# Towards High-Quality Test Suite Generation with ML-Based Boundary Value Analysis

Xiujing Guo*, Hiroyuki Okamura, and Tadashi Dohi
Graduate School of Advanced Science and Engineering, Hiroshima University, Hiroshima, Japan
guoxiujing6@gmail.com, okamu@hiroshima-u.ac.jp, dohi@hiroshima-u.ac.jp
*corresponding author

*Abstract*—In software testing, a protective measure to prevent faults in the code is to ensure that the behavior on the boundary between the sub-domains of the input space is correct. Therefore, designing test cases with boundary value analysis (BVA) can detect more errors and improve test efficiency. This paper presents an ML (machine learning) based approach to automatically generate boundary test cases. Our approach is twofold. First, we train an ML-based discriminator that determines whether a boundary exists between two test inputs. Second, using the outputs of the discriminator, we create test inputs based on Markov Chain Monte Carlo. We conduct experiments to compare the fault detection capabilities of the ML-based approach with concolic testing and manually-performed boundary analysis. Results indicate that the ML-based method outperforms the manually-performed boundary analysis in four of the seven programs tested and concolic testing in three of the seven programs tested.

*Keywords–software testing, random testing, boundary value analysis, Markov chain Monte Carlo, neural network*

## 1. Introduction

Software testing, which is crucial for verifying the reliability of software systems, is the execution of software systems to uncover faults. Static testing and dynamic testing are the two main divisions in software testing. Static testing entails looking into program codes and the documents that go with them without actually running the program. One of the typical methods for static testing is symbolic execution. On the other hand, dynamic testing, which is frequently used for software testing, confirms the correctness of a program by examining the dynamic behavior while the program is run with particular inputs. We concentrate on dynamic testing in our study. In dynamic testing, the creation of test cases is crucial for ensuring the quality of the product. The test case consists of test inputs and their expected outcomes by the test oracle. A common procedure of dynamic software testing is (i) to run the software under test (SUT) with the inputs of test cases, and (ii) to compare the test outcomes with the expected ones. The SUT has bugs if the test result differs from what was expected. Furthermore, from the perspective of software reliability, the software reliability may increase as the quantity of test cases increases. However, a large number of test cases cause a large amount of cost on software testing. Thus it is important to generate the high-quality test suite that has the high probability of bug finding with the small number of test cases.

Boundary value analysis (BVA) is one of the most popular methods to create the high-quality test cases effectively. The boundary value is defined as an input of software that changes the behavior of software with even a little change, and BVA extracts test inputs from the boundary values. In the context of black-box testing, BVA extracts the test inputs from the boundaries of equivalence partitions. In the BVA with white-box testing, the test inputs can be determined from branch and loop conditions on source codes. It is empirically known that many programming errors often occur on the boundary of the input domain. One protective measure to prevent vulnerabilities and failures in the code is to ensure correct behavior on the boundaries between the input space sub-domains [1]. Therefore, compared with other methods, by designing test cases with BVA, more boundary errors could be detected and the test efficiency is higher.

On the other hand, since BVA requires the analysis of specifications or source codes, the effort of BVA is not small. In the case of black-box testing, testers identify the equivalence partitions or sub-domains by analyzing the specification using partition analysis (PA), and create test inputs from the boundary between sub-domains [2]. This process generally relies on manual analysis. Since the complexity of the software system increases, software has large input spaces, heavily or non-linearly dependent inputs, and complex and highly structured inputs. Thus it is not feasible to manually analyze the input domain and the boundary values [1].

In recent years, there are several researches on the automation of BVA. Jeng and Forgacs [3] proposed a semi-automatic method that mixed the dynamic search method and the algebraic manipulations of the boundary conditions to generate test data for boundary value testing (BVT) more efficiently. Zhao et al. [4] considered string inputs and proposed a novel approach for automatically generating test points to better find problems at borders in code with string predicates. Ali S et al. [5] extended their search-based test data generation method in model-based testing, using a solver to automatically generate boundary values based on a set of heuristics. Feldt and Dobslaw [6] applied the idea of derivative in mathematical parlance to detect the maximum "change" area by combining the input and output distances, that is, the detection boundary. This method uses the program derivatives as a fitness function in search-based software testing for automated BVA. In order to generate boundary test cases through the above techniques, it is necessary to have the specification that clearly states the boundary formally.

This paper considers the BVA in white-box testing. The BVA in white-box testing focuses on a pair of input and its execution path, and the boundary is defined as the input that changes the execution path in some sense[1]. Compared to the BVA in black-box testing, one of the difficulties of BVA in white-box testing is to identify feasible test inputs. On BVA in black-box testing, we implicitly suppose that the input variables on specifications are independent, and then the boundary can be obtained from combinations of the boundary values for each input variables. On the other hand, input variables are dependent; for example, the program includes a condition

```
    x + y <= 10
```

for two input variables `x` and `y`. Then the boundary of this condition cannot be determined only by either of `x` and `y`. That is, even if the boundary is detected from source codes, we need to solve the problem of finding feasible input values. For this problem, Zhang et al. [7] used the SMT (satisfiability modulo theories) solver. However, SMT has weaknesses in scalability. In other words, it is difficult to apply the SMT-based approach to the large-sized programs.

In the paper, we propose another approach for the BVA in white-box testing, i.e., ML (machine learning) based approach. The idea behind our approach is twofold. First we train an ML-based discriminator that answers whether two test inputs have the same execution path or not. Second we generate test inputs based on Markov Chain Monte Carlo (MCMC) with the outputs of discriminator. Zhou et al. [8] first utilized MCMC methods for software random testing. MCMC-RT (MCMC Random Testing) uses the prior knowledge of software testing and is based on the statistical background of the probability of failure described by Bayesian inference. Our idea is an extension of MCMC-RT by introducing the ML-based discriminator. Our method needs the information on execution paths only, and does not require any program information identifying conditions.

The rest of this paper is organized as follows. Section II introduces the Markov chain Monte Carlo and describes the proposed method, illustrating how to use MCMC and ML models to automatically generate boundary test cases. Section III is the exploitation of ML model. Section IV presents experiments in a simple program and seven real programs. Finally, in Section V, we discuss the results of experiments and future works.

## 2. Generation of Boundary Values

### 2.1. Markov Chain Monte Carlo (MCMC)

MCMC is a general technique for efficiently generating samples drawn from a probability distribution with high-dimensional space. The idea of the MCMC is to simulate an ergodic Markov chain whose stationary distribution is consistent with a target distribution. The more steps of Markov chain simulation, the more closely the distribution of the sample matches the actual desired distribution.

It is significant to construct an appropriate Markov chain when using the MCMC method to generate samples. Different transfer construction methods will produce different MCMC methods. At present, the commonly used MCMC methods mainly include two Gibbs sampling [9] and Metropolis-Hastings (M-H) algorithm [10]. Since Gibbs sampling is a special case of the M–H algorithm. Here we only summarize the M-H algorithm.

The M-H algorithm produces a Markov chain whose limiting distribution is the target density $\pi(x)$. Let $x'$ be a candidate of the next state of Markov chain that is generated from a proposal distribution $Q(x'|x)$ where $x$ is the current state of Markov chain. This candidate becomes the next state of the Markov chain with the following acceptance probability:

$$P = \min\left(\frac{\pi(x')Q(x|x')}{\pi(x)Q(x'|x)}, 1\right) \tag{1}$$

In practice, we generate a uniform random number $U$, if $U$ is less than or equal to the acceptance probability $U \leq P$, the candidate is accepted, otherwise, the candidate is rejected. After repeating this process for several steps, the sample $x$ can be regarded as a sample drawn from $\pi(x)$.

### 2.2. MCMC for Boundary Values

Consider the MCMC for generating boundary values. As mentioned before, MCMC is essentially a method to generate samples from the target density $\pi(x)$. Our idea is to estimate the density function $\pi(x)$ on the input domain, which represents the likelihood that whether $x$ is a boundary value or not. When $\pi(x)$ is used as the target density of MCMC, the samples generated by MCMC are expected to be boundary values.

The key issue is how to estimate such a density function for boundary value. As an example, we consider a program that has only one input value $x$. If the program has one boundary, the density function for boundary value is expressed as a function with rapidly climbing and falling like in Fig. 1a. In the sense of mathematics, it is a delta function, and it is not easy to estimate such a function directly. On the other hand, Fig. 1b shows the cumulative distribution of Fig. 1a. Although it is a step function in the mathematical sense, it is possible to approximate such a function by a continuous function like a logistic function. If we obtain the cumulative distribution for boundary value $F(x)$, then the target density is approximated as

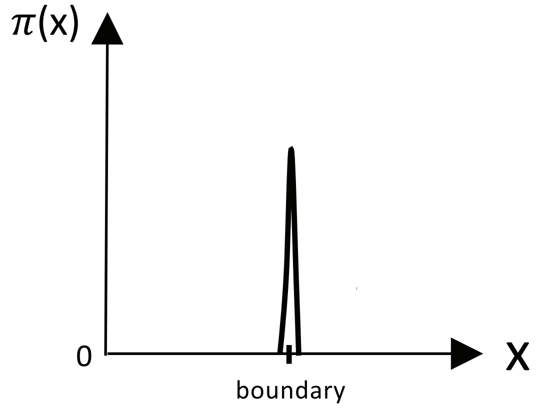
$$\pi(x) = \frac{F(x+h) - F(x)}{h}, \tag{2}$$

where $h$ is an arbitrary and sufficiently small value. Also, since $F(x)$ jumps at around the boundary, the value $F(x+h)-F(x)$ takes 1 when the test inputs $x+h$ and $x$ belong to different equivalence partitions. Otherwise, if $x+h$ and $x$ are in the same equivalence partition, $F(x+h) - F(x)$ becomes 0.

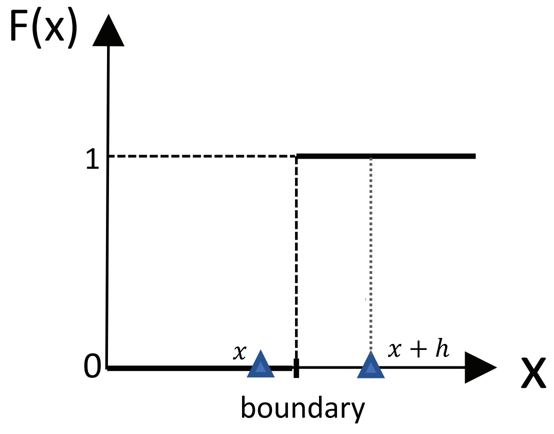
This idea is expanded to the case where high-dimensional test input space. Let $N(x, y)$ be the function meaning that $N(x, y) = 1$ if inputs $x$ and $y$ are in the different equivalence partitions. Otherwise, if inputs $x$ and $y$ are in the same equivalence partion, $N(x, y) = 0$. Then the target density is given by

$$\pi(x) = \frac{1}{m}\sum_{i=1}^{m} \frac{N(x+h_i, x)}{h_i}, \tag{3}$$

[1]The definition is formally given in Section III.



(a)



(b)

Figure 1: The probability density and the cumulative distribution of boundary value.

where $h_i$ is a small vector to put a perturbation. Figure 2 illustrates our approach in two-dimensional space. In the figure, there are two inputs $a$ and $b$ and one boundary. The circles represent the radius $R$ of perturbation. Since $X'$ is closer to the boundary than $X$, the likelihood that $X'$ and $X' + h_i$ belong to the different equivalence partitions is higher than $X$. That is, $\pi(X')$ may be greater than $\pi(X)$.

Based on the density function for boundary value $\pi(x)$, the

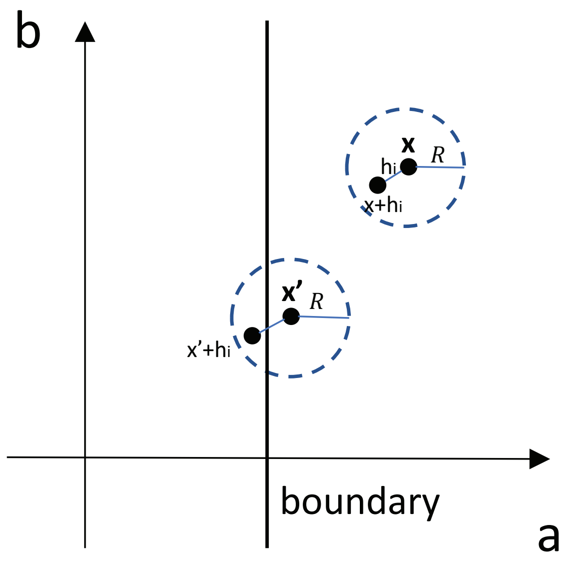


Figure 2: An example in two-dimensional space.

acceptance probability in the M-H algorithm becomes

$$P = \min\left(\frac{\sum_{i=1}^{m}\frac{N(x'+h_i,x')}{h_i}Q(x|x')}{\sum_{i=1}^{m}\frac{N(x+h_i,x)}{h_i}Q(x'|x)}, 1\right) \tag{4}$$

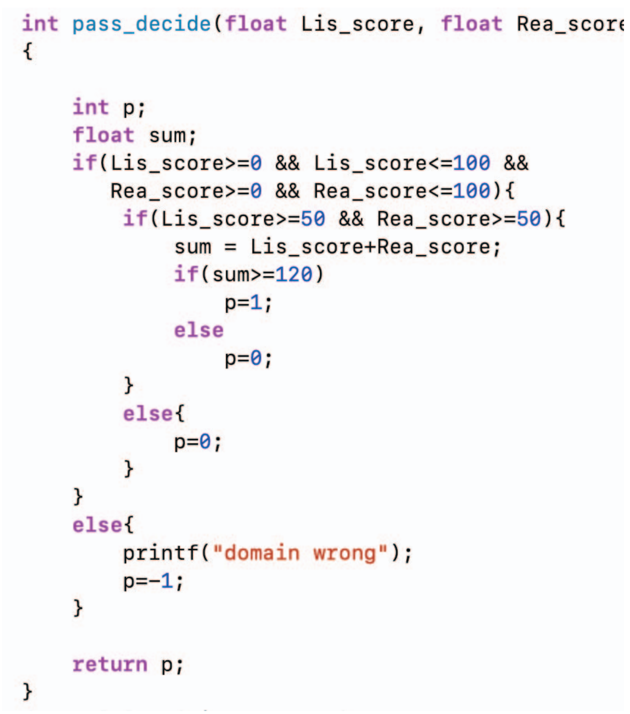
One of the purposes of generating boundary values is to generate test cases for software testing. In this sense, it is better that the generated boundary values cover the input domain of the software. In the M-H algorithm, the proposal distribution $Q(x'|x)$ is frequently designed by searching inputs close to the original (current) input $x$, i.e., the local search. However, the local search cannot ensure the coverage of test domain, and thus the paper considers the independent proposal distribution that does not depend on the original (current) input $Q(x')$. The typical example of such proposal distribution is the uniform distribution on input domain. When we use the uniform distribution on input domain, the acceptance probability simply becomes

$$P = \min\left(\frac{\sum_{i=1}^{m}\frac{N(x'+h_i,x')}{h_i}}{\sum_{i=1}^{m}\frac{N(x+h_i,x)}{h_i}}, 1\right). \tag{5}$$

## 3. Exploitation of ML Model

### 3.1. Model Architecture

To generate boundary values, we need the function $N(x, y)$ that outputs whether two inputs $x$ and $y$ belong to the different equivalence partitions. The simplest and direct approach is to monitor concrete paths by executing software with two inputs. However, since we need a number of executions of $N(x, y)$ in the scheme of MCMC, the direct approach is not appropriate

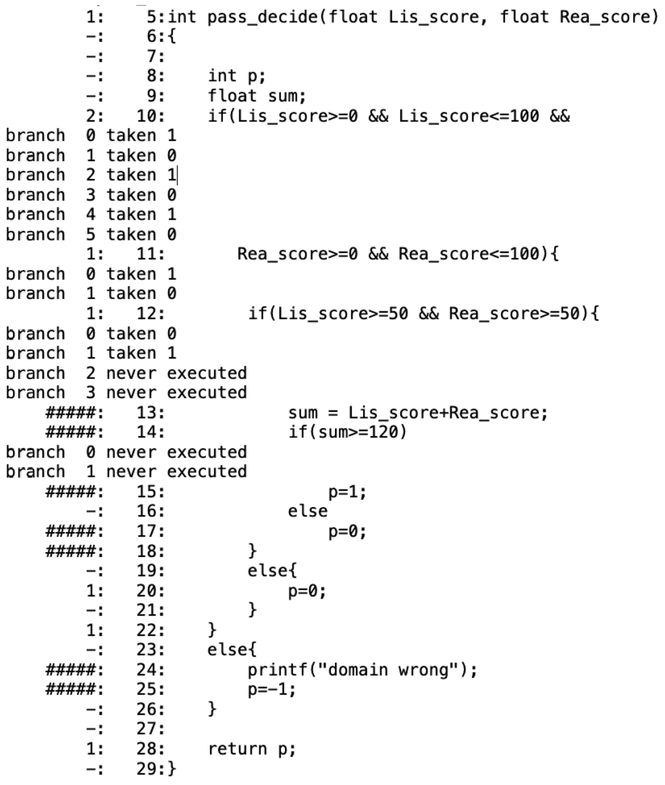


```c
int pass_decide(float Lis_score, float Rea_scor
{

    int p;
    float sum;
    if(Lis_score>=0 && Lis_score<=100 &&
       Rea_score>=0 && Rea_score<=100){
        if(Lis_score>=50 && Rea_score>=50){
            sum = Lis_score+Rea_score;
            if(sum>=120)
                p=1;
            else
                p=0;
        }
        else{
            p=0;
        }
    }
    else{
        printf("domain wrong");
        p=-1;
    }

    return p;
}
```

Figure 3: The source code of `En_testing.c`.



```
        1:    5:int pass_decide(float Lis_score, float Rea_score)
        -:    6:{
        -:    7:
        -:    8:    int p;
        -:    9:    float sum;
        2:   10:    if(Lis_score>=0 && Lis_score<=100 &&
branch  0 taken 1
branch  1 taken 0
branch  2 taken 1
branch  3 taken 0
branch  4 taken 1
branch  5 taken 0
        1:   11:       Rea_score>=0 && Rea_score<=100){
branch  0 taken 1
branch  1 taken 0
        1:   12:        if(Lis_score>=50 && Rea_score>=50){
branch  0 taken 0
branch  1 taken 1
branch  2 never executed
branch  3 never executed
    #####:   13:            sum = Lis_score+Rea_score;
    #####:   14:            if(sum>=120)
branch  0 never executed
branch  1 never executed
    #####:   15:                p=1;
        -:   16:            else
    #####:   17:                p=0;
    #####:   18:        }
        -:   19:        else{
        1:   20:            p=0;
        -:   21:        }
        1:   22:    }
        -:   23:    else{
    #####:   24:        printf("domain wrong");
    #####:   25:        p=-1;
        -:   26:    }
        -:   27:
        1:   28:    return p;
        -:   29:}
```

Figure 4: The result of Gcov for `En_testing.c.gcov`.

in this case. The second way is to create the function $N(x, y)$ with the static analysis such as symbolic execution beforehand. This method may be effective but the static analysis has a limitation on scalability. In the paper, we exploit a machine learning (ML) model to create the function $N(x, y)$. The ML-based approach is one of the data-driven approaches. The ML model is trained from the data, and the model mimics the trained data and interpolates unknown two inputs predicatively. Although the training of model requires much computational cost, the evaluation is done with less computation cost. This property is appropriate for the function in MCMC scheme.

In the paper, we use a neural network (NN) to represent the function $N(x, y)$. NNs are multilayer perceptrons, including an input layer or multiple hidden layers and an output layer. All these units are connected to each other through links with synaptic weights. These weights are updated as part of the training process and reflect the information learned during the training process. In our method, the input of the NNs is a set of input pairs, such as the vector $(x, x+h)$ and the output of the NNs is a label as a binary value indicating whether $x$ and $y$ are in the same partition or not.

## 3.2. Training Data

Before using NNs prediction, we first need to train the NNs with a set of training data. Since we focus on the white-box boundary value where equivalence partitions are defined by execution paths of program for test inputs, it is necessary to define the equivalence of execution path based on the execution log. In addition, NN requires a number of training data, and thus the training data collection should be automatically executed. In the paper, we provide the approach based on Gcov tool.

In white-box boundary value testing, we want to cover the boundary values for comparison predicates. Each atomic Boolean expression in the path condition is referred to here as a predicate. Predicates could be Boolean variables, comparison predicates ($>, >=, <, <=, =, \neq$), etc., and should not contain any Boolean operator (such as $\wedge, \vee, \neg$, etc.) [7]. Each comparison predicate contains two branches, and each branch has three states, marked as "`1, 0, -1`". Suppose we have a comparison predicate $(a > 0)$, contains two branches: $(a > 0)$ and $(a <= 0)$. For the branch in the program condition, when the branch is executed and is taken (satisfied) one or more times (if the condition is nested in a loop, or the function where the condition is located is called multiple times, the branch in the conditional expression may be taken multiple times), We mark the state of this branch as "`1`", when the branch is executed but is not taken, we mark the state of this branch as "`0`", and if the branch is not executed, mark the state of the branch as "`-1`". For conditional branches in a program loop, no matter how many times the loop is repeated, we mark the state of the branch that triggered the loop as "`1`". That is to say, when the branch is taken multiple times, the actual execution path is different according to the different taken times. But in order to facilitate the learning of the neural network, we ignore the number of times the branch is taken, as long as the branch is taken, its state is "`1`".

Regarding the method of obtaining the initial training data labels, we consider annotating source code to add instrumentation and use the Gcov tool to extract the branch execution information. Gcov [11] is a source code coverage analysis tool that cooperates with GCC to implement statement coverage and branch coverage testing of C/C++ files. Gcov annotates the source code to add instrumentation and accurately counts the number of executions of each statement in the program. In our work, we use Gcov to write the branch execution frequency to the output file when testing the program. The branch information in the file is then extracted as the execution path.

For example, Figure 3 is a simple C program for judging whether the English examination passed. We assume that the English examination consists of two parts; listening and reading. The full score of each part is 100 points. To pass the English examination, the following two conditions must be satisfied:

(1) Both the listening score $a$ and the reading score $b$ are greater than or equal to 50 points.
(2) The sum of listening score $a$ and reading score $b$ is greater than or equal to 120 points.

Figure 4 shows the result of Gcov on running the `En_testing.c.gcov` program with the input $a = 45$ and $b = 65$. 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, and the execution path of the program `En_testing.c` with the input $a = 45$ and $b = 65$ is the combination of all branch executions; `1,0,1,0,1,0,1,0,0,1,-1,-1,-1,-1`. Then we obtain the label of input pairs by calculating whether the execution paths corresponding to the two inputs are equal. If they are equal, the label is 0 (there is no boundary between the two inputs), and if they are not equal, the label is 1 (there is a boundary between the two inputs). In this paper, we apply the ML-based approach to the c language. For other languages, we can apply this method by simply changing the way paths are extracted.

## 4. Experiment

In this section, we present experiments to investigate the effectiveness of ML-based approach. We conducted two sets of experiments. One is to generate test cases for a simple C program and compare the effects of various parameter combinations. Another one is an experiment on several real programs.

### 4.1. Fault detection ability

We use mutation testing to study the fault detection rate of test sets. Mutation testing is a type of software testing in which certain statements of the source code are changed/mutated to check if the test cases are able to find errors in the source code. In the experiment, we injected (seeded) several faults into the program. Each seeded fault yields a faulty version. For each test set generated in the experiments, we run the whole test set on each faulty version and count the number of killed mutations. The kill rate is calculated by Eq. (6).

$$kill\_rate = \frac{the\ number\ of\ killed\ mutations}{total\ number\ of\ mutations} \tag{6}$$

We manually inject (seeded) 6 kinds of faults in the program under test. Off-by-one bugs (OBOB) are a kind of faults when some computation process uses some wrong value which is 1 more or 1 less than the correct value, and most boundary faults are Off-by-one bugs [7]. The rest of faults contain five common mutation operators [12]: constant replacement (CR), relational operator replacement (ROR), arithmetic operator replacement (AOR), scalar variable replacement (SVR), and logical operator replacement (LOR). We use the execution path and the output to determine if a fault is killed, that is, when at least one test case has an execution path different from the correct version, or at least one test case has an output different from the correct version, the mutation is killed.

### 4.2. RT and ART

In the RT approach, we randomly generate a test set consisting of n test cases and execute each mutated program with this test set to study the fault detection ability.

ART is executed by the algorithm described in [13]. In traditional ART algorithm, the executed set is incrementally updated with the selected element from the candidate set until a failure is revealed. However, to make the experimental results of ART and MLBVA comparable, we changed the experimental stopping condition of ART to generate n test cases incrementally. For each mutation, the test case generation process stops if the injected fault is detected when generating the $i$-th ($i \leq n$) test case, and the generated test set kills the mutation. If none of the generated n test cases detect the fault, the generated test set did not kill the mutation.

### 4.3. Experiment with a Simple C Program

In this experiment, we use a program `En_testing.c` described in the previous section. Figure 5 is a conceptual diagram of the boundaries. We injected (seeded) 25 faults into the program. Among them, 14 faulty versions contain off-by-one bugs (where we add/subtract 1 to the right-hand side of comparison predicates), and the rest of the faults contain five common mutation operators.

**Design Parameters of NNs and MCMC:** The NN in our experiments is a fully connected NN (dense layers) with one input layer, two hidden layers, and one output layer. Each hidden layer has 64 units and the activation function is 'relu'. The input size of the input layer is 4. The output

TABLE I: Kill rate of RT, ART and ML-based boundary value analysis (MLBVA).

| method | n=5 | n=10 | n=20 | n=50 |
|---|---|---|---|---|
| RT | 0.28 | 0.36 | 0.36 | 0.4 |
| ART | 0.28 | 0.28 | 0.4 | 0.44 |
| MLBVA (Dataset20000) | 0.36 | 0.36 | 0.56 | 0.68 |
| MLBVA (Dataset10000) | 0.28 | 0.4 | 0.52 | 0.64 |
| MLBVA (Dataset5000) | 0.4 | 0.4 | 0.48 | 0.64 |

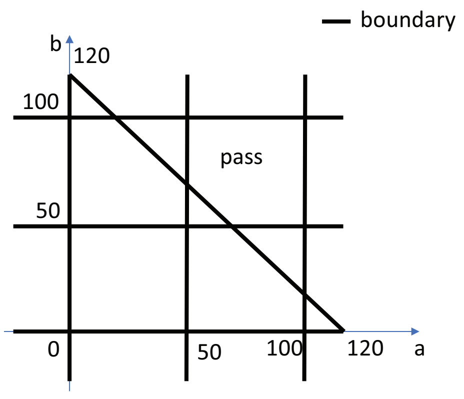


Figure 5: Conceptual diagram of the boundaries.

layer has an output size of 1 and the activation function is 'sigmoid'. In addition, the parameter learning rate and the number of training epochs (one epoch is the cycle when an entire dataset is passed forward and backward through the neural network only once.) are set to 0.01 and 50, respectively. In our experiments, we use three sets of initial training data, which are randomly generated from the input domain. The size of the dataset is 20000, 10000, and 5000, respectively, which are marked as Dataset20000, Dataset10000, and Dataset5000. Let $f(x) = \sum_{i=1}^{m} N(x+h_i, x)$, When both $f(x')$ and $f(x)$ are calculated as 0, we cannot judge which of the current sample $x$ and the candidate sample $x'$ is closer to the boundary. So in the experiment, we directly use the probability value output by the neural network as the value of f(x).

For MCMC, we use the uniform distribution as the proposal distribution. We use MCMC model to generate $n$ test cases from input domain and examine the cases $n = 5$, $n = 10$, $n = 20$, and $n = 50$. Selecting a point close to the boundary as the initial value of MCMC is more conducive to generating the boundary value. Therefore, we choose $(49, 49)$ as the initial value of MCMC. In our experiments, MCMC samples at each step and stops until $n$ test cases are generated. And the NNs will be retrained for every $10$ steps.

**Results and Discussion:** We use RT, ART, and ML-based approach to generate test cases for a simple program under test. Figure 6 shows the data distribution generated by RT. Figure 7 shows the data distributions generated by ML-based approach under parameters {n=5, n=10, n=20, n=50} with Dataset20000, Dataset10000, and Dataset5000, respectively. It can be seen intuitively from the distribution graph that the ML-based approach can generate test cases near the boundary. And to cover all boundaries, we need more test cases, such as n=50. We also record the number of faults detected by each test set. Table I shows the kill rates corresponding to the test sets generated by various methods. The results show that the kill rate of most test sets generated by ML-based approach is better than that of RT and ART. Compared with RT and ART, our proposed ML-based approach can generate better quality test cases and detect more faults.

## 4.4. Experiment with Real Programs

We select seven programs used in the existing literature to evaluate the effect of our approach on real programs. Machine learning can handle both numerical data and categorical data. During the data processing phase, machine learning models convert structured data into numerical data because the input layer of the neural network only accepts numerical input data. Therefore, in this experiment, we select seven programs with inputs of only numeric types, containing continuous data and discrete data. The descriptions of these subject programs are shown in Table II. And the details about all seven programs are shown in Table III, such as the dimensional number of program inputs, the range of input domain, line of code (LOC), fault information, and the number of boundary values.

**Concolic testing:** Concolic testing is a hybrid software verification technique that performs symbolic execution along a concrete execution path. Symbolic execution is a software testing technique that substitutes symbolic values for normal inputs to a program during program execution. By symbolizing the program inputs, the symbolic execution maintains a set of constraints for each execution path. After the execution, constraint solvers will be used to solve the constraint and determine what inputs cause the execution. Its purpose is to maximize code coverage. KLEE is a dynamic symbolic execution tool built on the LLVM compilation framework that automatically generates test cases and achieves high program coverage [14]. In this experiment, we use KLEE to generate

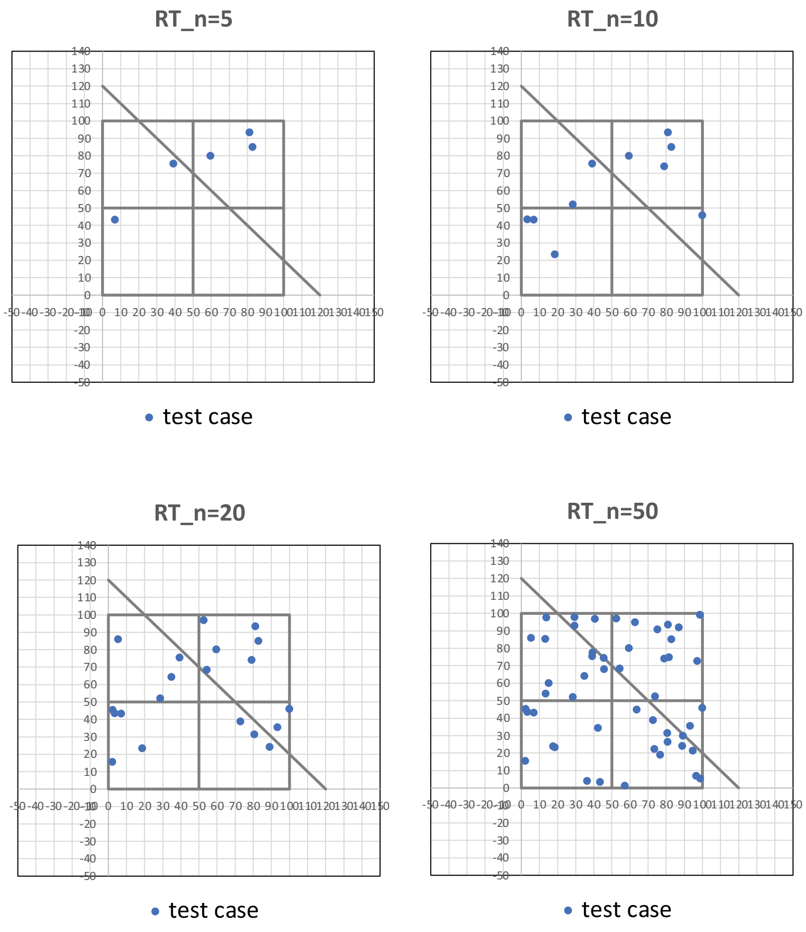


Figure 6: The data distribution generated by RT.

test cases for seven programs and compare the kill rate with our proposed ML-based method.

**Manually-Performed Boundary Value Analysis:**

In this experiment, we asked a student to generate a set of boundary values by manually analyzing the source code for comparing the fault detection ability with the ML-based approach.

In this work, the student uses an input that is on the edge of a given predicate as a boundary value. Suppose there is a path condition $(a > 0) \vee (b <= 0)$, including two boundaries $a = 0, b = 0$, therefore, the input (a,b)=(0,0), input (a,b)=(0,1) and input (a,b)=(1,0) can be used as three boundary values. In our experiment, the student selects the endpoints of all boundary lines in the input domain and the intersection points between the boundary lines as boundary values. We will compare the fault detection ability of boundary values obtained by manually analyzing the source code with the fault detection ability of ML-based approach that do not manually analyze the source code. The number of selected boundary values (num_Bvalues) for 7 programs are shown in Table III.

**Results and Discussion:** We examine the fault detection capabilities of RT, ART, Manually-performed boundary value analysis approach, concolic testing and ML-based approach, respectively. Because our methods show better performance when the number of initial datasets is n=50 in the experiment with a simple program. In the real program experiment, the MCMC generates $n = 50$ test cases from the input domain.

Table IV shows the comparison of kill rate with RT, ART, Manually-performed boundary value analysis approach (BVA), Concolic testing (KLEE), and ML-based approach (MLBVA). KLEE generates n test cases for each of the seven programs. Without analyzing the source code, our proposed ML-based

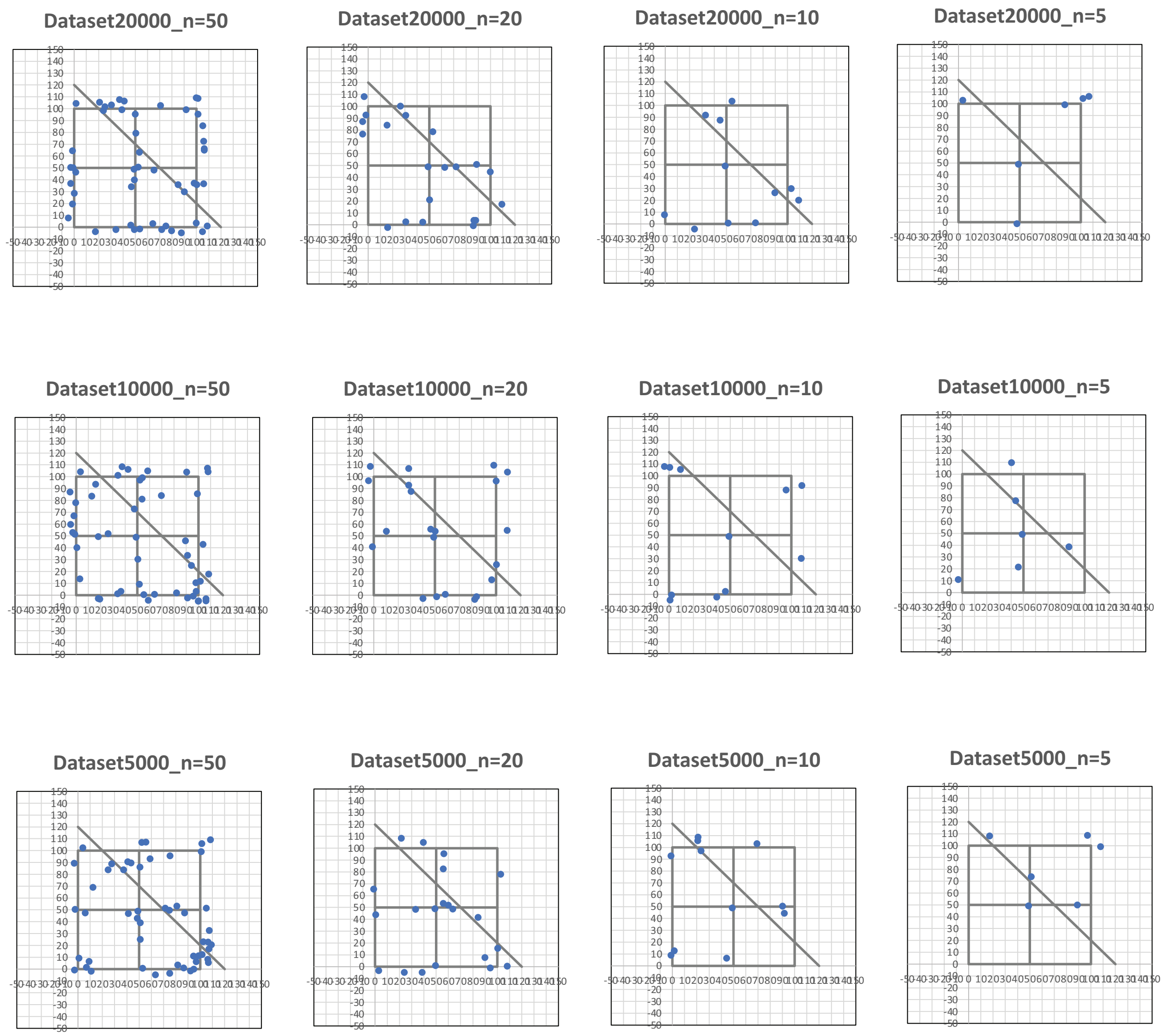


Figure 7: The data distribution generated by ML-based approach with Dataset.

TABLE II: Experimental programs

| Prog Name | Description |
|---|---|
| triType [15] | The type of a triangle |
| nextDate [16] | Calculate the following date of the given day |
| findMiddle [17] | Find the middle number among three numbers |
| bessj [18] | Bessel function J of general integer order |
| expint [18] | Exponential integral |
| plgndr [18] | Legendre polynomials |
| tcas [19] | Aircraft collision avoidance system |

TABLE III: Details of 7 subject programs

| Program | Dim | Input Domain | | Size(LOC) | Fault types | | | | | | Total Faults | num_Bvalues |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | From | To | | OBOB | CR | ROR | AOR | SVR | LOR | | |
| triType | 3 | (0, 0, 0) | (100, 100, 100) | 41 | 2 | | 7 | 2 | 1 | 6 | 18 | 15 |
| nextDate | 3 | (1800, -2, -2) | (3000, 14, 33) | 90 | 16 | | 7 | | | 8 | 31 | 42 |
| findMiddle | 3 | (-100, -100, -100) | (100, 100, 100) | 36 | | | 14 | | | 5 | 19 | 15 |
| bessj | 2 | (2,-1000) | (300, 15000) | 133 | 10 | 1 | 11 | 1 | 4 | | 27 | 21 |
| expint | 2 | (-10 , -10) | (1500, 1500) | 109 | 12 | 2 | 2 | 5 | 3 | 10 | 34 | 6 |
| plgndr | 3 | (-5,-5,-5) | (5,5,5) | 65 | 6 | | 6 | 2 | 3 | 2 | 19 | 12 |
| tcas | 12 | (0,0,0,0,0,0, 0,0,0,0,0,0) | (1000,1,1,50000,1000,50000, 3,1000,1000,2,2,1) | 182 | 17 | 4 | 2 | | 5 | 6 | 34 | 19 |

TABLE IV: Comparison of kill rate with RT, ART, Manually-performed boundary value analysis approach (BVA), Concolic testing (KLEE), and ML-based approach (MLBVA)

| Method | Kill rate | | | | | | |
|---|---|---|---|---|---|---|---|
| | triType | nextDate | findMiddle | bessj | expint | plgndr | tcas |
| RT | 0.61 | 0.58 | 0.36 | 0.52 | 0.47 | 0.63 | 0.29 |
| ART | 0.61 | 0.51 | 0.63 | 0.66 | 0.47 | 0.58 | 0.05 |
| BVA | 0.88 | 0.45 | **1** | 0.7 | 0.7 | 0.78 | 0.06 |
| MLBVA (Dataset20000) | 0.72 | 0.65 | **1** | 0.81 | 0.68 | **0.95** | 0.05 |
| MLBVA (Dataset10000) | 0.72 | 0.7 | 0.79 | **0.85** | **0.79** | **0.95** | 0.05 |
| MLBVA (Dataset5000) | 0.66 | 0.68 | 0.89 | 0.59 | 0.74 | 0.84 | 0.05 |
| KLEE | **1 (n=14)** | **0.9 (n=56)** | **1 (n=13)** | 0.37 (n=2) | 0.38 (n=4) | 0.84 (n=14221) | **1 (n=1290)** |

approach outperforms the Manually-performed boundary value analysis approach in four of the seven programs tested. And ML-based approach has better fault detection ability than RT and ART in program tests except for `tcas` program. Meanwhile, compared with the concolic testing method, the kill rate of test cases generated by the ML-based method outperforms concolic testing among the three programs.

Whether the ML-based approach can generate high-quality test cases near the boundary highly depends on the accuracy of the neural network's prediction of whether the sample is near the boundary. Table V shows the prediction accuracy of the neural network in the one-step sampling process of MCMC. Overall, when the neural network has higher prediction accuracy, the fault detection ability of the generated test cases will be stronger. In order to use neural networks to generate higher-quality test cases, we need to consider improving the performance of neural networks in the future, so that neural networks can better learn the boundary information of programs.

Table VI shows the time consumed by the experiment. The time cost of each method includes test case generation time and mutation testing time. Dataset10000 and Dataset20000

TABLE V: Prediction accuracy

| Initial_trainData | NN prediction accuracy | | | | | | |
|---|---|---|---|---|---|---|---|
| | triType | nextDate | findMiddle | bessj | expint | plgndr | tcas |
| Dataset20000 | 0.53 | 0.54 | 0.84 | 0.75 | 0.48 | 0.84 | 0.64 |
| Dataset10000 | 0.54 | 0.65 | 0.78 | 0.61 | 0.44 | 0.81 | 0.69 |
| Dataset5000 | 0.46 | 0.64 | 0.37 | 0.33 | 0.39 | 0.77 | 0.78 |

TABLE VI: Time cost

| method | Time (sec) | | | | | | |
|---|---|---|---|---|---|---|---|
| | triType | nextDate | findMiddle | bessj | expint | plgndr | tcas |
| RT | 249 | 452 | 287 | 384 | 514 | 259 | 449 |
| ART | 248 | 572 | 193 | 320 | 190 | 523 | 825 |
| MLBVA (Dataset20000) | 11605 | 11762 | 11950 | 11970 | 11828 | 11330 | 11679 |
| MLBVA (Dataset10000) | 6060 | 6292 | 6273 | 6221 | 6291 | 5871 | 6153 |
| MLBVA (Dataset5000) | 3319 | 3400 | 3242 | 3403 | 3464 | 3169 | 3392 |
| KLEE | 69 (n=14) | 470 (n=56) | 68 (n=13) | 21 (n=2) | 43 (n=4) | 76244 (n=14221) | 11650 (n=1290) |

have similar kill rates but with less time cost for Dataset10000. In the MLBVA method, the test generation time includes training data preparation time, neural network training time, and MCMC computation time. The most time-consuming of these is the preparation time of the training data. For example, to prepare $20,000$ training data for the `triType` program, $20,000$ executions of the program are needed to extract execution path information to generate labels, with a time consumption of $11,000$ seconds. This makes the MLBVA method more time-consuming than other methods. Therefore in order to reduce the time cost and improve the prediction accuracy of the neural network, we will consider using some structural coverage criteria in future research to help cover more parts of the code and thus obtain higher-quality training data instead of using random test case generation to generate training data.
Besides, there are many equal-conditional expressions in the programs such as `tcas`, and our currently proposed method is not good at generating exact data because the probability of generating exact data is low. We will also consider addressing this issue in future work.

## 5. Conclusion

In this paper, we proposed an ML (machine learning) based approach to automatically generate test cases with Boundary Value Analysis. First, we train an ML-based discriminator that determines whether two test inputs have the same execution path or not. Second, we create test inputs based on Markov Chain Monte Carlo (MCMC) using the discriminator's outputs.

We conducted a set of experiments on a simple program and seven real programs to exhibit the performance of ML-based approach. Our results showed that the ML-based approach could generate test cases close to the boundary for testing and has better fault detection ability than RT and ART. Besides, the ML-based method outperforms the manually-performed boundary analysis in four of the seven real programs tested, and outperforms the concolic testing in three of the seven real programs tested.

The accuracy of neural network predictions largely affects the performance of the ML-based approach. In order to use neural networks to generate higher-quality test cases, we need to consider improving the performance of neural networks in the future, so that neural networks can better learn the boundary information of programs. At the same time, high-quality training data will also improve the prediction accuracy of the neural network, and can greatly reduce the time cost, so we will also improve the training data generation strategy in future work. Besides, there are many equal-conditional expressions in the programs, and our currently proposed method is not good at generating exact data. To improve the fault detection ability of our proposed method, this problem needs to be solved in our future work. We will also consider applying our method to complex software systems with large input spaces and complex and highly structured inputs.

## References

[1] Dobslaw F, de Oliveira Neto F G, Feldt R. Boundary Value Exploration for Software Analysis[C]//2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2020: 346-353.

[2] Reid S C. An empirical analysis of equivalence partitioning, boundary value analysis and random testing[C]//Proceedings Fourth International Software Metrics Symposium. IEEE, 1997: 64-73.

[3] Jeng B, Forgács I. An automatic approach of domain test data generation[J]. Journal of Systems and Software, 1999, 49(1): 97-112.

[4] Zhao R, Lyu M R, Min Y. Automatic string test data generation for detecting domain errors[J]. Software Testing, Verification and Reliability, 2010, 20(3): 209-236.

[5] Ali S, Yue T, Qiu X, et al. Generating boundary values from OCL constraints using constraints rewriting and search algorithms[C]//2016 IEEE Congress on Evolutionary Computation (CEC). IEEE, 2016: 379-386.

[6] Feldt R, Dobslaw F. Towards automated boundary value testing with program derivatives and search[C]//International Symposium on Search Based Software Engineering. Springer, Cham, 2019: 155-163.

[7] Zhang Z, Wu T, Zhang J. Boundary value analysis in automatic white-box test generation[C]//2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2015: 239-249.

[8] Zhou B, Okamura H, Dohi T. Enhancing performance of random testing through Markov chain Monte Carlo methods[J]. IEEE Transactions on Computers, 2011, 62(1): 186-192.

[9] Brooks S. Markov chain Monte Carlo method and its application[J]. Journal of the royal statistical society: series D (the Statistician), 1998, 47(1): 69-100.

[10] Chib S, Greenberg E. Understanding the metropolis-hastings algorithm[J]. The american statistician, 1995, 49(4): 327-335.

[11] Gcov: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[12] Jia Y, Harman M. An analysis and survey of the development of mutation testing[J]. IEEE transactions on software engineering, 2010, 37(5): 649-678.

[13] Chen T Y, Leung H, Mak I K. Adaptive random testing[C]//Annual Asian Computing Science Conference. Springer, Berlin, Heidelberg, 2004: 320-329.

[14] Cadar C, Dunbar D, Engler D R. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs[C]//OSDI. 2008, 8: 209-224.

[15] Williams N, Marre B, Mouy P, et al. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis[C]//European Dependable Computing Conference. Springer, Berlin, Heidelberg, 2005: 281-292.

[16] Awedikian Z, Ayari K, Antoniol G. MC/DC automatic test input data generation[C]//Proceedings of the 11th Annual conference on Genetic and evolutionary computation. 2009: 1657-1664.

[17] Ghani K. Searching for test data[J]. Ph. D Thesis, 2009.

[18] Teukolsky S A, Flannery B P, Press W H, et al. Numerical recipes in C[J]. SMR, 1992, 693(1): 59-70.

[19] Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact[J]. Empirical Software Engineering, 2005, 10(4): 405-435.