IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 6, 2020, accepted October 9, 2020, date of publication October 13, 2020, date of current version October 26, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3030798

CSEFuzz: Fuzz Testing Based on Symbolic Execution

ZHANGWEI XIE, ZHANQI CUI, (Member, IEEE), JIAMING ZHANG, XIULEI LIU, AND LIWEI ZHENG

Computer School, Beijing Information Science and Technology University, Beijing 100101, China

Corresponding author: Zhanqi Cui (czq@bistu.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1400402, in part by the National Natural Science Foundation of China under Grant 61702041 and Grant 61601039, and in part by the Qin Xin Talents Cultivation Program of Beijing Information Science Technology University under Grant QXTCP C201906 and Grant QXTCP B201905.

ABSTRACT Fuzz testing has been successful in finding defects of various software packages. These defects include file parsing, image processing, Internet browsers, and network protocols. However, the quality of the initial seed test cases greatly influences the coverage and defect detection capability of fuzz testing. To address this issue, we propose CSEFuzz, a fuzz testing approach based on symbolic execution for defect detection. First, CSEFuzz generates candidate test cases by symbolic execution and collects coverage information of the test cases. Then, CSEFuzz extracts the test-case templates of the test cases and selects a set of test-case templates according to specific coverage criteria. Finally, CSEFuzz selects test cases according to the selected test-case templates, and the selected test cases are used as initial seed test cases for fuzz testing. Experiments are conducted on 11 open-source programs. The results show that in comparison with afl-cmin, which is the test-case selection command of Kelinci, CSEFuzz with a path coverage criterion reduces the time costs of the initial seed test selection and verification by 94.26%. In addition, compared with afl-cmin, 32 more paths are covered and 16 more defects are detected by CSEFuzz.

INDEX TERMS Fuzz testing, initial seeds, symbolic execution, test coverage criteria.

I. INTRODUCTION

Fuzz testing is an automated testing method to find defects by generating random inputs of the software under test and monitoring its execution. Fuzz testing executes the software dynamically and selects existing test cases to be mutated according to the coverage information to generate new test cases until no more program paths can be covered. The defects and vulnerabilities of the software can be detected, if some failures or abnormal behaviors are observed during fuzz testing. Fuzz testing has achieved good results in finding defects in various software packages, such as network protocols [1], mobile apps [2], wearables [3], and deep natural networks [4]. For instance, as of June 2020, Google's OSS-Fuzz project has helped developers find 20,000 defects in 300 open-source software packages.[1] The fuzz tester American Fuzzy Lop (AFL)[2] developed by Zalewski successfully found multiple defects hidden in more than 150 software packages, including PHP, OpenSSL, SQlite and Internet Explorer. Studies have shown that the code coverage will be affected by the quality of the initial seed test cases [5]. This is because fuzz testing usually generates new test cases by mutating the initial seed test cases. Initial seed test cases with high quality can help the fuzz tester to generate new test cases that can cover more and deeper paths more quickly. Among the existing fuzz testing methods, the common method for obtaining the initial seed test case set relies on designing test cases manually or uses test cases provided by open-source projects [6]. However, the quality of these test case sets cannot be guaranteed, and the coverage of the program paths is limited, which seriously restricts the effectiveness of fuzz testing.

Symbolic execution partly solves the lack of test cases in traditional test methods, and it has a good capability to

The associate editor coordinating the review of this manuscript and approving it for publication was Claudio Agostino Ardagna.

[1] OSS-Fuzz, https://github.com/google/oss-fuzz.

[2] American fuzzy lop (AFL). https://lcamtuf.coredump.cx/afl/

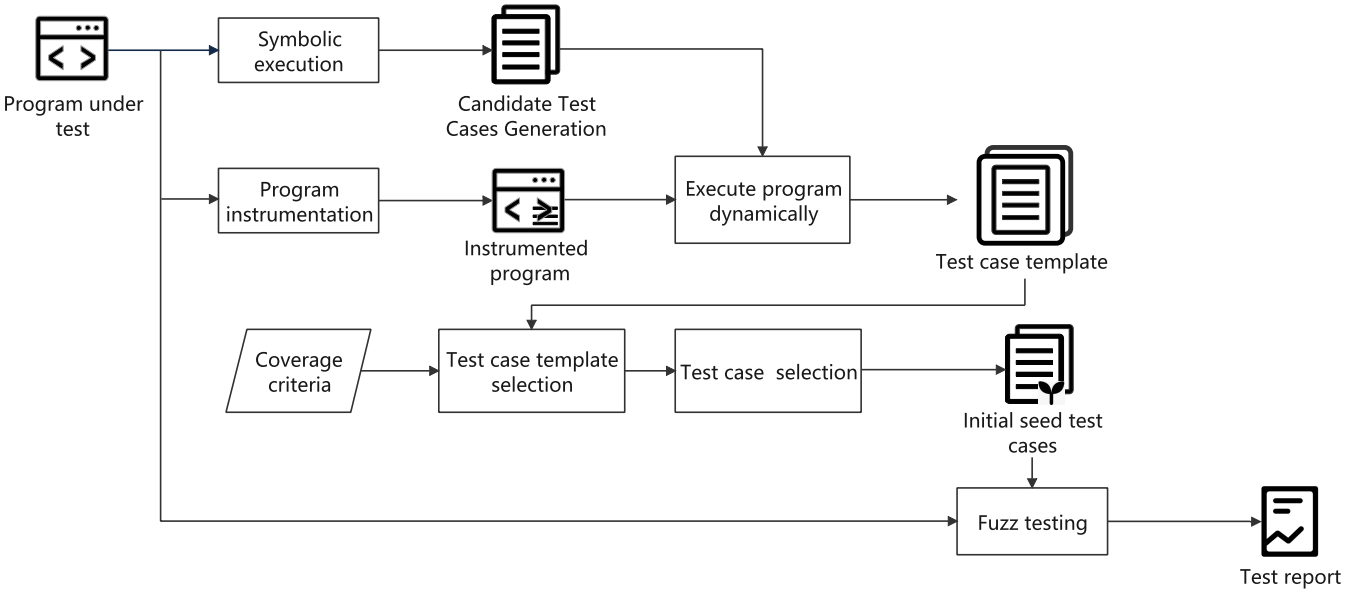

FIGURE 1. The framework of fuzz testing based on symbolic execution.

find defects [7]–[9]. However, in the process of symbolic execution, the tools usually generate a large number of test cases that cover the same path. If the test cases are directly provided to the fuzz tester as the initial seed test cases, much time will be wasted as the fuzz tester validates the seeds, which seriously affects the efficiency of the fuzz tester. For example, when using symbolic Pathfinder (SPF) [10] to test the experimental project rbt, 39,999 test cases were generated. When all of these test cases were directly provided to the fuzz tester Kelinci [11], it took more than 10 hours to validate the seed test cases. Although Kelinci also provides the seed selection command afl-cmin to filter the initial seeds, the time cost is still very heavy. For example, when using afl-cmin to handle the test cases generated by SPF for the rbt project, it still took more than 1 hour. Therefore, how to quickly select effective initial seeds from the candidate test cases is a key point in improving the efficiency of fuzz testing [6].

To solve this problem, this article proposes Coverage-guided Symbolic Execution for Fuzz testing (CSEFuzz), which is a defect detection approach based on combing symbolic execution and fuzz testing. First, CSEFuzz uses the symbolic execution technique to generate candidate test cases with high code coverage and dynamically executes the program to collect the coverage information of the candidate test cases. Then, CSEFuzz generates a test case template for the candidate test cases based on the coverage information and selects a subset of test case templates according to specific coverage criteria. Finally, CSEFuzz chooses the initial seed test cases on the basis of the selected test-case templates to perform fuzz testing. Experiments are conducted on 11 open-source programs. The results show that compared with the test case selection command afl-cmin provided by Kelinci, CSEFuzz, using different coverage criteria, can reduce the time cost for selecting and validating the initial seed test cases by 94.52%∼99.50%. In addition, compared to afl-cmin, 5∼32 more paths are covered and 12∼16 more defects are detected by CSEFuzz.

The main contributions of this article are as follows:

- We proposed a defect detection approach by fuzz testing based on symbolic execution. The symbolic execution generates test cases with high coverage for the program under test, and the test cases are used as the initial seed test cases for fuzz testing to improve the path coverage and defect detection capability of fuzz testing.
- We proposed to select initial seed test cases based on coverage information. The coverage of the test cases is collected by instrumentation and execution, and the initial seeds are selected according to specific coverage criteria to improve the efficiency of fuzz testing.
- A prototype tool called CSEFuzz was implemented based on the proposed approach, and case studies are conducted on a group of open-source programs to verify the effectiveness of the proposed approach.

The rest of this article is organized as follows: Section 2 details the approach of defect detection by fuzz testing based on symbolic execution, Section 3 introduces the experimental design and evaluation, Section 4 introduces related work, and Section 5 summarizes the paper.

II. FUZZ TESTING BASED ON SYMBOLIC EXECUTION

The framework of our approach is shown in Figure 1. First, our approach generates candidate test cases for the program under testing by symbolic execution. The test cases are used to drive the execution of the program after instrumentation and to collect the coverage information of each test case. Then, our approach extracts he 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, such as decision coverage, conditional coverage and path coverage. Finally, our approach selects representative test cases according to the coverage of the test case templates. The selected test cases are sent to the fuzz testing engine as initial seeds to start fuzz testing.

A. CANDIDATE TEST CASE GENERATION

The quality of the initial seed test cases will greatly influence the efficiency of fuzz testing. If a set of high-quality initial seed test cases can be provided before starting fuzz testing, this will help the fuzz tester to improve the path coverage and detect more defects. To obtain a set of initial seed test cases with high path coverage, this approach uses symbolic execution to collect program path constraints and uses a constraint solver to solve the path constraints to generate corresponding test cases.

During the symbolic execution, the tester first constructs a control flow graph of the program under test. Then, the tester starts traversing from the first statement of the program. If an assignment statement is encountered, then the relationship between the program variables and input variables will be updated. If a conditional statement is encountered, then two paths will be branched out. That is, the path with the current conditional statement takes **true** and **false** values. Two different path statuses are extended: one path status is added to the set of path statuses, which will be analyzed in the future, and the constraint solver is used to solve the constraint condition of the other path status. If the path condition is solved, then a test case is generated and saved as a candidate test case, and the next statement in the path is selected to continue traverse. If the path condition cannot be solved, then the analysis of the current path status is ended, and a new path status is taken from the path status set to be analyzed for symbolic execution. Finally, when the path status set is empty, the symbolic execution is stopped.

In the program, the value of some path constraints will not be affected by the changes of every input variable. However, the test cases generated by symbolic execution only include the values of the input variables that will affect the path constraints. In this situation, the generated test cases may lack the values of some input variables, which will cause the values of the test case inputs to be out of order. We take the following measures to solve this problem. First, all of the input variables of the program are extracted as a sequence $V=<v_1,v_2,v_3,\ldots,v_i,\ldots>$, where v_i represents the i-th input variable of the program. Then, when generating the j-th test case, the values of the input variables in the test cases are stored in a sequence V_j according to the order of the input variables in V. If the test case lacks the value of an input variable v_i, then the i-th input variable will be assigned with a special value flag. Finally, a completed sequence V_j is outputted as a test case.

For example, in the code snippet shown in Figure 2, the path constraint of path 1-2-8 is $y<=100$. This path constraint is only affected by the input variable y, so the test case generated by symbolic execution is $y=100$, and the values of the input variables x and zz are not included. To avoid the disorder of inputs when using the test case, the missing input variable is assigned with a specific value *flag*, that is, $<x=flag, y=100, z=flag>$.

```
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.

B. TEST CASE TEMPLATE GENERATION

The test case template is used to describe the program coverage information of test cases. Test cases that cover same program elements, share the same test case template. Assuming that test case t is used to execute the program under test that has been instrumented, the program elements executed by t are $\{item_1, item_2, \ldots, item_i, \ldots\}$ ($item_i$ can be any kind of program element, such as branches, conditions, and paths). Then, the test case template of t is described as $template(t) = \{item_1, item_2, \ldots, item_i, \ldots\}$. Test cases with the same test case template will cover the same program elements.

After extracting test case templates according to the program coverage information of the test cases, a 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.

Following the coverage criteria of white-box testing, this article uses 3 different coverage criteria to choose test case templates: decision coverage, condition coverage and path coverage. Among them, 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, and path coverage tries to cover every basic path of the program under test. The process of selecting the subset of test case templates based on specific coverage criteria is shown in Algorithm 1.

The input of Algorithm 1 is a set of test case templates *tcTemplate*, and the output is a set of selected test case templates *iniTemplates*. First, all the program elements contained in the test case template set *tcTemplate* are stored in the program elements set *Factors* (Line 1). Then, *template* and *count* are used to save the test case template that contains the maximum number of uncovered program elements and the number of the uncovered program elements that are covered by the current test case template (Lines 3-4). In lines 5-10, *template*, which can cover the maximum number of elements in *Factors*, is added to the selected test case template

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
```

iniTemplates. In lines 11-12, template is removed from the test case template set *tcTemplate*, and the program elements contained in *template* are deleted from *Factors*. At this time, the remaining uncovered program elements are stored in *Factors*. The above steps are repeated until *Factors* is empty, which means the test case templates in *iniTemplates* can cover all the program elements that can be covered by *tcTemplate*. Finally, the selected test case template set *iniTemplates* is output.

C. INITIAL SEED TEST-CASE SELECTION

After the test case templates are selected according to the specific coverage criteria, the corresponding initial seed test cases need to be selected. In the candidate test cases, multiple test cases can cover the same program elements. One test case can be randomly selected for each selected test case template to form a set of initial seed test cases.

The fuzz tester will validate the initial test cases after they are selected. Then, the validated initial seed test cases are mutated to obtain new test cases. By monitoring the execution of the program under test, the program path that is covered and the defects that are detected will be added to the test report.

The process of using initial seed test cases for fuzz testing is shown in Algorithm 2. First, the initial seed test cases in *Seeds* are validated by executing the program one by one. If an uncovered program path is covered during execution, then the test case is valid and is added to the effective seed test case set *effectiveSeeds* (Lines 2-5). Then, a seed test case *seed* is selected from *effectiveSeeds* to mutate and generate a new test case *newSeed* (Lines 8-9). If *newSeed* covers an uncovered path during execution, then the program path coverage information in the test report *testReport* is saved (Lines 10-12). If *newSeed* detects a defect during execution, then the defect is saved in the test report *testReport* (Lines 13-14). If the specified test time has been reached, then fuzz testing is ended. Finally, the test report of the program under test is output.

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 execution) 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 execution ) 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
```

III. EXPERIMENTS AND EVALUATIONS

To evaluate the approach of fuzz testing based on symbolic execution proposed in this article, we implemented the prototype tool CSEFuzz and conducted comparative experiments with Kelinci on a set of open-source programs in terms of the initial seed test case selection and validation, the number of paths covered and the number of defects detected.

A. TOOL IMPLEMENTATION

The CSEFuzz tool includes 3 modules: symbolic execution, initial seed test case selection and fuzz testing. Among them, the symbolic execution module collects path constraints and uses a constraint solver to generate test-case candidates with high path coverage. The initial seed test case selection module collects the corresponding coverage information of the candidate test cases by executing programs dynamically, generating test case templates according to the coverage information and selecting the initial seed test cases based on the test case templates. The fuzz testing module first validates the initial seed test cases and then mutates continuously to generate new test cases.

The symbolic execution module of CSEFuzz is implemented based on SPF.[3] The fuzz testing module is implemented based on Kelinci.[4] SPF is a symbolic execution tool for the Java programming language based on JPF [12]. SPF collects the constraint conditions of different paths in a program and uses constraint solvers to solve path constraints so that we can determine the feasibility of the program paths and corresponding test cases. Kelinci is a fuzz testing tool based on AFL for Java programs. Kelinci collects program execution information through instrumentation and uses genetic algorithms to generate new test cases iteratively to cover more program paths.

The development and experimentation environment of CSEFuzz is an Intel(R) Core(TM) i7-7700HQ, with 4 GB RAM running Ubuntu 16.04 and Sun JRE 1.8 (64-bit mode).

B. EXPERIMENTAL OBJECTS

In the experiment, test objects provided by Memoise[5] and SPF are used as the sources of experimental objects. The statistics tool cloc[6] is used to count the code lines of the test objects. In Table 1, the test objects provided by SPF are categorized with respect to their size. The test objects with sizes equal to or greater than 200 lines of code are chosen as experimental objects.

TABLE 1. SPF test object code line statistics.

Size of the Test Objects	Counts
Code lines >= 100	21
Code lines >= 200	6
Code lines >= 300	3

Memoise provided 7 test objects. Since the objects provided by Memoise and SPF both include Apollo and WBS, we selected 11 experimental objects in total. Among all of the experimental objects, the minimum and maximum number of Java files are 1 and 54, respectively, and the minimum and maximum sizes are 42 and 5957 lines of code. The statistic information of the selected experimental objects is listed in Table 2.

TABLE 2. The statistic information of experimental objects.

ID	Experimental Objects	Source	Count of Java Files	Line of Code
1	TCAS_V30	Memoise	1	152
2	BankAccount	Memoise/SPF	2	69
3	Apollo	Memoise	54	3448
4	MerArbiter-v2	Memoise	51	5957
5	LoopExample	Memoise	1	42
6	TwoLoopExample	Memoise	1	50
7	WBS	Memoise/SPF	1	215
8	rbt	SPF	2	679
9	TreeMapSimple	SPF	1	470
10	MathSin	SPF	1	290
11	fuzz/gram/test	SPF	3	226

C. EXPERIMENTAL DESIGN

During the experiment, we compared CSEFuzz and *afl-cmin* (*afl-cmin* is the command provided by Kelinci for selecting initial seed test cases) according to the following aspects: (1) time costs, by comparing the time costs of selecting the initial seed test cases and verifying the validity of seeds; (2) test efficiency, using initial seed test cases selected by different methods for fuzz testing and comparing the difference in the number of paths finally covered and the number of defects detected in a limited time.

To evaluate the influence of selecting initial seed test cases based on different coverage criteria, CSEFuzz implements four different coverage criteria. Among them, CSEFuzzn$_n$ indicates that no test case selection is performed, and all the test cases are used as seeds directly; CSEFuzz$_d$ indicates that the initial seed test cases are selected based on decision coverage; CSEFuzz$_c$ indicates that the initial seed test cases are selected based on condition coverage; and CSEFuzz$_p$ indicates that the initial seed test cases are selected based on path coverage.

In the experiment and evaluation, we plan to answer the following 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?

D. EXPERIMENTAL RESULTS AND EVALUATIONS

First, we counted the number of paths covered by Kelinci and CSEFuzz and the number of defects detected by Kelinci and CSEFuzz, as shown in Table 3. In the table, CSEFuzzn$_n$ indicates that the test cases generated by symbolic execution are used as the initial seed test cases for fuzz

[3]SPF, https://github.com/SymbolicPathFinder/jpf-symbc.
[4]Kelinci, https://github.com/isstac/kelinci.
[5]Memoise, https://userweb.cs.txstate.edu/~ g_y10/memoise/.
[6]cloc, https://github.com/AlDanial/cloc.

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.

testing directly. Considering the number of paths covered, for 3 experimental objects (*BankAccount*, *LoopExample*, and *fuzz/gram/test*), both Kelinci and CSEFuzz$_n$ cover the same number of paths. However, for the remaining experimental 8 objects (*TCAS_V30, Apollo, MerArbiter-v2, TwoLoopExample, WBS, rbt, TreeMapSimple,* and *MathSin*), CSEFuzz$_n$ covers 6, 1, 1, 3, 1, 30, 21, and 6 more paths than Kelinci. Considering the number of defects detected, for *BankAccount, MerAribiter-v2* and *WBS*, CSEFuzz$_n$ detects 2, 1, and 3 more defects than Kelinci, respectively.

Overall, the total number of paths covered by CSEFuzzn is 189, and the total number of paths covered by Kelinci is 120. CSEFuzz$_n$ covers 69 (57.50%) more paths than Kelinci. The number of defects detected by CSEFuzz$_n$ is 69, and the number of defects detected by Kelinci is 65. CSEFuzzn detects 4 (6.15%) more defects than Kelinci.**Therefore, the answer to RQ 1 is as follows: 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.**

Then, we compare the time costs of the test case selection for the different test methods. As shown in Table 4, 11 experimental objects are tested by using CSEFuzz with different test-case selection strategies and the *afl-cmin* command provided by Kelinci. The time costs for selecting and validating the initial seed test cases are compared.

In Table 4, CSEFuzzn does not perform initial seed test case selection, so the time cost of seed selection is 0. Considering the time cost of seed selection, CSEFuzz$_d$, CSEFuzz$_c$ and CSEFuzz$_p$ are reduced by 99.35%, 99.34% and 98.85% in comparison with *afl-cmin*, respectively. Considering the time costs of seed validation, CSEFuzz$_d$ costs the least amount of time, at 55.6 seconds. Compared with *afl-cmin*, the time costs of CSEFuzz$_d$ and CSEFuzz$_c$ in validating the seeds were reduced by 32.69% and 14.65%, respectively, while the time costs of CSEFuzz$_p$ and CSEFuzz$_n$ in validating the seeds increased by 4.1 times and 699.12 times.

Considering the total time costs of initial seed test-case selection and validation, CSEFuzz$_d$ costs 103.5 s, CSEFuzz$_c$ costs 119 s, CSEFuzz$_p$ costs 429.1 s, *afl-cmin* costs 7478.7 s, and CSEFuzz$_p$ costs 57747.6 s. Compared with *afl-cmin*, the total time costs of CSEFuzz$_d$, CSEFuzz$_c$, and CSEFuzz$_p$ were reduced by 98.62%, 98.41%, and 94.26%, respectively. CSEFuzz$_n$ increased by 7.72 times.

To analyze the reason for the time variations in the initial seed test-case selection and validation, Table 5 analyzes the number of initial seed test cases selected by the different methods. The data in Table 5 are organized by the number of initial seed test cases selected (the number of validated test cases). Taking *TCAS_V30* as an example, the number of initial seed test cases selected and validated by CSEFuzz$_c$ is 13(6), in which 13 represents the number of initial seed test cases selected, and 6 out of 13 are valid test cases. As shown in Table 5, CSEFuzz$_n$ selects the greatest number of initial seed test cases, which is 62549, followed by CSEFuzz$_p$, *afl-cmin*, CSEFuzz$_c$ and CSEFuzz$_d$ in descending order. Compared with CSEFuzz$_n$, the number of initial seed test cases selected by CSEFuzz$_d$, CSEFuzz$_c$, CSEFuzz$_p$ and *afl-cmin* decreased by 62,412, 62,349, 62,072 and 62,835, respectively. Compared with *afl-cmin*, the total number of initial seed test cases selected by CSEFuzz$_d$ and CSEFuzz$_c$ decreased by 27 and 9, respectively. CSEFuzz$_p$ increased by 313.

TABLE 5. Comparison of the number of initial seed test case selected and validated.

Experimental Objects	CSEFuzz$_n$	CSEFuzz$_d$	CSEFuzz$_c$	CSEFuzz$_p$	*afl-cmin*
TCAS_V30	589(11)	4(4)	13(6)	5(5)	10(10)
BankAccount	12(3)	2(2)	2(6)	2(2)	2(2)
Apollo	158(3)	2(2)	2(2)	2(2)	3(3))
MerArbiter-v2	296(5)	2(2)	2(2)	2(2)	4(4)
LoopExample	105(8)	2(1)	3(2)	2(2)	7(7)
TwoLoopExample	20403(17)	3(3)	5(4)	4(4)	10(10)
WBS	46(8)	9(6)	9(6)	12(7)	7(7)
rbt	39999(32)	8(8)	14(14)	293(30)	14(14)
TreeMapSimple	808(26)	12(11)	12(11)	62(23)	14(14)
MathSin	33(3)	2(2)	2(2)	2(2)	2(2)
fuzz/gram/test	1(1)	1(1)	1(1)	1(1)	1(1)
TOTAL	62459(117)	47(42)	65(56)	387(80)	74(74)

It can be seen from Table 5 that the initial seed test-case set selected by CSEFuzz$_n$ contains the greatest number of validated seed test cases, which is 117, followed by CSEFuzz$_p$, *afl-cmin*, CSEFuzz$_c$, and CSEFuzz$_d$ in descending order. Compared with CSEFuzz$_n$, the validated seed cases selected by CSEFuzz$_d$, CSEFuzz$_c$, CSEFuzz$_p$ and *afl-cmin* decreased by 75, 61, 37, and 43, respectively. Compared with *afl-cmin*, the number of validated seed test cases selected by CSEFuzz$_d$ and CSEFuzz$_c$ decreased by 32 and 18, respectively, while CSEFuzz$_p$ increased by 6.

From Table 4 and Table 5, which analyze the relationship between the number of initial seed test cases and the time cost of the test case validation, we find that the greater the size of the initial seed test cases, the greater the time cost for test case validation. Therefore, applying proper selection strategies to the initial seed test cases before validation can reduce the time cost of the initial seed validation and improve the efficiency of fuzz testing.

The answer to RQ 2 is as follows: Directly using test case candidates that are generated by symbolic execution will consume too much time for validating test cases. To improve test efficiency, proper selection strategies need to be applied to the initial seed test cases. Compared with the existing seed selection command *afl-cmin*, CSEFuzz$_d$, CSEFuzz$_c$ and CSEFuzz$_p$ can effectively reduce 98.62%, 98.41% and 94.26% of the time cost, respectively, for the initial seed test case selection and validation. Thus the efficiency of fuzz testing is improved by CSEFuzz.

TABLE 6. The number of paths covered by fuzz testing with different initial seed test cases.

Experimental Objects	CSEFuzzn	CSEFuzzd	CSEFuzzc	CSEFuzzp	*afl-cmin*
TCAS_V30	22	22	23	23	21
BankAccount	13	12	12	13	12
Apollo	8	8	8	9	9
MerArbiter-v2	12	12	13	12	12
LoopExample	11	11	11	12	11
TwoLoopExample	21	18	17	19	17
WBS	10	9	9	10	10
rbt	33	18	18	31	14
TreeMapSimple	27	18	18	25	17
MathSin	18	18	18	19	18
fuzz/gram/test	14	14	14	14	14
TOTAL	189	160	161	187	155

TABLE 7. The number of defects detected by fuzz testing with different initial seed test cases.

Experimental Objects	CSEFuzz$_n$	CSEFuzz$_d$	CSEFuzz$_c$	CSEFuzz$_p$	*afl-cmin*
TCAS_V30	21	37	34	38	21
BankAccount	4	3	3	3	4
Apollo	12	12	12	12	12
MerArbiter-v2	12	13	13	13	13
LoopExample	2	2	2	2	2
TwoLoopExample	4	4	4	4	4
WBS	6	6	6	6	6
rbt	3	3	3	3	3
TreeMapSimple	3	3	3	3	3
MathSin	2	2	2	2	2
fuzz/gram/test	0	0	0	0	0
TOTAL	69	85	82	86	70

Next, we use different coverage criteria to select initial seed test cases for conducting fuzz testing. The criteria are the number of paths covered and the number of defects detected by fuzz testing in a limited time, which is ten minutes (timing is started after test case validation is finished). The experimental results are listed in Table 6 and Table 7.

Table 6 lists the number of paths covered by fuzz testing with different methods in a limited time. It can be seen from Table 6 that CSEFuzz$_n$ covered the greatest number of paths, 189, followed by CSEFuzz$_p$, CSEFuzz$_c$, CSEFuzz$_d$, and *afl-cmin* in descending order. Compared with *afl-cmin*, the total number of paths covered by CSEFuzz$_d$, CSEFuzz$_c$, CSEFuzz$_p$, and CSEFuzz$_n$ increased by 5, 6, 32 and 34, respectively.

Table 7 shows the number of defects detected by fuzz testing with different methods in a limited time. It can be seen from Table 7 that the number of defects detected by CSEFuzz$_p$ is the largest, 86, followed by CSEFuzz$_d$, CSEFuzz$_c$, *afl-cmin* and CSEFuzz$_n$ in descending order. Compared with afl-cmin, the total number of defects detected by CSEFuzz$_d$, CSEFuzz$_c$ and CSEFuzz$_p$ increased by 15, 12 and 16, respectively.

In Table 7, for the experiment object *TCAS_V30*, the defects detected by CSEFuzz$_d$, CSEFuzz$_c$ and CSEFuzz$_p$

include 16 defects that were detected by the symbolic execution module. Meanwhile, *afl-cmin* and CSEFuzz$_n$ cannot detect these defects. This is because when testing *TCAS_V30*, the symbolic execution module found and generated 16 test cases that could trigger program errors. However, in the seed validation stage of *afl-cmi*n and CSEFuzz$_n$, which do not select test-case candidates, the initial seeds that can cause a program crash are deleted as invalid test cases. However, CSEFuzz$_d$, CSEFuzz$_c$ and CSEFuzz$_p$ need to execute test-case candidates and collect coverage information. Thus, the test cases generated by symbolic execution that can trigger crashes are validated by execution and are saved in the test report. Compared with *afl-cmin*, CSEFuzz$_d$, CSEFuzz$_c$ and CSEFuzz$_p$ can make better use of the information obtained by symbolic execution and detect 12-16 more defects in the experimental objects.

```
1    public static int Math(int x){
2        int IEEE_MAX = 100276;
3        if(x == IEEE_MAX){
4            assert(false);
5        } else {
6            x++;
7        }
8        return x;
9    }
```

FIGURE 3. Example of the relationship between path conditions and input variables.

The code snippet shown in Figure 3 is used to explain the why CSEFuzz covers more paths and detects more defects. The code snippet is adapted from the experimental object *MathSin*. In the code snippet, there are two paths and one defect (Line 4). The results of using symbolic execution to analyze the code are listed in Table 8. The symbolic execution covered two paths, detected one program defect, and generated two test cases for the two paths.

TABLE 8. The number of defects detected by fuzz testing with different initial seed test cases.

ID	Value of variable x	Path constraints	Defect Detected
1	1	x!=100276	No
2	100276	x==100276	Yes

Test cases generated by symbolic execution may directly trigger a program crash. When the test cases are provided to the fuzz tester Kelinci, if some test cases that can cause program crashes are included, then Kelinci will forcefully interrupt the test process and stop the fuzz testing. As a result, the initial seed test cases generated by symbolic execution that can trigger a program crash need to be filtered out before being sent to Kelinci. If *afl-cmin* is used to select the initial seed test cases, test case 2 in Table 8, which can cause a program crash, will be deleted from the initial seeds, and only test case 1 can be sent to the fuzz tester. Since test cases are randomly generated by fuzz testing, it is difficult to generate test cases that can satisfy the constraint condition "x==100,276." Therefore, if *afl-cmin* is used to process the initial seed test cases generated by the symbolic execution module, then the fuzz testing module can only cover one path in a limited execution time (ten minutes), and the defect in line 4 of Figure 3 cannot be detected.

To make full use of test cases generated by symbolic execution, CSEFuzz's initial seed test case selection module collects the coverage information corresponding to candidate test cases by dynamically executing the program. At the same time, the module checks whether a program defect is triggered. The test cases that can trigger program defects are validated and added to the test report. Therefore, CSEFuzz$_d$, CSEFuzz$_c$, and CSEFuzz$_p$ can detect the defects in Figure 3.

Compared with CSEFuzz$_n$, although the number of paths covered by CSEFuzz$_d$, CSEFuzz$_c$, and CSEFuzz$_p$ is reduced slightly, the time costs of the initial seed selection and validation are reduced, and more defects are triggered. Taking CSEFuzz$_p$ as an example, compared with CSEFuzz$_n$, the number of paths covered by CSEFuzz$_p$ decreases by 2, but the time costs of the initial seed test case selection and seed validation are reduced by 99.26%, and the number of detected defects is increased by 17. Compared with *afl-cmin*, CSEFuzz$_d$, CSEFuzz$_c$, and CSEFuzz$_p$ not only increase the number of paths covered and the number of defects detected but also reduce the total time costs of the initial seed selection and validation. Taking CSEFuzz$_p$ as an example, compared with *afl-cmin*, CSEFuzz$_p$ not only increases by 32 paths covered and 16 defects detected but also reduces the total time costs of the initial seed selection and seed validation by 94.26%.

he answer to RQ 3 is as follows: 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. The answer to RQ 4 is as follows: 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.

E. THREATS TO VALIDITY

In this subsection, we discuss the potential threats to the validity of our experimental studies.

The threats to internal validity are mainly uncontrolled internal factors that might influence the experimental results. The main internal threat is potential faults introduced during the implementation. To reduce this threat, we double-checked the implementation of the tools and the experiments. Moreover, we implemented CSEFuzz based on Kelinci and SPF to insure the correctness of the fuzz testing and symbolic execution.

Threats to external validity indicate whether the observed experimental results can be generalized to other subjects.

To alleviate this threat, the experimental objects are selected from both Memoise and SPF, which have been widely used. The size of the objects varies from 42 to 5957 lines of code. The performance and scalability of the CSEFuzz approach can be evaluated by using more projects.

Threats to construct validity are about whether the performance measures used in the experiments can reflect real-world situations. In this article, CSEFuzz is compared with Kelinci in terms of the time costs for seed selection and validation, the number of paths covered and the defects detected. In the future, CSEFuzz will be extensively compared with other publicly available fuzz testers for Java programs.

IV. RELATED WORK

This section summarizes the related work about fuzz testing and symbolic execution.

A. SYMBOLIC EXECUTION

Symbolic execution was proposed by King [7] for software analysis in 1975. The basic idea is to use symbolic values to represent input variables, convert them into symbolic expressions sentence by sentence in the process of running the program, collect the corresponding path constraints, and obtain test cases that meet the path constraints through the constraint solver. Initially, the path coverage of symbolic execution method was limited due to the capability of the constraint solver. In recent years, with the continuous improvement and optimization of symbol execution techniques, as well as the enhancement of hardware, symbol execution has been developed rapidly and is widely used in many fields.

Path explosion is one of the main factors that affect path coverage of symbol execution. The main solutions to path explosion are to optimize search algorithms and merge path states. KLEE [8] combines the optimal code coverage algorithm with the random path selection algorithm to achieve high code coverage. JDart [9] and SPF [10] both apply depth-first search algorithms. When dealing with loops, both SPF and KLEE restrict the search depth to prevent infinite loops. Godefroid [13] discussed the principle and problems of merging states. Merging redundant states can effectively reduce the number of paths to be searched, but new symbolic expressions may be introduced in constraint processing and parsing. In 2012, Kuznetsov *et al.* [14] proposed to automatically select states to be merged in symbol execution to improve the efficiency of symbol execution. In 2014, Avgerinos *et al.* [15] proposed the concept of verifesting (i.e., state fitting), which can reduce the state space of programs and improve the availability of dynamic symbol execution.

When the number of path constraints that need to be solved by the constraint solver is too complex, the symbol execution cannot cover the relevant paths. Therefore, EXE [16] proposed a constraint independence optimization method. The basic idea is to divide the set of constraints into multiple independent subsets of constraints and discard irrelevant constraints to simplify the constraint set and improve the efficiency of constraint solving.

Because the precision of floating-point operations in programs is limited and the results of floating-point operations may be inconsistent in different system environments, it is difficult for the constraint solver to deal with constraints related to floating-point operations. Therefore, Botella *et al.* [17] designed a special solver for floating-point operations, but it cannot handle constraints mixed with other data types. Other approaches for solving constraints with floating-point operations have been proposed, for instance, Collingbourne *et al.* [18] presented KLEE-FP for cross-checking an IEEE 754 floating-point program and its SIMD-vectorized version as an extension to KLEE.

In this article, symbolic execution is used to generate initial seed cases for fuzzy testing. The improvements in symbolic execution techniques can be helpful in improving the effectiveness of CSEFuzz.

B. FUZZ TESTING

Fuzz testing has been widely used as a software defect detection technology. The basic idea of fuzz testing is to generate an input randomly, monitor whether the input can trigger a program crash during execution, and thus find defects in the program. To improve the efficiency of fuzz testing, researchers have explored various solutions.

The early fuzz testing technique was simply a random testing technique that found errors in many programs. In 1989, Miller *et al.* [19] developed the first fuzz testing tool, Fuzz, by inputting random generated data into software on the UNIX system. Fuzz could crash more than 25-33% of the utility programs on the UNIX system. In 2005, iDefense company developed the file format fuzz testing tool FileFuzz [20] based on the Windows platform. COMRaider[7] and AxMan[8] were proposed for testing COM object interfaces.

After 2011, many related researchers combined machine learning algorithms with fuzz testing to improve the efficiency of generating test cases [21], such as the famous fuzz testing engine AFL. AFL is a security-oriented gray-box fuzz testing tool that uses genetic algorithms and byte-level operations to change the input provided by the user. In 2019, Zhou *et al.* proposed InsFuzz [22], which uses static analysis technology to infer the bytes that will affect the comparison instructions and called them key bytes. During execution, InsFuzz analyzes key bytes and comparison instructions to determine which bytes are worth mutating and how to mutate them so that it can detect more defects and cover more code. In 2018, Caroline Lemieux and Koushik Sen implemented FairFuzz [23] based on AFL. FairFuzz can automatically identify branches that can only be executed with rare inputs and has a novel mutation algorithm that allows for test cases to be mutated with a greater probability of executing rare

[7]COMRaider, http://sandsprite.com/iDef/COMRaider/
[8]AxMan ActiveX Fuzzer, https://www.hdm.io/tools/axman/

branches. Experiments showed that FairFuzz can achieve higher code coverage faster than AFL.

Taint analysis was also used to track taint data flows to obtain program information [24]–[27]. In 2018, Chen and Chen [28] proposed Angora to explore the state of a program by solving path constraints without using symbolic execution. Angora traces unexplored branches and tries to resolve path constraints on these branches. Angora performs dynamic taint tracing on the input and records into which conditional statements the input bytes flow. Then, Angora only mutates the bytes corresponding to the conditional statements that need to be covered so that it can reduce the cost of taint analysis. At the same time, new test cases can be quickly generated to cover new branches.

In 2018, Noller *et al.* [29] proposed Badger, which uses a hybrid test method of symbolic execution and fuzz testing to achieve high code coverage. Badger detects a special kind of defect that occurs when the program's worst time complexity or space complexity far exceeds the average level. Stephens *et al.* [30] proposed Driller to combine Angr [31] and AFL. The program under test is first tested by the fuzzer, and then, the dynamic symbolic execution engine is invoked when the fuzzer cannot find more paths. If new paths were covered, then the corresponding test input is provided to run the fuzzer again.

The CSEFuzz proposed in this article only needs to run symbolic execution once. Traditional white-box fuzz testing uses symbolic execution to solve when the fuzz testing encounters constraints that cannot be covered. During the entire testing process, symbolic execution needs to be performed multiple times. In addition, various coverage criteria are proposed to reduce the number of initial seed test cases generated by symbolic execution and improve the efficiency of fuzz testing.

V. CONCLUSION AND FUTURE WORK

The quality of initial seed test cases is an important factor that affects the coverage and defect detection capability of fuzz testing. In view of this issue, this article used symbolic execution to generate candidate test cases and then selected initial seed test cases for fuzz testing from the candidate test cases according to specific coverage criteria to improve the efficiency of fuzz testing. Based on the proposed approach, this article implemented the prototype tool CSEFuzz, compared it with Kelinci in terms of the time costs for initial seed selection and validation, the number of paths covered and the number of defects detected on a set of experimental objects provided by Memoise and SPF. The experimental results show that CSEFuzz not only greatly reduces the time costs of initial seed selection and validation but also improves the number of paths covered and defects detected. In the future, we will improve the efficiency of initial seed selection and use CSEFuzz to test more projects to verify the scalability of CSEFuzz.

ACKNOWLEDGMENT

Parts of this work are based on the master's thesis of the first author [32].

REFERENCES

[1] T. L. Munea, H. Lim, and T. Shon, "Network protocol fuzz testing for information systems and applications: A survey and taxonomy," *Multimedia Tools Appl.*, vol. 75, pp. 14745–14757, 2016.

[2] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "DroidFuzzer: Fuzzing the Android apps with intent-filter tag," in *Proc. Int. Conf. Adv. Mobile Comput. Multimedia (MoMM)*, New York, NY, USA, 2013, pp. 68–74.

[3] E. Barsallo Yi, A. Maji, and S. Bagchi, "How reliable is my wearable: A fuzz testing-based study," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Luxembourg City, Luxembourg, Jun. 2018, pp. 410–417.

[4] X. Xie, S. See, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, and J. Yin, "DeepHunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, Beijing, China, 2019, pp. 146–157.

[5] T. Yue, Y. Tang, B. Yu, P. Wang, and E. Wang, "LearnAFL: Greybox fuzzing with knowledge enhancement," *IEEE Access*, vol. 7, pp. 117029–117043, 2019, doi: 10.1109/ACCESS.2019.2936235.

[6] J. Li, B. Zhao, and C. Zhang, "Fuzzing: A survey," *Cybersecurity*, vol. 1, no. 1, pp. 6–19, Dec. 2018.

[7] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[8] S. Michel, P. Triantafillou, and G. Weikum, "KLEE: A framework for distributed top-K query algorithms," in *Proc. 31st Int. Conf. Very Large Data Bases*, Trondheim, Norway, 2005, pp. 637–648.

[9] K. Luckow, M. Dimjaevi, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamaric, and V. Raman, "JDart: A dynamic symbolic analysis framework," in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Germany: Springer, 2016, pp. 442–459, doi: 10.1007/978-3-662-49674-9_26.

[10] C. S Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic pathfinder: Integrating symbolic execution with model checking for java bytecode analysis," *Automated Softw. Eng.*, vol. 20, no. 3, pp. 391–425, Sep. 2013.

[11] R. Kersten, K. Luckow, and C. S Păsăreanu, "POSTER: AFL-based fuzzing for Java with kelinci," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Oct. 2017, pp. 2511–2513.

[12] W. Visser, C. S Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, Boston, MA, USA, 2004, pp. 97–107.

[13] P. Godefroid, "Compositional dynamic test generation," in *Proc. 34th Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 2007, pp. 47–54.

[14] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, 2012, pp. 193–204.

[15] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," *Commun. ACM*, vol. 59, no. 6, pp. 93–100, May 2016.

[16] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12 no. 2, pp. 1–38, 2008.

[17] B. Botella, A. Gotlieb, and C. Michel, "Symbolic execution of floating-point computations," *Softw. Test., Verification Rel.*, vol. 16, no. 2, pp. 97–121, 2006.

[18] P. Collingbourne, C. Cadar, and P. H. J. Kelly, "Symbolic crosschecking of floating-point and SIMD code," in *Proc. 6th Conf. Comput. Syst.* New York, NY, USA: Association Computing Machinery, 2011, pp. 315–328.

[19] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[20] M. Sutton and A. Greene, "The art of file format fuzzing," in *Proc. BlackHat USA Conf.*, 2005, pp. 1–31.

[21] L. Cheng, Y. Zhang, Y. Zhang, C. Wu, Z. Li, Y. Fu, and H. Li, "Optimizing seed inputs in fuzzing with machine learning," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion Proc. (ICSE-Companion)*, Montreal, QC, Canada, May 2019, pp. 244–245.

[22] H. Zhang, A. Zhou, P. Jia, L. Liu, J. Ma, and L. Liu, "InsFuzz: Fuzzing binaries with location sensitivity," *IEEE Access*, vol. 7, pp. 22434–22444, 2019.
[23] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, Montpellier, France, 2018, pp. 475–485.
[24] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A taint based approach for smart fuzzing," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation*, Montreal, QC, Canada, Apr. 2012, pp. 818–825.
[25] Y.-H. Choi, M.-W. Park, J.-H. Eom, and T.-M. Chung, "Dynamic binary analyzer for scanning vulnerabilities with taint analysis," *Multimedia Tools Appl.*, vol. 74, no. 7, pp. 2301–2320, Apr. 2015.
[26] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2011, pp. 427–430.
[27] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *Proc. 4th IEEE Int. Conf. Softw. Test., Verification Validation*, Berlin, Germany, Mar. 2011, pp. 1–14.
[28] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2018, pp. 711–725.
[29] Y. Noller, R. Kersten, and C. S Păsăreanu, "Badger: Complexity analysis with fuzzing and symbolic execution," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, Amsterdam, Holland, 2018, pp. 322–332.
[30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2016, pp. 1–16.
[31] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Diego, CA, USA, May 2016, pp. 138–157.
[32] Z. Xie, "Detecting software defects based on fuzz testing and symbolic execution," M.S. thesis, Comput. School, Beijing Inf. Sci. Technol. Univ., Beijing, China, 2020.

ZHANGWEI XIE received the master's degree in computer technology from Beijing Information Science and Technology University, in July 2020. His research interest includes software analysis and testing.

ZHANQI CUI (Member, IEEE) received the B.E. and Ph.D. degrees in software engineering and computer software and theory from Nanjing University, in 2005 and 2011, respectively. He was a Visiting Ph.D. Student with the University of Virginia, from September 2009 to September 2010. He is currently an Associate Professor with Beijing Information Science and Technology University. His research interest includes software analysis and testing.

JIAMING ZHANG received the bachelor's degree in software engineering from Beijing Information Science and Technology University, in July 2020, where he is currently pursuing the master's degree. His research interest includes software analysis and testing.

XIULEI LIU received the Ph.D. degree in computer science from the Beijing University of Posts and Telecommunications, in March 2013. He was a Visiting Ph.D. Student with CCSR, University of Surrey, from October 2008 to October 2010. He is currently an Associate Professor with Beijing Information Science and Technology University. His research interests include semantic sensor, semantic web, knowledge graph, and semantic information retrieval.

LIWEI ZHENG received the Ph.D. degree in computer software and theory from the Academy of Mathematics and Systems Science, Chinese Academy of Sciences, in 2009. He is currently an Associate Professor with Beijing Information Science and Technology University. His research interests include requirement engineering and trusted computing.