CS4110: Software Testing & Reverse Engineering

Software Reversing Lab 1

Liam Clark 4303423
Jean de Leeuw 4251849

March 20, 2017

Introduction

This report starts by explaining the ideas behind fuzzing and concolic execution, including two small examples. It will then describe an experiment done on the RERS reachability problems, using both a fuzzing tool (American fuzzy lop, AFL for short) and a concolic exection tool (KLEE). The findings of these tools will be analysed and discussed, and we explain why these findings might differ from each other. Finally we comment on the (future) usability of both AFL and KLEE..

Fuzzing

Programmers, and humans in general, have a tendency to only think about the happy path, leading to many security vulnerabilities. We have created a simple example (which can be found in the appendix at the end of the report) where the user input is not sanitized and directly used as an index into a buffer. Providing the program with more than 4 arguments will result in a segmentation fault.

This error was made possible because an input was provided that was not foreseen by the programmer. When you are writing code and trying to predict how it may break, you think of the correct input, and input that is completely wrong. After figuring this out, you create preventive measures to prevent the code from breaking from these inputs. However, humans have trouble manually detecting the many possible sequences of inputs that may lead to breaking the program, and that is were fuzzers come in.

Fuzzing is a automated testing technique that aims to break the program by trying randomized inputs. You provide a fuzzer with a sample input, and the fuzzer will mutate this into similar but slightly different inputs and will try these out on the program, hoping to find an error. The idea behind this is to "generate semi-valid inputs that are "valid enough" so that they are not directly rejected from the parser but able to exercise interesting behaviors deeper in the program and "invalid enough" so that they might stress different corner cases and expose errors in the parser. If the error is found the input that caused the error is saved and can be used to fix the cause of the error.

If you provide a fuzzer in our example with a simple starting alphabet, e.g. "1 2" it will mutate over time into something with more than 4 arguments, e.g. "1 2 2 23". This will cause the program to crash, indicating that the program contains a vulnerability.

Concolic Execution

Fuzzing is a relatively fast process, but has a large weak spot. It has trouble finding exact values. For example, if a branching statement needs an input variable to be exactly equal to 10.000 for a 32 bit integer, then the odds of arriving in that branching statement with random input is $\frac{1}{2^{32}}$, which is extremely small. These kind of conditionals can cause the fuzzer to become stuck.

In order to mitigate this weak point of fuzzers, symbolic execution can be used. Symbolic execution tracks variables with symbolic values instead of concrete values, for example instead of inputting $x = 5$ into a program, symbolic execution based tools put in $x = \lambda$. This allows for operations executed on the symbolic values to be tracked, for example multiplying the value by 2 becomes $x = 2 * \lambda$ instead of $x = 10$. These operations serve as a list of constraints, and by negating and reversing them using constraint solvers, the tool is capable of finding an exact input value for x that will pass the conditional. We have created another simple example (which can also be found in the appendix at the end of the report), that which highlights the strong point of symbolic execution.

In order to reach the crash (klee.assert(0);), the conditional $a - b > 0$ needs to be fulfilled, where a is the programs first input, and b the second input minus the first. Since this is a simple constraint, randomly guessing (like a fuzzer) would probably be able to decipher this at some point, but imagine such conditionals within multiple similar conditionals and it quickly becomes very difficult for fuzzers.

So here a tool which uses symbolic execution would fill in the x for the programs first input, and y for the programs second input. Then $a = x$ and $b = y - x$. The conditional would then become $x - (y - x) > 0$, which in turn becomes $2x - y > 0$, and finally $x > \frac{1}{2}y$. It is then easy for the tool to see that in order to reach this branch, the value of the second argument provided to the program needs to be smaller than two times the value of the first argument.

Synbolic execution can be a very slow process, since solving the constraints (statisfiability) is a NP-complete problem. This is why symbolic execution on itself is not desirable either, even though it would result in a perfect answer whether certain parts of code can be reached from other parts of the code.

Concolic execution is the middle ground, and the word itself is a combination of ***conc***rete exection (fuzzing) and symb***olic*** execution. The prinicple itself is very simple. Concolic execution tools use fuzzing whenever possible, allowing to generally pass through the non extremely specific branches, and then uses symbolic execution whenever the fuzzing gets stuck for too long. When the symbolic execution has figured out the answer for that one specific conditional, the fuzzing continues. Allowing for a good trade off between exhaustive testing and speed.

AFL and KLEE

American Fuzzy Lop (AFL)

American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. This substantially improves the amount of code that is covered by AFL in the binary.

AFL needs to be provided input in the form of test cases so that it has a basis of what a (semi) valid input is (as explained in the "Fuzzing" section). AFL will then mutate the given test cases into new test cases using genetic algorithms, using the test cases that trigger new internal states (and are therefore more effective at finding bugs) as a basis for the new test cases.

This makes AFL good at quickly finding input that is "valid enough" to not directly be rejected by the program but is able to exercise interesting behaviours deeper in the program and "invalid enough" so that it might stress different corner cases and expose errors in the program, as described in the "Fuzzing" section.

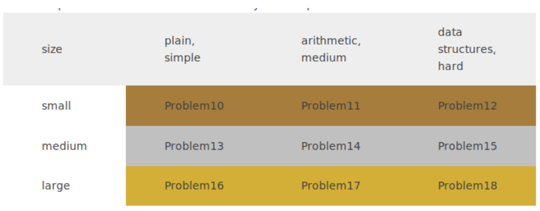
KLEE

KLEE is a concolic testing tool that allows to symbolically track parts of the program, for example, the user input. It also allows the user to specify which parts of the code the user is interested in if KLEE can reach this part. KLEE will then to go through the code and generate a list of constraints that the input needs to satisfy in order to reach those parts of the code. KLEE then uses a satisfiability solver to check whether these constraints are satisfiable and if so, creates a test case with values that satisfy these constraints.

Experimenting with AFL and KLEE

The RERS Challenge is a challenge to find as many possible crashes in highly obfuscated code. Each problem has a different difficulty and contains a 100 different possible crashes.

On the RERS Challenge 2016 site we found that the following table describing the difficulty of each problem:



size	plain, simple	arithmetic, medium	data structures, hard
small	Problem10	Problem11	Problem12
medium	Problem13	Problem14	Problem15
large	Problem16	Problem17	Problem18

Figure 1: Difficulty table

Based on this table we decided to run both AFL and KLEE on problems 10, 14 and 18. These problems representing each category and become gradually more difficult. We have set the following rules for the experiment:

- The time limit is 10 minutes.
- KLEE tools will have up to 20 symbolic inputs. Due to the small size of this report we decided to keep this constant to not introduce more variables. This means that if KLEE has not reached the allowed time limit but completed the search space with 20 symbolic values it will not be enlarged.
- For AFL we gave every possible allowed input (specified at the top of the source code) as starting test cases.

An analysis of the results

The produced output files

AFL

AFL creates several files as output, three files containing statistics (fuzz_bitmap, fuzzer_stats and plot_data) and three directories, crashes, hangs and queue. For our experiment we are only interested in the crashes directory.
The crashes directory contains the unique test cases that cause the program to receive a fatal signal. Each of those files contain the input that AFL has given to the program which caused this fatal signal. These can be used in order to reproduce the crashes. By reproducing all the crashes that have occurred and capturing their output, we can list all errors that AFL triggered.

We will compare this to the inputs of KLEE later on in the report.

KLEE

KLEE creates a *.ktest* file per path in the program. The ktest file contains the test case generated by KLEE to arrive in that path. These ktest files can then be read by the ktest-tool tool, which contains the input in hexadecimal format.
KLEE also generates *.err* files for every path where KLEE encountered an error. These are especially interesting for us as they represent the crashes by the program. Each err file has a corresponding ktest file, which contains the input used to arrive at the corresponding crash, which in turn can be used to reproduce the crashes.

The differences in the errors found by both tools

We ran AFL and KLEE in accordance with the rules of the experiment explained in the "Experiments of both AFL and KLEE on several RERS reachability problems" section.
By using a several custom made scripts we were able to extract all the unique crashes discovered by both tools. We then compared the results of AFL to KLEE and vice versa and removed the common set, leaving us with only the errors found by one tool and not the other. These results can be found in the table below:

	Problem 10	**Problem 14**	**Problem 18**
AFL	$\emptyset$	0	0
KLEE	$\emptyset$	6, 20, 94	10, 28, 70, 71, 76, 96, 99

Problem 10

For problem 10, both tools found exactly the same errors, so after removing the common set only the empty set (∅) remains.

Problem 14

Here we see that AFL was able to discover the inputs needed to invoke the error 0, which KLEE was not able to find. And KLEE was able to discover the inputs needed to invoke errors 6, 20 and 94, which AFL was not able to find. We start with the errors discovered by KLEE but not by AFL.

If we take a closer look at the inputs needed to invoke errors 6, 20 and 94 (listed below) we can see that all three errors start with an input of 6 followed by 2. And errors #6 and #94 even start with a shared sequence of 6, 2, 3, 2, 9.

Input for error #6	Input for error #20	Input for error #94
6	6	6
2	2	2
3	5	3
2	6	2
9	3	9
1	8	8
7	3	9
9	9	1
10	10	10
	5	1
	5	2

As we have described in the "Concolic execution" section, fuzzers have trouble finding the exact values or specific sequences needed in order to progress through the code. This is due to fuzzers using randomization in order to find solutions, and the odds to randomly find a specific sequence is (very) small. AFL also uses mutation in order to speed up the process, but mutation also contains a random element. All three of the displayed errors start with a specific sequence, and two of those (6 and 94) with a even more specific sequence. We suspect that AFL was never able to unlock this path in the code by simple not randomly guessing this input. It could also be that AFL was more successful in finding crashes in other paths of the code and therefore the mutations favored those paths, making a completely different sequence to be chosen even less likely. This also explains why KLEE, which discovers inputs through symbolically tracking variables instead of randomization, was able to discover these crashes.

This leaves us with the one error that AFL was able to find but KLEE was unable to find: error 0. We start again by looking at the input required to invoke this error, found below.

The input is relatively simple, and there seems to be no reason as to why KLEE would not be able to discover the error. Our own rule about the limit of having only up to 20 symbolic inputs also do not seem to be the problem, since this input only contains three variables. The asterisk symbol is a invalid symbol and is therefore rejected by the code.

Input for error #0
6
7
4*

We manually tested and the error also occurs by omitting the asterisk, so that is also not the issue. This only leaves one possibility.

KLEE had to be forced to stop on this problem, since it hit the time limit of 10 minutes. This means that it did not complete its entire search space yet. This means that there were still branches KLEE was discovering and was trying to determine the inputs needed to unlock these branches. We can only assume that KLEE was working on other branches and did not reach this branch yet/put this branch in some sort of processing queue. Since the error has been proven to be reachable (AFL was able to reach it) and only 3 inputs were needed, which falls within the bounds of the 20 that we have set up in our rules, KLEE should be able to discover this error as well when given enough time.

Problem 18

For consistency reasons, we start again by discussing the errors KLEE was able to find and AFL was not able to find.

As displayed by the table, the amount of errors KLEE was able to find compared to AFL in problem 18 becomes even more significant. The inputs required to invoke these errors can be found below.

Error #10	Error #28	Error #70	Error #71	Error #76	Error #96	Error #99
16	16	16	16	16	16	16
3	11	11	11	11	11	11
1	3	3	3	3	4	4
12	13	4	4	4	1	9
12	7	5	5	5	8	
16	3	16	10	10	17	
14	18		1	6	5	
5	11				12	
3	18				5	
14	10				15	
8	5					
12	5					
18	8					
	4					

All these errors except error 10 share the same starting sequence 16 followed by 11. Error 10 only starts with 16. We suspect the same reasons here as to why AFL failed to discover these errors as described in problem 14. AFL probably failed to discover the errors from paths starting with 16 (and 11), and therefore also failed to find all the paths derived from this path (the even more specific sequences, e.g. 16, 11, 3, 4, 5).

Once again, this leaves us with the one error that AFL was able to find but KLEE was unable to find: error 0. We start once more by looking at the input required to invoke this error, found below.

Input for error #0
16
3
17

9

Once again, the input is relatively simple, and there seems to be no reason as to why KLEE would not be able to discover the error. The amount of inputs is also still less than 20. Due to this, we conclude that KLEE was unable to find this test case due to the same reasons as explained above, in the discussion about error #0 of problem 14.

Reflection on the (future) usability of AFL and KLEE

AFL and KLEE are both great tools for both developers wanting to thoroughly test their system, as well as users trying to break into the system of others. AFL is great for quickly finding "shallow" vulnerabilities in the system, so for example vulnerabilities in the parser of the user input. KLEE on the other hand is better at finding exploits that reside deeper in the system, being able to determine and create the input format that is desired by the parser and then find exploits in the system beyond the parser. The weakness of KLEE being that there are so many possible paths, each of them needing a satisfiability solver, which is a slow process.

AFL and KLEE might seem similar at first, but they both satisfy different needs, and are probably best used together, as can also be seen in our experiments where they were both able to discover errors that the other did not find. We hope that this report was able to highlight the strengths, weaknesses and differences between these tools. A good paper that really helps in understanding the strengths, weaknesses and differences is *Driller: Augmenting Fuzzing Through Selective Symbolic Execution* by Stephens N., et al. which can be found here.

Appendix

Code example of fuzzing

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void own_str_copy(char** buffer, int index, char* stringToCopy)
   {
  int len = strlen(stringToCopy) + 1;
  buffer[index] = malloc(len);

  for(int i = 0; i < len; i++) {
      buffer[index][i] = stringToCopy[i];
  }
}

int main(int argc, char** argv) {
    char** teachers = malloc(4 * sizeof(char*));
    own_str_copy(teachers, 1, "arie");
    own_str_copy(teachers, 2, "andy");
    own_str_copy(teachers, 3, "sicco");

    printf("%s", teachers[argc]);
    return 0;
}
```

Code example of concolic execution

```c
#include <assert.h>
#include <stdio.h>
#include <klee/klee.h>


int main(int argc, char** argv) {
        int length = 10;
        int program[length];
        klee_make_symbolic(program, sizeof(program), "program");

        int a = program[0];
        int b = program[1] - a;

        if (a - b > 0) {
         klee_assert(0);
        }

        return 0;
}
```