Software testing and reverse engineering, assignment 1 (Reversing)

Harm Griffioen (4303598), Eva Anker (4311426), Mark Pasterkamp (4281500)

March 2017

1 Introduction

Finding bugs in programs is often a hard task. Therefore many tools are created to make life easier. This document explains what fuzzing and concolic execution are. It also reviews some of the tools that can be used to find bugs and possibly vulnerabilities in software.

2 Fuzzing

Using fuzzing, one can automatically test software. It tests the software for bugs and vulnerabilities by providing random data as input which can be invalid or unexpected. Most of the time, a fuzzer uses an input format to know what kind of input is expected. The input format is defined in a file and it gets used to generate tests. The goal of fuzzing is to give the program a certain kind of input so that it crashes. Fuzzers can also be used to find memory leaks. Fuzzing can be done in white-box, grey-box and black-box. A whitebox fuzzer knows the source code of the program [4]. A greybox fuzzer does not know the program by source code, but is able to see the compiled version of the code. A blackbox fuzzer knows nothing about the code, not even the compiled version. This makes blackbox fuzzing more versatile. Blackbox fuzzing can for example be done on a web server to which you only have access to the front-end. Blackbox testing evaluates inputs faster than the other two, but does not have the same chances to find bugs as it is largely based on random inputs.

Whitebox fuzzing can be done using symbolic execution. With symbolic execution, the program can be tested without providing concrete inputs. The values of variables in the software are replaced by symbolic expressions. Then the software traverses all branches. When the software crashes or violates an assertion, the symbolic expression built from the branching statements is used to give concrete values to the inputs.

Blackbox fuzzing can be done by continuously generating random inputs. Some programs are really hard to test using blackbox fuzzing. Take for example the next simple program:

```
void test(int x) {
    int y = x + 20;
    if(y = 120) ERROR;
}
```

Using blackbox fuzzing, this program has a $\frac{1}{2^{32}}$ chance of reaching the error each time it is ran if x is a 32-bit integer. This means that the chance that blackbox testing finds this case in K runs is $K * \frac{1}{2^{32}}$. Whitebox testing will do this differently. A form of whitebox fuzzing is symbolic execution, another form of whitebox fuzzing is concolic execution, which is explained in the next section. Symbolic execution works as follows: The program has the execution tree as seen in figure 1. The program evaluates the branch and creates the symbolic expression . When evaluating this, the fuzzer will create a true and a false case. In case this is false, the value of x is any value that fits in the integer and is not 100. The fuzzer will assign a value to this. If the condition evaluates to true, x must be 100.

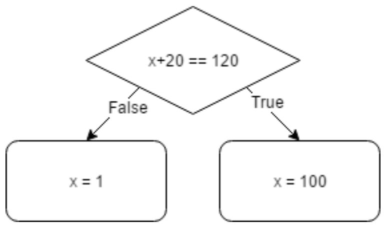

Figure 1: Execution tree for simple program.

When programs become very large, whitebox and greybox fuzzing will become really slow, because the symbolic expressions will become really large. Blackbox fuzzing might then be the better choice, even if there is access to the source.

3 Concolic Execution

Concolic execution is a combination of concrete and symbolic execution. Concolic execution tries to find the different path conditions. These conditions are solved with a SAT-solver and then concrete values are solved from that. During execution, when concolic execution finds an assignment or condition to a variable, it creates a proposition variable for it (for instance: x = 5 will be converted to p = (x = 5)). When the execution reaches branching statement, it creates SAT-clauses out of the conditions. Then, a SAT-solver tries to find an assignment to the proposition variables such that all the SAT-clauses are satisfied. Afterwards, the concrete variables are calculated according to the conditions imposed by the proposition variables. If a solution can be found, the problem is solved. Otherwise the SAT-solver has to find another assignment satisfying the SAT-clauses.

We will demonstrate this according to the following example:

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

Here we have the following proposition variables: $p1 = (z = 2 * y)$, $p2 = (x == 100000))$, $p3 = (x < z)$. To reach the error, we need to find the solution for: $p1 \wedge p2 \wedge p3$. The solution to this will trigger the branch with the error.

Now we have to find concrete variables for x and y such that $x == 100000$, $z = 2 * y$ and $x < z$. One of such solutions generated from the SAT solver can be $x = 100000$ and $y = 100000$.

4 American Fuzzy Lop (AFL)

AFL is a fuzzer that uses greybox techniques and genetic algorithms. The genetic algorithm of AFL mutates input and rates them on fitness [4]. The fitness is based on unique code coverage, that means triggering a path in the program that is not covered by other inputs. AFL tracks the states of the program using instrumentation. The instrumentation is a piece of code that is injected by AFL at each branch of the application that is being tested. This way AFL knows which path is currently traversed in the program. With the genetic algorithm, the fuzzer will give more fitness to inputs that let the program behave in a different way. For a fuzzer (and also for a concolic execution program), loops are problematic as they introduce a lot of extra branches. AFL reduces the size of the path space of loop iterations to reduce the amount of paths that need to be considered. To detect a crash, AFL waits until the application is stopped and look at the return code. So in short: AFL is a fuzzer that uses genetic algorithms to mutate input to the program, based on the previous runs of the program that generated unique paths.

4.1 Disadvantages of AFL

The disadvantage of AFL, as with basically every fuzzer, is the generation of a specific input. For example in the following program:

```
void test(int x) {
    int y = x + 20;
    if(y = 120) ERROR;
}
```

It is easy to see that the error will be reached when 'x = 100'. AFL will however most likely not be able to find this value easily. Finding new execution paths depends on the random input of the fuzzer, therefore finding a new execution path is not always easy using AFL.

4.2 Strengths of AFL

AFL is really easy to set up and start fuzzing. It is also really fast in generating new input. The genetic algorithm makes sure that the inputs generated by AFL are likely to find new execution paths. Sometimes while running, AFL does not find any errors for a long time. But once it finds an error it is likely to find some more errors really quickly as it enters a new execution path with the genetic algorithm. This way AFL distinguishes itself from random fuzzers.

5 KLEE

KLEE is a concolic execution engine built on top of the LLVM compiler infrastructure. LLVM is an open source C and C++ compiler. The main goal of KLEE is to go through every branch in the program and check if some inputs causes the program to crash by means of concolic execution. Before KLEE can look at the program, you first need to mention to KLEE which variables are made symbolic. This can be done in the source code before you compile the program to bitcode. At the start of the program, no constrained are put on the symbolic variables, but this changes when a branch condition occurs. When a conditional that depends on a symbolic input is encountered, KLEE tries to go down both paths by conceptually forking itself meaning that KLEE will clone the current process and put the appropriate constraints on the symbolic variable to go down both paths.

5.1 Advantages of KLEE

The advantages of KLEE is that it is well suited to find exact branch condition compared to, let's say AFL. This is because KLEE goes through the source code and puts constraints on its symbolic variables when it finds a branch condition.

5.2 Disadvantages of KLEE

Klee also has its disadvantages because of the way how concolic execution works. KLEE tends to be a bit time consuming, because KLEE first interprets the application code and then does constraint solving (finding a solution to an np-complete problem). Another issue with KLEE (and concolic execution in general) is state explosion: the number of paths grow exponentially as the concolic execution engine explores the program. An example of such a program is the recursive fibonacci implementation. Because of the recursive definition of the program, the amount of paths KLEE tries to explore grows exponentially.

6 ANGR

ANGR is a framework for binary analysis, namely static and dynamic symbolic analysis. It aims to be easily implementable, so multiple techniques can be easily compared. The static analysis consists of generating different sorts of flow and dependency graphs. As well as, performing a backwards slice, which is the programing with some parts missing, whilst remaining executable.

The dynamic symbolic execution engine of ANGR, has implemented two different methods (one deprecated) in which the execution can speed up due to executing multiple paths in bulk. These multiple paths are sorted and stored in stashes, and can still be used as if they were executed after each other. So, there is still the possibility to step, filter, merge and move along paths.

Whilst analyzing a binary with ANGR you can give it zero or more memory addresses it needs to find or avoid. This way the multiple paths a certain function is executed can be mapped, as all these paths will be stored together in the found stash. The other standard possible stashes are deadended, active, avoided and errored. Other stashes can also be stored, as for example unsat.

6.1 Difference with AFL

As AFL is a grey box fuzzer, it depends on the input that is generated by its genetic algorithm, where ANGR does not need input and traverses every path. Another difference is that AFL randomly tries and only finds errors, where with ANGR you can search specifically for a function that does not trigger an error. ANGR is a lot slower than AFL in finding bugs.

6.2 Difference with KLEE

ANGR runs on binaries whilst KLEE runs on LLVM. In order to execute KLEE the code needs to be modified in order to run it through the engine, whilst for ANGR this is not needed. The possibility to do static analysis is missing in KLEE in comparison with ANGR. KLEE is faster than ANGR in finding bugs.

7 RERS reachability problems

We have used AFL, KLEE and ANGR on problem 10 and 11 from the 2016 RER reachability problems. We will go over the results in this section.

7.1 AFL

AFL outputs a lot of files, each containing different inputs. When using the inputs that is stated in one of the crash files, the crash can be replicated. There are crash-files, hangs-files and queue-files. The crash files give inputs that crash the program and the hangs files give input that create an infinite loop in the program. The queue files are the files that AFL uses as input for the genetic algorithm, therefore these files might contain useful test cases. The ‘unique’ crashes found are not as unique as one might think. The error that is thrown can be the same, but the path that is used to get there can be different.

Table 1: AFL RERS problems

Problem	**Unique chrashes**	**Unique hangs**	**Paths found**	**Executions done**
Problem 10	65	10	109	15.263.079
Problem 11	112	19	664	14.640.963

As can be seen from table 1, both problems are ran many times. The fuzzer continuously runs until it is stopped by the user and is more likely to find bugs when it is ran for a longer period of time. Problem 11 is a larger problem than problem 10, which can be seen from the amount of paths found. Fuzzing does not always cover every branch when it is stopped, therefore the code coverage of fuzzing a problem will most likely not be 100

7.1.1 Future usability of AFL

When computers get more and more processing power in the future, there might be a point where concolic execution will be fast enough to use on every program. This would mean that fuzzing will not be useful for the set of programs where one can use concolic execution (one needs the source code). There will however be problems where the source code is not known, these problems will be in need of a tool like AFL. Therefore AFL will remain relevant. That AFL is relevant cannot be doubted after seeing the wall of fame on the AFL website: `http://lcamtuf.coredump.cx/afl/`, in which they show the programs where bugs were found using AFL.

7.2 KLEE

Klee had no problems with terminating on Problem 10. However, for Problem 11 we stopped the execution after one hour. Klee returns files with the same format as AFL. These files can be put into the program to recreate the crashes.

Table 2: KLEE RERS problems

Problem	**Tot. Instructions**	**Time (s)**	**Instructions %**	**Branches %**	**Unique instructions**
Problem 10	69994485	302,76	90,10	75,55	4877
Problem 11	594423155	3630,96	88,70	66,73	15182

In table 2, the total amount of covered instructions can be seen. As well as the unique instruction count that is covered. The percentages show how many of the total branches and instructions are covered by KLEE.

7.2.1 Future usability of KLEE

Concolic execution is getting more and more used and so is KLEE, a popular concolic execution tool. The main disadvantage of KLEE is speed and that is getting solved in the future, as computers get faster every year. This will even make the gap between fuzzing and concolic execution smaller, and therefore KLEE might become more popular. This makes that the future seems bright for KLEE.

7.3 ANGR

The output of ANGR is stored in the different stashes. In the stashes, all the different paths are grouped. There is no list of output or such. This allows for total freedom of what data you want to have in the end, and what you want to do with it. For us we found that the input variables are relevant, these are always concrete values. So, when the condition is 0 ¡ int x ¡ 3, the output is (1) or (2). The path that has been taken can be displayed, however the name of the error that we encountered could not be found.

Table 3: ANGR RERS problems

Problem	**Time**	**Deadended**	**Active**	**Found**	**Errored**	**Unsatisfiable**
Problem 10	2 hours	54	305	32	82	160
Problem 11	2 hours	62	279	0	37	0

Both problems have been stopped after 2 hours of running. In table 3 all the values for the different stashes are stated. The deadended stash are the branches that can not continue executing. Active are all the currently running paths. Found are all the paths that have matched the stop condition, in our case if an verified error occurred. These are not unique as there can be multiple paths leading to one error. Errored are when a branch is stopped due to an exception in the code. Unsatisfiable are all paths with contradicting constraints.

7.3.1 Future usability of ANGR

The main drawback of ANGR is the speed. As computers are getting faster, the implications of this drawback will decrease. As ANGR has not been around that long, it has not yet proven his usefulness in finding bugs. However the results we found during this project seem promising and they are actively developing it to for example support LLVM. Therefore it seems that ANGR will become a valuable tool in the future.

7.4 Result comparison

In table 4, the results of the three tools are compared. The actual results[1] are also displayed, to have a reference at how good they perform. From this table we can derive that AFL gives the best result, as it has missed only one in total. Whilst KLEE performs as good as AFL in problem 10, with problem 11 it only found half of the errors. However it was stopped and thus these results could eb improved. ANGR does not seems to give good result, however it was also stopped before it could finish and therefore the results could become better if it can run for a longer time.

Table 4: The amount of unique errors found by AFL, KLEE and ANGR

Problem	**Solution**	**AFL**	**KLEE**	**ANGR**
10	45	45	45	10
11	22	21	9	11

7.4.1 Different solutions

The problems that are found by ANGR are all at most 4 inputs long. This means that ANGR was increasing the input size and had not yet arrived at longer inputs. For problem 11 there are no solutions smaller than 5 inputs and this explains why no solution has been found.

KLEE only found solutions smaller than 20 inputs. For example error 62 consists of 92 inputs and is therefore not found by either ANGR or KLEE, but is by AFL. Error 53 is the only error not found by AFL and is 54 characters long and is therefor not found by KLEE either.

References

[1] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[2] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[3] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[4] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016.

[1]Solutions are retrieved from http://www.rers-challenge.org/2016/index.php?page=results