Modern Software Testing Techniques: Fuzzing and Concolic execution

Guledal, Prashanth*, Sarkar, Aritra† and Subramanian, Vivek‡
Delft University of Technology
*4610849, †4597982, ‡4601211

1 Introduction

Software testing is one of the most expensive task of any software development. It is estimated to account for nearly 50% of software development cost [1]. One of the estimates [1] informs that US software producers lose 21.2 billion dollars annually because of inadequate testing and errors found by their customers. This gives a clear indication that improvement in the quality of software testing techniques is the need of the hour for both software developers & users in reducing the costs in the long run. In turn, billions of dollars are invested by companies worldwide every year in developing smart and cost effective ways of doing software testing.

This has led to the development of novel approaches such as directed fuzzing, symbolic execution, concolic execution, etc., that are drastically changing the way we think about software testing. In a quest to automatically generate test cases that achieve very high coverage on a diverse set of complex and environmentally intensive programs, two of the state of the art tools stand out, which are American Fuzzy Lop (AFL) and KLEE. AFL is a set of utilities that aid in fuzzing applications for finding obscure bugs. KLEE on the other hand is based on symbolic execution and is a very highly regarded bug finding tool. These two tools, that have contributed largely in transforming the software testing landscape in recent years are the main points of discussion in this report.

We dwell into the testing of Rigorous Examination of Reactive Systems (RERS) Reachability problems 2016 using these tools to analyze the importance of the underlying methods behind the tools used and critically compare between these two tools.

Section 2 gives brief explanation of fuzzing in software testing context. Section 3 gives an intuitive overview on Concolic execution. Section 4 describes fuzzing tools, AFL & a symbolic execution tool, KLEE. Section 5 details about the experimental framework using these tools on selected RERS 2016 problems. Section 6 details on the analysis of results obtained from these 2 tools. Finally, section 7 concludes the report.

2 Fuzzing

Fuzzing or fuzz testing is a type of testing in which automated or semi-automated testing techniques are utilized in discovering coding errors and security loopholes in software by inputting invalid or random data to the system. It is a kind of black box software testing technique. In short, unexpected or random inputs might lead to unexpected results. The key phases involved in this form of testing are (sequentially):

- Identify the target system
- Identify inputs
- Generate fuzzed data
- Execute the test using fuzz data
- Monitor system behavior
- Log defects

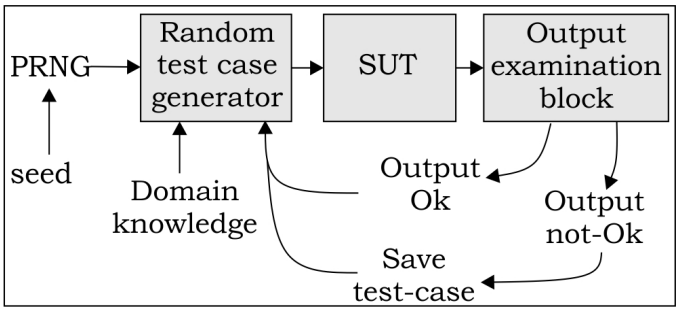

Fig. 1: Block diagram view of fuzzing

These points are clearly depicted in Fig. 1. It has a random test case generator. The test case generator takes its input from pseudo-random numbers. Pseudo-random number generator (PRNG) generates algorithmically pseudo-random numbers. To start using a PRNG, we give it a seed which completely determines the sequence of random numbers it's going to generate. One other input that goes into a random test case generator, usually to make a good one, a significant amount of domain knowledge which means understanding some properties of the software under test (SUT). Generated test cases come out of the random test case generator and they go into the SUT. The software under test executes and produces some output. The output is then inspected to determine whether the output of the SUT is

either ok or not. If it is ok, testing is continued with new random inputs. On the other hand, if the output is not okay, then we save the test case somewhere for later inspection and continue with random testing further.

Lets consider a piece of C code in Listing 1. Lets assume it generates some random values of x (=2) & y (=3) based on some initial seed. Now, these inputs does not help in reaching the error condition in line 4 as ($6 > 5$). Fuzzer continues testing with new set of inputs for x and y, till it reaches all parts of the code. In our case, it can reach abort condition with many set of x & y inputs, one such set being x=2 & y=1. It saves this case and continues further with next set of random inputs. But in reality there might be cases in which some part of the code could not be reached with any set of inputs. This makes the fuzzing process to run infinitely.

Listing 1: A simple C code

```
1 void ftest (int x, int y) {
2 int z = 3 + y;
3 if (z < (10/x)) {
4 abort ( ); // error
5 } }
```

Many strategies are used for fuzzing. Some of the important ones are mentioned here. *Mutation based fuzzers* which modify existing data samples to create new test data. *Generation based fuzzers* define new data based on the input of the model. *Protocol based fuzzer*, the most successful fuzzer which has a detailed knowledge of protocol format being tested. A wide range of fuzzing tools have been developed as open source tools such as AFL, SPIKE proxy, OWASP WSFuzzer, etc., to exploit the software under consideration for possible bugs.

There are many advantages of fuzz testing. It detects many important bugs such as assertion failures, memory leaks, invalid inputs and correctness bugs. It is a statistically proven method to identify bugs in a short period of time which greatly improves software testing. Bugs found in fuzzing are sometimes severe and most of the time used by hackers including crashes, memory leak, unhandled exception, etc. In many corporations such as Microsoft, fuzzing is mandatory for every untrusted interface of every product.

Some of the cons cannot be ignored while doing fuzz testing. Fuzzing alone cannot provide a complete picture of an overall security threat or bugs. It is less effective for dealing with security threats that do not cause program crashes, such as some viruses, worms, Trojan, etc. To get a complete code coverage it needs a significant amount of time which is a difficult asset these days. This is because of the fact that random data may fail to trigger complex control flow decisions deep within the program's logic.

3 Concolic Execution

Concolic execution is one of the promising alternatives to fuzzing. Concolic execution is a technique that uses both CONCrete and symbOLIC execution to solve a constraint path, execute program with concrete values and collect symbolic constraints during execution.

Symbolic execution is really the workhorse of modern program analysis. The beauty of symbolic execution as a technique is that compared to testing, for example, it gives us the ability to reason about how our program is going to behave on a potentially infinite set of possible inputs. It allows us to explore spaces of inputs that would be completely infeasible and impractical to explore by, say, random testing, or even by having a very large number of testers banging at the code. On the other hand, compared to more traditional static analysis techniques it has the advantage that when it discovers a problem it can actually produce an input and a trace that can be run on a real program and execute that program on that input. And we can actually tell that it is a real bug.

Concolic execution attempts to reconcile between Fuzzing and Symbolic execution. The basic idea is to have the concrete execution drive the symbolic execution. In turn, it uses concrete, random testing to test a program path. The program runs as usual, but in addition it also maintains the usual symbolic information. Every time a single path from a branch is selected, the information is sent to the Satisfiability (SAT) solver to find rules or values for the input variables, so that the branch will be selected on the next iteration of input variables. In essence, the SAT solver is called for every branch, but *not* for every executed statement.

Listing 2: A simple C code

```
1 int doubleIt ( int v ) { return 2v; }
2 void testMe (int x, int y) {
3 int z = doubleIt (y);
4 if (z == x) {
5 if (x > y + 10)
6 abort ( ); // error
7 } }
```

Let's consider a simple example shown in Listing 2 to demonstrate working of concolic execution and to appreciate the elegance of the method. The first step is selecting random input values for x and y. Suppose the selected values are x=22 & y=7. We will keep both the concrete store (ct), the symbolic store (st) and path constraint (pct) as shown in Fig. 2.

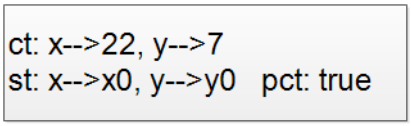

Fig. 2: Interpreter constraints 1

After reaching the 1st branch statement, the values are updated as in Fig. 3 and are passed to the interpreter.

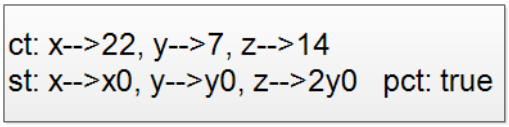

Fig. 3: Interpreter constraints 2

The concrete execution will now take the *'else'* branch of (z == x). Therefore, path constraint is updated to ($x0 \neq 2*y0$). At this point, concolic execution decides that it would like to explore the *'true'* branch of (x == z) and hence it needs

to generate concrete inputs in order to explore it. It, in turn negates the path constraint, obtaining *pct : x0 = 2*y0*.

It then calls the SMT solver to find a satisfying assignment of that constraint. Let's assume the SMT solver returns x0=2 & y0=1. With this input set, it's now possible to reach the true condition of the first branch statement and update the constraints as in Fig. 4.

```
ct: x-->2, y-->1, z-->2
st: x-->x0, y-->y0, z-->2y0   pct: x0=2y0
```

Fig. 4: Interpreter constraints 3

Further continuing, we reach the *'else'* branch of (x > y+10) by updating the constraints as shown in Fig. 5.

```
ct: x-->2, y-->1, z-->2
st: x-->x0, y-->y0, z-->2y0
pct: (x0=2y0) ^ (x0 <=y0+10)
```

Fig. 5: Interpreter constraints 4

Again, concolic execution may want to explore the *'true'* branch of (x > y+10). So, it negates the conjunct (x0 ≤ y0+10) obtaining: *pct: (x0 = 2*y0) ∧ (x0 > y0+10)*.

Now the satisfying assignment of x=30 & y=15 is found by the interpreter which can reach the error condition as clearly depicted with a flow in Fig. 6.

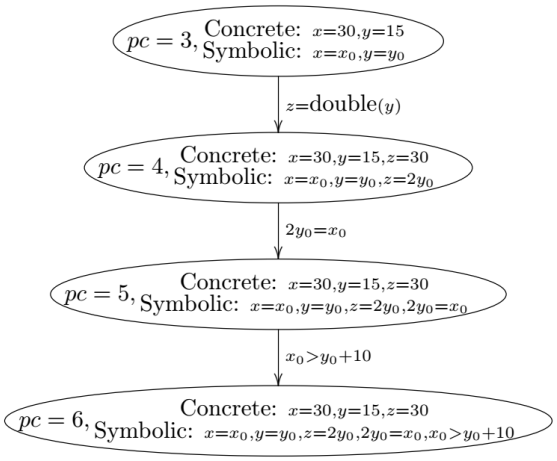

Fig. 6: Execution path with x= 30 & y= 15 which ends in an error

Concolic execution is used exhaustively in industries for finding bugs as part of their software development. It is particularly used in code coverage to generate templates for an input, generating the maximum code coverage possible. These templates can be used to fuzz a program. With some big improvements, it can be used to detect stack overflows and heap overflows. Unlike fuzzing, symbolic execution time required for any given program does not increase exponentially with size of input, rather increases with amount of branches (possible paths) through the program. This eases use of symbolic testing on much larger programs than just using random inputs.

There are many challenges to symbolic execution which is a main part of concolic execution. The main disadvantages with symbolic execution are path explosions, terrible at handling unbounded loops, etc. Concolic testing suffers heavily if the code under test is complex or composed of multiple components with independent functionalities and specific dependencies. It cannot handle multi-threaded programs which are very commonly used these days. It cannot handle certain datatypes such as float, double & pointers as the constraint solvers do not support these types. Many of these are subjects of ongoing research to find novel techniques to overcome. For example, programs that generate very large symbolic representations that cannot be solved in practice cannot be tested effectively. Erete et al. [3] propose to use domain and contextual information to optimize the performance of constraint solvers during symbolic execution.

4 Testing Tools

In this section, the two important tools AFL & KLEE are introduced along with the brief idea on their workings.

4.1 AFL

American Fuzzy Lop (AFL) is one of the most popular fuzzer today. Some of the success stories of AFL includes finding bugs in bash, Mozilla Firefox, Qt, PuTTY, VLC, Adobe Reader, wireshark, etc. One of the major reason behind the popularity of AFL is that, fuzz testing can be initiated with almost no configurability (like fuzzing ratios).

AFL is based on genetic algorithm. A sample input file needs to be provided to start fuzzing. It uses the file as the seed input for further mutation and test case generation. Also, the fuzzer generates an elegant test corpora on-the-fly, which is used internally as seed for specialized test cases. Fuzzing strategies play the most important role in the design of AFL. Conservative changes to the input file achieves limited coverage. On the other hand, aggressive tweaks cause most inputs to fail parsing at a very early stage. AFL provides a feedback loop that can measure what types of changes to the input file actually result in the discovery of new branches in the code. It approaches new input file by going through a series of progressively more complex, but exhaustive and deterministic fuzzing strategies before diving into purely random behaviors. This generates elegant and simple test cases first and helps evaluating the value of each new strategy. The fuzz strategies [2] of AFL in increasing order of difficulty are:

- Walking bit flips
- Walking byte flips
- Simple arithmetics
- Known integers
- Stacked tweaks
- Test case splicing

File size has a dramatic impact on fuzzing performance, both because large files make the target binary slower, and because they reduce the likelihood that a mutation would touch important format control structures, rather than redundant data blocks.

4.2 KLEE

KLEE is a robust symbolic execution tool for automatically generating high code coverage test cases by interpreting Low Level Virtual Machine (LLVM) bitcode. It generates this bitcode using Clang front-end by converting from C code. It has an inherent capability in finding deep bugs in complex programs. Simple debugging and error reporting, along with the capability to find functional correctness errors are possible with KLEE, which lets test cases to be run on raw versions of the code. KLEE has two main goals [4], one of which is to hit every line of executable code in the program and the other is to detect at each dangerous operation (e.g., dereference, assertion) if any input value exists that could cause an error. The paths followed by KLEE are the same as that would be followed by an unmodified version of the program, which drastically reduces the possibility of false positives.

Let's answer an important question 'How does KLEE work?' The key principle KLEE applies whenever it runs the program is to explore every possible path. First, it makes inputs to the program as a symbolic one and executes the program by tracking all the constraints on inputs with each instruction being run. On its execution path, if it encounters any conditional that depends on a symbolic input, a constraint solver is invoked to check the suitability of the conditions so that the direction to a particular path is determined. Now, in cases of branches such as if-else, execution is not constrained to follow a single path. This is where KLEE tries to be smart and fork the execution conceptually. This in turn makes the KLEE clone the current process and follow both paths, adding the appropriate constraint to the path conditions of each process.

When KLEE encounters a bug or a process exits, the path condition records the entire set of constraints on the input that are necessary to drive the program down that path. The interpreter loop, which evaluates instructions until execution is complete forms the main core of KLEE. KLEE selects a process to interpret at each instruction step using various search strategies [4].

What are the key advantages in using KLEE? Well, KLEE is good at finding and showing paths that can lead to any of the below conditions:

- Assertion violation
- Access outside of allocated memory (including null pointer dereference and double free)
- Division by zero
- Integer overshift (shifting an integer by more than its size)
- Call to abort()

It was also recently extended to handle integer overflow checking.

Limitations: KLEE cannot check all paths of a big program. For example, it failed to check on *sort* utility. It was a halting problem (undecidable problem), & KLEE is a heuristic, so it's able to check only some paths in limited time. It is very slow for large programs. It took about 89-hours to generate tests for 141000 lines of code in COREUTILS. It cannot handle floating point types, threads, asm, memory objects of variable size, etc. It lacks models for system environments such as sockets & multithreading. Being an open source project, these limitations are continuously researched by open source community to find new solutions.

5 Experimental Framework

In this section, elaborate details on the experimental setup is discussed for testing using both AFL and KLEE. Various reachability problems of Rigorous Examination of Reactive Systems (RERS) with different code sizes and difficulty levels are chosen for testing. In particular, problems 10, 11, 12, 13, 15, 16 & 18 are selected for test analysis. As both tools are open source based, exhaustive guide is available on their website for installing them. Once they are installed, next step is to compile the code using respective compilers and run with the tools for further analysis. Let's see how these are done for both tools.

AFL: Few changes on source code needs to be made in order to work with this tool. First step is to define assert error function and replace it with its external definition. Then, modify the *scanf* function to avoid hang in when the input ends. After these steps, create two directories in each of the problem directory under consideration and name them as 'test' and 'findings'. Create an input file file in the test directory as a seed for fuzzing. It is important to give many supported data types in this seed file so as to direct AFL to find better random test cases. Now compile the source code with AFL compiler and run it using afl-fuzz command.

The tool immediately starts finding crashes and hang-ins. These outputs are stored in findings directory which could be used for tracing back the crash and analysing the error states. The fuzzing process needs to be run for minimum of 1 complete queue cycle as per its recommendation, so it is run accordingly for each problem. There are three sub-directories created within the findings directory along with some other files containing some statistical information and are updated in real time. The *queue* directory contains test cases for every distinctive execution path, plus all the starting files given by the user. The *crashes* directory holds the unique test cases that cause the tested program to receive a fatal signal while the *hangs* directory contains unique test cases that cause the tested program to time out. Crashes and hangs are considered unique if the associated execution paths involve any state transitions not seen in previously-recorded faults.

KLEE: It is a very simple tool which needs very few steps in order for source code to work with it. Similar to AFL, assert error functions needs to be replaced and KLEE header file needs to be included. Final step is to make the inputs symbolic and we are good to go. Compile the source code with *clang* command to create a bit code file. Use this generated file with a klee command to start the dynamic testing.

KLEE names the output directory klee-out-N where N is the lowest available number which contains all the information to further analyze crashes. Each test 'n' has its associated '.ktest' & '.kquery' file containing the test cases and the constraints generated by KLEE on that path. It also generates '.<error-type>.err' files for paths where KLEE found an error and contains information about the error in textual form. There are many other files containing various

statistical information such as run.stats, warnings.txt, etc. In addition, there are global files having KLEE execution information [6].

6 Analysis of Results

This section deals with the analysis of achieved results for various RERS reachability problems.

6.1 Analysis method

We have huge amounts of crash data to analyse. A systematic approach is needed to deal with this huge chunk of data. In this regard, we created a simple bash and java scripts to analyse this data. Bash script runs the files in a crash directory of AFL along with output (a.out) file for each problem to generate a text file containing only *reached error* details. The 'klee' command which is used for KLEE testing, is extended with "&> report.txt" to output all the error details to the text file for each problem. Java script uses these two text files (one each from AFL and KLEE) for each problem along with their respective solution file to analyse the data inside. This script compares the errors in the solution file with AFL text file as well as KLEE text file to generate output with 4 columns (**Error number, AFL count number, KLEE count number and comment**) in each row. **Error number** in each row corresponds to the reachability errors mentioned in the solution file, **AFL count number** indicates the number of times the tool reached the error, similarly, **KLEE count number** indicates how many times it reached the particular error and **comment** generates additional information such as 'not detected' or detected just by KLEE or AFL. All the generated data are ported to excel for further analysis.

6.2 Insights on results

To get a birds eye view of the type of problems KLEE and AFL are efficient in, the results obtained from running 7 RERS16 problems are analysed. The table in Appendix A lists all the 100 possible errors. The RERS16 solution files indicate which of these errors are reachable for each of the problems. Each error is reachable by a suitable number of input sequence states as per the solution file. These values are put in the cells of the corresponding error number. A value of 0 indicates the error is unreachable for the particular problem. The cells are coloured based on the experiment results. Cell colour coding could be understood from the legend provided. Careful observation of this data reveal that, the longer the pattern length (i.e. input sequence states), the probability of tools reaching the error diminishes. Also, it can be observed that, for a fixed pattern length, it is easier to reach the error in the code for a small sized problem, compared to a large sized one.

Lets see some interesting observations from these findings. We observe that, for Problem 10, *all* reacheable errors were found by both tools. For Problem 11, two observations stand out. First, one error was not reached, the pattern length of 92 which was considerably longer than the rest. This seconds our observation that longer pattern lengths are difficult to reach even with symbolic execution tool, KLEE. Second, **5 errors were found only by AFL**. Most

Fig. 7: Venn Diagram showing the observed Error Reachabilities.

probable reason based on our observations is that KLEE fails to detect pattern lengths greater than 25 for this problem. It is elaborated later in this section to reveal some interesting thresholds. AFL being a fuzzer can work successfully even for large pattern lengths but under the condition that the code is simple and small sized.

For all the other problems (12-18), the trend changes. For example, consider Problem 12. One error is not reached, **error 41 is reached only by KLEE**, rest of which are found by both tools. We noticed one important finding here. For the medium and large size problems with sufficient complexity, there is **no error which AFL finds and KLEE cannot find**. This is clearly depicted in Fig. 7, which shows KLEE has higher code coverage than AFL. The only exception being 5 errors found out only by AFL in the Fig. 7, corresponds to small size Problem 11 as previously explored. This is not difficult to apprehend. KLEE, being a symbolic tool gives high code coverage even for large-size & complex problems using optimized SAT solver. AFL fails here to give high code coverage as it lacks the sufficient knowledge on the structure and flow of the code under examination. This proves that fuzzers are only as good as they are going to get! In other words, they aren't yet matured enough to give high code coverage, although they have an inherent ability to find some real bugs in short period of time.

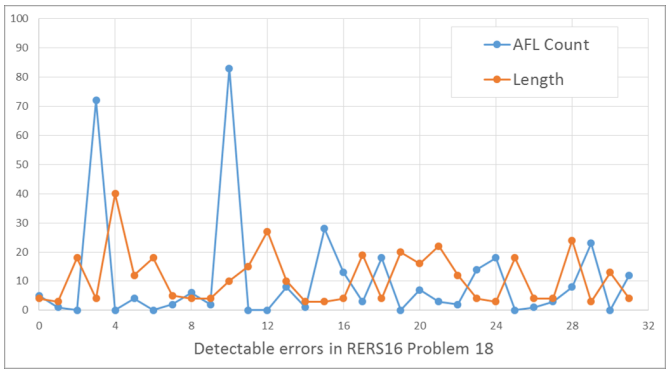

Fig. 8: AFL unique findings versus error pattern length

One of the points drawn from these observations is, **AFL is very much dependent on the code size**, which is also

indicated on its technical whitepaper. From our analysis, in a code spanning more than 100 Kilo Lines of Code (KLOC), the random mutated sequence finds hard to target an error pattern. The number of times a particular error is reached is also proportional to the pattern length as seen in Fig. 8.

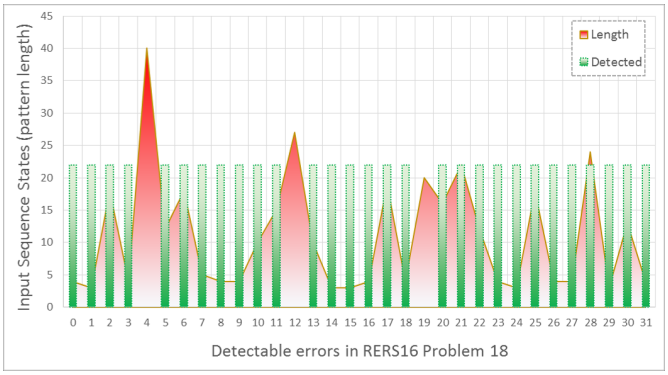

Fig. 9: KLEE reachability versus input sequence length, showing a maximum threshold (∼22) for error detection.

Unlike AFL which is dependent on code size, **KLEE is dependent on the pattern length** as seen from the results of Problem 11. Since KLEE is based internally on a SAT solver, a large pattern length can quickly become unruly even with tree trimming and other advanced expansion-space reduction techniques. The positive correlation of pattern length with KLEE reachability for RERS16 Problem 18 is shown in Fig. 9. The input sequence states is plotted for the reachable errors. It points to the fact that probability of reaching errors diminishes with increasing pattern length. From the figure, it is easy to find out that the probabilistic threshold is around 22 for this particular code by the KLEE tool. Error_63 (point-19) and Error_93 (point-28) are outliers.

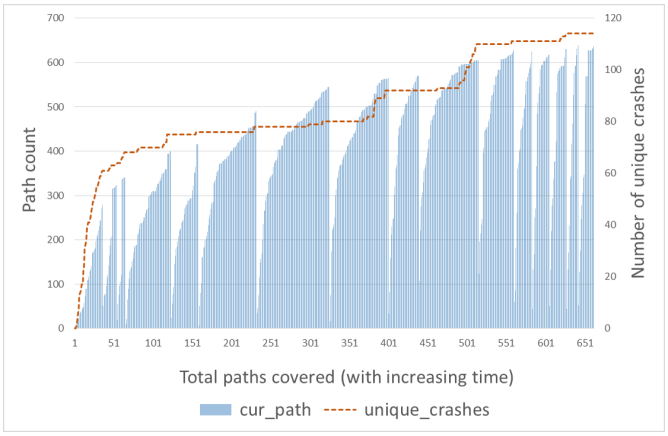

Fig. 10: AFL execution plot for RERS16 Problem 16 (cur_path refers to current path)

Lastly, KLEE was run exhaustively on all the selected problems, however, AFL, being a fuzzer, is dependent on random pattern generation. It finds a lot of errors initially, but gradually goes into a diminishing return plateau as shown in Fig. 10 (in orange). The plots of new unique error per unit runtime were generated at regular intervals. The fuzzer is aborted when the slope becomes flat, after sufficient path coverage. The time for which AFL was run cannot be completely justified based on the problem size scaling, and was decided based on the feasibility in the experimental setup. The typical AFL runtime status curve of Fig. 10 also shows the explored path counts per cycle (in blue). Paths refer to discovered sequences by fuzzing the input data. More information on AFL parameters can be found in status_screen.txt file in the AFL source's document directory. We observe, for the initial cycles, the number of paths explored are more (wider seesaw structure). The unique paths covered by AFL diminishes after the total path reach 550 for this case. As expected, it is also the region where the number of unique crashes diminishes, and we abort the fuzzing operation.

7 Conclusion and the Future Impressions

We have discussed exhaustively on tools, AFL and KLEE, their usage and critical analysis on their comparison based on the results. AFL, with shady working details known yet, manage to provide reasonable code coverage in short period of time, is a go-to tool for small to medium size simple-softwares. In order to test large size and complex softwares, enhanced and elegant tools such as KLEE is a front runner. This was established from the fact that there are many errors reached only by KLEE for medium to large scale complex problems. One problem we faced while comparing the tools is that their workings differ drastically, hence it is not easy to pin down the particular observation to the particular feature of the tool.

Many challenges exists for KLEE ranging from simple ones such as unsupported data types (float, double) to more complicated ones such as support for multi-threaded programs which are very common these days with many fields trying to adopt it as a way to decrease program time. KLEE being actively pursued by developers provides numerous opportunities in the future. Three important field of application would be to do testing of embedded softwares (firmwares), pre-silicon validation [5] (validation activities performed on a simulation or emulation model that transpires prior to fabricating actual silicon) and analyzing malwares. Software testing is an active and ongoing research field with many novel ideas being explored to simplify, enhance and take its automation to the next level. Making it applicable to a larger code base is one of its core aim.

References

[1] Tassey, Gregory. *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology, RTI Project 7007.011 (2002).

[2] https://lcamtuf.blogspot.nl/2014/08/binary-fuzzing-strategies-what-works.html

[3] Erete, Ikpeme, and Alessandro Orso. *Optimizing constraint solving to better support symbolic execution*. Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on. IEEE, 2011.

[4] Crisitan Cadar, Daniel Dunbar, Dawson Engler. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. OSDI. Vol 8. 2008

[5] Kannavara, Raghudeep, et al. *Challenges and opportunities with concolic testing*. Aerospace and Electronics Conference (NAECON), 2015 National. IEEE, 2015.

[6] http://klee.github.io/docs/files/

APPENDIX

Error #	Problem Number						
	10	11	12	13	15	16	18
0	0	0	3	0	0	0	4
1	0	0	3	14	0	11	0
2	0	0	0	0	0	0	0
3	0	0	0	0	4	0	3
4	0	0	0	0	0	3	0
5	0	25	0	0	0	0	0
6	0	18	9	3	0	0	0
7	0	0	0	0	0	8	0
9	0	0	0	0	0	0	0
10	0	0	0	0	4	3	18
11	0	9	0	0	0	0	4
12	0	55	0	14	0	3	0
13	0	0	0	3	0	0	40
14	4	0	0	0	0	0	12
15	3	0	5	3	11	13	0
16	5	0	0	0	4	0	18
17	0	0	0	0	0	0	5
18	0	0	0	3	0	11	0
19	0	0	3	0	0	0	0
20	2	0	0	31	0	3	4
21	0	0	0	3	0	0	0
22	4	0	0	0	4	0	0
23	0	0	0	10	11	10	0
24	5	0	16	0	0	0	4
25	5	11	0	3	0	0	10
26	4	23	10	0	4	0	0
27	0	0	3	3	0	0	0
28	6	0	0	0	0	0	15
29	2	0	0	0	19	23	0
30	7	0	0	4	0	0	27
31	5	92	0	0	0	0	0
32	0	5	0	0	17	3	0
33	0	0	0	4	0	0	0

Error #	Problem Number						
	10	11	12	13	15	16	18
34	0	0	6	0	0	3	0
35	3	0	0	0	0	3	0
36	0	0	0	0	0	0	0
37	5	0	0	0	0	0	0
38	5	23	0	0	0	24	10
39	3	43	4	0	0	0	3
40	0	14	0	10	8	0	0
41	0	0	11	0	0	0	0
42	3	30	0	0	7	0	0
43	3	0	0	0	0	0	0
44	0	0	10	0	0	12	0
45	0	0	0	0	4	0	0
46	0	0	10	0	4	0	3
47	0	0	0	3	23	0	0
48	0	0	0	0	0	14	4
49	0	46	0	0	8	0	0
50	2	0	0	4	0	0	0
52	5	19	0	0	0	0	0
53	5	54	0	11	0	0	19
54	4	0	0	13	0	0	0
55	4	0	0	0	0	9	0
56	0	0	4	0	0	11	0
57	0	0	13	0	11	0	0
58	6	0	0	0	11	0	0
59	0	0	0	4	0	0	0
60	0	0	0	0	23	0	0
61	6	0	0	0	4	15	0
62	6	48	0	0	0	0	4
63	0	21	0	0	4	0	20
64	0	0	4	20	0	0	0
65	7	0	0	0	3	4	0
66	0	0	0	0	0	0	0
67	0	0	0	0	5	3	0

Error #	Problem Number						
	10	11	12	13	15	16	18
68	0	0	0	0	0	3	0
69	0	0	0	0	0	0	0
70	0	0	0	0	0	4	16
71	0	0	4	0	0	0	22
72	0	0	10	0	12	11	0
73	0	0	0	0	17	0	0
75	3	0	0	0	3	13	0
76	4	11	0	0	0	0	12
77	3	0	0	0	0	0	0
78	7	0	0	4	3	0	4
79	0	0	5	0	0	0	0
80	2	0	0	0	0	0	0
81	0	0	0	10	0	3	3
82	0	0	0	0	4	0	0
83	0	0	0	3	0	0	18
84	0	0	0	0	0	18	0
85	0	10	0	0	0	0	4
86	7	0	0	0	0	3	0
87	7	0	0	3	14	0	0
88	7	0	0	0	0	0	0
89	2	0	7	0	4	0	0
90	7	0	0	0	0	0	0
91	5	14	0	0	11	0	4
92	0	0	0	4	0	3	0
93	6	13	6	7	9	14	24
94	7	0	0	10	0	0	0
95	3	0	0	8	3	4	3
96	2	0	0	4	0	0	13
97	6	0	0	0	0	0	0
98	5	22	0	12	0	3	0
99	4	0	0	0	0	0	4

Legend		
Both	Only KLEE	Not reachable
None	Only AFL	

RERS 2016 problems' charateristics			
Prob #	Size	KLOC	Type
10	small	1.6	plain, simple
11	small	2.3	
12	small	15.6	
13	medium	118.3	arithmetic, medium
15	medium	155.7	
16	large	674.9	data structures, hard
18	large	593.9	