Fuzzing vs Concolic Execution

Rob van Emous[1] and Patrick van Looy[2]

I. Introduction

Finding bugs or flaws in existing computer systems and software is fairly difficult. For a number of years, the primary method of finding bugs was to search for them by hand, possibly assisted by a clever debugger. Most of the time, this is a very slow process susceptible to human errors. That is why a large part of Computer Security has moved to using automated methods for finding bugs within a program. These methods vary greatly in the extent to which they can automatically inspect the code and make it fail in a consistent manner. Static analysis tools like CPPCheck, are very fast, but only focus on finding bugs in software by analysing the code itself [1]. It does not actually run the program or generate tests. Code instrumentation tools like Valgrind, go a bit further by actually running the software and analysing its (memory related) behaviour [2]. Unfortunately, the analysis of these tools is only as thorough as the given input allows for. As you can see, a combination of these types of testing tools which can spot deep errors and generate its own high coverage test files would be the best solution. That is exactly the kind of service we can expect from the state of the art tools based on fuzzing, symbolic and concolic execution techniques.

We will first provide a detailed explanation of fuzzing and concolic execution in Section II and III. Then we will discuss two tools based on these techniques in Section IV and experiment with them in Section V. Finally, an analysis of these experiments is conducted in Section VI including a study of the current and future usability of these tools.

*This work was not supported by any organization

[1]R.J. van Emous is with Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, 7500 AE Enschede, The Netherlands r.j.vanemous at student.utwente.nl

[2]P. van Looy is with Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, 7500 AE Enschede, The Netherlands p.vanlooy at student.utwente.nl

II. Fuzzing

Let us first explain fuzzing by using an analogy. Fuzzing is like sending a hungry and blind mouse into a labyrinth which contains non-smelling bits of cheese at the start of every path. The mouse cannot see or smell the cheese, so it has to walk through the maze according to a certain strategy, but it cannot use any senses as to how close it is to finding cheese bits and thus new paths. The mouse will keep (randomly) altering its strategy until all cheese bits have been found. In fuzzing, the maze would be the program, the mouse the fuzzer, its strategy a test file and the cheese every new found branch in the program. You can probably imagine that when the maze contains a lot of complex paths and branches it would be infeasible to find all the cheese.

Fuzz testing or fuzzing is a software testing technique used to discover coding errors and security loopholes in software, operating systems or networks by inputting massive amounts of random data, called fuzz, to the system in an attempt to make it crash. If a vulnerability is found, a tool called a fuzz tester (or fuzzer) indicates potential causes. Fuzz testing was originally developed by Barton Miller at the University of Wisconsin in 1989.

Fuzzers work best for problems that can cause a program to crash, such as buffer overflow, cross-site scripting, denial of service attacks, format bugs and SQL injection. These schemes are often used by malicious hackers intent on wreaking the greatest possible amount of havoc in the least possible time. Fuzz testing is less effective for dealing with security threats that do not cause program crashes, such as spyware, some viruses, worms, Trojans and keyloggers.

Fuzz testing is simple and offers a high benefit-to-cost ratio. Fuzz testing can often reveal defects that are overlooked when software is written and debugged. Nevertheless, fuzz testing usually finds only the most serious faults. Fuzz testing alone cannot provide a

complete picture of the overall security, quality or effectiveness of a program in a particular situation or application. Fuzzers are most effective when used in conjunction with extensive black box testing, beta testing and other proven debugging methods.

III. Concolic Execution

Concolic testing is a relatively new testing technique. The word 'concolic' is actually a portmanteau of 'concrete' and 'symbolic' and the way in which this technique operates is a combination of those two techniques. Concrete execution is what tools like Valgrind use: it tests software using varying concrete input values. Obviously, this runs fairly easily and quickly but is not really able to find difficult-to-reach parts in the software. Symbolic execution, on the other hand, substitutes every input and variable value in the program for a symbolic value or function. Because of this, approach it is able to calculate the exact values any variable can take and also calculate the value constraints to reach a certain part of the program. Unfortunately, this detailed analysis of software quickly becomes infeasible with large programs which branch into a lot of paths. A branch is any part of the program in which a decision is made regarding which lines of code to execute next (which path to take) based on some value. Thus in C++ programs that would be all if-else, while, for, switch and try-catch statements.

Concolic testing tries to take advantage of the strengths of both techniques. First, it runs a program with (random) concrete values. Then, for every branch in the program, it negates the condition statement to reach the other path. To be able to negate a condition, it has to be able to work with symbolic statements and its constraint solvers. Therefore, concolic testing is both rather fast and thorough and therefore a very efficient testing technique.

IV. AFL & KLEE explained

In this section, we will elaborate more on two well-known tools for fuzzing and concolic execution. Specifically, we will show how they work and what type of testing they are good at. Finally, we will briefly describe a tool called Driller which combines fuzzing as well as concolic execution.

A. AFL

One of the most well-known fuzzer is "American Fuzzy Lop", a fairly effective fuzzer developed using compile-time instrumentation and genetic algorithms to generate additional test cases to check against the application [3]. Due to some of the optimisations made over other fuzzers, and its use of optimisations at the instrumentation level, it is one of the fastest binary-only fuzzers available [4]. It has found bugs in over 75 different pieces of software and has multiple CVEs dedicated to bugs found by AFL. In fact, while the software was explicitly designed to not use static analysis or symbolic execution for performance reasons, it was still able to generate input that included unseeded magic values by using the basic path exploration techniques that are included as part of afl-fuzz [5]. The one downside to afl-fuzz is that many of its features and optimisations do require compile-time instrumentation to function and are thus less effective with closed-source binaries. However, afl-fuzz still accomplishes a great deal while staying true to its original design goals of being fast, usable and reliable for the general use [6]. As afl-fuzz is improved, so too are the optimisations and performance hacks that can be used by other Fuzzing systems aimed at fuzzing primarily closed-source binaries.

B. KLEE

KLEE is a symbolic execution tool that uses source code and the LLVM compiler to generate tests that attain a high coverage throughout program execution [7]. The general design of KLEE is using LLVM to create an Intermediate Representation (IR) of the program that can be used to easily build up constraints and symbolic formulas throughout the program execution, which can later be solved to generate new test cases that explore other branches. Unlike other systems that do not require source code, KLEE is able to use the source and generated IR in order to create complete constraints for each instruction without any approximations. This allows almost perfect taint analysis in exchange for the source code requirement allowing further optimisations to be performed and tighter constraints when calculating new test cases. However, even with the generated IR, there are still some classes of code that cannot be easily instrumented by KLEE, including floating point computation and inline assembly code. KLEE performs fairly well as a Symbolic Fuzzer and is able to handle many programs with available source code.

C. Driller

Interestingly, there also exists a tool which combines fuzzing and concolic execution called Driller [8]. It starts by fuzzing the program using AFL until it is stuck, thus when no more branches found for some time. Then it uses the concolic execution engine called ANGR [9] to solve the branch conditions on which it is stuck and continues fuzzing on the newfound paths. We will not go into this new and effective type of testing, but limit ourselves to separate fuzzers and concolic execution engines.

V. Experiments with AFL & KLEE

In this section, both AFL and KLEE will be tested based on several RERS 2017 problems [10]. RERS 2017 refers to the 7th International Challenge on the Rigorous Examination of Reactive Systems. The goal of this challenge is to provide a basis for the comparison of verification techniques and available tools to reveal the strengths and weaknesses of specific approaches. Every problem program contains a specific set of difficult-to-reach error states that the testing tool has to find. Therefore, these problems are perfect for testing the performance of AFL and KLEE.

More specifically, several linear temporal logic (LTL) problems will be used. LTL refers to an infinite sequence of states where, based on a linear time perspective, each point in time has a unique successor. It is a convenient formalism for specifying and verifying properties of reactive systems. There are nine LTL problems in which every triple becomes a bit more difficult. Next to that, the N-th problem of every triple is of the same type: plain, arithmetic or data structures. We will test the three medium difficulty problems: 4, 5 and 6. After compiling the programs to work with AFL and KLEE, AFL also required some sample input to get it started. This is done by looking at the alphabet of all valid inputs for these programs and supplying AFL with a file containing all unique characters.

After the setup, we ran both AFL and KLEE (single threaded) for three hours on all three problems. We choose this amount of time because in [11], which shows the performance of AFL on the RERS 2016 problems, about one to four hours is enough to find most medium difficulty error states. Afterwards, we used the input files in the crashes-folder of AFL and the **.err* files in the output folder of KLEE to discover which errors were found by both tools. This can be seen in table I. Next to that, as an alternative measure of testing completeness, the line coverage of the tests generated by both tools is calculated. KLEE automatically generates these statistics, and for AFl we used afl-cov [12] to calculate the coverage. This is displayed in table II.

TABLE I
Number of errors found

	AFL	KLEE
Problem 4	7/21	1/21
Problem 5	20/31	1/31
Problem 6	20/36	1/36
Total	47/88	3/88

TABLE II
Line coverage

	AFL	KLEE
Problem 4	52.20%	91.78%
Problem 5	20.60%	5.59%
Problem 6	15.50%	41.45%
Average	29.43%	46.27%

VI. Analysing reachability problems

Both AFl and KLEE produce their majority of outputs in their output files. KLEE saves all unique tests which result in either normal program execution or in an error. AFL also saves all unique normal program execution or error tests, but next to that also the input which hangs the program. In the remainder of this section, we will discuss the results of running AFL and KLEE and the way in which they differ.

A. AFL

As section V shows, AFL found between one-third and two-thirds of the errors in all three problems. Strangely, the line coverage Of AFL on all problems is much lower: between about 50% on problem 4 and only about 15% on problem 6.
Just the number of errors found by AFL is not that interesting, that is why we also investigated the type of errors it found. More specifically, we are interested in the influence of required input length of an error state

to the percentage found by AFL. This percentage is averaged over all three problems. Also, When a certain length range is not required for any of the error states, AFL is considered to have found 100% of the errors of this length range. The results are shown in figure 1.

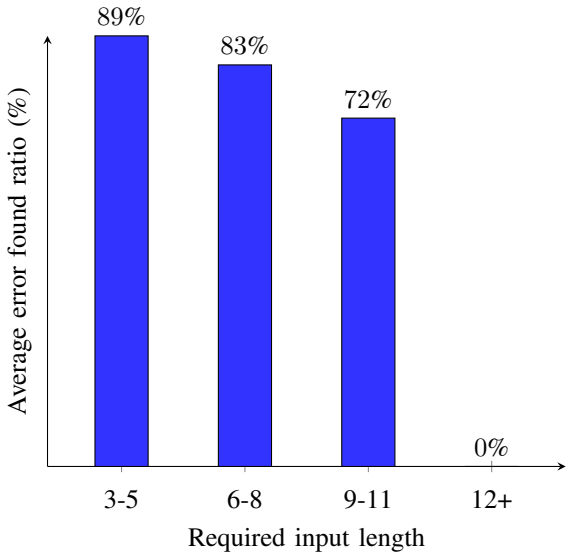

Fig. 1. Error input length to error found ratio

As shown in the figure above, there is a strong relationship between the required input length to reach an error and the chance of AFL finding it. This behaviour is as expected because the chance of finding a (very) long input which causes a crash is very low when using guided fuzzing techniques.

B. KLEE

As section V shows, KLEE only found one error within each of the three problems. Moreover, it did not matter whether KLEE was run for a few minutes or the full three hours, in both cases, it ended up with the same results. This could be seen in the generated output files where in case of an error, a **.err* file was created. Furthermore, we saw that with Problem 4 coverage was the highest at about 90%, where the coverage in Problem 5 was already fairly low at about 40% and even lower with Problem 6 at about 6%.
Unfortunately, the very small amount of errors KLEE found cannot be used to generate the KLEE equivalent of the 'error input length to error found ratio' figure of AFL. That is why in table III, the errors are only listed along with their error input length for each specific problem.

TABLE III
ERRORS FOUND BY KLEE

Problem	Error number	Input length
4	12	7
5	7	3
6	32	4

C. Differences between AFL & KLEE

Thus end up with a mostly higher line coverage when using KLEE compared to AFL, but this does not show in the number of found error states. Although we did run KLEE for a long time, we highly doubt that KLEE would only be able to find 1 error in every one of these problems. It could be caused by the fact that it does not handle LTL problems well, but this has not been documented or researched to our best knowledge. Beforehand, we would have estimated that KLEE would find an equal ratio of all types (input lengths) of errors because symbolic execution in not really as hindered by long inputs as fuzzing would be. However, due to symbolic execution issues like path explosion, it would probably not have found all errors of every type. Just like a breadth-first search would. Next to that, we would have estimated that AFL would be able to quickly find short-input-type errors, but would require a lot more time to find long-input type errors. As the analysis shows, our first estimation cannot be confirmed because we only have very little KLEE results. In contrast, the second estimation indeed proved to be true.

D. (Future) usability

The ever increasing software complexity requires continuous advancements in software testing methodologies. Fuzzing, symbolic and concolic testing are progressively moving beyond academic research and gaining ground in the software industry as viable software testing methodologies. While no single method can guarantee to find all types of software defects, a combination of tools and techniques is the most practicable approach to software testing. As we have seen, both AFL and KLEE found different flaws in the tested code. Therefore, a combination of both techniques, like in Driller, would be favourable since it is likely that more vulnerabilities will be found. Furthermore, optimising speed and methods for reducing path explosion would be of great benefit in both fuzzing and concolic testing.

REFERENCES

[1] D. Marjamäki, "Cppcheck: a tool for static c/c++ code analysis," 2013.

[2] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[3] M. Zalewski, "American fuzzy lop," 2015. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[4] ——, "Technical whitepaper for afl," 2015. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt

[5] ——, "Afl: nobody expects cdata sections in xml." 2015. [Online]. Available: https://lcamtuf.blogspot.nl/2014/11/afl-fuzz-nobody-expects-cdata-sections.html

[6] ——, "Afl: Historical notes," 2015. [Online]. Available: http://lcamtuf.coredump.cx/afl/historical_notes.txt

[7] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[8] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[9] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 138–157.

[10] R. institute, "Download ltl problems," 2017. [Online]. Available: http://www.rers-challenge.org/2017/index.php?page=ltlProblems

[11] R. Smetsers, J. Moerman, M. Janssen, and S. Verwer, "Complementing model learning with mutation-based fuzzing," *arXiv preprint arXiv:1611.02429*, 2016.

[12] M. Rash, "Afl fuzzing code coverage," 2017. [Online]. Available: https://github.com/mrash/afl-cov