

**[6 pages, not counting pp. 1, 2 & 9]**

## **Software Reversing Lab 1: AFL, KLEE and ANGR**

CS4110 Software testing & Reverse engineering



# Table of contents

1	Introduction .....	3
2	Methodology .....	3
3	Explanation of fuzzing and concolic execution.....	3
3.1	Fuzzing .....	3
3.2	Concolic execution .....	4
4	Description of the software .....	5
4.1	American Fuzzy Lop .....	5
4.2	KLEE LLVM Execution Engine .....	5
4.3	ANGR.....	5
4.4	Discussion .....	5
5	Experiments and comparison .....	6
5.1	American Fuzzy Lop .....	6
5.2	KLEE LLVM Execution Engine .....	7
5.3	ANGR.....	7
5.4	Discussion .....	8
6	Conclusion.....	8
	References .....	9

# 1 Introduction

This report is written to educate students about software reverse engineering techniques. The authors of this report have decided to describe and analyze both the two mandatory programs and the optional ANGR program for bonus points. The three reverse engineering programs under analysis include American Fuzzy Lop (AFL), KLEE and ANGR.

The purpose of this report is to describe techniques for finding bugs and explain the workings of the three programs that implement these techniques. In addition, the effectiveness of the programs is described. Furthermore, experiments with and output comparisons between programs will be performed. The differences between the programs will be discussed at the end of each chapter.

This report consists of the introduction in the first chapter. The second chapter contains the methodology that will explain how the analysis and comparison is done. Third, a description and explanation of the three reverse engineering programs is given, followed by a discussion. In the fourth chapter, the experiments and comparison are explained. The fifth chapter contains the conclusion.

## 2 Methodology

The report is written using five mandatory readings (Cadar & Sen, 2013), (Cadar, Dunbar, & Engler, 2008), (Godefroid, Levin, & Molnar, 2012), (Shoshitaishvili, 2016), (Stephens, 2016) which are all related to concolic testing and fuzzing techniques. Besides the mandatory readings the websites of the discussed programs are visited for further understanding and downloading the programs. The webpages visited include:

1. AFL: <http://lcamtuf.coredump.cx/afl/>
2. KLEE: <https://klee.github.io/>
3. ANGR: <http://angr.io/>

To test the reverse engineering programs VMWare was installed with Ubuntu 15.10 x64 running in a virtual machine. Experiments are conducted on RERS 2016<sup>1</sup> and RERS 2017<sup>2</sup> challenges that serve as a benchmark for comparing verification tools. The RERS code needs to be adapted before it can be used (Verwer, 2017). It is expected that the three programs will find different errors on a particular problem, an explanation for this occurrence will be given based on the underlying techniques each program utilizes. Any additional, non-mandatory, readings will be included in the reference list. Finding additional readings is done using the snowballing principle (Verschuren & Doorewaard, 2010).

## 3 Explanation of fuzzing and concolic execution

This chapter discusses two types of software reverse engineering programs: fuzzing and concolic execution.

### 3.1 Fuzzing

Fuzzing is a technique that executes a program with a wide set of inputs while checking whether this wide set of inputs cause the application to crash (Stephens, 2016). A strength of fuzzers is finding “general” input (i.e. many different values that can trigger meaningful program behavior). Disadvantages of fuzzers are: the inability to generate “specific” input (i.e. specific user input required to pass complex checks in a program), fuzzers suffer from lack of guidance. For instance, a fuzzer randomly guessing the correct input of a 4 Bytes string to trigger a bug is 1 in  $2^{32}$ . Therefore, a common limitation of programs that apply random inputs which trigger vulnerabilities is that mostly shallow bugs are found and they struggle to reach deeper paths in executables.

A fuzzing example in pseudocode is (example adapted from (Böck, 2017)):

```
for $i in {1000..3000}; do fuzzer -r 0.01 -s "$i-filename.*"
```

---

<sup>1</sup> <http://www.rers-challenge.org/2016/problems/Reachability/ReachabilityRERS2016.zip>

<sup>2</sup> <http://rers-challenge.org/2017/problems/training/RERS17TrainingReachability.zip>

In the example above 2000 tries of fuzzing are performed using the `-s` argument `$i-filename.*` where `$i` is an integer and `*` (an asterisk) is random input used as a file extension. The `-r` argument dictates how much percent (i.e.  $0.01 = 1\%$ ) of the `*` extension changes at every iteration. Fuzzers without any awareness of the internal structure are referred to as blackbox fuzzers. It is a simple yet effective technique for finding security vulnerabilities in software (Godefroid, Levin, & Molnar, 2012). In addition, simple fuzzers do not take input validity requirements into account such as a required header used by an image processor. The mentioned limitations lead to low code coverage and thus bugs that are nested deep within the code are often overlooked. In contrast, the performance of fuzzers can be significantly improved if they possess knowledge of the input format by supplying some sample inputs or apply code instrumentation.

## 3.2 Concolic execution

Modern symbolic execution techniques alleviate the problems found in fuzzers with concolic execution. Concolic execution is a portmanteau of ‘concrete’ and ‘symbolic’ execution. In concolic execution a predetermined set of input variables is treated as symbolic variables (i.e. being a function of one or more symbolic representations like `symvar = a + b`) which can change during testing (Stephens, 2016) (smath.info, 2010). The symbolic variables in concolic execution are initially tested with an arbitrary input. During re-execution of the test a set of symbolic constraints and path conditions is established for reaching alternative paths. A path constraint is satisfied if there exists a valid assignment of concrete values to the symbolic values. If the constraint solver fails, then concolic testing handles this situation by replacing some of the symbolic values with concrete values so that the resultant constraints are simplified and can be solved. This manner of testing is much more focused compared to fuzzing as concolic execution can also negate a testing condition found in a path and trace-back to visit a new execution path. Concolic execution is therefore able to more concisely find unvisited compartments (bounded by if-then statements) in programs. However, finding outrageous new inputs, such as fuzzers are able to do, is beyond the working principles of concolic execution. Furthermore, the number of paths grows exponentially with the number of branches in the code which is known as path explosion. Given a fixed time budget, it is critical to explore the most relevant paths first using search heuristics. Figure 1 shows on the left hand symbolic execution using variable `A` and which shows that `A == 1234` will lead to one compartment of the program being tested. Because concolic testing is able to negate tested inputs it can effortlessly back trace and use the `A != 1234` statement to enter and test the other program compartment. In contrast, ‘dumb’ single path concolic execution, only exploring one path, is shown on the right.

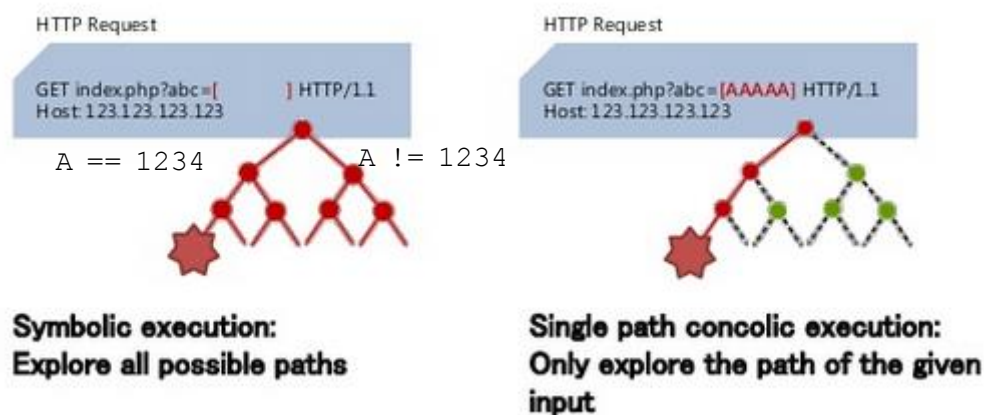


Figure 1 Symbolic execution negation example '`A == 1234`' and '`A != 1234`'; Adapted from (Huang, 2017)

## 4 Description of the software

This chapter describes the workings of the reverse software engineering solutions.

### 4.1 American Fuzzy Lop

Finding vulnerabilities in software can primarily be done in two ways. First, by code inspection (i.e. disassembling binaries to assembler code). Second, by blackbox fuzzing which is feeding program inputs with randomly modified input. Blackbox fuzzing aims at triggering a bug such as a buffer overflow. Blackbox fuzzing has low code coverage (Godefroid, Levin, & Molnar, 2012). American Fuzzy Lop (AFL) is a whitebox fuzzing tool which symbolically executes programs and takes the constraints of inputs into account to create dynamic tests. Moreover, whitebox fuzzing has therefore greater code coverage or even full program path coverage called program verification (Godefroid, Levin, & Molnar, 2012) (Stephens, 2016). The tested program should first be compiled with a utility program to enable control flow tracking. Any behavioral changes as a response to the input can then be detected by the fuzzer. If there is no access to the source code, then blackbox testing is supported as well.

### 4.2 KLEE LLVM Execution Engine

KLEE is a symbolic execution tool which can significantly beat the coverage of developer's own hand-written test suites. KLEE was able to find serious bugs that have been in software for over 15 years (Cadaru, Dunbar, & Engler, 2008). Symbolic execution (or symbolic evaluation) allows for analyzing a program to determine what inputs cause what parts of a program to execute. KLEE is able to generate automatically generated high-coverage test inputs that perform better than the poor performance of manual and random testing approaches (Cadaru, Dunbar, & Engler, 2008). The symbolic input can be anything, including out of the ordinary or outrageous inputs that normal users would never use under normal usage of a program. However, reproducibility of a found bug depends on the code being deterministic.

The goal of KLEE is twofold. First, to hit every line of executable code in a program by running the program in every conceivable way. And second, to check each reachable line of code in a program against an input that satisfies the path constraint for errors. Moreover, KLEE generates test cases for bugs that are found and has good code coverage compared to manually built test suites (Cadaru, Dunbar, & Engler, 2008). An interesting addition to KLEE is testing for failing systems calls such as write fails because a disk is full.

### 4.3 ANGR

Angr is a binary analysis technique and is created in an attempt to mitigate the phenomenon of wasted effort which often occurs when a research prototype is abandoned (Shoshitaishvili, 2016). Furthermore, Angr features building blocks that enable users to utilize many types of analyses, including both static and dynamic techniques. Angr is thus able to combine software reverse engineering techniques, including both fuzzing (e.g. AFL) and concolic execution (e.g. KLEE) techniques. An advantage of Angr is that it is imported in Python which allows for writing complex reverse engineering solutions. A disadvantage is that a user needs to write his own reverse engineering code using Angr.

### 4.4 Discussion
































































Table 1 shows the features, strengths and weaknesses, of different programs. The feature list is distilled from multiple academic sources (Cadaru, Dunbar, & Engler, 2008) (Godefroid, Levin, & Molnar, 2012) (Stephens, 2016). Concolic execution is slower than fuzzing executing each test cycle, because concolic execution needs to interpret application code while a fuzzer natively executes a program (i.e. compiled code). Furthermore symbolic execution suffers from state explosion which means the number of paths grows exponentially when concolic execution explores possible programs paths to take (Stephens, 2016). Other features and properties of the different programs are found in Table 1.

Explanation of 🌀 “extra explanation needed” features in Table 1:

1. AFL and KLEE are written from scratch
2. Many installation steps are needed which are prone to compilation and dependency errors. However, a readily available Docker solution saves installation & dependency installation time.



Table 1 Features:  feature present,  feature absent,  extra explanation needed, “-” means unsure

	Feature	Driller	SAGE	AFL	KLEE	ANGR
1.	Uses random fuzzing (dynamic analysis)		-			-
2.	Uses genetic fuzzing		-			-
3.	Compartment/compartment aware fuzzing			-		-
4.	White-box fuzzing					-
5.	Black-box fuzzing	-	-			-
6.	Uses static concolic/symbolic execution	-			-	
7.	Uses dynamic concolic/symbolic execution					
8.	Inverting conditional control (if-then)					-
9.	Mitigation of path explosion					
10.	Automatic testing					
11.	Code coverage	-		-		-
12.	Speed optimization	-				
13.	Memory usage optimization			-		
14.	Intermediate representation (IR)	-	-	-	LLVM	VEX
15.	Solver	-	Z3	-	STP	Claripy
16.	Multi core support	-				-
17.	Test of compiled programs (e.g. via arguments)	-				-
18.	Tests byte code of programs	-				-
19.	Scalability optimization	-		-		-
20.	Find bugs not found by manual test cases	-				-
21.	Program uses concepts or code of	AFL	-	 <sup>1</sup>	 <sup>1</sup>	-
22.	Ease of installation	-	-	Easy	Difficult <sup>2</sup>	Easy

## 5 Experiments and comparison

Experiments will be done on several problems from the RERS 2016 and 2017 challenges. Problems 4 to 6 of RERS 2017 Reachability and problems 13 to 15 of RERS 2016 reachability training will be tested (Verwer, 2017). This report will explore both the smaller programs of RERS 2017 only (i.e. Problem 4 to 6) and medium sized problems of RERS 2016 (i.e. Problem 13 to 15). Large problems are not included due to computing resources and time constraints. Both AFL and KLEE are used on each problem and the results will be compared between each approach. Only the solutions of the RERS 2017 problems 4, 5 and 6 are made available and can be used to calculate the overall error detection rate.

### 5.1 American Fuzzy Lop

AFL gradually finds solutions over time. Relative to the directory of execution of AFL, two directories need to be created: `testcases` and `findings`. A user puts a testcase per file in the `testcases` directory. AFL benefits greatly from well-formed testcases which is why a testcase is created for every printable character to make a fair comparison with KLEE. Otherwise AFL would have a difficult time discovering these on its own. A condensed explanation of what actions are taken to generate the crashes is shown below:

```
cd $HOME
mkdir testcases findings && echo $'1\n' > testcases/1.txt
afl-gcc Problem4.c -o Problem4
afl-fuzz -i testcases/ -o findings/ Problem4
```

The crash results of the AFL white-box fuzzing test can be found in the directory `findings/crashes/`. The files found in this directory contain the binary inputs used to make Problem4 crash. These inputs can be fed to Problem4 to crash the program or violate assertions. It is possible to find duplicate inputs by using `afl-cmin` while `afl-tmin` minimizes the needed input to crash a program (Zalewski, 2017).

AFL was able to find 20/20 errors in Problem4, 26/32 errors in Problem5 and 20/36 errors in Problem6. Detailed results, including timing results, can be found in Figure 2.

## 5.2 KLEE LLVM Execution Engine

In contrast to normal compilation using gcc or Clang, in order to use KLEE a user needs to compile the source code with llvm-gcc which behaves the same as gcc. However, llvm-gcc produces LLVM bytecode object files instead of binary executables. The parameters and their meaning are shown in Table 2. An example of compiling a program with llvm-gcc and running KLEE is:

```
llvm-gcc --emit-llvm tr.c -o tr.bc
klee --max-time 2 --sym-args 10 10 --sym-files 1 2000 --max-fail 1 tr.bc
```

Table 2 Parameters meaning of KLEE

	Parameter	Function
1.	--max-time n	Run KLEE for n minutes
2.	--sym-args n m	Run up to two (n,m) command line arguments, up to n and m characters long
3.	--sym-files n m	Make standard input and n additional file symbolic, each contains m bytes of data
4.	--max-fail n	Allow system calls to fail at most n times

Only the more useful files of a KLEE run are discussed in detail. ‘Info’ is a text file containing high level information like what command was used to start KLEE and how it performed in terms of explored paths and the number of queries. ‘warnings.txt’ and ‘messages.txt’ contain a list of irregular situations encountered during execution. Additional files are present that provide information about the queries made to the solver before optimization. Most importantly, ‘test<N>’.ktest’ are the generated test cases by KLEE for each path N which can be easily replayed. However, tens of thousands of test cases are being generated while only the ones that cause an assertion violation or error are interesting for bug hunting. Whenever these situations occur KLEE creates additional files test<N>.<error-type>.err with more information about the error. In the current set up, KLEE fails to generate a .err file for each error encountered as discussed above so an ad-hoc solution was made that prints the errors to separate files with a timestamp. Afterwards, these errors are compared with the included solution (if present) for verification. KLEE is able to detect all errors given unlimited time, a scarce resource that in practice is not available in abundance.

KLEE was able to find 20/20 errors in Problem4, 24/32 errors in Problem5 and 29/36 errors in Problem6. Detailed results, including timing results, can be found in Figure 2.

## 5.3 ANGR

ANGR is an open source multi-architecture binary analysis framework that integrates many of the state-of-the-art binary analysis techniques in the literature (Shoshitaishvili, 2016). ANGR offers building blocks for many static and dynamic techniques that can be used in combination to leverage their different strengths. It is also ideal to compare the effectiveness of different approaches, because they are all implemented on a common analysis engine. A pre-requisite for almost all static techniques for vulnerability discovery is the recovery of a Control Flow Graph (CFG) where blocks of code are represented as nodes and the edges as control flow transfers between them. Associated challenges like indirect jumps are being addressed by ANGR. Once the CFG has been retrieved, a static analysis method, called value-set analysis, will attempt to make an over-approximation of the program state (values in registers and memory locations) at any given point in the program. Possible targets of indirect jumps or memory write operation are then used to detect overlapping buffers which may indicate a vulnerability like buffer overflow.

The dynamic symbolic execution module contains implementations of symbolic execution (Ramos & Engler, 2015) and symbolic-assisted fuzzing (Stephens, 2016). The latter approach strikes the balance between cheap fuzzing time and expensive symbolic execution time. Not only does ANGR detect vulnerabilities, but it can also generate inputs that exploit vulnerabilities in a way that allows an attacker to take control of the program’s execution.

Indeed, unlike KLEE and AFL which are specific implementations of vulnerability analysis techniques, ANGR provides a unified binary analysis framework on which many existing techniques can be compared and combined. Furthermore, ANGR performs only binary analysis which is not directly possible with KLEE. This also means that ANGR is unable to perform compile-time code instrumentation like AFL and KLEE.

## 5.4 Discussion

The results between using American Fuzzy Lop (AFL) and KLEE is striking both in the time needed to find crashes and the total number of crashes found. AFL needs between 85% and 15,000% (i.e. Problem15: KLEE 5 seconds versus AFL 750 seconds) more time than KLEE until it stops finding new solutions within 20 minutes. In contrast, AFL finds only between 69% up to 88% of the solutions compared to KLEE. However, Problem5 from RERS2017 is an exception where AFL discovers two additional errors that require a long input string to trigger. A possible explanation is that the effect of path explosion degrades KLEE's performance. Additionally, the focus of problem 5 is on arithmetic including non-linear arithmetic like the modulo operation which is hard to solve during symbolic execution. KLEE in general outperforms AFL and KLEE spends most of its time finding the outliers. The comparison is shown in Figure 2.

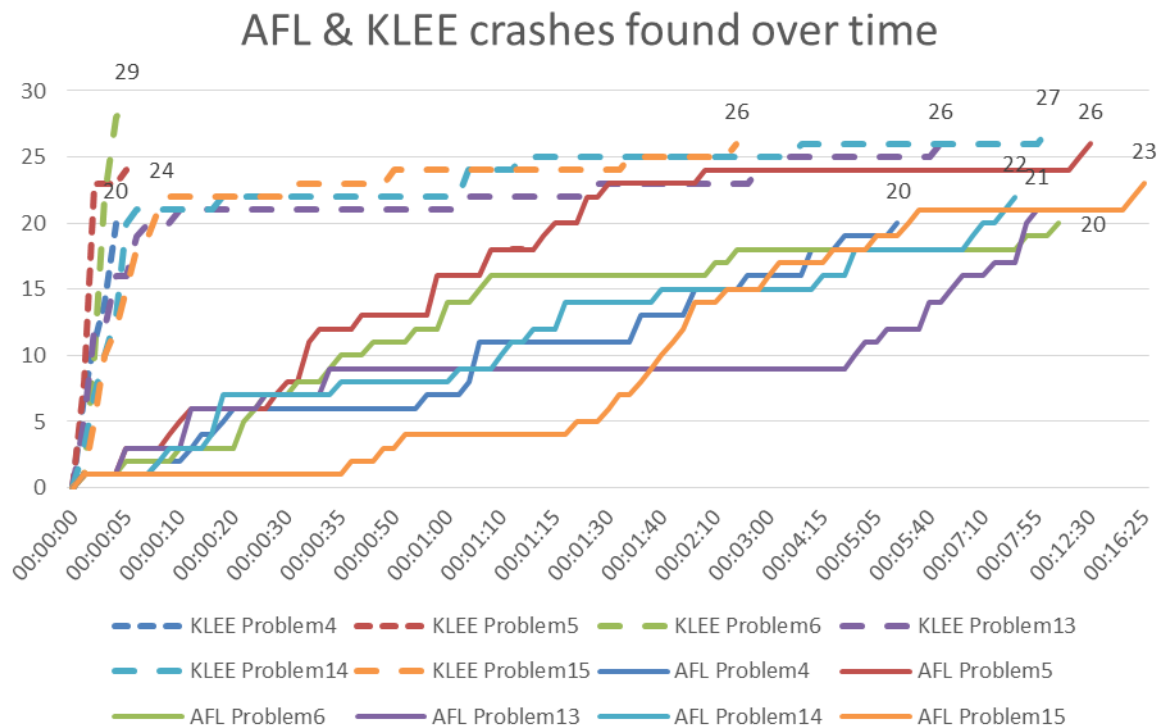


Figure 2 AFL & KLEE unique crashes found over time

## 6 Conclusion

The purpose of this report is to discuss and compare fuzzing and concolic execution. In particular AFL, KLEE and ANGR are of interest. Fuzzing is of great additional value to companies, besides writing manual test cases. Companies, such as Microsoft, employ fuzzing as a mandatory practice to test interfaces. However, fuzzing (based on AFL results) is slow in finding errors. KLEE is both superior in finding more errors and completes finding errors in a blazing fast fashion compared to AFL (i.e. up to 15,000% faster). Especially when programs get larger is where KLEE outperforms AFL. Even though fuzzing has a faster execution time per test cycle due to native execution, concolic execution is much faster in finding new execution paths.

In conclusion, this report is not about what solution is superior, but what tool suits what test cases best. If combining multiple reverse software engineering techniques and having freedom in writing tests is important than Angr provides this best. However, Angr ask for reverse engineering savvy programmers opposed to the easier to use AFL and KLEE programs. A staged usage of AFL, KLEE and Angr could form the ideal solution when considering time investment and results. AFL could scout for preliminary errors while KLEE delves further in specific cases or vice versa. When suspicion arises about certain (borderline) test cases the heavier tool Angr could be deployed.



# References

- Böck, H. (2017, March 5). *Tutorial - Beginner's Guide to Fuzzing*. Retrieved from fuzzing-project.org: <https://fuzzing-project.org/tutorial1.html>
- Cadar, C., & Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM* 56.2, 82-90.
- Cadar, C., Dunbar, D., & Engler, D. R. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI. Vol. 8.*, 209-224.
- Gentle, J. (2014, December 12). *Bug hunting with American Fuzzy Lop*. Retrieved from osephg.com: <https://josephg.com/blog/bug-hunting-with-american-fuzzy-lop/>
- Godefroid, P., Levin, M. Y., & Molnar, D. (2012). SAGE: whitebox fuzzing for security testing. *Queue* 10.1, 20.
- Huang, S.-K. (2017, february 11). *CRAXweb: Automatic web application testing and attack generation* . Retrieved from slideshare.net: <https://www.slideshare.net/shihkunhuang/craxweb-automatic-web-application-testing-and-attack-generation>
- Ramos, D., & Engler, D. (2015). Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX security Symposium*, (pp. 49-64).
- Shoshitaishvili, Y. (2016). SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis. *Security and Privacy (SP), 2016 IEEE Symposium on. IEEE*, 138-157.
- smath.info. (2010, July 01). *symbolic.sm*. Retrieved from smath.info: <http://smath.info/wiki/GetFile.aspx?File=Symbolic%20mathematic%2Fsymbolic.pdf>
- Stephens, N. (2016). Driller: Augmenting fuzzing through selective symbolic execution. *Proceedings of the Network and Distributed System Security Symposium*. (pp. 21-24). San Diego: UC Santa Barbara.
- Verschuren, P., & Doorewaard, H. (2010). *Designing a Research Project*. The Hague: Eleven International Publishing.
- Verwer, S. (2017, March 01). *TU Delft-CS4110-20162017/syllabus*. Retrieved from github.com: [https://github.com/TU Delft-CS4110-20162017/syllabus/blob/master/AFL\\_Fuzzing.md](https://github.com/TU Delft-CS4110-20162017/syllabus/blob/master/AFL_Fuzzing.md)
- Zalewski, M. (2017, March 10). *American Fuzzy Lop*. Retrieved from [http://lcamtuf.coredump.cx/afl/README.txt](http://lcamtuf.coredump.cx: http://lcamtuf.coredump.cx/afl/README.txt)