Energy-bounded Learning for Robust Models of Code

Nghi D. Q. Bui
Trustworthiness Lab
Huawei Ireland Research Centre
nghi.bui@huawei.com

Yijun Yu
Trustworthiness Lab
Huawei Ireland Research Centre
yijun.yu@huawei.com

Abstract
In programming, learning code representations has a variety of applications, including code classification, code search, comment generation, bug prediction, and so on. Various representations of code in terms of tokens, syntax trees, dependency graphs, code navigation paths, or a combination of their variants have been proposed, however, existing vanilla learning techniques have a major limitation in robustness, i.e., it is easy for the models to make incorrect predictions when the inputs are altered in a subtle way. To enhance the robustness, existing approaches focus on recognizing adversarial samples rather than on the valid samples that fall outside a given distribution, which we refer to as out-of-distribution (OOD) samples. Recognizing such OOD samples is the novel problem investigated in this paper. To this end, we propose to first augment the in-distribution datasets with out-of-distribution samples such that, when trained together, they will enhance the model’s robustness. We propose the use of an energy-bounded learning objective function to assign a higher score to in-distribution samples and a lower score to out-of-distribution samples in order to incorporate such out-of-distribution samples into the training process of source code models. In terms of OOD detection and adversarial samples detection, our evaluation results demonstrate a greater robustness for existing source code models to become more accurate at recognizing OOD data while being more resistant to adversarial attacks at the same time. Furthermore, the proposed energy-bounded score outperforms all existing OOD detection scores by a large margin, including the softmax confidence score, the Mahalanobis score, and ODIN.

1 Introduction
Learning code representations (a.k.a. embeddings) and developing a prediction model for programs have been found to be beneficial for a variety of programming tasks, including program functionality classification [13, 46], code search [20, 29, 54], code comments generation [4, 27, 59], bug prediction [34, 67], program translation [11, 21], etc.

While the performance of source code models continues to improve, the real world is open and full of unknowns, presenting significant challenges for such learning models to handle diverse inputs reliably. Existing source code models suffer from two kinds of robustness problems: (1) adversarial robustness: small, seemingly innocuous perturbations to the input that lead to incorrect predictions [7, 50, 63]; (2) out-of-distribution (OOD) uncertainty arises when a machine learning model sees an input that differs from its training data, and while still predicting on an existing class label by the model regardless. To the best of our knowledge, adversarial robustness for code has been studied recently [7, 50, 52, 53, 62, 63], while the OOD uncertainty detection has been neglected in the research on the robustness of code models.

In this work, we seek to address the OOD robustness problem by proposing a novel learning method that brings two benefits together: (1) making source code models more resilient to adversarial samples; (2) enabling the detection of OOD samples.

We observe that current learning models only capture very specific patterns from limited training datasets, i.e., the datasets are classified by a given set of labels, and the models can only know what is known from the datasets to make known predictions. In other words, they learn known-knowns and unknown-knowns, rather than known-unknowns and unknown-unknowns.

In general, when training and testing data are drawn from the same distribution, modern neural networks are known to generalize effectively. However, when the neural networks are used in real-world applications, there is often limited control over the distribution of testing data. The training data may not contain sufficient number of classes beforehand. Regardless of the size of the training dataset, therefore, the model cannot encompass all possible labels, which may lead to an issue that the model may simply output a label based on the value of the softmax layer for classification, even though the testing sample does not belong to any of the labelled classes.

To tackle this problem, we aim to enable source code models to say "I don’t know" whenever possible instead of making a blind prediction by pretending that they knew the answer. For example, POJ104 [43] is a well-known dataset
for code classification task that contains 52000 samples and 104 classes. Many code learning models have been trained on this dataset, including Tree-based CNN [43], ASTNN [64], TreeCaps [9], etc. However, when one deploys any of these models to real-world usage, a new/unseen program must still be classified into one of the 104 classes, which does not hold. Ideally, the code classification model should be able to tell if the new code snippet does not really belong to any of these classes.

Apart from code classification, a significant variety of programming tasks can be formulated as classification problems. For instance, Alon et al. [5] formulates the method name prediction task as an extreme classification problem. Hoang et al. [26] develops a tool for the patch identification task which can also be formulated as a classification problem. Numerous studies on bug detection have also been conducted as classification tasks [40, 45, 47, 65]. Given that OOD is a general problems for classification tasks, all of these programming tasks will suffer from it and might expose risks when deployed in real-world usages. Therefore, it is our belief that determining where inputs are OOD is essential for source code model in real-world applications.

To overcome the above-mentioned limitation in learning methods, we propose to use an auxiliary dataset in addition to the main dataset to train for a specific code learning task. The auxiliary dataset is used as an external resource to improve the model’s prediction capability, i.e., the model will know when not to predict something outside of its knowledge. In the case of code learning, a collection of code snippets from any code hosting repository, such as Github, can be used as the auxiliary dataset. As a result, the auxiliary dataset has the benefit of being inexpensive to collect because one does not need to label its elements, as we will demonstrate later in technical details. Now, we aim to train the model to generate a single scalar value as the measurement score, so that in-distribution samples have high scores and out-of-distribution samples have low scores.

A softmax confidence score has been used in prior work for OOD detection in vision learning and natural language processing to safeguard against OOD inputs [33, 37], where OOD is defined as an input with a low softmax confidence score. However, neural networks can yield arbitrarily high softmax confidence for inputs that are far away from the training data. On the other hand, energy-based models have been shown to improve image synthesis calibration, robustness, and OOD detection [19]. Inspired by the pioneer work, we propose an energy-bounded objective function for the auxiliary dataset that will be jointly trained with the cross-entropy objective function for the main dataset. The additional knowledge from the auxiliary dataset will be leveraged to the model during this jointly training phase by pushing the OOD samples farther away, in distance, from the in-distribution samples.

Since our energy-bounded learning method is agnostic to the cross-entropy objective functions, it is applicable to a wide range of source code models and a several programming tasks. In this work, we evaluated the method on the code representations such as Tree-based CNN, TreeCaps, Code2Vec, Bi-LSTM, and Transformer, with programming tasks including code classification and method name prediction. The results show that training the source code model with an auxiliary dataset and using an energy-bounded objective improves the model’s robustness in terms of both adversarial and OOD detection robustness.

In summary, our contributions are listed as follows.

- We are the first to explore a novel research direction to detect OOD samples to enhance the robustness of source code models;
- We propose to use the energy as the score to detect OOD samples at inference time. We show that the energy score outperforms the other scores, such as softmax confidence score [25], the Mahalanobis score [33], and ODIN [37] on novel OOD evaluation benchmarks of source code models.
- We propose a novel energy-bounded loss function as an alternative to the cross-entropy loss function at training time for source code models. The source code models that employ this loss are trained on an auxiliary dataset (out-distribution) in addition to the main dataset (in-distribution), which is intended to improve the models’ robustness. Our results demonstrate that using an energy-bounded loss function significantly improves the robustness of source code models in terms of adversarial robustness and OOD detection.
- We leverage several semantics-equivalent program transformations as adversarial attacks because a weak code model could regard such transformed programs as being a new class of program. Therefore, we use these transformations to test the adversarial robustness of the energy-bound enhanced learning against subtle changes in the input code on the percentage of predictions changed metric;
- Using code classification and method name prediction tasks, our evaluations demonstrate that using an energy-bounded loss function and an auxiliary dataset increases the capacity of big code models to discriminate between in- and out-of-distribution samples while retaining classification performance.
- We release our code on the anonymous repository https://github.com/pldi22/energy_bounded and the auxiliary dataset for OOD benchmark.

The structure of the remainder of the paper is organized as follows. Section 2 presents related work, Section 3 introduces background concepts necessary for our approach, which is detailed in Section 4. Section 5 presents the evaluation of our
approach, and Section 6 analyses the case study for threats to validity, with a discussion of future work in Section 7. Finally, we conclude in Section 8.

2 Related Work

Code Representation Learning. There has been a huge interest in applying deep learning techniques to programming tasks such as program functionality classification [43, 64], bug localization [22, 48], function name prediction [18], code clone detection [64], program refactoring [27], program translation [11], and code synthesis [8]. Allamanis et al. [1] extend ASTs to graphs by adding a variety of code dependencies as edges among the tree nodes, intended to represent code semantics, and apply Gated Graph Neural Networks (GGNN) [36] to learn the graphs from code; Code2vec [4], Code2seq [3], and ASTNN [64] are designed based on splitting ASTs into smaller ones, either as a bag of path-contexts or as flattened subtrees representing individual statements. They use various kinds of Recurrent Neural Networks (RNNs) to learn such code representations.

Robustness of Neural Networks. A considerable body of research has been conducted on the robustness of artificial intelligence (AI) systems in general, and deep neural networks in particular. Szegedy et al. [57] is the first to discover that deep neural networks are vulnerable to small perturbations that are imperceptible to human eyes. Many following up works [10, 17, 31, 42] further demonstrated the severity of the robustness issues with a variety of attacking methods. While aforementioned approaches only apply to models for image classification, new attacks have been proposed that target models in other domains, such as natural language processing [28, 35, 66] and graphs [14, 68]. In addition, a new line of research about methods to ensure the robustness of source code models has been proposed [7, 53, 62, 63]. Aside from these methods, Rabin et al. [51] proposed a method to leverage program transformation to measure the generalizability of source code models.

OOD Detection. Significant progress has recently been made in identifying OODs and training models that are robust to such examples (defenses). However, the majority of the research has targeted computer-vision and natural language processing tasks [16, 25, 33, 37]. The softmax confidence score has become a common baseline for OOD detection [25]. Many following up methods propose different scores to detect the OOD samples, such as the Mahalanobis score [33], adding a temperature scaling factor for softmax score [37], etc. DeVries and Taylor [16] propose to learn the confidence score by attaching an auxiliary branch to a pre-trained classifier and deriving an OOD score. While most of these methods have been proposed for either vision learning or natural language processing, no previous work has been found to know whether these methods are applicable to code representation learning, nor knowing whether they require any adaptation.

3 Background

In this section, we present the background concepts to prepare for the understanding of our solution in Section 4.

3.1 Code Representation Learning

The method of learning source code representations typically consists of two stages: (1) converting a code snippet to an intermediate representation (IR), such as token streams, ASTs, AST paths, or graphs; and (2) designing a neural network capable of processing such intermediate representations.

This type of neural network is also referred to as an encoder because it takes the input code IR and maps it to a code vector embedding $\tilde{u}$, typically a combination of various types of code elements, then feed the embedding vector into the next layer(s) of training for an objective function associated with the learning system’s specific task.

Furthermore, cross-entropy is a loss function that is mostly used across the techniques to model source code ($\approx 20\%$) [61]. In our work, we propose an energy-bounded objective as an alternative to the cross-entropy loss function in order to enhance the robustness of source code models.

3.2 Energy-based Models

The essence of the energy-based model (EBM) [32] is based on the construction of a function $E(x): \mathbb{R}^D \rightarrow \mathbb{R}$ that maps each point $x$ in the input space to a single, non-probabilistic scalar called the energy. A collection of energy values could be turned into a probability density $p(x)$ through the Gibbs distribution:

$$p(y | x) = \frac{e^{-E(x, y)/T}}{\int_{y'} e^{-E(x, y')/T}} = \frac{e^{-E(x, y)/T}}{e^{-E(x)/T}},$$

where the denominator $\int_{y'} e^{-E(x, y')/T}$ is called the partition function, which marginalizes over $y$, and $T$ is the temperature parameter.

The energy $E(x)$ of a given data point $x \in \mathbb{R}^D$ can be expressed as the negative logarithm of the partition function:

$$E(x) = -T \cdot \log \int_{y'} e^{-E(x, y')/T} \; (2)$$

3.3 OOD detection

OOD detection is the task of deciding whether or not a sample is from a learned distribution $D_{in}$. Samples from $D_{in}$ are called in-distribution, and otherwise are said to be out-of-distribution (OOD) from $D_{out}$. Therefore, OOD detection is a binary classification problem that relies on a score to differentiate between in- and out-of-distribution samples. We call the classifier as the out-of-distribution detector (OODD).
Out-of-distribution Detector. An out-of-distribution detector should produce values that are distinguishable between in- and out-of-distribution. For each code sample $x$, we first feed $x$ into a pre-trained code-processing neural network (such as Tree-based CNN), then calculate the confidence score $s$ and compare it to a threshold $\delta$.

Mathematically, the out-of-distribution detector can be described as:

$$
g(x; \delta) = \begin{cases} 1 & \text{if } s \leq \delta, \\ 0 & \text{if } s > \delta. \end{cases}$$

3.4 Adversarial Attack and Robustness

Adversarial robustness for source code models include two steps. The first step is to generate samples that are adversarial to the normal samples, i.e., in hope that the classifier would find it difficult to associate them with the correct labels. The second step is to train the source code models in such a way that it can defend itself against an adversarial attack.

For code learning models, a general idea to generate adversarial samples is to apply some program transformations to obtain from a program a variant of itself. The requirement is that the semantics of the original program must be preserved by the transformed programs, i.e., given an input, the transformed program must produce the same output as does the original program.

The adversarial robustness of a code learning model refers to the minimal performance degradation when adversarial samples are used. Generative adversarial networks (GAN) could enhance the robustness of a given neural learning networks. In this work, we do not use them because our purpose is to measure the adversarial robustness through the generated samples.

4 Our Approach

Usually the general steps for training source code models are carried out as follows. Initially, the code samples needs to be mapped into an appropriate representation, and then the code representation is fed into a code encoder that is specifically designed to process such code presentation, e.g., with the code classification task using Tree-based CNN [43], the code samples in the main training dataset will be represented as the Abstract Syntax Tree (AST), and the AST will be fed into the TBCNN to obtain the code embedding. The code embedding is then fed into a softmax layer for classification, and the cross-entropy loss function is utilized as the objective to train the code classification model.

Figure 1 presents an overview of our framework. First, an auxiliary dataset including additional code samples will be treated as out-distribution data, while the samples in the main dataset will be treated as in-distribution data. The code samples in the OOD dataset will be encoded in exactly the same way as the samples in the main dataset, except for the steps after obtaining the code embeddings and logits:

the energy of both the auxiliary and main code embeddings are computed using an energy-bounded loss function. The intuition of introducing the energy-bound loss function is to assign distinct score ranges to separate out-of-distribution from in-distribution data. The cross-entropy and energy-bounded loss functions are trained jointly in our end-to-end learning framework as the total loss.

4.1 Energy-bounded Loss Function at Training Time

Here we present the energy-bounded loss function to train source code model with an auxiliary dataset. While an energy score may be advantageous for a pre-trained neural network, the energy difference between in- and out-of-distribution samples is not always optimal for distinction. Following [39], we propose an energy-bounded learning objective in which the neural network is designed to purposely generate a gap between in-distribution and out-of-distribution data by assigning lower energy to in-distribution data and larger energy to out-of-distribution data.

Specifically, the energy-based classifier is trained using the following objective function:

$$\min_\theta \mathbb{E}_{(x, y) \sim D_{\text{in}}} \left[ -\log F_{y}(x) \right] + \lambda \cdot L_{\text{energy}} \quad (3)$$

where $F(x)$ is the softmax output of the classification model and $D_{\text{in}}$ is the in-distribution training data.

The overall training objective combines the standard cross-entropy loss, along with a regularization loss defined in terms of energy:

$$L_{\text{energy}} = \mathbb{E}_{(x_{\text{in}}, y) \sim D_{\text{in}}} \left( \max(0, E(x_{\text{in}}) - m_{\text{in}}) \right)^2 \quad (4)$$

$$+ \mathbb{E}_{x_{\text{out}} \sim D_{\text{out}}} \left( \max(0, m_{\text{out}} - E(x_{\text{out}})) \right)^2 \quad (5)$$

where $D_{\text{out}}$ is the unlabeled auxiliary OOD training data.

In particular, the energy is regularized using two squared hinge loss terms with separate margin hyperparameters $m_{\text{in}}$ and $m_{\text{out}}$. In one term, the model penalizes in-distribution samples that produce energy higher than the specified margin parameter $m_{\text{in}}$. Similarly, in another term, the model penalizes the out-of-distribution samples with energy lower than the margin parameter $m_{\text{out}}$. In other words, the loss function penalizes the samples with energy $E(x) \in [m_{\text{in}}, m_{\text{out}}]$.

In our evaluation, we choose $\lambda = 0.1$. $m_{\text{in}}$ is chosen from $\{-3, -5, -7\}$, and $m_{\text{out}}$ is chosen from $\{-15, -19, -23, -27\}$.

4.2 Using Energy on Pretrained Models for OOD Detection at Inference Time

Once the model is trained, the energy score can be derived from it for OOD detection. Here we intuitively present the steps to derive the energy score.

The energy-based model has an inherent connection with modern machine learning, especially discriminative models.
Figure 1. An overview of our approach: the encoder processes both the in-distribution dataset and the out-of-distribution dataset to get the corresponding embeddings. Then the embeddings are fed into a feed-forward neural network to get the logits. The scalar energy scores $E_{\text{in}}$ and $E_{\text{out}}$ for in- and out-of-distribution samples respectively are computed from the logits. Next, the energy-bounded loss function $L_{\text{energy}}$ is computed from $E_{\text{in}}$ and $E_{\text{out}}$ and the cross-entropy loss $L_{\text{cross-entropy}}$ is computed from the in-distribution’s logits. Finally, $L_{\text{energy}}$ and $L_{\text{cross-entropy}}$ will be jointly trained into a total loss function $L_{\text{total}}$ for the end-to-end training process.

To see this, consider a discriminative neural classifier $f(x) : \mathbb{R}^D \rightarrow \mathbb{R}^K$, which maps an input $x \in \mathbb{R}^D$ to $K$ real-valued numbers known as logits.

Typically, these logits are used to derive a categorical distribution using the softmax function:

$$p(y \mid x) = \frac{e^{f_y(x)/T}}{\sum_i e^{f_i(x)/T}},$$

(6)

where $f_y(x)$ indicates that the $y^{\text{th}}$ index of $f(x)$, i.e., the logit is corresponding to the $y^{\text{th}}$ class label.

By connecting Eq. 1 and Eq. 6, the energy can be defined for a given input $(x, y)$ as $E(x, y) = -f_y(x)$. More importantly, without changing the parameterization of the neural network $f(x)$, we can express the free energy function $E(x; f)$ over $x \in \mathbb{R}^D$ in terms of the denominator of the softmax activation:

$$E(x; f) = -T \cdot \log \sum_i e^{f_i(x)/T}.$$  

(7)

It should be noted that the energy score can be derived from a pre-trained model that was trained using either the cross-entropy or the energy-bounded loss function. We will demonstrate the use of the energy score in the Evaluation section.

5 Evaluation

We describe two ways for measuring the robustness of code learning models. The first way is to determine whether or not a sample is out-of-distribution, i.e., OOD Detection. The second way is to determine whether a code learning model can continue to predict correctly against adversarial samples, i.e., Adversarial Robustness.

Notably, these two robustness measurements must be evaluated in accordance with specific code learning tasks. In Section 5.1, we describe the code learning tasks and the source code models for each of the task. Then in Sections 5.3 and 5.4, we describe the settings and the evaluation results for OOD Detection and Adversarial Robustness measurements, respectively.

5.1 Code Learning Tasks

The code learning tasks we consider in this paper are code classification, method name prediction. All of these tasks share a similar framework such that the code representation is fed into the code encoder, then the code encoder is used to predict some labels such that the tag name of code snippet, the method name of a function, or text description of a function body. The code encoder can be any of the well-known models that have been used for code learning recently, such as Tree-based CNN [44], Code2vec [5], Code2seq [3], BiLSTM [55] and Transformer [58]. Due to the generality of deep learning representations, each of these models can be adapted into these code learning tasks, even when their original design is for other tasks.

Below we describe the goal of each code learning task:

- **Code Classification**: Given a piece of code, this task is to classify the functionality it belongs to. This task may go under different names depending on the purposes, such as malware classification, patch identification, bug triage, etc. Even though the names are different, the general goal is similar, i.e., to classify a piece of code into a given class. We choose ‘code classification’ to represent this family of problems because their processes are similar;
• **Method Name Prediction**: Given a piece of code (without its function header), this task is to predict a meaningful name that reflects the functionality of the code, e.g., `get_computation_result`. It should be noted that this task can be formulated in a variety of ways. The first way, e.g., Code2Vec [5], is to treat this problem as an extreme classification task, where classes are the set of method names which can be a huge number. The second way, e.g., Code2seq [3], is to treat this problem as a generation task, meaning that one can use a decoder to generate a sequence of sub-tokens. Since our aim is to consider OOD for classification task, we use the first approach like Code2Vec to predict the method name out of a large number of classes.

5.2 Training Settings

We train the code classification models and the method name prediction models using two different loss functions. The first one is the well-known cross-entropy loss function used in most of the source code models [61]. The second one is the energy-bounded loss function we presented in this study (Equation 4). In general, we call the source code models trained on cross-entropy loss as CE Models, and the energy-bounded loss as EB Models.

Note that the code snippet is processed differently for the appropriate model. For TBCNN, Code2vec and Code2seq models, code must be parsed into an AST for these models to process. We use tree-sitter, an incremental parsing system which supports multiple programming languages [12] as the parser to parse code into the AST. For Bi-LSTM and Transformer, one can treat code simply as a sequence of tokens.

To train the neural networks, a common setting used among all these techniques is that they all utilize both node type and token information to initialize a node in ASTs to make it consistent and fair among all of the baselines, we follow [9] to set both the dimensionality of type embeddings and text embeddings to 100. Note that we make the source code models as strong as possible by choosing the hyper-parameters above as the optimal settings according to their papers or code. Also, note that the goal of this work is not to increase the performance of source code models on certain code learning tasks. Our goal is to train such models on EB loss function such that the performance of such EB models are comparable to the CE models, and the robustness of the EB models should be significantly higher than the CE models.

Table 1 shows the performance when we train such models using the cross-entropy loss (CE) and energy-bounded loss (EB). The metric to measure the quality of code classification (CC) is accuracy (Acc), and the metric for method name prediction (MNP) is F1 score on the sub-tokens. As seen, the performance of any of the models are comparable between CE and EB. As one can observe, the accuracy of EB of different models are similar to CE of the original models. **This shows that training with the energy-bounded loss does not affect on the prediction capability of source code models.**

5.3 Out-of-distribution Detection

5.3.1 Baselines. We aim to compare our energy score with the other scores for OOD detection, they are:

- **Maximum softmax probability (MSP)** [25]: the softmax score has been used as a strong baselines to detect OODs, i.e., OOD samples tends to have a lower value of the softmax score than the in-distribution samples, thus it can be use to detect the OOD samples.
- **Mahalanobis Distance** [33]: this score is to measure the probability density of test sample on feature spaces of DNNs utilizing the concept of a "generative" classifier.
- **ODIN** [37]: this score is an improved version of the MSP score, which uses an additional temperature scaling of the softmax score to improve OOD detection.

In contrast, we used the energy score in two different manners. First, an energy score is derived from Equation 7 for a pre-trained CE model, this is denoted as Eneg. Second, a similar energy score is also derived from Equation 7, but from a pre-trained EB model instead. By training the source code model with an auxiliary dataset, we would like to know whether or not there is any superior performance using the energy-bounded loss function compared to the simply derived energy score from a CE model.

5.3.2 Datasets.

**Code Classification.** For in-distribution dataset, we use the POJ dataset from [44] that has been widely used as the benchmark for code classification, which comprises of 52000 C programs of 104 classes. For out-distribution dataset, we use the data from Project CodeNet [49], a large-scale dataset that consisting of over 14 million code samples and about 500 million lines of code in 55 different programming languages. Because the POJ dataset is for C language only, we select the samples in Project CodeNet that are written in C, which resulting in around 750k samples. The number of OOD samples is much larger than those in the in-distribution dataset. To deal with imbalanced datasets and speed up the

| Tasks | CC (Acc) | MNP (F1) |
|-------|----------|----------|
| Loss  | CE       | EB       | CE | EB |
| TBCNN | 95.12    | 95.35    | 40.89 | 39.86 |
| TreeCaps | 97.24 | 97.56    | 48.29 | 47.01 |
| Code2Vec | -      | -        | 27.49 | 28.83 |
| Bi-LSTM | 84.59   | 85.92    | 38.25 | 38.69 |
| Transformer | 93.49 | 93.80    | 45.39 | 44.28 |

Table 1. Performance of Code Classification (CC) and Method Name Prediction (MNP) of source code models when training with the Cross-Entropy loss (CE) and the Energy-Bounded loss (EB), respectively.
training progress, we randomly sample 60k samples from these 750k samples. Then for both the in- and out-distribution dataset, we split them into training/testing/validation with the ratio 70/20/10 into data portions called \( \text{in} – \text{training} \), \( \text{in} – \text{testing} \), \( \text{in} – \text{validation} \) and \( \text{out} – \text{training} \), \( \text{out} – \text{testing} \), \( \text{out} – \text{validation} \).

**Method Name Prediction.** For in-distribution dataset, we use the Java-Small dataset from [3] that has been widely used as the benchmark for method name prediction, which comprises of 700k samples. This dataset have been split into training/testing/validation by projects, which can considered as \( \text{in} – \text{training} \), \( \text{in} – \text{testing} \), \( \text{in} – \text{validation} \). For out-distribution dataset, we also use the data from Project CodeNet [49]. Because the Java-Small dataset is for Java, we only select the samples in Project CodeNet that are written in Java, which resulting in 2M samples. We also randomly select 700k samples from these 2M samples and split it into training/testing/validation with the ratio 70/20/10 as the \( \text{out} – \text{training} \), \( \text{out} – \text{testing} \), \( \text{out} – \text{validation} \).

### 5.3.3 Metrics

To evaluate out-of-distribution detection methods, we treat the OOD examples as the positive class, and then evaluate four metrics: area under the receiver operating characteristic curve (AUROC), area under the precision-recall curve (AUPR), the false positive rate at \( N \% \) true positive rate (FPR), and the test error of the \( \text{in} – \text{testing} \) (Test Error).

AUROC and AUPR are holistic metrics that summarize the performance of a detection method across multiple thresholds. They are also useful when the anomaly score threshold is not known \textit{a priori}. AUROC can be thought of as the probability that an anomalous example is given a higher OOD score than an in-distribution example [15]. Thus, a higher AUROC is better, and a random detector has an AUROC of 50%. AUPR is useful when anomalous examples are infrequent [41], as it takes the base rate of anomalies into account.

Whereas AUROC and AUPR represent the detection performance across various thresholds, the FPRN metric represents the performance at one strict threshold. By observing the performance at a strict threshold, we can make clear comparisons among strong detectors. The FPRN metric [6, 30, 38] is the probability that an in-distribution example (negative) raises a false alarm when \( N \% \) of anomalous examples (positive) are detected, so a lower FPRN is better.

Capturing nearly all anomalies with few false alarms can be of high practical value. The Test Error is also a metric derived from FPRN that measure the misclassification probability when TPR is 95%.

### 5.3.4 Evaluation Results

Table 2 shows the main evaluation results. The abbreviated metric names “MSP” stands for the Maximum Softmax Probability score, “Maha” stands for the Mahalanobis score. “ODIN” is the MSP score with additional temperature scaling factor, “Energy” is the energy score computed from the Equation 7. Note that all of these scores are computed from a CE Model.

The remaining metrics are “energy-bounded,” which also refer to the energy score derived in Equation 7, but this score is computed from the EB models instead. The metrics for evaluation are FPR95, AUROC, AUPR and Test Error. Here the marker \( \downarrow \) indicates that smaller values are better and the marker \( \uparrow \) indicates that larger values are better.

We can see that, on average, the OOD detection performance in term of FPR95, AUROC, AUPR using the energy score outperforms all of the other scores for any of the CE Models on both of the code classification and method name prediction task. This shows that the energy score is a powerful measurement to detect OOD samples. Moreover, if we train the source code models using the energy-bounded loss function, the derived energy score can distinguish in- and out-distribution samples even better, i.e., we can see in Table 1 that the Energy-bounded score for all of the metrics (FPR95, AUROC, AUPR, Test Error) across all of the models are the best among the other measurement scores.

It should be noted that the performance of the Bi-LSTM is mostly worse than the other code models in any of the metric, it is because Bi-LSTM is not a strong model for code learning tasks, but we still involve this into the evaluation to show that the energy-bounded learning method is even applicable to a weaker model.

To gain further insights, we compare the energy score distribution for in- and out-of-distribution data for the code classification task. Figure 2 compares the energy and softmax score histogram distributions, derived from the TBCNN model trained on cross-entropy loss (TBCNN-CE), and another TBCNN model trained on energy-bound loss (TBCNN-EB) for the code classification task. As we can see in Figure 2a, the softmax histograms derived from TBCNN-CE for in- and out- distribution makes it difficult to distinguish the two distributions, resulting in FPR95 value of 99.71%. On the other hand, using the energy histograms derived from TBCNN-CE in Figure 2b makes it better to distinguish the two distributions, resulting in FPR95 value of 68.62%. Finally, Figure 2c shows the energy histograms derived from TBCNN-EB, resulting in the perfect value of FPR95=0.0%. This demonstrates the superior performance of the energy-bound learning model trained with the auxiliary dataset.

### 5.4 Adversarial Robustness

We consider the task to measure the robustness of each model by applying the semantically-preserving program transformations to a set of test programs. We follow Rabin et al. [50], Wang and Su [60] to transform programs in 5 ways that change code syntax but preserve code functionality:

- **Variable Renaming (VN),** a refactoring transformation that renames a variable in code, where the new name of the
Figure 2. Distribution of softmax scores vs. energy scores from TBCNN-CE vs. TBCNN-EB. (a) FPR95 = 99.71 for the softmax confidence score derived from the TBCNN-CE model. (b) FPR95 = 68.62 for the energy score derived from the TBCNN-CE model. (c) FPR95 = 0.00 for the energy score derived from the TBCNN-EB model.

Table 2. OOD Performance. CC stands for Code Classification, MNP stands for Method Name Prediction. ↓ indicates smaller values are better and ↑ indicates larger values are better.

| Tasks | Models | FPR95 ↓ | AUROC ↑ | AUPR ↑ | Test Error ↓ |
|-------|--------|---------|---------|--------|--------------|
| CC    | TBCNN  | 99.71 / 89.85 / 74.58 / 68.62 / 0.00 | 14.74 / 43.66 / 51.98 / 74.05 / 99.71 | 48.33 / 61.39 / 84.39 / 99.98 | 33.09 / 36.32 / 31.24 / 32.37 / 6.71 |
|       | Bi-LSTM | 99.87 / 94.96 / 89.43 / 92.33 / 7.96 | 64.33 / 76.65 / 88.34 / 95.82 / 96.99 | 33.89 / 34.56 / 31.32 / 35.35 / 6.42 | |
|       | TreeCaps | 99.63 / 93.82 / 88.28 / 97.34 / 0.00 | 62.46 / 79.14 / 62.62 / 98.25 / 99.99 | 65.07 / 68.09 / 64.30 / 97.82 / 99.99 | 31.92 / 30.97 / 40.07 / 67.49 / 0.71 |
|       | Transformer | 99.68 / 86.92 / 84.41 / 88.85 / 0.00 | 20.58 / 45.45 / 60.32 / 73.95 / 99.97 | 84.21 / 61.32 / 65.26 / 84.90 / 99.95 | 32.59 / 28.56 / 35.41 / 37.17 / 0.90 |
| MNP   | TBCNN  | 89.71 / 80.85 / 74.58 / 68.28 / 6.56 | 25.56 / 34.55 / 40.57 / 58.35 / 93.26 | 39.29 / 85.56 / 40.59 / 64.41 / 92.49 | 25.53 / 24.55 / 27.59 / 24.31 / 12.29 |
|       | Bi-LSTM | 86.16 / 82.85 / 74.58 / 10.28 / 14.89 | 62.46 / 79.14 / 62.62 / 98.25 / 99.99 | 65.07 / 68.09 / 64.30 / 97.82 / 99.99 | 31.92 / 30.97 / 40.07 / 67.49 / 0.71 |
|       | Code2Vec | 89.71 / 89.36 / 74.58 / 68.28 / 6.56 | 25.56 / 34.55 / 40.57 / 58.35 / 93.26 | 39.29 / 85.56 / 40.59 / 64.41 / 92.49 | 25.53 / 24.55 / 27.59 / 24.31 / 12.29 |
|       | TreeCaps | 80.36 / 92.84 / 63.56 / 60.23 / 2.34 | 38.32 / 48.15 / 60.32 / 84.97 / 97.23 | 42.63 / 71.82 / 67.99 / 92.89 / 97.22 | 28.83 / 24.71 / 45.32 / 39.49 / 10.75 |
|       | Transformer | 89.71 / 89.36 / 74.58 / 68.28 / 6.56 | 25.56 / 34.55 / 40.57 / 58.35 / 93.26 | 39.29 / 85.56 / 40.59 / 64.41 / 92.49 | 25.53 / 24.55 / 27.59 / 24.31 / 12.29 |

variable is taken randomly from a set of variable vocabulary in the training set;
• Unused Statement (US), inserting an unused string declaration to a randomly selected basic block in the code.
• Permute Statement (PS), swapping two independent statements (i.e., with no dependence) in a basic block in the code.
• Loop Exchange (LX) replaces for loops with while loops or vice versa. We traverse the AST to identify the node the
defines the for loop (or the while loop) then replace one with another with modifications on the initialization, the condition, and the afterthought.

- Switch to If (SF) replaces a switch statement in the method with its equivalent if statement. We traverse the AST to identify a switch statement, then extract the subtree of each case statement of the switch and assign it to a new if statement.

We then examine if the models make the same predictions as the prior predictions for the original programs after the programs have been transformed. We use the method name prediction task with the similar datasets used in OOD detection for this Adversarial Robust measurement.

5.4.1 Metrics. We use percentage of predictions changed (PPC) as the metric used by [50, 60, 63] to measure the robustness of the code models. Formally, suppose $P$ denotes a set of test programs, a semantic-preserving program transformation $\mathcal{T}$ that transforms $P$ into a set of transformed programs $P' = \{ p' = \mathcal{T}(p) | p \in P \}$, and a source code model $\mathcal{M}$ that can make predictions for any program $p$: $\mathcal{M}(p) = l$, where $l \in L$ denotes a predicted label for $p$ according to a set of labels $L$ learned by $\mathcal{M}$, we compute the percentage of predictions changed as:

$$PPC = \frac{|\{ p' \in P' | \mathcal{M}(p) \neq \mathcal{M}(p') \} |}{|P'\}| \times 100 \quad (8)$$

Lower $PPC$ values for $\mathcal{M}$ suggest higher robustness as they can maintain more of correct predictions with respect to the transformation.

5.4.2 Results. Table 3 shows the results of this task by comparing the before/after percentage of prediction changes with respect to the five semantic-preserving transformations. It is clear that for all these transformations, EB outperforms CE in terms of the adversarial robustness (by an average of 6.7% across the transformations).

Although additional types of program transformations could be used to enhance model robustness, the analysis of current transformations already indicates that training all source code models with energy-bounded loss makes them more robust to adversarial attacks. In future work, one could introduce more semantic-preserving program transformations to enhance the robustness of the models.

6 Case Study

Our goal in this study is to use the pre-trained EB models to determine if a code snippets in the in-testing are out-of-distribution or not. We choose the method name prediction since code classification is not suitable for this analysis. Because the in-testing of code classification is split from the same data distribution, while the in-testing from method name prediction is not split by the same data distribution, but is collected by projects, e.g., Wildfly and Gradle are 2 projects in the in-testing, thus determine if the code snippets in these projects are OOD can bring benefit such that it can reduce the false-positives produced by the model’s prediction.

To do so, we need to define a threshold $\delta$ on the energy score, such that when the energy score is higher than $\delta$, then the sample is considered as out-distribution, otherwise it will be considered as in-distribution. We use the ROC curve to help us to find a good threshold. A good threshold in our case is to satisfy that the True Positive Rate (TPR) must be high, the the False Positive Rate (FPR) must be low, i.e., miss-classifications are low. Since our dataset is balanced, we choose the threshold that maximize TPR – FPR.

In this study, we employ the TreeCaps-EB model for method name prediction. Thus, finding the $max(TPR – FPR)$ value for the energy threshold from the test samples yields a value of $-15.82$. Then, we utilize this threshold to assess whether or not the code snippets contained in the in-testing section are OOD. According to our analysis, 64% of the testing samples in the in-testing part are OOD. This means that if we are certain that the model will fail to predict a suitable name for these samples, we can disregard them during the testing phase. By eliminating these 64% test samples from the testing phase, we can enhance the F1 score for the TreeCaps-EB method name prediction from 39.86% to 74.38%. To ensure that eliminating these 64% is indeed significant, we randomly select another 64% of test samples to eliminate and examine the remainder; this results in an F1 score of 44.34% only. Comparing to the F1=74.38% increased from excluding the 64% OODs, the F1=44.35% from excluding 64% randomly is significantly lower. This means that 64% of OOD test samples assist us in excluding samples for which the model cannot make an accurate prediction.

It is worth noting that code learning models, even the state-of-the-art ones such as TreeCaps, are not always able to get correct results for the classification tasks that involve many class labels. For example, Figure 3 shows some code snippets where TreeCaps-EB model could not predict the correct name for the method name prediction task. These samples have been predicted to be OOD by our method. Therefore, it is possible to use OODD to enhance the understanding that there is a limitation in code learning models in such tasks.

In summary, threats of validity in this case study can be summarised as follows:

**Construct validity.** Even though we have chosen an energy threshold $\delta = -15.82$ based on the principle to maximize the TPR-FPR, it is still an empirical parameter which may vary by datasets of case studies. It would require further study to know whether there could be better signals to determine the threshold. Another potential threat to

---

2 https://github.com/wildfly/wildfly
3 https://github.com/gradle/gradle
Table 3. Model robustness between cross-entropy loss (CE) vs energy-bounded loss (EB), measured as percentage of predictions changed wrt. semantic-preserving program transformations. The lower the more robust.

| Model     | VR CE | VR EB | US CE | US EB | PS CE | PS EB | LX CE | LX EB | SF CE | SF EB |
|-----------|------|------|-------|-------|-------|-------|-------|-------|-------|-------|
| TBCNN     | 22.45% | 15.48% | 24.92% | 14.60% | 18.49% | 15.33% | 19.84% | 15.33% | 25.09% | 18.50% |
| TreeCaps  | 16.39% | 10.49% | 19.95% | 15.33% | 17.39% | 11.20% | 23.22% | 15.29% |
| Code2Vec  | 24.40% | 20.59% | 24.06% | 20.58% | 19.55% | 17.29% | 23.88% | 18.49% | 29.31% | 18.02% |
| Bi-LSTM   | 28.29% | 22.60% | 29.92% | 20.99% | 26.29% | 20.55% | 27.23% | 22.50% |
| Transformer | 20.09% | 16.33% | 21.11% | 16.29% | 20.94% | 16.30% | 22.98% | 16.23% | 27.95% | 21.28% |

validity is the choice of procedural programming languages. Since existing code classification and method name prediction tasks have not been evaluated on functional programming paradigms, we do not extend our scope to them either. In future we may extend the scope to other paradigms after the code learning models have been evaluated to them.

**Internal and external validity.** By using semantics-preserving program transformations, we minimize the potential errors introduced by us because all the labels of out-of-distribution samples are determined by the algorithm. The two tasks being used for this study have been bench marked by other researchers in this field. We have not introduced additional datasets and labels that could bias the analysis.

7 Discussion

In this section, we would like to explore our technique in further detail. To begin with, our technique was assessed exclusively on classification-based tasks, but many programming problems might be formulated differently. For instance, extreme summarization [2] is to summarize a program into textual description, program translation [11] is to translate a program from one language into another. These two tasks can be considered as translation tasks in general, and there is a need to quantify the uncertainty of the translated results, i.e., the model should not produce incorrect translation results if the output is uncertain. Indeed, numerous works have proposed techniques for testing such machine translation tasks for natural language processing [23, 24, 56], but these techniques focus exclusively on the erroneous in the translated sentences, rather than on their uncertainty. Additionally, the methods are limited to natural language processing, but our approach is suitable to code representation learning.

Second, there are a few other ways for evaluating the robustness of source code models against adversarial attacks. For instance, Yefet et al. [62] proposes a gradient-based technique for generating adversarial examples that can be used to attack source code models. This method has been used to generate adversarial examples to bypass method name prediction and bug detection task. In our situation, we use a more straightforward way suggested by Rabin et al. [51] to apply a predefined set of transformation operators to the code snippets. We leave the task to use the gradient-based method to evaluate against our energy-based models in the future.

Figure 3. Examples of code snippets that our energy-based learning algorithm identified as being out-of-distribution successfully. We illustrate the ground-truth method name in each case study alongside the prediction provided by the TreeCaps model from the Softmax layer. All of the predicted names are incorrect or incomplete.
8 Conclusion

In this paper, we propose an energy-bound loss function as an alternative for the cross-entropy loss function commonly used to train source code models. Along with the main dataset (in-distribution dataset), the energy-bound objective is trained using an auxiliary dataset (out-of-distribution dataset). We hypothesize that this training technique improves the robustness of source code models while preserving their predictive ability and the energy score produced from such models is used to detect code snippets that are out-of-distributions. The results of our evaluation on two programming-related tasks indicate that the energy-bound score surpasses all of the other scores for the OOD detection task. Additionally, for the method name prediction task, models trained on the energy-bound loss function demonstrate higher robustness than those models trained on the cross-entropy loss function. It is therefore recommendable to consider this alternative in classification-based code learning.

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In International Conference on Learning Representations.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In International Conference on Machine Learning, PMLR, 2991–2100.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2vec: Generating Structures from Synchronous Representations of Code. In International Conference on Learning Representations. https://openreview.net/forum?id=H1gKY609rX

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. Proc. ACM Programming Languages 3, POPL (Jan. 2019), 40:1–40:29.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages 3, POPL (2019), 1–29.

[6] Vassileios Baltas, Edgar Riba, Daniel Ponsa, and Krystian Mikolajczyk. 2016. Learning local feature descriptors with triplets and shallow convolutional neural networks... In Bmvc; Vol. 1. 3.

[7] Pavol Bielik and Martin Vechev. 2020. Adversarial Robustness for Code. arXiv preprint arXiv:2002.04694 (2020).

[8] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net. https://openreview.net/forum?id=Bke44sASFX

[9] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. In Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 35. 30–38.

[10] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In 2017 ieee symposium on security and privacy (sp). IEEE, 39–57.

[11] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. arXiv preprint arXiv:1802.03691 (2018).

[12] Timothy Clem and Patrick Thomson. 2021. Static analysis at GitHub: an experience report. Queue 19, 4 (2021), 42–67.

[13] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. 2013. Large-scale malware classification using random projections and neural networks. In IEEE International Conference on Acoustics, Speech and Signal Processing. IEEE, 3422–3426.

[14] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. 2018. Adversarial Attack on Graph Structured Data. Proceedings of the 35th International Conference on Machine Learning, PMLR 80 (Jul 2018). http://par.nsf.gov/biblio/10095927

[15] Jesse Davis and Mark Goodrich. 2006. The relationship between Precision-Recall and ROC curves. In Proceedings of the 23rd international conference on Machine learning, 233–240.

[16] Terrance DeVries and Graham W Taylor. 2018. Learning confidence for out-of-distribution detection in neural networks. arXiv preprint arXiv:1802.04865 (2018).

[17] Yinpeng Dong, Fangzhou Liao, Tianyu Pang, Hang Su, Jun Zhu, Xiaolin Hu, and Jianguo Li. 2018. Boosting adversarial attacks with momentum. In Proceedings of the IEEE conference on computer vision and pattern recognition. 9185–9193.

[18] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net. https://openreview.net/forum?id=H1ers0Rtm

[19] Will Craftholt, Kuan-Chieh Wang, Jörn-Henrik Jacobsen, David Duvendau, Mohammad Norouzi, and Kevin Swersky. 2019. Your classifier is secretly an energy based model and you should treat it like one. arXiv preprint arXiv:1912.03263 (2019).

[20] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In 40th ICSE. 993–944.

[21] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepPM: Migrate APIs with Multi-modal Sequence to Sequence Learning. In International Joint Conference on Artificial Intelligence (Melbourne, Australia). 3675–3681.

[22] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Neural Attribution for Semantic Bug-Localization in Student Programs. In Advances in Neural Information Processing Systems. 11861–11871.

[23] Shashij Gupta, Pinjia He, Clara Meister, and Zhendong Su. 2020. Machine translation testing via pathological invariance. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 863–875.

[24] Pinjia He, Clara Meister, and Zhendong Su. 2021. Testing Machine Translation via Referential Transparency. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 410–422.

[25] Dan Hendrycks and Kevin Gimpel. 2016. A baseline for detecting misclassified and out-of-distribution examples in neural networks. arXiv preprint arXiv:1610.02136 (2016).

[26] Thong Hoang, Julia Lawall, Richard J Oentaryo, Yuan Tian, and David Du. 2019. PatchNet: a tool for deep patch classification. In 2019 IEEE/ACM 41st International Conference on Software Engineering Companion Proceedings (ICSE-Companion). IEEE, 83–86.

[27] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In International Conference on Program Comprehension. ACM, 290–210.

[28] Robin Jia and Perry Liang. 2017. Adversarial examples for evaluating reading comprehension systems. arXiv preprint arXiv:1707.07328 (2017).

[29] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: a code-to-code search engine. 946–957.

[30] Vijay Kumar BG, Gustavo Carneiro, and Ian Reid. 2016. Learning local image descriptors with deep siamese and triplet convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition. 5385–5394.

[31] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial machine learning at scale. arXiv preprint arXiv:1611.01236 (2016).
[32] Yann LeCun, Sumit Chopra, Raia Hadsell, M Ranzato, and F Huang. 2006. A tutorial on energy-based learning. *Predicting structured data* 1, 0 (2006).

[33] Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. 2018. A simple unified framework for detecting out-of-distribution samples and adversarial attacks. *Advances in neural information processing systems* 31 (2018).

[34] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. 2017. Software defect prediction via convolutional neural network. In *IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 318–328.

[35] Jiwei Li, Will Monroe, and Dan Jurafsky. 2016. Understanding neural networks through representation erasure. *arXiv preprint arXiv:1612.08820* (2016).

[36] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Generative Adversarial Nets. In *International Conference on Learning Representations*.

[37] Shiyu Liang, Yixuan Li, and R. Srikanth. 2018. Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=H1VGLxRZ

[38] Si Liu, Risheek Garrepalli, Thomas Dietterich, Alan Fern, and Dan Hendrycks. 2018. Open category detection with PAC guarantees. In *International Conference on Machine Learning*. PMLR, 3169–3178.

[39] Weitang Liu, Xiaoyun Wang, John D Owens, and Yixuan Li. 2020. Energy-based out-of-distribution detection. *arXiv preprint arXiv:2010.03759* (2020).

[40] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yaping Liu. 2021. Deep Learning for Android Malware Defenses: A Systematic Literature Review. *arXiv preprint arXiv:2103.05292* (2021).

[41] Christopher Manning and Hinrich Schutze. 1999. *Foundations of statistical natural language processing*. MIT press.

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

[43] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jun. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI Conference on Artificial Intelligence*.

[44] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jun. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 1287–1293.

[45] Saeed Nari and Ali A Ghorbani. 2013. Automated malware classification based on network behavior. In *2013 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 642–647.

[46] R. Nix and J. Zhang. 2017. Classification of Android apps and malware using deep neural networks. In *International Joint Conference on Neural Networks*. 1871–1878.

[47] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marin-nescu, and Anil Thomas. 2015. Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1916–1920.

[48] Michael Pradel and Koushik Sen. 2018. Deep Bugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 147.

[49] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *arXiv preprint arXiv:2105.12655* (2021).

[50] Md Rabin, Rafiqul Islam, Nghi DQ Bui, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2020. On the Generalizability of Neural Program Analyzers with respect to Semantic-Preserving Program Transformations. *arXiv preprint arXiv:2008.01566* (2020).

[51] Md Rafiul Islam Rabin, Nghi DQ Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. 2021. On the generalizability of Neural Program Models with respect to semantic-preserving program transformations. *Information and Software Technology* 135 (2021), 106552.

[52] Md Rafiqul Islam Rabin, Ke Wang, and Mohammad Amin Alipour. 2019. Testing Neural Program Analyzers. In *34th IEEE/ACM International Conference on Automated Software Engineering (Late Breaking Research-Track)*.

[53] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Al Alburgougli, Somesh Jha, and Thomas Reps. 2020. Semantic Robustness of Models of Source Code. *arXiv preprint arXiv:2002.03043* (2020).

[54] Saksham Sachdev, Hongyu Li, Sifei Lan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on Source Code: A Neural Code Search. In *2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA), 31–41. https://doi.org/10.1145/3211346.3211353

[55] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[56] Zeyu Sun, Jie M Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. 2020. Automatic testing and improvement of machine translation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 974–985.

[57] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *ICLR*.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[59] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *33rd ASE* (Montpellier, France). New York, NY, USA, 397–407. https://doi.org/10.1145/3238147.3238206

[60] K Wang and Zhendong Su. 2019. Learning blended, precise semantic program embeddings. *arXiv*, vol. abs/1907.02316 (2019).

[61] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2020. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *arXiv preprint arXiv:2009.06520* (2020).

[62] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[63] Huangzhao Zhang, Zhao Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Programming Models. In *34th AAAI Conference on Artificial Intelligence*.

[64] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Philip S. Yu. 2018. Energy-based out-of-distribution detection. *arXiv preprint* arXiv:1805.09820 (2018).

[65] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F Bissyandé, Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Nghi DQ Bui. 2020. A systematic literature review on source code analysis tools and systems of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–38.

[66] Zhengli Zhao, Dheeru Dua, and Sameer Singh. 2018. Generating Natural Adversarial Examples. In *International Conference on Learning Representations*. https://openreview.net/forum?id=H1BLLgZC6

[67] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by
Learning Comprehensive Program Semantics via Graph Neural Networks. In Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 10197–10207. http://papers.nips.cc/paper/9209-devign-effective-vulnerability-identification-by-learning-comprehensive-program-semantics-via-graph-neural-networks

[68] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. 2018. Adversarial attacks on neural networks for graph data. In Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2847–2856.