Mitigating the Effect of Class Imbalance in Fault Localization Using Context-aware Generative Adversarial Network

Yan Lei[1,2*] , Tiantian Wen[1], Huan Xie[1], Lingfeng Fu[1], Chunyan Liu[1], Lei Xu[3], Hongxia Sun[4]
[1]School of Big Data & Software Engineering Chongqing University, Chongqing, China
[2]Peng Cheng Laboratory, ShenZhen, China
[3]Haier Smart Home Co., Ltd., Qingdao, China
[4]Qingdao Haidacheng Purchasing Service Co., Ltd., Qingdao, China
{yanlei, tiantianwen, huanxie, lingfengfu, chunyanliu}@cqu.edu.cn, {xulei1, sunhongxia}@haier.com

***Abstract*—Fault localization (FL) analyzes the execution information of a test suite to pinpoint the root cause of a failure. The class imbalance of a test suite, *i.e.,* the imbalanced class proportion between passing test cases (*i.e.,* majority class) and failing ones (*i.e.,* minority class), adversely affects FL effectiveness.**

To mitigate the effect of class imbalance in FL, we propose CGAN4FL: a data augmentation approach using Context-aware Generative Adversarial Network for Fault Localization. Specifically, CGAN4FL uses program dependencies to construct a failure-inducing context showing how a failure is caused. Then, CGAN4FL leverages a generative adversarial network to analyze the failure-inducing context and synthesize the minority class of test cases (*i.e.,* failing test cases). Finally, CGAN4FL augments the synthesized data into original test cases to acquire a class-balanced dataset for FL. Our experiments show that CGAN4FL significantly improves FL effectiveness, *e.g.,* promoting MLP-FL by 200.00%, 25.49%, and 17.81% under the Top-1, Top-5, and Top-10 respectively.

***Index Terms*—fault localization, class imbalance, program dependencies, generative adversarial network**

I. Introduction

To reduce debugging cost [1], [2], it is essential to develop effective approaches in the software debugging process. In the literature, various fault localization (FL) approaches (*e.g.,* [3]–[11]) have been proposed to pinpoint potential locations of faulty code over the past several decades.

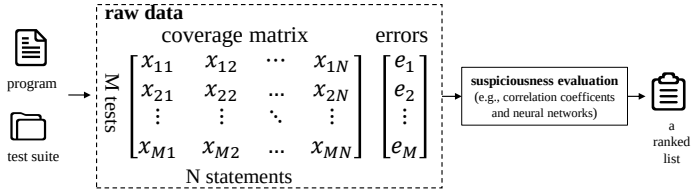

Fig. 1. Typical workflow of FL.

Fig. 1 shows the typical workflow of FL, *e.g.,* spectrum-based fault localization (SFL) [7], [12] and deep learning-based fault localization (DLFL) [11], [13], [14]. FL executes the test cases of a test suite and collects the coverage information (denoted as coverage matrix) and test results (represented as errors) of each test case. The coverage matrix and errors are raw data for FL. In the raw data, each row of the coverage matrix represents the coverage information of a test case, and each column corresponds to the coverage information of a statement in all test cases of a test suite. Specifically, for an element x_{ij} in the coverage matrix, $x_{ij} = 1$ means that the *i*-th test case executes the *j*-th statement and otherwise $x_{ij} = 0$; for an element e_i in the errors, $e_i = 1$ denotes that the *i*-th test case is a failed test case and otherwise $e_i = 0$. After the raw data have been acquired, many FL approaches use them directly as input, and develop different suspiciousness evaluation algorithms (*e.g.,* SFL using correlation coefficients and DLFL using neural networks) to evaluate the suspiciousness value for each statement. Finally, FL outputs a ranked list of program statements in descending order of suspiciousness values for manual or automated debugging [15]–[23].

Thus the raw data is indispensable for conducting effective FL. There are two classes of test cases: passing test cases and failing ones. In practice, failing test cases are much less than passing test cases. It leads to a class imbalance problem in the raw data, *i.e.,* the data with failing labels are much less than the data with passing labels. The existing studies [24]–[26] have shown that the class imbalance problem inevitably introduces a bias [27]–[29] into the suspiciousness evaluation of FL and adversely affects FL effectiveness.

Since failing test cases are usually irregularly distributed and occupy a very small portion in the input domain, it is difficult to directly generate valid failing test cases in practice to address the imbalance problem. Inspired by the recent wide use of data augmentation approaches [30], [31], we can augment the raw data of FL by synthesizing new raw data with failing labels to acquire balanced raw data, *i.e.,* the class ratio between the raw data with failing labels and passing labels is balanced. In this way, a suspiciousness evaluation algorithm of FL can afterward utilize the class-balanced raw data to improve its accuracy.

Based on the above analysis, we seek a data augmentation

*Corresponding author.

solution to the raw data of FL for addressing the class imbalance problem in FL. Generative adversarial network (GAN) [30] is amongst the most popular data augmentation approach. However, a failure-inducing context is useful for FL to acquire a reduced searching scope, and the original GAN does not consider a failure-inducing context into its data augmentation process, causing the augmentation to be potentially inaccurate. It means that we should further incorporate a failure-inducing context into GAN to guide its data augmentation for FL.

Therefore, we propose CGAN4FL: a data augmentation approach using Context-aware Generative Adversarial Network for Fault Localization, to mitigate the effect of class imbalance in FL. CGAN4FL analyzes program dependencies via program slicing [32] to construct a failure-inducing context, showing how a subset of statements propagates among each other to cause a program failure. Then, CGAN4FL combines the failure-inducing context into a generative adversarial network to devise a context-aware generative adversarial network, which can synthesize the raw data of FL with failing labels. Finally, we add the new synthesized failing raw data into the original raw data to acquire class-balanced raw data, where the data with failing labels have the same number as the data with passing labels.

To evaluate our approach CGAN4FL, we design and conduct large-scale experiments on six representative subject programs. We apply CGAN4FL for six state-of-the-art FL approaches, and further compare CGAN4FL with two representative data optimization approaches. The experimental results show that our approach improves the effectiveness of the six state-of-the-art FL approaches, and outperforms the two representative data optimization approaches. Specifically, the experimental results indicate that our approach compared with the six state-of-the-art FL approaches improves the effectiveness of fault localization by 125.34%, 56.32%, and 59.71% respectively on Top-1, Top-5, and Top-10 metrics on average.

The main contributions of this paper can be summarized as follows:

- We propose a data augmentation approach CGAN4FL, which synthesizes failing raw data for mitigating the effect of class imbalance problem in FL.
- We present a context-aware generative adversarial network which integrates a failure-inducing context into the data augmentation process of a generative adversarial network to guide data synthesization for FL.
- We conduct large-scale experiments and compare our approach with six state-of-the-art FL approaches and two representative data augmentation approaches, showing that CGAN4FL significantly improves fault localization effectiveness.
- We open source the replication package online including all relevant code[1].

The remainder of this paper is organized as follows. Section II introduces background information. Section III presents our approach CGAN4FL. Section IV and Section V show the experimental results and discussion. Section VI discusses related work and Section VII draws the conclusion.

[1]https://anonymous.4open.science/r/CGAN4FL-B448

II. Background

A. Generative Adversarial Network

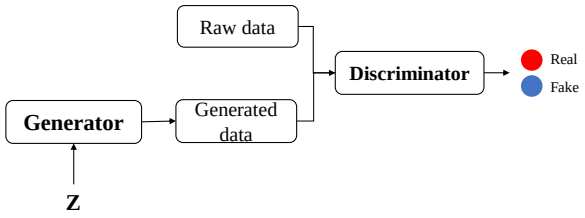

Fig. 2. Basic framework of a GAN.

Generative Adversarial Network (GAN) [30] is a deep learning framework that learns to generate adversarial data. Fig. 2 shows the basic framework of a GAN. GAN contains two components: the generator G and the discriminator D. The generator G is responsible for generating fake data that look like real data from the latent variable z while the discriminator D distinguishes whether the data belongs to raw data or generated data as accurately as possible. G and D are aggressive since they compete in order to accomplish their own objectives. The purpose of the model training is to minimize the loss of G and maximize the loss of D. Specifically when a generator has a lower loss, it means that the generated data is almost identical to real data; and when a discriminator obtains a higher loss, it means that it is hard to discriminate between real and generated data.

This adversarial learning situation can be formulated as Eq. (1) with parametrized networks G and D.

$$\min_G \max_D V(G,\, D) = \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z))] \tag{1}$$

In Eq. (1), $p_{data}\,(x)$ and $p_z\,(z)$ represent the real data probability distribution defined in data space $\mathcal{X}$ and the probability distribution of z defined in latent space $\mathcal{Z}$. $V\,(G,\, D)$ is a binary cross entropy function that is commonly used in binary classification problems [33]. It should be noted that G maps z from $\mathcal{Z}$ into the element of $\mathcal{X}$, while D takes an input x and determines whether x is real data or fake data generated by G.

Since the goal of D is to identify real or fake samples, $V\,(G,\, D)$ is a natural choice for this goal with its ability to solve binary classification problems. From the perspective of D, if the input data is real, the output of D should be close to maximum; if the input data comes from G, D will minimize its output. Thus, the $\log\left(1 - D\left(G\left(z\right)\right)\right)$ term is added to Eq. (1). At the same time, G plans to cheat D and thus it tries to maximize D's output when the input data is generated by it. Consequently, D tries to maximize $V\,(G,\, D)$ while G tries to minimize $V\,(G,\, D)$, forming a type of adversarial relationship.

The existing studies [24]–[26] have shown that the class imbalance problem of raw data adversely affects FL effectiveness, and it is crucial to address the class imbalance problem in FL. One of the best merits of GAN is that they generate data that is similar to real data. Due to this merit, they have many different applications in the real world, *e.g.,* generating images, text, audio, and video that are indistinguishable from real data [31], [34]–[36]. Inspired by the merit of GAN, our study utilizes the ability of GAN to mitigate the effect of class imbalance in FL via synthesizing minority class data for acquiring a class-balanced dataset.

B. Fault Localization

Fault localization (FL) typically collects and abstracts the runtime information of a test suite as the raw data (*i.e.,* the coverage matrix and errors in Fig. 1); then takes the raw data as input to evaluate the suspiciousness value for each statement; finally outputs a ranked list of program elements in descending order of suspiciousness values. There are many granularity types of program elements, *e.g.,* statements, methods, and files. Our study adopts the most widely-used granularity type of program elements, *i.e.,* statements. This section will introduce two popular FL techniques (*i.e.,* spectrum-based fault localization and deep learning-based fault localization), and they all use the raw data in Fig. 1 as input for suspiciousness evaluation. Our experiments will also apply our approach CGAN4FL for these FL techniques to evaluate its effectiveness.

Spectrum-based Fault Localization (SFL). SFL [7], [12] has been intensively studied in the literature. The basic idea of SFL is that the suspiciousness of a statement should increase when it is executed more frequently by failing test cases; its suspiciousness should decrease when it is executed more frequently by passing test cases.

To implement the above idea, SFL uses the raw data (*i.e.,* coverage matrix and errors) to define the four variables for each statement. Let s_j be a statement in the program. Eq. (2) defines the four variables for s_j as follows:

$$\begin{aligned} a_{np}(s_j) &= |\{i|x_{ij} = 0 \wedge e_i = 0\}| \\ a_{nf}(s_j) &= |\{i|x_{ij} = 0 \wedge e_i = 1\}| \\ a_{ep}(s_j) &= |\{i|x_{ij} = 1 \wedge e_i = 0\}| \\ a_{ef}(s_j) &= |\{i|x_{ij} = 1 \wedge e_i = 1\}| \end{aligned} \tag{2}$$

Where, $a_{np}(s_j)$ and $a_{nf}(s_j)$ represent the numbers of test cases that do *not* execute the statement s_j and return the *passing* and *failing* test results, respectively; $a_{ep}(s_j)$ and $a_{ef}(s_j)$ stand for the numbers of test cases that *execute* s_j, and return the *passing* and *failing* testing results, respectively.

Based on the four variables, SFL devises many suspiciousness evaluation formulas to evaluate the suspiciousness of a statement [7], [37]–[40]. The existing work [41] has empirically identified the three most effective SFL formulas (*i.e.,* Ochiai [12], DStar[2] [42], and Barinel [43]) in locating real faults. Since our study focuses on locating real faults, our experiments use the three SFL formulas. Based on the four variables defined in Eq. (2), Eq. (3), Eq. (4) and Eq. (5) show the definitions of the three SFL formulas to compute the suspiciousness of a statement s_j.

$$Ochiai(s_j) = \frac{a_{ef}(s_j)}{\sqrt{(a_{ef}(s_j) + a_{nf}(s_j)) \times (a_{ef}(s_j) + a_{ep}(s_j))}} \tag{3}$$

$$Dstar(s_j) = \frac{a_{ef}(s_j)^*}{a_{ep}(s_j) + a_{nf}(s_j)} \tag{4}$$

$$Barinel(s_j) = 1 - \frac{a_{ep}(s_j)}{a_{ep}(s_j) + a_{ef}(s_j)} \tag{5}$$

Deep Learning-based Fault Localization (DLFL). DLFL [11], [13], [14] has recently attracted much attention and acquired promising results. The basic idea of DLFL is that it utilizes the learning ability [44], [45] of neural networks to learn a FL model which reflects the relationship between a statement and a failure. Fig. 3 shows the typical architecture of DLFL. DLFL usually has three parts: input layer, deep learning component with several hidden layers, and output layer. Next, we will introduce three representative DLFL approaches, *i.e.,* MLP-FL [14], CNN-FL [11] and RNN-FL [13], and our experiments apply our approach CGAN4FL to the three DLFL approaches to evaluate its effectiveness.

In the input layer, DLFL takes the raw data (*i.e.,* coverage matrix and errors) in Fig. 1 as input. Specifically, *k* rows of the coverage matrix and its corresponding errors vector, *i.e.,* the coverage information of *k* test cases and their corresponding test results, are used as input. As shown in Fig. 3, these *k* test cases are the rows starting from the *i*-th row, where $i \in \{1, 1+k, 1+2k, \ldots, 1+(\lceil M/k \rceil - 1) \times k\}$. In the part of deep learning components, different fault localization methods use different neural networks. For example, MLP-FL [14] uses multi-layer perceptron, CNN-FL [11] adopts convolutional neural network, and RNN-FL [13] utilizes recurrent neural network. In the output layer, the model uses *sigmoid* function to make sure that the output results are between 0 and 1. Each element in the result of *sigmoid* function could differ from its corresponding element in the target vector. The parameters of the model are updated using the backpropagation algorithm with the intention of minimizing the difference between training result y and errors vector e (*i.e.,* the errors in Fig. 1). The network is trained iteratively. Finally, DLFL learns a trained model, which can reflect the relationship between a statement and a failure. With the trained model, DLFL can evaluate the suspiciousness value for statements.

III. Approach

This section will introduce our approach CGAN4FL: a data augmentation approach using Context-aware Generative Adversarial Network for Fault Localization. As shown in Fig. 4, CGAN4FL first uses program dependencies to construct a failure-inducing context; then combines the failure-inducing context into the GAN training to learn a context-aware GAN model; finally uses the trained context-aware GAN model to

[2]The '*' in Dstar formula is usually assigned to 2.

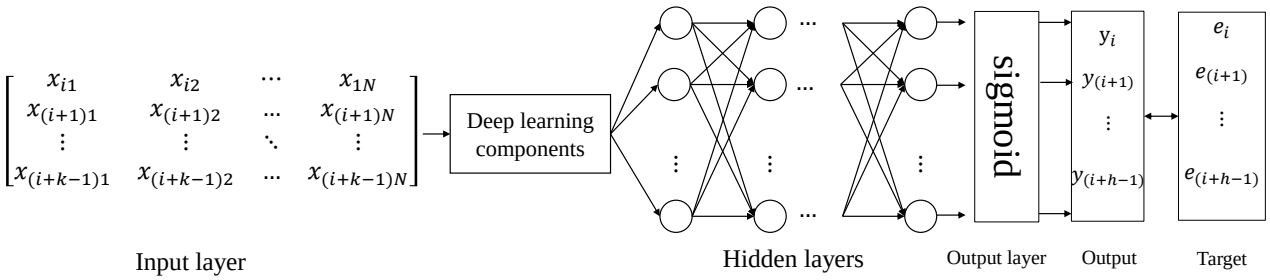

Fig. 3. Architecture of DLFL.

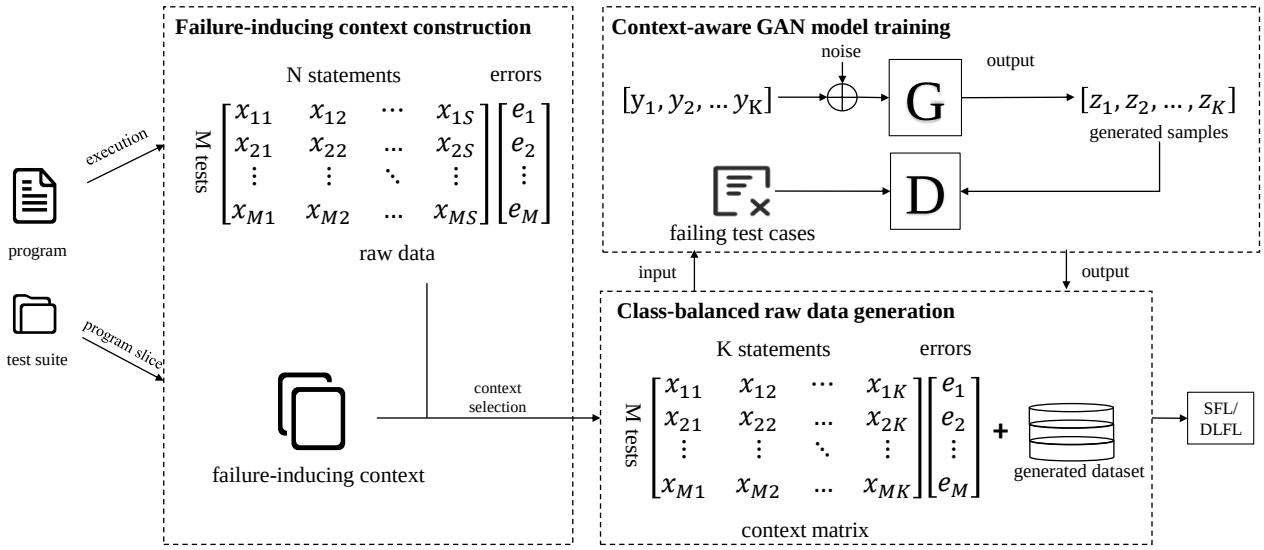

Fig. 4. Architecture of CGAN4FL.

generate new failing raw data until a class-balanced dataset is acquired, where the raw data with failing labels have the same number as the raw data with passing labels.

A. Failure-inducing Context Construction

A failure-inducing context shows how a subset of program elements (*e.g.,* statements) act on each other to cause a failure, and it is useful for FL to acquire a reduced searching scope. Therefore, we intend to integrate a failure-inducing context into the GAN training to guide its data augmentation. To implement the above idea, CGAN4FL adopts the widely-used program dependencies via program slicing [46] to construct a failure-inducing context, showing the dependencies between a subset of statements that cause a failure. The program slicing technique [46] extracts the program dependencies among statements to pick out a subset of statements whose execution leads to the incorrect output (*i.e.,* a failure). A few approaches have evaluated the effectiveness of dynamic slices in fault localization [47]–[49]. The subset of statements is a program slice, *i.e.,* a failure-inducing context in our approach CGAN4FL. Thus, we define a failure-inducing context as follows:

> **A failure-inducing context:** statements that directly or indirectly affect the computation of the faulty output value of a failure through chains of dynamic data and/or control dependencies.

To compute a failure-inducing context using program slicing, we use the following slicing criterion *contextSC*.

$$contextSC = (outStm, outVar, failTest) \quad (6)$$

In Eq. (6), $outStm$ is an output statement whose value of a variable (*i.e.,* $outVar$) is incorrect in the execution of a failing test case (*i.e.,* $failTest$). Dynamic slicing collects runtime information along the execution path of a test case, *i.e.,* the set of executed statements of a test case. It means that a test case with a smaller set of executed statements is usually easier for a dynamic slicing tool to perform efficient instrumentation and produce compressed traces for space optimization. Thus, for multiple failing test cases, the one with the least executed statements usually is beneficial for the efficiency of constructing a failure-inducing context. From the efficiency aspect, CGAN4FL will choose the failing test case having the least executed statements to construct a slicing criterion in Eq. (6).

Suppose that a failure-inducing context has K statements. It means that these K statements interact with each other to cause an incorrect output (*i.e.,* a program failure). Since the statements not in the failure-inducing context do not affect the incorrect output, we combine the failure-inducing context via keeping the coverage information of these statements in the failure-inducing context and removing others. Thus, CGAN4FL finally acquires a new $M \times K$ matrix called context matrix which records the execution information of the failure-inducing context in the test suite.

B. Context-aware GAN Model Training

After constructing a failure-inducing context, we acquire a $M \times K$ context matrix from the original $M \times N$ matrix (*i.e.,* original raw data). The context matrix shows the runtime information of these statements whose execution leads to the incorrect output of a program. CGAN4FL uses the context matrix as the input of the GAN model, *i.e.,* CGAN4FL trains the GAN to generate a new synthesized vector by the discriminating network D and the generating network G with all failing test cases selected from the context matrix as samples. It means that CGAN4FL will learn the features of all failing test cases, *i.e.,* the newly synthesized test cases will cover the common feature of all failing test cases. Thus, CGAN4FL will mark the newly synthesized test cases (*i.e.,* the newly synthesized vectors with the same structure of raw data) as failing labels. We add the new failing synthesized test cases to the context matrix and form a new matrix (*i.e.,* new raw data) whose failing data and passing data are balanced. Finally, the new raw data are used as the new input for the FL approach (*e.g.,* SFL and DLFL) to improve its effectiveness.

The model training part of Fig. 4 shows the specific CGAN4FL training procedure. CGAN4FL trains D (discriminator) first to initiate the training procedure of the GAN model. CGAN4FL selects the *K*-dimensional vector $[y_1, y_2, \ldots, y_K]$, and inputs it into G (generator) after noise processing to obtain the generated data $[z_1, z_2, \ldots, z_K]$ (*i.e.,* the synthesized data). Then, it further selects the original failing test cases in the original raw data (*i.e.,* $[x_{i1}, x_{i2}, \ldots, x_{iK}]$, $i \in \{1, 2, \ldots, M\}$ and $e_i = 1$) as the real data, and uses the generated data $[z_1, z_2, \ldots, z_K]$ as the generated sample to be spliced together and input them into D (discriminator). D gives label 1 to real data $[x_{i1}, x_{i2}, \ldots, x_{iK}]$, and 0 to generated data $[z_1, z_2, \ldots, z_K]$. The difference between D's output score and the label is trained using loss backpropagation. After D training is complete, CGAN4FL starts G training with the fixed parameters of D. In the G training process, D and G are regarded as a whole. $[y_1, y_2, \ldots, y_K]$ processed by the noise is used as the input, and then G outputs generated data $[z_1, z_2, \ldots, z_K]$. The discriminator D with fixed parameters is used for scoring. The difference between the output score and label 1 is used as the loss backpropagation to train G.

Throughout the GAN model training process, G is weak at the beginning, and D can easily distinguish between real data and generated data. With the gradual increase of training G, D cannot distinguish between real data and generated data. Eq. (1) is the training process of the minimax two-player game between generator G and discriminator D. D maximizes the objective function to identify whether the generated data $[z_1, z_2, \ldots, z_K]$ are fake. In contrast, G constantly minimizes the distribution difference between the real data and the generated data, *i.e.,* minimizing D's discrimination of generated data. Finally, a Nash equilibrium [50] is reached.

C. Class-balanced Raw Data Generation

After context-aware GAN training, CGAN4FL learns a context-aware GAN model which generates synthesized failing data (*i.e.,* the new synthesized failing vectors with the same structure as the original raw data) for FL. The trained model will generate synthesized failing data and add them to the original raw data until we acquire a class-balanced dataset, where the number of failing vectors and the number of passing vectors are the same in the new raw data. CGAN4FL inputs the new class-balanced raw data into the suspiciousness evaluation algorithm of the FL approach to mitigate the effect of the class imbalance problem in FL. Finally, with the new class-balanced raw data, FL outputs a ranked list of all statements in descending order of suspiciousness values.

D. An Illustrative Example

To illustrate how the methodology of CGAN4FL works, Fig. 5 shows an example of applying CGAN4FL. As shown in Fig. 5, there is a faulty program *P* with 16 statements including a fault at line 3, in which the number 0 should be 6 instead. We use one SFL approach (*i.e.,* GP02 [39]) to locate the faulty statement for our illustrative example. The cells below each statement indicate whether the statement is executed by the test case or not (*i.e.,* 0 for not executed and 1 for executed). The cells below the 'Result' which is the errors vector for the coverage matrix as shown in Fig. 1 represent whether the test result of a test case is failing or passing (*i.e.,* 1 for falling and 0 for passing). The original test suite is class-imbalanced since it has four passing test cases (*i.e.,* t_2, t_3, t_4, and t_5) and two failing test cases (*i.e.,* t_1 and t_6).

For acquiring a class-balanced dataset, we need to generate two pieces of new raw data with failing labels. CGAN4FL first uses the failing test case t_1 to compute the failure-inducing context using program slicing. According to Eq. (6), we set $(S_{14}, \mathrm{d1}, t_1)$ as the slicing criterion since the output value of the variable d1 in the output statement S_{14} is incorrect when executing the failing test case t_1. As shown in Fig. 5, the failure-inducing context regarding t_1 is $\{S_1, S_3, S_7, S_{14}\}$. Then, based on the failure-inducing context, CGAN4FL uses GAN to generate two pieces of synthesized failing raw data (*i.e.,* t_7 and t_8) marked with yellow in Fig. 5. Finally, CGAN4FL adds the two failing synthesized failing test cases into the context matrix to form a new raw data, and GP02 uses the new raw data to conduct the suspiciousness evaluation for each statement.

The bottom rows are the FL results of original GP02 and GP02 with CGAN4FL, *i.e.,* the two ranked lists of statements in descending order of suspiciousness in Fig. 5

Program P			Bugline information

S_1:Read(a,b,c)
S_2:d1=0,d2=0,d3=0;
S_3:if(b < 0){
S_4:d1 = b;
S_5:d2 = c;
S_6:d3 = a;}
S_7:else {d1 = b+1;
	S_8: d2 = c+1;
	S_9: if(a < 0){
	S_{10}:a = a+c;}
	S_{11}:else a = a+b;
	S_{12}:d3 = a+1;}
	S_{13}:if(c>0){
	S_{14}:output(d1);}
		S_{15}:else {output(d2);
		S_{16}:output(d3);}
		The failure-inducing context with t_1: $\{S_1, S_3, S_7, S_{14}\}$
			S_3 is faulty. Correct form: If(b<6){

test	a,b,c	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}	Result
t_1	-1,5,3	1	1	1	0	0	0	1	1	1	1	0	1	1	1	0	0	1
t_2	-2,-7,5	1	1	1	1	1	1	0	0	0	0	0	0	1	1	0	0	0
t_3	5,-6,-8	1	1	1	1	1	1	0	0	0	0	0	0	1	0	1	1	0
t_4	-5,8,-8	1	1	1	0	0	0	1	1	1	1	0	1	1	0	1	1	0
t_5	4,7,11	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0
t_6	4,2,1	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	1
t_7	-	0.91		0.92				0.63							0.35			1
t_8	-	0.99		0.99				0.59							0.38			1
GP02	suspiciouness	6	6	6	4.24	4.24	4.24	8.24	8.24	8.24	6.46	6.46	8.24	6	8.24	4.24	4.24	
	rank	10	11	**12**	14	15	16	1	2	3	5	6	4	13	7	8	9	
CGAN4FL (GP02)	suspiciouness	9.80		9.82				10.68							9.70			
	rank	3		**2**				1							4			

Fig. 5. An example illustrating CGAN4FL.

marked with different colors. Without using CGAN4FL, the ranked list of the statements using GP02 marked with blue is $\{S_7, S_8, S_9, S_{12}, S_{10}, S_{11}, S_{14}, S_{15}, S_{16}, S_1, S_2, S_3, S_{13}, S_4, S_5, S_6\}$. After applying our approach CGAN4FL, the ranked list of the statements using GP02 is $\{S_7, S_3, S_1, S_{14}\}$. We can observe that the faulty statement S_3 is ranked 12th place with the original raw data while CGAN4FL ranks the faulty statement S_3 2nd place. It means that CGAN4FL yields better FL results than the original GP02, mitigating the effect of the class imbalance in FL.

IV. Experiments

A. Datasets

To evaluate the effectiveness of our approach CGAN4FL, we adopt the Defects4J [51] that has been widely used in the software testing and debugging community [19], [52]–[54]. We use all the six representative subject programs of Defects4J[3] (*i.e.,* Chart, Closure, Math, Mockito, Lang, and Time) and all faults from these programs are real faults.

TABLE I summarizes the information of the six subject programs. For each program, it lists a brief functional description (column 'Description'), the number of faulty versions used (column 'Versions'), the number of thousand lines of statements (column 'LoC(K)'), the number of test cases (column 'Test'). Since it is time-consuming to collect the inputs (*i.e.,* raw data) of the six large programs of Defects4J, we reuse the coverage matrix and the errors collected by Pearson *et al.* [55].

[3]https://github.com/rjust/defects4j

B. Experiment Settings

For every faulty version of the six subject programs shown in TABLE I, we set the training period of context-aware GAN model to 1,000, and the dimension of hidden variable z defined in latent space $\mathcal{Z}$ to 100. Our experiments were conducted on a 64-bit Linux server with 40 cores of 2.4GHz CPU and 252GB RAM. The operating system is Ubuntu 20.04.

C. Evaluation Metrics

We adopt the four widely-used FL evaluation metrics to evaluate the effectiveness of our approach. Their definitions are as follows:

- Number of Top-K [56], [57]: It is the number of faulty versions with at least one faulty statement that is within the first K position of the rank list produced by the FL technique. In the previous study, many respondents view fault localization as successful only if it can localize bugs in the top 10 positions from a practical perspective [56], [57]. Following the prior work [56], [57], we assign K with the value of 1, 5, and 10 for our evaluation. A higher value of Top-K means better FL effectiveness.
- Mean Average Rank (MAR) [5]: For a faulty version, the average rank is the mean rank of all faulty statements in the rank list. MAR is the mean average value for the project that includes several faulty versions. A lower value of MAR indicates better FL effectiveness.
- Mean First Rank (MFR) [5]: It first computes the rank that any of the statements are located first for a faulty version. Then compute the mean value of the ranks for

TABLE I
SUBJECT PROGRAMS

Program	Description	Versions	LoC(K)	Test
Chart	Java chart library	26	96	2205
Lang	Apache commons-lang	65	22	2245
Math	Apache commons-math	106	85	3602
Closure	Closure compiler	133	90	7927
Time	Standard date and time library	27	28	4130
Mokito	Mocking framework for Java	38	67	1075
Total	-	395	388	21184

TABLE II
THE RESULTS OF TOP-1, TOP-5 , TOP-10, MAR AND MFR OF FL BASELINES AND CGAN4FL.

Metrics	Scenario	Ochiai	Dstar	Barinel	MLP-FL	CNN-FL	RNN-FL
Top-1	baseline	38	38	35	9	6	27
	CGAN4FL	**50**	**41**	**42**	**27**	**34**	**34**
Top-5	baseline	113	111	110	51	23	72
	CGAN4FL	**123**	**121**	**119**	**64**	**86**	**81**
Top-10	baseline	156	152	154	73	29	98
	CGAN4FL	**177**	**160**	**167**	**86**	**116**	**111**
MFR	baseline	371.79	363.80	373.81	987.94	1514.86	602.75
	CGAN4FL	**227.70**	**261.58**	**211.70**	**259.24**	**479.01**	**296.04**
MAR	baseline	676.08	704.25	680.49	1319.10	1823.73	948.71
	CGAN4FL	**307.82**	**338.76**	**283.07**	**369.70**	**549.95**	**333.76**

the project. A lower value of MFR shows better FL effectiveness.

- Relative Improvement (RImp) [11]: This metric can see the improvement of one fault localization approach relative to another fault localization approach. It is to compare the total number of statements that need to be examined to find the first faulty statement using CGAN4FL versus the number that needs to be examined by using baselines. A lower value of RImp indicates better FL effectiveness.

D. Research Questions and Results

To evaluate the effectiveness of our approach, we design and conduct the experiments to investigate the following research questions:

- **RQ1:** How does CGAN4FL perform in localizing real faults compared with original state-of-the-art FL approaches? This RQ aims at investigating whether CGAN4FL improves FL effectiveness after applying our approach. If the effectiveness of the FL approach increases after applying CGAN4FL, it means that CGAN4FL can mitigate the effect of class imbalance in FL.
- **RQ2:** How effective is CGAN4FL as compared with the representative data optimization approaches? This RQ is to further verify the ability of CGAN4FL to mitigate the effect of class imbalance in FL via comparing other representative data optimization approaches. If CGAN4FL outperforms other representative data optimization approaches, it means that CGAN4FL is more effective than other representative approaches in addressing the class imbalance problem of FL.
- **RQ3:** Does each component contributes to the effectiveness of CGAN4FL? This RQ is to check whether each component of CGAN4FL (*i.e.,* a GAN or a failure-inducing context) contributes to the effectiveness of CGAN4FL. We use three cases: original FL (denoted as baseline), CGAN4FL only using GAN (denoted as CGAN4FL(GAN)), CGAN4FL using GAN and failure-inducing context (denoted as CGAN4FL(GAN+context)). If we acquire a FL effectiveness relationship: CGAN4FL(GAN+context) > CGAN4FL(GAN) > baseline, it means that the GAN (due to CGAN4FL(GAN) > baseline) and the failure-inducing context (due to CGAN4FL(GAN+context) > CGAN4FL(GAN)) both contribute to CGAN4FL.

RQ1. How does CGAN4FL perform in localizing real faults compared with original state-of-the-art FL approaches?

There are two main types of FL: spectrum-based fault localization (SFL) and deep learning-based fault localization (DLFL). Recent studies [13], [41] have shown the most effective SFL approaches (*i.e.,* **Dstar** [10], **Ochiai** [2], and **Barinel** [43]) and DLFL approaches (*i.e.,* **MLP-FL** [14], **CNN-FL** [11], and **RNN-FL** [13]) in locating real faults. Thus, we use the six state-of-the-art FL approaches as the baselines and apply CGAN4FL to them to compare their effectiveness. For details of these FL approaches, please refer to Section II-B.

TABLE II shows the Top-K, MAR, and MFR results of the comparisons of the FL baselines and our approach CGAN4FL. It illustrates two scenarios: a baseline without using CGAN4FL (referred to as baseline) and using CGAN4FL (referred to as CGAN4FL). For the convenience of reading, we bold the experimental results in the tables, indicating which approach performs better.

As shown in TABLE II, CGAN4FL significantly outperforms all the baselines. For SFL approaches, take Ochiai as an example. The number of faults that CGAN4FL can locate is 50, 123, and 177 for the Top-1, Top-5, and Top-10 metrics, respectively. The results denote the Top-1, Top-5, and Top-10 metrics have increased by 31.58%, 8.85%, and 13.46% as compared with Ochiai. For DLFL approaches, take MLP-FL as an example. The number of faults that CGAN4FL can locate is 27, 64, and 86 for the Top-1, Top-5, and Top-10 metrics, respectively, *i.e.,* the Top-1, Top-5, and Top-10 metrics have increased by 200.00%, 25.49%, 17.81% as compared with the MLP-FL. Furthermore, the MFR and MAR metrics show that the rank of CGAN4FL is lower than that of baselines for all six FL techniques. The results show that CGAN4FL can always locate one buggy line first and find all buggy lines with the least effort.

Fig. 6 visually shows the MFR distribution of the FL baselines and CGAN4FL. The results show that CGAN4FL significantly improves FL effectiveness.

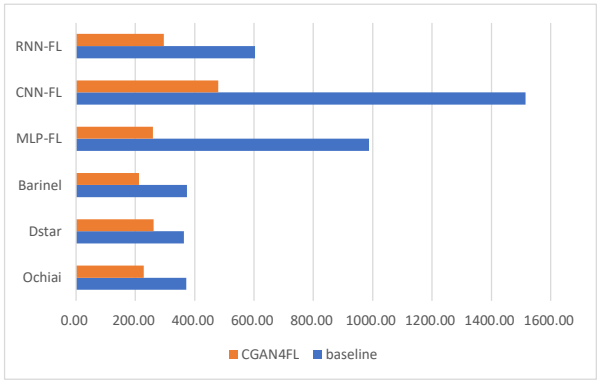

Fig. 6. MFR distribution of FL baselines and CGAN4FL

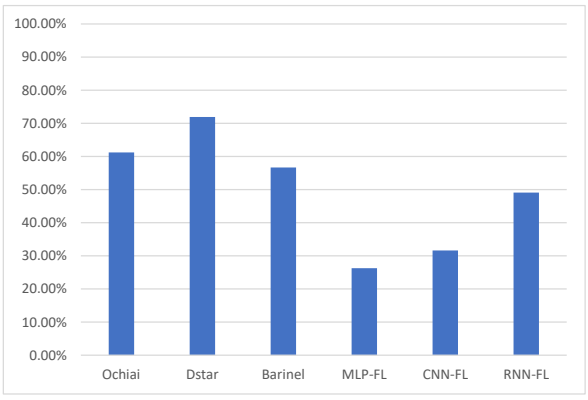

Fig. 7. RImp distribution of CGAN4FL vs FL baselines.

Fig. 7 shows the RImp distribution of our approach CGAN4FL vs the six FL baselines. As shown in Fig. 7, all RImp values are less than 100%, showing that our approach improves all the baselines after applying CGAN4FL. Take the MLP-FL as an example. MLP-FL will examine 987.94 lines on average to locate the first bug in all faulty versions (*i.e.,* MFR), while CGAN4FL only checks 259.24 lines of code. Thus, the value of RImp is 26.24%, indicating that for locating the first faulty statement, the number of statements to be checked by CGAN4FL is 26.24% of the original MLP-FL.

Summary for RQ1*: In RQ1, we discuss the effectiveness of the six state-of-the-art FL approaches using CGAN4FL and without using CGAN4FL. The experimental results show that CGAN4FL is effective to improve FL effectiveness by mitigating the effect of class imbalance in FL.*

RQ2. How effective is CGAN4FL as compared with the representative data optimization approaches?

To further evaluate the ability of CGAN4FL to mitigate the effect of class imbalance in FL, we compare our approach with two representative data optimization approaches, *i.e.,* resampling [25], [26], [58] and undersampling [59]. Resampling and undersampling acquire a class-balanced dataset by replicating minority samples and removing the majority samples, respectively. For more details, resampling and undersampling can refer to Gao *et al.* [58] and Wang *et al.* [59], respectively.

TABLE III shows the Top-K, MAR, and MFR results of the two representative data optimization approaches and our approach CGAN4FL. As shown in TABLE III, CGAN4FL outperforms resampling and undersampling in all cases of SFL and most cases of DLFL. Taking Ochiai as an example, the number of faults CGAN4FL can locate is 50, 123, and 177 for Top-1, Top-5, and Top-10 metrics, respectively. The results indicate that the Top-1, Top-5, and Top-10 metrics have increased by 284.62%, 136.54%, and 126.92% respectively as compared with undersampling, and increased by 47.06%,33.70%, and 40.48% respectively as compared with resampling. Furthermore, the MFR and the MAR of CGAN4FL are lower than the two representative data optimization approaches in almost all cases, showing that CGAN4FL performs better than the two representative approaches.

Furthermore, Fig. 8 visually shows the MFR distribution of resampling, undersampling, and CGAN4FL, indicating that CGAN4FL is the best of these three scenarios except for one case of resampling is better in CNN-FL. Fig. 9 shows the RImp distribution under two scenarios: CGAN4FL vs resampling and CGAN4FL vs undersampling. As shown in Fig. 9, the RImp values of all the cases are less than 100% except for one case of CGAN4FL vs resampling in CNN-FL. Thus, we can conclude that CGAN4FL performs better than resampling and undersampling in addressing the class imbalance problem in FL.

Summary for RQ2*: In RQ2, we compare CGAN4FL with two representative data optimization approaches,* i.e., *resampling and undersampling. The experimental results show that CGAN4FL is more effective than the two representative data optimization approaches in almost all cases for mitigating the effect of class imbalance in FL.*

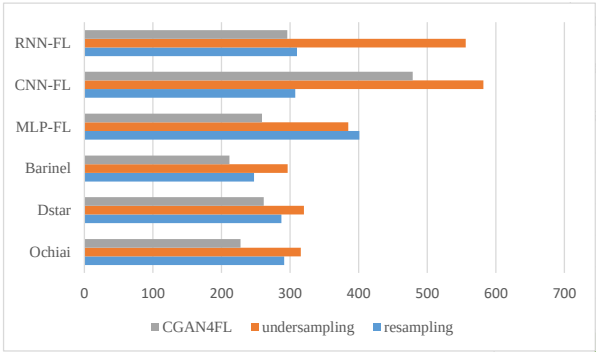

Fig. 8. MFR distribution of resampling, undersampling and CGAN4FL

RQ3. Does each component contributes to the effectiveness of CGAN4FL?

TABLE III
THE RESULTS OF TOP-1, TOP-5, TOP-10, MAR AND MFR OF DATA OPTIMIZATION APPROACHES AND CGAN4FL.

Metrics	Scenario	Ochiai	Dstar	Barinel	MLP-FL	CNN-FL	RNN-FL
Top-1	resampling	34	34	33	8	**34**	**34**
	undersampling	13	13	13	15	9	14
	CGAN4FL	**50**	**41**	**42**	**27**	**34**	**34**
Top-5	resampling	92	92	97	53	**92**	**92**
	undersampling	52	52	51	50	37	51
	CGAN4FL	**123**	**121**	**119**	**64**	86	81
Top-10	resampling	126	126	139	73	**125**	**128**
	undersampling	78	78	79	69	58	80
	CGAN4FL	**177**	**160**	**167**	**86**	116	111
MFR	resampling	291.58	287.53	247.59	400.81	**307.36**	310.00
	undersampling	315.66	320.02	296.23	384.75	582.04	555.91
	CGAN4FL	**227.70**	**261.58**	**211.70**	**259.24**	479.01	**296.04**
MAR	resampling	625.38	668.06	575.41	736.89	660.98	685.14
	undersampling	711.35	732.99	680.32	774.98	1014.07	971.23
	CGAN4FL	**307.82**	**338.76**	**283.07**	**369.70**	**549.95**	**333.76**

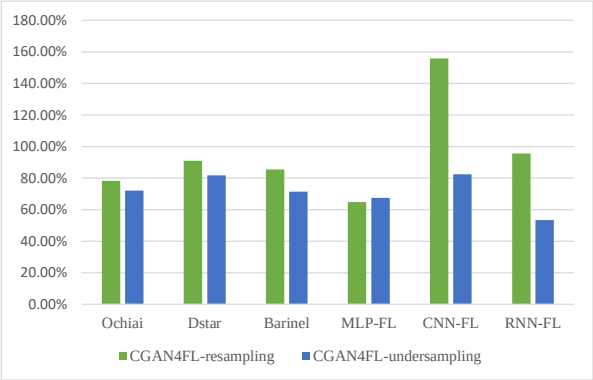

Fig. 9. RImp distribution of CGAN4FL vs resampling and CGAN4FL vs undersampling.

To check whether each component contributes to the effectiveness of CGAN4FL, we use Wilcoxon-Signed-Rank Test (WSR) [60] to verify whether the effectiveness relationship (*i.e.,* CGAN4FL(GAN+context) > CGAN4FL(GAN) > baseline) is satisfied or not, where CGAN4FL(GAN+context), CGAN4FL(GAN), and baseline denote CGAN4FL using GAN and failure-inducing context, CGAN4FL only using GAN, and original FL respectively. For each FL approach, we perform two paired Wilcoxon-Signed-Rank tests (*i.e.,* CGAN4FL(GAN+context) vs CGAN4FL(GAN) and CGAN4FL(GAN) vs baseline) by using the ranks of the faulty statements as the pairs of measurements.

TABLE IV shows the statistical results of all the tests at the σ level of 0.05. The 'conclusion' column gives the conclusion according to p-value. Take Ochiai as an example. For CGAN4FL(GAN) vs baseline, the the p-value of greater, less, and two-sided are 1, 2.68e-08, and 5.36e-08 respectively. According to the definition of WSR, it means that the MFR value of CGAN4FL(GAN) (*i.e.,* Ochiai using CGAN4FL(GAN)) is less than that of the baseline (*i.e.,* original Ochiai), leading to a BETTER result.

TABLE IV
STATISTICAL COMPARISON OF CGAN4FL(GAN) VS BASELINE AND CGAN4FL(GAN+CONTEXT) VS CGAN4FL(GAN).

method	comparison	greater	less	two-sided	conclusion
Ochiai	CGAN4FL(GAN) vs baseline	1	2.68e-08	5.36e-08	**BETTER**
	CGAN4FL(GAN+context) vs CGAN4FL(GAN)	1	1.43e-03	2.86e-03	**BETTER**
Dstar	CGAN4FL(GAN) vs baseline	1	6.07e-06	1.21e-05	**BETTER**
	CGAN4FL(GAN+context) vs CGAN4FL(GAN)	1	4.14e-10	8.28e-10	**BETTER**
Barinel	CGAN4FL(GAN) vs baseline	1	8.05e-04	1.61e-03	**BETTER**
	CGAN4FL(GAN+context) vs CGAN4FL(GAN)	1	3.57e-03	7.13e-03	**BETTER**
MLP-FL	CGAN4FL(GAN) vs baseline	1	1.42e-02	2.84e-02	**BETTER**
	CGAN4FL(GAN+context) vs CGAN4FL(GAN)	1	1.84e-10	3.67e-10	**BETTER**
CNN-FL	CGAN4FL(GAN) vs baseline	1	1.70e-07	3.39e-07	**BETTER**
	CGAN4FL(GAN+context) vs CGAN4FL(GAN)	1	1.35e-05	2.71e-05	**BETTER**
RNN-FL	CGAN4FL(GAN) vs baseline	1	9.87e-04	1.97e-03	**BETTER**
	CGAN4FL(GAN+context) vs CGAN4FL(GAN)	1	3.07e-13	6.14e-13	**BETTER**

For CGAN4FL(GAN+context) vs CGAN4FL(GAN), the p-value of greater, less, and two-sided are 1, 1.43e-03, and 2.86e-03 respectively. It means that the MFR value of CGAN4FL(GAN+context) (*i.e.,* Ochiai using our approach CGAN4FL) is less than that of CGAN4FL(GAN) (*i.e.,* Ochiai using CGAN4FL(GAN)), also leading to a BETTER result. From the table, we can observe that both CGAN4FL(GAN) vs baseline and CGAN4FL(GAN+context) vs CGAN4FL (GAN) obtain BETTER results in all cases, meaning that the effectiveness relationship CGAN4FL(GAN+context) > CGAN4FL(GAN) > baseline is satisfied. Thus, each component of CGAN4FL contributes to its effectiveness.

Summary for RQ3*: In RQ3, we make statistical comparisons in two scenarios: CGAN4FL(GAN+context) vs CGAN4FL(GAN) and CGAN4FL(GAN) vs baseline. The experimental results show that the effectiveness relationship CGAN4FL(GAN+context) > CGAN4FL(GAN) > baseline holds. Thus, we can conclude that each component of CGAN4FL* (i.e., *failure-inducing context and GAN) contributes to its effectiveness.*

V. DISCUSSION

A. Threats to Validity

The implementation of baselines and our approach. Our implementation of baselines and CGAN4FL may potentially contain bugs. For the three SFL approaches (*i.e.,* Dstar, Ochiai, and Barinel), we implement them according to their formulas and then manually test their correctness. For the three DLFL approaches (*i.e.,* MLP-FL, RNN-FL, and CNN-FL), we acquire the source code of CNN-FL from the authors and implement the other two DLFL approaches via replacing the deep learning component with MLP and RNN from the source code of CNN-FL. Since a neural network has many parameters for its construction (*e.g.,* learning rate, batch size), some parameters of MLP-FL, RNN-FL, and CNN-FL may differ from the original paper. Besides the implementation of baselines, we implement our pipeline of failure-inducing context construction, context-aware GAN model training, and

class-balanced raw data generation, which may also potentially include bugs. To mitigate those threats, we check our code implementation rigorously and make all relevant code publicly available (see the footnote in Section I).

The generalizability. We conduct our experiments on the real faults dataset benchmark, Defects4J, which is widely used in fault localization and program repair community. Although the subject programs selected in our experiments are all from the real world and our approach performs well on these programs, it may be not effective for other programs since no dataset can cover all possible cases of faults in practice. Thus, it is worthwhile to conduct more experiments on more large-sized programs with real faults to further verify the effectiveness of our approach in mitigating the effect of class imbalance in FL.

B. Reasons for CGAN4FL Is Effective

Our experiments demonstrate that CGAN4FL is more effective than the compared baselines. The main reasons are threefold: (1) CGAN4FL takes full advantage of the program dependency to capture the statements closely related to the faulty statements. In other words, the utilization of program slicing removes the fault-irrelevant statements precisely. (2) The GAN is a powerful model that could generate synthesized samples that are like real samples. More importantly, we provide the GAN model with the expert knowledge in software debugging (*i.e.,* the fault-relevant statements obtained by program slicing), which could potentially improve the accuracy and efficiency of the model. (3) The advantages of both the program analysis technique (*i.e.,* the program slicing) and the advanced generative network (*i.e.,* the GAN model) are well combined by CGAN4FL to gain the high-quality generated input data. Further, the high-quality and class-balanced input data could be beneficial to the state-of-the-art SFL and DLFL approaches.

VI. Related work

A. Fault Localization

Spectrum-based fault localization (SFL) [7], [12] and Deep learning-based fault localization (DLFL) [11], [13], [61] are the two most popular FL approaches. Researchers have proposed many SFL techniques (*e.g.,* Tarantula [4], Ochiai [12], Jaccard [12], and DStar [42]), and enhanced program spectrum (*e.g.,* [62], [63]) and a test suite (*e.g.,* [5], [64]) for more improvement. With a large number of SFL techniques, many studies [7], [12], [42], [65] have explored the best SFL techniques. Yoo *et al.* [66] have found that there is no SFL technique claiming that it can outperform all others under every scenario. Even if the best SFL technique does not exist. The existing studies [39], [67] have found a group of optimal SFL techniques, *i.e.,* the group of optimal SFL techniques cannot outperform each other whereas they can outperform all the other SFL techniques outside the group. DLFL [11], [13], [61] uses deep learning to locate a fault and recently attracts much attention. Wang *et al.* propose the FL approach BPNN-FL [68] using BP neural network model as a pipeline for learning input and output relationships. Then, Wong *et al.* [64] improve their BPNN-FL approach by removing irrelevant statements. Similar to the idea of Wong *et al.*, many researchers directly use the raw data as training data, and propose different DLFL approaches using different neural networks (*e.g.,* MLP-FL [14], CNN-FL [11], and RNN-FL [13]). In contrast to devising an effective FL approach, our work focuses on addressing the class imbalance problem in FL and can be used in tandem with these FL approaches.

B. Class Imbalance

In recent decades, researchers have proposed many methods to address the class imbalance problem. The typical methods are data-level, algorithm-level, hybrid, and ensemble learning methods. The data-level methods add a preprocessing step to mitigate the effect of class imbalance in the learning process [69]–[71]. The algorithm-level methods create or modify deep learning algorithms for addressing the class imbalance problem [72], [73]. The hybrid methods combine algorithm-level and data-level methods [74]–[76]. The ensemble learning methods [77], [78] use ensembles to increase the accuracy of classification by training several different classifiers and combining their decisions to output a single class label. Specifically, with regard to ensemble learning methods, there are lots of different approaches, *e.g.,* SMOTEBoost [79], RUSBoost [80], IIVotes [81], EasyEnsemble [82], and SMOTEBagging [83]. These works focus on addressing the class imbalance problem in the artificial intelligence field. In contrast, our work focuses on mitigating the effect of class imbalance in a different research field (*i.e.,* fault localization).

VII. Conclusion

In this paper, we propose CGAN4FL: a data augmentation approach that uses context-aware GAN to mitigate the effect of the class imbalance in FL. CGAN4FL embraces two main ideas: (1) data augmentation is a potential and effective solution to the class imbalance problem in FL; (2) a failure-inducing context is useful for guiding and acquiring a more precise data augmentation process. To implement the above ideas, we use program dependencies to construct a failure-inducing context showing how a failure is caused, and integrate the context into a generative adversarial network to learn the features of minority class and synthesize minority class data for generating a class-balanced dataset for FL. The experiments show that our approach CGAN4FL is effective to mitigate the effect of class imbalance in FL.

In the future, we intend to use more large-sized program to further verify our approach, and explore other generative networks for more improvement.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China (No. 62272072), the Fundamental Research Funds for the Central Universities (No. 2022CDJDX-005), and the Major Key Project of PCL (No. PCL2021A06).

REFERENCES

[1] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, p. 1, 2002.

[2] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[3] J. A. Jones, "Fault localization using visualization of test information," in *Proceedings. 26th International Conference on Software Engineering*. IEEE, 2004, pp. 54–56.

[4] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[5] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 169–180.

[6] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[7] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[8] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 273–283.

[9] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S.-C. Cheung, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2348–2368, 2019.

[10] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 2012, pp. 21–30.

[11] Z. Zhang, Y. Lei, X. Mao, and P. Li, "Cnn-fl: An effective approach for localizing faults using convolutional neural networks," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 445–455.

[12] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[13] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and X. Zhang, "A study of effectiveness of deep learning in locating real faults," *Information and Software Technology*, vol. 131, p. 106486, 2021.

[14] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," *Mathematical Problems in Engineering*, vol. 2016, 2016.

[15] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 267–278.

[16] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 165–176.

[17] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[18] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, "Automatic repair of real bugs: An experience report on the defects4j dataset," 2015.

[19] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[20] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[21] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[22] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 740–751.

[23] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[24] C. Gong, Z. Zheng, W. Li, and P. Hao, "Effects of class imbalance in test suites: an empirical study of spectrum-based fault localization," in *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. IEEE, 2012, pp. 470–475.

[25] L. Zhang, L. Yan, Z. Zhang, J. Zhang, W. Chan, and Z. Zheng, "A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization," *Journal of Systems and Software*, vol. 129, pp. 35–57, 2017.

[26] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, "Improving deep-learning-based fault localization with resampling," *Journal of Software: Evolution and Process*, vol. 33, no. 3, p. e2312, 2021.

[27] Y. Sun, A. K. Wong, and M. S. Kamel, "Classification of imbalanced data: A review," *International journal of pattern recognition and artificial intelligence*, vol. 23, no. 04, pp. 687–719, 2009.

[28] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on knowledge and data engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[29] N. V. Chawla, "Data mining for imbalanced datasets: An overview," *Data mining and knowledge discovery handbook*, pp. 875–886, 2009.

[30] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[31] A. Antoniou, A. Storkey, and H. Edwards, "Data augmentation generative adversarial networks," *arXiv preprint arXiv:1711.04340*, 2017.

[32] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPlan Notices*, vol. 25, no. 6, pp. 246–256, 1990.

[33] X. Mao, Q. Li, H. Xie, R. Y. Lau, Z. Wang, and S. Paul Smolley, "Least squares generative adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2794–2802.

[34] Y. Balaji, M. R. Min, B. Bai, R. Chellappa, and H. P. Graf, "Conditional gan with discriminative filter generation for text-to-video synthesis." in *IJCAI*, vol. 1, no. 2019, 2019, p. 2.

[35] Y. Zhang, Z. Gan, and L. Carin, "Generating text via adversarial training," in *NIPS workshop on Adversarial Training*, vol. 21, 2016, pp. 21–32.

[36] J.-H. Kim, S.-H. Lee, J.-H. Lee, and S.-W. Lee, "Fre-gan: Adversarial frequency-consistent audio synthesis," *arXiv preprint arXiv:2106.02297*, 2021.

[37] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei, and J. Sun, "A similarity-aware approach to testing based fault localization," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 291–294.

[38] D. Hao, L. Zhang, H. Zhong, H. Mei, and J. Sun, "Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 683–686.

[39] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, pp. 1–40, 2013.

[40] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 244–258.

[41] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 609–620.

[42] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[43] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Spectrum-based multiple fault localization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 88–99.

[44] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 3–14.

[45] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[46] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.

[47] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, 2005, pp. 33–42.

[48] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering*, vol. 12, no. 2, pp. 143–160, 2007.

[49] X. Li and A. Orso, "More accurate dynamic slicing for better supporting software debugging," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 28–38.

[50] L. J. Ratliff, S. A. Burden, and S. S. Sastry, "Characterization and computation of local nash equilibria in continuous games," in *2013 51st Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2013, pp. 917–924.

[51] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[52] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 177–188.

[53] H. Xie, Y. Lei, M. Yan, Y. Yu, X. Xia, and X. Mao, "A universal data augmentation approach for fault localization," in *International Conference on Software Engineering (ICSE'22)*, 2022.

[54] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, 2017, pp. 261–272.

[55] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating & improving fault localization techniques," *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03*, 2016.

[56] S. R. Heris and M. Keyvanpour, "Effectiveness of weighted neural network on accuracy of software fault localization," in *2019 5th International Conference on Web Research (ICWR)*. IEEE, 2019, pp. 100–104.

[57] Y. Küçük, T. A. Henderson, and A. Podgurski, "Improving fault localization by integrating value and predicate based causal inference techniques," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 649–660.

[58] Y. Gao, Z. Zhang, L. Zhang, C. Gong, and Z. Zheng, "A theoretical study: The impact of cloning failed test cases on the effectiveness of fault localization," in *2013 13th International Conference on Quality Software*. IEEE, 2013, pp. 288–291.

[59] H. Wang, B. Du, J. He, Y. Liu, and X. Chen, "Ietcr: An information entropy based test case reduction strategy for mutation-based fault localization," *IEEE Access*, vol. 8, pp. 124 297–124 310, 2020.

[60] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[61] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 664–676.

[62] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in *European conference on object-oriented programming*. Springer, 2005, pp. 528–550.

[63] C. Yilmaz, A. Paradkar, and C. Williams, "Time will tell," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 81–90.

[64] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 04, pp. 573–597, 2009.

[65] T.-D. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? a case with spectrum-based fault localization," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 380–383.

[66] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist," *RN*, vol. 14, no. 14, p. 14, 2014.

[67] X. Xie, F. C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *Proceedings of the 2013 International Symposium on Search Based Software Engineering*. Springer Berlin Heidelberg, 2013, pp. 224–238.

[68] W. E. Wong, L. Zhao, Y. Qi, K.-Y. Cai, and J. Dong, "Effective fault localization using bp neural networks." in *SEKE*. Citeseer, 2007, pp. 374–379.

[69] G. E. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *ACM SIGKDD explorations newsletter*, vol. 6, no. 1, pp. 20–29, 2004.

[70] H. Lee, M. Park, and J. Kim, "Plankton classification on imbalanced large scale database via convolutional neural networks with transfer learning," in *2016 IEEE international conference on image processing (ICIP)*. IEEE, 2016, pp. 3713–3717.

[71] M. Buda, A. Maki, and M. A. Mazurowski, "A systematic study of the class imbalance problem in convolutional neural networks," *Neural networks*, vol. 106, pp. 249–259, 2018.

[72] S. Wang, W. Liu, J. Wu, L. Cao, Q. Meng, and P. J. Kennedy, "Training deep neural networks on imbalanced data sets," in *2016 international joint conference on neural networks (IJCNN)*. IEEE, 2016, pp. 4368–4374.

[73] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[74] C. Huang, Y. Li, C. C. Loy, and X. Tang, "Learning deep representation for imbalanced classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 5375–5384.

[75] S. Ando and C. Y. Huang, "Deep over-sampling framework for classifying imbalanced data," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2017, pp. 770–785.

[76] Q. Dong, S. Gong, and X. Zhu, "Imbalanced deep learning by minority class incremental rectification," *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 6, pp. 1367–1381, 2018.

[77] R. Polikar, "Ensemble based systems in decision making," *IEEE Circuits and systems magazine*, vol. 6, no. 3, pp. 21–45, 2006.

[78] L. Rokach, "Ensemble-based classifiers," *Artificial intelligence review*, vol. 33, no. 1, pp. 1–39, 2010.

[79] N. V. Chawla, A. Lazarevic, L. O. Hall, and K. W. Bowyer, "Smoteboost: Improving prediction of the minority class in boosting," in *European conference on principles of data mining and knowledge discovery*. Springer, 2003, pp. 107–119.

[80] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Napolitano, "Rusboost: A hybrid approach to alleviating class imbalance," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 40, no. 1, pp. 185–197, 2009.

[81] J. Błaszczyński, M. Deckert, J. Stefanowski, and S. Wilk, "Integrating selective pre-processing of imbalanced data with ivotes ensemble," in *International conference on rough sets and current trends in computing*. Springer, 2010, pp. 148–157.

[82] X.-Y. Liu, J. Wu, and Z.-H. Zhou, "Exploratory undersampling for class-imbalance learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 2, pp. 539–550, 2008.

[83] S. Wang and X. Yao, "Diversity analysis on imbalanced data sets by using ensemble models," in *2009 IEEE symposium on computational intelligence and data mining*. IEEE, 2009, pp. 324–331.