Generate Test Data from C/C++ Source Code using Weighted CFG and Boundary Values

Tran Nguyen Huong*†, Do Minh Kha†‡, Hoang-Viet Tran†, Pham Ngoc Hung†
*National College for Education, Hanoi, Vietnam
†VNU University of Engineering and Technology, Hanoi, Vietnam
Email: {17028005, 17020827, 15028003, hungpn}@vnu.edu.vn
‡Corresponding author

***Abstract*—This paper presents two test data automatic generation methods which are based on weighted control flow graph (named WCFT) and boundary values of input parameters (named BVTG). Firstly, WCFT method generates a CFG from a given unit function, updates weight for it, then generates test data from the greatest weight test paths. In the meantime, WCFT can find dead code that can be used for automatic source code errors fix. Secondly, BVTG method generates test data from boundary values of input parameters of the given unit function. The combination of the two generated test data sets from these two methods will improve the error detection ability while maintaining a high code coverage. An implemented tool (named WCFT4Cpp) and experimental results are also presented to show the effectiveness of the two proposed methods in both time required to generate test data and error detection ability.**

***Index Terms*—Unit testing, test data generation, static testing, concolic testing, bounded testing, SMT-Solver.**

I. Introduction

Nowadays, generating test data automatically from source code has been an important subject in both software engineering and industry. Most of methods and articles which have been published to address this problem use the symbolic execution method [9] to generate test data by solving the constraint expressions based on an SMT-Solver. In those methods, source code is analyzed to generate the control flow graph (CFG) which will be used to generate test data. There are two main approaches in generating test data from a given CFG: dynamic and static. Dynamic test data generation is a process which generates test data bases on the combination of a source code analyzer and a test driver [2], [3], [6], [14]. Dynamic testing contains two main methods: Execution Generated Testing (EGT) and Concolic Testing. EGT method is applied in such automatic test data generation tool as KLEE [3] - a well known tool for its effectiveness. The key idea of EGT is to generate test data directly when running the program. This method proves its effectiveness when finding hidden errors because EGT checks every possibilities which may happen. One of EGT's disadvantages is its low performance when the function under testing has loops with large loop number or has recursive calls. The second method is *concolic testing*. The initial idea of concolic testing was mentioned in DART [6] and was officially introduced in CUTE [12]. Later, this method has been continuously improved in such tools as PathCrawler [14], CUTE [12], CAUT [13], and CREST [2] which have been being used a lot in real software projects in practice. The key idea of this method is to generate next test data from its previous ones. This method is known for its high coverage and effective error detection ability. However, its disadvantage is when the program under testing has non deterministic behaviors, imprecise symbolic representations, incomplete theorem proving, etc. To our knowledge, dynamic test data generation methods are time consuming, especially with a great amount of test data, because the method must continuously execute the program under testing for each test data. In the meantime, static test data generation is a process which generates test data based on the given program structure. This method is not only less time consuming in comparison with the dynamic one but also can generate a smaller number of test data with higher coverage. However, statically generating test data faces the difficulty of big source code and complex data structure. Current researches have focused on improving the source code analysis to fully support the syntax of the programming language, finding test paths, optimizing the constraint expressions, and selecting suitable SMT-Solver, etc., to generate test data for complex unit functions. Among these researches, the most effective ones are researches which are based on test paths because they come directly from the source code of software. There are many improvements on finding test paths methods in which the researches to find infeasible test paths has gained the most focus. According to Hedley and Hennell [8], up to 12,5% test paths are infeasible in unit functions. Removing these test paths can dramatically improve the test data generation process. Symbolic execution is one of the method to find infeasible test paths from source code [11] [4]. However, symbolic execution causes the test data generation time to increase while it can only find out a small part of infeasible test paths [5] [1] [7]. Although these researches have gained very good results, some functions inside the Genetic Algorithm are still done manually which cause a lot of effort. Duc-Anh et al. improved the test path generation process from CFG. In his method, source code is parsed to get its corresponding CFG. Then, the CFG is traversed to find test paths by using a backtracking algorithm [10] (in this paper, we refer to this method as STCFG). In the step of finding test paths, at decisive vertices, the feasibility of the test path from the initial vertex to the decisive vertex is checked. This method prevents us from generating infeasible test paths (if a test path contains an

infeasible part, it becomes infeasible). The main disadvantage of this method is that it costs a lot of time in solving the constraint expressions when the given CFG has many decisive vertices and a few infeasible paths. This method has not given the causes of infeasible paths.

Both static and dynamic testing have the same purpose of generating a smaller set of test data with greater coverage. Both methods generate test data based on the source code analysis, generating constraint expressions, and retrieving test data using SMT-Solvers. It is the fact that those solvers generate random values in the input parameter value ranges which cannot be their boundary values. Those values can satisfy the required coverage but cannot find out errors in boundary values. For high quality software, even when the coverage is satisfied, black box testing is required to find errors. In practice, to find test data from boundary values, we need to read the software specification. This process is hard to be done automatically because software specification is normally in natural language. To solve this problem, we base on the source code to generate test data from boundary values. In our opinion, the set of boundary points found from source code always contains the set of boundary points from specification thanks to the tuning process from requirement to design, and to source code. As a result, boundary values related to test data found from source code can find errors in source code and software specification which is greatly needed in software companies.

This paper proposes two methods to generate test data statically which can deal with disadvantages of the above previous researches. The first method is to use weighted CFG (named WCFT - Weighted Control Flow Testing) to select test paths which are the most weighted ones and have not been visited to reduce the test data generation time while finding infeasible test paths. The second method (named BVTG - Boundary Values Test data Generation) is to find out boundary values of input parameters based on branch statements. These values will be used to generate test data which can find errors caused by boundary values. The test data set which combines test data generated from these two proposed method will have higher error detection ability with the same code coverage. Experiments are performed with the implemented tool called WCFT4Cpp to show the effectiveness of the proposed methods.

The rest of this paper is organized as follows. Section II presents the method to generate test data from a weighted CFG. The method to generate test data from boundary values is presented in Section III. Experiments of two proposed methods with results are shown in Section IV. Finally, the paper is concluded in Section V.

II. Generate Test Data From Weighted Control Flow Graph

In this paper, *control follow graph (CFG)*, *test path*, and *path* are important concepts which can be found in Duc-Anh et al.'s paper [10]. We give two other main concepts which are used in this paper.

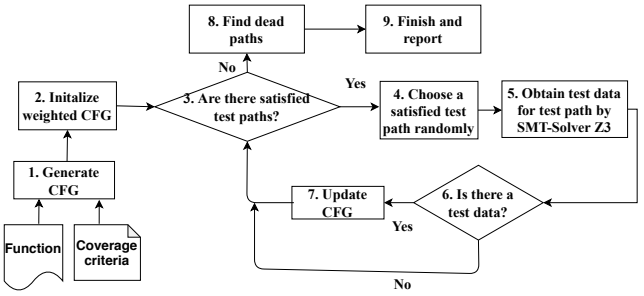

Figure 1: An overview of WCFT test data generation method.

Definition 1 (Dead path): In a given CFG, a path which is not covered by any test data is call a dead path.

Definition 2 (Dead code): A piece of code which is not covered by any test data is called dead code.

A. Generate CFG for a Unit Function

Recently, a well-known method to generate test data statically which guarantees the statement coverage (C1), branch coverage (C2), and MC/DC coverage (C3) is to generate from the CFG generated from a given source code. From this CFG, test paths can be found. Then, test data can be generated by using a strategy from longest to shortest test paths or vice versa. To satisfy the C1, C2, C3 coverages, that strategy is good enough. However, to find out the infeasible paths and dead code, we need a better strategy. For this reason, this paper proposes a method (named WCFT) to generate test data statically satisfying a given coverage with an appropriate strategy. The overview of the proposed method is shown in Figure 1.

Given a unit function and a specific coverage criteria, the required CFG will be generated using the method proposed by Duc-Anh et al. [10] (step 1). Then, the weight for every edges of this CFG is initialized and all vertices are marked as *not visited* (step 2). In the third step, we check if there is any test path satisfies the following two conditions: the test path which has the greatest weight and has not been visited. If no satisfied test path, we come to the step 8. Otherwise, we come to step 4. In the forth step, if there are many satisfied test paths, the process randomly chooses one. From the selected test path, the constraint expression is generated [10] and passed to the SMT-Solver Z3 (step 5). In the meantime, the test path is marked as *visited*. In the sixth step, if the solution exists for that test path, we come to step 7 which is to update the weight for the CFG and store the solution of the test path under processing. Then, we come to step 3. If the solution does not exist, we also come to step 3. In step 8, we have a CFG whose weight has been updated, (called UCFG - Updated Control Flow Graph). From this UCFG, we can find dead path (if exists). The first vertex of dead path is the branch statement which makes the CFG have infeasible paths. Other vertices of a dead path, except the last one, are corresponding to dead code. The test data generation for loop is done the same as described in Duc-Anh et al.'s paper [10] and is not related to the weighted CFG.

Generating the corresponding CFG of the unit function under testing is the first step in the proposed method. Details of the CFG generation algorithm are described in Algorithm 1. The input of the algorithm are source code of a unit function written in C/C++ language f and a coverage criterion t. The output is a CFG $graph$ satisfying the given coverage criterion. The algorithm starts by initializing $graph$ to be an empty

Algorithm 1: CFG generation

input : f: source code
t: coverage criterion
output : $graph$: CFG
1: $graph$ = an empty graph
2: B = a list of blocks by dividing f
3: G = a graph by linking all blocks in B to each other
4: Update $graph$ by replacing f with G
5: **if** G contains $return/break/continue$ statements **then**
6: Update the destination of $return/break/continue$ pointers to destinations
7: **end if**
8: **for** each block M in B **do**
9: **if** block M can be divided into smaller blocks **then**
10: **call** Algorithm 1 (M, t)
11: **end if**
12: **end for**

graph (line 1). The given source code f is divided into a list of blocks named B: $block_0$, $block_1$, ..., $block_{n-1}$, $block_n$ (line 2). In this case, the type of each block may be a statement, or a control block. Subsequently, a graph G describing the order execution of all above blocks is generated (line 3). CFG $graph$ is then updated by replacing the vertices of f with the graph G (line 4). After that, if graph G contains vertices corresponding to $break/continue/return$ statements (line 5), CFG $graph$ continues to be updated by pointing these vertices right to destinations (line 6). Next, each block M of the list B is checked whether it can continue being divided into smaller blocks (line 9). If it can, this means that block M has not satisfied the given coverage criterion. CFG $graph$ is then updated by parsing these smaller blocks by calling Algorithm 1 itself for M and t (line 10). Otherwise, B satisfies the given coverage criterion. The algorithm terminates when all blocks in $graph$ cannot be divided into smaller blocks.

B. Generate Test Paths From a CFG

When we have the generated CFG from a given unit function, we can obtain the list of test paths from that CFG. From these test paths, test data can be generated as described in sections below. Because this is a static method to generate test data, we need a method to process loops appropriately. In the proposed method, we allow user to specify the maximum number of loop times when generating the corresponding CFG of a unit function. This number is called *depth* and used as a parameter to the algorithm which generates test paths. Details of the test paths generation process are shown in Algorithm 2. The algorithm accepts the first vertex v of CFG and the maximum number of loop times $depth$ as input parameters. $path$ is a global parameter which is used to store test paths

Algorithm 2: Generate test paths from a CFG

input : v The first vertex of the CFG corresponding to C3 coverage
$depth$: the maximum number of iterations for a loop
$path$: a global variable to store a test path
output : P: a list of feasible test paths
1: **if** v == $NULL$ or v is the end vertex **then**
2: Add $path$ to P
3: **else if** the occurrence number of v in $path$ <=$depth$ **then**
4: Add v to the end of $path$
5: **for** each adjacent vertex u of v **do**
6: **call** Algorithm 2 (u, $depth$, $path$)
7: **end for**
8: Remove the latest vertex added in $path$ from it
9: **end if**

while traversing the CFG. At first, the algorithm check if v is the last vertex of the CFG or is NULL. If yes, $path$ is added to P (line 2). Otherwise, if the number of times v appears in $path$ is not greater than $depth$, v is added to $path$ (line 4). Later, the adjacent vertices of v will be visited by calling Algorithm 2 itself (line 5 to 7). Finally, the last vertex added to $path$ is removed in order to visit other vertices of the CFG. Algorithm 2 is based on the Algorithm 2 - CFG Traverse presented in [10] in which the step of checking the feasibility of a test path when checking the decisive vertex is removed.

C. Update Weight For a CFG and Generate Test Data

The next step of the proposed method is to update weight for a CFG. From this updated CFG, we can generate required test data. These two steps are integrated in one algorithm shown in Algorithm 3. Algorithm 3 returns the list of test data

Algorithm 3: Generate test data and update weight for a CFG

input : *graph*: a given CFG
output : *S*: The list of test data corresponding to *graph*;
UCFG: The weight updated CFG
1: Initialize weight for every edges of the CFG to be *1* and set all test paths to be not visited
2: **while** *graph* has a not visited test path **do**
3: *t* := test path which has the greatest sum of weights and is not visited
4: *constraint* := constraint expression generated from *t*
5: *solution* = solution of constraint from SMT-Solver Z3
6: **if** (*solution* is not null) **then**
7: S.append(*solution*)
8: Update test path *t* in graph by adding *1* to all its edges
9: **end if**
10: Mark t to be visited
11: **end while**
12: UCFG = graph

stored in S which guarantees the given coverage criteria and a weight updated CFG after generating test data. The algorithm accepts $graph$ which is the generated CFG from a given unit function. Algorithm 3 starts by initializing all weights of its edges to be *1*, and setting all vertices to be not visited (line 1).

After that, the algorithm checks if $graph$ has any *not visited* test path (line 2)? If there is no such test path, UCFG is set to be the updated graph and the algorithm stops (line 12). If there exist test paths which are not visited, the test path which has the greatest sum of weight is selected, denoted by t (line 3). Then, the algorithm generates t's corresponding constraint expression (line 4) and uses SMT-Solver Z3 to solve the newly generated expression (line 5). In line 6, the algorithm checks if the constraint expression of t has a solution? If yes, the solution is added to S and *1* is added to all edges of the test path (line 7 to 8). Later, the algorithm marks the path t to be visited (line 10). Finally, the algorithm comes back to line 2 to generate test data and update $graph$ until the condition is *false*. From UCFG, we can find out dead code from dead paths which have edges whose weights are not updated.

D. Find Dead Paths

In unit functions, we cannot avoid errors when executing condition statements where there exist condition expressions which are *always true* or *always false*. Those expressions

Algorithm 4: Dead path detection

```
input  : UCFG: a given UCFG
output : paths: a list of dead path
 1: for (Each testPath in UCFG) do
 2:     deadPath := ∅
 3:     for (Each edge ∈ testPath) do
 4:         if (edge.getWeight() == 1) then
 5:             deadPath.append(edge)
 6:         else
 7:             if (deadPath <> ∅) and deadPath ∉ paths then
 8:                 paths.append(deadPath)
 9:             end if
10:             deadPath := ∅
11:         end if
12:     end for
13:     if (deadPath <> ∅ ) and (deadPath ∉ paths) then
14:         paths.append(deadPath)
15:     end if
16: end for
```

make the corresponding statements always be executed or not. This leads to errors and difficulties when maintaining the projects. As a result, automatically detecting such dead code is important in reducing the errors debugging and source code maintaining time. This paper presents a method to automatically detect dead code by using the UCFG. To find dead code, we need to find out the dead paths, i.e., no test case can cover. The first vertex of a dead path is the statement which causes the infeasible path. All other vertices of that dead path, except the first and the last vertices, are corresponding to its dead code. The proposed method to find infeasible paths *paths* is included in Algorithm 4. With each of the test paths of UCFG (line 1), the algorithm checks each of its edges *edge* (line 3) to find a set of adjacent edges which have weights of *1*. If *edge*'s weight is *1*, it will be added to *deadPath* (line 5). Otherwise, the algorithm checks if *deadPath* contains any *edge* and if *paths* contains *deadPath* or not (line 7). If yes, *deadPath* is added to *paths* (line 8) and reset to be an empty path in order to find another dead path (line 10). If the last edge of a test path has weight of *1*, the algorithm comes to line 13 to line 15 where *deadPath* is added to paths. When the algorithm stops, *paths* contains all dead paths of the given UCFG.

III. Generate Test Data for Boundary Values

To test a unit function with boundary values, a tester needs to read the function's specification. Then, he divides the valid value ranges of each input parameters into equivalence class partitions and generates test data from the boundaries values of those partitions. This manual task often costs a lot of effort, but contains many errors. This paper proposes a fully automatic method (named BVTG) to generate test data for boundary values corresponding to equivalence class partitions of input parameters which are in primitive types and for conditional statements. Initially, we generate the required CFG using Algorithm 1 with C3 coverage in order to separate compound conditions into single conditions.

The inputs of Algorithm 5 are the first vertex v of the CFG corresponding to C3 coverage, the maximum number of loop times for loops statements *depth*, a global variable *path* which stores visited test paths, lower boundary value *start*, upper boundary point *end*, and the *step* which is used to generate test data for boundary values. The output of the algorithm is S which contains all boundary test data. Initially, we check if v is not NULL, not the last vertex of the given CFG, and the appearance number of v is not greater than *depth* (line 1). If v does not satisfy the above condition, the algorithm stops. Otherwise, the algorithm adds v to *path* (line 2). If v represents a "*if, else*" condition statement (line 3), the algorithm backs up *condition* for later use (line 4). After that, the statement is assigned to *condition* variable (line 5). The algorithm replaces the comparison operator of this condition with "==" operator (line 6). If the two sides of the *condition* are in *boolean* type, the algorithm assigns the *solution* of *path* to *testData* and adds a random value of the rest input parameters to *testData* (line 8, line 9). Otherwise, the algorithm finds the *solution* from the first vertex to v by adding *i* to the right side of $condition$, where *i* is from *start* to *end* and the step is *step* (line 10, line 13). Then, the generated solution is stored in *testData* (line 14). Later, the algorithm adds random values of the missing input parameters to *testData* (line 16) and adds this *testData* to *S* (line 17). After that, the algorithm restores the *condition* statement (line 21) with the backed up value in line 4. Finally, if the current *path* is feasible (line 23), the algorithm continues calling itself with all adjacent vertices of v to find all boundary test data.

IV. Experiments

To evaluate the effectiveness of the proposed methods, we implemented them in a tool named WCFT4Cpp which is based on CFT4CUnit [10] and the SMT-Solver Z3. The tool has already contained implementation for STCFG and Concolic methods which are implemented by Duc-Anh [10]. We performed two experiments to evaluate the proposed methods

Algorithm 5: Boundary test data generation

input : v: The first vertex of CFG corresponding to C3 coverage
$depth$: the maximum number of iterations for a loop
$path$: a global variable used to store a discovered test path
$start$: lower boundary for the input parameter
end: upper boundary for the input parameter
$step$: the step when generate test data from boundary values
output : S: a list of test data

1: **if** (v != $NULL$) and (v is not the end vertex) and (the appearance times of *v* in path <= *depth*) **then**
2: $path$.add(v)
3: **if** v is a decisive vertex corresponding to *if, else* statements **then**
4: Backup *condition*
5: *condition* = the condition statement corresponding to v
6: path.normalize(v) /* Replace condition operator in v by "==" operator */
7: **if** *condition* has two sides in boolean type **then**
8: *testData* = the *solution* of *path*
9: Add random value of the rest input parameters to *testData*
10: Add *testData* to *S*
11: **else**
12: **for** *i* from *start* to *end* where step is *step* **do**
13: Add *i* to the right side of *condition*
14: *testData*= the *solution* of *path*
15: **if** *testData* !=NULL **then**
16: Add random value of the missing input parameters to *testData*
17: Add *testData* to *S*
18: **end if**
19: **end for**
20: **end if**
21: Restore *condition* using the backed up value in line 4
22: **end if**
23: **if** path is feasible **then**
24: **for** u is the adjacent vertex of v **do**
25: **Call** Algorithm 5 (*u, depth, path, start, end, step*)
26: **end for**
27: **end if**
28: Remove the last added vertex in path
29: **end if**

Table I: Comparison of STCFG with WCFT

Input			WCFT	STCFG	Branch
function	depth	LOCs	Time (s)	Time (s)	Cover
leapYear	0	6	0.19	0.14	100%
isTriangle	0	6	0.16	0.13	100%
PDF	0	6	0.15	0.11	100%
divisionTest	0	7	0.17	0.14	100%
CheckValiday	0	9	0.47	0.87	100%
TriType	0	12	1.61	2.92	100%
Grade	0	13	0.35	0.73	100%
foo	0	15	0.34	0.32	88%
calculateZodiac	0	60	1.36	5.65	100%
simpleWhileTest	1	6	0.17	0.1	100%
	4		0.27	0.62	100%
GCD	0	14	0.68	3.6	100%
	4		18.79	49.7	100%
Average	1	14	1.19	0.96	100%
	4		10.5	23.34	100%

Table II: Error detection comparison of BVTG, STCFG, and Concolic methods

Function	BVTG		STCFG		Concolic	
	Test data	Detected error	Test data	Detected error	Test data	Detected error
divisionTest	3	1/2	2	1/2	45	0/2
Grade	21	5/5	6	1/5	8	0/5
PDF	6	3/4	2	1/4	2	1/4
isTriangle	9	2/2	2	1/2	2	1/2
Tritype	28	3/3	16	2/3	30	2/3
leapYear	4	1/1	2	0/1	2	1/1

with existing methods: compare WCFT with STCFG [10] and compare BVTG with Concolic testing [6], [12]. We performed each of test functions 20 times. The average results of all tests are shown in Table I, Table II. Experiments are performed on a machine which runs Windows 10, Intel Core i5, 8250U, 1.60 GHz, and 8GB RAM using Mingw32 in IDE Dev-Cpp 4.9. The implemented tool and tested functions are available on this web page https://testdatakse.herokuapp.com/.

A. Comparison between WCFT and STCFG

In this experiment, we compare the required time to generate test data using C2 coverage criteria between the two methods: WCFT and STCFG. With each test cases, inputs are unit functions under testing and the maximum number of loop times for loop statements shown in column "*depth*". The number of lines of code for each functions is shown in column "LOCs". The time required to generate test data of WCFT and STCFG is shown in "*Time (s)*" corresponding to each method names. The result C2 coverage is shown in column "*Branch coverage*". The corresponding output is the list of test data. If the function does not have loop statement, *depth* is set to *0*. The comparison of test data generation time is shown on Table I.Unit functions *Tritype, Grade, Average, GCD* are retrieved from the research of Duc-Anh et al. [10], these functions *leapYear, isTriangle, CheckValidDay, caculate-Zodiac* are collected from the internet, and *foo, divisionTest, PDF, simpleWhileTest* are made by ourselves.

Table I shows that the C2 coverage results of the two methods are the same while there are much difference in the time required to generate test data. It takes WCFT more time to generate test data than STCFG in the following cases:

- Simple functions with no loop and few branch statements such as *leapYear, isTriangle, divisionTest, PDF, foo*.
- Functions with single loops and small *depth* (mentioned in Algorithm 5) such as *simpleWhileTest*.

This is because the corresponding CFGs of those functions have smaller number of vertices. This leads to a fast test paths generation speed. With the same test paths generation time, it takes WCFT some more time to update weight for each of generated test paths. These results show that STCFG has better performance than WCFT to generate test data for simple functions. However, in real projects in practice, we usually have more complex functions with which WCFT shows better performance.

It takes WCFT less time to generate test data than *STCFG* in the following cases:

- Functions which have many condition statements such as *CheckValidDay, TriType, Grade, calculateZodiac*.
- Function which have loops with big *depth* such as *GCD, Average* (especially, the bigger *depth* is, the slower STCFG is).

The reason for this is that the corresponding CFGs of those functions have many decisive points. This required STCFG to use SMT-Solver more times to solve the feasibility of test paths from the first vertex of CFG until the vertex under checking. In some cases, this checking process is redundant when the CFG have few or do not have any feasible test paths. These results implies that WCFT has better performance than STCFG to generate test data for more complex functions which usually appear in projects in practice.

B. Error Detection Comparison of BVTG, STCFG, and Concolic methods

To compare error detection ability of three methods BVTG, STCFG, and Concolic, we have added errors to functions *divisionTest, Grade, PDF, isTriangle, Tritype, leapYear* and tested them with those testing methods. Experimental results are shown in Table II. In this experiment, for those unit functions which have not got any loop, we set the value of *depth* to *0*. The input values for Algorithm 5's parameters *start, end, step* are *-1,1,1*, respectively. The coverage for all three methods is *C2*. In Table II, the number of generated test data for each methods is shown in column "*Test data*" corresponding to the method names. The ratio of the number of detected errors to number of added errors is shown in column "*Detected error*" corresponding to the method's name, too. From experimental results shown in Table II, we have the following observations.

- In all cases, BVTG generates more test data than STCFG. However, it can detect more errors than STCFG.
- In 2 out of 6 cases (*divisionTest* and *Tritype*), BVTG can generates fewer test data than Concolic method. However, BVTG can detect more errors than Concolic method.
- In 4 out of 6 cases (*Grade, PDF, isTriangle*, and *leapYear*), BVTG can generate more test data while it can detect more errors than Concolic method.

Experimental results show that BVTG outperforms STCFG and Concolic methods in the error detection ability. The combination of the generated test data from these two proposed methods will have higher error detection ability with the same code coverage. This is one of the key value for the two methods to be applied successfully in practice.

V. Conclusion

The paper proposed two methods for generating test data from source code WCFT and BVTG. WCFT method is to generate test data by using static testing bases on the weighted CFG of a given unit function under testing. In this method, we generate the weighted CFG and select the test path with greatest weight to generate test data. With this improvement, the test data generation time is reduced, infeasible test path and dead code can be found. This results in several methods which optimize source code based on weighted CFG. BVTG method is to generate test data bases on boundary values. The combination of the generated test data from these two methods can detect potential errors in boundary values while maintaining its high code coverage. The two methods are implemented in a tool named WCFT4Cpp. Experiments results for some common source code in research community show that the two methods outperform STCFG and Concolic methods in test data generating time and in detecting boundary values related errors.

Acknowledgments

This research is supported by the research project No. CN.20.26 granted by University of Engineering and Technology, Vietnam National University, Hanoi (VNU-UET).

References

[1] M. A. Ahmed and I. Hermadi. GA-Based multiple paths test data generator. *Comput. Oper. Res.*, 35(10):3107–3124, Oct. 2008.

[2] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. of the 2008 23rd IEEE/ACM Int. Conf. on Automated Softw. Eng.*, ASE '08, page 443–446, USA, 2008. IEEE Computer Society.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[4] M. Delahaye, B. Botella, and A. Gotlieb. Infeasible path generalization in dynamic symbolic execution. *Information and Softw. Technol.*, 58, 08 2014.

[5] A. S. Ghiduk. Automatic generation of basis test paths using variable length genetic algorithm. *Inf. Process. Lett.*, 114(6):304–316, June 2014.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[7] D. Gong, W. Zhang, and X. Yao. Evolutionary generation of test data for many paths coverage based on grouping. *J. Syst. Softw.*, 84(12):2222–2233, Dec. 2011.

[8] D. Hedley and M. A. Hennell. The causes and effects of infeasible paths in computer programs. In *Proc. of the 8th Int. Conf. on Softw. Eng.*, ICSE '85, page 259–266, Washington, DC, USA, 1985. IEEE Computer Society Press.

[9] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[10] D. Nguyen, P. N. Hung, and V. Nguyen. A method for automated unit testing of C programs. In *2016 3rd National Foundation for Science and Technol. Development Conf. on Information and Computer Science (NICS)*, pages 17–22, 2016.

[11] M. Papadakis and N. Malevris. A symbolic execution tool based on the elimination of infeasible paths. In *Proc. of the 2010 Fifth Int. Conf. on Softw. Eng. Advances*, ICSEA '10, page 435–440, USA, 2010. IEEE Computer Society.

[12] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the 10th European Softw. Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int. Symposium on Foundations of Softw. Eng.*, ESEC/FSE-13, page 263–272, New York, NY, USA, 2005. Association for Computing Machinery.

[13] Z. Wang, X. Yu, T. Sun, G. Pu, Z. Ding, and J. Hu. Test data generation for derived types in C program. In *Proc. of the 2009 Third IEEE Int. Symposium on Theoretical Aspects of Softw. Eng.*, TASE '09, page 155–162, USA, 2009. IEEE Computer Society.

[14] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Proc. of the 5th European Conf. on Dependable Computing*, EDCC'05, page 281–292, Berlin, Heidelberg, 2005. Springer-Verlag.