CoCoFuzzing: Testing Neural Code Models with Coverage-Guided Fuzzing

Moshi Wei*, Yuchao Huang†, Jinqiu Yang‡, Junjie Wang†, Song Wang*
*York University, Canada
{moshiwei, wangsong}@yorku.ca
‡Concordia University, Canada
{jinqiuy}@encs.concordia.ca
†Institute of Software, Chinese Academy of Sciences, China
{hycsoge, junjie}@iscas.ac.cn

***Abstract*—Deep learning-based code processing models have demonstrated good performance for tasks such as method name prediction, program summarization, and comment generation. However, despite the tremendous advancements, deep learning models are frequently susceptible to adversarial attacks, which pose a significant threat to the robustness and generalizability of these models by causing them to misclassify unexpected inputs. To address the issue above, numerous deep learning testing approaches have been proposed; however, these approaches primarily target testing deep learning applications in the domains of image, audio, and text analysis, etc., and cannot be "directly applied" to "neural models for code" due to the unique properties of programs.**

In this paper, we propose a coverage-based fuzzing framework, `CoCoFuzzing`, for testing deep learning-based code processing models. In particular, we first propose ten mutation operators to automatically generate validly and semantically preserving source code examples as tests, followed by a neuron coverage-based approach for guiding the generation of tests. The performance of `CoCoFuzzing` is evaluated using three state-of-the-art neural code models, i.e., NeuralCodeSum, CODE2SEQ, and CODE2VEC. Our experiment results indicate that `CoCoFuzzing` can generate validly and semantically preserving source code examples for testing the robustness and generalizability of these models and enhancing neuron coverage. Furthermore, these tests can be used for adversarial retraining to improve the performance of neural code models.

***Index Terms*—Robustness, code model, language model, fuzzy logic, deep learning**

I. Introduction

Recent applications of deep learning (DL) have successfully accelerated a variety of tasks in automated source code processing, including the prediction of variable names [1], [2], code summarization [3]–[8], and API recommendation [9]–[11]. The majority of these deep learning-based code models (i.e., neural code models) have demonstrated excellent performance. However, it is well-known that deep learning models are susceptible to adversarial attacks [12], [13], in which a subtly-modified input can cause neural networks to misclassify data and result in severe DL model errors. The reliability and robustness of neural networks can be improved through more exhaustive testing. Several testing strategies for deep learning, such as coverage-guided fuzzing [14], adversarial-generative fuzzing [15], and adversarial examples generation [16] have been proposed to address the issue. However, the majority of these approaches primarily focus on generating tests for deep learning models in the domains of image, audio, and text analysis by employing mutations such as image adjustment, image scaling, image rotation, noise addition, etc., which cannot be applied directly to neural code models. In addition, unlike adversarial example generation for images, audio, and natural languages, the structured nature of programming languages introduces new challenges, i.e., the program must strictly adhere to the rigid lexical, grammatical, and syntactic constraints and tests generated for a program are expected to preserve the original program semantics.

Recently, Some adversarial example generation strategies for source code were recently proposed. Zhang et al. [17] performed variable name replacements to perturb the programs. Yefet et al. [18] proposed to use both variables renaming and dead code (i.e., unused variable declaration) insertion to generate semantically equivalent adversarial examples. However, the types of perturbations introduced by the two transformations are limited, as we have observed numerous other types of noise or perturbation in real-world software programs with the same semantics [19]. Consequently, using only the two operators for exhaustively testing neural code models could result in the omission of numerous edge cases.

In this paper, we propose a coverage-based fuzzing framework, `CoCoFuzzing`, to test neural code models. Specifically, ten mutation operators are proposed and implemented to represent various real-world, semantically preserving transformations of programs in order to automatically generate valid and semantically preserving source code examples as tests. Then, a neuron coverage-based guidance mechanism is used to systemically explore various program transformation types and guide the generation of tests. We investigate the performance of `CoCoFuzzing` on three state-of-the-art typical neural code models, i.e., NeuralCodeSum [3] (leverages a self-attention-based neural network to generate summarization of programs), CODE2SEQ [1] (builds an Abstract Syntax Tree (AST)-based Recurrent Neural Network (RNN) to predict method names), and CODE2VEC [2] (uses a path-based attention model for learning code embeddings to represent a method).

Our experiment results indicate that CoCoFuzzing can generate valid and semantically preserving tests for evaluating the robustness and generalizability of neural code models. Specifically, the newly-generated tests by CoCoFuzzing can reduce the performance of NeuralCodeSum, CODE2SEQ, and CODE2VEC by 84.81%, 22.06%, and 27.58% respectively. In addition, we find that adversarial retraining can be used to improve the performance of the neural code models of interest using these new tests by CoCoFuzzing. Specifically, the performance of NeuralCodeSum can be improved by 35.15%, CODE2SEQ by 8.83%, and CODE2VEC by 34.14% by retraining the models with synthetic data generated by CoCoFuzzing.

This paper makes the following contributions:

- This study proposes a coverage-based fuzzing framework CoCoFuzzing for testing the robustness of neural code models. To the best of our knowledge, CoCoFuzzing is the first fuzzing framework for testing neural code models. The implementation of our tool is made available for the replication of this study[1].
- This study proposes and implements a set of ten mutation operators that represent various real-world semantically preserving transformations of programs in order to automatically generate tests.
- This study conducts an experiment with a neuron coverage-based guidance mechanism for systemically exploring the large search space comprised of various types of program transformations and evaluating the fuzzing's adequacy.
- This study evaluates CoCoFuzzing against three state-of-the-art neural code process models, namely NeuralCodeSum, CODE2SEQ, and CODE2VEC. The results of the experiment demonstrate the efficacy of CoCoFuzzing.

The rest of this paper is organized as follows. Section II presents the background of neural code models. Section III describes the methodology of our proposed CoCoFuzzing. Section IV shows the setup of our experiments. Section V presents the result of our study. Section VI discusses the threats to the validity of this work. Section VII presents related studies. Section VIII concludes this paper.

II. Background

In this section, we present the background of neural code models and fuzzing testing.

A. Neural Code Models

The expanding availability of open-source repositories creates new opportunities for applying deep learning to accelerate code processing tasks such as prediction of variable names [1], [2], [20], code summarization [3]–[5], [21], and API recommendation [9]–[11]. Recurrent Neural Networks (RNNs) and Attention Neural Networks (ANNs) are the two types of deep neural networks used in neural code processing models. Below is a brief explanation of each architecture.

[1]https://doi.org/10.5281/zenodo.4000441

Recurrent Neural Network Architecture. A recurrent neural network is a neural network with a hidden state h and an optional output y which operates on a sequence of variable length $x = (x_1, ..., x_T)$. And at each time step t, the RNN's hidden state h_t is updated by $h_t = f(h_{t-1}, x_t)$, where f is a non-linear activation function. Due to the vanishing and exploding gradient problems, standard RNNs are incapable of learning "long-term dependencies", meaning they cannot propagate information that appeared earlier in the input sequence. Long short-term memory (LSTM) [22] has been proposed to address the above issue. It introduces additional internal states, known as memory cells, that are not affected by vanishing gradients and regulate the propagation of information.

Attention Network Architecture. Recent applications of attention neural models in the field of natural language processing have yielded very promising results [23]. A neural attention mechanism enables a neural network to focus a subset of its inputs when processing a large amount of data. Attention neural networks have three primary variants based on network characteristics: global and local attention [24], hard and soft attention [25], and self-attention [26]. Google recently proposed the Transformer model [27] for natural language processing tasks; it is the first transduction model to rely solely on self-attention to compute input and output representations.

B. Fuzz Testing

Software fuzzing is a technique for testing a program that generates mutants by modifying valid seed inputs [28]. The mutants fail tests if they exhibit abnormal behaviors (such as crashing the system being tested); otherwise, they pass. Coverage-guided fuzzing has been proposed and is widely used to discover numerous critical bugs in real software [29], in which a fuzzing process maintains an input data corpus for the program under consideration. Changes are made to those inputs based on a mutation procedure, and mutated inputs are retained in the corpus when they exhibit new "coverage". Traditional software fuzz testing techniques utilize code coverage metrics that track which lines of code have been executed and which branches have been followed [28]. However, these traditional fuzzing frameworks could not be directly applied to deep neural network-based software due to a fundamental difference in the programming paradigm and the development process [14], i.e., a neural network run on different inputs will frequently execute the same lines of code and follow the same branches, but may produce significantly different behavior due to the difference in input values.

Recently, many coverage-guided fuzz testing frameworks such as DeepHunter [14], TensorFuzz [30], and DeepTest [31] have been proposed to test deep neural networks. These frameworks apply neuron coverage metrics to guide the fuzzing test to deep learning applications in the domain of image processing, and they perform image transformations such as blurring and shearing to generate

III. The Approach of CoCoFuzzing

In this section, we describe the approach of CoCoFuzzing. Figure 1 shows the overview and

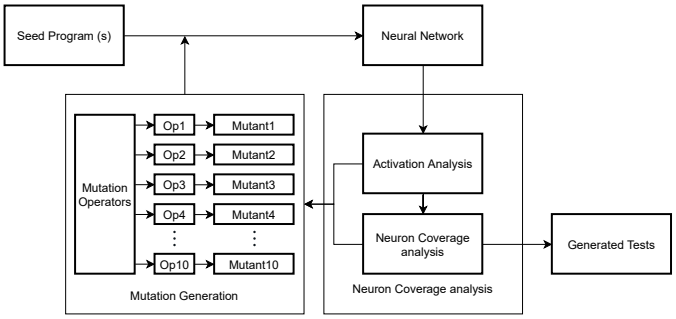


Fig. 1: The overview of our proposed CoCoFuzzing.

Algorithm 1 Coverage-Guided Test Generation in CoCoFuzzing

Input: Seed Program List *S*; Target Neural Code Model *NM*; Mutation Operators *Ops*; Maximum number of mutations *MAX*;
Variables: sets of activated neurons NC_p, NC_{curr},NC_{best}
Seed Program *P*;
mutation operator *op*;
number of accumulative mutation $numTries$;
the best mutant among all mutants *bestMutant*;
neuron activation count of the best mutant *bestAC*;
neuron activation count of current mutant *currentAC*;
functions: *neuronActivation* calculates the activated neuron by Program P;
newNeurons calculates the new activated neuron compare to NC_p;
Output: generated tests *T*;

```
1: for P in S do
2:     NA_p ← neuronActivation(P, NM)
3:     for numTries = 1, numTries <= MAX, numTries++ do
4:         bestMutant ← null
5:         NA_best ← ∅
6:         bestAC ← 0
7:         for op in Ops do
8:             currMutant = mutate(P, op)
9:             NA_curr ← newNeurons(currMutant, NA_p, NM)
10:            currentAC ← size(NA_curr)
11:            if currentAC > bestAC then
12:                bestAC ← currentAC
13:                NA_best ← NA_curr
14:                bestMutant ← currMutant
15:            end if
16:        end for
17:        if bestMutant! = null then
18:            P ← bestMutant
19:            NA_p ← NA_best ∪NA_p
20:            T.push(P)
21:        end if
22:    end for
23: end for
```

Algorithm 1 describes the main algorithm in detail. CoCoFuzzing takes a set of initial seed programs and a neural code model as the input, and produces new test sets iteratively through mutation generation (Section III-C) and neuron coverage analysis (Section III-B).

A. Overview of CoCoFuzzing

Algorithm 1 shows how CoCoFuzzing works step by step. CoCoFuzzing begins by randomly selecting one seed program from the seed queue (lines 1–2), then transforming the input source code based on a predefined list of mutation operators (line 7). Given the non-trivial search space constructed by combinations of mutation operators, CoCoFuzzing uses a neuron coverage-guided (NC-guided) approach (lines 3–23), to search for specific types of mutated programs. At a high level, CoCoFuzzing searches for program mutations that activate the most number of new neurons (lines 11–14) while controlling the maximum mutations on a single seed program (line 3 and a threshold *MAX*), for both naturalness and the viability of implementing metamorphic testing (i.e., mutated programs can use the same test oracle as the seed program).

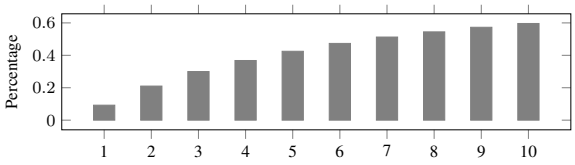


Fig. 2: Average percentages of noise code in the mutants generated with different MAX (i.e., X axis).

Note that in traditional coverage-guided fuzzers for computer programs [32], a seed can be reused and mutated multiple times, e.g., these tools frequently use either time constraints or the number of generated mutants as the termination condition, and a single seed can produce thousands of mutants [33]. However, a similar process cannot be applied directly to testing neural code models because there are no explicit oracles (i.e., program crashes) available to evaluate the outcome of a seed program that has been mutated multiple times [14]. To bypass the oracle challenge in testing neural code models, we employ metamorphic testing, similar to the approach of Tian et al. [31] If the transformation is semantically preserving, the mutated and original programs share the same oracle. Note that mutating a program may impair its naturalness, as the inserted or mutated code may serve as noise for the original program. Therefore, we use a threshold MAX to limit the maximum number of mutations applied to a seed program, meaning a seed program can be mutated a total of MAX times. Note that the design of the MAX controller does not guarantee that the inserted mutation code is natural. A robust code model should be able to disregard noise code regardless of its naturalness.

To determine an appropriate value for MAX, one thousand sample programs are selected at random from the test datasets of the three neural code models under investigation (see section IV-A for details). Random MAX mutation operators are selected for each sample and applied to each instance to generate mutants. We experiment with MAX values ranging from $1to10$. The generated code is assumed to be completely unnatural in the most conservative scenario. We determine the naturalness of the mutated program based on the proportion of code that was generated.

Figure 2 shows the average percentage of code generated against the seed program for varying MAX values. The naturalness of the code decreases dramatically as MAX increases. For instance, when MAX is equal to 1, only 9.25% of the code in a mutated program is noise (i.e., generated

code), whereas when MAX is increased to 10, the average noise code percentage is 59.60%.

Prior research demonstrated that there is approximately 28% noise code in real-world software projects [34]. Consequently, for the remainder of the experiments in this paper, We set MAX to three, which reduces average noise by 30%.

In terms of syntactical constraints, mutation operators are designed so that the mutated program is semantically equivalent to the original. For a single mutation operation, mutation operators only add dead code to the program, so that the execution of the mutated program is identical to that of the original. Note that, in this work, mutation operations are orthogonal to one another, which ensures that the accumulation of mutations in an original program does not alter its execution. For example, a dead branch operator does not utilize the unused variable created by other operator (see Section III-C).

B. Neuron Coverage Analysis

Algorithm 1 describes how `CoCoFuzzing` searches for new test data using neuron coverage (line 5—25). In each iteration (line 10–18), `CoCoFuzzing` evaluates all mutation operators and identifies the mutated program that activates the most number of new neurons. This mutated program is generated as one set of test data (line 24) and also saved for the subsequent mutation iteration. Controlled by the threshold *MAX* (line 5), this mutant could be continuously mutated in the subsequent iteration until the empirically determined threshold of three mutations per original seed program is reached.

Two methods *neuronActivation* and *newNeurons* in Algorithm 1 are for neuron coverage analysis. The *neuronActivation* method uses input and a trained neural model to generate a set of activated neurons as the output. For instance, two listeners are injected to monitor the neurons of a trained PyTorch model. Then, using the injected listeners, we collected the output of each neuron and ran the inference or prediction of the trained model using the input. Similar to prior work [14], the value of each neuron is scaled to the range of 0 to 1 and compared to a threshold for each neuron. If the scaled output is greater than the threshold (i.e., 0.4, the same value as in [31]), the neuron is considered active; otherwise, it is considered inactive.

The *newNeurons* method is very similar to *neuronActivation*, which takes a model, one input data, and a set of activated neurons (i.e., NC_p in line 12) as the input. The method *newNeurons* computes the activated neurons based on the input and returns a set of newly activated neurons based on the input but not NC_p. The neuron coverage analysis implementation in `CoCoFuzzing` is model-independent and can be broadly applied to a variety of deep learning models.

C. Mutation Generation

In `CoCoFuzzing`, we adopt and implement specialized mutation operators for programming mutations. To circumvent the oracle problem, we propose mutation operators that maintain program semantics. Therefore, the test oracles of the mutated programs should be identical to those of the original programs, i.e. metamorphic testing. In testing neural models, previous research [14] has proposed mutation operators for image and audio inputs, such as image scaling, image rotation, noise addition, etc. As source code programs must adhere to rigid lexical, grammatical, and syntactical constraints, these operators cannot be directly implemented.

In this paper, we propose a set of ten mutation operators that can be used to generate semantically equivalent methods. Mutation operations range from common refactorings, such as variable renaming, to more intrusive ones, such as adding unreachable branches. Table I provides a summary of the ten mutation operators and how each operator can be utilized to modify a program. Notably, among the ten operators, *Op1* and *Op10* have been proposed and experimented in prior studies [17] to generate adversarial examples for neural code models. In this work, the other eight mutation operators are used for the first time. Various types of program transformations are targeted by these mutation operators. The open-source java parser package javalang[2] Given a program T and a mutation operator O, we first use the Abstract Syntax Tree (AST) parser from the javalang package to convert T to a list of sub-ASTs, with each element representing the AST representation of a statement. The algorithm then iterates through the sub-AST list and identifies potential positions for O based on its specific transformation requirements. The following outlines the specific mutation generation process for each operator.

- **(Op1) Dead store [18]:** inserts an unused variable declaration with one primitive type (e.g., string, int, double, and long, etc.) to a randomly selected basic block in the program. The name of the variable is a string of eight characters randomly generated in the form of [a-z]. Only one dead store is added in each transformation by this operator.
- **(Op2 and Op3) Obfuscating:** rewrites a numerical value or variable and its usages in a statement by adding and deleting the same random numerical value of the same type. For example, $x = 1.0;$ can be mutated to $x = 1.0 + 0.1 - 0.1;$ or $x = 1.0 + 0 - 0;$. If one program contains more than one numerical variable, we randomly pick one to perform the transformation. This operator only works on assignment, declaration, or return statements.
- **(Op4) Duplication:** duplicates a randomly selected assignment statement and inserts it immediately after its current location. To avoid side effects, the applicable assignment statement is limited to the ones without using method invocation.
- **(Op5 to Op9) Unreachable loops/branches:** inserts an unreachable loop or branch (including *if* statement, *for* statement, *while* statement, and *switch* statement) into a randomly selected basic block in the program. The condition of the inserted loop or branch is always false to make it unreachable.
- **(Op10) Renaming [17], [18]:** renames a local variable declared in a program. If there exist multiple variables, we randomly select one for the mutation. The new name of the variable will be in the form of [a-z].

[2]https://pypi.org/project/javalang/ is used for code parsing and tokenization.

TABLE I: Ten Semantic-Preserving Mutation Operators Applied in CoCoFuzzing.

NO.	Operator name	Description
Op1 [18]	dead store	Inserting unused variable declarations
Op2	numerical obfuscating	Obfuscating the numerical variables via adding/deleting a same numerical value
Op3	adding zero	Obfuscating the numerical values via adding zero
Op4	duplication	Duplicating assignment statements
Op5	unreachable *if*	Inserting unreachable *if* statements
Op6	unreachable *if-else*	Inserting unreachable *if-else* statements
Op7	unreachable *switch*	Inserting unreachable *switch* statements
Op8	unreachable *for*	Inserting unreachable *for* statements
Op9	unreachable *while*	Inserting unreachable *while* statements
Op10 [17], [18]	renaming	Renaming user-defined variables

Note that some of the proposed operators, such as *Op1* and *Op5–Op9*, can be inserted into any locations in a given program. In this work, we randomly pick a location to apply these mutation operators because our experiments indicate that there is no statistically significant correlation between the chosen location and the effectiveness of these mutation operators (details are in Section VI-A).

IV. Experiment Setup

We conduct all experiments using the Google Cloud Platform. The hardware of our experiment machine is comprised of two n1-highmem-2 virtual central processing units (vCPU) with 13-gigabyte memory in total and one NVIDIA Tesla T4 GPU. We use PyTorch 1.4 for running NeuralCodeSum, TensorFlow 1.15 for CODE2SEQ, and TensorFlow 2.1 for CODE2VEC.

A. Subject Models and Datasets

1) Studied Models: In this work, three state-of-the-art neural code models that adopt different neural network characteristics, i.e., NeuralCodeSum [3], CODE2SEQ [1], and CODE2VEC [2] are used to evaluate CoCoFuzzing. For our experiments, we employ the pre-trained models of the three neural code models studied, which are made available in their respective papers. In learning code embedding from large-scale data and generating a high-level summary given a method, all three models are similar. However, the three models are distinct in numerous ways. They utilize distinct model architectures first. Second, their representations of code differ. Thirdly, the level of detail in the summaries generated by the three models is distinct; NeuralCodeSum generates a complete English sentence as output, whereas CODE2SEQ and CODE2VEC produce the name of the method. We briefly describe the details of the three models below.

NeuralCodeSum [3] uses Transformer [3] (comprised of stacked multi-head attention and parameterized linear transformation layers for both the encoder and decoder) to generate a natural language summary from a source code snippet. Both the code and the summary consist of token sequences represented by vector sequences. To allow the Transformer to utilize the order information of source code tokens, NeuralCodeSum encodes both the absolute position and pairwise relationship of source code tokens.

CODE2SEQ [1] uses an encoder-decoder architecture to encode paths node-by-node and generate label as sequences. In CODE2SEQ, the encoder represents the body of a method as a set of AST paths, with each path compressed to a fixed-length vector using a bi-directional LSTM that encodes paths node-by-node. The decoder employs attention to choose relevant paths during decoding and predicts sub-tokens of the target sequence at each step when generating the method's name.

CODE2VEC [2] proposes a path-based attention model for learning vectors for arbitrary-sized snippet of code. The model allows the embedding of a program into a continuous space. Specifically, it first extracts syntactic paths from within a code snippet, i.e., ASTs, and then it represents them as a bag of distributed vector representations. Then, using an attention network, a learned weighted average of the path vectors will be computed in order to generate a single code vector.

TABLE II: Experimental datasets. 'Pro' is the number of projects.

Model	Language	#Pro	#Training	#Validation	#Test	#Method
NeuralCodeSum	Java	9.7k	69.7k	8.7k	8.7k	87.1K
CODE2SEQ CODE2VEC	Java	11	692.0k	23.8k	57.1k	772.9K

2) Dataset: We perform our experiments using the original datasets associated with each of the three studied neural code models. Table II lists the basic statistics of the three datasets. Specifically, the dataset of NeuralCodeSum contains 9.7K open-source Java projects hosted in GitHub and each of the projects has at least 20 stars. 87.1k methods that have JavaDoc comments were collected.

CODE2SEQ created three Java datasets, which are different in size, the popularity of the open-source software where the data is from, and the distribution of training, validation, and testing. Among the three Java datasets by CODE2SEQ, we decided to use one, i.e., *Java-small*. Despite its relatively small size among the three, it is commonly used by prior studies [2], [5], including CODE2VEC. Also, all the projects in *Java-small* are large-scale and mature open-source software, in comparison with the other two datasets (i.e., many are from smaller and less popular projects).

Java-small contains 11 relatively large Java projects 9 for training, 1 for validation, and 1 for testing. Overall, it contains about 772.9K methods.

B. Evaluation Metrics

We use the same evaluation metrics with the original papers of NeuralCodeSum, CODE2SEQ, and CODE2VEC for comparison purpose. NeuralCodeSum adopts BLEU [35] to evaluate its performance on predicting the summarization of programs. bilingual evaluation understudy(BLEU) is widely used to assess the quality of machine translation systems [36]. BLEU's output is always a number between 0 and 1. This value indicates how similar the predicting summarization is to the ground truth, with values closer to 1 representing higher similarity.

Differently, CODE2VEC and CODE2SEQ used precision, recall, and F1 for measuring performance. Precision and Recall are computed on a per-subtoken basis. F1 is the weighted average of Precision and Recall.

C. Baseline

1) Baseline for Mutation Operators: As two of the ten mutation operators are used in prior studies [17], [18], namely *Op1* (i.e., dead code inserting) and *Op10* (i.e.,renaming). We treat them as baselines to evaluate the eight new mutation operators, i.e., *Op2– Op9*.

2) Baseline for NC-Guided Test Generation: `CoCoFuzzing` uses neuron coverage to guide the test generation via the combination of mutation operators. To evaluate the effectiveness of `CoCoFuzzing` we have also designed a baseline approach that generates tests without neuron coverage guidance, i.e., *Random@K*, which randomly picks K mutation operators to generate mutants as tests. K indicates the maximum number of mutations tries on a seed program. As described in Section III, `CoCoFuzzing` limits the maximum number of mutation tries to three, thus in our experiment we also set K to three. Given a seed program, *Random@3* first randomly selects a mutation operator op_i and mutate the sample to get the mutant $M1$. Then, it mutates $M1$ with another randomly selected mutation operator op_j to get a new mutant $M2$, after that it further mutates $M2$ with a randomly selected mutation operator op_h to generate a new mutant M3. Through the process, op_i, op_j, and op_h can be the same operator. *Random@3* produces a total of 3 mutants for a given seed program.

D. Research Questions

We have implemented `CoCoFuzzing` as a self-contained fuzz testing framework in Python based on deep learning framework Keras, TensorFlow, and Pytorch. With `CoCoFuzzing`, we perform a large-scale comparative study to answer the following four research questions.

RQ1: *Are the neural code models robust against simple perturbations?* Robustness has been extensively studied in classic deep learning application domains, e.g., image processing, speech recognition, and Natural Language Processing (NLP). Numerous of these deep learning models have been shown to be susceptible to simple perturbations. As the first study to examine the robustness of neural code models, this RQ investigates whether they are susceptible to the same issues.
RQ2: *What is the effectiveness of each mutation operator?*

This RQ illustrates the effectiveness of each mutation operator (a total of ten in Table I) regarding its capability of introducing perturbations that can affect the performance of neural code models and activate different neurons. Furthermore, this RQ also examines the effectiveness of the two baseline operators (i.e., *Op1* and *Op10*) in comparison to the other eight new mutation operators.

RQ3: *What is the effectiveness of the NC-guided test generation in `CoCoFuzzing`?*

`CoCoFuzzing` uses neuron coverage to guide the search for new tests that are continuously transformed through multiple mutation operators. This RQ explores the performance of `CoCoFuzzing` and compares it with our constructed baseline, i.e., *Random@3*.

TABLE III: The results of testing the three neural code models on the test data before (i.e., *1K original*) and after (*1K mutants*) introducing simple perturbations.

Model	Test data	Performance (%)
NeuralCodeSum	*1K original*	BLEU = 40.82
	1K mutants	BLEU = 12.46
CODE2SEQ	*1K original*	F1 = 71.16
	1K mutants	F1 = 66.96
CODE2VEC	*1K original*	F1 = 47.68
	1K mutants	F1 = 45.56

RQ4: *Is `CoCoFuzzing` useful for improving neural code models?*

This RQ explores whether the synthetic programs can improve the neural code models, i.e., whether retraining with `CoCoFuzzing`'s synthetic programs can make these models more robust.

V. Result Analysis

This section shows the result analysis for answering the research questions asked in Section IV-D.

A. RQ1: Robustness of Neural Code Models

Approach. To answer this question, we re-use the original test sets of each model as a start point of the experiment.

Specifically, for each studied neural code model, 1,000 samples are randomly selected from its original test dataset. For each sample, one of the ten mutation operators is randomly selected from Table I and apply the mutation operator to the sample to generate a test. Note that, given a sample program, it is possible that some mutation operators are not applicable, e.g., *Op4*-duplication cannot be applied if a program does not have any assignment statements. If this happens, another mutation operator is randomly selected until one mutated program is generated. Based on our experiments, at least one of the ten mutation operators can be applied to any of the samples. Hence, we have 1,000 mutated programs from the 1,000 samples for each neural model. Then the three pre-trained neural models with the 1,000 sample (i.e., *1k original*) and the generated 1,000 mutants (i.e., *1k mutants*) are evaluated.

Result Analysis. Table III shows the impacts of the simple perturbations on the performance of the neural models. In particular, we show the performance (either BLEU or F1) of the studied neural code models under two test datasets, i.e., before and after perturbations. For all the three neural code models, we notice the performance on *1k mutants* decreases compared to *1k original*. Specifically, for NeuralCodeSum, the BLEU score reduces 69.5% (from 40.82 to 12.46). The F1 scores of CODE2SEQ and CODE2VEC reduce 5.9% (from 71.16% to 66.96%) and 4.44% (from 47.68% to 45.56%) respectively.

Although all three neural code models are susceptible to semantic-equivalent transformations; however, the impact of simple perturbations on the performance differs across the three neural models.

Conclusion. As we can see the performance of NeuralCodeSum has declined significantly compared to CODE2SEQ

TABLE IV: Performance of different operators on NeuralCodeSum, CODE2SEQ, and CODE2VEC. 'Original' shows the *1k original* test set. 'Op1'–'Op10' represent the test sets with the perturbations introduced by one mutation operator respectively. Numbers in the brackets are the performance decline of the examined neural models on the mutated test set compared with the original test set.

	NeuralCodeSum BLEU (%)	**CODE2SEQ** F1 (%)	**CODE2VEC** F1 (%)
Original	40.82	71.16	47.68
Op1	8.59 (78.95%↓)	68.23 (4.29%↓)	45.32 (5.05%↓)
Op2	8.88 (78.26%↓)	70.75 (0.58%↓)	47.50 (0.23%↓)
Op3	8.84 (78.34%↓)	70.70 (0.65%↓)	47.15 (0.97%↓)
Op4	9.29 (77.21%↓)	70.85 (0.43%↓)	47.10 (1.08%↓)
Op5	**6.02 (85.25%↓)**	60.57 (17.49%↓)	44.73 (6.32%↓)
Op6	6.16 (84.90%↓)	**58.65 (21.33%↓)**	44.78 (6.32%↓)
Op7	6.23 (84.73%↓)	59.82 (18.96%↓)	46.13 (3.21%↓)
Op8	7.19 (82.38%↓)	64.67 (10.04%↓)	43.81 (8.69%↓)
Op9	7.16 (82.45%↓)	63.72 (11.68%↓)	**41.04 (16.01%↓)**
Op10	37.08 (9.16%↓)	71.06 (0.14%↓)	47.49 (0.25%↓)

TABLE V: The average neuron coverage of the test sets generated using different mutation operators. Numbers in the brackets are the average number of newly activated neurons.

	NeuralCodeSum	**CODE2SEQ**	**CODE2VEC**
Original	44.94%	90.79%	60.99%
Op1	**47.33%** (722.80)	**91.46%** (39.08)	**61.48%** (19.13)
Op2	**47.07%** (702.65)	90.73% (16.72)	**61.00%** (32.90)
Op3	**47.06%** (702.32)	90.76% (11.17)	60.98% (25.12)
Op4	**45.75%** (700.31)	90.68% (19.80)	**61.17%** (31.00)
Op5	**47.55%** (745.05)	88.61% (55.83)	**62.37%** (50.82)
Op6	**47.49%** (742.95)	89.53% (55.44)	**62.33%** (44.40)
Op7	**47.48%** (741.40)	88.98% (53.99)	**62.04%** (41.91)
Op8	**48.11%** (768.31)	**91.36%** (52.39)	**62.66%** (48.44)
Op9	**48.10%** (772.08)	**91.50%** (53.21)	**62.70%** (49.41)
Op10	44.91% (459.86)	**90.86%** (21.82)	60.97% (25.17)

and CODE2VEC. The main reason is that NeuralCodeSum uses one token sequence to represent the entire body of a method and most of our proposed mutation operators can introduce new tokens into the method body, which impacts the representation vector and may further impact the performance of the model. While CODE2SEQ and CODE2VEC use both AST paths and AST token information to represent the entire body of a method, which are more stable than the token-based representation of NeuralCodeSum. Thus these two models are less susceptible to the perturbation introduced by random mutation.

Answer to RQ1: The studied neural code models face robustness issues as their performance is negatively impacted by simple perturbations to test data. However, the negative impacts vary due to the use of different representations of code by each neural code model.

B. RQ2: Comparison Across Different Mutation Operators

Approach. To answer this question, we use the same *1K original* test sets for the three neural code models collected in RQ1 (Section V-A). For each mutation operator in Table I, we apply it on the *1K original* test dataset to generate new test data. In total, 10 new test sets are generated, i.e., each test set is generated by applying one particular mutation operator. Then the performance of the three neural code models are evaluated on each of the new test sets.

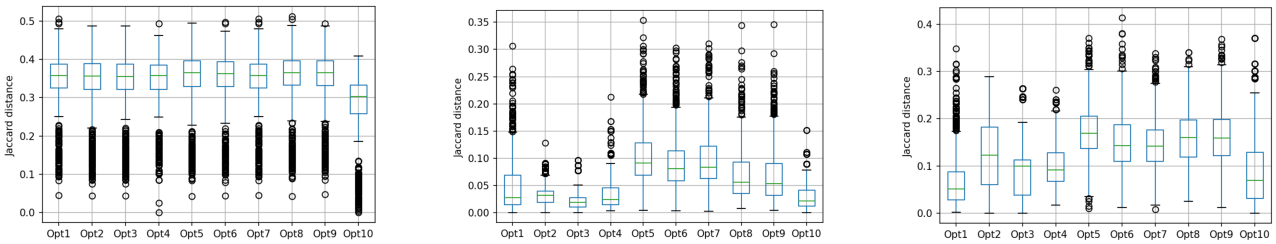
We investigate the neuron coverage differences between each new test set and the original test set. In particular, we compute the neuron coverage of each set of test data and then the average neuron coverage of each set of test data. Additionally, we calculate the average number of newly activated neurons in each test set based on a pairwise comparison between original test data and mutated data, i.e. the activated neurons in mutated data that are not activated by original data.

In addition, we examine the difference between the two sets of activated neurons, one activated by the original test set and the other by the new test set with mutations. For each test set, we calculate the Jaccard distance between each mutant sample and the original sample. Given two sets of neurons $N1$ and $N2$ respectively. We measure their Jaccard distance by $1-\frac{N1\cap N2}{N1\cup N2}$. The Jaccard distance can have a value between 0 and 1, with 1 indicating no overlap between the two sets and 0 indicating complete overlap.

Result Analysis. ***Comparison Across the Ten Mutation Operators***. Overall, from the results shown in Table IV, we observe that the performance of each model decreases on each of the 10 new test sets, i.e., the performance decline of NeuralCodeSum ranges from 9.16% (*Op10*) to 85.25% (*Op5*) and the performance decline of CODE2SEQ and CODE2VEC ranges from 0.14% (*Op10*) to 21.33% (*Op6*) and 0.23% (*Op2*) to 16.01% (*Op9*) respectively.

We investigate further the variance in performance decline among the ten operators. Compared to the other operators, operator ten has the least effect on the NeuralCodeSum model. This is because the NeuralCodeSum model's prediction depends not only on the tokens in the input but also on the hidden ordering information in the input sequence. Since the new code is inserted in the middle of the original sequence, this information is lost following mutation operations one through nine. Because it only replaces one or two tokens in the sequence, rather than introducing a new sequence of tokens into the input, operator ten has the least impact. We also observe that the CODE2SEQ and CODE2VEC models are relatively resistant to Op2 (numerical obfuscation), Op3 (adding zero), Op4 (duplication), and Op10 (numerical duplication) (renaming). This is due to the fact that the aforementioned mutation operations have minimal impact on the input of the two models. The inputs of the two models are AST paths, and the aforementioned operations only modify one or two tokens or duplicate one path of the inputs, which has a smaller impact than other mutation operators that introduce new AST paths.

In addition, we can also see that the most effective mutation operators for each model are different, e.g., *Op5* is the most effective mutation operator for NeuralCodeSum, while for CODE2SEQ and CODE2VEC, the most effective mutation operator is *Op6* and *Op9* respectively. We further conduct the Mann-Whitney U test ($p < 0.05$) to compare the performance of the three neural code models under test data with (i.e., *Op1*–*Op10*) and without perturbations (i.e., original test set). The results suggest that the performance decline caused by simple perturbations is statistically significant in all the ten test sets and for all the three studied neural code models.



(a) NeuralCodeSum (b) CODE2SEQ (c) CODE2VEC

Fig. 3: Difference in neuron coverage caused by different mutation operators in the three models.

Table V shows the comparison results across different mutation operators with regards to the impact on neuron coverage. Overall, the impact of each operator on neuron coverage varies across the studied neural code models. For example *Op2–Op7* significantly improve the neuron coverage on NeuralCodeSum, while decreasing the neuron coverage on CODE2SEQ. This may be caused by the unique architecture of different models and properties of the test data. Interestingly, despite the decreased neuron coverage, we notice that all the mutation operators can activate new neurons compared to the original test sets. This supports our design choice of using newly activated neurons instead of neuron coverage when guiding the search in fuzzing (line 12, Algorithm 1).

Figure 3a, Figure 3b, and Figure 3c show the distribution of the Jaccard distance between the neuron coverage caused by each of the ten test datasets and the *1k original* test data for NeuralCodeSum, CODE2SEQ, and CODE2VEC respectively. From these figures, we can see that the Jaccard distances vary for different operators and these results also confirm that different mutation operators activate different neurons at different rates.

TABLE VI: Comparison results across original test sets, newly generated test sets using Random@3 and NC-guided (`CoCoFuzzing`) strategies. NC denotes Neuron Coverage and JD is the Jaccard Distance.

Model	Strategies	#New Tests	Performance (%)	NC(%)	JD
NeuralCodeSum	*1k original*	-	BLEU = 40.82	44.94	-
	Random@3	3,000	BLEU=8.59 (78.95%↓)	47.39	0.29
	NC-guided	2,906	BLEU=6.20 (84.81%↓)	48.95	0.32
CODE2SEQ	*1k original*	-	F1 = 71.16	90.79	-
	Random@3	3,000	F1=63.36 (10.96%↓)	95.15	0.05
	NC-guided	2,969	F1=55.46 (22.06%↓)	95.71	0.08
CODE2VEC	*1k original*	-	F1 = 47.68	60.99	-
	Random@3	3,000	F1=42.93 (9.96%↓)	72.24	0.17
	NC-guided	3,000	F1=34.53 (27.58%↓)	75.23	0.25

Op2–Op9 v.s. Op1 and Op10. As we described in Section III-C, operators *Op1* and *Op10* are proposed and used in prior studies [17], [18] to generate adversarial examples for neural code models. Regarding the effectiveness on reducing the performance of a neural model for code, five (i.e., *Op5–Op8*) of the eight operators (i.e., *Op2–Op9*) can outperform *Op1* and all can outperform *Op10* on NeuralCodeSum. We can also observe similar results on CODE2SEQ and CODE2VEC, i.e., five and four of the new proposed eight mutation operators can outperform *Op1* respectively and *Op10* is worse than all the eight new operators.

Conclusion. In this work, we use all the ten mutation operators for the mutation-based test case generation, as each operator represents a unique type of semantic-preserving transformations, which may trigger a different part of the examined neural code models, their different neuron coverage also confirms this.

Answer to RQ2: All the ten mutation operators are shown to be effective in introducing perturbations that can significantly impact the performance of the studied neural code models. Our detailed analysis reveals that a mutated test set often activates a different set of neurons compared with the original test set. Last, the eight new mutation operators (i.e., *Op2–Op9*) are comparable or outperform the two operators (*Op1* and *Op10*) used in prior studies.

C. RQ3: Effectiveness of `CoCoFuzzing`

Approach. We reuse the *1k original* test data collected in RQ1 (Section V-A) to explore the performance of the NC-guided mutation generation in `CoCoFuzzing`. For each test program in *1k original* dataset, we use `CoCoFuzzing` and the baseline approach *Random@3* to generate new test data respectively. Note that, the number of new test data generated by `CoCoFuzzing` has an upper bound, i.e., three times of the seed programs (see details in Section III-A). The actual generated test set may contain less than the upper bound as the test data that does not activate new neurons is discarded. Meanwhile, *Random@3* generates three new mutants for each seed program and yields a total of 3,000 generated programs. Thus the number of generated mutants of these two approaches might not be the same. Then we examine the performance of the three models on the two new test sets (i.e., `CoCoFuzzing` and *Random@3*) respectively. Last, we calculate the average neuron coverage of each test set and the average Jaccard distance between each mutated test data and its original test data (i.e., one seed program in the *1k original* test set). **Result Analysis.** As we can see from the results in Table VI, both `CoCoFuzzing` and *Random@3* can generate new tests that detect more classification errors on the neural code models. With the newly generated tests, the performance of each examined model decreases significantly and the decline rate can be up to 84.81% (i.e., NC-guided on NeuralCodeSum). In terms of neuron coverage, the NC-guided strategy (`CoCoFuzzing`) achieves a lightly higher neuron coverage than Random@3 and *1k original*.

TABLE VII: Model retrain scenarios and the corresponding performance. **TrData** indicates the randomly selected 10K training data from the original training dataset. **TrData+CoCoFuzzing** indicates an enhanced training dataset by combining **TrData** and the synthetic inputs generated by applying CoCoFuzzing on **TrData**. **TrData+Random** indicates an enhanced training dataset by combining **TrData** and the synthetic inputs generated by applying Random@3 on **TrData**.

Model	Training data	Performance (%)
NeuralCodeSum	TrData	BLEU = 16.50
	TrData+Random	BLEU = 20.32
	TrData+CoCoFuzzing	**BLEU = 22.30**
CODE2SEQ	TrData	F1 = 22.98
	TrData+Random	F1 = 23.16
	TrData+CoCoFuzzing	**F1 = 25.01**
CODE2VEC	TrData	F1 = 11.54
	TrData+Random	F1 = 15.13
	TrData+CoCoFuzzing	**F1 = 15.48**

Conclusion. Compared with the 10 test sets by applying each mutation operator individually (Table V), the NC-guided strategy also achieves the highest neuron coverage. Based on Jaccard distance, on average, the NC-guide strategy generates a new test set that activates more neurons compared to *Random@3*, which suggests that the neuron coverage metric can provide positive and meaningful guidance to the CoCoFuzzing.

Answer to RQ3: By utilizing coverage-guided fuzzing strategy, CoCoFuzzing is more effective in testing neural code models than the baseline *Random@3*, i.e., a lower BLEU or F1 value, a higher neuron coverage, and a higher ratio of newly activated neurons.

D. RQ4: Usefulness of CoCoFuzzing on Improving Models

Approach. Similar to Deeptest [31], as a proof-of-concept, we showcase the usefulness of the mutated test data by using a subset of the entire training data. In particular, the three neural models are trained with 10k randomly selected original training data from scratch.The validation dataset provided by each of the three models are used during the training process.

Then for each neural code model, CoCoFuzzing and *Random@3* are applied on the selected 10K training data to generate two new test sets. Combining the mutated test sets and the 10K randomly selected training data, two sets of enhanced training datasets are collected, i.e., one enhanced by CoCoFuzzing and one by *Random@3*. We then re-train each of the three models with the two enhanced training datasets respectively. Finally, we evaluate these re-trained models on the *1K original* test dataset.

Result Analysis. Table VII compares the performance across three types of training sets, i.e., original data, original data enhanced by Random@3 strategy, and original data enhanced by CoCoFuzzing. As we can see from the table, in all cases, the performance of the re-trained model improved significantly over the original model and the improvements are 35.15% on NeuralCodeSum, 8.83% on CODE2SEQ, and 34.14% on CODE2VEC.

Conclusion. Compared with the original models, all three models can be improved by retraining the models with synthetic data generated by CoCoFuzzing.

Answer to RQ4: Performance of the three neural code models can be improved 35.15% on NeuralCodeSum, 8.83% on CODE2SEQ, and 34.14% on CODE2VEC by retraining the models with synthetic data generated by CoCoFuzzing.

VI. Discussion

This section discusses open questions regarding the effectiveness of CoCoFuzzing.

A. Impact of the Applied Locations of Mutation Operators

Some of the mutation operators in CoCoFuzzing can be applied in any location of a given program (i.e., location independent). For example, *Op1* inserts an unused variable declaration into a randomly selected basic block of in a program, thus for any non-empty program, there exists more than one location for *Op1*. Among the ten mutation operators listed in Table I, *Op1* and *Op5–Op9* are location independent. To better understand the impact of this randomness on the performance of CoCoFuzzing, for each location independent operator, we apply it on the *1K original* test dataset to get a new test dataset. Then we collect the performance of the three models on the new test datasets. We rerun the above process 10 times and calculate the standard deviation values and the distribution of the performance of the three neural models. Table VIII shows the detailed results.

We find that, for all the location-independent operators, their effectiveness on the neural code model is insensitive to the applied locations, i.e., the variance is small. Our One-Way ANOVA test results show that there is no significant difference in the performance of the 10 runs, which suggests that the locations of generated mutants do not significantly impact the effectiveness of CoCoFuzzing. Thus, in our experiments, we randomly pick a location to apply these mutation operators.

B. Distribution of the Selected Mutation Operators

CoCoFuzzing adopts a neuron coverage guidance algorithm to generate new tests with the ten pre-defined mutation operators. To further understand the operator selection process in CoCoFuzzing, we collected the selected operator for each mutant generated by CoCoFuzzing for each of the three neural code models, i.e., NeuralCodeSum, CODE2SEQ, and CODE2VEC. Table IX shows the percentage of each operator used among the tests generated by CoCoFuzzing on each model. Overall, we can see that distribution of the selected operators varies dramatically among different neural code models, e.g., *Op10* was used among 5% of all the generated tests in NeuralCodeSum while less than 1% of the generated tests from CODE2SEQ used *Op10*. In addition, we can also see that some of the operators are dominating across different models e.g, *Op5* has been used in more than 15% of the generated tests on each model. While these also

TABLE VIII: Statistics of the impact of the position independent mutation operations used in this study. **Performance Distribution** indicates the distribution of the performance of a model with 10 different test datasets generated by a mutation operator. **SD** is the standard deviation of the performance of a model with different test datasets.

Model	Operator	Performance Distribution Average (± Range)	SD
NeuralCodeSum BLEU(%)	Op1	7.10 (±0.28)	0.16
	Op5	5.78 (±0.43)	0.27
	Op6	5.88 (±0.38)	0.23
	Op7	5.92 (±0.40)	0.24
	Op8	6.50 (±0.00)	0.00
	Op9	6.61 (±0.00)	0.00
CODE2SEQ F1(%)	Op1	65.05 (±1.47)	0.01
	Op5	63.98 (±1.02)	0.006
	Op6	62.09 (±1.56)	0.009
	Op7	64.02 (±1.26)	0.008
	Op8	64.65 (±1.34)	0.008
	Op9	65.66 (±1.14)	0.008
CODE2VEC F1(%)	Op1	60.51 (±2.46)	0.02
	Op5	59.95 (±2.01)	0.01
	Op6	57.81 (±2.44)	0.01
	Op7	60.34 (±1.29)	0.009
	Op8	58.42 (±2.14)	0.01
	Op9	54.62 (±1.91)	0.01

TABLE IX: The distribution of selected mutation operators on the three models.

Operator	NeuralCodeSum	CODE2SEQ	CODE2VEC
Op1	9.05%	6.46%	0.87%
Op2	8.39%	10.83%	15.30%
Op3	4.43%	2.13%	2.63%
Op4	2.82%	0.63%	1.23%
Op5	**15.79%**	**19.00%**	**26.07%**
Op6	14.83%	14.83%	11.03%
Op7	13.97%	9.13%	5.53%
Op8	12.73%	18.33%	16.67%
Op9	13.21%	16.50%	19.37%
Op10	4.74%	1.10%	1.30%

exist in the operators that are selected less frequent across the three models, i.e., *Op1*,*Op3*, *Op4*, and *Op10* are used in less than 10% of the generated test cases across the three models. One of the possible reasons is that *Op5* activates relatively more new neurons than other operators (As shown in Table V). We have conducted a Spearman rank correlation to compute the correlation between the number of newly activated neurons of mutants generated by an operator and the frequency of an operator used in a neural code model. The Spearman correlation values are 0.78 in NeuralCodeSum, 0.68 in CODE2SEQ, and 0.92 in CODE2VEC, which indicates that the frequency of an operator used in a neuron code model is positively correlated with its ability to activate new neurons (compared to the original tests).

C. Cost effectiveness

In `CoCoFuzzing`, a mutated program extends the original program up to three times. `CoCoFuzzing` applies each mutation operator to the original program and selects the most efficient operator based on the neuron coverage in each mutation. Thus, for each original program, `CoCoFuzzing` executes and evaluates all eleven operators three times, resulting in a complexity of $O(n)$ where n is the number of inputs. In our experiment, we observe that 1,000 testing examples take approximately 10 hours to execute. It is equivalent to an average of 36 seconds per test case.

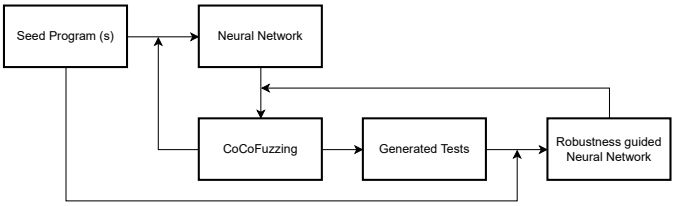


Fig. 4: The re-training pipline of our proposed `CoCoFuzzing`.

D. Apply to New Model

Figure 4 shows the retraining pipeline of using `CoCoFuzzing` for robustness-guided training of a new model. Specifically, there are three steps to apply `CoCoFuzzing` to a given new model M. First, inject the neuron activation listener into the activation layer of the model, which is usually the second-to-last layer. Second, running the model with `CoCoFuzzing` and collecting the mutated dataset generated. Third, combining the training dataset and the mutated dataset into a new training dataset for re-training. The effectiveness of robustness-guided training with `CoCoFuzzing` can be enhanced by repeatedly performing steps two and three.

E. Limitations of `CoCoFuzzing`

One limitation of `CoCoFuzzing` is that the ability of error discovery significantly depends on the mutated input, which may be less effective than a mathematical approach such as gradient fuzzing [30]. In addition, the performance of mutation testing also relies on the human-designed mutation operators. In this work, we design `CoCoFuzzing` to show that the existing code generation model is not robust against dead code mutation test rather than exhaustively discovering malfunctions of the code models, we keep the task of defining optimal mutation operators as our future work.

Another limitation of `CoCoFuzzing` is that neuron coverage may not be the best metric for operator selection. Recent study [37] shows that the neuron coverage metric rises from 0 to close to 100% by giving only 25 examples. The neuron coverage metric is quickly saturated and the difference of activation between each testing example decreases dramatically, which means that only the first few testing examples are meaningful to the metric. However, `CoCoFuzzing` only evaluates the neuron coverage across three accumulative mutations, which means the impact of metric saturation is minor to our study.

F. Threats to Validity

The selection of the studied neural code models and experimental projects could be a threat to validity. In this work, we only evaluated `CoCoFuzzing` on NeuralCodeSum, CODE2SEQ, and CODE2VEC with Java programs. Therefore,

TABLE X: Estimated execution time of CoCoFuzzing on different testing datasets.

	CoCoFuzzing	Java-small	Java-med	Java-large
data - training	10,000	665,115	3,004,536	15,344,512
data - test	1,000	56,165	411,751	417,003
training - Code2Vec	4 hrs	11 days*	50 days*	255 days*
training - Code2Seq	6 hrs	16.5 days*	75 days*	383.5 days*
training - NeuralCodeSum	10 hrs	27.7 days*	125 days*	639.3 days*
mutation - Code2Vec	1 day	56 days*	411 days*	411 days*
mutation - Code2Seq	1 day	56 days*	411 days*	411 days*
mutation - NeuralCodeSum	6 hrs	14 days*	103 days*	104 days*

our results may not generalize to other neural code models or other programming languages. We leave the evaluation of the general applicability of our approach as future work. To generate new tests, this paper adopts ten different mutation operators for program transformations. Although these mutation operators have been examined can help generate effective mutants, these mutation operators may not represent many possible transformations for software programs. In addition, most of the used mutation operators are designed for object-oriented programming languages, e.g., Java, which might not work for other program languages.

The naturalness of the mutated source code can be a threat to validity as well. In our experiments (Figure 2), we observe that the percentage of the dead code in the mutated program increases as we mutate the program multiple times, thus we need a trade-off between the naturalness of the source code and the number of mutants inserted. The existing study found that on average there are 28% dead code in the open-source projects [34]. To make the mutated program close to the statistics of natural programs, we set the controller MAX equal to 3, which limits the percentage of dead code to less than 30% in our experiments on average. However, the optimal value for MAX could be different for different projects or projects with different programming languages.

Table X shows the data size and mutation time cost of Java-subset used in this work (CoCoFuzzing), Java-small, Java-mid and Java-large. The statistics of data size are from CODE2SEQ [1]. The time cost labeled with * is the estimation execution time. The time cost of CoCoFuzzing on Java-subset is calculated based on the actual execution time on Google Cloud Platform, from where we bought computation power to conduct our experiments. We estimate the execution time of Java-small, Java-med, and Java-large based on the execution time of the data used in this paper.

Although the training complexity of each of the three studied neural code models can be different, their lower boundary is the training complexity is linear, i.e., the training time is linear to the size of the dataset. Thus, the estimated time in Table I is the lower boundaries. For example, the size of the Java-small dataset is about 66 times more than the size of CoCoFuzzing dataset, so the estimated execution time of Java-small is about 66 times more than the execution time of CoCoFuzzing, which is around 11 days. The same estimation method applies to other models and datasets.

The calculation of neuron coverage is the bottleneck of the testing execution. The neuron coverage calculation necessitates intensive memory I/O operations and storage space for the weights. In addition, memory is required for the storage of multiple weights, as the activation difference between mutations must be computed. Due to the aforementioned restrictions, performance optimization strategies like multi-threading and parallel computing are ineligible.

We will extend our approach with more mutation operators for supporting more program languages. CoCoFuzzing uses a threshold MAX to limit the maximum number of mutation tries on a seed program. We set MAX to three in this work to simulate the natural statistics of unused code in software projects, while the performance of CoCoFuzzing could vary with different values of MAX. We plan to explore the effectiveness of CoCoFuzzing with more MAX values in the future. In this work, following existing studies [14], [31] we use neuron coverage to guide the generation of valid mutants.

In this work, following existing fuzzers for testing deep learning models, we also use neuron coverage to guide the generation of tests. Although there have been increasing discussions on whether neuron coverage is a meaningful metric, our experiments show that it works for testing neural code models. We plan to examine the effectiveness of CoCoFuzzing with more criteria. In this work, we did not examine CoCoFuzzing on the pre-trained models (e.g., CodeBERT [38] and GraphCodeBERT [39]) due to resource limitation. We believe CoCoFuzzing is also applicable to CodeBERT and GraphCodeBERT, as they are transformer-based models that are similar to the NeuralCodeSum model. We will continue to apply CoCoFuzzing to these large pre-trained models in the future.

VII. Related Work

This section presents the existing research related to our work.

A. Testing Deep Learning Models

In recent years, there are many studies on testing deep learning models. Pei et al. [13] proposed DeepXplore to systematically find inputs that can trigger inconsistencies between multiple deep neural networks. They introduced neuron coverage as a systematic metric for measuring how much of the internal logic of a model have been tested. DeepXplore improves the classification accuracy by 3% via retraining the deep learning models with augmented data.

Tian et al. [31] proposed DeepTest for failure detection on Deep Neural Network (DNN)-based Autonomous driving systems. They adopted the neuron coverage metric as the criteria to generate synthetic inputs to test deep learning models. DeepTest improves the accuracy of examined models up to 46% by re-training the model with an augmented dataset generated by the neuron-coverage-guided greedy search. We also use neuron coverage as guidance to our Mutation operator selection algorithm in our work. In addition, CoCoFuzzing applies neuron coverage to maximize the difference of neuron activation of code models, while DeepTest focuses on image processing. Ma et al. [12] purposed DeepGauge with 5 coverage criteria for deep learning models, which extend the coverage metrics from neuron-level to layer-level. Xie et

al. [14] proposed DeepHunter, a fuzzing testing-based tool for testing deep learning models, in which they examined different types of seed selection strategy and test criteria. DeepHunter applied both neuron coverage and coverage criteria from DeepGauge in mutation generation. Odena et al. [30] presented TensorFuzz which applied fuzz-based coverage testing for deep learning systems. Guo et al. [40] proposed DLFuzz, a differential fuzzing testing framework to guide deep learning systems exposing incorrect behaviors. Wang et al. [41] propose a novel approach to detect artificially generated adversarial examples for Deep Neural Network models at runtime. The main difference between our work and the above studies is that most of the existing tools focus on general deep learning models in classic deep learning application domains, e.g., image processing, speech recognition, and natural language processing (NLP). While `CoCoFuzzing` is the first fuzzing framework for neural code models.

B. Adversarial Machine Learning.

Adversarial machine learning primarily focuses on generating adversarial examples to improve the performance of deep learning models.

Gradient ascent-based adversarial examples generation such as FGSM (Goodfellow et al. [42]) and BIM (Kurakin et al. [43]), which leverages the gradient of the model for finding adversarial example similar to the original input, has been widely used to accelerate the adversarial example generation problem for deep learning applications in the domains of image processing, speech recognition, and natural language processing. Recently, Yefet et al. [18] proposed DAMP, i.e., a gradient-based adversarial example generation technique, to generate adversarial examples for deep learning models in the domain of source code process. DAMP adopted two semantic preserving transformation operators, i.e., renaming variables and dead code inserting. Zhang et al. [17] proposed MHM that generated adversarial examples by renaming variables based on a sampling algorithm. Their experimental results demonstrate that MHM could effectively generate adversarial examples to attack the subject code process models. Ma et al. [44] introduced DeepMutation, a testing framework for deep learning systems based on FNN models. DeepMutation applies eight mutation operators for FNN models. On top of DeepMutation, Hu et al. [45] introduced DeepMutation++ with 9 mutation operators on RNN models. Vahdat et al. [46] proposed a search-based testing framework for adversarial robustness testing. The differences to `CoCoFuzzing` are, firstly, out of the 10 operators `CoCoFuzzing` uses, there are 8 different operators compared to Vahdat et al.'s work. For the two similar operators (Renaming and Argument adding), `CoCoFuzzing` rename the variables with an 8-characters-length random string, while Vahdat et al.'s work uses the synonym of variables. Secondly, they used an evaluation metric derived from DeepMutation++ [45] to guide the mutation, while `CoCoFuzzing` uses neuron coverage (NC) to guide the mutation.

C. Robustness of Neural Code Models

Gehr et al. [47] proposed AI2, a neural network analyzer that can automatically certify the robustness and safety properties of a neural network. It introduces two modified neural network layers, abstract transformers, and max-pooling layer, to effectively evaluate the real-world neural networks that use transformers and max-pooling. The goal of AI2 is to formally verify the safety and robustness of a model while our work aims to test and improve the robustness of a deep learning model via neuron-coverage-guided mutation testing. Hong et al. [48] demonstrated that CODE2VEC cannot be effectively leveraged when being readily generalized to other code-to-text downstream tasks, i.e. comments generation, code authorship identification, and code clones detection. Their experiment shows that CODE2VEC does not perform better than a simpler method in the downstream tasks. Our work focuses on testing the robustness of CODE2VEC on its original task, while their study is on re-using the trained CODE2VEC model for other downstream tasks. Pavol and Martin [49] introduced an adversarial training approach for a code model that is specific to the type-inference task. they train a model [50] that only gives prediction when the certainty of result is high and does not give prediction otherwise. Then, they apply adversarial training [42] and collect the information of how the part of the program affects the prediction. Last, they train multiple smaller but more robust models with subsets of the training data according to the information collected. Since the task of their model is to infer the type of variable of a piece of program, they could replace input tokens using the adversarial training approach without the concern of execution consistency. Adversarial training cannot be directly applied to our case since we have to make sure the ground truth is preserved after mutation. Rhys and Panos [51] observed that variable renaming could significantly affect CODE2VEC prediction. Re-training CODE2VEC with Variable Obfuscation improves the performance of CODE2VEC on 6 out of 7 models. Our research includes more mutation operators and more models, which have a larger scale compared to this work. Goutham et al. [52] introduces an adversarial training approach for source code models. The training target is to achieve transform robustness by re-train the model with an augmented dataset generated with 8 transform operators. Our mutation proposed 6 types of dead code injection while their work only considers adding dead if statement (similar to Op4). Md Rafiqul et al. [53] explored the generalizability of source code models by metamorphic testing using 6 types of transformation. They evaluated the performance of 3 models including CODE2VEC, and CODE2SEQ with datasets containing different percentages of mutation. The observation is that the prediction change of a model increases as the degree of mutation increases. Compared to our research, this work focuses on how mutation affects the generalizability of code models, while we further explored how an augmented dataset can help increase the robustness of code models. Zhang et al. [54] conducted a survey on machine learning testing. The survey provides a landscape on machine learning testing with several important topics including fairness, efficiency, robustness, and security.

VIII. Conclusion

This paper presents a coverage-based fuzzing framework, `CoCoFuzzing`, for testing deep learning-based models for code processing. In particular, we first propose and implement ten mutation operators to automatically generate valid (i.e., semantically preserving) source code examples as tests; Then, we propose an approach based on neuron coverage to guide the generation of tests. We investigate the performance of `CoCoFuzzing` on three state-of-the-art and typical neural code models, namely NeuralCodeSum, CODE2SEQ, and CODE2VEC. Our experiment results demonstrate that `CoCoFuzzing` can generate diverse and valid tests for evaluating the robustness and generalizability of neural code models. Moreover, the generated tests can be used for adversarial retraining to improve the performance of the target neural code models.

IX. Acknowledgments

The authors thank the anonymous reviewers for their feedback which helped improve this paper.

References

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

[2] 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.

[3] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[4] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[5] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*, 2016, pp. 2091–2100.

[6] M. Chen and X. Wan, "Neural comment generation for source code with auxiliary code classification task," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 522–529.

[7] Y. Huang, M. Wei, S. Wang, J. Wang, and Q. Wang, "Yet another combination of ir-and neural-based comment generation," *Information and Software Technology*, vol. 152, p. 107001, 2022.

[8] L. Shi, F. Mu, X. Chen, S. Wang, J. Wang, Y. Yang, G. Li, X. Xia, and Q. Wang, "Are we building on the rock? on the importance of data preprocessing for code summarization," *arXiv preprint arXiv:2207.05579*, 2022.

[9] M. Wei, N. S. Harzevili, Y. Huang, J. Wang, and S. Wang, "Clear: contrastive learning for api recommendation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 376–387.

[10] G. Kang, J. Liu, B. Cao, and M. Cao, "Nafm: neural and attentional factorization machine for web api recommendation," in *2020 IEEE international conference on web services (ICWS)*. IEEE, 2020, pp. 330–337.

[11] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 631–642.

[12] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 120–131.

[13] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[14] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146–157.

[15] P. Zhang, Q. Dai, and P. Pelliccione, "Cagfuzz: Coverage-guided adversarial generative fuzzing testing of deep learning systems," *arXiv preprint arXiv:1911.07931*, 2019.

[16] M. Alzantot, Y. Sharma, A. Elgohary, B.-J. Ho, M. Srivastava, and K.-W. Chang, "Generating natural language adversarial examples," *arXiv preprint arXiv:1804.07998*, 2018.

[17] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1169–1176.

[18] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *arXiv preprint arXiv:1910.07517*, 2019.

[19] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[20] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.

[21] Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[22] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[23] W. Yin, H. Schütze, B. Xiang, and B. Zhou, "Abcnn: Attention-based convolutional neural network for modeling sentence pairs," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 259–272, 2016.

[24] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[25] T. Shen, T. Zhou, G. Long, J. Jiang, S. Wang, and C. Zhang, "Reinforced self-attention network: a hybrid of hard and soft attention for sequence modeling," *arXiv preprint arXiv:1801.10296*, 2018.

[26] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, "Self-attention generative adversarial networks," in *International Conference on Machine Learning*, 2019, pp. 7354–7363.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[28] M. Zalewski, "American fuzzy lop," 2014.

[29] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker, "Announcing oss-fuzz: Continuous fuzzing for open source software," *Google Testing Blog*, 2016.

[30] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4901–4911.

[31] 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*, 2018, pp. 303–314.

[32] K. Serebryany, "Libfuzzer: A library for coverage-guided fuzz testing (within llvm)."

[33] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 861–875.

[34] S. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K.-H. Prommer, "How much does unused code matter for maintenance?" in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1102–1111.

[35] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[36] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[37] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 109–119.

[38] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[39] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[40] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.

[41] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, "Adversarial sample detection for deep neural network through model mutation testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1245–1256.

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

[43] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533*, 2016.

[44] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2018, pp. 100–111.

[45] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1158–1161.

[46] M. Vahdat Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," *arXiv e-prints*, pp. arXiv–2101, 2021.

[47] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 3–18.

[48] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1–12.

[49] P. Bielik and M. Vechev, "Adversarial robustness for code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 896–907.

[50] Z. Liu, Z. Wang, P. P. Liang, R. R. Salakhutdinov, L.-P. Morency, and M. Ueda, "Deep gamblers: Learning to abstain with portfolio theory," *Advances in Neural Information Processing Systems*, vol. 32, pp. 10 623–10 633, 2019.

[51] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 243–253.

[52] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," *arXiv preprint arXiv:2002.03043*, 2020.

[53] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, "On the generalizability of neural program models with respect to semantic-preserving program transformations," *Information and Software Technology*, vol. 135, p. 106552, 2021.

[54] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, 2020.