2seq does.

RQ4 Summary. We find strong evidence across our four datasets and two model architectures in support of robust training improving performance on the domain-adaptation task. To the best of our knowledge, we are the first to report such an effect in the space of models for code.

E. RQ5: Cross-Language Transfer

What is the effect of robust training on the performance of models for the cross-language-transfer task? Does robustness play a role in how models of code may perform on unseen languages?

Rationale: It seemed natural, after investigating domain adaptation, to also investigate cross-language transfer. One may hope that good models of code are naturally able to work across different programming languages and, therefore, it would be useful to understand the relationship between robustness and cross-language transfer.

Metrics: We measured F1 scores, for our seq2seq models, under both normal and robust training. We trained on data from one language (either Java or Python) and tested on data from the opposite language. We focus on seq2seq for this evaluation because code2seq cannot be trained on one language and (directly) applied to another. This test data both comes from an unseen dataset, and is in a language the model has never seen.

TABLE IV: F1, for our seq2seq model, across both normally trained and robustly trained models on the Java to Python *cross-language* transfer task. (Higher F1 scores are better.)

Model	Trained On	Tested On	F1 (seq2seq)
Normal	c2s/java-small	csn/python	19.36
Robust	c2s/java-small	csn/python	**22.21 [+2.86]**
Normal	csn/java	csn/python	**22.15**
Robust	csn/java	csn/python	16.38 [-5.78]
Normal	c2s/java-small	sri/py150	21.94
Robust	c2s/java-small	sri/py150	**22.73 [+0.80]**
Normal	csn/java	sri/py150	**23.79**
Robust	csn/java	sri/py150	14.54 [-9.25]

TABLE V: F1, for our seq2seq model, across both normally trained and robustly trained models on the Python to Java *cross-language* transfer task. (Higher F1 scores are better.)

Model	Trained On	Tested On	F1 (seq2seq)
Normal	csn/python	c2s/java-small	25.23
Robust	csn/python	c2s/java-small	**32.13 [+6.90]**
Normal	csn/python	csn/java	17.39
Robust	csn/python	csn/java	**24.02 [+6.63]**
Normal	sri/py150	c2s/java-small	24.71
Robust	sri/py150	c2s/java-small	**26.59 [+1.88]**
Normal	sri/py150	csn/java	15.06
Robust	sri/py150	csn/java	**16.80 [+1.74]**

Results: Tables IV and V show results for both Java-to-Python and Python-to-Java cross-language transfer. We were surprised to find that, in the case of transfer performance for models trained on Java and evaluated on Python, robust training had a clear *negative* effect—that is, normally trained models retained more of their performance on the unseen Python test sets. But, again to our surprise, we found a stronger *positive* effect for robustly trained models trained on Python and evaluated on Java. This situation is somewhat perplexing: one might hope that either robust training always improves cross-language transfer, or never does. In general, the data we collected warrants further study of the interplay between robustness and a model of code's ability to transfer across languages.

RQ5 Summary. We found robust training to have unclear effects on cross-language model transfer. In the case of training on Java and applying the learned models to Python, robust training had a *negative* effect (dropping F1, on average, 3 points); but, in the opposite task of training on Python and evaluating on Java, we found robust training to have a stronger *positive* effect (increasing F1, on average, 4 points).

VI. Related Work

In concurrent work,[2] Bielik and Vechev [31] combine adversarial training with abstention and AST pruning to train robust models of code. There are a number of key differences with our work: (1) We consider a richer space of transformations

[2]A preprint of our work appeared earlier on arXiv than [31].

for the adversary, including inserting parameterized dead-code. (2) We use a strong gradient-based adversary and program sketches for completing transformations, while they use a greedy search through the space of transformations with a small number of candidates. (3) Our adversarial-training approach is more efficient, because it does not solve an expensive ILP problem to prune ASTs or train multiple models, but it is possible that we can incorporate their AST pruning in our framework.

A. Adversarial Examples

In test-time attacks, an adversary perturbs an example so that it is misclassified by a model (untargeted attack) or the perturbed example is classified as an attacker-specified label (targeted) [32, 2, 33, 34, 8]. Initially, test-time attacks were explored in the context of images. Our discrete domain is closer to test-time attacks in natural language processing (NLP). There are several test-time attacks in NLP that consider discrete transformations, such as substituting words or introducing typos [35, 36, 23, 37, 38]. A key difference between our domain and NLP is that in the case of programs one has to worry about semantics—the program has to work even after transformations.

Recently, more consideration has been given to adversarial examples in the software-engineering domain. Rabin et al. [39] consider semantics-preserving transforms and their effects on various neural program analyzers (including code2seq). Compton et al. [40] consider adversarial examples based primarily on variable renaming; they create more robust models via training with dataset augmentation. As we show experimentally, dataset augmentation does not result in robust models compared to training based on gradient-based optimization.

Many ideas from software testing, such as fuzzing and search-based techniques, have recently been successfully applied to discovering adversarial examples and other forms of bugs in neural networks [41, 42, 43, 44]. These approaches can be used to generate examples for data augmentation; however, they are generally too heavyweight to incorporate within training.

B. Deep Learning for Source Code

Recent years have seen huge progress in deep learning for source-code tasks—see Allamanis et al. [45]. In this paper, we evaluate two popular models for learning from source code: seq2seq [46] and code2seq [13]. The seq2seq model (sub-)tokenizes the program, analogous to NLP, and uses a variant of recurrent neural networks to generate predictions. This idea has appeared in numerous papers, e.g., the pioneering work of Raychev et al. [47] for code completion. We also evaluate code2seq, which uses an AST-paths encoding pioneered by Alon et al. [27]. Researchers have considered more structured networks, like graph neural networks [12] and tree-LSTMs [48]. These would be interesting to consider for future experimentation in the context of adversarial training. The task we evaluated on, code summarization, was first introduced by Allamanis et al. [12].

VII. Threats to Validity

There are several threats to the validity of our approach. First, we make use of a limited number of datasets and models. We attempted to diversify both the datasets we used (by choosing four sets from three unique sources in two languages) and the models (by picking two distinct model architectures). Nonetheless, there remains the possibility that our results may not generalize (in particular, to tasks outside the code-summarization task we chose to study). We also made the choice to subsample our data; although we did not observe a large impact to trained-model performance, it is still possible that subsampling had an impact on our results. Additionally, we make use of a limited set of transformations. We took care to implement many transformations, and through the sequencing of many transformations, we attempted to further increase the power of our adversary. Nevertheless, it is possible that stronger adversaries exist. Finally, our toolchain is complex and it is possible that there are bugs in the implementation. To guard against this, we have manually inspected the data in various representations, at various points in our pipeline, to spot-check our framework.

VIII. Conclusion

AVERLOC is a generic framework that has (already) allowed others to push the boundaries of adversarial machine learning for models of code. Through extensive evaluation, we have demonstrated the efficacy of our framework and learned several surprising facts. Namely, we found that a state-of-the-art architecture (code2seq) is harder to make robust than a simpler (seq2seq) baseline (Table II); additionally, we found code2seq to have surprising weaknesses compared to our simpler baseline model (Table I); finally, we found that robust models perform better against unseen data—however, robust models are not clearly better at the cross-language-transfer task (Tables III, IV and V). To the best of our knowledge, we present the first results on the interplay between robust models of code and the domain-adaptation and cross-language-transfer tasks. Finally, we recommend that those seeking to build new models of code consider the effects of semantics-preserving transformations and explore robust training. To this end we have already made public the code, data, and models we use in this evaluation; these resources and corresponding documentation can be viewed here: [49].

Acknowledgments

Supported, in part, by a gift from Rajiv and Ritu Batra; by Facebook under a Probability and Programming Research Award; by NSF grants CCF-{1704117, 1918211, 1652140, 2046710, FMitF-1836978} and SaTC-Frontiers-1804648; by ARO grant number W911NF-17-1-0405; by DARPA-GARD problem under agreement number 885000; by ONR under grants N00014-17-1-2889 and N00014-19-1-2318; and by a Microsoft Research PhD Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

References

[1] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[2] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, "Evasion attacks against machine learning at test time," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2013, pp. 387–402.

[3] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[4] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 506–519.

[5] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning visual classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 1625–1634.

[6] A. N. Bhagoji, W. He, B. Li, and D. Song, "Practical black-box attacks on deep neural networks using efficient query mechanisms," in *European Conference on Computer Vision*. Springer, 2018, pp. 158–174.

[7] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter, "Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1528–1540.

[8] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 39–57.

[9] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJzIBfZAb

[10] Z. Kolter and A. Madry, "Ibm/pytorch-seq2seq," Feb 2020. [Online]. Available: https://adversarial-ml-tutorial.org

[11] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 91–105. [Online]. Available: https://doi.org/10.1145/3385412.3385997

[12] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001

[13] https://github.com/tech-srl/code2seq.

[14] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 152–162.

[15] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 141–151.

[16] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," 2020.

[17] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, "Neural program repair by jointly learning to localize and repair," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=ByloJ20qtm

[18] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "Typewriter: Neural type prediction with search-based validation," 2020.

[19] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," 2020.

[20] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to error-handling specification mining," in *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), FSE 2018, Lake Buena Vista, Florida, November 4 - 9, 2018*, 2018.

[21] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[22] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006, pp. 404–415.

[23] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "Hotflip: White-box adversarial examples for text classification," 2017.

[24] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428230

[25] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document

[26] Berkerpeksag, "berkerpeksag/astor," Jan 2020. [Online]. Available: https://github.com/berkerpeksag/astor

[27] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[28] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 402–411. [Online]. Available: https://doi.org/10.1145/1062455.1062530

[29] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U.-M. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=PH5PH9ZO_4

[30] R. Volpi, H. Namkoong, O. Sener, J. C. Duchi, V. Murino, and S. Savarese, "Generalizing to unseen domains via adversarial data augmentation," in *NeurIPS*, 2018.

[31] P. Bielik and M. Vechev, "Adversarial robustness for code," 2020.

[32] A. Athalye, N. Carlini, and D. Wagner, "Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples," *arXiv preprint arXiv:1802.00420*, 2018.

[33] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin, "Black-box adversarial attacks with limited queries and information," *arXiv preprint arXiv:1804.08598*, 2018.

[34] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh, "Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. ACM, 2017, pp. 15–26.

[35] Q. Lei, L. Wu, P.-Y. Chen, A. G. Dimakis, I. S. Dhillon, and M. Witbrock, "Discrete adversarial attacks and submodular optimization with applications to text classification," in *SysML*, 2019.

[36] P. K. Mudrakarta, A. Taly, M. Sundararajan, and K. Dhamdhere, "Did the model understand the question?" in *ACL*, 2018.

[37] W. E. Zhang, Q. Z. Sheng, A. Alhazmi, and C. LI, "Adversarial attacks on deep learning models in natural language processing: A survey," *arXiv preprint arXiv:1901.06796*, 2019.

[38] S. Garg and G. Ramakrishnan, "Bae: Bert-based adversarial examples for text classification," 2020.

[39] M. Rabin, R. Islam, N. D. Bui, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of neural program analyzers with respect to semantic-preserving program transformations," *arXiv preprint arXiv:2008.01566*, 2020.

[40] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," 2020.

[41] Y. Tian, Z. Zhong, V. Ordonez, G. Kaiser, and B. Ray, "Testing dnn image classifiers for confusion & bias errors," 2020.

[42] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks," 2020.

[43] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 303–314. [Online]. Available: https://doi.org/10.1145/3180155.3180220

[44] F. Zhang, S. P. Chowdhury, and M. Christakis, "Deepsearch: A simple and effective blackbox attack for deep neural networks," 2020.

[45] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[46] IBM, "Ibm/pytorch-seq2seq," Jan 2020. [Online]. Available: https://github.com/IBM/pytorch-seq2seq

[47] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

[48] J. Zhao, A. Albarghouthi, V. Rastogi, S. Jha, and D. Octeau, "Neural-augmented static analysis of android communication," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 342–353.

[49] FrameworkURL, "Averloc framework for robust training," Feb 2020. [Online]. Available: https://github.com/jjhenkel/averloc