fuzzing

Rick Proost - 4173619

Wim Spaargaren - 4178068

Introduction

This document contains an explanation about the differences between fuzzing and concolic execution. First Fuzzing is explained, after that an explanation on concolic testing is provided. In the next paragraph the fuzzing tool AFL will be explained followed by the concolic testing tool called KLEE. These tools are used to experiment on RERS reachability from which the results are explained in the last paragraph.

Fuzzing

What is fuzzing

Fuzz testing or fuzzing is a software testing technique used to discover coding errors and security loopholes in software, operating systems or networks. This is done by inputting massive amounts of random data, called fuzz, to the system in an attempt to make it crash. Typically, fuzzers are used to test programs that take structured inputs. This structure can be specified for most fuzzers and is used to distinguish valid from invalid input. Fuzzers can then use fuzzing strategies[1] to randomize valid input for testing software. A fuzzer is effective because huge amounts of test data can be generated in a relatively short time, and many edge cases get handled quickly. A common pitfall is that fuzzers detect every error possible, but fuzzers will not catch every possible vulnerability.

Common strategies for fuzzing

- Walking bit flips: sequential, ordered bit flips
- Walking byte flips: extension on bit flip, per 8-, 16, 32-bit
- Simple arithmetics: given input +- x
- Known integers: max integer, 256, 1024, etc

Different fuzzers

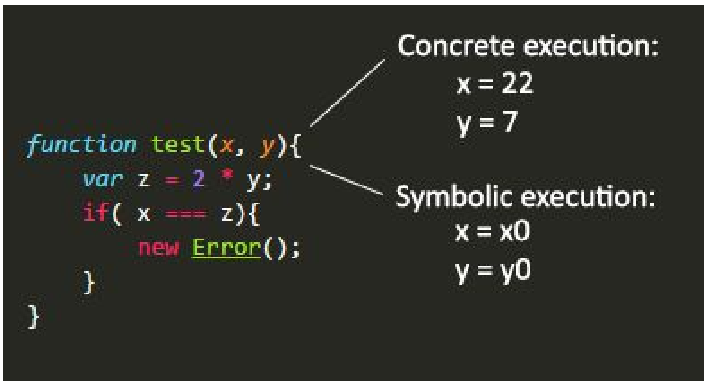
There are three main types of fuzzers which are combinable.
A fuzzer can be mutation-based, generating new input from modifying existing input.
A fuzzer is dumb or smart depending on whether it is aware of the structure of the input. When the fuzzer is not aware of the structure, generating valid input can take a long time.
A fuzzer can be white-, grey-, or blackbox, which depends on the depth of awareness of the program structure.

Blackbox means that the fuzzer is not aware of anything related to the program, generating random input, but can learn from the response.
Whitebox fuzzing means that the fuzzer uses analyzation tools to systematically verify that all target code paths are hit. Most of the times whitebox fuzzers find bugs deeper within the software more quickly than blackbox fuzzing, but generating valid test data can also become slow.
Greybox fuzzing means using the programs instrumentation to measure effectiveness of the input.

Concolic execution

Concolic execution is a hybrid software verification technique, which combines both symbolic and concrete execution. To get a better understanding of concolic execution, both symbolic and concrete execution will be further explained.
Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute. Symbolic execution does not actually execute code, but only analyze it. Symbolic execution only uses symbolic values for inputs, instead of obtaining actual inputs as normal execution of the program would do.
Concrete execution is execution with actual values, just like normal execution. So instead of using symbolic values like symbolic execution, concrete execution uses real values to execute the program.
Concolic execution combines these two execution methods. When concolic execution arrives at a branch in the code, it asks for the symbolic value. This symbolic value is given to a SMT solver, to find a satisfying assignment for that symbolic value. At this point the concolic execution has a concrete value to use. By using this symbolic value from the symbolic execution not every concrete value has to be tried to reach every statement, but concrete values to reach every statement can be derived from these symbolic values. By using the symbolic execution with concrete values, concolic execution can be used to generate tests which aim is to give a coverage as high as possible. Concolic execution programs, like KLEE for example, are able to do this.

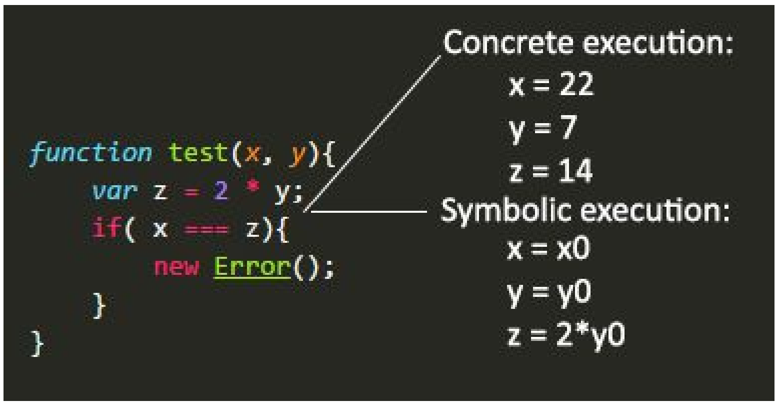
To further illustrate concolic execution an example is given. Concolic execution will be carried out on a function test, as can be seen in Figure 1. Suppose x and y have a value where x = 22 and y = 7. The symbolic execution will give symbolic values to x and y, namely x = x0 and y = y0.



```
function test(x, y){
    var z = 2 * y;
    if( x === z){
        new Error();
    }
}
```

Figure 1: Start of concolic execution

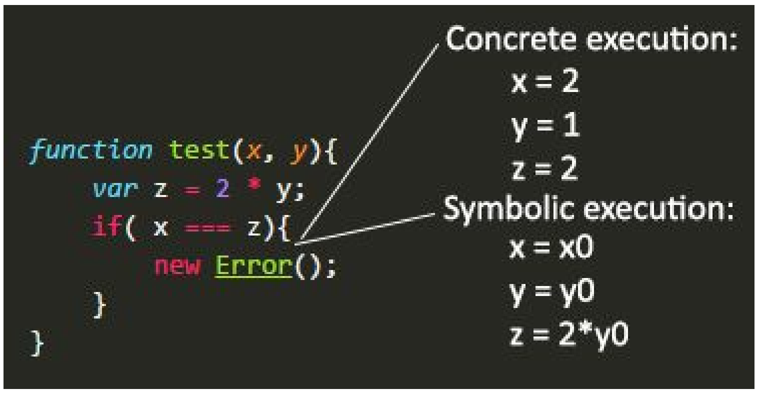
Now concolic execution will move to the next statement and give a value to z. Since z = 2 * y, the value 14 will be assigned to z. As can be seen in Figure 2. The symbolic execution will assign symbolic value to z where z = 2*y0. Since x === z is false, the concrete execution will take the else branch. Now symbolic execution will know that the path condition for the else branch is x0 != 2 * y0.



```
function test(x, y){
    var z = 2 * y;
    if( x === z){
        new Error();
    }
}
```

Figure 2: Concolic execution assignment of z

At this point concolic execution decides it want to explore the if branch as well. Which can be reached if x === z, thus concrete execution needs to generate concrete inputs in order to explore it. To do this it can take the negation of the symbolic execution path condition which was found for the else branch. The negation of x0 != 2 * y0 is x0 = 2 * y0. Now concolic execution calls an SMT solver to find a satisfying assignment of this constraint. Suppose in this example, SMT finds x = 2 and y = 1, which satisfies x0 = 2 * y0. Now concolic execution will run the function with this input and takes the if branch. At this point it reached the error in the function and all branches of this function are executed.



```
function test(x, y){
    var z = 2 * y;
    if( x === z){
        new Error();
    }
}
```

Figure 3: New concrete values assigned by the SMT solver

American fuzzy lop

The fuzzer used for this report is American fuzzy lop (AFL)[2], this fuzzer uses coverage-based greybox fuzzing (CGF)[3]. A random testing approach that requires no program analysis. A new test is generated by slightly mutating a seed input. If the test exercises a new and interesting path, it is added to the set of seeds; otherwise, it is discarded.
With AFL it is possible to compile the target code, define the structure of the test input and run the fuzzer. For every crash, the generated input is saved, and for every unique crash the input is saved in another directory. Crashes are considered "unique" if the associated execution paths involve any state transitions not seen in previously-recorded faults. The

generated files can be reused in AFL for further debugging, it is also possible to visualize data processing of AFL by plotting.

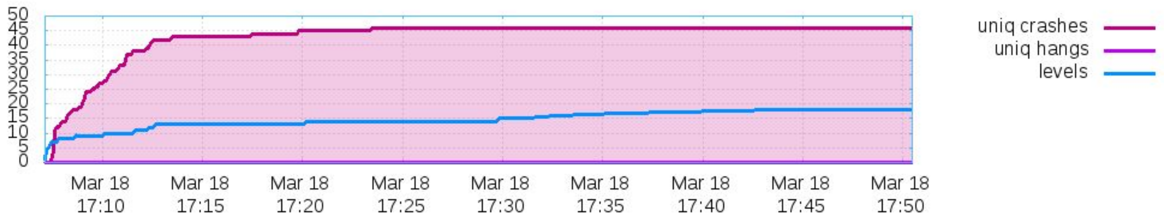


Figure 4: Plot AFL execution data

KLEE

KLEE is an concolic execution engine based on LLVM architecture [5]. KLEE is capable of automatically generating tests that achieve high coverage [6]. KLEE is able to do so by using concolic execution, as described before. At a high-level, KLEE works as follows. Instead of running code on manually- or randomly-constructed input, KLEE runs on symbolic input, initially allowed to be anything. KLEE substitute program inputs with symbolic values and replace corresponding concrete program operations with ones that manipulate symbolic values. KLEE is proven to be good at automatically generating tests that, on average, cover over 90% of the lines in programs [6].

Analysis

Both KLEE and AFL were ran on RERS17TrainingReachability/Problem10/Problem10.c. Other problems were checked, but these problems took too long for KLEE to complete or generated too much test files, which made Linux crash.

AFL

The output of AFL is divided in three different folders and the root of the findings folder. The first part, in the root of your findings folder, are the details of the AFL session, containing the AFL fuzzer status screen details and data to visualize (plot) the session details.
The other three folders are "crashes", "hangs" and "queue". Queue contains the generated test cases for every distinctive execution path, plus all the starting files given by the user. Both the crashes and hangs folders contain the unique crashes and hangs test files.
The generated output can be used to debug the software, by inserting it manually or with the "crash exploration" mode of AFL. In "crash exploration" mode, the fuzzer takes one or more crashing test cases as the input, and uses it to very quickly enumerate all code paths that can be reached in the program, while keeping it in the crashing state. The output is a small corpus of files that can be very rapidly examined to see what degree of control can be obtained over the faulting address.

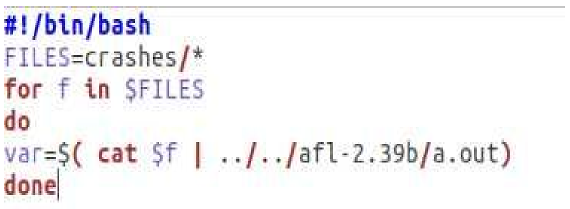
KLEE

The output of klee is putted in a single folder called klee-out. This folder contains a number of files. First there is a file called info which contains information about the run time of KLEE and the number of generated tests and completed paths. After that the folder contains two stats files, called run.istats and run.stats. These files contain statistics about the program which was analyzed by KLEE. The run.stats file can produce a coverage report with the klee-stats tool. After this there are .ktest files. These files contain test cases for the program which is ran. These .ktest files can be run again to test the program.

Results

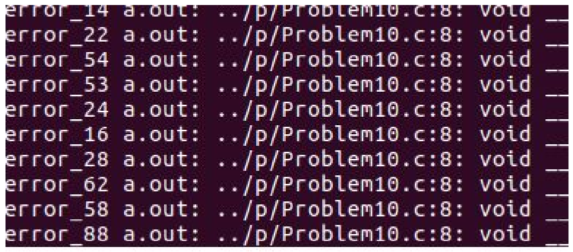
AFL

To know which errors were reached for Problem10 of the RERS challenge by AFL, it is required to run the unique crashes to get the output. Because there were 60+ unique crashes the following small script gave us the desired results.



```
#!/bin/bash
FILES=crashes/*
for f in $FILES
do
var=$( cat $f | ../../afl-2.39b/a.out)
done
```

Figure 5: Script



```
error_14 a.out: ../p/Problem10.c:8: void
error_22 a.out: ../p/Problem10.c:8: void
error_54 a.out: ../p/Problem10.c:8: void
error_53 a.out: ../p/Problem10.c:8: void
error_24 a.out: ../p/Problem10.c:8: void
error_16 a.out: ../p/Problem10.c:8: void
error_28 a.out: ../p/Problem10.c:8: void
error_62 a.out: ../p/Problem10.c:8: void
error_58 a.out: ../p/Problem10.c:8: void
error_88 a.out: ../p/Problem10.c:8: void
```

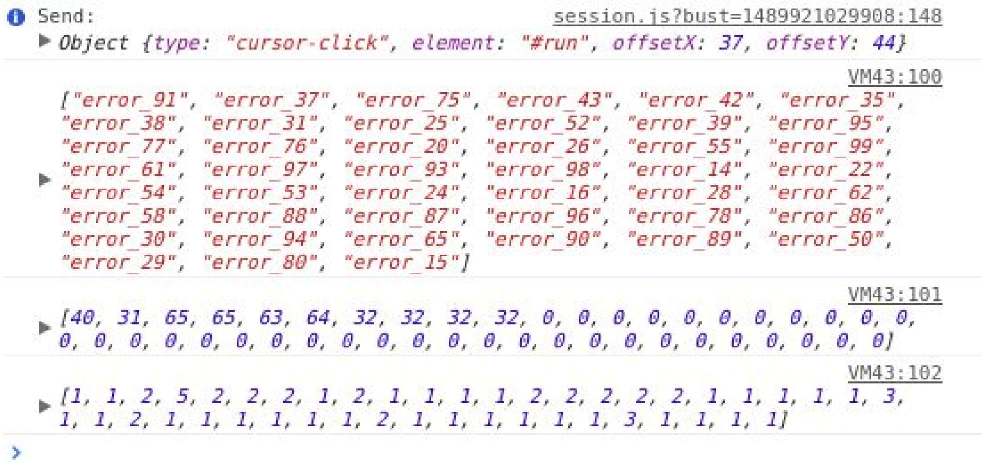
Figure 6: Subset of results by AFL

KLEE

When KLEE is done running, it outputs all different errors it has found. So to know which errors were reached by KLEE, this output can simply be copied.

Differences

To see the differences in found errors by KLEE and AFL, a small javascript script was written. This script first combined all errors reached by KLEE and AFL and after that outputs how many times KLEE and AFL reached those errors. This can be seen in Figure 7. KLEE reached only 10 errors and AFL reached 45 errors. As can be seen, there were no errors which KLEE found, but AFL did not. The reason KLEE only found 10 errors is due to the fact that program[length] in function main() was set to 20 as in the tutorial. This way a restriction is set on the symbolic values KLEE will use. This is why KLEE does not try every available path in the program.



```
Send:                                          session.js?bust=1489921029908:148
▶ Object {type: "cursor-click", element: "#run", offsetX: 37, offsetY: 44}
                                                                      VM43:100
  ["error_91", "error_37", "error_75", "error_43", "error_42", "error_35",
  "error_38", "error_31", "error_25", "error_52", "error_39", "error_95",
  "error_77", "error_76", "error_20", "error_26", "error_55", "error_99",
▶ "error_61", "error_97", "error_93", "error_98", "error_14", "error_22",
  "error_54", "error_53", "error_24", "error_16", "error_28", "error_62",
  "error_58", "error_88", "error_87", "error_96", "error_78", "error_86",
  "error_30", "error_94", "error_65", "error_90", "error_89", "error_50",
  "error_29", "error_80", "error_15"]
                                                                      VM43:101
▶ [40, 31, 65, 65, 63, 64, 32, 32, 32, 32, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
                                                                      VM43:102
▶ [1, 1, 2, 5, 2, 2, 2, 1, 2, 1, 1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 3,
  1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 3, 1, 1, 1, 1]
>
```

Figure 7: Errors found by KLEE and AFL respectively

Reflect on the (future) usability of AFL and KLEE

AFL is still actively developed, there are many forks made to support more languages or just improve the fuzzer, like AFLFast[3][7], where the fork was eventually merged to improve AFL itself. Fuzzing is one of the most powerful and proven strategies for identifying security issues in real-world software. It is responsible for the vast majority of remote code execution found to date in security-critical software. Because AFL is one of the best performing fuzzers to date, we think it will be used a lot more in the future. However, there are some known shortcomings like:

- No direct support for fuzzing network services, background daemons, or interactive apps that require UI interaction to work
- AFL does not output human-readable coverage data (there are 3rd party tools)
- Occasionally, sentient machines rise against their creators. If this happens to you, please consult http://lcamtuf.coredump.cx/prep/.

KLEE has a lot of potential. Since the long-term goal of KLEE is to take an arbitrary program and routinely get 90%+ code coverage. If the developers of KLEE manage to succeed in this goal and the generation of this coverage can be reached in a limited amount of time, we think it could be used widely for testing purposes.

References

1. Binary fuzzing strategies: what works, what doesn't. Retrieved from: https://lcamtuf.blogspot.nl/2014/08/binary-fuzzing-strategies-what-works.html
2. American fuzzy lop. Retrieved from: http://lcamtuf.coredump.cx/afl/
3. Coverage-based Greybox Fuzzing as Markov Chain. Retrieved from: https://www.comp.nus.edu.sg/~mboehme/paper/CCS16.pdf
4. Effective Fuzzing Strategies. Retrieved from: http://resources.sei.cmu.edu/asset_files/Presentation/2010_017_001_53566.pdf
5. The LLVM Compiler Infrastructure. Retrieved from: http://llvm.org/
6. KLEE: Unassisted and Automatic Gneration of High-Coverage Tests for Complex Systems Programs. Retrieved from: http://www.doc.ic.ac.uk/~cristic/papers/klee-osdi-08.pdf
7. AFLFast. Retrieved from: https://github.com/mboehme/aflfast