深度学习系统的性能提升

陈俊洁 天津大学

科技生态圈峰会 + 深度研习

——1000 + 技术团队的共同选择

KEYLINK ing

K+峰会

K+峰会 深圳站
K+全球软件研发行业创新峰会
会议时间：2024.05.24-25

K+峰会 上海站
K+全球软件研发行业创新峰会
会议时间：2024.09.20-21

AiDD峰会

AiDD峰会 深圳站
AI+软件研发数字峰会
会议时间：2023.11.24-25

AiDD峰会 北京站
AI+软件研发数字峰会
会议时间：2024.07.19-20

AiDD峰会 深圳站
AI+软件研发数字峰会
会议时间：2024.11.15-16

演讲嘉宾

陈俊洁

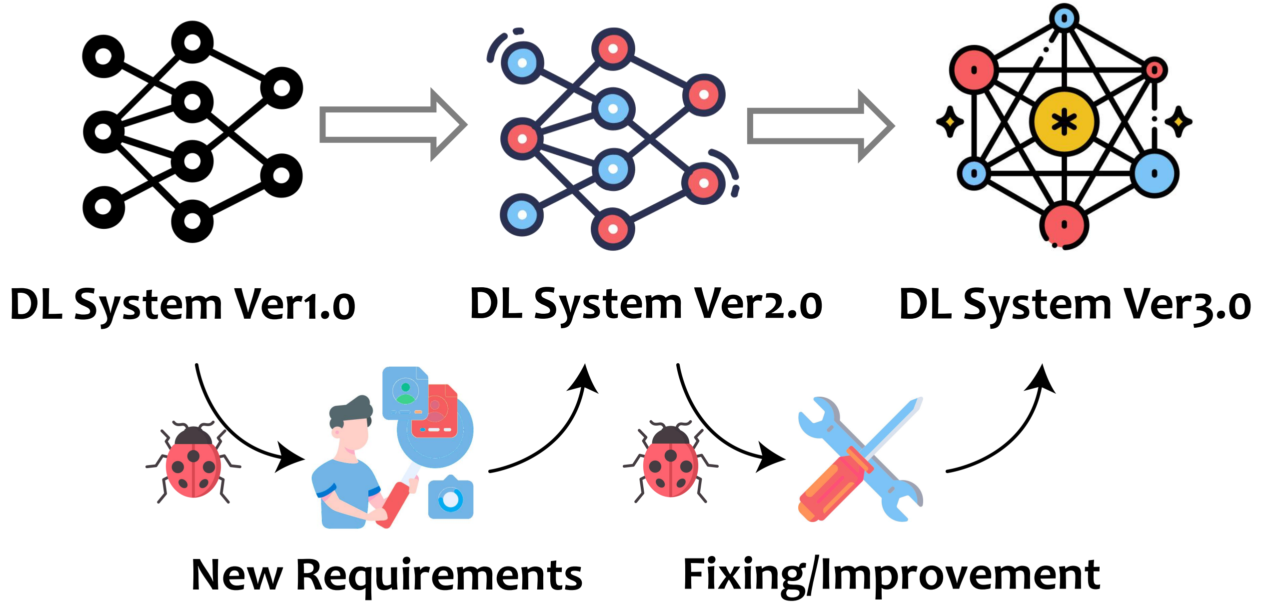
国家优青，天津大学特聘研究员，博导，软件工程团队负责人

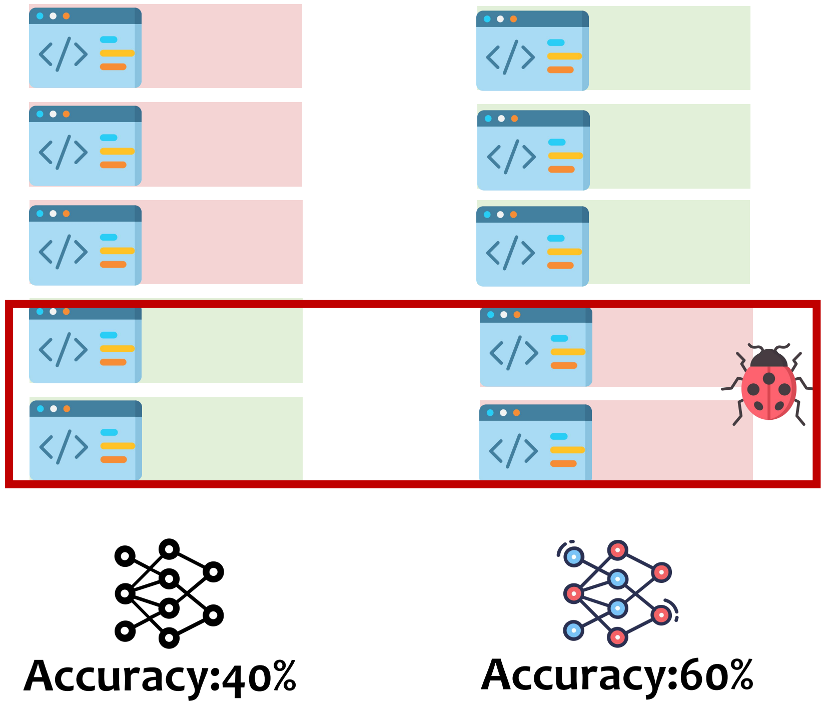
研究方向主要为基础软件测试、可信人工智能、数据驱动的软件工程等。荣获中国科协青年托举人才、CCF优博、电子学会自然科学一等奖等奖项。近年共发表学术论文70篇，其中CCF A类论文50余篇，获六项最佳论文奖（包括五项CCF-A类会议ACM SIGSOFT Distinguished Paper Award，以及一项CCF-B类会议ISSRE的Best Research Paper Award）。成果在华为、百度等多家知名企业落地。担任CCF-A类会议ASE 2021评审过程主席，Dagstuhl研讨会联合主席，以及软件工程领域全部CCF-A类会议的程序委员会成员等。

目录
CONTENTS

PART 01
深度学习系统的回归性能提升

Regression in Deep Learning Systems





It is important to detect regression faults!

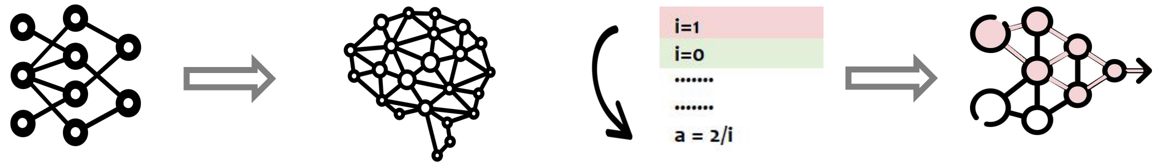
Existing Works Have Limitations

- Regression Fuzzing in Traditional Software
  - locates code changes in software evolution and utilize them to guide the regression fuzzing

1. DL Systems do not have explicit logical structures
2. Neuron change nearly affect all the neurons while code change only affect limited parts

- Fuzzing for Deep Learning Models
  - **DeepHunter:** Fuzzing guided by fine-grained neuron coverage **in a specific version**
  - **DiffChaser:** Detect disagreements in Quantization by generating test cases toward decision boundary

1. Ignore the difference between different versions of the DL models
2. Overlook important properties of the testing, such as fidelity and diversity.

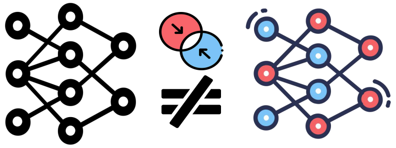


DL System Ver1.0 Accuracy=91%

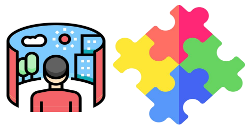
DL System Ver2.0 Accuracy=91.5%

Code Change

Neuron Change



Ignores Difference: Poor Fault-Triggering



Overlooks Fidelity & Diversity

SOTA techniques can not be directly adapt to solve this issue.

Our Idea of DRFuzz

Challenge 1: Fault-Triggering

Solution: **Amplifying the prediction difference** between versions through **effective mutation** to trigger more faults.

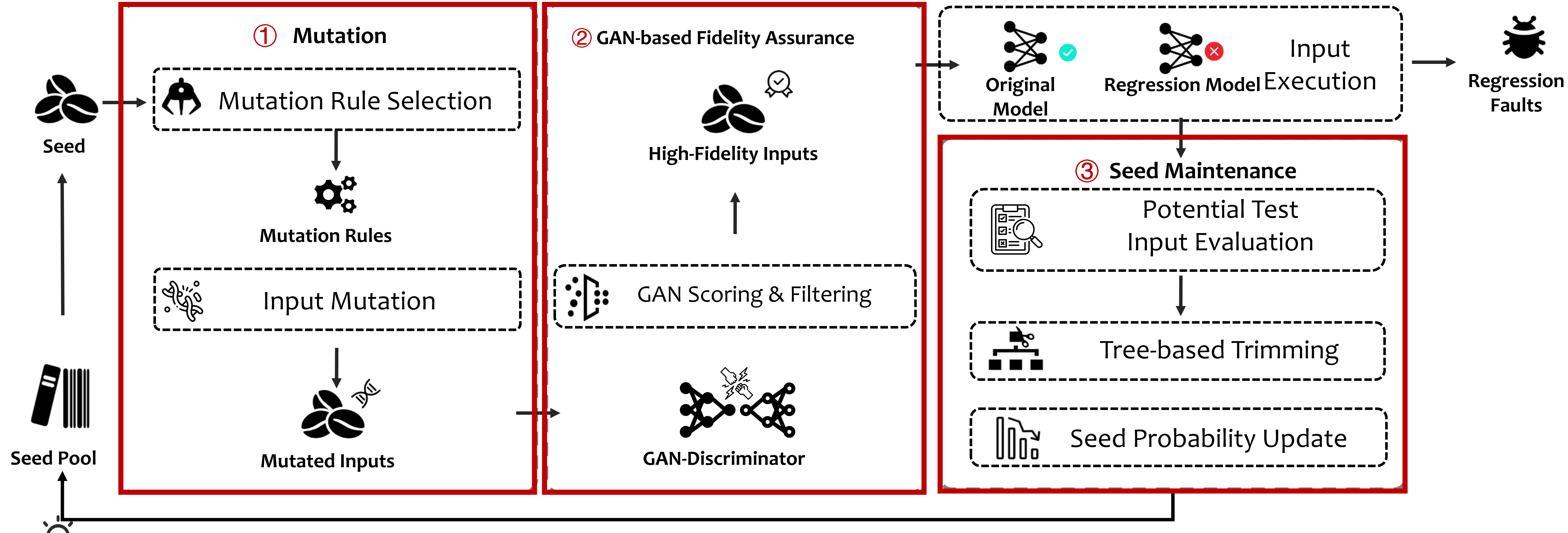
Challenge 2: Fidelity

Solution: Designing **GAN-based fidelity assurance** method to ensure fidelity.

Challenge 3: Diversity

Solution: Using **seed maintenance** to generate test inputs trigger different regression faults.

Our Approach: DRFuzz



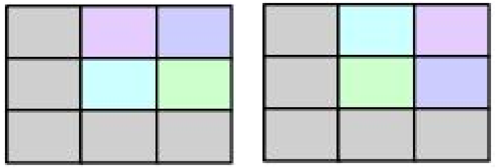
① Mutation	② GAN-based Fidelity Assurance	③ Seed Maintenance
* Generate **fault-triggering** test inputs	* Guarantee **fidelity** of test inputs	* Improve **Diversity** of test inputs

Mutation

Mutation Rules: We design **16** mutation rules:
Pixel-Level Mutation & **Image-Level Mutation**

1 Pixel-Level Mutation:

Pixel Coloring Reverse



Pixel Shuffling

2 Image-Level Mutation:

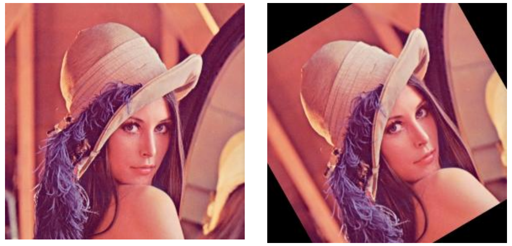


Image Rotating

Image Translation

MCMC-guided Mutation Rule Selection： Mutation rules that can generate test inputs with **high fidelity** and amplify the prediction difference towards **becoming a regression fault**, should be selected frequently.

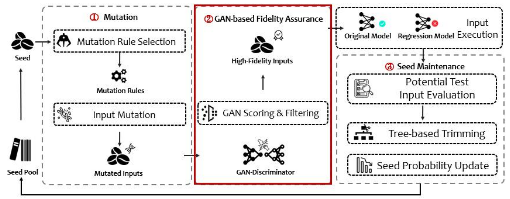
$$Reward = \frac{\#DiffTriggerInputs}{\#TotalSelect} \times \frac{\#FidelInputs}{\#TotalSelect}$$

Regression Fault-triggering **Fidelity**

$$< MR_1, MR_2, MR_3, \cdots\cdots, MR_n >$$

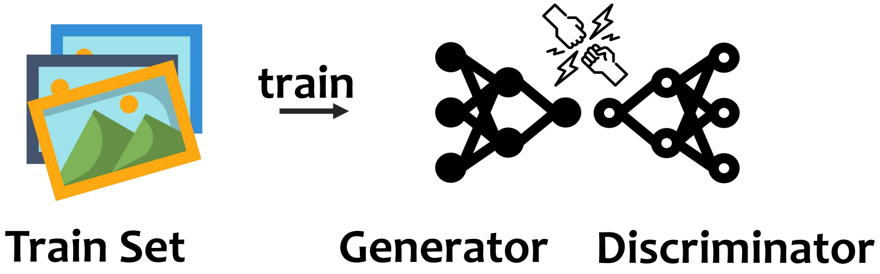
$$P(MR_a | MR_b) = min(1, (1-p)^{k_a - k_b})$$

GAN-based Fidelity Assurance

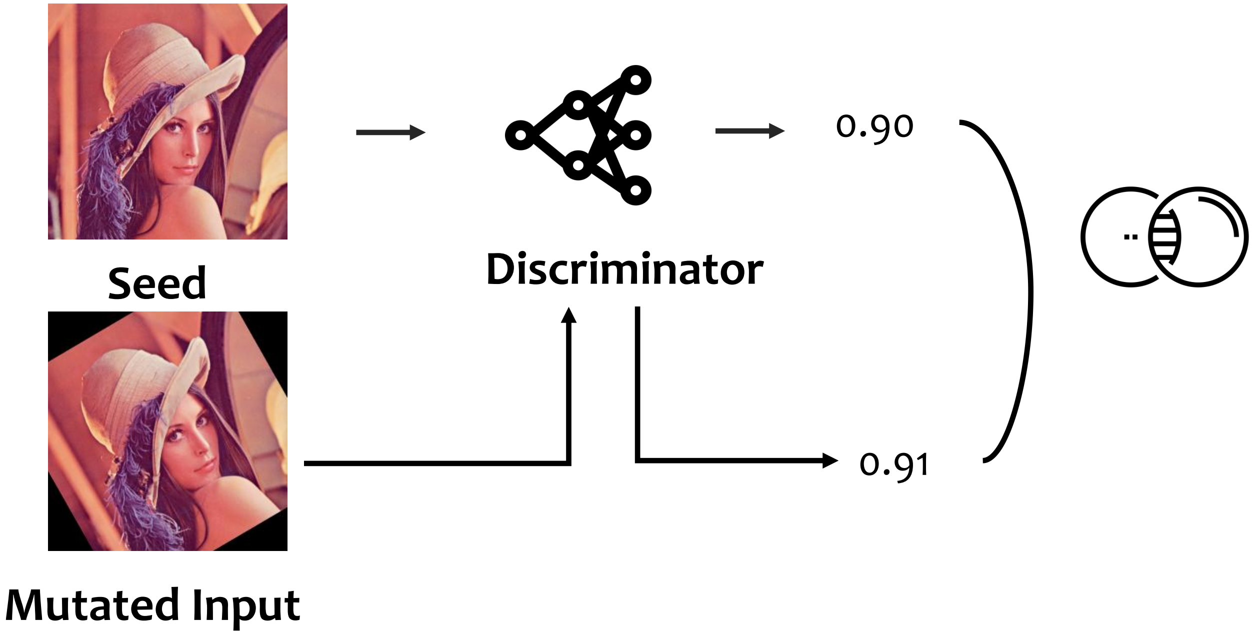


- Using **DCGAN (GAN-based approach)** preserve semantics to reducing discarding test inputs with **high fidelity** from image-level mutation rules.

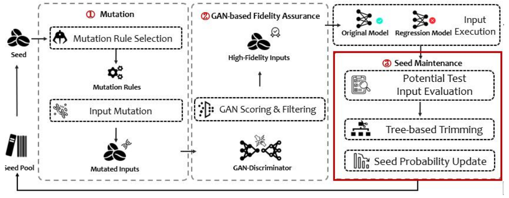
1 Training Phase:



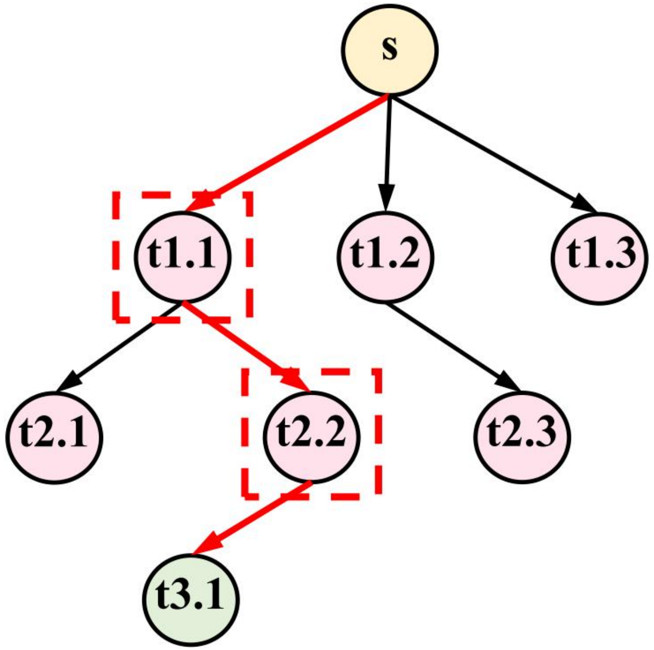
2 Predicting Phase:



Seed Maintenance

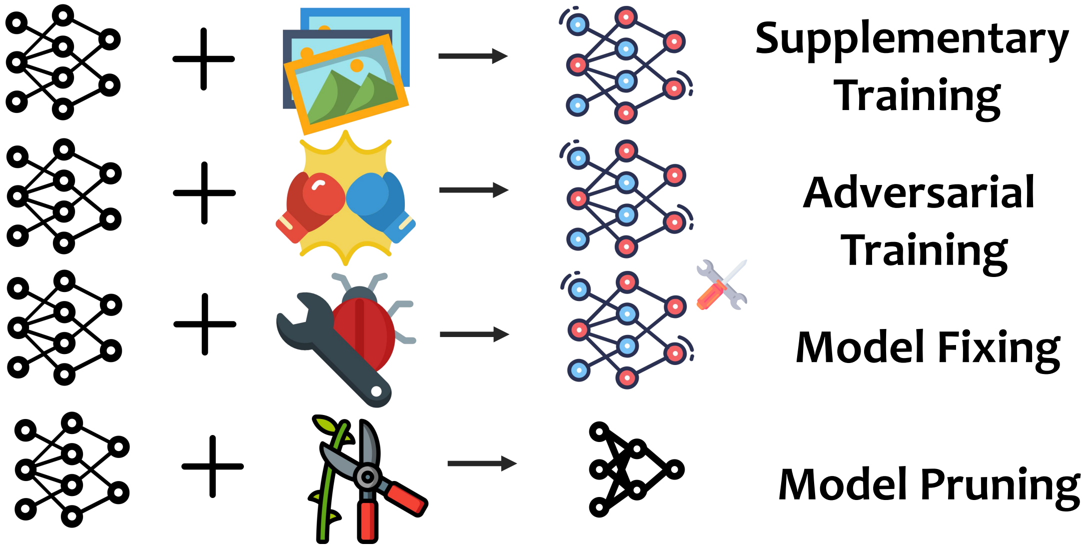


- **Tree-based Trimming** The Trimming process aims to trigger more **diverse** faulty behaviors by removing redundant seed to adjust seed selection probability.



Subjects and Regression Scenarios

Task	Name	Train Set	Test Set	Model
Digit Recognition	MNIST	60k	10k	LeNet5
Object Recognition	Cifar-10	60k	10k	VGG16
Clothes Recognition	FASHION-MNIST	60k	10k	AlexNet
Road Number Recognition	SVHN	73,257	26,032	ResNet18



The subjects are diverse, involving different tasks/models/regression scenarios.

RQ1: Effectiveness

Effectiveness on Different Regression Scenarios

Regression Scenario	Approach	#RFI	#RF	#Seed	#GF
SUPPLY	DiffChaser	12,489	991	846	18,529
	DeepHunter	3,450	1,832	1,402	26,854
	DRFuzz	**43,265**	**13,391**	**6,272**	**207,917**
ADV	DiffChaser	7,543	514	417	15,366
	DeepHunter	4,319	2,196	1,422	25,290
	DRFuzz	**45,620**	**13,545**	**6,198**	**252,035**
FIXING	DiffChaser	14,066	1,172	859	20,036
	DeepHunter	3,850	2,362	1,608	19,202
	DRFuzz	**76,555**	**19,359**	**7,267**	**228,039**
PRUNE	DiffChaser	56,211	2,983	2,015	67,656
	DeepHunter	8,210	3,752	2,152	30,200
	DRFuzz	**86,040**	**18,975**	**7,690**	**185,464**

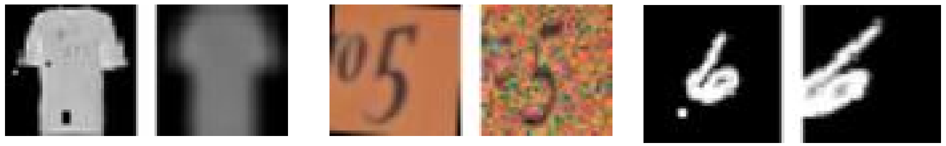
#RFI: Regression fault-triggering test inputs;
#RF: Dynamic diversity of test inputs;
[Seed, Faulty Behavior]
#Seed: Static Diversity of test inputs; (Seed)
#GF: general faults detected on the regression model;

DRFuzz outperforms the compared approaches stably on all the regression scenarios in terms of various metrics.

RQ2: Ablation

Ablation Experiment Results

Approach	#RFI	#RF	#Seed	#GF
DRFuzz	**70,093**	**16,464**	**6,942**	**231,675**
DRFuzz-r (No **MCMC**)	53,037	14,309	6,523	185,354
DRFuzz-NG (No **GAN**)	83,042	21,044	7,748	279,544
DRFuzz-NSM (No **Seed Maintenance**)	36,936	7,109	3,239	136,723



blurry **noisy** **over-changed**

DRFuzz (left) vs DRFuzz-NG (right)

The GAN-based Fidelity Assurance technique can filter out more than 20% of fault-triggering inputs with low fidelity

RQ3: Robustness Enhancement

Finetuning Accuracy on Different Regression Scenarios

Scenario	Train\Test	DiffChaser	DeepHunter	DRFuzz	$\Uparrow_{Acc}$(%)
SUPPLY	DiffChaser	67.11%	49.62%	53.35%	-0.97%
	DeepHunter	61.97%	72.83%	60.13%	-0.06%
	DRFuzz	**73.25%**	**74.09%**	**84.98%**	**0.34%**
ADV:CW	DiffChaser	72.96%	60.39%	58.84%	0.39%
	DeepHunter	71.84%	75.25%	64.12%	0.66%
	DRFuzz	**80.68%**	**79.88%**	**87.03%**	**0.81%**
ADV:BIM	DiffChaser	**77.47%**	50.39%	55.70%	-0.25%
	DeepHunter	64.13%	**68.43%**	58.50%	**0.04%**
	DRFuzz	76.87%	67.64%	**83.23%**	-0.04%
FIXING	DiffChaser	**64.25%**	50.70%	48.52%	-2.30%
	DeepHunter	55.13%	65.02%	53.99%	-1.38%
	DRFuzz	52.26%	**66.63%**	**77.72%**	**-0.12%**
PRUNE	DiffChaser	**75.61%**	55.55%	53.46%	3.66%
	DeepHunter	63.84%	**76.10%**	59.74%	3.95%
	DRFuzz	74.35%	70.37%	**81.53%**	**4.04%**

Finetuning DL models with the test inputs generated by DRFuzz can fix 77.72%~ 87.03% regression faults from DRFuzz and can defend 52.26%~ 80.68% attack from DiffChaser and 66.63%~ 79.88% attack from DeepHunter.

PART 02

深度代码模型的鲁棒性能力提升

Deep Code Models

Authorship Attribution

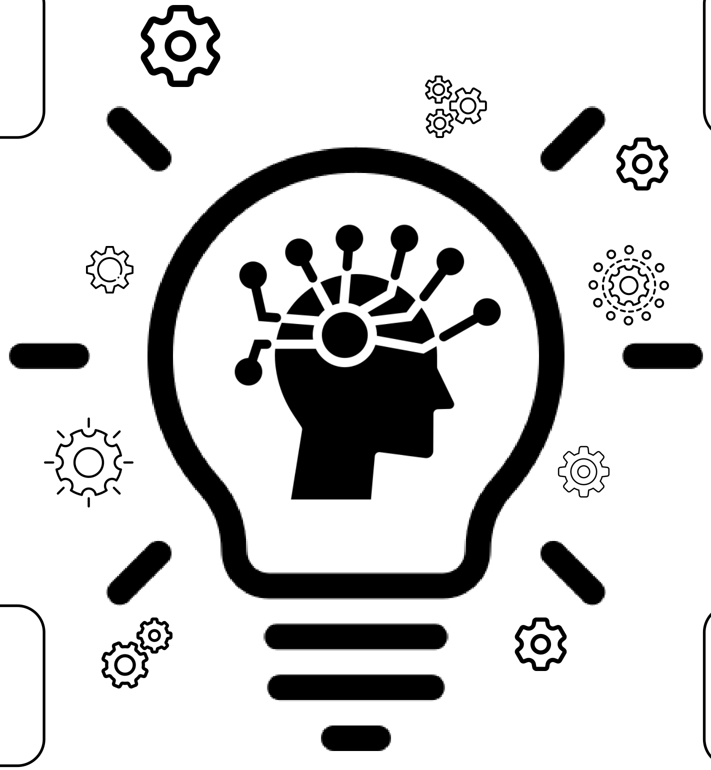
Functionality Classification

Clone Detection

Code Completion

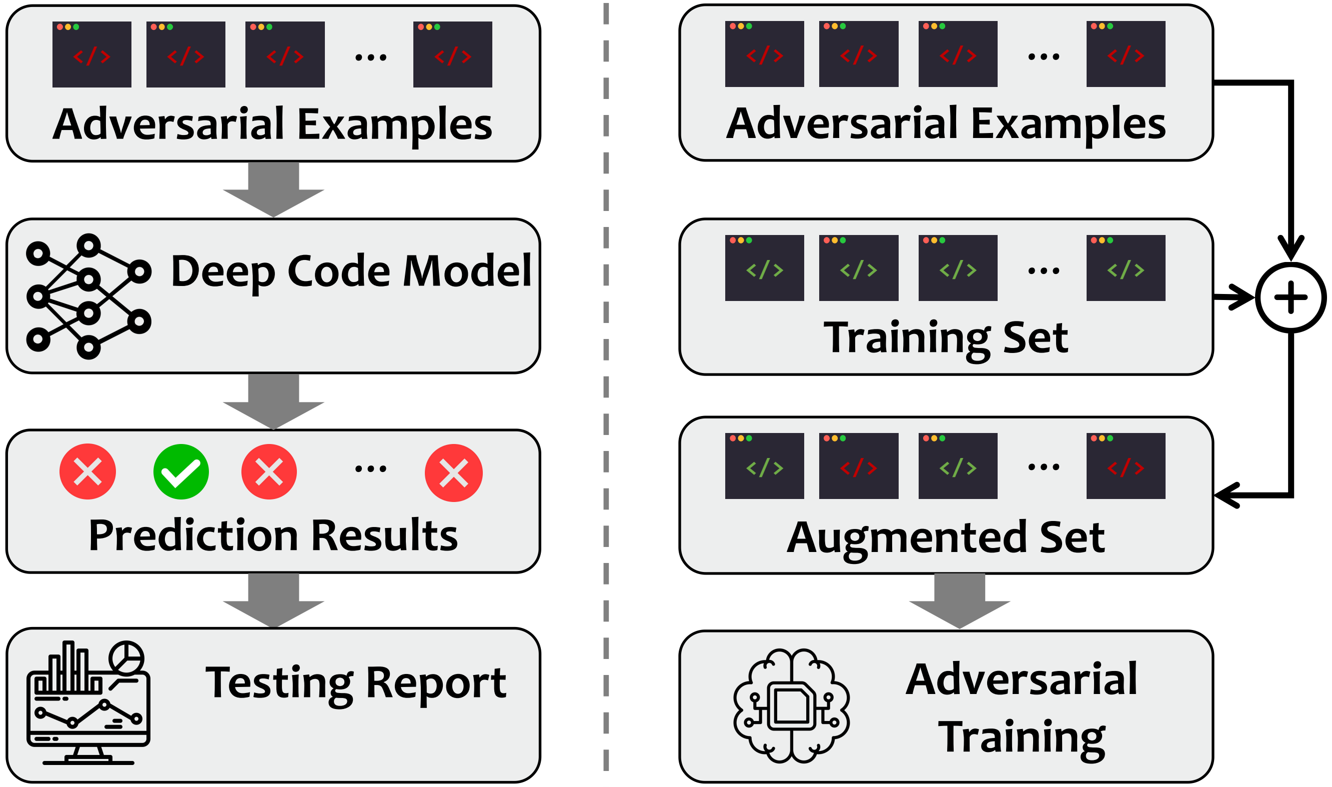
Code Generation

...



DL have been widely used to process source code!

Model Robustness is Critical



1 Testing

2 Enhancement

Unique Characteristics of Adversarial Examples for Deep Code Models:

1. The inputs (i.e., source code) for deep code models are discrete.
2. Source code has to strictly stick to complex grammar and semantics constraints.

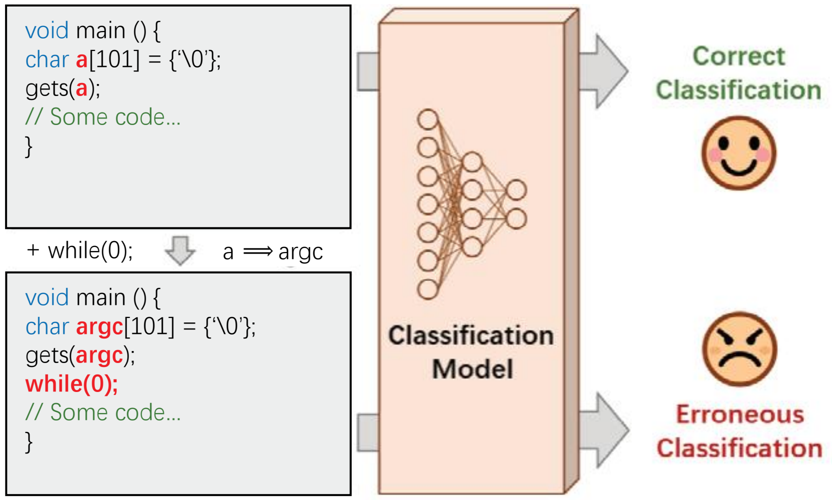
Conclusion: the existing adversarial example generation techniques in other areas are hardly applicable to deep code model

Adversarial examples are important to test & enhance model robustness!

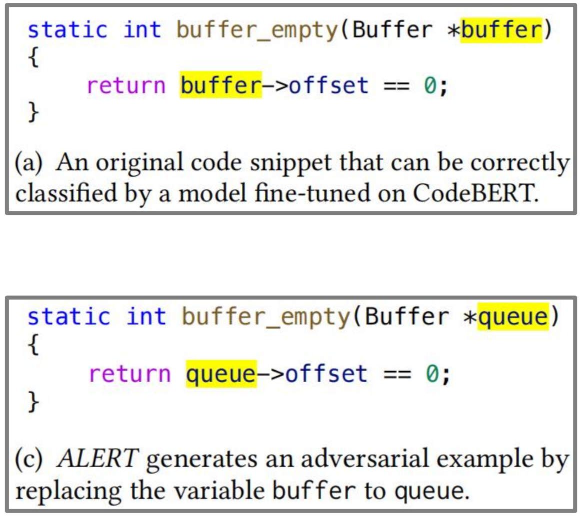
Deep Code Models are not Robust

Workflow of current techniques

- Designing semantic-preserving code transformation rules.
  - ➢ identifier renaming, etc.
- Searching ingredients from the space for transforming an original input to a semantic-preserving adversarial example.
  - ➢ Model prediction changes, etc.



Adversarial Example Generated by **CARROT**



Adversarial Example Generated by **ALERT**

Semantic-preserving adversarial examples can alter the prediction results!

Limitations

1 The Ingredient Space is Enormous

Almost Infinite

n^m

Complexity

```
void f1(int a[], int n){
  int i; int j; int k;
  for (i = 0; i < n; i++) {
    for (j=0;j<((n-i)-1);j++){
      if (a[j] > a[j + 1]){
        k = a[j];
        a[j] = a[j + 1];
        a[j + 1] = k;
      }
    }
  }
}
```

Ground-truth Label: *sort*
Prediction Results: *sort (96.52%)*

Identifiers	Ingredients
a	aa, array, at, area, au, am, alpha, ata, ad, auto, argc, ac, ar, ab ...
n	nu, sn, nc, len, cn, m, ns, pn, nb, nn, np, x, un, nan, fn, num, nt ...
i	it, chi, li, ui, ci, ia, ei, iii, oi, ini, ji, ai, phi, bi, gi, ie, ik ...
j	jump, js, jit, jc, jan, jp, ji, kj, bj, oj, adj, jl, aj, jj, je, ja ...
k	uk, ko, ku, kw, sk, key, ck, ak, mk, ky, tk, ks, kin, ke, km, rank ...

Target Input　　Identifiers　　Ingredients

2 Greedy model prediction changes guided search process is likely to fall into optimum.

3 Frequently invoking the target model could affect test efficiency via adversarial example generation.

SOTA techniques still suffer from effectiveness & efficiency Issues!

Novel Perspective: Code-Difference-Guided Adversarial Example Generation

Target Input

```
void f1(int a[], int n){
  int i; int j; int k;
  for (i=0; i<n; i++) {
    for (j=0; j<((n-i)-1); j++) {
      if (a[j]>a[j+1]){
        k = a[j];
        a[j] = a[j + 1];
        a[j + 1] = k;
      }
    }
  }
}
```

Ground-truth Label: *sort*
Prediction Results: *sort (96.52%)*

Reference Input

```
int f2(int t[], int len){
  int i; int j;
  i = 0; j = 0;
  while (len != 0) {
    t[i] = len % 10;
    len /= 10;
    i = i + 1;
  }
  while (j < i){
    if (t[j] != t[(i - j) - 1]) return 0;
    j = j + 1;
  }
  return 1;
}
```

Ground-truth Label: *palindrome*
Prediction Results: *palindrome (99.98%)*

Adversarial Example

```
void f3(int t[], int len){
  int i; int j; int k;
  i = 0;
  while (i < len) {
    j = 0;
    while (j < ((len - i) - 1)) {
      if (t[j] > t[j + 1]){
        k = t[j];
        t[j] = t[j + 1];
        t[j + 1] = k;
      } j = j + 1;
    } i = i + 1;
  }
}
```

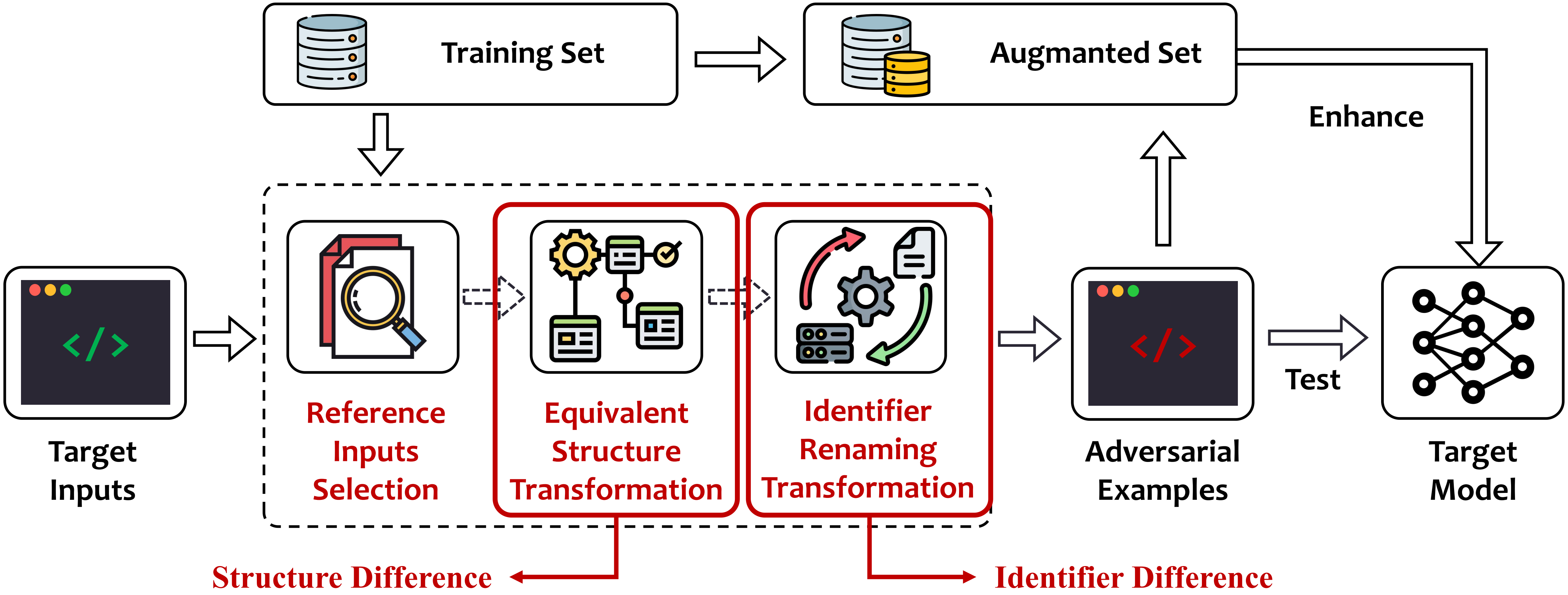
Ground-truth Label: *sort*
Prediction Results: *palindrome (90.88%)*

Have Different Semantics & Small Code Difference

Complexity: $n^m \rightarrow m^2$

Preserve the Semantics of f1 & Reduce Code Difference Brought by f2

Our Approach: CODA

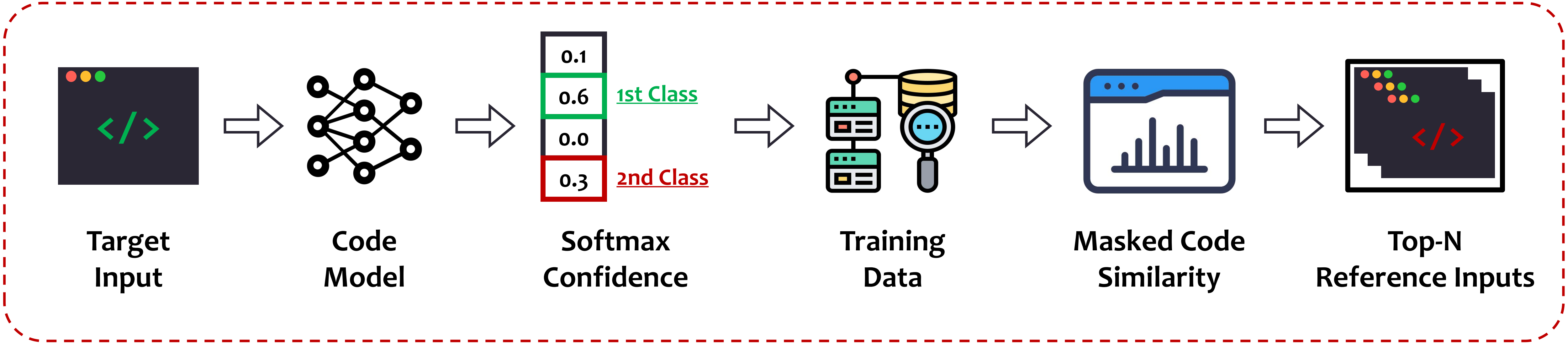


Overview of CODA

Reference Inputs Selection

- How to **select reference inputs** for reducing the ingredient space?

1. The prediction result is more likely to be changed from **1st Class** to **2nd Class**.
2. Smaller code difference can effectively limit the number of ingredients.



Equivalent Structure Transformation

- How to **reduce structure difference** between target input and reference inputs?

  1. applying equivalent structure transformations rule in a probabilistic way to reduce occurring distribution difference

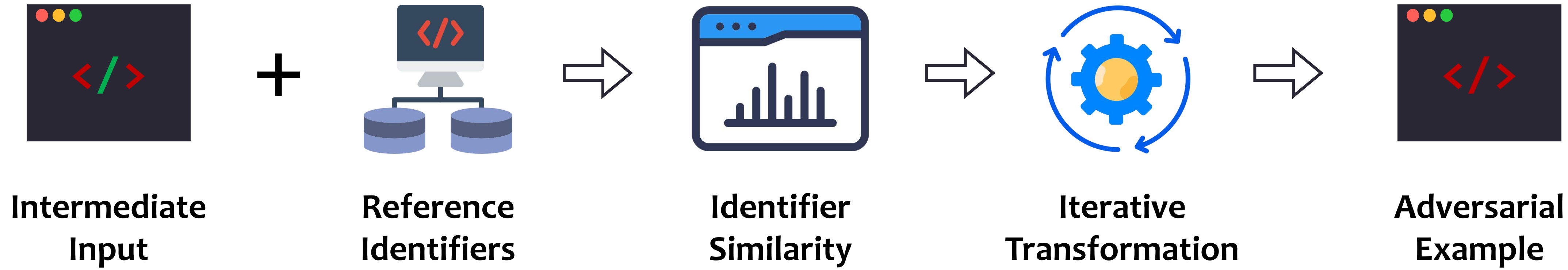
  2. considering all common kinds of code structures (i.e., *loop*, *branch*, and *sequential*).

Transformation	**Description**	**Example Before Transformation**	**Example After Transformation**
R_1-*loop*	equivalent transformation among `for` structure and `while` structure	`for ( i=0; i<9; i++ ) { Body; }`	`i=0; while ( i<9 ) { Body; i++; }`
R_2-*branch*	equivalent transformation between `if-else(-if)` structure and `if-if` structure	`if ( A ) { BodyA; } else if ( B ) { BodyB; }`	`if ( A ) { BodyA; } if ( !A && B ) { BodyB; }`
R_3-*calculation*	equivalent numerical calculation transformation, e.g., ++, --, +=, -=, *=, /=, %=, <<=, >>=, &=, |= , ^=	`i += 1;`	`i = i + 1;`
R_4-*constant*	equivalent transformation between a constant and a variable assigned by the same constant	`println("Hello, World!");`	`String i = "Hello, World!"; println(i);`

Identifier Renaming Transformation

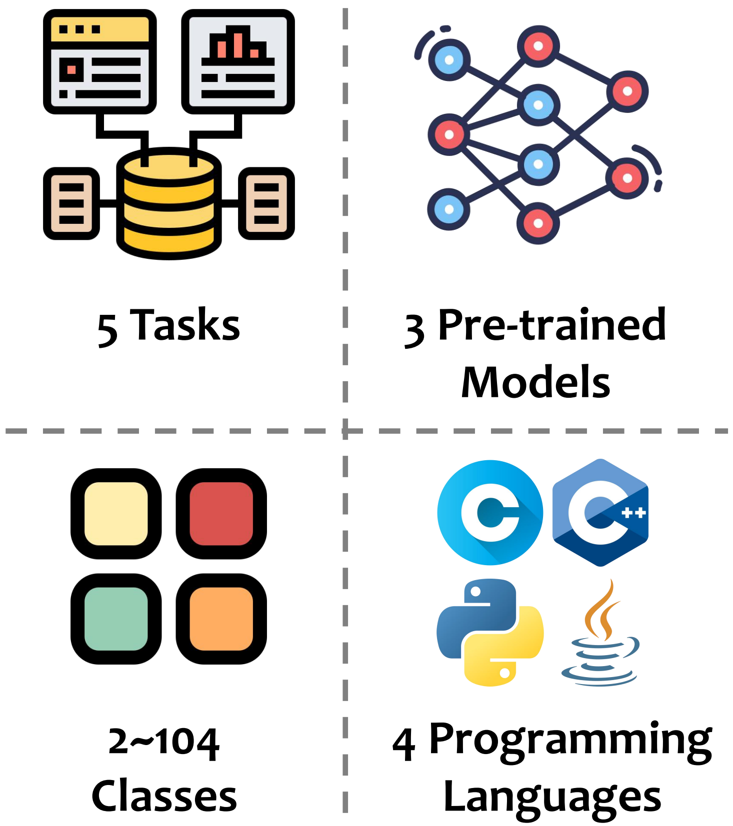
- How to **reduce identifier difference** between target input and reference inputs?

1. Identifier renaming transformation refers to replacing the identifier in the target input with the identifier in reference inputs.
2. To ensure the naturalness, we consider the semantic similarity between identifiers and design an iterative transformation process.



Subjects

Task	Train/Validate/Test	Class	Language	Model	Acc.
Vulnerability Prediction	21,854/2,732/2,732	2	C	CodeBERT GraphCodeBERT CodeT5	63.76% 63.65% 63.83%
Clone Detection	90,102/4,000/4,000	2	Java	CodeBERT GraphCodeBERT CodeT5	96.97% 97.36% 98.08%
Authorship Attribution	528/–/132	66	Python	CodeBERT GraphCodeBERT CodeT5	90.35% 89.48% 92.30%
Functionality Classification	41,581/–/10,395	104	C	CodeBERT GraphCodeBERT CodeT5	98.18% 98.66% 98.79%
Defect Prediction	27,058/–/6,764	4	C/C++	CodeBERT GraphCodeBERT CodeT5	84.37% 83.98% 81.54%



The subjects are diverse, involving different tasks/models/classes/languages.

RQ1: Effectiveness and Efficiency

Metric:

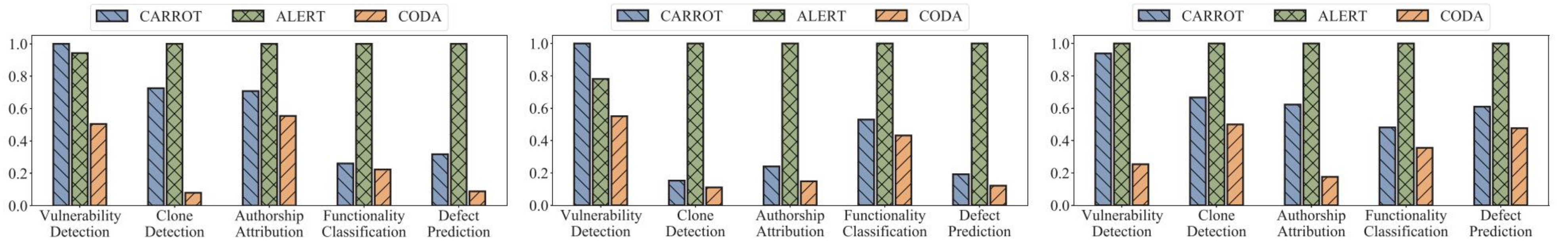
Rate of Revealed Faults ↑

Task	CodeBERT			GraphCodeBERT			CodeT5		
	CARROT	ALERT	CODA	CARROT	ALERT	CODA	CARROT	ALERT	CODA
Vulnerability Prediction	33.72%	53.62%	**89.58%**	37.40%	76.95%	**94.72%**	84.32%	82.69%	**98.87%**
Clone Detection	20.78%	27.79%	**44.65%**	3.50%	7.96%	**27.37%**	12.89%	14.29%	**42.07%**
Authorship Attribution	44.44%	35.78%	**79.05%**	31.68%	61.47%	**92.00%**	20.56%	66.41%	**97.17%**
Functionality Classification	44.15%	10.04%	**56.74%**	42.76%	11.22%	**57.44%**	38.26%	35.37%	**78.07%**
Defect Prediction	71.59%	65.15%	**95.18%**	79.08%	75.87%	**96.58%**	38.26%	35.37%	**78.07%**
Average	42.94%	38.48%	**73.04%**	38.88%	46.69%	**73.62%**	33.91%	40.99%	**70.96%**

CODA outperforms ALERT&CARROT in terms of the rate of revealed faults (RFR).

Metric:

Model Invocations ↓



(a) Model invocations on testing CodeBERT (b) Model invocations on testing GraphCodeBERT (c) Model invocations on testing CodeT5

CODA performs less time and fewer model invocations than ALERT&CARROT.

RQ2: Model Robustness Enhancement

Evaluation Set

Task	Model	Ori			CARROT			ALERT			CODA		
		CARROT	ALERT	CODA	CARROT	ALERT	CODA	CARROT	ALERT	CODA	CARROT	ALERT	CODA
Vulnerability Prediction	CodeBERT	62.96%	62.77%	**63.03%**	29.14%	21.11%	**29.69%**	23.43%	26.27%	**34.44%**	32.16%	31.73%	**38.82%**
	GraphCodeBERT	**62.99%**	62.88%	62.92%	12.37%	19.59%	**21.65%**	16.33%	17.35%	**23.71%**	25.77%	24.74%	**34.02%**
	CodeT5	63.69%	63.81%	**63.92%**	52.03%	39.76%	**82.03%**	42.26%	**49.11%**	44.26%	41.43%	45.52%	**52.54%**
Clone Detection	CodeBERT	97.39%	96.45%	**97.45%**	83.15%	42.31%	**94.44%**	52.65%	72.46%	**75.32%**	38.51%	71.45%	**89.78%**
	GraphCodeBERT	97.01%	97.22%	**97.43%**	75.00%	66.67%	**77.50%**	79.17%	84.29%	**92.31%**	35.71%	57.69%	**92.97%**
	CodeT5	97.73%	97.14%	**98.10%**	67.77%	57.63%	**75.85%**	69.94%	64.36%	**81.63%**	42.15%	51.74%	**79.88%**
Authorship Attribution	CodeBERT	90.55%	89.39%	**90.91%**	**45.06%**	40.67%	41.03%	51.25%	56.25%	**58.82%**	45.67%	43.33%	**76.47%**
	GraphCodeBERT	89.39%	88.72%	**90.35%**	**81.75%**	67.08%	72.40%	79.41%	78.67%	**100.00%**	45.59%	80.39%	**84.75%**
	CodeT5	92.43%	92.68%	**93.03%**	70.95%	65.91%	**73.48%**	55.73%	71.88%	**76.44%**	44.31%	52.56%	**72.37%**
Functionality Classification	CodeBERT	98.11%	98.52%	**98.56%**	**83.46%**	72.80%	81.51%	70.83%	71.75%	**79.41%**	78.92%	71.18%	**95.43%**
	GraphCodeBERT	98.48%	98.55%	**98.72%**	67.53%	75.19%	**77.27%**	32.04%	52.62%	**62.98%**	91.22%	90.81%	**93.08%**
	CodeT5	97.92%	98.46%	**98.63%**	25.31%	21.33%	**27.36%**	41.07%	57.14%	**57.42%**	24.87%	59.58%	**63.76%**
Defect Prediction	CodeBERT	83.50%	84.16%	**84.44%**	52.73%	25.81%	**66.03%**	74.88%	75.87%	**83.12%**	76.86%	68.66%	**85.36%**
	GraphCodeBERT	83.34%	84.00%	**84.53%**	68.20%	48.54%	**74.88%**	52.73%	**63.91%**	59.45%	67.08%	68.66%	**76.14%**
	CodeT5	80.92%	81.32%	**81.57%**	31.48%	34.08%	**37.73%**	31.75%	42.22%	**55.77%**	54.45%	54.18%	**73.83%**
Average		86.43%	86.40%	**86.91%**	56.40%	46.57%	**62.19%**	51.56%	58.94%	**65.67%**	49.65%	58.15%	**73.95%**

Augmented Training Set

Metric:

Accuracy ↑

CODA helps enhance the model robustness more effectively than ALERT&CARROT, in terms of reducing faults revealed by the adversarial examples.

RQ3: Contribution of Main Components

We constructed three variants of CODA:

- w/o RIS (Referrence Inputs Selection)
- w/o EST (Equivalent Structure Transformation)
- w/o CDG (Code Difference Guidance in EST)
- w/o IRT (Identifier Renaming Transformation)

Metric:

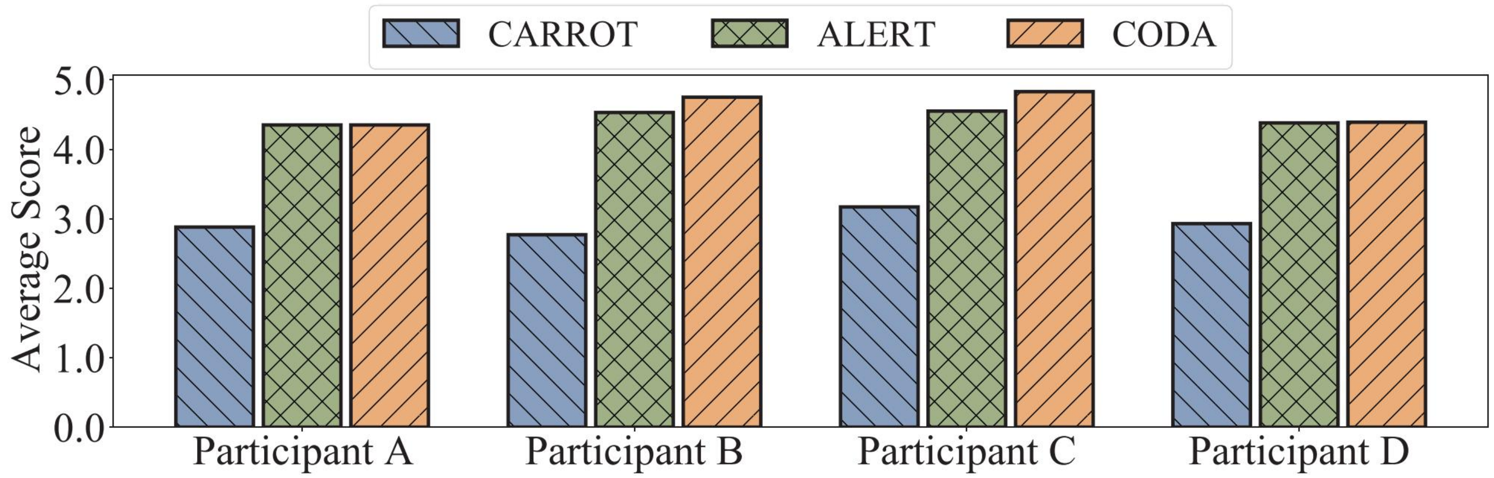
Rate of Revealed Faults ↑

Model	w/o RIS	w/o EST	w/o CDG	w/o IRT	CODA
CodeBERT	30.83%	62.73%	63.08%	35.14%	**73.04%**
GraphCodeBERT	29.49%	62.41%	61.98%	26.24%	**73.62%**
CodeT5	26.75%	50.74%	57.98%	38.21%	**70.96%**

All the three components make contributions to the overall effectiveness of CODA.

RQ4: Naturalness of Adversarial Examples

User Study (5-point Likert scale)



4 participates

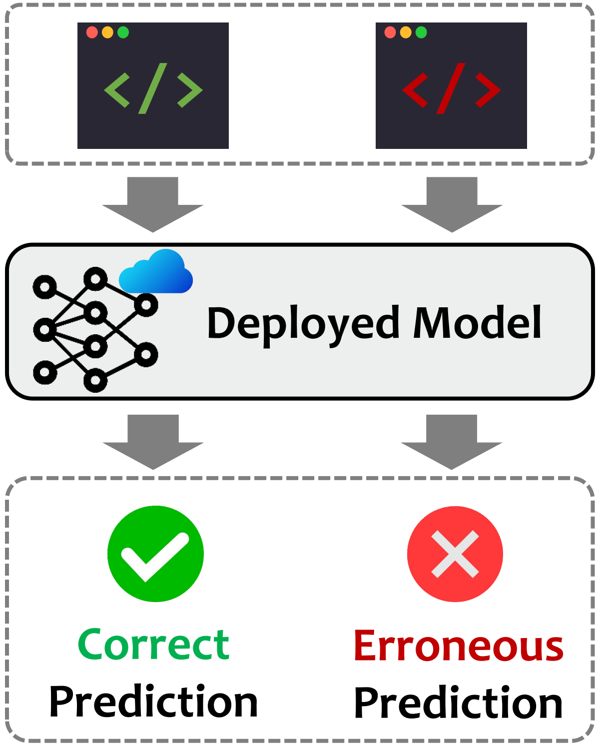
450 code snippets

The adversarial examples generated by CODA are natural closely to the naturalness-aware ALERT.

PART 03

深度代码模型部署后性能即时提升

Performance Issues with Deployed Deep Code Models



Accuracy < 100%

Existing strategies

1. **Designing more advanced networks** for retraining models
2. **Incorporating more data** for fine-tuning models

Limitations

1. **Time-consuming** caused by manual labeling & heavy computations
2. **Largely inexplicable** caused by complex parameters and datasets

Challenges in enhancing deployed model performance

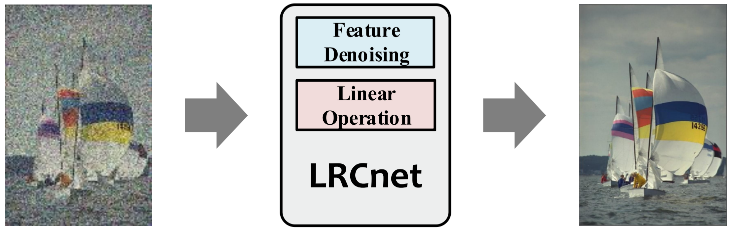
It's crucial to improve the performance of deployed deep code models!

Many Mispredictions are Caused by Noise in Inputs

Denoising in image processing field [1]

Reason:
complex environment, image quatization …

Formate:
continuous pixel values

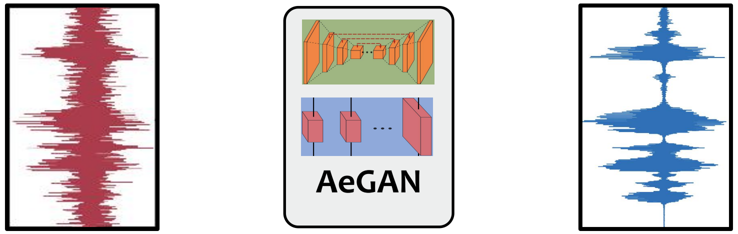


Noisy Image — Denoised Image

Denoising in speech recognition field [2]

Reason:
background noise, difference speaker …

Formate:
continuous signal values



Noisy Speech — Denoised Speech

Advantages of Input Denoising

1. Improving the model performance **on-the-fly**
2. **Retraining-free,** efficiency boost
3. **Enhancing explainable ability** of technique for correcting mispredictions

Limitations for Denoising Code

1. Denoising in **Continuous Space** vs. **Discrete Inputs**
2. **Complex syntactic** & **semantic constraints** in Code

[1] Ren J, Zhang Z, et al. "**Robust low-rank convolution network for image denoising.**" MM 2022.
[2] Abdulatif S, Armanious K, et al. "**Aegan: Time-frequency speech denoising via generative adversarial networks.**" EUSIPCO 2022.

Input Denoising for Deep Code Models

```python
def sort (x_list, y_length):
  a1_selection = 0
  flag = True
  while flag:
    flag = False
      for i in range(1, y_length):
        j = y_length - i
        if x_list[j] < x_list[j-1]:
          x_list[j], x_list[j-1] = \
            x_list[j-1], x_list[j]
          flag = True
          a1_selection += 1
  return x_list, a1_selection
```

Ground-truth Label: Bubble Sort
Prediction Result: Selection Sort
Noisy Identifier: `a1_selection`

(1) Noisy Code

```python
def sort (x_list, y_length):
  count = 0
  flag = True
  while flag:
    flag = False
    for i in range(1, y_length):
      j = y_length - i
      if x_list[j] < x_list[j-1]:
        x_list[j], x_list[j-1] = \
          x_list[j-1], x_list[j]
        flag = True
        count += 1
  return x_list, count
```

Ground-truth Label: Bubble Sort
Prediction Result: Bubble Sort
Denoised Identifier: `count`

(2) Denoised Code

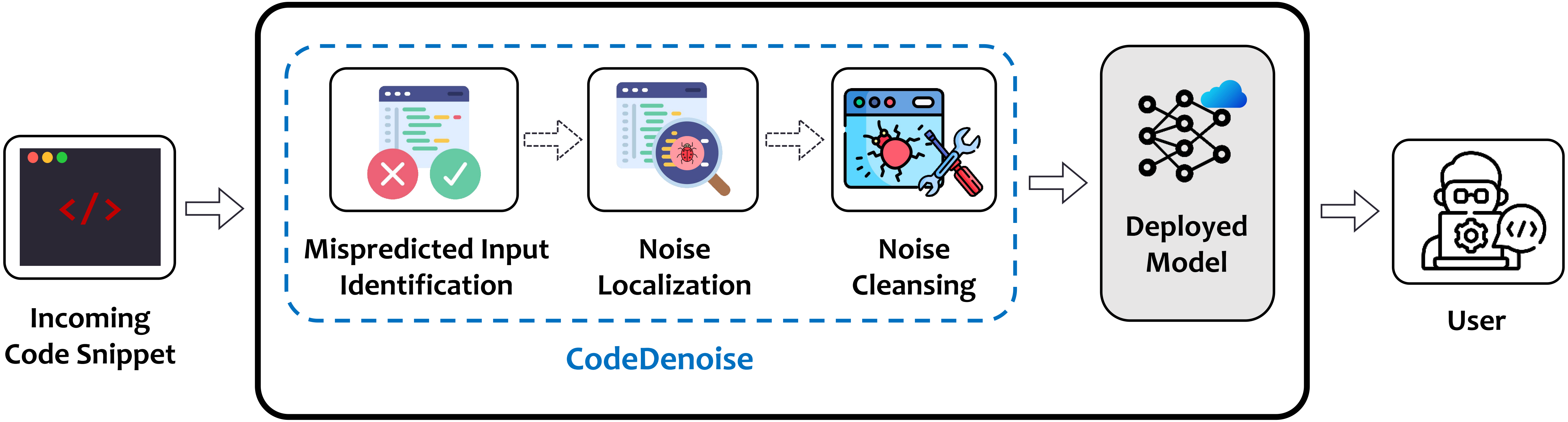
Noisy identifiers: the identifier makes the largest contribution to the misprediction.

This motivates the potential of on-the-fly improving performance of (deployed) deep code models through identifier-level input denoising.

Challenges

1. How to **identify mispredicted inputs** from the incomming code snippets?
2. How to **localize noise** (identifier-level) resulting in misprediction from a given code snippet?
3. How to **cleanse noise** to make the code snippet be predicted correctly?

Overview of CodeDenoise



The usage of CodeDenoise in practice:

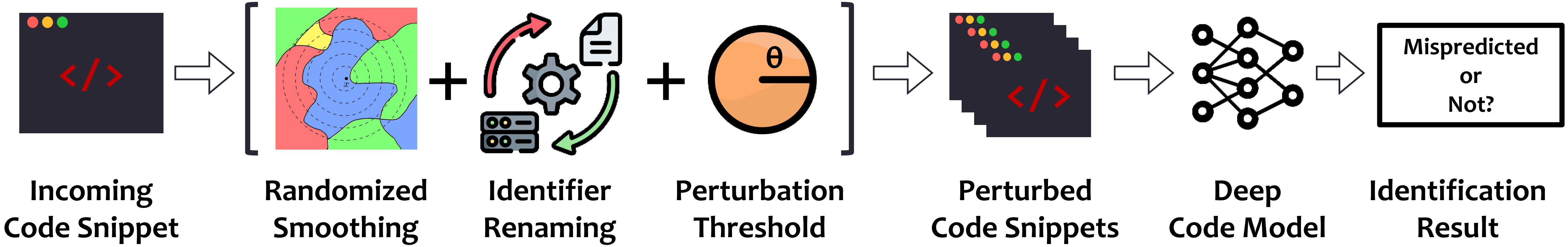
- We treat CodeDenoise as a **post-processing module** and **intergrate it with the deployed code model** as a system for making predictions in practice.

Mispredicted Input Identification

C1 - How to **identify mispredicted inputs** from the incoming code snippets?

1. In the field of CV, ***randomized smoothing*** is widely used to certify the classification result of a given image by checking the results of randomly perturbed images in the neighborhood.

2. To design adapted randomized smoothing for deep code models, we should:
(1) define the perturbation strategy (2) and control the perturbation degree on input code.

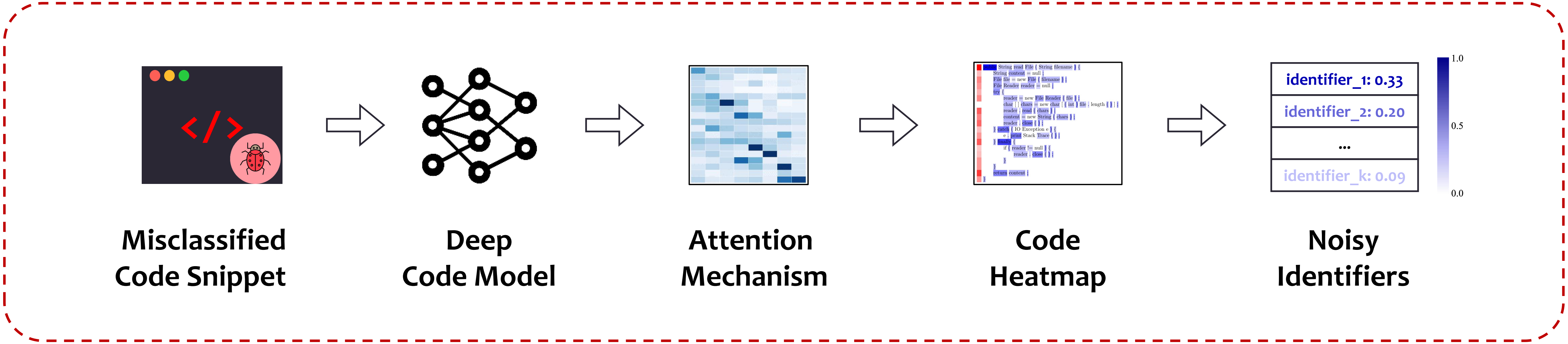


Noise Localization

- **C2** - How to **localize noise** resulting in misprediction from a given code snippet?

1. The attention mechanism is widely used to analyze the contribution of each element in the code(in particular, it is the core of the state-of-the-art Transformer architecture).

2. **Insight:** for mispredicted inputs, the identifiers with larger contributions to the misprediction are more likely to be noise in the code snippet.

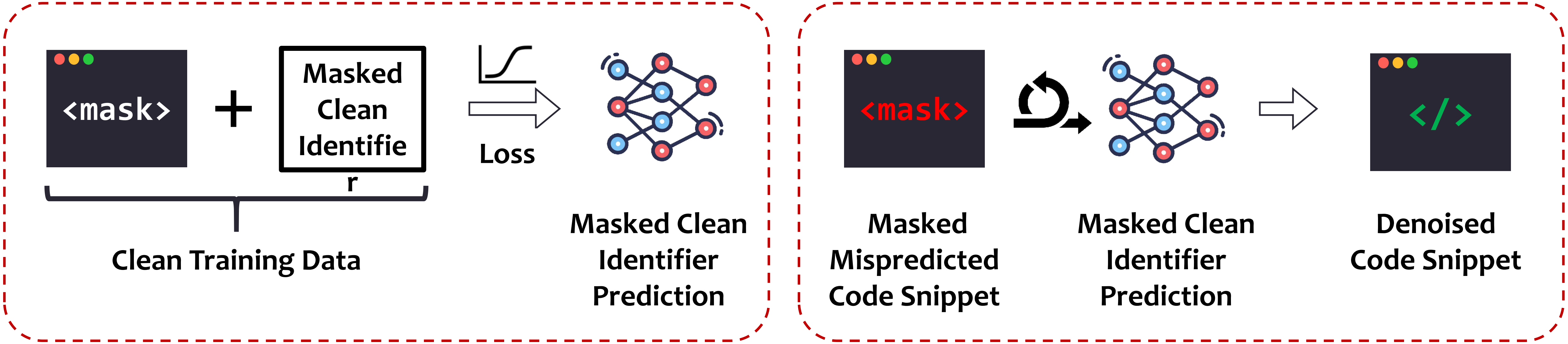


Noise Cleansing

- C3 - How to **cleanse noise** to make the code snippet be predicted correctly?

1. Exiting masked identifier prediction (MIP) models aim to predict the tokens at the masked locations, but they only consider the naturalness but not cleanliness.

2. To predict a clean identifier to replace the noisy identifier, CodeDenoise builds a masked clean identifier prediction (MCIP) model based on clean training data.

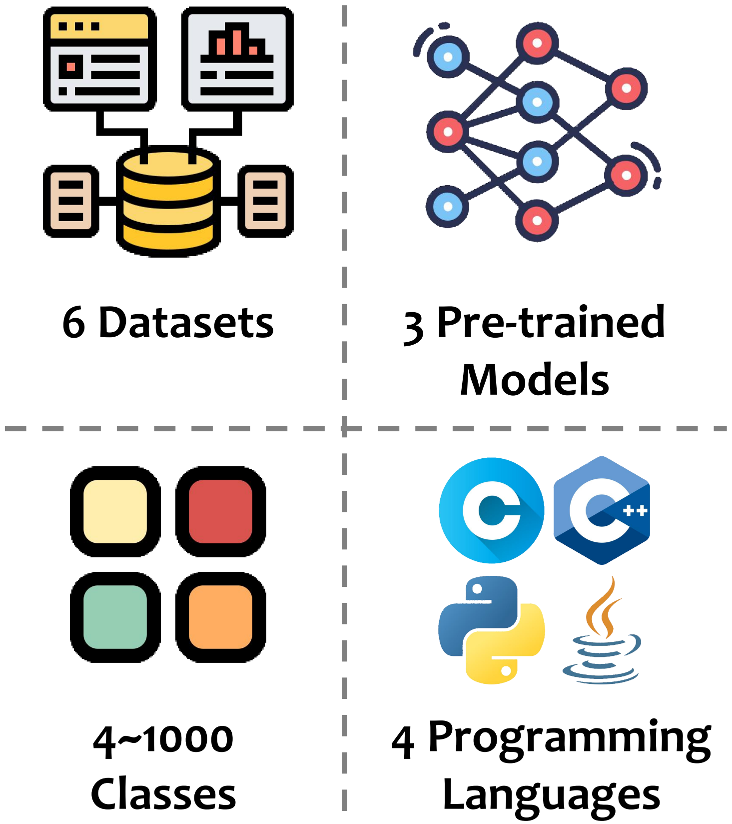


Training Phase

Inference Phase

Subjects

Task	Train/Validate/Test	Class	Language	Model	Acc.
Authorship Attribution	528/–/132	66	Python	CodeBERT GraphCodeBERT CodeT5	83.58% 77.27% 83.33%
Defect Prediction	27,058/–/6,764	4	C/C++	CodeBERT GraphCodeBERT CodeT5	85.47% 83.90% 82.29%
Functionality Classification C104	41,581/–/10,395	104	C	CodeBERT GraphCodeBERT CodeT5	97.87% 98.61% 98.60%
Functionality Classification C++1000	320,000/80,000/100,000	1000	C++	CodeBERT GraphCodeBERT CodeT5	85.00% 81.62% 86.49%
Functionality Classification Python800	153,600/38,400/48,000	800	Python	CodeBERT GraphCodeBERT CodeT5	93.91% 97.39% 97.62%
Functionality Classification Java250	48,000/11,909/15,000	250	Java	CodeBERT GraphCodeBERT CodeT5	96.30% 97.79% 97.48%



The subjects are diverse, involving different tasks/models/classes/languages.

RQ1: Effectiveness and Efficiency of CodeDenoise

Metric:

Correction Success Rate ↑
Mis-Correction Rate ↓

Task	CodeBERT		GraphCodeBERT		CodeT5	
	Fine-tuning	CodeDenoise	Fine-tuning	CodeDenoise	Fine-tuning	CodeDenoise
Authorship Attribution	20.00%/1.79%	**30.00%/0.00%**	10.00%/0.00%	**20.00%/0.00%**	10.00%/1.79%	**40.00%/0.00%**
Defect Prediction	5.98%/0.59%	**22.47%/0.24%**	8.51%/1.44%	**28.73%/0.18%**	5.15%/0.29%	**16.64%/0.18%**
Functionality Classification C104	7.32%/0.08%	**17.07%/0.02%**	5.88%/0.06%	**14.12%/0.04%**	13.41%/0.06%	**15.85%/0.04%**
Functionality Classification C++1000	1.42%/0.17%	**27.32%/0.05%**	1.95%/0.34%	**5.14%/0.05%**	1.14%/0.15%	**14.13%/0.04%**
Functionality Classification Python800	4.19%/0.09%	**28.76%/0.03%**	7.18%/0.08%	**20.48%/0.05%**	3.35%/0.06%	**19.55%/0.02%**
Functionality Classification Java250	23.00%/0.07%	**31.71%/0.04%**	16.67%/0.26%	**23.21%/0.25%**	26.83%/0.03%	**27.80%/0.03%**
Average	10.32%/0.46%	**26.22%/0.06%**	8.37%/0.36%	**18.61%/0.09%**	9.98%/0.39%	**22.33%/0.05%**

CodeDenoise outperforms Fine-tuning with larger correction success rate and smaller mis-correction rate.

Metric:

Overall Accuracy ↑

Task	CodeBERT			GraphCodeBERT			CodeT5		
	Ori	Fine-tuning	CodeDenoise	Ori	Fine-tuning	CodeDenoise	Ori	Fine-tuning	CodeDenoise
Authorship Attribution	84.85%	86.36%	**89.39%**	84.85%	86.36%	**87.88%**	84.85%	84.85%	**90.91%**
Defect Prediction	85.66%	86.01%	**88.68%**	84.36%	84.48%	**88.70%**	82.76%	83.41%	**85.48%**
Functionality Classification C104	97.63%	97.73%	**98.02%**	98.36%	98.40%	**98.56%**	98.42%	98.58%	**98.63%**
Functionality Classification C++1000	84.93%	85.00%	**89.00%**	81.68%	81.77%	**82.59%**	86.50%	86.52%	**88.37%**
Functionality Classification Python800	97.12%	97.15%	**97.92%**	98.43%	98.46%	**98.71%**	97.76%	97.78%	**98.18%**
Functionality Classification Java250	96.17%	96.99%	**97.35%**	97.76%	97.88%	**98.04%**	97.27%	97.97%	**98.00%**
Average	91.06%	91.54%	**93.39%**	90.91%	91.23%	**92.42%**	91.26%	91.52%	**93.26%**

CodeDenoise outperforms Fine-tuning in terms of ovelall accuracy.

RQ2: Contribution of Each Main Component

We constructed four variants of CodeDenoise:

- **CodeDenoise** $_{\text{deepgini}}$**:** Randomized-smoothing-based strategy → DeepGini-based strategy
- **CodeDenoise** $_{\text{randR}}$**:** Attention-based strategy → Random strategy
- **CodeDenoise** $_{\text{randC}}$**:** MCIP-based strategy → Random strategy
- **CodeDenoise** $_{\text{MIP}}$**:** MCIP-based strategy → MIP-based strategy

Metrics	CodeDenoise $_{\text{deepgini}}$	CodeDenoise $_{\text{randL}}$	CodeDenoise $_{\text{randC}}$	CodeDenoise $_{\text{MIP}}$	CodeDenoise
Correction Success Rate ↑	16.91%	14.65%	10.84%	15.50%	**21.91%**
Mis-correction Rate ↓	0.52%	0.41%	0.52%	0.34%	**0.09%**
#Identifier Changes ↓	2.25	3.79	3.27	2.27	**1.58**

All the three components make contributions to the overall effectiveness of CodeDenoise.

RQ3: Influence of Hyper-parameters

We studied the influence of two hyper-parameters in CodeDenoise:

- **θ:** the threshold to limit the perturbation degree
- **N:** the number of perturbed code snippets

θ	1	2	3	4	5
Correction Success Rate ↑	21.91%	22.85%	23.95%	25.27%	26.08%
Mis-correction Rate ↓	0.09%	0.14%	0.16%	0.20%	0.29%
Time (s) ↓	0.48	0.63	1.03	1.43	1.70

N	×1	×2	×3	×4	×5
Correction Success Rate ↑	21.91%	23.30%	24.66%	25.25%	25.99%
Mis-correction Rate ↓	0.09%	0.09%	0.08%	0.08%	0.08%
Time (s) ↓	0.48	0.71	0.87	1.13	1.63

We obtained default settings by balancing effectiveness and efficiency for practical use.

THANKS