CodeImprove: Program Adaptation for Deep Code Models

Ravishka Rathnasuriya*
University of Texas at Dallas
USA
ravishka.rathnasuriya@utdallas.edu

Zijie Zhao+
University of Pennsylvania
USA
tez@seas.upenn.edu

Wei Yang
University of Texas at Dallas
USA
wei.yang@utdallas.edu

***Abstract*—Leveraging deep learning (DL)-based code analysis tools to solve software engineering tasks is becoming increasingly popular. Code models often suffer performance degradation due to various reasons (e.g., code data shifts). Retraining is often required to address these issues, but frequent model updates are costly in labeling and deployment. In this paper, we explore an alternative solution: Adapting the program inputs to the code models. This can be achieved by two steps: 1) input validation that focuses on identifying whether an input is an out-of-scope input program that are beyond a model's handling capability, and 2) input adaptation that adapts out-of-scope inputs to become in-scope inputs. Validating program input is challenging, as current techniques focus on continuous inputs such as image data and fail with discrete inputs like code data, which have unique characteristics and are processed differently by deep learning models. Adapting out-of-scope programs is also challenging due to their vast search spaces. Therefore, in this paper, we propose CodeImprove, which distinguishes out-of-scope from normal inputs and converts such out-of-scope inputs back to in-scope inputs through program transformation. In particular, we propose a validity score metric to identify out-of-scope inputs and leverage genetics algorithms to apply semantic preserving program transformation to convert out-of-scope inputs to in-scope inputs. Our experimental results show CodeImprove can enhance upto 8.78% of accuracy, and 51.28% of relative improvements in three code models on two SE tasks. Additionally, our input validation is promising in detecting out-of-scope inputs (AUC score of 0.924).**

***Index Terms*—Input Validation, Program Transformation**

I. Introduction

In the field of software engineering, code analysis tools play a crucial role in addressing critical tasks such as vulnerability detection, defect prediction, and clone detection [1]–[4]. Code analysis tools often suffer performance degradation due to various reasons [5]–[10]. Traditionally, the primary approach to overcoming these obstacles has been to refine the tools to handle various challenging cases, which might involve updating their versions or incorporating new heuristics. However, this process can be complex and resource-intensive. Recently, researchers have identified an alternative strategy: *adapting the inputs to the tools* [9], [10]. This approach has gained traction, especially in scenarios where refining the tools becomes impractical. This adaptation of inputs serves as an effective strategy to circumvent the limitations of tool modification, ensuring that code analysis tools remain effective, even for cases they were initially unable to address before the adaptation of inputs.

While existing input adaptation efforts have primarily focused on traditional symbolic reasoning-based tools [9], [10], there is a growing need for developing input adaptation strategies for (deep) learning-based code analysis tools/models [11]–[14]. This need arises from the challenges and high costs associated with improving these models to address their limitations. Commonly, improving these tools involves retraining and replacing the underlying deep learning models [15]–[17]. Such processes not only lead to increased labeling and computing efforts but also risk reducing the models' generalization capabilities, alongside potential compatibility and version control issues. Additionally, model replacement may need the creation of new architectures and datasets, which incurs substantial costs related to rebuilding and redeploying the systems.

To address the challenges, an alternate cost-effective solution is to adapt the program inputs to learning-based tools without altering the original tool. This strategy of input adaptation is particularly beneficial in scenarios like agile development, where the code base rapidly changes, as it maintains the model's applicability without the need for frequent retraining. Additionally, exploring different input adaptation techniques is more resource-efficient than the continuous retraining of a model to discover the optimal approach.

The process of adapting an input to a code model generally involves two key steps: (1) *input validation* that aims at identifying out-of-scope inputs that fall outside the model's capacity (i.e., inputs prone to being mishandled) , and (2) *input adaptation*, where the out-of-scope inputs are converted by semantic-preserving transformations to become in-scope inputs that are within the model's handling capabilities (i.e., inputs that the model is likely to process correctly).

The process of validating and adapting inputs for code models encompasses distinct challenges. First, in terms of input validation, it is difficult to create a metric that accurately predicts the likelihood of a model generating a correct or incorrect output for a given input. The machine learning community has developed some methodologies known as uncertainty metrics [18]–[28] to estimate a model's level of uncertainty for

[*] Corresponding author.

[+] At the time of submission, Zijie Zhao was serving as an intern at the University of Texas at Dallas, USA.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00139

a specific input. However, these metrics, which are primarily designed for image data, often fall short when applied to code data in identifying out-of-scope inputs. The root of this issue lies in the nature of image data, which is typically continuous and dense with fewer semantic nuances, allowing for smooth gradients that make the uncertainty metrics more reliable. In contrast, code data is inherently discrete, structured, and filled with abstractions such as control flows and data dependencies, which can result in abrupt changes in the model's output and make the uncertainty less predictable. Moreover, while image data mainly undergoes noise, corruption, or compression as matrix transformations, code data faces changes in syntax and programming paradigms. Our preliminary study (Section III) suggests that these essential distinctions pose a substantial challenge to the direct application of standard uncertainty metrics to code data.

Second, once specific inputs have been identified for adaptation, transforming these inputs into the model's handling scope is also challenging. First, navigating the vast search spaces involved in code transformation is a complex task. Considering the numerous possible transformations that can be applied to code and the fact that many of these transformations can be applied repeatedly to produce various forms of code, the resulting array of potential variations creates a complex landscape for exploration. Moreover, any modifications made during the program transformation process must maintain the original program functionality and semantics of the code. Therefore, efforts to modify the code to enhance model compatibility must be carefully balanced to avoid unintentionally changing its fundamental meaning or functionality, thus posing a dilemma between improving model performance and preserving code semantics.

To address these challenges, we propose CodeImprove (Figure 1), the first techniques for input validation and input adaptation of code inputs. For input validation, we identified that existing uncertainty metrics misrepresent the model's handling capability on code inputs, leading to overconfident predictions for out-of-scope code inputs (Section III). We observe that the relevance of different aspects of the input, such as structural information or variable names, can shift dynamically across the model's layers. Traditional uncertainty metrics, which typically focus on the outputs of the final few layers, fail to capture this layer-by-layer processing.

Based on such observation, we propose a Dropout-based Sub-Model Generation (DSMG) approach to find an optimal hidden state representation that accurately identifies in-scope versus out-of-scope inputs. By analyzing sub-models derived from the original DL model, CodeImprove can delve into how inputs are processed at each layer. DSMG allows CodeImprove to generate sub-models that provide deeper insights into the transformation of inputs through the network. CodeImprove utilizes the confidence levels of these sub-models' predictions as a new metric for assessing the validity of inputs, offering a more reliable measure that captures the complexities of code input processing in DL models.

Following input validation, CodeImprove employs Adaptation by Evolutionary Search (AES). We develop a list of basic semantic preserving transformations and leverage DSMG's validation score as a guiding metric to combine these basic transformations into a composite transformation that effectively covert the input from being out-of-scope to in-scope.

We evaluated our technique with pre-trained transformer-based language models on software engineering tasks such as vulnerability detection and defect prediction. Our experimental results report promising results and show CodeImprove can enhance 8.78% of absolute accuracy, and 51.28% of relative improvements in three code models on two code tasks. Notably, our validity score computation that validates out-of-scope inputs obtained promising results (AUC score of 0.924).

We summarize our contributions in this paper below:

- **Novel Perspective**. We propose a novel perspective of differentiating out-of-scope from in-scope inputs, as well as adapting these out-of-scope inputs to become in-scope inputs. To the best of our knowledge, our novel perspective is the first attempt to adapt inference-time inputs for deep code models through program transformation.
- **Tool Implementation**. We implement CodeImprove following the novel perspective (1) by implementing a sub-model generation technique from the original code model for code data, (2) designing a validity score metric to distinguish out-of-scope inputs from in-scope inputs utilizing the generated sub models, and (3) designing a genetic algorithm based technique to adapt out-of-scope inputs to become in-scope inputs by applying program transformation.
- **Comprehensive Evaluation**. We conducted an extensive study on three popular pre-trained models and two code-base tasks, demonstrating the effectiveness and efficiency of CodeImprove's input validation and input adaptation on test data.
- **Public Artifact**. We release all experimental data and source code at the project Github repository [29] for future research, practical use, and experiment replication.

II. Background

A. Problem Definition

Given a code model M and an input code snippet x, the class with the highest probability is the final prediction result of M for x, denoted as $y = M(x)$. During deployment, ensuring the correctness of every prediction is challenging. Thus, the objective is to enhance model performance on test inputs through code adaptation.

The validation metric $V(M, x)$ evaluates M's uncertainty on input x to determine whether the input is in-scope or out-of-scope. If $V(M, x)$ is less than the predefined threshold c, it indicates uncertainty, and x requires adaptation. Otherwise, the x is considered in-scope. The set of transformations $T = \{T_1, T_2, \ldots, T_n\}$ refers to a sequence of code transformation operators applied to the out-of-scope input x, resulting in a modified input x'. Let $\hat{y}$ represent the ground truth of x and let $y' = M(x')$ be the prediction result after adapting x to x' via T.

The goal is to compute V and apply T such that the loss function of the adapted prediction $L(y')$ is smaller than the original loss $L(y)$. We aim to find V and T to make $L(y') < L(y)$ where:

$$y' = \begin{cases} M(T_1, T_2, \ldots, T_n(x)), & \text{if } V(M,x) \leq c, \\ y, & \text{if } V(M,x) > c, \end{cases}$$

The loss $L(y)$ is characterized by the distance from the ground truth $\hat{y}$:

$$L(y) = \|y - \hat{y}\|$$

This can be generalized to any distance metric (e.g., L_1,L_2, L_∞) [30]–[33] to accommodate for any SE task.

The challenge is to develop an effective validation metric V (Oracle problem) and a determine a sequence of transformations T (Search problem) that adapt out-of-scope inputs.

B. Oracle Problem- Developing V

During deployment, determining whether a prediction is correct without manual analysis is challenging. An effective validation metric V is needed to automatically guide M to accurately make decisions, thus reducing false positives. A substantial progress has been made in this direction such as handling uncertainty [18]–[28], deep emsemble [34], and cross-layer dissection [19].

C. Search Problem- Transformation Sequence T

Beyond validation, the process of adapting the out-of-scope inputs to become in-scope inputs necessitates efficient search algorithms to explore program syntax for potential transformations. The goal of search techniques is to optimize the code transformations while preserving the program semantics. Search techniques like random search [35], hill climbing search [36], and genetics algorithms [37] offer solutions for code transformations. The search strategy begins with a set of candidate solution(s) generated by applying semantic preserving code transformation. These candidates are then evaluated using V to select the most promising one. The search algorithm iteratively refines the candidates until a termination criterion is reached or an optimal solution is found.

III. Preliminary Study on Input Validation for Code Models

In our preliminary study, we assess the applicability of existing uncertainty metrics within the realm of code models, aiming to identify a dependable threshold score that can differentiate in-scope and out-of-scope program inputs. We perform the study on a comprehensive set of metrics [8], [18]–[20], [22] from the existing literature. Our goal is to answer the research question: How effective are the existing uncertainty metrics in distinguishing in/out-of-scope program inputs?

A. Experimental Method and Setup

Uncertainty Metrics: Our study includes eight different uncertainty metrics. We utilize vanilla [20] that computes maximum softmax probability as the confidence. Temp Scale [18] is a post-hoc calibrated confidence metric applied to the validation set, where the BFGS optimizer [38] is used to train a calibration temperature with a learning rate of 0.01. Our study includes confidence-based uncertainty metrics, such as least confidence [26], which calculates the difference between 100% confidence and the most confidently predicted label; margin confidence [26], which determines the difference between the top two most confident softmax predictions; and ratio confidence [26], which computes the ratio between the top two most confident softmax predictions. We also include uncertainty metrics that were designed using information theory [27], [28]. Entropy computes the average amount of surprise/ uncertainty for a given outcome. Predictive entropy quantifies the uncertainty associated with the outcomes for a given set of observed inputs. Mutual information measures the amount of information obtained from one random variable given another using entropy and conditional entropy. Monte-Carlo Dropout (MCD) [21] quantifies uncertainty by averaging the logits over multiple dropout samples. Deep Ensembles (DE) [34] quantifies uncertainty by averaging the outputs from multiple independently trained models with different initial seeds. Both employ the sampled winning score (SWS) as the primary uncertainty metric.

Dataset and Models: To evaluate the effectiveness of detecting out-of-scope data through uncertainty quantification, we consider two code tasks and two associated datasets (i.e., defect prediction with CodeChef dataset and vulnerability detection on Devign dataset) on three pre-trained models. More information on datasets and subject models is explained in Section V-B.

TABLE I: AUC Comparison on Distinguishing Out-of-Scope Inputs for Selected Uncertainty Metrics

Experiment	**Vulnerability Detection**			**Defect Prediction**		
	CodeBERT	RoBERTa	GraphCodeBERT	CodeBERT	RoBERTa	GraphCodeBERT
Vanilla	0.552	0.574	0.455	0.595	0.580	0.494
Temp. Scaling	0.420	0.572	0.442	0.403	0.413	0.493
Predicitive Entropy	0.582	0.454	0.508	0.573	0.579	0.491
Entropy	0.414	0.556	0.482	0.436	0.391	0.426
Mutual Information	0.586	0.443	0.531	0.566	0.608	0.579
Least Confidence	0.589	0.452	0.558	0.595	0.593	0.508
Ratio Confidence	0.464	0.521	0.486	0.553	0.484	0.469
Margin Confidence	0.473	0.521	0.489	0.562	0.532	0.470
MCD	0.624	0.613	0.617	0.614	0.607	0.616
DE	0.507	0.519	0.507	0.561	0.562	0.571

Evaluation Metric: We used the Area Under Curve (AUC) [39] based on True Positive Rate (TPR) and False Positive Rate (FPR) data to measure how effective a technique is in distinguishing an out-of-scope inputs from in-scope inputs. AUC quantifies the probability that a positive example receives a higher predictive score than a negative sample. For example, a random classifier yields an AUC of 0.5, while a perfect classifier achieves an AUC of 1.

To compute the AUC scores, we define positive and negative samples based on the correspondence between predicted outputs and ground truth labels. A positive sample indicates correct predictions (in-scope), labeled as 1, while a negative sample signifies misclassified (out-of-scope), labeled as 0. For instance, given a well-trained model $f(\cdot)(|\theta)$, an input pair (x, y) has ground truth 1 if $f(x|\theta)$ is an exact match of y, and 0 otherwise. During evaluation, each uncertainty method predicts a score reflecting the model's capability in handling the input.

B. Results and Analysis:

Table I shows AUC scores for the evaluated uncertainty metrics. The majority of these metrics exhibit AUC scores close to 0.5, akin to what one would expect from a random classifier, with none surpassing an AUC score of 0.624. From these results, it is inferred that the uncertainty metrics under consideration are ineffective at distinguishing between in-scope and out-of-scope code inputs.

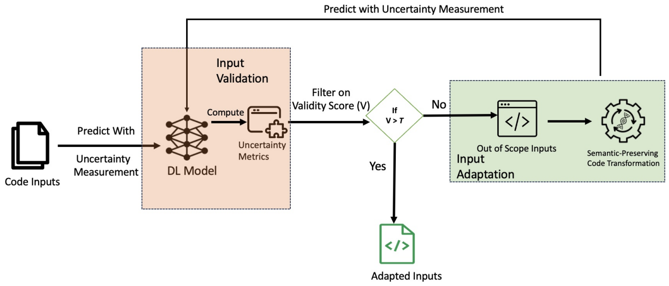


Fig. 1: Overview of CodeImprove

IV. Design of CodeImprove

Figure 1 provides an overview of CodeImprove. CodeImprove consists of two main technical phases: *Input validation* phase to detect out-of-scope inputs from normal inputs, and *Input adaptation* phase to transform out-of-scope inputs to in-scope inputs.

A. Input Validation

The goal of this phase is to design a guiding metric that distinguishes between in-scope and out-of-scope test data for a trained DL model and its associated dataset. Our preliminary investigation (Section III) demonstrates that traditional uncertainty metrics are inadequate for software engineering (SE) tasks, motivating the need for a more granular approach to understanding input behavior.

We aim to identify an optimal hidden state representation that distinguishes between in-scope and out-of-scope inputs. Code inputs fed into the DL model's layers often experience shifting focus across various aspects such as from structural information to variable names. Existing uncertainty metrics which rely on outputs from the final few layers fails to capture the this dynamic layerwise processing. To bridge this gap, we explore sub-models extracted from original model to gain deeper insights into the processing of inputs. These sub-models enable us to measure confidence at multiple levels, forming the basis for a more reliable input validation metric.

A key motivation for our approach is the need to compute epistemic uncertainty, which arises from the model's lack of knowledge about certain inputs and reflects how confident the model is when making predictions. Accurately estimating epistemic uncertainty requires introducing model variance, which captures the variability in predictions when slight changes are applied to the model's architecture or processing. Existing approaches, such as dropout-based techniques [21] or ensemble methods [34], aim to create model variance but face significant limitations in the context of code data. These methods do not investigate layerwise processing, ignoring the dynamic transformations that inputs undergo at different model layers, nor do they explore hidden layer representations, which are crucial for understanding how the model processes structured data like code. Most importantly, they are primarily designed for continuous input spaces like images, where smooth gradients and variations can be leveraged for uncertainty estimation.

Our approach introduces sub-models to address these limitations. These sub-models are generated by selectively extracting representations from intermediate layers of the original model and introducing architectural diversity inspired by sub-ensemble methods. [40] Each sub-model independently processes the same input, providing a detailed view of how the original model transforms inputs across layers. The key insight is that the degree of agreement among sub-model predictions correlates strongly with the trustworthiness of the model's overall prediction. When sub-models produce consistent predictions, it indicates that the input is in-scope. Conversely, significant disagreement among sub-models suggests the input is likely out-of-scope.

Training sub-models focuses on capturing layer-specific uncertainties by freezing earlier layers and retraining only the dense layers, which act as classifiers for each task. This approach enhances the signal strength for determining model confidence. Additionally, studies have shown [41] that shallower layers in code models often outperform deeper ones for specific tasks, making the sub-model strategy particularly effective for code input validation. By dissecting predictions layer by layer, sub-models can pinpoint nuances in data representation, resulting in more accurate identification of in-scope and out-of-scope inputs.

Sub-Model Generation. Figure 2 shows the overview of our Dropout-based Sub-Model Generation (DSMG). DSMG constructs diverse sub-models to capture the hidden state representations of the original model, enabling a more detailed analysis of input processing. Each sub-model consists two components: the first part is inherited from the original DL model, containing all structures and parameter values from the first layer up to an intermediate layer k (for Sub-Model$_k$), and the second part is a newly trained dense layer linking layer k to the output, customized for the specific software engineering (SE) task.

To generate first part, dropout-based hidden representations are extracted from each layer, introducing controlled randomness to highlight distinct processing behaviors. To generate second part, the dense layer is trained independently for each sub-model using cross-entropy as the loss function, which has demonstrated effectiveness for classification tasks in deep neural networks [42], [43]. Dropout regularization is applied during training to prevent overfitting and enhance generalization. Notably, this sub-model generation process is performed offline, tailored to the SE task at hand, and does not affect the model during deployment.

Dropout-based layerwise hidden representation. In DSMG, hidden state representations are generated by selectively choosing specific layers and applying dropout to nodes within these selected layers. Dropout randomly omits a subset of nodes in a chosen layer during sub-model generation, while retaining the remaining active nodes. This approach introduces diversity along two axes: the depth of layers considered and the variation induced by dropout. Such diversity enables DSMG to effectively capture the model's processing of different input patterns and structures, providing a richer understanding of its behavior. This enhanced representation equips CodeImprove to better identify and differentiate in-scope inputs from out-of-scope inputs.

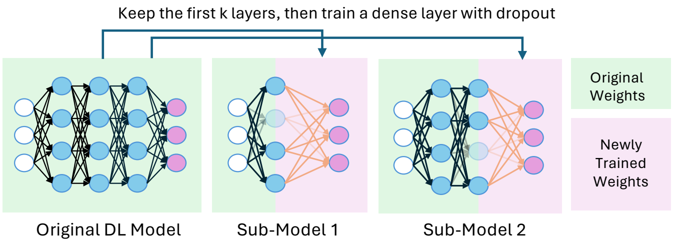


Fig. 2: Overview of Sub-Model Generation

Sub-model Validity Measurement. Equation 1 outlines the computation of the validity score, which relies on understanding the processing of inputs across layers. For a given input X fed into a DL model M with n labels, M predicts X as label l_x. Let $submodel_k$ be the softmax probability vector associated with the sub-model k.

To calculate the validity score, we differentiate between two scenarios: l_x being a correct prediction or an incorrect one. For correct predictions, we employ the Best Versus Second Best (BVSB) strategy, which measures the difference between the highest predicted probability (labeled as l_x) and the second-highest predicted probability (labeled as l_s). This difference reflects the confidence of the sub-model, with a larger difference indicating higher confidence. For incorrect predictions, the BVSB strategy instead compares the actual highest predicted probability (labeled as l_h) with the probability assigned to the original model's predicted label (l_x) highlighting the uncertainty between the sub-model's confidence and the incorrect prediction. This approach, formulated in Equation 2 ensures that the validity score effectively captures the reliability of the sub-model's predictions.

Once each sub-model computes the respective validity score for each input, we utilize the dissector's approach [19] (i.e., weight growth types) to compute the validity for the whole DL model (Equation 3).

$$\text{ValidityScore}_k(l_x, \text{submodel}_k) = \begin{cases} \text{submodel}_k[l_x] + B(\text{submodel}_k), & l_x \text{ with the highest probability} \\ \text{submodel}_k[l_x] - B(\text{submodel}_k), & \text{otherwise} \end{cases} \tag{1}$$

$$B(\text{submodel}_k) = \begin{cases} \text{submodel}_k[l_x] - \text{submodel}_k[l_s], & \text{if } l_x \text{ is correct,} \\ \text{submodel}_k[l_h] - \text{submodel}_k[l_x], & \text{if } l_x \text{ is incorrect.} \end{cases} \tag{2}$$

$$Final_{score}(l_x) = \frac{\sum_{k=1}^{n} ValidityScore_k(l_x, submodel_k) \times weight_k}{\sum_{k=1}^{n} weight_k} \tag{3}$$

B. Input Adaptation

Given an out-of-scope input, the goal of this phase is to covert the input to become an in-scope input. The challenge lies in exploring the large search space of possible program modifications while preserving semantic correctness. To tackle this, we define a set of semantic-preserving transformations. Guided by the DSMG validation score (Section IV-A), transformations are iteratively combined and refined to align inputs with the model's capabilities. This process, formalized as Adaptation by Evolutionary Search (AES), efficiently optimizes the transformations to achieve the best solution.

Algorithm 1: High-Level AES Algorithm

```
Input: Pre-trained Model $M$, Testing Dataset $T$,
       Out-of-scope Input ids $ids$, Maximum Iterations
       $max\_iter$, Mutation Rate $rate$, Fitness Threshold
       $threshold$
Output: New Test program dataset $N$
$N \leftarrow []$
foreach $sample$ in $T$ do
    if $sample.id$ is not in $ids$ then
        $N$.append($sample.code$)
        continue
    $i \leftarrow 0$
    $best\_candidate \leftarrow sample.code$
    $fitness \leftarrow [0, \dots, 0]$
    while $i < max\_iter$ and $fitness[best\_candidate] <$
      $threshold$ do
        $initial\_pop \leftarrow$ genpop($best\_candidate$)
        $fitness \leftarrow$ fitness($initial\_pop$)
        $new\_pop \leftarrow$ select($initial\_pop$)
        $pop \leftarrow$ evolve($new\_pop, rate$)
        $fitness \leftarrow$ fitness($pop$)
        $best\_candidate \leftarrow$ select_best($pop$)
        $i \leftarrow i + 1$
    if $fitness[best\_candidate] > threshold$ then
        $N$.append($best\_candidate$)
    else
        $N$.append($sample.code$)
return $N$
```

Program Transformations. To construct the set of basic program transformations, we considered all common kinds of code structures, including loop structures, branch structures,

TABLE II: List of Code Transformation

No	Transformation Operator	Description
1	changeName	Function name and variable name renaming
2	changeFor	The for-loop is transformed into a while-loop.
3	changeWhile	The while-loop is transformed into a for-loop
4	changeDo	The do-loop is transformed into a while-loop.
5	changeIfElseIf	Transformation of if elseif to if else
6	changeIf	Transformation of if else to if elseif
7	changeSwitch	Transformation of the Switch statement to the if elseif statement.
8	changeRelation	Transformation of relational expressions (e.g., a $<$ b to b>a).
9	changeUnary	Modifications to unary operations (e.g., i++ to i = i+1)
10	changeIncrement	Modifications to incremental operations (e.g., i+=1 to i = i+1).
11	changeConstant	Modifying Constant (e.g., 0 −> 8-8)
12	changeDefine	Modifications to variable definitions (e.g., int b=0 to int b; b=0).
13	changeAddJunk	Insert Junk Code that will never be executed. (e.g., if (0){printf();})
14	changeExchangeCod	Exchange the order of statements without dependencies (e.g., declaration statements)
15	changeDeleteComments	Deleting statements that print debugging hints and comment.(e.g., printf())

operator, and expression changes (Table II). All transformations are carefully compiled to ensure that the adapted program not only undergoes transformation but also upholds the semantics of the original code. Our list of transformation operators is designed to be task-preserving by ensuring that the functionality and behavior of the code remain unchanged. While there is potential to broaden this list of transformations, CodeImprove already shows significant improvement over existing techniques based on the current list. Due to the space limitation, we list all these specific atomic operations at our project homepage [44]. Then, we illustrate how to apply these operators to search for the best solution that deep code models can adapt.

Adaptation by Evolutionary Search: One of the primary requirements before applying transformation operators is to identify syntactic features, i.e., the places of code fragments applicable for transformation. Finding appropriate syntactic features is essentially an optimization problem. CodeImprove addresses this problem by counting the number of code fragments present in each operator for a given code snippet. For example, if there are four identifiers in a source code snippet, then the count of the operator 1 is $K = 4$.

After identifying the number of syntactic features to be transformed, CodeImprove needs a transformation strategy to generate a diverse pool of candidates. CodeImprove achieves this by implementing a genetic algorithm-based [45]–[47] strategy comprising initialization, a fitness function, crossover and mutation operators, and termination criteria to guide the transformation process. Algorithm 1 shows the overview of our transformation strategy. The inputs for the AES algorithm are the trained DL model M, the test data used to evaluate the M's performance, the identified out-of-scope data, the maximum number of iterations that the AES should evolve for, and the required fitness score that the solution of AES should achieve. The output of our genetic algorithm is a new dataset that includes the transformed source code. To avoid randomness when applying transformations, CodeImprove transforms all K counts of features in each operator.

Initially, CodeImprove creates a starting population by applying 15 operators to each out-of-scope input (Line 10), with each individual in this population representing a potential solution. The fitness of each individual, determined by the DSMG's validation score (Section IV-A), reflects how closely a solution approaches the problem's target, with higher scores indicating better candidates (Line 11).

Then, the genetic algorithm iterates through cycles of evaluation, selection, and reproduction. In the selection phase, the top 50% of candidates are chosen based on their fitness scores (Line 12). During the reproduction phase, CodeImprove performs genetic operators (i.e., crossover and mutation) to generate new solutions (Line 13). During crossover, for each candidate, CodeImprove applies a sequence of transformations. Then, we add the new samples to our population. For each crossover variant, we mutate the syntactic feature with a random transformation to maintain a diverse population. The algorithm terminates when the candidate code has reached a higher validation score based on our guiding metric or the fixed number of generations has reached. In the end, the algorithm returns the solution with the highest fitness value (Line 15). Next, we will describe the experimental evaluation.

V. Evaluation

We conducted all our experiments on a server equipped with an Intel Xeon E5-26 CPU and eight NVIDIA 1080 Ti GPUs. We set up 0.3 and 0.2 as the threshold based on our validation score on detecting out-of-scope inputs for vulnerability detection and defect prediction tasks, respectively, on all subjects. We set up maximum iteration to three and different crossover rates (i.e., 0.16, 0.33, 0.66, and 1) when applying sequences of transformation operators on each subject matter described in Section V.

A. Research Questions

RQ1. Overall Performance: What is the overall performance of CodeImprove?

RQ2. Input Validation: How effective is the out-of-scope program data detection?

RQ3. Input Adaptation: How effective to convert out-of-scope data to become in-scope data?

TABLE III: Effectiveness of CodeImprove

Experiment	**Model**	**Vulnerability Detection**						**Defect Prediction**					
		A(%)	P(%)	R(%)	F1(%)	RI(%)	CSR(%↑)/MCR(%↓)	A(%)	P(%)	R(%)	F1(%)	RI(%)	CSR(%↑)/MCR(%↓)
Original set up	CodeBERT	62.74	62.31	47.81	54.11	-	-	81.98	82.12	81.98	81.45	-	-
	RoBERTa	61.56	57.71	61.11	59.36	-	-	80.02	79.91	80.02	79.40	-	-
	GraphCodeBERT	62.40	61.50	48.20	54.09	-	-	81.91	81.77	81.91	81.56	-	-
InputReflector	CodeBERT	62.48	61.97	47.4	53.72	-1.4	1.6/0.05	80.85	81.01	80.85	80.24	-19.4	2.4/5.4
	RoBERTa	62.48	60.6	52.2	56.1	9.4	21.3/6.59	79.73	79.59	79.73	79.03	-6.6	0.3/0.4
	GraphCodeBERT	62.81	60.34	55.53	57.84	4.9	18.0/4.7	80.32	80.13	80.32	79.92	-21.8	1.42/2.1
CodeImprove	CodeBERT	**71.52**	**70.79**	**64.70**	**67.61**	**51.28**	**39.9/2.6**	**84.53**	**84.22**	**84.43**	**83.99**	**44.2**	**32.8/1.0**
	RoBERTa	**68.81**	**64.67**	**70.76**	**67.58**	**74.4**	**36.2/1.96**	**82.91**	**82.99**	**82.91**	**82.36**	**65.6**	**32.2/1.0**
	GraphCodeBERT	**65.26**	**64.22**	**55.06**	**59.29**	**35.1**	**23.1/1.9**	**83.45**	**83.55**	**83.45**	**83.09**	**21.1**	**27.2/1.4**

RQ4. Setting Sensitivity: How sensitive is CodeImprove's performance under different experimental setups?

RQ5. Semantic Preserving: Does program transformations of CodeImprove preserve semantics?

RQ6. Runtime Overhead: What is the overhead of CodeImprove in adapting a program to DL models?

B. Subjects

1) Datasets and Tasks: CodeImprove is evaluated on two code-based classification tasks: vulnerability prediction with devign dataset [48], and defect prediction with codeChef dataset [49]. Although this study used two datasets, CodeImprove applies to all code model-based tasks, with plans for broader future evaluations.

For **vulnerability detection**, the Devign dataset [48] consists of 27,318 functions extracted from FFmpeg and Qemu open-source C projects. The functions are labeled as containing vulnerabilities or being clean, with 14,858 vulnerable and 12,460 clean samples. The dataset is split into train/validation/test sets with sizes 21,854/2,732/2,732.

For **defect prediction**, the CodeChef dataset includes 33,822 C/C++ functions from the CodeChef platform [49]. Samples are labeled with categories such as no defect, wrong output, timeout error, or runtime error, with 11,362 no defect, 13,656 wrong output, 5,101 timeout error, and 3,703 runtime error samples. The dataset is divided into train/validation/test sets with sizes 21,647/5,411/6,764.

2) Models: We employed state-of-the-art pre-trained models, namely CodeBERT [50], RoBERTa [51], and GraphCodeBERT [13] which have been widely utilized in previous studies [4], [52]–[54]. These models were fine-tuned on our tasks using the corresponding datasets, adhering to the recommended settings proposed in previous literature [4]. Hyperparameters were set to match the original configurations. CodeImprove is designed for use with all types of code models. Our study includes a diverse range of tasks, pre-trained models, and class numbers, ensuring a comprehensive evaluation of CodeImprove's performance.

C. Evaluation Metrics

We use a diverse set of metrics to measure CodeImprove's effectiveness for our six RQs.

The **accuracy (A)** is the proportion of correctly classified samples out of all samples. The **precision (P)** is the percentage of correctly predicted positive samples out of all positive predictions. The **recall (R)** measures the percentage of correctly predicted positive samples that were retrieved out of all actual positive samples. The **F1 score (F1)** is the harmonic mean of precision and recall. **Relative improvement (RI)** quantifies the accuracy improvement relative to the difference between training and test accuracy.

Correction success rate (CSR) [52] is the ratio of successfully corrected mispredictions to the total identified mispredictions. **Mis-correction rate (MCR)** [52] measures the negative effect caused, which is the ratio of correct predictions changed to mispredictions to the total number of correct predictions in the test set.

Correction validation rate (CVR) is the ratio of successfully validated mispredictions to the total possible mispredicted inputs to be validated. **Mis-correction validation rate (MVR)** is the ratio of correct predictions validated as mispredictions to the total number of correct predictions in the test set. **AUC** score evaluates the effectiveness of the input validation process. **Transformations per Second (TPS)** measures the rate of transformations CodeImprove can apply per second. Next, we will describe the results of the experiments.

VI. Results and Analysis

We report and analyze experimental results, and answer the preceding research questions in turn.

A. RQ1: Overall Performance of CodeImprove

Baseline: CodeImprove is the first technique to improve the code model's performance through program transformations. Thus, we cannot find direct baselines for comparison. To address this, we draw inspiration from a technique in the image domain for comparative analysis, i.e. the InputReflector **(IRef)** [17] that detects deviating inputs and substitutes them with the most similar sample from the training set. We are unable to include CodeDenoise [52] as a baseline due to the evaluation methods of this work, which splits the test set into two subsets as T1 and T2 and subsequently assesses results on T2 set. However, the splitting criteria of datasets is not provided in the project website. Moreover, the work is similar to the adversarial style, which denoises the program identifiers and corrects the program with the supervision of the model's predictions. CodeImprove demonstrates better performance compared to CodeDenoise [52] where it only fixes 20.45% of inputs while CodeImprove fixes 32.8% for defect prediction task on the CodeBERT model.

Process: **IRef** utilizes two models, the siamese network [55] and the quadruple network [17], to detect deviating inputs and repair them. During training, these models rely on three datasets: the original set, a transformed set (human recognizable), and an extremely transformed set (human unrecognizable). These transformed sets are created by applying varying degrees of transformation to the original data. However, for code data, generating such datasets is challenging as transformations do not follow a continuous degree like in image data. Therefore, we utilized only two sets: the original set and a transformed version.

We adapted the IRef loss functions for these two sets and fed hidden layer outputs from the original model into the siamese and quadruple networks. To repair out-of-scope inputs, IRef searches for the most similar data in the training set and exchanges their labels. In contrast, CodeImprove applies all 15 semantic-preserving transformations during the crossover step. Effectiveness is evaluated using metrics such as accuracy, precision, recall, F1-score, RI, CSR, and MCR.

Result: Table III presents the comparison between CodeImprove and IRef, illustrating that CodeImprove consistently outperforms IRef. Notably, we observe the following: (1) CodeImprove consistently achieved the best model improvements with upto 8.78% in accuracy, 8.48% in precision, 16.9% in recall, and 13.5% in F1-score on all the subjects; (2) CodeImprove is capable of correcting around 23.1% to 39.9% of the mispredicted inputs on both vulnerability detection and defect prediction tasks; (3) CodeImprove shows notable RI improvements, ranging from 21.1% to 51.28%, particularly excelling with RoBERTa models; (4) Techniques designed for image data (e.g., **IRef**) fail in the context of code. IRef negatively impacts performance, especially for CodeBERT, as it focuses on out-of-distribution inputs rather than inputs prone to misprediction within the same distribution. This limitation arises from IRef's inability to effectively handle the syntactic and semantic similarities between transformed and original code; and (5) CodeImprove introduces minimal negative effects, with only 2.6% of correct predictions misclassified in the worst case. CodeImprove successfully adapts out-of-scope inputs to in-scope inputs, as demonstrated in Table III.

RQ1 - What is the overall performance of CodeImprove?

CodeImprove was effective in adapting out-of-scope inputs for both SE tasks on three subject models with higher accuracy, precision, recall, F1-score, CSR, and RI.(Table III).

B. RQ2: Effectiveness of Out-of-Scope Data Detection

Baseline: We compared CodeImprove's DSMG with the Dissector [19]. Additionally, we evaluated the DSMG approach with the uncertainty metrics in our preliminary study (Section III): Vanilla, temperature-scaling, predictive entropy, entropy, mutual information, least confidence, ratio confidence, and margin confidence, monte-carlo dropout, and deep ensemble.

Process: We compute the AUC score, CVR, and MVR to evaluate the effectiveness of out-of-scope data detection on all baseline approaches.

TABLE IV: Effectiveness of Input validation

Experiment	**Model**	**Vulnerability Detection**		**Defect Prediction**	
		CVR(%↑)/MVR(%↓)	AUC	CVR(%↑)/MVR(%↓)	AUC
Vanilla	CodeBERT	34.7/32.1	0.552	32.4/22.2	0.595
	RoBERTa	13.7/21.2	0.574	10.6/8.7	0.580
	GraphCodeBERT	20.6/16.9	0.455	18.8/20.8	0.494
Temp. Scaling	CodeBERT	38.8/31.8	0.420	36.5/27.6	0.403
	RoBERTa	10.0/15.4	0.572	28.5/16.2	0.413
	GraphCodeBERT	15.8/11.9	0.442	18.2/19.3	0.493
Predictive Entropy	CodeBERT	43.5/35.3	0.582	12.3/4.6	0.573
	RoBERTa	28.3/23.1	0.454	42.8/30.1	0.579
	GraphCodeBERT	13.1/14.6	0.508	25.6/29.0	0.491
Entropy	CodeBERT	12.5/30.8	0.414	39.1/27.8	0.436
	RoBERTa	7.1/8.6	0.556	29.9/16.7	0.391
	GraphCodeBERT	16.1/17.7	0.482	23.2/16.4	0.426
Mutual Information	CodeBERT	8.6/25.7	0.586	30.7/22.2	0.566
	RoBERTa	25.7/21.1	0.443	44.4/30.1	0.608
	GraphCodeBERT	35.2/41.1	0.531	19.2/14.5	0.579
Least Confidence	CodeBERT	16.7/14.5	0.589	19.1/9.7	0.595
	RoBERTa	24.7/32.1	0.452	10.7/13.9	0.593
	GraphCodeBERT	22.0/18.0	0.558	16.7/20.2	0.508
Ratio Confidence	CodeBERT	8.4/14.5	0.464	11.7/11.5	0.553
	RoBERTa	17.1/14.4	0.521	6.3/9.7	0.484
	GraphCodeBERT	47.3/42.8	0.486	26.9/21.7	0.469
Margin Confidence	CodeBERT	6.6/11.2	0.473	10.6/10.1	0.562
	RoBERTa	37.2/28.5	0.521	9.2/5.7	0.532
	GraphCodeBERT	35.1/32.6	0.489	8.3/5.1	0.470
MCD	CodeBERT	28.3/16.8	0.624	17.9/4.9	0.614
	RoBERTa	36.2/34.1	0.613	28.2/7.3	0.607
	GraphCodeBERT	38.7/30.3	0.617	20.5/5.3	0.616
DE	CodeBERT	44.4/43.9	0.507	24.7/20.8	0.561
	RoBERTa	48.1/33.5	0.519	24.2/17.7	0.562
	GraphCodeBERT	36.1/38.1	0.507	23.2/19.1	0.571
Dissector	CodeBERT	68.7/15.1	0.850	53.3/3.6	0.889
	RoBERTa	54.3/14.8	0.819	47.1/5.4	0.828
	GraphCodeBERT	40.1/16.7	0.757	53.4/4.6	0.873
CodeImprove	CodeBERT	**70.4/13.7**	**0.876**	**57.9/3.0**	**0.911**
	RoBERTa	**60.4/14.7**	**0.825**	**58.1/3.1**	**0.924**
	GraphCodeBERT	**47.2/13.0**	**0.781**	**56.1/3.3**	**0.909**

TABLE V: Effectiveness of Sub-model Decomposition

Experiment 3-4	**Model**	**Vulnerability Detection** **AUC**	**Defect Prediction** **AUC**
Hidden States	CodeBERT	0.557	0.607
	RoBERTa	0.534	0.582
	GraphCodeBERT	0.451	0.593
CodeImprove	CodeBERT	**0.876**	**0.911**
	RoBERTa	**0.825**	**0.924**
	GraphCodeBERT	**0.781**	**0.909**

Results: Table IV shows the results of out-of-scope data detection. Based on the results, we observe that: (1) CodeImprove achieved higher AUC scores across all models and tasks (i.e., AUC 0.781- 0.924); (2) While Dissector performs better than other uncertainty metrics, CodeImprove still surpasses it in AUC scores; (3) CodeImprove demonstrates a higher CVR across all subjects, detecting up to 70.4% of out-of-scope inputs for CodeBERT on vulnerability detection tasks and consistently outperforming other methods in CVR; (4) The MVR on CodeImprove is lower than other approaches, concluding that CodeImprove is better at differentiating in-scope inputs. MVR for defect prediction task shows 3.0%, 3.1%, and 3.3% for CodeBERT, RoBERTa, and GraphCodeBERT models; (5) MCD and DE average predictions over multiple forward passes through the same network. This approach limits diversity in the predictions, which may contribute to their poorer performance in AUC compared to CodeImprove; and (6) other uncertainty metrics did not produce promising results on AUC, CVR, or MVR. For example, predictive entropy obtained a CVR of 43.5% and an MVR of 35.3%, which are not significant indicators of effective performance.

Comparison of Using Sub-models vs. Hidden State Outputs from the Original Model: Sub-models are trained using the layerwise hidden states of the original model, but it is also possible to directly access these hidden states for detecting out-of-scope inputs. We conducted an experiment comparing trained sub-models with the direct use of hidden states. Due to the high dimensionality of the hidden states, we applied a linear transformation and subsequently applied Equations 1 - 3. Table V presents the statistics of the AUC comparison between these two methods.

Based on the results in Table V, we observe that training sub-models (AUC 0.781-0.924) outperforms direct use of hidden states (AUC 0.451 - 0.607) across both software engineering tasks. These results signify the performance of trained sub-models for out-of-scope input validation. We summarize several factors contribute to the lower effectiveness of directly using hidden states: 1) **Ineffective Feature Utilization and Transformation**: Effective input validation requires a mapping between the feature space and the class space. Without training a dense layer, this process would merely reduce dimensions without learning this mapping. Training a dense layer allows it to learn the most relevant features from the hidden states and establish an accurate mapping from the feature space to the class space. This reduces significant information loss and enhances overall performance; and 2) **Lack of Adaptability**: Trained dense layer in sub-models can adapt to the characteristics and distribution of the training data, making them more effective for each SE task. Without training, the model lacks this adaptability, resulting in poor effectiveness.

> **RQ2** - How effective is the out-of-scope program data detection?
>
> CodeImprove effectively distinguishes out-of-scope from in-scope inputs (AUC: 0.781-0.924, CVR: 47.2%-70.4%, MVR: 3.0%-14.7%), and is more suitable than existing techniques for various SE tasks.

C. RQ3: Effectiveness of Search Strategies to Adapt Out-of-Scope Inputs.

Baseline: We employed two search strategies; namely random search (**CodeImprove-rand**) [35], and Hill climbing algorithm (**CodeImprove-HC**) [36]. **CodeImprove-rand** applies random transformations until identifying the optimal candidate. **CodeImprove-HC** follows the principles of the hill climbing algorithm.

Process: For CodeImprove-rand, transformation operators are applied randomly until the algorithm identifies the best candidate. In CodeImprove-HC, the process starts with an initial solution obtained through a random transformation. The algorithm then iteratively applies a single transformation operator to improve the fitness score. Once the current solution surpasses the fitness threshold, the algorithm terminates, having reached the optimal solution. To ensure fairness, all techniques are limited to 15 transformations per solution. In CodeImprove, each candidate solution undergoes all 15 operators during the crossover phase.

Results: Table VI compares the three approaches, highlighting the following key observations: (1) CodeImprove obtained the best accuracy across all subjects (up to 8.78%) while both CodeImprove-rand and CodeImprove-HC did not achieve the performance of CodeImprove (up to 2.13%); (2) Although CodeImprove-HC and CodeImprove-rand improve the model performance, we find that these search algorithms stop at the local minima (i.e., once the algorithm identifies a better candidate, the process terminates). However, CodeImprove will evolve for multiple generations i.e., in our case, is three to find the best candidate; (3) In terms of correcting mispredictions CodeImprove performs the best (i.e., CSR up to 39.9%); (4) Although CodeImprove-rand and CodeImprove-HC shows lower values of MCR, note that its CSR values are really low, therefore, unable to correct mispredictions in a large scale; and (5) In conclusion, CodeImprove is a stable approach to adapt program inputs.

> **RQ3** - How effective to convert out-of-scope data to become in-scope data?
>
> CodeImprove is better at correcting out-of-scope inputs compared to other search algorithms such as random search and hill climbing.

D. RQ4: Influence of Hyper-parameters

Process: We studies the influence of number of transformation operators (N) applied during the crossover for each candidate. We investigated the effectiveness and efficiency of Codeimprove under different settings, i.e., N = {2, 5, 10, 15}. We applied variable renaming and one random transformation operator from Table II for $N = 2$, operators 1-5 for N=5, operator 1-10 for N=10, and operators 1-15 for N=15.

Results: Table VII show the results of CodeImprove under the hyper-parameter settings of N in terms of CSR, MCR, and RI. We observed that as N increases, more mispredicted inputs can be corrected, resulting in a larger RI. Meanwhile more correctly predicted inputs are identified as mispredicted ones due to more transformations leading to slightly higher MCR. When N = 5 for CodeBERT on vulnerability detection task, CodeImprove achieved a higher CSR (i.e., 40.3%) than N=10 (i.e., CSR of 38.7%), however the RI was lower than N =10 due to higher MCR value. Therefore, it is necessary to maintain the balance between CSR and MCR during the transformation. Moreover, CodeImprove has achieved greater performance on all cases for N, indicating its practical applicability.

> **RQ4** - How sensitive is CodeImprove's performance under different experimental setups?
>
> CodeImprove is effective when different number of transformation operators are applied during the crossover (i.e., RI shows 48.4% - 51.28% for vulnerability detection and 26.1%-32.8% for defect detection tasks).

TABLE VI: Effectiveness of Search Strategy for Input Adaptation

Experiment	Model	Vulnerability Detection						Defect Prediction					
		A(%)	P(%)	R(%)	F1(%)	RI(%↑)	CSR(%↑)/MCR(%↓)	A(%)	P(%)	R(%)	F1(%)	RI(%↑)	CSR(%↑)/MCR(%↓)
CodeImprove-rand	CodeBERT	64.86	64.64	51.87	57.55	12.3	11.1/2.16	82.17	82.08	82.18	81.69	3.2	8.6/0.8
	RoBERTa	63.57	59.84	62.95	61.35	20.7	6.11.1/1.5	80.95	80.85	80.95	80.39	20.9	17.3/1.36
	GraphCodeBERT	63.21	62.45	49.96	55.51	9.9	7.42/0.82	82.12	82.18	82.12	81.75	2.8	13.9/1.4
CodeImprove-HC	CodeBERT	63.79	63.43	50.03	55.94	6.1	6.8/1.9	82.22	82.24	82.27	81.79	5.03	8.3/0.7
	RoBERTa	62.70	58.97	61.84	60.37	11.7	6.7/0.7	80.54	80.44	80.54	79.98	11.8	11.9/1.1
	GraphCodeBERT	62.99	62.33	49.17	54.97	7.2	5.15/0.5	82.04	82.03	82.03	81.69	1.7	10.6/1.1
CodeImprove	CodeBERT	**71.52**	**70.79**	**64.70**	**67.61**	**51.28**	**39.9/4.5**	**84.5**	**84.2**	**84.4**	**83.99**	**44.2**	**32.8/1.0**
	RoBERTa	**68.81**	**64.67**	**70.76**	**67.58**	**74.4**	**36.2/1.96**	**82.91**	**82.99**	**82.91**	**82.36**	**65.6**	**32.13/1.0**
	GraphCodeBERT	**65.26**	**64.22**	**55.06**	**59.29**	**35.1**	**23.1/1.9**	**83.45**	**83.55**	**83.45**	**83.09**	**21.1**	**27.2/1.4**

TABLE VII: Sensitivity Study

Experiment	Model	Vulnerability Detection		Defect Prediction	
		RI(%↑)	CSR(%↑)/MCR(%↓)	RI(%↑)	CSR(%↑)/MCR(%↓)
N=1	CodeBERT	48.4	37.6/4.32	28.1	26.1/1.33
	RoBERTa	73.0	35.1/1.78	55.5	29.71/1.25
	GraphCodeBERT	32.9	21.6/1.8	18.5	25.6/1.53
N=5	CodeBERT	50.8	40.3/5.0	30.2	27.1/1.31
	RoBERTa	71.1	34.3/1.78	56.94	30.1/1.23
	GraphCodeBERT	34.2	21.4/1.6	18.95	25.9/1.55
N=10	CodeBERT	51.1	38.7/3.83	42.7	32.2/1.31
	RoBERTa	71.3	35.4/2.14	59.68	30.3/1.12
	GraphCodeBERT	35.1	22.5/1.8	19.9	26.6/1.5
N=15	CodeBERT	51.28	39.9/4.51	44.2	32.8/1.0
	RoBERTa	74.4	36.2/2.14	65.6	32.3/1.08
	GraphCodeBERT	35.1	23.1/1.9	21.1	27.1/1.4

E. RQ5: Semantic Preservation in CodeImprove's Program Transformation

Process: The objective of this RQ is to examine whether the adapted programs maintain the semantics of the original inputs. We investigate the effectiveness of applying semantic preserving program transformations to adapt out-of-scope inputs. Based on our investigation we provide an example in Figure 3. Additional examples are on our project website due to space restrictions [44].

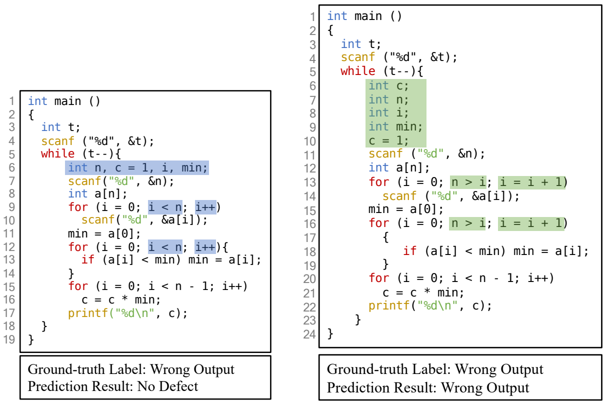
Figure 3 illustrates how CodeImprove revises a misprediction. As illustrated in Figure 3a, the CodeBERT model incorrectly predicts the input to *no defect*, although the ground truth label is actually *wrong output*. During the validation phase, CodeImprove identifies this out-of-scope input with a validity score of 0.0221, significantly lower than our threshold of 0.2. To adapt this input, CodeImprove applies semantic preserving transformations. These transformations include splitting lines, changing code order, splitting declarations, and separating variable assignments at line 6. Additionally, the relational and incremental operators were altered in lines 9 and 12. These changes create a syntax shift that affects the model's interpretation, resulting in different embeddings. The transformed version is shown in Figure 3b. After these transformations, the model correctly predicts the label as *wrong output* with an improved validity score of 0.7377.

RQ5 - Does program transformations of CodeImprove preserve semantics?

CodeImprove can generate semantic preserving program transformations.

F. RQ6: Overhead of CodeImprove

Process: To apply CodeImprove in real-time, we compute the overhead of applying transformations for an input. We calculate the TPS, which measures the number of transformations applied per second by CodeImprove. This metric is averaged across all N variants of CodeImprove studied under RQ4. Additionally, we evaluate both offline and online overhead. The offline stage involves the sub-model training, while the online stage measures the time required to adapt an input with CodeImprove. Table VIII shows the statistics of TPS, and Figure 4 shows the time overhead of CodeImprove.



```
1  int main ()
2  {
3    int t;
4    scanf ("%d", &t);
5    while (t--){
6        int n, c = 1, i, min;
7        scanf("%d", &n);
8        int a[n];
9        for (i = 0; i < n; i++)
10         scanf("%d", &a[i]);
11       min = a[0];
12       for (i = 0; i < n; i++){
13         if (a[i] < min) min = a[i];
14       }
15       for (i = 0; i < n - 1; i++)
16         c = c * min;
17       printf("%d\n", c);
18   }
19 }
```

Ground-truth Label: Wrong Output
Prediction Result: No Defect

(a) Before Transformation

```
1  int main ()
2  {
3    int t;
4    scanf ("%d", &t);
5    while (t--){
6        int c;
7        int n;
8        int i;
9        int min;
10       c = 1;
11       scanf ("%d", &n);
12       int a[n];
13       for (i = 0; n > i; i = i + 1)
14         scanf ("%d", &a[i]);
15       min = a[0];
16       for (i = 0; n > i; i = i + 1)
17         {
18             if (a[i] < min) min = a[i];
19         }
20       for (i = 0; i < n - 1; i++)
21         c = c * min;
22       printf("%d\n", c);
23     }
24 }
```

Ground-truth Label: Wrong Output
Prediction Result: Wrong Output

(b) After Transformation

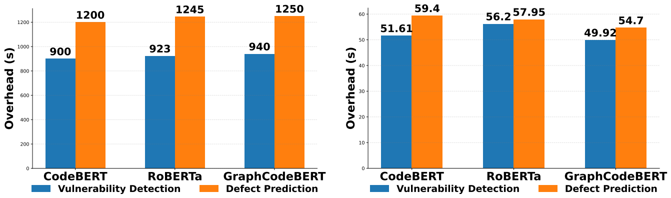
Fig. 3: An example of a M→C transformation

TABLE VIII: TPS of Codeimprove

Experiment	Vulnerability Detection			Defect Prediction		
	CodeBERT	RoBERTa	GraphCodeBERT	CodeBERT	RoBERTa	GraphCodeBERT
Overhead	1.51	1.2	1.43	2.04	1.95	1.75

Based on Table VIII, CodeImprove is capable of applying transformations at a rate of 1.2 TPS to 2.04 TPS for each SE task across all code models. Figure 4b confirms that CodeImprove takes approximately 49.92s to 59.4s to adapt an input across all models. We plan to further minimize the transformation times in future work. Moreover, CodeImprove is more efficient and offers a practical, scalable solution to enhance model performance without the significant cost and time investment required by traditional methods such as retraining and replacement.

It is important to note that the sub-model training procedure is treated as an offline stage, minimizing its impact on overall performance. Figure 4a shows that training a sub-model takes around 900s to 940s for the vulnerability detection and 1200s to 1250s for the defect prediction across all models on a machine with an NVIDIA GeForce GTX 1080 GPU. This process only needs to be done once and incurs significantly lower costs compared to regular retraining or fine-tuning.



(a) Offline time overhead (in second)

(b) Online time overhead (in seconds)

Fig. 4: Time Overhead of CodeImprove

RQ6 - What is the overhead of CodeImprove in adapting a program to DL models?

CodeImprove was highly efficient in adapting an out-of-scope input through semantic preserving program transformations in real-time (1.2TPS - 2.04TPS).

VII. Threat Analyses and Limitations

Our selection of the two subject datasets, namely, Devign, and CodeChef with their associated code models, might threaten the external validity of our experimental conclusions. We tried to alleviate this threat by following efforts: (1) the two datasets are very popular and have been widely used in relevant research [4], [52]–[54]; (2) their associated DL models are commonly used in SE tasks; (3) these datasets and models differ from each other by varying topics, labels (from two to four), and model accuracies (from 61.56% to 81.98%), which make these subjects diverse and representative. Therefore, our experimental conclusion should generally hold, although specific data could be inevitably different for other subject.

Threats to external validity may arise from the techniques selected for experimental comparisons, including uncertainty metrics [18]–[28] and dissector [19]. Due to inherent differences, existing methods for image data (SelfChecker [16] and InputReflector [17]) cannot be directly applied to code data. Therefore, we chose dissector as a baseline for input validation, as it identifies out-of-scope inputs similarly to our focus. Additionally, we sampled a subset of uncertainty metrics to assess their effectiveness in identifying out-of-scope inputs [18]–[28].

Our internal threat mainly comes from the lack of ground truths for distinguishing out-of-scope inputs from in-scope inputs, limitations on applying CodeImprove in different subjects, and applying different program transformation rules. We used mispredictions and correct predictions to simulate out-of-scope inputs and in-scope inputs respectively. Such estimation can be rough, however the logic may holds (RQ2). Regarding the lack of subject matter, our assumption was to investigate how our propose technique performed on different SE tasks and code models. Therefore, we select a subset of SE tasks and code models (RQ1, RQ3 and RQ4). In future, we plan to extend and apply CodeImprove on other subjects matter (e.g. clone detection, functionality classification etc.). CodeImprove uses 15 semantic-preserving transformation rules, but we plan to add more and update our website with new results. Due to the diversity of our subjects, we believe our conclusions are generally applicable, and CodeImprove can support future research.

VIII. Related Work

Input Validation for Deep Learning Model. Several work has been proposed to compute the trustworthiness of a DL model. Hendrycks et. al [20] propose a baseline for detecting misclassified samples. [18] propose re-calibration of probabilities on a held-out validation set. [19] proposed Dissector, to validate inputs by crossing-layer dissection. [16] proposed SelfChecker by leveraging Kernel Density Estimation(KDE) [56]. ConfidNet [25] is designed to learn the confidence criterion using True Class Probability for predicting failures. InputReflector [17] identifies failure-inducing inputs on Image data. In addition to the aforementioned techniques on code data, we show that existing uncertainty metrics [8], [18]–[20], [22], [24], [25] do not perform promising results on code data.

Related Work on Code Inputs. Several techniques [2], [6], [54], [57], [58] for generating adversarial code to challenge these models have been proposed in recent years. Tian et al. [52] claim they designed a code-difference-guided generation technique, which can improve the efficiency further. CodeDenoise [1] is the most advanced technique to improve the performance of deployed models without retraining, however, it can only relieve the noise introduced by different identifier names and consequential mispredictions. In contrast, CodeImprove leverages 15 unique transformation operators. To the best of our knowledge, CodeImprove is the first attempt to use the inference-time program transformation technique to enhance the performance of code models

IX. Conclusion

This paper proposes CodeImprove for validation and adaptation of out-of-scope inputs to become in-scope inputs for code models. CodeImprove employs a validity score metric by leveraging dropout based sub-model generation (DSMG) technique to distinguish out-of-scope inputs from in-scope inputs and applies semantic preserving program transformations on these inputs. Experimental evaluation confirmed that CodeImprove's effectiveness, highlighting its potential for broader application scenarios in input validation and adaptation.

Data Availability

Our artifacts are available at [29], containing datasets, code to reproduce the results in this paper. Our project website is available at [44]

Acknowledgments

This work was partially supported by NSF grants NSF CCF-2146443 and Amazon Trust AI Research Award. We extend our gratitude to Mirazul Haque for hiss guidance throughout this project.

References

[1] Z. Tian, J. Chen, and X. Zhang, "On-the-fly improving performance of deep code models via input denoising," arXiv preprint arXiv:2308.09969, 2023.

[2] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in Proceedings of the 44th International Conference on Software Engineering, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1482–1493. [Online]. Available: https://doi.org/10.1145/3510003.3510146

[3] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue, and Y. Xu, "Challenging Machine Learning-based Clone Detectors via Semantic-preserving Code Transformations," IEEE Transactions on Software Engineering, vol. 49, no. 5, pp. 3052–3070, May 2023.

[4] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation," arXiv preprint arXiv:2102.04664, 2021.

[5] Z. Yang, Z. Sun, T. Z. Yue, P. Devanbu, and D. Lo, "Robustness, security, privacy, explainability, efficiency, and usability of large language models for code," arXiv preprint arXiv:2403.07506, 2024.

[6] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," Proceedings of the ACM on Programming Languages, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[7] Q. Hu, Y. Guo, X. Xie, M. Cordy, M. Papadakis, L. Ma, and Y. Le Traon, "Codes: towards code model generalization under distribution shift," in International Conference on Software Engineering (ICSE): New Ideas and Emerging Results (NIER), 2023.

[8] Y. Li, S. Chen, and W. Yang, "Estimating predictive uncertainty under program data distribution shift," arXiv preprint arXiv:2107.10989, 2021.

[9] R. van Tonder and C. L. Goues, "Tailoring programs for static analysis via program transformation," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 824–834.

[10] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018, pp. 697–710.

[11] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," May 2019.

[12] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," Feb. 2020.

[13] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," 2021.

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, Pinto et al., "Evaluating Large Language Models Trained on Code," Jul. 2021.

[15] S. Yu, T. Wang, and J. Wang, "Data Augmentation by Program Transformation," Journal of Systems and Software, vol. 190, p. 111304, Aug. 2022.

[16] Y. Xiao, I. Beschastnikh, D. S. Rosenblum, C. Sun, S. Elbaum, Y. Lin, and J. S. Dong, "Self-checking deep neural networks in deployment," 2021.

[17] Y. Xiao, Y. Lin, I. Beschastnikh, C. Sun, D. S. Rosenblum, and J. S. Dong, "Repairing failure-inducing inputs with input reflection," in The 37th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2022.

[18] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," 2017.

[19] H. Wang, J. Xu, C. Xu, X. Ma, and J. Lu, "Dissector: Input validation for deep learning applications by crossing-layer dissection," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 727–738.

[20] D. Hendrycks and K. Gimpel, "A baseline for detecting misclassified and out-of-distribution examples in neural networks," 2018.

[21] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in international conference on machine learning. PMLR, 2016, pp. 1050–1059.

[22] 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.

[23] Y. Xiao and W. Y. Wang, "Quantifying uncertainties in natural language processing tasks," in Proceedings of the AAAI conference on artificial intelligence, vol. 33, no. 01, 2019, pp. 7322–7329.

[24] V. T. Vasudevan, A. Sethy, and A. R. Ghias, "Towards better confidence estimation for neural models," in ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2019, pp. 7335–7339.

[25] C. Corbière, N. Thome, A. Bar-Hen, M. Cord, and P. Pérez, "Addressing failure prediction by learning model confidence," Advances in Neural Information Processing Systems, vol. 32, 2019.

[26] R. M. Monarch, Human-in-the-Loop Machine Learning: Active learning and annotation for human-centered AI. Simon and Schuster, 2021.

[27] J. Steinhardt and P. S. Liang, "Unsupervised risk estimation using only conditional independence structure," Advances in Neural Information Processing Systems, vol. 29, 2016.

[28] C. E. Shannon, "A mathematical theory of communication," The Bell system technical journal, vol. 27, no. 3, pp. 379–423, 1948.

[29] CodeImprove, "Codeimprove repository." [Online]. Available: https://github.com/CodeImprove/CodeImprove

[30] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in 2017 ieee symposium on security and privacy (sp). Ieee, 2017, pp. 39–57.

[31] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in 2016 IEEE symposium on security and privacy (SP). IEEE, 2016, pp. 582–597.

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

[33] N. Papernot and P. McDaniel, "On the effectiveness of defensive distillation," arXiv preprint arXiv:1607.05113, 2016.

[34] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," Advances in neural information processing systems, vol. 30, 2017.

[35] Z. B. Zabinsky et al., "Random search algorithms," Department of Industrial and Systems Engineering, University of Washington, USA, 2009.

[36] B. Selman and C. P. Gomes, "Hill-climbing search," Encyclopedia of cognitive science, vol. 81, p. 82, 2006.

[37] M. Harman and B. F. Jones, "Search-based software engineering," Information and software Technology, vol. 43, no. 14, pp. 833–839, 2001.

[38] t. I. N. N. C. C. S. International Neural Network Society (INNS), R. Battiti, and F. Masulli, "Bfgs optimization for faster and automated supervised learning," in International Neural Network Conference: July 9–13, 1990 Palais Des Congres—Paris—France. Springer, 1990, pp. 757–760.

[39] J. Davis and M. Goadrich, "The relationship between precision-recall and roc curves," in Proceedings of the 23rd international conference on Machine learning, 2006, pp. 233–240.

[40] J. Gawlikowski, C. R. N. Tassi, M. Ali, J. Lee, M. Humt, J. Feng, A. Kruspe, R. Triebel, P. Jung, R. Roscher et al., "A survey of uncertainty in deep neural networks," Artificial Intelligence Review, vol. 56, no. Suppl 1, pp. 1513–1589, 2023.

[41] W. Ma, S. Liu, M. Zhao, X. Xie, W. Wang, Q. Hu, J. Zhang, and Y. Liu, "Unveiling code pre-trained models: Investigating syntax and semantics capacities," ACM Transactions on Software Engineering and Methodology, 2024.

[42] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2017, pp. 4700–4708.

[43] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[44] CodeImprove, "Codeimprove." [Online]. Available: https://codeimprove.github.io/CodeImprove-io/

[45] M. Kumar, D. M. Husain, N. Upreti, and D. Gupta, "Genetic algorithm: Review and application," Available at SSRN 3529843, 2010.

[46] S. Forrest, "Genetic algorithms," ACM computing surveys (CSUR), vol. 28, no. 1, pp. 77–80, 1996.

[47] D. Whitley, "A genetic algorithm tutorial," Statistics and computing, vol. 4, pp. 65–85, 1994.

[48] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," Advances in neural information processing systems, vol. 32, 2019.

[49] A. V. Phan and M. Le Nguyen, "Convolutional neural networks on assembly code for predicting software defects," in 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES), 2017, pp. 37–42.

[50] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Sep. 2020.

[51] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," Jul. 2019.

[52] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," 2023.

[53] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1482–1493.

[54] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, and Z. Jin, "Towards robustness of deep program processing models—detection, estimation, and enhancement," ACM Trans. Softw. Eng. Methodol., vol. 31, no. 3, apr 2022. [Online]. Available: https://doi.org/10.1145/3511887

[55] A. He, C. Luo, X. Tian, and W. Zeng, "A twofold siamese network for real-time object tracking," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 4834–4843.

[56] G. R. Terrell and D. W. Scott, "Variable kernel density estimation," The Annals of Statistics, pp. 1236–1265, 1992.

[57] 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.

[58] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U.-M. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," arXiv preprint arXiv:2103.11882, 2021.