Automated testing

Ka-Wing Man 4330714 k.w.man@student.tudelft.nl
Mitchell Olsthoorn 4294882 M.J.G.Olsthoorn@student.tudelft.nl

March 20, 2017

1 Introduction

In today's world, software is everywhere. All companies uses software to keep track of their business. Their use includes for administration, making complex calculations such as optimization problems or even as a communication tool. What is inevitable in software are bugs. Bugs are flaws in the computer software that produces unexpected and incorrect results. These bugs might not harm your software if they are not triggered or if the impact of the bug is not big. But some bugs harmful enough to break the software. If software are not doing what they are intended to do (because of bugs), this may then result in company losses due to refunds and wasted time developing these software. These bugs are easily made by human errors, but they are hard to find when testing and debugging. That is why developers have created tools to make finding bugs easier by doing it automated.

In this report, we study the automated methods that generates test cases for the analyzed code. These test cases can in turn be used to test the program.

2 Fuzzing

Fuzzing is a automated testing method to find hidden bugs in programs. Normally when programmers test for bugs in their software, they go through their code and try to find input values that the system can be tested on. These values normally include default edge cases such as negative numbers, empty strings or a null object. This method is however very time consuming and tedious. So most tests only consist of the standard inputs.

Fuzzing tries to automate this process for the programmers by using a White-, Gray- or Black Box software testing technique. White Box techniques assume that the fuzzing program has clear access to the code. A Black Box is a piece of program that you can run, pass input and receive output, but you can not see what it does. This is one use case where fuzzing can also help where normal testing can not be applied. A Gray Box testing technique is a combination of a White Box and a Black Box technique. Normally fuzzers work using the Black Box testing technique, but the fuzzer can use a White- or Gray- Box method to speed up the process of finding out all the branches (different choices in conditional statements) in the code.

Fuzzing works by using malformed/semi-malformed data injection. How this data is gathered depends on how smart the fuzzer is. A very dumb fuzzer uses purely random values as the only input. This could be for example piping /var/random into the program. A dumb fuzzer requires very little work, but this little work can produce a lot of results. That is one of the big advantages of fuzzing. However most of the time the input to a program is a little more structured, so an effective fuzzer would only generate semi-valid inputs. Meaning that the input is valid enough that it is not directly rejected by the parser because it is the wrong kind of input, but still generates enough variation to cause crashes in the program.

Generally there are two kinds of fuzzers that can be distinguished: Mutations and generators. Mutators take a list of accepted valid inputs, on which it can run itself. It then uses random mutations to alter these inputs to generate new ones. This would be an example of a fuzzer that is in between smart and dumb. It generates much better results than a dumb fuzzer, but it misses the intelligence of the syntax of the input. Generators on the other hand are usually smarter. The generator will try to find out the inner workings of the programs and generate sensible input. A generator can also use static fuzzing vectors (values that are know to be dangerous for the program). The output of this data is then used as the input to the program.

Both these types of fuzzers then watches the output of the Black Box for unexpected results, which are called crashes in fuzzing. Each crash could be a possible bug in the program. Because this whole process is automated, the fuzzing technique can test a vast amount of possibilities that could ever have been tested by the conventional way of testing.

There is also a new experimental type of fuzzer called: evolutionary fuzzer. This type of fuzzer uses heuristically and machine learning to find out what the program does and find the right values to use as inputs. Currently there is no implementation of a evolutionary fuzzer.

A simple example of what a fuzzer might do is, when you have for example a program that takes as input a choice of one of the three answers to a question. The valid inputs would then be 0, 1 or 2. The fuzzer will look at the inputs ans tries in this case different integers, for example 3, 100 or -1. Because integers are stored in a fixed size variable, any of these values might be possible inputs. If the conditional statements used inside the program might not be implemented properly the program may crash. The fuzzers generator could also give strings as a possible input.

3 Concolic Execution

In software testing, there are several techniques to generate test input values to test your program.

In one of the techniques called concrete execution, random values are chosen for the variables in the program. This results in that some paths of branches may not be taken with the random values.

Another technique called symbolic execution, on the contrary, generates a set of concrete input values according to a set of rules. It works as the following: Symbolic execution maintains a symbolic state σ and a path constraint PC. In σ, variables in the program being analyzed are mapped to symbolic expressions and PC is a quantifier-free first-order formula over the symbolic expressions. The symbolic state σ starts with an empty set of mapping and PC starts with the initialization of *true*. Both σ and PC are updated throughout the symbolic execution following these rules:

- For every read-statement var = read_input(), add the mapping $\{var \mapsto s\}$, where s is a unconstrained symbolic value and read_input() is a function that receives input.
- For every variable assignment var = e, add the mapping var to $\sigma(e)$, obtained by evaluating e to the current symbolic state σ.
- For every conditional statement if(e) then S else R, PC is updated to $PC \wedge \sigma(e)$ ("then" branch). A second path constraint PC' is created and updated to $PC \wedge \neg\sigma(e)$ ("else" branch).

```
1: x = read_input();
2: y = read_input();
3:
4: z = 2 * y;
5: if (z == x) {
6:         if (x > y + 6) {
7:         output("error");
8:         }
9: }
```

Figure 1: Example code

Figure 1 is used as an example to illustrate execution of symbolic execution. After the first two lines, The symbolic state is updated to $\sigma = \{x = x_0, y = x_0\}$. In line 4, the symbolic state is updated once again to $\sigma = \{x = x_0, y = y_0, z = 2 * y_0\}$. After that, the if-statement in line 5 yields the path constraints $(2 * y_0 = x_0)$ (if the if-statement yields true) and $(2 * y_0 \neq x_0)$ (if the if-statement yields false). The if-statement starting from line 6 creates two new path constraints $(2 * y_0 = x_0) \wedge (x_0 > y_0 + 6)$ and $(2 * y_0 = x_0) \wedge (x_0 \leq y_0 + 6)$. Note that $(2 * y_0 = x_0)$ comes from the path constraint of when the if-statement of line 5 yields true and therefore it will be further used for the path constraints inside the loop. The end result yield the symbolic state

$\sigma = \{x = x_0, y = y_0, z = 2 * y_0\}$ and the path constraints $(2 * y_0 \neq x_0)$, $(2 * y_0 = x_0) \wedge (x_0 > y_0 + 6)$ and $(2 * y_0 = x_0) \wedge (x_0 \leq y_0 + 6)$. After the symbolic execution instance exits or hits an error, that instance will get terminated and a constraint solver will solve the path constraints. The satisfying assignment are then generated as test inputs for the analyzed program. The program will take exactly the same paths as the symbolic execution and terminates in the same way.

Symbolic execution as described above is often called classical symbolic execution. A modern use of symbolic execution is mixed with use of concrete execution. This technique is called Concolic execution. It maintains both a concrete and a symbolic state. The concrete state maps all variables to their concrete value and the symbolic state only maps variables with symbolic values, just like symbolic execution. Since concolic execution needs to maintain a concrete state, it needs concrete initial values for the input. Concolic execution starts with given or random initial values, updates the symbolic constraints at every if-else-then-statement. Finally, it uses the constraint solver to infer variants of the previous inputs. This is done so alternatives paths may be taken during the next executions. The whole process is repeated until all paths are taken.

Figure 1 is once again used as an example to illustrate the steps of concolic execution. First, concolic execution gives two random values to x and y. Let's say x = 5, y = 17. In the if-statement in line 5, the result of the boolean expression $z == x$ yields false because z = 34 and x = 10. This generates the path constraint $x_0 \neq 2y_0$. It hits an exit, because no else-statement is available and it is the end of the code. Now, because the if-statement in line 5 resulted in false, for the next execution, the result of $z == x$ has to become true. The path constraint is therefore negated to $x_0 = 2y_0$. The solver solves this constraint and may come up with two values like {x = 4, y = 2}. For the second execution, {x = 20, y = 10} will be used. This time, the if-statement in line 5 yields true. Line 6 will be executed and the path constraints are updated to $(x_0 \neq 2y_0) \wedge (x_0 > y_0 + 6)$. The result of the if-statement $(x > y + 6)$ yields true. The program outputs ERROR and end. For the third execution, the path constraint will be negated to $(x_0 \neq 2y_0) \wedge (x_0 \leq y_0 + 6)$. A solver may solve this as x = 2, y = 1 and will proceed like above. After the third execution, concolic execution will report that all path has taken once and will terminate test input generation.

4 AFL and Klee

AFL

AFL fuzzer is a security-foccussed fuzzer that uses genetic algorithms to find interesting test cases. In comparison to other fuzzers AFL is very practical. It is built to have very little overhead by using highly effective minimalistic fuzzing strategies that do not require any configuration, but can still handy complex and real-word use cases. AFl has been used to find bugs in big software projects like OpenSSL, PHP and Bash to name a few.

AFL two things to work: a function to execute the program and a sample input. With these two pieces of information it tries to run the functions with the sample input. If this is successful it is going to alter the input to a smaller form a tries if it is successful. After the fuzzer derives the smallest input that still makes the function succeed, it will begin the process of fuzzing by applying numerous modifications to the derived input. If the output of these runs process a hang (program takes longer than a specified timeout) or a crash, it could have discovered a bug or a path through the program that is not supposed to be there. This input is saved so it can be examined later.

To make it easier for AFL to conclude what the program is doing with the supplied input, it provides a compiler that when used insert snippets of code to track control flow. If this is not possible because the code is not open a Black Box approach is used.

Klee

KLEE is a tool used to apply symbolic execution on code. The goal of KLEE is to run every line of code in the program that is being analyzed. it also checks each line against all possible values to find if certain inputs can trigger errors. KLEE's basic strategy is to replace a programs input with symbolic variables. Initially, there are no constraints for those values. If KLEE hits a conditional statement, it forks the execution and both executions has their own constraints to the path conditions. It has the ability to check each dangerous operation like assertions and memory accesses. Unlike other tools, KLEE does not require source modifications or any manual work that the user needs to do, other than giving a command in the command line.

5 Experiments of AFL and Klee

To compare the two methods of automated testing of code, different experiments have been conducted to see if one gives better results than the other. For these experiments we used the c (c-code) problems from the RERS 2017 reachability problems collection, specifically problems 10, 11 and 12. These problems can be downloaded from http://rers-challenge.org/2017/problems/training/RERS17TrainingReachability.zip and are highly obfuscated. In the two paragraphs below there is explained how these problems where executed using the different testing techniques.

AFL

To use AFL for testing these problems, first AFL of course needs to be installed. As stated in the previous section about the working of AFL the code needs to provide a function for it to run and a sample file to pass to the program. To comply with these requirements, the code was adapted to have a main function compatible with AFL and also to provide a way for AFL to see if an crash has occurred. Instructions for these modifications can be found at: https://github.com/TUDelft-CS4110-20162017/syllabus/blob/master/AFL_Fuzzing.md. To start fuzzing the problems first needed to be compiled using the AFP compiler: afl-clang. To give the fuzzer a place to store and read its information two folders were created one for storing the inputs and one for storing the crashes that were found. After that the fuzzing can start by executing afl-fuzz with the appropriate arguments. After two hours of running non-stop the program was stopped to examine the results.

Klee

Docker has been installed for the use of KLEE. Docker is tool used as a container to deploy applications. The containers are isolated from each other and the underlying system. Klee is one of the applications that can be deployed in a Docker container. After the installation of Docker and getting the KLEE Docker image, a KLEE Docker container can be created by running the following command on a Linux machine:

```
$ docker run --rm -ti --ulimit='stack=-1:-1' klee/klee
```

After this, a container is created and KLEE can be used within the container. First the RERS reachability problem code need to be compiled using Clang. After that, KLEE is executed on the compilation. The execution of KLEE took minutes to hours to complete.

6 Results

6.1 Output

AFL

Inside the results folder of the AFL process there are multiple files and folders. There are two files: fuzzer_stats and plot_data. This first gives general statistics of the run and the second file makes it possible to plot the findings of the crashes and hangs. There are also 3 folders: crashes, hangs and queue. The folder crashes contains the unique crashes found by the run. The folder hangs contains the hangs detected during the run. The folder queue contains interesting test case that caused the crashes and hangs. To check which crashes are found, they can be put through the program and the corresponding error will show. This can be used to identify which specific part of your application crashed.

KLEE

KLEE outputs a folder called klee-out-x, where x is a number incremented for each execution of KLEE. The most important produced outputs are the .ktest files in the folder. They contain the generated input values from the KLEE execution. K-test tool can be used to inspect these files. They can be used to replay inputs.

6.2 Comparison

AFL has found some errors more in the higher range. This could be because AFL ran faster and KLEE was stopped before it could discover those. For the difference in the lower range numbers, an explanation could be the different input generation between the testing methods. Where AFL uses manipulation on user provided input, KLEE uses its scanning ability to make a input for every conditional path through the code. This means KLEE will find more logic error style bugs and AFL more syntax error style bugs, because it can generate inputs that are unexpected.

Problem 10

These numbers can be found by AFL but not by KLEE: 55, 86, 87, 88, 93, 94, 95, 96, 97, 98, 99

Problem 11

These numbers can be found by AFL but not by KLEE: 93, 98
These numbers can be found by KLEE but not by AFL: 50

Problem 12

The same numbers can be found in both the execution of AFL and KLEE

We can conclude from this that none of the tools works better than the other in finding the numbers. There are some numbers found by AFL and not by KLEE and there are numbers found by KLEE and not by AFL.

Another thing we can conclude is that AFL works faster KLEE, because in the given time constraint, it finds higher numbers. In problem 10 and 11, AFL's highest number if higher than the highest number of KLEE.

6.3 Usability and Future Use

Both Klee and AFL turn out to find bugs that could not be found without automated testing. These input values that are discovered by the programs can be used together with the error that can be retrieved with the input to find the piece of code that could possibly have a bug and needs to be investigated. These kinds of testing methods should be used by all major software projects, because testing is one of the major components of coding.

	AFL	KLEE
Problem 10	14, 15, 16	14, 15, 16
	20, 22, 24, 25, 26, 28, 29	20, 22, 24, 25, 26, 28, 29
	30, 31, 35, 37, 38, 39	30, 31, 35, 37, 38, 39
	42, 43	42, 43
	50, 52, 53, 54, 55, 58	50, 52, 53, 54, 58
	61, 62, 65	61, 62, 65
	75, 76, 77, 78	75, 76, 77, 78
	80, 86, 87, 88, 89	88, 89
	90, 91, 93, 94, 95, 96, 97, 98, 99	90, 91
Problem 11	5, 6, 11	5, 6, 11
	25, 26	25, 26
	32, 38, 39	32, 38, 39
	40, 42, 49	40, 42, 49
	51, 52	50, 51, 52
	61, 62	61, 62
	76	76
	85, 89	85, 89
	91, 93, 98	91
Program 12	0, 1, 6	0, 1, 6
	15, 19	15, 19
	27	27
	34, 39	34, 39
	41, 44	41, 44
	56	56
	64	64
	71, 72	71, 72
	89	89
	93	93