F-006

Toward Automated Software Test Input Generation with Diffusion Models

Yujin Zhu　Xiujing Guo　Hiroyuki Okamura　Tadashi Dohi

1 INTRODUCTION

Software testing is a crucial process for ensuring the reliability and stability of software systems. Its primary objective is to identify and correct errors and defects in a program to enhance software quality. In modern software development, testing not only impacts product reliability and performance but also affects the development timeline and costs. Thus, software testing is a key component of software engineering and is worth thorough research.

We propose a new approach for generating test inputs and execution paths using Diffusion models.Diffusion models work through a process of iterative denoising, gradually transforming noise into meaningful data. This approach provides greater stability and can produce a broader range of test input, making it highly suitable for software testing applications. In our framework, diffusion models are used to generate test inputs that evaluate program behavior on programs using LLVM as an intermediate language.By training the diffusion model with execution path information, we can generate diverse test inputs that increase code coverage without requiring detailed analysis of branch expressions.

2 DENOISING DIFFUSION PROBABILISTIC MODEL

Denoising Diffusion Probabilistic Models (DDPMs) represent a class of generative models that have gained significant attention due to their ability to generate high-quality data with diverse distributions [1].The core concept of DDPMs is based on a diffusion process that gradually adds noise to data and a reverse process that progressively removes noise to recover the original data. This forward and reverse diffusion process is a key characteristic that distinguishes DDPMs from other generative models.

In the forward process, start with a sample x_0 from the real-world data distribution $q(x_0)$. The diffusion process is then defined through a series of steps where noise is incrementally added. Specifically, for each time step t in the set {1, 2, . . . , T}, the conditional distribution for the next state in the diffusion process, x_t, is given by

$$q(x_t \mid x_{t-1}) = \mathcal{N}\left(x_t; \sqrt{1-\beta_t} \cdot x_{t-1}, \beta_t \cdot I\right), \quad (1)$$

where x_t is the data at step t, β_t is the noise scale, and I is the identity matrix. The process starts with original data (x_0), then adds noise through T steps to create a distribution of noise (x_T).

In the reverse process, For each time step$t$ (from$T$down to 1), the conditional distribution for generating the previous state x_{t-1} from x_t, is defined as:

$$p(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_t(x_t, t), \Sigma_t(x_t, t)), \quad (2)$$

where $\mu_t(x_t, t)$ is the mean predicted by the model, and $\Sigma_t(x_t, t)$ is the variance indicating the uncertainty in the prediction.

Training DDPMs involves learning a model that can accurately predict the denoising step, essentially learning the distribution of the original data. This is achieved by minimizing the difference between the predicted denoising and the original data at each time step. The objective function typically used in DDPMs involves a variational lower bound (VLB), where the model is trained to maximize the likelihood of recovering the original data from the noisy data.

The loss function for each time step t is defined as:

$$\mathbb{E}_{t,\epsilon}[\|\epsilon - \epsilon_\theta(x_t, t)\|^2], \quad (3)$$

where ϵ is the noise in the forward process, and $\epsilon_\theta(x_t, t)$ is the predicted denoising result. The expected value is calculated by sampling across various time steps t and corresponding noise.

In our study, we focus on branch coverage as the metric for evaluating test inputs. And define the execution path as the combination of the execution state of each branch in the program under a specific input. To extract execution paths, we use the LLVM-cov tool [2], a source code coverage analysis and profiling tool that works with the LLVM compiler framework.

When executing a program, we use LLVM-cov to write branch frequencies to the output file, and then extract the branch information in the file as a path. For example, Fig. 1 is a C program designed to determine whether and what kind of triangle a three-sided triangle forms. When the input score is $3, 4, 5$, we compile the program with LLVM and use LLVM-cov to collect branch coverage information to get the "triangle.c.gcov" file, The resulting output file contains detailed branch execution data, indicating which branches were taken and which were not. This data can be converted into a simplified representation where: A branch marked as "taken" is represented by "1", indicating that the branch was executed at least once. A branch marked as "not taken" is represented by

```c
int Triangle_Classifier(float A , float B ,float C){

    bool isTriangle = (A + B > C) && (B + C > A) && (C + A > B);
    if (isTriangle) {
        printf("These sides form a triangle.\n");
        if (A == B && B == C) {
            printf("Triangle type: Equilateral\n");
        } else if (A == B || B == C || C == A) {
            printf("Triangle type: Isosceles\n");
        } else {
            printf("Triangle type: Scalene\n");
        }
    } else {
        printf("These sides do not form a triangle.\n");
    }


    return 0;
}
```

Figure 1: A code of `triangle.c`.

Table 1: The functions names.

Functions	Function name	Number of branches
F1	Triangle$_C lassifier$	18
F2	max$_m in$	26

“0”, indicating that the branch was not executed. A branch marked as “never executed” is represented by “-1”, indicating that it was entirely bypassed during program execution. Thus, “$-1, 0, 1$” is treated as the three execution states of the branch, and the execution path for the program under the input of $3, 4, 5$ is defined as a combination of branch states: $1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1$. This approach provides a straightforward and systematic way to determine branch coverage in LLVM-based programs.

3 EXPERIMENT

In this section, we conduct a series of experiments to evaluate the effectiveness of diffusion models.We have considered two Functions and the total number of branches is 18 and 26.The name of the selected functions and the number of branches it contains are shown in Table 1.

In both two experiments, We set up three different noise generation networks and use generated 30 initial training data to generate 30 test inputs.We conducted 6 experiments for each data set and took the average value.Table 2 shows the coverage of test inputs generated by DDPMs based method.Table 3 shows verage path accuracy generated by DDPMs based method.

For each branch in the path information, compare the predicted state with the actual state to evaluate the generated program path. If the predicted state matches the actual state, count it as a correct prediction. “accuracy” can be calculated as

$$accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ branches} \quad (4)$$

Table 2: The coverage achievement of ddpms

Functions	Network	Initial coverage	Trained coverage
F1	Network1	61.11%	62.22%
F1	Network2	61.11%	68.89%
F1	Network3	61.11%	61.11%
F2	Network1	76.92%	82.31%
F2	Network2	76.92%	74.62%
F2	Network3	76.92%	80.00%

Table 3: Average path accuracy

Functions	Network	Initial coverage	path accuracy
F1	Network1	61.11%	62.22%
F1	Network2	61.11%	68.89%
F1	Network3	61.11%	61.11%
F2	Network1	76.92%	70.08%
F2	Network2	76.92%	73.08%
F2	Network3	76.92%	72.81%

References

[1] Ho, J., Jain, A., Abbeel, P.Denoising diffusion probabilistic models. Advances in neural information processing systems, 33, 6840- 6851(2020).

[2] llvm-cov: https://llvm.org/docs/CommandGuide/llvm-cov.htm