TU Delft, EWI
Software Testing & Reverse Engineering, Assignment 1

Jeroen Vrijenhoef 1307037 j.j.m.vrijenhoef@student.tudelft.nl
Rasmus Välling 4561058 r.valling-1@student.tudelft.nl
Michal Loin 4587324 m.f.loin-1@student.tudelft.nl

March 20, 2017

1 Introduction

Software testing and reverse engineering are important concepts, when talking about software security. These techniques are used by programmers and testers alike, in order to find vulnerabilities in their software and secure them. However, potential adversaries can also use those techniques to find potential exploits, unblock hidden parts of software, remove copy protection or simply understand the main principles about the code.

There are multiple ways reverse engineering can be performed. The most common division is to consider black-box and white-box approaches. Black-box reverse engineering is typically used to obtain any information about the software , when white-box can help understanding obfuscated code. As a black-box we usually consider software we have no knowledge about. There is usually no possibility to check internal operation in any other way, than providing a set of inputs and observing the behavior. This approach is widely used by adversaries, since more often they have no insights into the software. One should notice, that actions of attackers can inspire the defenders, which can test security in the same way, to make sure that such a given approach is not going to be successful. A white-box approach differs significantly in the concept. Here, the testers are provided with a lot of additional information, such as implementation's source code or the design of the system. Such information can be difficult to obtain, for example by decompiling the product, but certainly can increase number of possibilities and overall efficiency of the process.

2 Fuzzing

Fuzz testing, also known as fuzzing, is an automated testing technique. The main idea behind it is to provide the system with unexpected, random or invalid inputs, in order to cause a system crash, memory leak or other problem. Program using any input, including commands, files etc. can be tested using this approach. Without any additions, fuzzing is a black-box testing. The testing tool takes correctly formed inputs to the system under test, usually provided by the user, and changes them randomly, in order find unwanted behavior. Native fuzzers are not very efficient, because it might be difficult to trigger different parts of the program, resulting in poor code coverage. Let us consider a simple example:

```
void example_test(int x) {
    if (x == 0xDEADBEEF)
        assert(0);
}
```

In order to find an error, the tool would have to provide the exact value of x, which is very unlikely. Therefore, modern fuzzers introduce additional mechanisms, that increase their performance.

One of possible improvements is allowing the system performing tests to have inside view of the implementation's code. Such an approach is called white-box fuzzing. The testing software would investigate the code and generate inputs, that would cause as high code coverage as possible. The principle of operation of such fuzzers is not very complicated. At first, the tested code is scanned for conditional statements, which are collected and solved. It enables executing the program in a way, that guarantees high branch coverage. Later, a program is executed with collected inputs and the behavior is observed. Considering the example above, the fuzzer would identify the conditional statement and generate a test to make sure it would be both fulfilled and unsatisfied. In theory, this could lead to complete code coverage. However, in most cases this is impossible. The first reason for that could be the size of a program and extremely large number of possible paths. The other reason is that internal constraint solvers could have problems with difficult complex program statements, such as pointer operations [4].

3 Concolic Execution

Concrete Symbolic Execution, or Concolic execution for short performs a combination of concrete dynamic execution and symbolic execution for software testing. This means that while the program is executed a symbolic state as well as the concrete state of program variables is kept. The concrete state maps all variables with their concrete values while the symbolic state only maps variables that do not have a concrete value. When the program reaches a conditional branch the symbolic constraints on the inputs for this branch are gathered and a constraint solver is used to find a solution for the inputs where the conditional branch will lead to a new execution path (see also [2]).

Because of this constraint solving ability concolic execution is very good at solving conditional statements that require specific input values. For example a check if variable x is equal to value 0xDEADBEEF would be easily found by a constraint solver (see the code example above). Where a regular fuzzer that relies only on concrete input values could take a lot of time before it produces the required input by chance.

One serious limitation of concolic execution however is how it handles unbounded loops. The constraint solver will find an infinite amount of solutions for these unbounded statements causing the symbolic execution engine to run forever. Other well documented limitations on concolic execution are the path explosion problem and the environment problem [2]. Given the huge amount of possible program paths in all but the smallest programs and it's exponential increase with every static code branch concolic execution can run out of resources or be very slow in covering the entire code. The environment problem is related to the possibility of the program executing system specific library calls that are not under control by the symbolic execution engine as this can lead to consistency problems when attempting to replicate generated crash results.

4 Tools

Two relatively different tools were used for testing experiments in this project. For fuzzing American fuzzy lop [8] is used and for concolic execution KLEE [13] has been used. Both of the tools are briefly explained in this section giving a short overview, how they work, what are they good for, what are their limitations and what results have they shown in practice.

4.1 American fuzzy lop

American fuzzy lop (AFL) is an instrumentation-guided code coverage driven genetic fuzzer capable of synthesizing complex file semantics [6] [8]. It also comes with a unique crash explorer, a test case minimizer, a fault-triggering allocator, and a syntax analyzer [6] [8].

Genetic algorithm is a natural selection inspired evolutionary algorithm that makes use of bio-inspired operators such as mutation, crossover and selection. AFL uses several mutation operations: [5]

- bit-flipping - flip the value of random bits or sets of bits in the input,
- arithmetic - walk the power of two bit values and add or subtract,
- byte-flipping - reverse byte ordering,
- interesting values - 0s and 1s, high-order bits, and maximum values.

Code coverage measures the degree to which a source code has been executed by a test suite. AFL aims for a high code coverage. Higher code coverage also means it is more likely to find bugs.

AFL has several benefits. The engine makes use of a number of strategies for fuzzing and high-gain test case preprocessing that have been thoroughly researched [5]. Low-level compile-time, binary-only instrumentation and other optimizations make it relatively fast compared to other fuzzers. It is easy to set up and get started with as there are relatively few instructions you need to give it and parameters to fine-tune.

On the other hand, AFL has several limitations to it. First of all, as it is instrumentation-guided, the source code or equivalent is nice to have for testing. The tool is capable of fuzzing binaries but such process is much slower. Secondly, it has limited capabilities handling programs with custom signal handlers, when the actual format of tested data is wrapped with encryption, checksums, cryptographic signatures, or compression. Several other known limitations and areas needing improvement are explained in README [7].

AFL's trophy case of found bugs include many notable software such as commonly used browsers like Mozilla Firefox, Internet Explorer, and Apple Safari; some brand name vulnerabilities such as Stagefright bug in Android and many more [8].

4.2 KLEE

KLEE is a software testing tool based on symbolic execution and constraint solving techniques [1]. It automatically generates the test suites and aims for high code coverage. KLEE uses instrumentation approach to testing which means that callback hooks are inserted in the program such that symbolic execution is done in background during normal execution of program. It has two goals:

- hit every line of executable code in the program
- detect at each dangerous operation if any input value exists that could cause an error

There are several limitations to KLEE:

- does not scale well when the number of paths through code is large
- handling code that interacts with its surrounding environment
- limited by the power of the constraint solvers it uses
- source code or equivalent is required for precise symbolic execution.

It employs several heuristics to overcome some of its inherent limitations. To combat exponential search space it uses Random Path Selection and Coverage-Optimized Search. To speed up constraint solving which is inherently expensive and invoked at every branch it eliminates irrelevant constraints and caches solutions. The goal of environment interaction is to explore all possible legal interactions with the environment. Experiments on GNU Coreutils and Busybox Suite for Embedded Devices resulted in higher code coverage than developers manually written test suites and also revealed several new bugs [1].

5 Experiments with AFL & KLEE

Experiments were conducted on the Rigorous Examination of Reactive Systems 2016 (RERS2016) [3] reachability problem set Problem10 to Problem18. Every single program was tested for a maximum of 1 hour with each tool. For the last three problems extended period of testing 12 hours was conducted with AFL and KLEE. RERS2016 problems were evenly divided between group members. Every problem was tested with KLEE and AFL on the same machine to get consistent results with regard to how much time was given for each tool.

AFL version 2.39b was used. The instrumentation was injected to source code and then compiled with compiler wrappers. The source code was compiled into a binary with

```
afl-gcc Problem1X.c
```

Test input files contained the legal inputs specified in each problem source code followed by a blank line. Fuzzing was run by

```
afl-fuzz -i tests/ -o findings/ ./a.out
```

KLEE was run in a docker container provided by the tool developers. In a similar manner to AFL the binary was instrumented to use KLEE by modifying the source code and then compiling with a KLEE compiler wrappers. Then the programs were executed with the KLEE tool. Testing was conducted up to 1 hour per program and up to 12 hours for problems 16, 17, and 18, but most tests finished execution much earlier.

6 Analysis Results

Outputs

AFL shows a status screen while it is running containing relevant statistics. It also creates several directories and files that are updated in real time while the fuzzing is in process. Queue folder contains the test cases for every distinctive execution path. Crashes folder contains unique test cases that caused the tested program to receive a fatal signal. Hangs folder contains the test cases that caused the program to timeout.

Overall statistics can be seen in `fuzzer_stats` and `plot_data` text files. Each of the crash file contains

AFL	KLEE
{ 2, 3 }	{ 2, 1 }
{ 2, 3, 3 }	{ 2, 5 }

Table 1: Missed test cases

the input given to the program that produced the crash and the name includes different information such as the ID of the crash, crash signal received from the program, operation used by the fuzzer etcetera. Files in hangs and queue folder are similarly structured. The crash files can be used to verify if they are actually exploitable. It is then necessary to run the tested program and give it the same input as in the crash file.

KLEE will output files to `klee-output-N` folder. There are several text files which contain general information (duration, number of paths, etc.), human readable LLVM assembly code that was run, and different files associated with each test case that KLEE ran. Each of the paths generated by KLEE will produce several path files of the format `test<N>.<type>`. Overall statistics can be seen by using `klee-stat` tool. Klee test files can be handily replayed back to the tested program. In order to do this some environment variables need to be changed and the program compiled with extra flags.

6.1 Differences

To find any differences in reachable error codes between AFL and KLEE, Problem10 from the ReachabilityProblems2016 set was examined. Both tools were ran for a maximum time of one hour (KLEE finished before that time). To ease this analysis we extracted the input sequences from the results from both tools and sorted them, first according to sequence size, then according to input values. Table 1 shows two missed test cases per tool.

There are two big differences between the results from the two tools. The first is that for a certain error state, AFL will be able to find multiple different inputs that reach this state, where KLEE will not as they present no new branch path for KLEE to take. The second big difference is that KLEE is overall better in finding larger sequences of inputs that reach error states (see figure 1 in the Appendix). This result can be explained by the fact that AFL relies on randomly generated input to find new program paths (and thus error states). If a large number of inputs being a certain value is required to reach a certain error state, the chance of AFL randomly generating these inputs is very slim. Whereas KLEE with it's constraint solver will have no problem finding these values.

6.1.1 General Overview

The tests were performed on different machines. However, every problem was tested by both AFL and KLEE on the same machine, in order to keep the credibility of outcomes. After execution of the tests the results were collected. In case of AFL, we focused on the following characteristics:

- Total paths - total number of entries in the queue
- Paths found - number of entries discovered through local fuzzing
- Max depth - number of test mutations from the initial set
- Unique crashes - number of unique crashes recorded
- Unique hangs - number of unique hangs recorded

We are using the terms used by the authors of AFL, therefore some of them might be confusing. As a queue, one should understand a process of fuzzer going over all interesting test cases discovered so far and fuzzing them. Definition of path is a trace that the program has to overcome, while executing a particular test case. It is important to notice that AFL distinguishes between crashes and hangs. A crash happens when the software detects a fatal signal, for example *Abort*. A hang on the other hand is an ambiguous state. It basically means that the tested program timed out. However, it is difficult to distinguish between a timeout caused by error, such as deadlock, and complex program structure, that could after long time of execution end up as crash or successful operation. Results were collected from generated *fuzzer_stats* files and can be observed in Table 2. Despite the fact that first two problems had significantly smaller amount of lines of code, AFL was able to generate more tests there. Also, they are the only problem, where any crashes were discovered - the greater part finished with a hang.

Results collected from KLEE tests can be found in Table 3. The statistics were gathered from generated *info* and *run.stats* files. We can observe, that Problem 10 and Problem 11 produced the largest

Name	Total paths	Paths found	Max depth	Unique crashes	Unique hangs
Problem 10	107	102	29	58	1
Problem 11	655	650	19	104	0
Problem 12	98	88	10	0	0
Problem 13	20	10	8	0	5
Problem 14	23	13	5	0	5
Problem 15	52	32	8	0	5
Problem 16	67	66	12	0	9
Problem 17	21	20	5	0	7
Problem 18	13	12	10	0	4

Table 2: AFL results

Name	Explored paths	Completed paths	Generated tests	Instruction coverage	Branch coverage
Problem 10	81238	81238	27652	90.1%	71.55%
Problem 11	525974	525974	384939	88.7%	66.73%
Problem 12	54	54	29	3.86%	4.41%
Problem 13	2	2	2	0.84%	2.64%
Problem 14	2	2	2	1.12%	3.05%
Problem 15	6	6	4	0.53%	1.06%
Problem 16	47	47	25	0.37%	1.28%
Problem 17	4	4	3	0.09%	0.86%
Problem 18	2	2	2	0.07%	0.25%

Table 3: KLEE results

amount of tests, what was similar case to results produced by AFL. Those two problems were examined best, achieving high instruction and branch coverage. There are several possible reasons for that. At first, we can observe significant change in complexity of the problems. The number of inputs increased from 5 to 10, and number of lines of code increased 10 times. That could cause a path explosion and prevent KLEE from generating best results. Another hypothesis is that first two problem have internal structure, that is more suitable for KLEE testing than other problems. As described in section 4.2, there exist some software elements that can drastically increase the number of paths, for example loops, and therefore limit the capabilities of KLEE.

6.2 Reflection and Future

Despite having an impressive hall of bugs AFL as a tool will most likely become obsolete in the future as a tool to find software bugs. As other tools that rely on different underlying (combinations of) technologies will likely find all the 'low hanging fruit' bugs that AFL is able to find in a much faster and efficient way.

In the case of KLEE it is slightly different. As this tool relies on concolic execution it will be better equipped to identifying difficult to find bugs that reside deep in code branch trees. However, because Klee relies on concolic execution alone to find it's bugs (which is not very efficient in solving branches that require specific inputs and has some other inherent problems like path explosion and the environment problem) it's usability will be surpassed by other tools which employ combinations of these techniques in a smarter and more efficient way.

One significant issue discovered in testing was large amount of produced files. In last three tests, involving longer execution time, produced outputs were so numerous, that the PC had troubles processing them.

7 ANGR

Angr is an open source binary analysis framework [11]. It implements several different state-of-the-art binary analysis techniques in the literature and current research. This is done in an attempt to systematize the field and encourage the development of next-generation binary analysis techniques. The framework facilitates the comparison between different static and dynamic analyses. The project is developed in Python and there is a docker container that works out of the box. Angr has a GUI Angr Management for doing the analysis [12].

Angr is composed of three different components, a binary loader, static analysis routines and symbolic execution engine. The symbolic execution engine in Angr tries to find input needed to reach certain paths. It interprets the application, tracks the constraints on variables and then when a required condition is triggered concretizes to obtain a possible input. The constraint solver converts from a set of constraints to a set of concrete values that would satisfy them. All of the limitations of symbolic execution also apply to the symbolic execution engine. Static analysis routines are implemented to guide symbolic execution. There are three different techniques implemented control-flow graph, data-flow analysis, and value-set analysis. Control-flow graph is used to visualise the flow of the program and can be used to find interesting parts of the program. Data-flow analysis is implemented to execute small slices of program to figure out the semantics of these parts. This helps narrow down possible executions and facilitates reversing. Value-set analysis tries to narrow down the values a certain variable can take [10] [9].

The advantages of Angr are that it is open source, it implements a wide array of different analysis techniques from current research efforts, it is versatile and well-encapsulated, expandable and architecture independent. It provides a framework for comparing current techniques and even for combining different techniques to achieve better testing and reverse engineering results. In some cases it can even provide a proof-of-concept exploit for discovered vulnerabilities. A disadvantage of Angr could be that it is implemented in python which might not achieve speeds that compiled binaries offer. In essence it cannot be directly compared to AFL and KLEE as it implements techniques from symbolic execution, fuzzing and concrete execution.

We installed Angr and tried to figure out how it can actually be applied to the RERS2016 problem set. However, due to time constraints we did not get any significant results.

References

[1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[2] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[3] RERS Community. The rers challenge 2016 - reachability problems, October 2016.

[4] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[5] lcamtuf. Binary fuzzing strategies: what works, what doesn't, August 2014.

[6] lcamtuf. Technical 'whitepaper' for afl-fuzz, March 2016.

[7] lcamtuf. Afl readme, March 2017.

[8] lcamtuf. american fuzzy lop (2.39b), March 2017.

[9] Shellphish. Black hat 2015 - angr presentation, July 2015.

[10] Shellphish. Defcon 23 - angr presentation, July 2015.

[11] Shellphish. Angr.io, March 2017.

[12] 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. 2016.

[13] The KLEE Team. Klee llvm execution engine, March 2017.

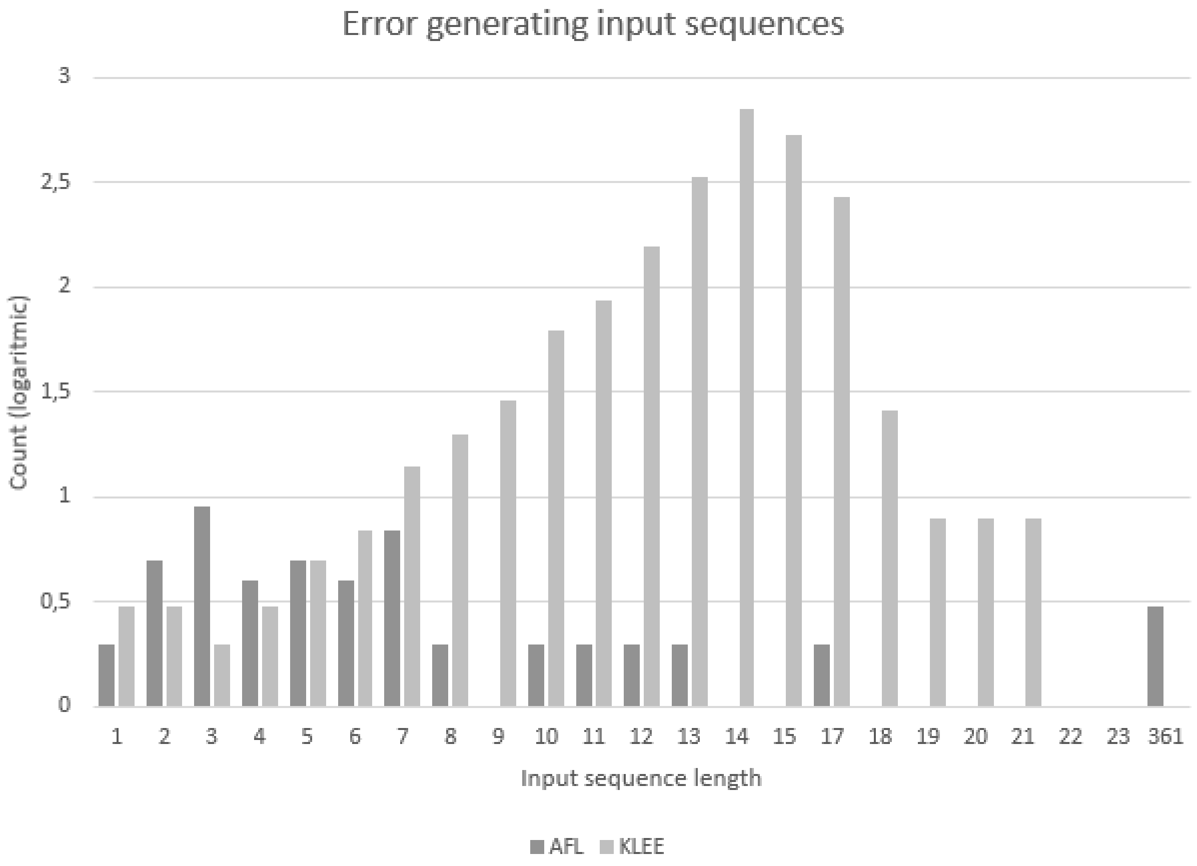

Figure 1: Error generating input sequence length versus count