CS453 - Automated Software Testing

FAULT INDUCING TEST CASE GENERATION WITH DELTA-DEBUGGING

Muhammad Irfan Akbar 20180854
Irfan Ariq 20204831
Fathony Achmad 20180825
Muhammad Waleed Anwar 20170851

22 June 2021

I. Abstract

There are many ways to find faults in a program, and one of the ways is generating test cases that trigger the fault. We propose a new test case generation technique that leverages delta debugging to generate the set of test cases. We introduce a new metric that is the percentage of fault-inducing input (PFI) as the metrics of our test case generator. The idea behind PFI is that a set of test cases with a higher PFI tends to trigger more faults in a program. Therefore, our proposed technique aims to produce a set of test cases with a high number of PFI.

We evaluate our proposed technique against fuzzer with statement coverage. We compare the average of PFI on a set of test cases generated by our proposed technique and a set of test cases generated by a statement coverage fuzzer. The result shows our proposed technique fails to produce higher PFI compared to the statement coverage fuzzer.

Our Github repository: https://github.com/mirfana23/CS453-Final-Project

II. Introduction

Recent years have seen the rise of a practice in the industry and academia, known as the programming exam. The idea is that candidates are given a/a set of problem(s) and are told to solve them within a given time period. Instead of testing these problems using a set of fixed test cases that candidates can adapt to as opposed to actually writing a correct program, the idea is to use some test cases as a seed to generate similar test cases. This not only adds a degree of randomness that the candidates cannot predict, but by virtue of randomness, may end up testing some other fault/aspect of the program that was not included in the original test cases.

However, test case generation, depending on the task at hand, can be quite a computationally heavy process. In order to trim this as much as possible, we aim to first, for lack of a better word, optimize the test case inputs by isolating the fault-inducing part using Delta Debugging (DD). We believe that by removing the irrelevant parts of the test case, we will generate more “concentrated” test cases, in terms of the percentage of fault-inducing input (the ratio of the fault-inducing input with respect to the overall length of the test case). The idea is that the higher the percentage, the more we are testing with each test case.

Our proposed method will, therefore, only accept the generated test cases with the same or higher fault-inducing percentage. We will compare the test cases generated with and without applying DD to them first, to see whether it results in more, and or, better test cases generated.

III. Related Works

The delta debugging used is roughly the same as taught and tested in the

course. Our work, at least that particular part of it, is thus based on the three papers that were included in the assignment as related reading. As for fuzzing we base our work on The Fuzzing Book, the Almanac for all things related to the subject.

IV. Method

IV.I. Overview

From the initial set of test cases that we generate randomly, we will perform a delta debugging to determine the minimum failure-inducing input. Furthermore, we performed our modified fuzzing technique by tracking the percentage of fault-inducing input, in the hope to find more faulty test cases that have bigger fault-inducing input percentages and therefore make a better test cases suite.

To evaluate the performance of our proposed technique, we compared the result of our modified fuzzer performance with the normal fuzzer that uses a normal code coverage. Therefore, we built the normal fuzzer and the evaluator to compare the performance of both fuzzers.

We will compare the average percentage of fault-inducing input of each test case generated by both the normal fuzzer and our proposed test case generator. The average percentage of fault-inducing input will measure how fast our test case generation generates such output in comparison to the normal fuzzer.

Last, a fault is not always coming from a program that results in an error. For example, a program that is working but produces an undesired output is also a faulty program. Therefore, we have to test our implementation with programs that have a clear oracle. For Instance, a program that has a reference solution. Hence, we mostly obtained the program from a competitive programming website.

IV.II. Preliminary Knowledge

Before discussing further the programs that we make. There is one terminology that we will use a lot when discussing our method. We are going to use a percentage of fault-inducing input when talking about the method and also the result.

The percentage of fault of inducing input is a measurement of input that causes the fault to total lengths of input. Basically, we count how many inputs cause a fault with the help of delta debugging and divide them to the total lengths. For example, given an input that has a length of ten, and after delta debugging, the input is reduced to a length of four. Therefore, the percentage of fault-inducing input will be 40%.

IV.III. Baseline Test Case Generation

We will later compare the performance of our proposed test case generation with the baseline that is a normal fuzzer. Therefore, we made our own normal fuzzer that utilized statement coverage.

```
# 4. start fuzzing
i = 0
print("START TO RUN THE FUZZER")
start_time = time.time()
timer = 60
while True:
    if timer < (time.time() - start_time):
        break
    new_input = input_fuzzer.get_fuzzed_input()
    str_new_input = str(new_input)
    # pass_fail, raise_cov, total_cov = pyscript_runner.run_check([new_input])
    pass_fail, raise_cov, total_cov = pyscript_runner.run_check([str_new_input])
    if pass_fail == ScriptRunner.FAIL:
        input_fuzzer.add_crash(new_input)
    elif raise_cov:
        input_fuzzer.add_input(new_input)
    print("#{} - input:'{}' - total_cov:{}".format((i+1), new_input, total_cov))
    i += 1
```

IV.IV. Proposed Test Case Generation

We can further divide this part into two main implementations

Proposed test case generator

Here is the step of the test case generator:

1. Run the fuzzer until it gets a fault inducing input
2. Record the percentage of fault inducing input
3. If the percentage is equal to or above the previous threshold, insert the test input into the queue
4. Set the percentage threshold accordingly
5. Repeat from (1) with the percentage threshold set in (4) and run the fuzzer in a certain runtime

By doing such steps, we got a set of test cases that are readily available to be used in the comparison performance later.

Runner

The task of the runner is to run the proposed test case generator to the faulty program. Furthermore, an input is considered faulty if it either makes the program produce an error or makes the program produce an undesired result. Therefore, the runner will also run the reference program alongside the target program and compare the output from both programs. If the outcome is different the input indeed makes the target program faulty.

```
current_time = time.time()
#set fuzzer timer
timer = 60

#fuzzing process
while True:
    elapsed = time.time() - current_time

    if timer < elapsed:
        break

    inp = random_fuzzer.fuzz()
    result, outcome = mystery.run(inp)
    if outcome == mystery.FAIL:
        results.append(result)
        dd_reducer = DeltaDebuggingReducer(mystery, log_test=False)
        ddres = dd_reducer.reduce(result)
        pfires = fault_percentage(ddres, result)
        if pfires >= maxpfi:
            push_queue(pfires, result)
```

IV.V. Comparison with the Normal Fuzzer

The last step before the evaluation is to compare the test cases generated by both tests case generators. We did it by first calculating the average percentage of fault-inducing input generated by both methods.

```
# 3. reduce the testcase
for test in base_seed:
    result, outcome = mystery.run(test)
    if(outcome == mystery.FAIL):
        dd_reducer = DeltaDebuggingReducer(mystery, log_test=False)
        ddres = dd_reducer.reduce(result)
        pfires = fault_percentage(ddres, result)
        base_pfi.append(pfires)
for test in proposed_seed:
    result, outcome = mystery.run(test)
    if(outcome == mystery.FAIL):
        dd_reducer = DeltaDebuggingReducer(mystery, log_test=False)
        ddres = dd_reducer.reduce(result)
        pfires = fault_percentage(ddres, result)
        proposed_pfi.append(pfires)


# 4. calculate and present the calculated pfi
base_avg_pfi = 100*sum(base_pfi)/len(base_pfi)
prop_avg_pfi = 100*sum(proposed_pfi)/len(proposed_pfi)
print("Average percentage of fault inducing input (baseline fuzzer) : %.2f%%" %(base_avg_pfi))
print("Average percentage of fault inducing input (proposed fuzzer) : %.2f%%" %(prop_avg_pfi))
```

Second, we also compared the time needed for both fuzzer to get the fault of the program. We will use both aspects to generate our result in the latter part.

V. Result

V.I. Research Question

- How fast does the proposed test case generation can induce fault?
- How does it compare with the Statement Coverage fuzzer?

V.II. Result

- How fast does the proposed test case generation can induce fault?

In evaluating our proposed test case generation, we first measure how fast our program can induce the first fault input. The result of the first fault input found of some target program are summarized in this table:

Target program	**Time to first Fault (s)**
Dummy.py	0.074
Flatten.py	0.078
Get_factors.py	0.17
Lis.py	0.088
Next_permutation.py	0.086
Quicksort.py	0.11

As can be seen from the table above, our test case generation can find faults in a reasonable time, with consideration of the size of the program. The first fault input found can be crucial in finding other fault input.

- How does it compare with the Statement Coverage fuzzer?

We compare our test case generation program with the widely used fuzzer that is based on statement coverage. The result of the comparison of the 2 methods is summarized in this table:

Test Program	**Baseline Average PFI (%)**	**Proposed Average PFI(%)**
Get_factors.py	59.85	5
Lis.py	45	15
Next_permutation.py	45.56	10
Quicksort.py	19.09	10

The average percentage of fault-inducing input will measure how fast our test

case generation produces the expected test case (test cases with lots of fault-inducing input), in comparison to the statement coverage fuzzer.

As can be seen from the above table, the baseline fuzzer (i.e. the statement coverage fuzzer) still outperforms the proposed test case generation. The result may be caused by the quality of the base seed used for the baseline fuzzer, and also the number of test cases that are produced by respective techniques.

V.III. Future Work

Currently, in executing our work, we need to make manual adjustments according to the test program. In the future, we should automate it to create adjustments with the input format of the test program. We can do so by parsing the input into the data structure required by the program. In addition, the execution of the program needs to be improved. Currently, multiple executions are done in reducing the test input to find the fault including part. The execution cost may not matter much in smaller programs we have provided but will have a significant impact on bigger , more complex programs. Lastly, we believe that a better solution needs to be found, as the proposed test case generation has shown no improvement in comparison to the statement coverage fuzzer. Hypothetically, we may be able to do so by changing to a smarter method of our input generation rather than random input.

VI. Conclusion

We proposed a test case generation technique and introduced PFI as the metrics. We use delta debugging to calculate the PFI of the generated test case. The result shows that our proposed technique produces less PFI on all target subjects compared to fuzzer with statement coverage. Therefore, our proposed technique fails to outperform fuzzer with statement coverage.

VII. Reference

- A. Zeller. Yesterday, my program worked. Today, it does not. Why?. in Proc. Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7), Toulouse, France, September 1999.
- L. Kirschner, E. Soremekun, A. Zeller. Debugging Inputs. ICSE ’20, May 23–29, 2020, Seoul, Republic of Korea
- Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler: "Tours through the Book". In Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler, "The Fuzzing Book", https://www.fuzzingbook.org/html/Tours.html. Retrieved 2021-06-21
- A. Zeller, R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering Volume 28 Issue 2. February 2002. pp 183–200