CSEFuzz: Fuzz Testing Based on Symbolic Execution

Zhangwei Xie
Computer School, Beijing Information Science and Technology University, Beijing, China
Zhanqi Cui
Computer School, Beijing Information Science and Technology University, Beijing, China
Jiaming Zhang
Computer School, Beijing Information Science and Technology University, Beijing, China
Xiulei Liu
Computer School, Beijing Information Science and Technology University, Beijing, China
Liwei Zheng
Computer School, Beijing Information Science and Technology University, Beijing, China

Meta Data & Stats

Published in:	IEEE Access (Volume: 8)
Year:	2020
Number of Authors:	5
Citations:	1
Pages (PDF):	11
Figures:	0
References:	32
Formals:	0 definitions

Table of Content

1. Introduction
2. Overview
3. Our Technique
4. Instance Analyses
5. Experiments
6. Related work
7. Conclusion
8. Acknowledgments
9. References

Research Questions

RQ 1: Can the test cases generated by symbolic execution help improve the efficiency of fuzz testing?

RQ 2: Compared with the *afl-cmin* command provided by Kelinci, can CSEFuzz lower the time costs of selecting initial seed test cases?

RQ 3: Compared with the *afl-cmin* command provided by Kelinci, can CSEFuzz improve the paths covered and the number of defects detected by fuzz testing?

RQ 4: Will different initial seed test case selection strategies affect the efficiency of fuzz testing?

CSEFuzz Framework

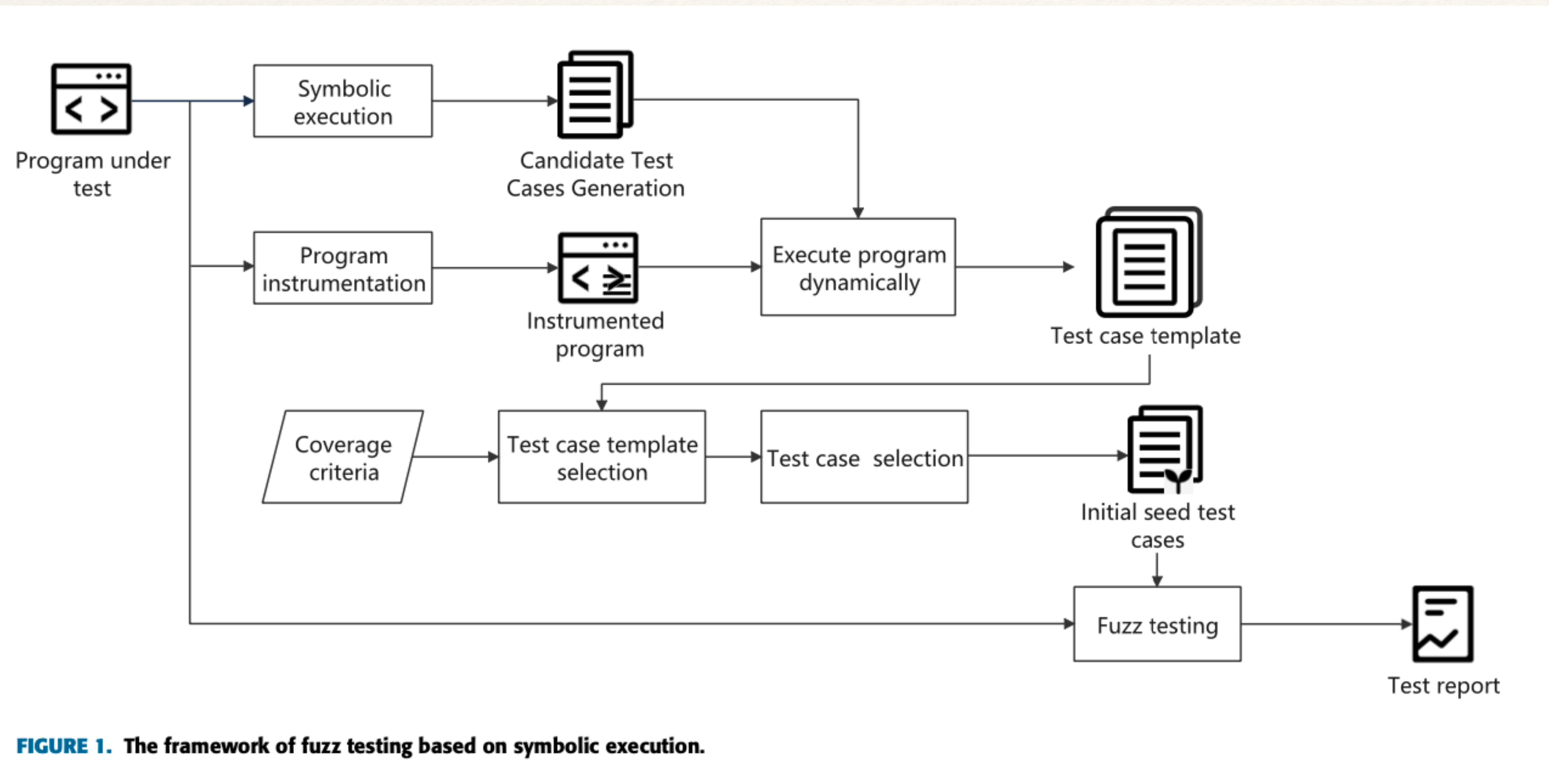

FIGURE 1. The framework of fuzz testing based on symbolic execution.

- generates candidate test cases for the program under testing by symbolic execution.
- extracts the test case templates of the test case candidates based on the coverage information and selects a subset of test case templates according to different coverage criteria
- selects representative test cases according to the coverage of the test case templates

Candidate Test Case Generation

```
1    public static int test(int x, int y,int z){
2        if(y >100){
3            System.out.println("hello");
4            if(x+z>102){
5                System.out.println("hello");
6            }
7        }
8    }
```

FIGURE 2. Example of the relationship between path conditions and input variables.

- the tester first constructs a control flow graph of the program under test
- the tester starts traversing from the first statement of the program
- If the statement is encountered, then the relationship between the program variables and input variables will be updated
- a conditional statement is encountered, then two paths will be branched out
- if solved - saved as a candidate test case
- Example: 1-2-8: y <= 100, <x=flag, y=100, z=flag>

Test Case Template Generation

Algorithm 1 Select a Subset of Test Case Templates According to Specific Coverage Criteria

```
Input: set tcTemplate;
Output: set iniTemplates;
 1: set Factors:= all the program elements in test case template set tcTemplate    /*According to different coverage strategies, the program elements can be decisions, conditions, and basic paths*/
 2: while (Factors != ∅) do
 3:    test case template;    //to store the test case template that covers the maximum number of elements in Factors
 4:    count = 0;    //store the number of program elements that are covered by the current test case template
 5:    for each tc in tcTemplate do
 6:       num:= number of program elements in Factor that are covered by the current test case template tc
 7:       if (num > count) then
 8:          template:= tc
 9:       end if
10:    end for
11:    initemplates.add(template);
12:    Remove program elements contained in the test case template from Factors
13:    tcTemplate.delete(template)
14: end while
15: return iniTemplate
```

- is used to describe the program coverage information of test cases
- test case *t* are {$item_1$, $item_2$, ..., $item_i$, ...} ($item_i$ can be any kind of program element, such as branches, conditions, and paths). template(t) = {$item_1$, $item_2$, . . . , $item_i$, . . . }
- subset of test case templates can be selected according to different coverage criteria to achieve the goal of covering as many program elements as possible with as few test case templates as possible
- white-box testing uses 3 different coverage criteria to choose test case templates: decision coverage, condition coverage and path coverage
- decision coverage tries to cover different branches of each decision in the program under test
- condition coverage tries to cover different values of each condition of the decisions in the program under test
- path coverage tries to cover every basic path of the program under test

Initial Seed Test Case Selection

Algorithm 2 Use of the Initial Seed Test Cases for Fuzz Testing

Input: tested program P, initial seed test case set *Seeds*

Output: test report *testReport*

```
1:  effectiveSeeds = ∅;   //Effective seed test case set
2:  for each seed ∈ Seeds do
      //Validate the seed test case
3:    Execute(seed);   //use the seed to execute the program
      and monitor the execution result
4:    if (a new program path is covered during seed's exe-
      cution) then
5:      effectiveSeeds.add(seed);
6:    end if
7:  end for
8:  while (true) do
9:    seed:= choose a seed test case from effectiveSeeds
10:   newSeed:= mutate(seed);   //generate a new test case
      by mutating
11:   Execute(newSeed);   //use newSeed to execute the
      program and monitor the execution result
12:   if ( newSeed covered a new program path during exe-
      cution ) then
13:     save the new program path in test report testReport
14:   end if
15:   if ( newSeed trigger crash during execution ) then
16:     save the defect in test report testReport
17:   end if
18:   if ( reach the specific time ) then
19:     break;
20:   end if
21: end while
22: return testReport
```

- the corresponding initial seed test cases need to be selected
- in the candidate test cases, multiple test cases can cover the same program elements
- the fuzz tester will validate the initial test cases after they are selected
- the validated initial seed test cases are mutated to obtain new test cases
- the program path that is covered and the defects that are detected will be added to the test report.

Experimental Results

TABLE 3. **Comparison of CSEFuzzn and Kelinci in terms of path coverage and defects detection.**

Experimental Objects	Kelinci		CSEFuzzn	
	Number of Paths Covered	Number of Defects Detected	Number of Paths Covered	Number of Defects Detected
TCAS_V30	16	21	22	21
BankAccount	13	2	13	4
Apollo	7	12	8	12
MerArbiter-v2	11	11	12	12
LoopExample	11	2	11	2
TwoLoopExample	18	4	21	4
WBS	9	5	10	6
rbt	3	3	33	3
TreeMapSimple	6	3	27	3
MathSin	12	2	18	2
fuzz/gram/test	14	0	14	0
TOTAL	120	65	189	69

TABLE 4. **Time costs of initial seed test case selection and validation (in seconds).**

Experimental Objects	CSEFuzzn		CSEFuzzd		CSEFuzzc		CSEFuzzp		*afl-cmin*	
	ISS[1]	ISV[2]	ISS	ISV	ISS	ISV	ISS	ISV	ISS	ISV
TCAS_V30	0.0	590.7	2.7	3.9	2.8	14.1	2.7	3.8	67.5	10.8
BankAccount	0.0	10.1	1.1	2.0	1.2	2.0	1.1	2.0	5.7	2.4
Apollo	0.0	136.8	2.1	2.2	2.1	2.3	2.1	2.2	18.6	3.3
MerArbiter-v2	0.0	301.1	2.0	2.0	2.0	2.0	2.0	2.0	35.8	4.7
LoopExample	0.0	104.3	1.0	2.0	1.0	3.0	1.0	2.0	12.4	9.7
TwoLoopExample	0.0	16832.9	13.6	5.0	14.1	8.0	14.9	5.9	2320.9	13.1
WBS	0.0	27.0	2.0	10.0	2.0	9.9	2.0	11.1	6.6	9.4
rbt	0.0	38969.9	18.5	9.9	18.5	9.7	55.0	256.1	4827.5	13.1
TreeMapSimple	0.0	716.1	2.9	11.9	2.8	12.8	2.6	52.1	95.1	12.1
MathSin	0.0	56.7	1.0	4.7	1.0	4.7	1.0	4.5	5.0	2.0
fuzz/gram/test	0.0	2.0	1.0	2.0	1.0	2.0	1.0	2.0	1.0	2.0
TOTAL	0.0	57747.6	47.9	55.6	48.5	70.5	85.4	343.7	7396.1	82.6

[1] ISS is the time cost of the initial seed selection.
[2] ISV is the time cost of the initial seed validation.

RQ Answers

- RQ1: by using the test cases generated by symbolic execution as initial seed test cases, more program paths are covered and more defects are detected. The efficiency of fuzz testing can be improved by using seeds generated by symbolic execution.
- RQ2: directly using test case candidates that are generated by symbolic execution will consume too much time for validating test cases
- RQ3: compared with command afl-cmin, which was provided by Kelinci, CSEFuzz is based on decision, condition and path coverage criteria and can improve the path coverage of fuzz testing and the number of defects detected
- RQ4: different test case selection strategies will affect the results of CSEFuzz. CSEFuzz based on a path coverage criterion performs best in the covered paths and the total time costs for initial seed selection and defects detected. The efficiency of fuzz testing can be improved by choosing proper seed selection strategies

Feedback

- *Problem statement(research statement is clear) +*
- *Innovation (the work brings new innovation ideas) +*
- *Contribution(CSEFuzz) +*
- *Logical correcteness +*
- *Proof of statements(there are no questional, unproven statments or conclusions) +*
- *Readablity -*

What is good/interesing about the paper

- *Structured*
- *Detailed example*
- *Novel approach*

What could be better

- *These is no code base*
- *Pseudocode*
- *Readability*

8. Conclusion