IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 11, 2022, accepted February 11, 2022, date of publication February 22, 2022, date of current version March 1, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3153347

Automated Software Test Data Generation With Generative Adversarial Networks

XIUJING GUO, (Member, IEEE), HIROYUKI OKAMURA, (Member, IEEE), AND TADASHI DOHI, (Member, IEEE)
Graduate School of Advanced Science and Engineering, Hiroshima University, Hiroshima 739-8511, Japan
Corresponding author: Xiujing Guo (guoxiujing1994@yahoo.co.jp)

ABSTRACT With the rapid increase of software scale and complexity, the cost of traditional software testing methods will increase faster than the scale of software. In order to improve test efficiency, it is particularly important to automatically generate high-quality test cases. This paper introduces a framework for automatic test data generation based on the generative adversarial network (GAN). GAN is employed to train a generative model over execution path information to learn the behavior of the software. Then we can use the trained generative model to produce new test data, and select the test data that can improve the branch coverage according to our proposed selection strategy. Compared to prior work, our proposed method is able to handle programs under test with large-scale branches without analyzing branch expressions. In the experiment, we exhibit the performance of our method by using two modules in GNU Scientific Library. In particular, we consider the application of our method in two testing scenarios; unit testing and integration testing, and conduct a series of experiments to compare the performance of three types of GAN models. Results indicate that the WGAN-GP shows the best performance in our framework. Compared with the random testing method, the WGAN-GP based framework improves the test coverage of five functions out of the seven in the unit testing.

INDEX TERMS Software testing, test case generation, test input, execution path, GAN.

I. INTRODUCTION

Software testing is one of the most important activities to build a reliable software system. In general, the software testing is divided into two categories; static testing and dynamic testing. The static testing involves inspections of program codes and their associated documents without concrete execution of the program. The symbolic execution is one of the typical approaches for static testing. On the other hand, the dynamic testing verifies the correctness of program by checking the dynamic behavior when the program runs with some inputs, and is commonly used for the software testing. In our research, we focus on the dynamic testing.

In the dynamic testing, the test case generation plays a central role to ensure the software quality. The test case consists of test inputs and their expected outcomes by the test oracle. A common procedure of software testing is (i) to execute the software under test (SUT) with the inputs of test cases, and (ii) to compare the test outcomes with the expected ones. If the test outcome is different from the expected one, the SUT involves bugs. On the other hand, from the viewpoint of software reliability, the software reliability may grow as the number of test cases increases. However, a large number of test cases cause the large amount of cost on software testing. Thus it is a challenge to consider how to reduce the number of test cases keeping a certain level of software reliability. The artificial intelligence (AI) is one of the key technologies to address this issue, and the AI-based software testing is expected to be a killer application for highly-reliable software development.

Though the AI becomes one of the most attractive technologies in recent years, it has already been applied to software testing in the past literature; for example, the automatic generation of test cases and test codes, the analysis of the large-scale test logs, automatic exploratory testing, and defect location. In particular, there have been different approaches for automation of software testing with AI, and they are roughly divided into three categories:

The associate editor coordinating the review of this manuscript and approving it for publication was Jerry Chun-Wei Lin.

1) Search-based software testing (SBST): The SBST is a method to generate test inputs with search techniques, e.g., meta-heuristic optimization methods, so that it can achieve high coverage [1]. Fraser and Arcuri [2] presented an extension of SBST that is able to exercise automated oracles and to produce high coverage test suites at the same time. Zhan and Clark [3] applied search techniques to test data generation for Simulink models. In this approach, they proposed a full test-set generation framework, which successfully combines random testing with the search-based targeted test-data generation techniques to enable us to generate effective test sets.
2) Test oracle generation: The test oracle is a mechanism to check whether the test result is correct or not. In the real situation, this is done by the expected test outcomes. However, since the expected test outcomes should be manually generated, it is the most costly process in software testing. There are several types of research results for the generation of the expected output with the artificial neural network (ANN) [4]–[7]. Valueian *et al.* [4] proposed a classifier-based method using ANNs that can build automated oracles for embedded software that has low observability and/or produces unstructured or semi-structured outputs. In the proposed approach, the oracles need input data tagged with two labels of ''pass'' and ''fail'' rather than outputs and any execution trace.
3) Fuzzing data generation: Lyu *et al.* [8] employed a GAN model to learn the characteristics of valuable files and then generated valuable seed files for the fuzzing test. Li *et al.* in [9] introduced a method using Wasserstein generative adversarial networks (WGANs) to generate fuzzing data for ICS testing. Moreover, Joffe and Clark [10] proposed an SBST framework based on a deconvolutional generative NN and compared the performance of framework with a famous fuzzing tool: AFL (American fuzzy lop) fuzzer. Zong *et al.* in [11] proposed a method based on deep learning to predict the reachability of inputs before executing the target program, helping grey-box fuzzing filter out the unreachable ones to improve the performance of fuzzing. For industrial control systems, Hu *et al.* [12] proposed an automatical and intelligent fuzzing framework(GANFuzz) for testing implementations of industrial network protocols, in which the protocol grammar is learned by deep learning. More recently, Lv *et al.* [13] designed an automated and intelligent fuzzing framework BLSTM-DCNNFuzz for industry control protocols.

Our objective is to generate test data that can achieve the full test coverage. The test coverage is a fundamental metric representing the quality of software testing. The test coverage is defined as a fraction of executed test path in testing over the entire execution paths of SUT [15]. In general, since it is difficult to measure the entire execution paths, statement and branch coverage are used instead of path coverage in practice. As mentioned before, the SBST is one of the most popular methods for this purpose. For example, to achieve the branch coverage [16], the SBST uses symbolic execution to extract branch conditions and uses an optimization algorithm to find the test input that satisfies the branch condition. However, with the increasing scale and complexity of the software system, it is complicated and time-consuming to carry out large-scale branch coverage.

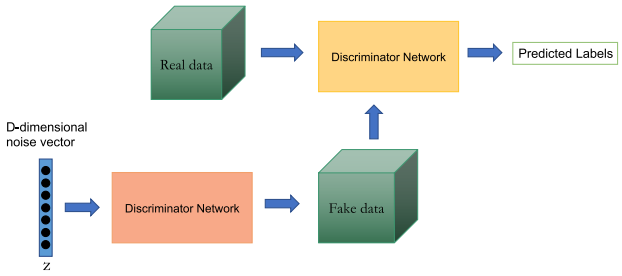

FIGURE 1. The architecture of GAN.

In this paper, we propose a software testing framework with generative adversarial networks (GANs). Concretely, we consider a GAN model to generate test data for the SUT, and a test strategy to increase the test coverage with the GAN. The GAN is a NN model for the data generation from given training data. A GAN consists of a generator and a discriminator, and the most important feature of GAN is to reinforce the ability of data generation for the generator by competing to the discriminator to detect fake data [17]. Our contributions are

1) We propose a GAN model to generate test inputs and their associated execution paths simultaneously. That is, in our model, the discriminator learns program execution paths for SUT and the generator generates test cases including test inputs and expected paths.
2) We discuss a test data selection strategy to increase the test coverage.
3) We investigate the applicability of our framework for two software testing scenarios; unit testing and integration testing. The experimental comparison proves that our method is better than the common random test.

The remaining parts of the paper are organized as follows. Section II is dedicated to the review of GAN models. Section III describes the details of the proposed framework. In Section IV, we exhibit the experiment for the proposed software testing with GAN. Section V concludes the paper with some remarks and presents the future direction.

II. GAN: GENERATIVE ADVERSARIAL NETWORK

In this section, we first introduce the GAN model and then present the input and output of GAN in our framework.

GANs are algorithmic architectures that use two neural networks, pitting one against the other (thus the ''adversarial'') in order to learn the underlying distribution of the

```
#include<stdio.h>
void main()
{
    int marks;
    printf("Enter your marks ");
    scanf("%d",&marks);
    if(marks<0 || marks>100)
    {
        printf("Wrong Entry");
    }
    else if(marks<60)
    {
        printf("Grade F");
    }
    else if(marks>=60 && marks<70)
    {
        printf("Grade D");
    }
    else if(marks>=70 && marks<80)
    {
        printf("Grade C");
    }
    else if(marks>=80 && marks<90)
    {
        printf("Grade B");
    }
    else
    {
        printf("Grade A");
    }

}
```

FIGURE 2. A code of `grade.c`.

```
    -:    0:Source:grade.c
    -:    0:Graph:grade.gcno
    -:    0:Data:grade.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:#include<stdio.h>
function main called 1 returned 100% blocks executed 61%
    1:    2:void main()
    -:    3:{
    -:    4:    int marks;
    1:    5:    printf("Enter your marks ");
    1:    6:    scanf("%d",&marks);
    1:    7:    if(marks<0 || marks>100)
branch  0 taken 1
branch  1 taken 0
branch  2 taken 0
branch  3 taken 1
    -:    8:    {
#####:    9:        printf("Wrong Entry");
#####:   10:    }
    1:   11:    else if(marks<60)
branch  0 taken 0
branch  1 taken 1
    -:   12:    {
#####:   13:        printf("Grade F");
#####:   14:    }
    1:   15:    else if(marks>=60 && marks<70)
branch  0 taken 1
branch  1 taken 0
branch  2 taken 0
branch  3 taken 1
    -:   16:    {
#####:   17:        printf("Grade D");
#####:   18:    }
    1:   19:    else if(marks>=70 && marks<80)
branch  0 taken 1
branch  1 taken 0
branch  2 taken 1
branch  3 taken 0
    -:   20:    {
    1:   21:        printf("Grade C");
    1:   22:    }
#####:   23:    else if(marks>=80 && marks<90)
branch  0 never executed
branch  1 never executed
branch  2 never executed
branch  3 never executed
    -:   24:    {
#####:   25:        printf("Grade B");
#####:   26:    }
    -:   27:    else
    -:   28:    {
#####:   29:        printf("Grade A");
    -:   30:    }
    -:   31:
    1:   32:}
    -:   33:
```

FIGURE 3. A code of `grade.c.gcov`.

training data so that it can generate new data instances that resemble the training data [14]. Fig. 1 is the architecture of GAN. It consists of a generator and a discriminator. The generator takes random noise z and generates fake data. The discriminator determines if the generated data are real or fake. GAN estimates generative models via an adversarial process. In the adversarial process, the generator aims to maximize the failure rate of the discriminator while the discriminator aims to minimize it. The GAN model converges when the Nash equilibrium is reached [17].

One of the most challenging problems when AI technologies are applied to unstructured data such as computer program and software testing is to determine what data are used as inputs of the AI system. Our main idea is to use not only the test inputs but also the corresponding execution paths as inputs of GAN. A pair of test inputs and program execution paths includes rich information on the program itself. Also, it is not difficult to observe and collect the execution path for an input in the dynamic testing and it is more reasonable in terms of cost than symbolic execution used in the SBST.

In this paper, we focus on branch coverage. Therefore, the execution path in our work is extracted by Gcov tool [18] and is defined as the combination of the execution of each branch in the program under a specific input. Gcov is a source code coverage analysis and statement-by-statement profiling tool that can generate exact counts of the number of times each statement in a program is executed, and that annotates source code to add instrumentation. Gcov is released with GCC and cooperates with GCC to realize statement coverage and branch coverage testing of C/C++ files. When executing a program, we use Gcov to write branch frequencies to the output file, and then extract the branch information in the file as a path. For example, Fig. 2 is a C program for checking the grade of the students based on marks. When the input score is 79, one uses GCC and Gcov to compile, run and collect the information to get the "`grade.c.gcov`" file, as shown in Fig. 3. We convert the "`taken a(a>0)`" to "`1`" to represent that the branch is taken at least once, convert the "`taken 0`" to "`0`" to represent that the branch is not taken, and convert the "`never executed`" to "`−1`" to represent that the branch is not executed. Then the execution path of the program `grade.c` under the input 79 is the combination of all branch executions; `1,0,0,1,0,1,1,0,0,1,1,0,1,0,−1,−1,−1,−1`.

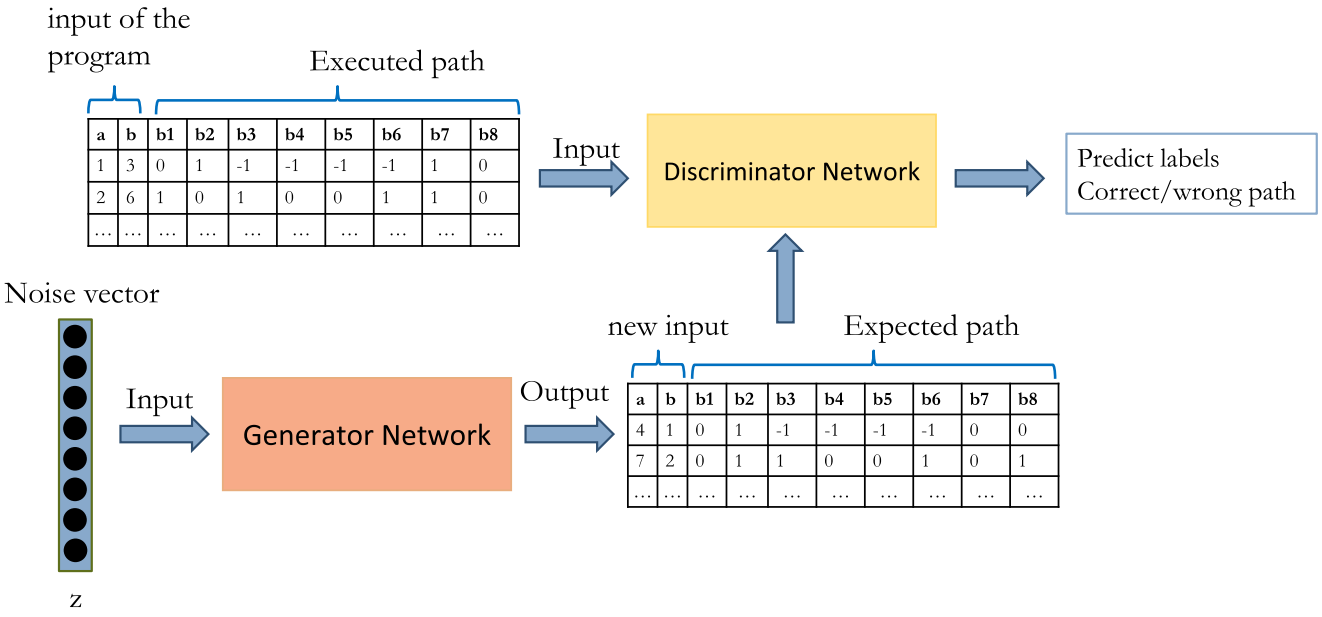

FIGURE 4. The input and output of GAN in our framework ("b1 = 0" represents that branch1 is not taken; "b2 = 1" represents that branch2 is taken at least once; "b3 = −1" represents that branch3 is not executed).

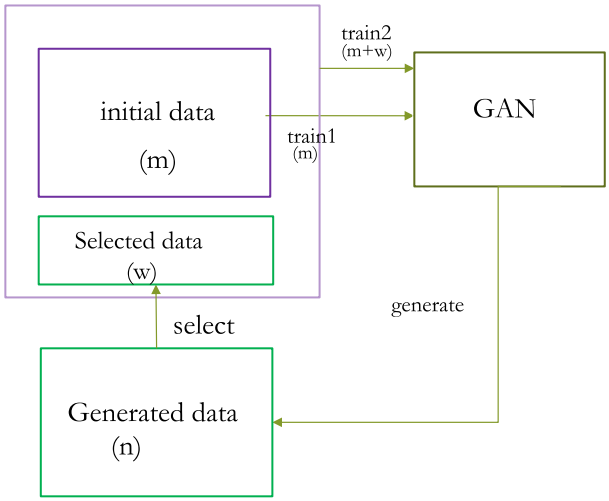

FIGURE 5. A framework for the test case generation indenting to increase test coverage.

Fig. 4 illustrates the input and output of GAN in our framework. The input of GAN is a noise vector and a set of inputs of the program with their corresponding executed paths extracted by Gcov. The output of GAN is a set of new inputs of the program with their corresponding expected paths generated by generator. In the successfully trained GAN model, the generator can generate the correct path for a specific generated input, and the discriminator can accurately determine whether the path corresponding to the specific input is correct. Therefore, the generator can be used to generate test cases and improve test coverage based on generated path information, and the discriminator can be used as an evaluator.

III. SOFTWARE TESTING WITH GAN

We propose a framework for automated test data generation and aim to achieve the full branch coverage in software testing. In our framework, we use GAN to iteratively generate test inputs for software testing. Fig. 5 illustrates the structure of the framework. It contains the following four steps:

Step 1: Select m test inputs and their corresponding paths as training data.

Step 2: Train GAN with training data to generate n data.

Step 3: According to the path information of the generated data, select $w(w < n)$ inputs data from n data sets that may cover not-executed branches in the training data.

Step 4: Add the selected w input data and their corresponding executed paths to the training data. Go to Step 2.

Fig. 6 illustrate an example of our framework. We assume that the program under test has two inputs a and b, and four branches $b1 - b4$. In step1, we select 10 test inputs and their corresponding executed paths as training data. In the training data, branch 2 and branch 4 are not taken. In step2, train GAN with training data to generate n test inputs and their corresponding expected paths. To improve branch coverage, we want to generate test data that covers branch 2 and branch 4. Therefore, in step 3 we select the inputs "a = 12, b = 3; …; a = 2, b = 9; a = 2, b = 8; a = 2, b = 6" where generated path information shows that branch 2 and branch 4 are taken. In step4, we execute the program with selected inputs to extract real paths, add them into the training data. We can see that in the extracted real paths of the selected inputs, branch 2 is covered and branch 4 is not covered. With a high-precision GAN model, our framework has the ability to improve test coverage.

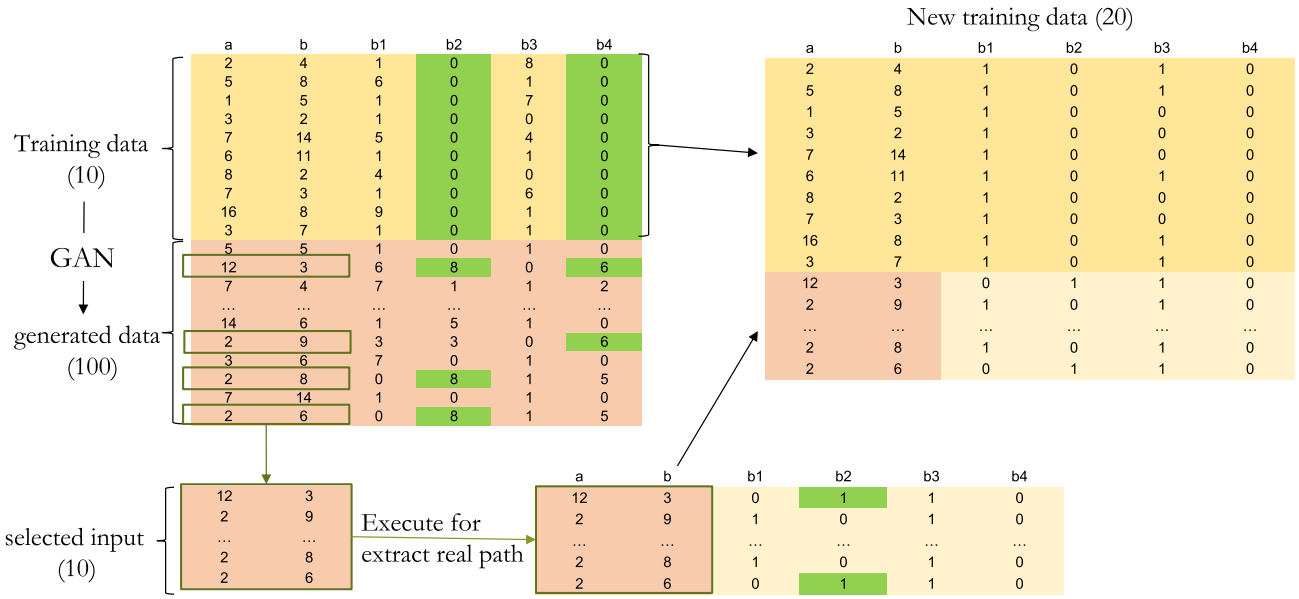

FIGURE 6. An example of proposed framework.

TABLE 1. The functions names.

gamma_inc.c	Function1	gsl_sf_gamma_inc_Q_e
	Function2	gsl_sf_gamma_inc_P_e
	Function3	gsl_sf_gamma_inc_e
hyperg_1F1.c	Function1	hyperg_1F1_1_small_a_bgt0
	Function2	hyperg_1F1_ab_posint
	Function3	hyperg_1F1_ab_pos
	Function4	hyperg_1F1_ab_neg
	Function5	gsl_sf_hyperg_1F1_inc_e

IV. EXPERIMENT

In this section, we conduct a series of experiments to evaluate the effectiveness of the proposed framework. First, we applied our framework to unit testing and integration testing, and next selected three GAN models: WGAN-GP [20], BiGAN [22], and standard GAN [17] in our experiments. Our experiments aim to evaluate which type of GAN is more suitable for the framework, and which test level is suitable for our framework.

A. EXPERIMENTAL SETUP

In the past few years, different variants of GAN have been proposed. In our experiments, we choose three GAN models to apply to our framework: WGAN-GP, BiGAN, and the standard GAN. WGAN-GP is an improvement of WGAN [21] with a gradient norm penalty. The main difference from the standard GAN is only the cost function. WGAN-GP is based on the Wasserstein distance that has a smoother gradient everywhere. Therefore it has the higher training stability than the standard GAN [20]. BiGAN is a type of GAN where the generator not only maps latent samples to generated data, but also has an inverse mapping from data to the latent representation [22].

We apply our proposed framework to two software testing levels: unit testing and integration testing, conduct two sets of experiments on WGAN-GP, BiGAN, and the standard

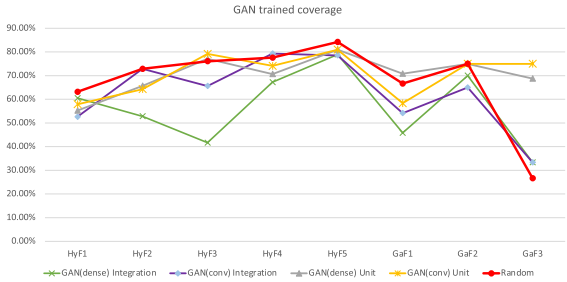

FIGURE 7. Coverage of GAN generated data.

GAN. Unit testing is a type of software testing in which individual units or components of the software are tested. The purpose is to verify that each unit of the software code executes as expected. A unit can be a single function, method, procedure, module, or object. Integration testing is a level of software testing in which individual units/components are combined and tested as a whole. The purpose of this test is to expose errors in the interaction between integration units. Integration testing can be performed by the following strategies: incremental approach and big bang approach. In the incremental test method, the test is performed by integrating two or more logically related modules, and then the function of the application is tested. Then, other related modules are integrated incrementally, and this process is repeated until all logic-related modules are successfully integrated and tested. Big bang integration test is an integration test strategy that combines all the units at once to form a complete system, and then tests the unity of different units as one entity. In our unit test experiment, the GAN model generates paths in function units. In the integration test experiment, we adopt the big bang strategy and use GAN to generate paths for all functions at once.

TABLE 2. GAN: Comparison of the coverage achievement.

		No. of branches	methods	GAN(dense)		GAN(conv)		RAND
				Initial coverage	Trained coverage	Initial coverage	Trained coverage	Random coverage
hyperg_1F1.c	All	594	Integration	34.84%	42.93%	34.84%%	51.34%	45.12%
	Function1	38	Integration	47.36%	60.52%	47.36%	52.63%	63.15%
			Unit	36.84%	55.26%	36.84%	57.89%	
	Function2	70	Integration	41.42%	52.85%	41.42%	72.85%	72.85%
			Unit	41.43%	65.71%	41.43%	64.28%	
	Function3	96	Integration	35.41%	41.66%	35.41%	65.62%	76.04%
			Unit	71.87%	77.08%	71.87%	79.16%	
	Function4	58	Integration	44.82%	67.24%	44.82%	79.31%	77.58%
			Unit	56.89%	70.68%	56.89%	74.13%	
	Function5	42	Integration	63.15%	78.94%	63.15%	78.57%	84.21%
			Unit	59.52%	80.95%	59.52%	80.95%	
gamma_inc.c	All	138	Integration	49.27%	51.45%	49.27%	52.17%	50.00%
	Function1	24	Integration	45.83%	45.83%	45.83%	54.16%	66.66%
			Unit	54.16%	70.83%	54.16%	58.33%	
	Function2	20	Integration	55.00%	70.00%	55.00%	65.00%	75.00%
			Unit	75.00%	75.00%	75.00%	75.00%	
	Function3	16	Integration	33.33%	33.33%	33.33%	33.33%	27%
			Unit	68.75%	68.75%	68.75%	75.00%	

TABLE 3. BiGAN: Comparison of the coverage achievement.

		No. of branches	methods	BiGAN(dense)		BiGAN(conv)		RAND
				Initial coverage	Trained coverage	Initial coverage	Trained coverage	Random coverage
hyperg_1F1.c	All	594	Integration	34.84%	34.65%	34.84%	46.80%	45.12%
	Function1	38	Integration	47.36%	47.36%	47.36%	52.63%	63.15%
			Unit	36.84%	52.63%	36.84%	52.63%	
	Function2	70	Integration	41.42%	41.42%	41.42%	67.14%	72.85%
			Unit	41.43%	41.43%	41.43%	65.71%	
	Function3	96	Integration	35.41%	43.75%	35.41%	55.20%	76.04%
			Unit	71.87%	72.91%	71.87%	76.04%	
	Function4	58	Integration	44.82%	40.74%	44.82%	65.51%	77.58%
			Unit	56.89%	60.34%	56.89%	62.06%	
	Function5	42	Integration	63.15%	73.68%	63.15%	78.57%	84.21%
			Unit	59.52%	71.42%	59.52%	76.19%	
gamma_inc.c	All	138	Integration	49.27%	56.52%	49.27%	52.89%	50.00%
	Function1	24	Integration	45.83%	66.66%	45.83%	54.16%	66.66%
			Unit	54.16%	75.00%	54.16%	58.33%	
	Function2	20	Integration	55.00%	80.00%	55.00%	70.00%	75.00%
			Unit	75.00%	85.00%	75.00%	75.00%	
	Function3	16	Integration	33.33%	33.33%	33.33%	33.33%	27%
			Unit	68.75%	68.75%	68.75%	75.00%	

In both experiments, the parameters of the framework are set to $m = 100$, $n = 100$, $w = 10$, and are iterated 10 times to generate 100 test data. In the integration testing, the 100 initial data are randomly generated. In the unit testing, we do not directly consider the input parameters of a single function, but consider the input of the main function. Therefore, the initial data of each function in unit testing is the selected input data that can pass the objective function. Regarding the network structure of the generator and discriminator in the GAN model, we applied a simple fully connected neural network (dense layers) and a convolutional neural network for comparison. The GAN model trains 5000 epochs per iteration in the unit testing, and 10000 epochs per iteration in the integration testing (one epoch is the cycle when an entire dataset is passed forward and backward through the neural network only once.) Training hyperparameters are shown in Appendix. In the BiGAN models, the encoder model is an inverse mapping from data to the latent representation. Since the authors of the paper [20] recommend to use layer normalization as a drop-in

TABLE 4. WGAN-GP: Comparison of the coverage achievement.

		No. of branches	methods	WGAN-GP(dense)		WGAN-GP(conv)		RAND
				Initial coverage	Trained coverage	Initial coverage	Trained coverage	Random coverage
hyperg_1F1.c	All	594	Integration	34.84%	51.42%	34.84%	51.34%	45.12%
	Function1	38	Integration	47.36%	57.89%	47.36%	60.52%	63.15%
			Unit	36.84%	63.15%	36.84%	55.26%	
	Function2	70	Integration	41.42%	62.85%	41.42%	74.28%	72.85%
			Unit	41.43%	84.28%	41.43%	75.71%	
	Function3	96	Integration	35.41%	75.00%	35.41%	64.58%	76.04%
			Unit	71.87%	83.33%	71.87%	90.62%	
	Function4	58	Integration	44.82%	74.13%	44.82%	72.41%	77.58%
			Unit	56.89%	84.48%	56.89%	70.68%	
	Function5	42	Integration	63.15%	76.31%	63.15%	76.19%	84.21%
			Unit	59.52%	80.95%	59.52%	76.19%	
gamma_inc.c	All	138	Integration	49.27%	52.17%	49.27%	52.89%	50.00%
	Function1	24	Integration	45.83%	54.16%	45.83%	54.16%	66.66%
			Unit	54.16%	75.00%	54.16%	66.66%	
	Function2	20	Integration	55.00%	65.00%	55.00%	70.00%	75.00%
			Unit	75.00%	80.00%	75.00%	90.00%	
	Function3	16	Integration	33.33%	33.33%	33.33%	45.45%	27%
			Unit	68.75%	80.00%	68.75%	75.00%	

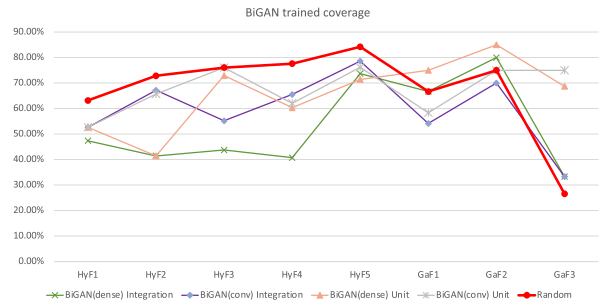

FIGURE 8. Coverage of BiGAN generated data.

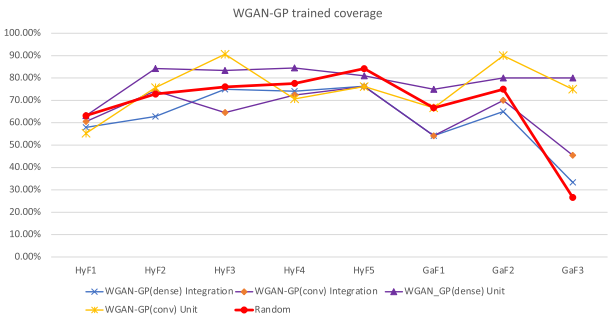

FIGURE 9. Coverage of WGAN-GP generated data.

replacement for batch normalization in WGAN-GP to help stabilize training. The same authors also use the RMSProp optimizer in their experiment. Therefore, we also use the RMSProp optimizer and layer normalization in WGAN-GP, and use the Adam optimizer and batch normalization in other GAN models.

We have considered two modules in the GNU Scientific Library (a numerical library for C and C++ programmers) [19]: the `gamma_inc.c` module and the `hyperg_1F1.c` module. The `gamma_inc.c` module is used to compute the incomplete Gamma function. It contains 13 functions including branches, where the total number of branches is 138. The `hyperg_1F1.c` module is used to calculate the confluent hypergeometric function. It contains 21 functions including branches, where the total number of branches is 592. In the integration testing, the GAN model generates path information including all branches. In the unit testing, we focus on several functions in these modules. Concretely, 3 functions in `gamma_inc.c` module and 5 functions in `hyperg_1F1.c` module are picked up to make their test coverage increase (see Table 1).

We compared the test coverage of the data generated by the experiment and compared the accuracy of the generated path, to verify whether the framework can generate data that can improve the test coverage, and to investigate which version of GAN model is more suitable for our framework. We also conducted a random test to generate 200 test data to compare the test coverage with the GAN method.

B. RESULTS

Test coverage is the main evaluation metric in our experiments. Tables 2 to 4 show the test coverage results of integration testing and unit testing under three models and the random test's test coverage. The initial coverage represents the percent of branches executed by 100 initial training data. The trained coverage represents the percentage of branches executed by 200 input data, where 200 input data contains 100 initial training data and 100 GAN generated data. In order to facilitate comparison, we divide the data generated in the integration testing into function units to compare with the unit testing results. Figs. 7 to 9 present the analysis of the

TABLE 5. Comparison of the accuracy of the paths predicted by the three GAN models.

		No. of branches	methods	Average path accuracy					
				GAN(dense)	GAN(conv)	BiGAN(dense)	BiGAN(conv)	WGAN-GP(dense)	WGAN-GP(conv)
hyperg_1F1.c	All	594	Integration	28.33%	36.45%	51.43%	29.04%	44.82%	78.23%
	Function1	38	Integration	26.00%	35.02%	34.26%	28.65%	37.78%	94.52%
			Unit	41.39%	41.84%	17.36%	65.00%	86.13%	75.31%
	Function2	70	Integration	23.22%	37.87%	49.78%	29.42%	42.27%	92.31%
			Unit	41.98%	37.68%	62.87%	33.00%	82.77%	64.41%
	Function3	96	Integration	25.64%	36.16%	49.71%	29.41%	38.10%	93.00%
			Unit	36.16%	40.98%	28.55%	35.13%	73.03%	59.63%
	Function4	58	Integration	28.89%	36.74%	41.37%	30.77%	27.96%	53.17%
			Unit	40.10%	28.15%	53.68%	37.01%	79.00%	52.93%
	Function5	42	Integration	37.11%	35.92%	39.97%	31.14%	41.54%	42.66%
			Unit	33.16%	43.83%	23.88%	26.19%	84.36%	41.97%
gamma_inc.c	All	138	Integration	30.07%	37.47%	22.48%	28.60%	28.20%	34.09%
	Function1	24	Integration	20.83%	32.25%	34.20%	35.29%	17.91%	30.83%
			Unit	32.66%	32.41%	22.41%	37.66%	76.42%	17.12%
	Function2	20	Integration	15.00%	30.55%	39.55%	37.75%	23.35%	31.60%
			Unit	39.90%	31.47%	22.97%	31.23%	64.35%	27.75%
	Function3	16	Integration	31.25%	28.75%	24.81%	28.25%	36.43%	34.68%
			Unit	35.62%	28.06%	24.25%	27.50%	82.81%	38.37%

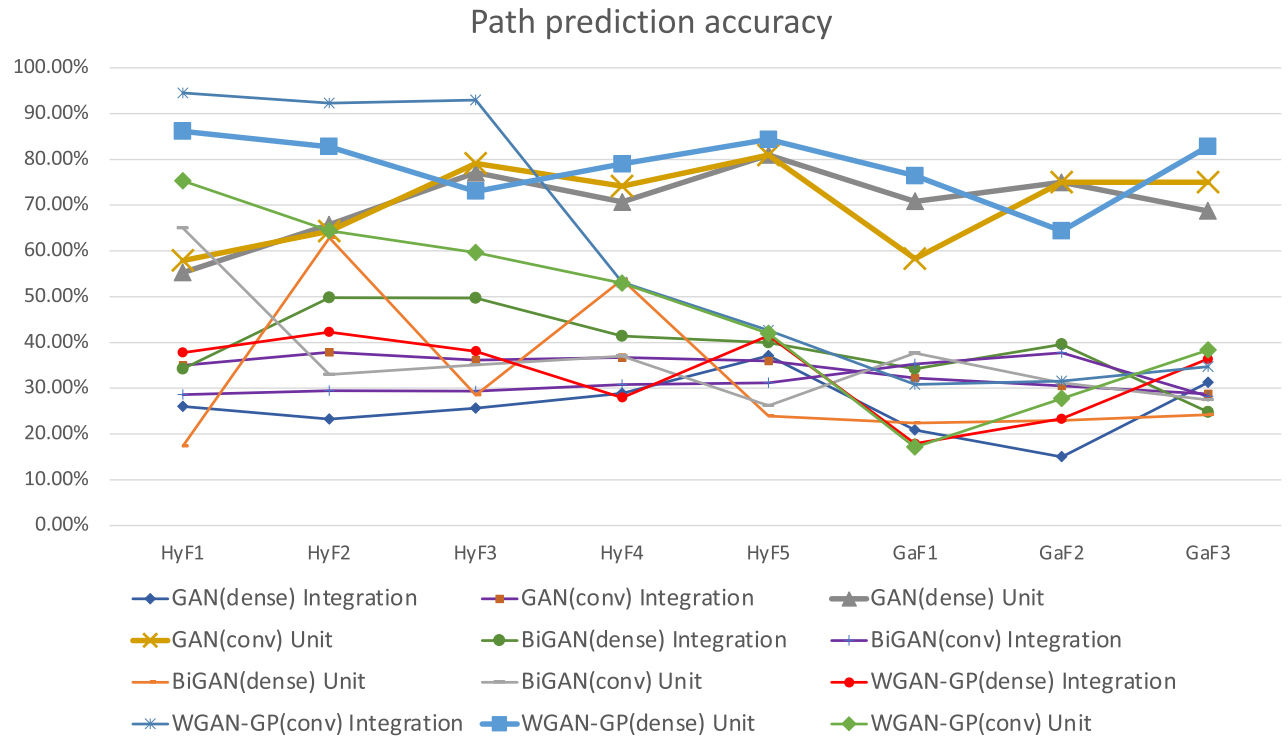

FIGURE 10. Path prediction accuracy.

coverage results. As can be seen in Fig. 7 that in addition to Function 1 in the hyperg_1F1.c module, the test coverage of data generated by GAN (dense) in unit tests is higher than that of data generated in integration tests. Also, the test coverage of data generated by GAN (conv) in unit tests is higher than that of data generated in integration tests except for Function 2 and Function 4 in the hyperg_1F1.c module. Despite of this result, only a few of the data generated by the GAN model have a coverage rate that exceeds the random testing. Furthermore, BiGAN does not show good performance in our framework, because the coverage of the data generated by BiGAN is also mostly not better than random testing (see Fig. 8). However, in the unit testing, the coverage of test cases generated by WGAN-GP (dense) is higher than that of randomly generated test cases except for Function 1 and Function 5 in the hyperg_1F1.c module.

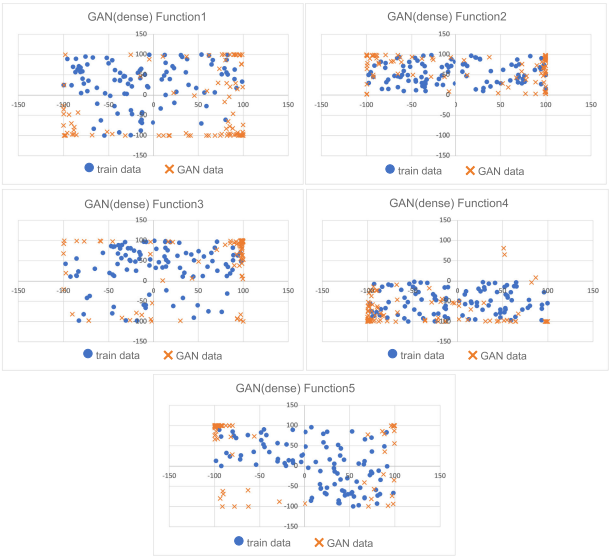

FIGURE 11. Data distribution is generated by the standard GAN with fully connected network.

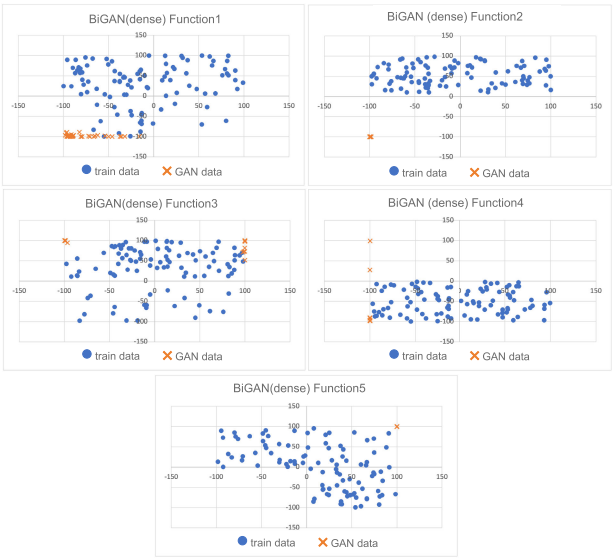

FIGURE 13. Data distribution is generated by BiGAN with fully connected network.

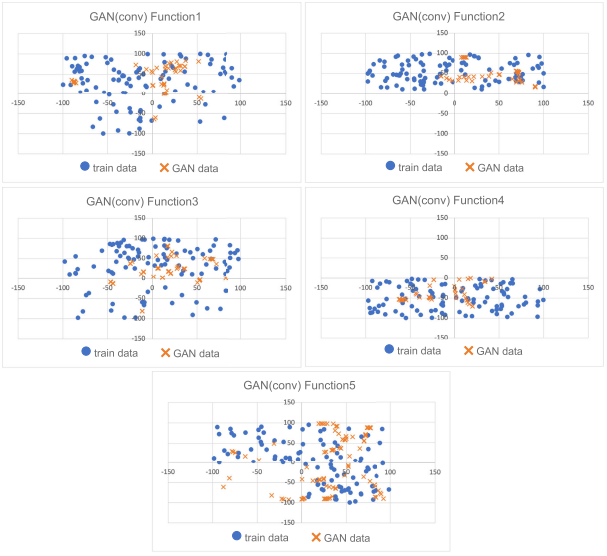

FIGURE 12. Data distribution is generated by the standard GAN with convolutional network.

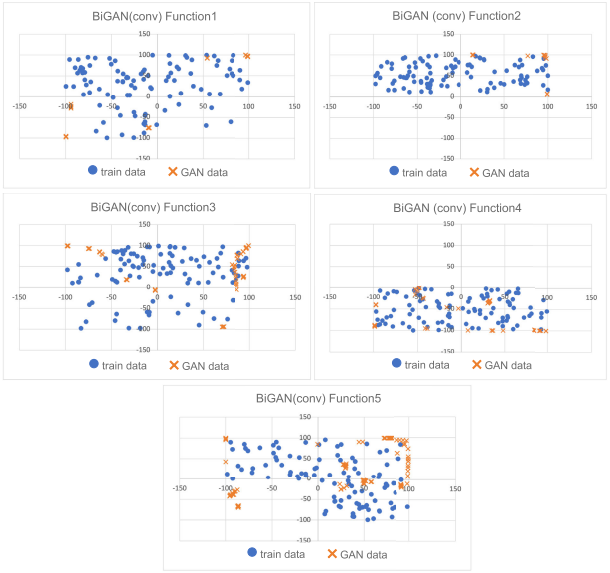

FIGURE 14. Data distribution is generated by BiGAN with convolutional network.

Further, the WGAN-GP (conv) improves the test coverage of Function 3 in hyperg_1F1.c module and Function 2 in gamma_inc.c module.

Our framework leverages high-accuracy GAN models to improve test coverage. Therefore, we also analyze the accuracy of the paths predicted by the three GAN models. Table 5 shows the test coverage results, and the accuracy is calculated by

$$\text{Accuracy} = \frac{\text{The number of correct branchs}}{\text{The number of total branchs}} \times 100\%. \quad (1)$$

Fig. 10 graphically shows the path prediction accuracy of three sets of models. On the whole, the WGAN-GP (dense), the GAN (dense), and the GAN (conv) have higher path prediction accuracy than the others in unit testing.

We also analyze the distribution of data generated by three types of GAN models. Fig. 11- Fig. 16 illustrate the data distribution of 3 functions in hyperg_1F1.c module generated by the standard GAN, biGAN, and WGAN-GP, respectively. As can be seen from these figure, both fully connected BiGAN and convolutional BiGAN have problems with mode collapse, that is, the generator collapses and produces a limited variety of samples. Meanwhile, the data generated by the GAN model using the fully connected neural network is also easy to move to the edge. Only convolutional GAN and WGAN-GP can generate diverse data for our strategy.

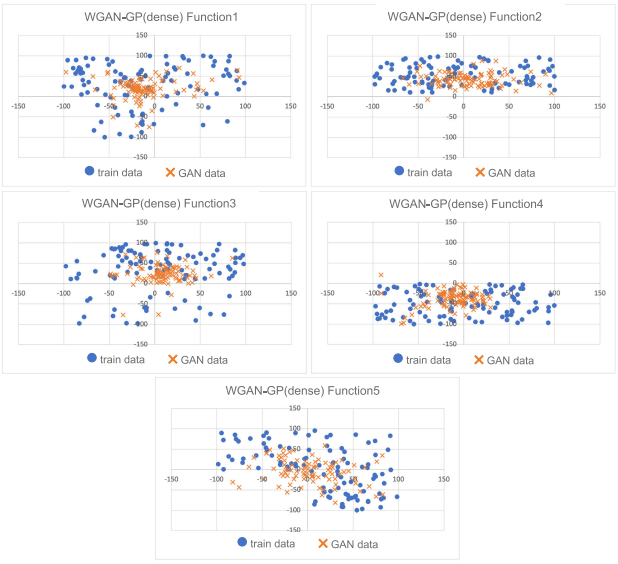

FIGURE 15. Data distribution is generated by WGAN-GP with fully connected network.

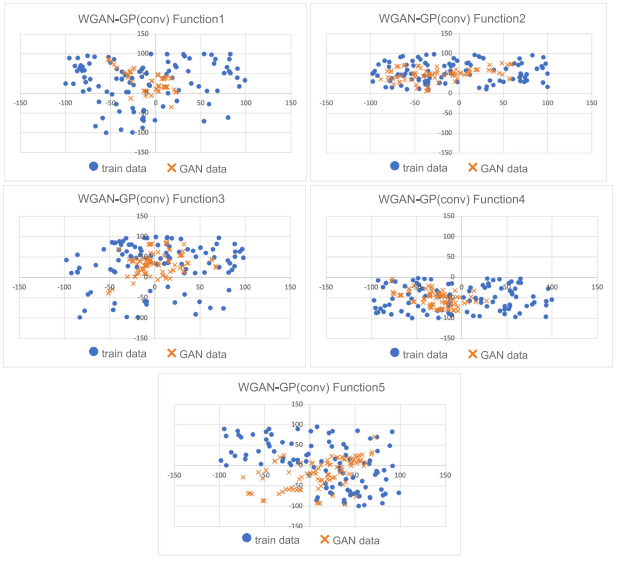

FIGURE 16. Data distribution is generated by WGAN-GP with convolutional network.

In summary, compared with standard GAN and BiGAN, WGAN-GP can create new data instances that resemble our training data, so that it showed better performance in our framework. And our framework is more appropriate for unit testing. This is because taking the function as a unit can reduce the complexity of the program and improve the model's path prediction accuracy. Overall, our proposed approach outperforms random testing in improving test coverage. However, in terms of time cost, GAN-based methods consume more time than random methods. In the experiment, we trained GAN ten times to generate 100 test cases, and each training takes an average of one minute. Therefore, in future work, we also need to consider optimizing the GAN model to shorten the training time and reduce the time cost of our proposed framework.

V. CONCLUDING REMARK

In this paper, we proposed an automatic test data generation framework based on Generative Adversarial Network (GAN), aim at achieving full branch coverage. In the framework, we used not only the test inputs but also the corresponding execution path as inputs of GAN. The generator of GAN was used to generate test input and the corresponding expected path. According to the path information, we selected the data that may cover the new branch. The discriminator was used to reinforce the generation ability of the generator so that the generator could generate data that conforms to the true distribution. In this way, our coverage improvement strategy was meaningful.

We conducted the experiments to examine the effectiveness of our proposed framework. Firstly, we applied the framework in two software testing levels: integration testing and unit testing, and then compared the path accuracy and test coverage of three types of GAN models with two structures: fully connected neural network and convolutional neural network. The results showed that, compared with the standard GAN and BiGAN, WGAN-GP provided the better performance in the experiment. Compared with the integration testing application, our framework was more effective in the unit test scheme. Besides, the convolution neural network did not show better prediction performance than the fully connected neural network.

Using GAN to generate test data has a technical problem that some branches are difficult to cover. It can be divided into the following three categories. The first is "==" conditional branch. It is difficult for GAN to generate a specific value, especially the type of program input is float. The second is the branch with complex conditional expression. When the gap between the target data and the training data is large, it is also difficult to cover it with the GAN method. In order to achieve full branch coverage, we need to do more work to tackle the above problem.

APPENDIX A
HYPER PARAMETERS

GAN

	Hidden Layers	Hidden Units	Activations Functions	Learning Rate	Batch norm	optimizer
Generater	2	64	LeakyReLU	0.001	TRUE	Adam
Discriminator	2	64	LeakyReLU	0.001	TRUE	Adam

CAN(conv)

	Hidden Layers	Activations Functions	Learning Rate	Batch norm	optimizer
Generater	Conv1DTranspose()	LeakyReLU	0.0001	TRUE	Adam
Discriminator	Conv1D()	LeakyReLU	0.00001	TRUE	Adam

biGAN

	Hidden Layers	Activations Functions	Learning Rate	Batch norm	optimizer
Generater	2	LeakyReLU	0.001	TRUE	Adam
encoder	2	LeakyReLU	0.001	TRUE	Adam
Discriminator	2	LeakyReLU	0.001	TRUE	Adam

biGAN(conv)

	Hidden Layers	Hidden Units	Activations Functions	Learning Rate	Batch norm	optimizer
Generater	Conv1DTranspose()	64	LeakyReLU	0.001	TRUE	Adam
encoder	Conv1D()	64	LeakyReLU	0.001	TRUE	Adam
Discriminator	Conv1D()	64	LeakyReLU	0.001	TRUE	Adam

wGAN-GP

	Hidden Layers	Hidden Units	Activations Functions	Learning Rate	Layer norm	optimizer	GP_WEIGHT
Generater	3	64	LeakyReLU	0.001	TRUE	RMSprop	10
Discriminator	3	64	LeakyReLU	0.001	TRUE	RMSprop	

wGAN-GP(conv)

	Hidden Layers	Activation Functions	Learning Rate	Layer norm	optimizer	GP_WEIGHT
Generater	Conv1DTranspose()	LeakyReLU	0.001	TRUE	RMSprop	10
Discriminator	Conv1D()	LeakyReLU	0.001	TRUE	RMSprop	

REFERENCES

[1] P. McMinn, "Search-based software testing: Past, present and future," in *Proc. IEEE 4th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar. 2011, pp. 153–163.

[2] G. Fraser and A. Arcuri, "1600 faults in 100 projects: Automatically finding faults while achieving high coverage with EvoSuite," *Empirical Softw. Eng.*, vol. 20, no. 3, pp. 611–639, Jun. 2015.

[3] Y. Zhan and J. A. Clark, "A search-based framework for automatic testing of *MATLAB*/*Simulink* models," *J. Syst. Softw.*, vol. 81, no. 2, pp. 262–285, Feb. 2008.

[4] M. Valueian, N. Attar, H. Haghighi, and M. Vahidi-Asl, "Constructing automated test Oracle for low observable software," *Scientia Iranica* vol. 27, no. 3, pp. 1333–1351, 2020.

[5] A. Singhal and A. Bansal, "Generation of test Oracles using neural network and decision tree model," in *Proc. 5th Int. Conf.-Confluence Next Gener. Inf. Technol. Summit (Confluence)*, Sep. 2014, pp. 313–318.

[6] S. R. Shahamiri, W. M. N. W. Kadir, S. Ibrahim, and S. Z. M. Hashim, "An automated framework for software test Oracle," *Inf. Softw. Technol.*, vol. 53, no. 7, pp. 774–788, Jul. 2011.

[7] S. R. Shahamiri, W. M. N. Wan-Kadir, S. Ibrahim, and S. Z. M. Hashim, "Artificial neural networks as multi-networks automated test oracle," *Automated Softw. Eng.*, vol. 19, no. 3, pp. 303–334, Sep. 2012.

[8] C. Lyu, S. Ji, Y. Li, J. Zhou, J. Chen, and J. Chen, "SmartSeed: Smart seed generation for efficient fuzzing," 2018, *arXiv:1807.02606*.

[9] Z. Li, H. Zhao, J. Shi, Y. Huang, and J. Xiong, "An intelligent fuzzing data generation method based on deep adversarial learning," *IEEE Access*, vol. 7, pp. 49327–49340, 2019.

[10] L. Joffe and D. J. Clark, "A generative neural network framework for automated software testing," 2020, *arXiv:2006.16335*.

[11] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, 2020, pp. 2255–2269.

[12] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "GANFuzz: A GAN-based industrial network protocol fuzzing framework," in *Proc. 15th ACM Int. Conf. Comput. Frontiers*, May 2018, pp. 138–145.

[13] W. Lv, J. Xiong, J. Shi, Y. Huang, and S. Qin, "A deep convolution generative adversarial networks based fuzzing framework for industry control protocols," *J. Intell. Manuf.*, vol. 32, no. 2, pp. 441–457, Feb. 2021.

[14] Y. Hong, U. Hwang, J. Yoo, and S. Yoon, "How generative adversarial networks and their variants work: An overview," *ACM Comput. Surveys*, vol. 52, no. 1, pp. 1–43, Jan. 2020.

[15] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surveys*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

[16] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *Proc. IEEE 8th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2015, pp. 1–12.

[17] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," in *Proc. Int. Conf. Neural Inf. Process. Syst. (NIPS)*, 2014, pp. 2672–2680.

[18] *Gcov*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[19] *GSL*. [Online]. Available: https://www.gnu.org/software/gsl/doc/html/intro.html

[20] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of Wasserstein GANs," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 5769–5779.

[21] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *Proc. Int. Conf. Mach. Learn.*, Jul. 2017, pp. 214–223.

[22] J. Donahue, P. Krähenbühl, and T. Darrell, "Adversarial feature learning," *CoRR*, vol. abs/1605.09782, pp. 1–18, May 2016.

XIUJING GUO (Member, IEEE) received the B.S.E. degree in engineering from Dalian Jiaotong University, Dalian, China, in 2018, and the M.S. degree in engineering from Hiroshima University, Higashihiroshima, Japan, where she is currently pursuing the D.Eng. degree in engineering. Her research interests include software testing, dependable computing, and machine learning.

HIROYUKI OKAMURA (Member, IEEE) received the B.S.E., M.S., and D.Eng. degrees in engineering from Hiroshima University, Higashihiroshima, Japan, in 1995, 1997, and 2001, respectively.

In 1998, he joined Hiroshima University as an Assistant Professor, where he has been an Associate Professor with the Department of Information Engineering, Graduate School of Engineering, since 2003. He has been a Processor with the Department of Information Engineering, Graduate School of Engineering, since 2018. His research interests include performance evaluation, dependable computing, and applied statistics.

Dr. Okamura is a member of the Association for Computing Machinery, the Operations Research Society of Japan, the Institute of Electrical, Information and Communication Engineers, the Information Processing Society of Japan, the Reliability Engineering Association of Japan, and the Institute of Electrical and Electronics Engineers.

TADASHI DOHI (Member, IEEE) received the B.S.E., M.S., and D.Eng. degrees in engineering from Hiroshima University, Higashihiroshima, Japan, in 1989, 1991, and 1995, respectively.

In 1992 and 2000, he was a Visiting Researcher with the Faculty of Commerce and Business Administration, The University of British Columbia, Canada, and the Hudson School of Engineering, Duke University, USA, respectively, on leave at Hiroshima University. Since 2002, he has been a Professor with Hiroshima University, where he has been appointed as the Vice Dean of the School of Informatics and Data Science, since 2018. His research interests include reliability engineering, software reliability, and dependable computing.

Dr. Dohi is a member of the Operations Research Society of Japan, the Institute of Electrical, Information and Communication Engineers, the Information Processing Society of Japan, and the Reliability Engineering Association of Japan. He is also a member of the Editorial Board of the IEEE TRANSACTIONS ON RELIABILITY.