Project Report

KleeFL

Seeding fuzzers with symbolic execution

Julian Fietkau
fietkau@campus.tu-berlin.de
Matrikelnummer: 370062
April 24, 2016

Chair for Security in Telecommunications
Technische Universität Berlin

Advisor: Prof. Dr. Jean-Pierre Seifert
Co-Advisor: Bhargava Shastry

Abstract

Many years of vulnerability research have shown that bugs are an even present danger in any software project. They make the application and hence the users and data vulnerable to a variety of threads yielding a wide range of potential damage. Thus, different tools and approaches has been developed to detect these flaws autonomously and establishing a necessary security baseline. In the area of dynamic program analysis the usage of symbolic execution and fuzzing has become one of the most popular approaches for white-box testing. While either of them carry their special assets and drawbacks, both underlie theoretically limitation pretty much as complexity problems.
To extend their capabilities we will present a hybrid white-box fuzzer, called KleeFL that combine symbolic execution along with fuzzing applicable for usual C/C++ applications. Therefore, we initially assign the complex task of code discovery to symbolic execution and doing simple and fast modifications on the discovered seeds with high-performance fuzzers afterwards. This way, we are able to reveal bugs which remain hidden for common fuzzers even on a long run, while avoiding limitations of symbolic execution on the other hand. Considering that, we will finally prove the hypothesis that fuzzing can produce better results in cooperation with symbolic execution by testing the approach against a simple example application as well on a real-world application.

Contents

1 Introduction

Typically each software project has to face the problem of occurring software bugs that causes it to produce an unexpected results or behave in unintended ways. This make an application and hence its users and data vulnerable to a variety of threads affecting their confidentiality, integrity and availability. While some bugs may only have subtle effect on the programs functionality, they can also cause crashes or freezes of the application and furthermore enable an opportunity for malicious users to bypass security mechanisms and obtain unauthorized privileges.
While proving the absence of bugs seems impossible, different tools and approaches have been developed for years to detect these flaws automatically. Thereby usage of symbolic execution and fuzzing has become one of the most favoured approaches in dynamic program analysis, in general software testing and nevertheless for security oriented penetration testing. Booth of them carry their special assets and drawbacks but at last they all underlie theoretically limitation pretty much as complexity problems and hence can only provide partial solutions.
According to Michael Zalewski, the inventor of a popular fuzzer called AFL, fuzzing seems to be the most promising approach in the professional practice, referring to the majority of bugs discovered by fuzzing compared to the small amount of disclosures encountered with symbolic execution he had found on public bug trackers. [1]
However, fuzzing carry big limitations already solved very well by symbolic execution frameworks. An easy example for this is finding the error on this small computation:

$$\texttt{value} = \frac{100}{42 - \texttt{user\_input}}$$

With symbolic execution we will find the inherent division by zero bug in seconds, when some preconditions are satisfied. However how long will it take a fuzzer which primarily do bit flips or insert typical boundary values randomly and does not understand the language?
Because of this question, we decided to try a new approach by combining both worlds: symbolic execution and fuzzing. More precisely we want to try to prove the hypothesis that **fuzzing can produce better results in cooperation with symbolic execution**. Therefore, we will introduce a white-box fuzzer, called KleeFL, that utilize symbolic execution to provide good seeds for a fuzzer. Based on that we will test KleeFL initially on a prepared binary and later even on a real-world example.

2 Related work

2.1 American Fuzzy Lop

American fuzzy lop or AFL is an open-source fuzzer, which is developed by Michael Zalewski. It is based on a genetic algorithm that tries to find interesting test cases to trigger new internal states in a target binary. This is mostly archived by applying bit flips or replace bytes of the input file with various integers and dictionary values that possibly trigger edge cases. Unlike other fuzzers, AFL bring his own compiler that enhance the binary with compile-time instrumentation to optimize the fuzzing performance. Beyond that, it is capable to minmize the overhead by spawning hundreds of processes each second for fuzzing by applying forkservers at special locations in the code. A typical setup of AFL includes the following steps:

1. Building the target binary with AFL compiler
2. Selecting small and valid input files and setup proper arguments for the binary
3. Starting the fuzzer and analyze the findings including crashes and timeouts

AFL has already a remarkable "trophy case" for multiple software projects like Mozilla Firefox, OpenSSH, Adobe Flash, clang/llvm, sqlite and many more. [2]

Additional information as well as the current version of AFL are available at:
`http://lcamtuf.coredump.cx/afl/`

2.2 KLEE

KLEE is a open-source symbolic virtual machine to analyze Low Level Virtual Machine (llvm) bitcode. It is capable of automatically generating inputs and arguments to yield high coverage tests on complex and environmentally-intensive programs. [3]
In general symbolic execution assign symbolic values instead of concrete values, for example: variable x = λ instead of x = 5. Imagine a normal execution of this variable like: multiply x by a factor (e.g. z = x * 2) and evaluate a conditional branch (e.g. z >= 10) afterwards which return true or false depending on the prior computation. During symbolic execution the program will assign z = λ * 2. When reaching the conditional branch the symbolic execution is capable to go each way by forking each solution possibility. Each fork will then run independently with a copy of the current program state and an additional path constraint (e.g. λ * 2 >= 10 or λ * 2 < 10) until it terminates. Finally, the symbolic execution try to concretize the value of λ by solving the collected equations and path constrains.
The main limitations of symbolic execution are the exponential growth of possible solutions on conditional branches as well as code that interacts with the environment which need to be modelled adequately. KLEE uses a variety of methods to handle these weaknesses by:

- Compact state representation to minimize needed resourced per state

- Query optimisation to reduce the load on the solver by simplifying or eliminating queries by value concretisation, expression rewriting and constraint simplification.
    - Thats why KLEE is often termed “concolic execution“ which refers to the hybrid verification approach of using symbolic execution along with concrete execution.
    - For solving, KLEE uses its own constraint solver, called KLEAVER, which is based on the STP (Simple Theorem Prover) constraint solver.
- State scheduling by apply good heuristics to determines which state to run next.
- Environment modelling to partly model the behaviour of various system calls and creates constraints accordingly done by a special klee-ulibc

[4]

In their research the developers of KLEE tried to analyze the GNU COREUTILS utility suite and equivalent tools from BUSYBOX embedded system suite with KLEE. Their findings show significant high line coverage on COREUTILS with an average over 90% per tool (median: over 94%) which beat the coverage of the developers hand-written test suites. While doing the same for 75 equivalent tools in the BUSYBOX the results where even better, including 100% coverage on 31 of them. [3]

2.3 SAGE

Another related work called SAGE (Scalable, Automated, Guided Execution) was publicized by the Microsoft team in 2012. SAGE is a tool employing x86 instruction-level tracing and emulation for whitebox fuzzing of arbitrary file-reading Windows applications and is inspired by recent advances in symbolic execution and dynamic test generation. It records an actual run of a program on a well-formed input, symbolically evaluates the recorded trace, and gathers constraints on inputs capturing how the program uses these. The collected constraints are then negated one by one and solved with a constraint solver, producing new inputs that exercise different control paths in the program. This process is repeated along with a code-coverage maximizing heuristic designed to find defects quickly. They furthermore promote SAGE to scale even on large input files and long execution traces with hundreds of millions of instructions and additionally present some interesting experiments on several Windows applications.
Without any format-specific knowledge, SAGE was able to detect the MS07-017 ANI vulnerability, which was missed by extensive blackbox fuzzing and static analysis tools. Furthermore, even on an early stage of development, SAGE has already discovered 30+ new bugs in large Windows applications including image processors, media players, and file decoders. Several of these are potentially exploitable memory access violations. Unfortunately SAGE is a proprietary tool. [5]

3 Design

In general KleeFL shall run in 6 consecutive phases:

1. Passing the source code of a software project to KleeFL as input
2. Building the project for AFL (instrumented binary) & KLEE (llvm bitcode)
3. Running KLEE on the llvm bitcode of a binary from the project
4. Prepare KLEE findings and pass it to AFL as input seeds
5. Running multiple AFL instances on discovered inputs
6. Analyze AFL findings and build a report including configuration & crashes

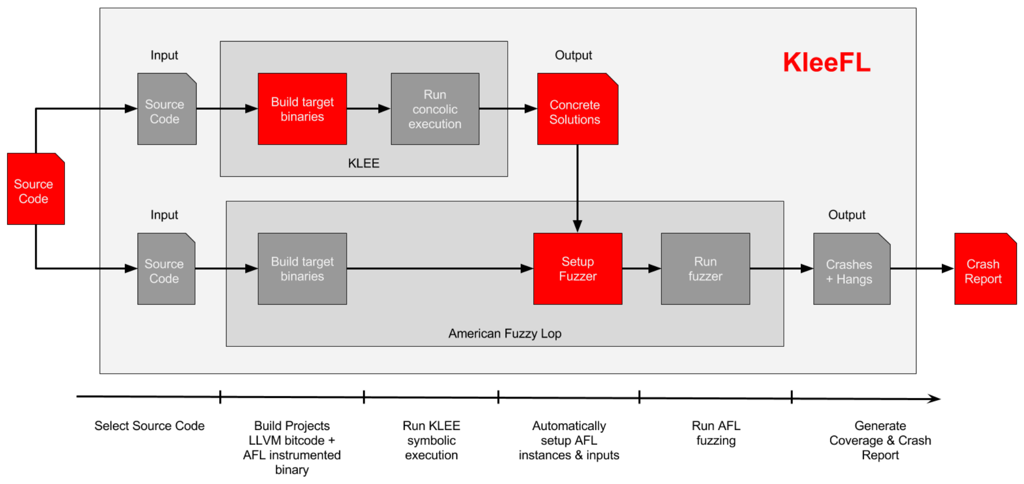


Figure 1: Overview of KleeFL design covering the 6 consecutive phases

The software architecture of KleeFL is a simple pipeline architecture. This way we can use KLEE and AFL as two unmodified basic blocks. Furthermore we are able to build the KleeFL tool as a collection of single Python/Bash scrips which are separated by their purpose.
To reduce the setup time and efforts we also provide a virtual development environment based on Vagrant. Vagrant is a high-level wrapper for virtualization software like VirtualBox, VMware, KVM, etc. and additionally provide a set of tools for configuration and provisioning. This way KleeFL users just need to download our prepared Vagrant box based on Ubuntu 14.06 including builds of all necessary tools like KLEE, AFL, LLVM 3.4, STP and many more. After this setup a user can login into the box and work on a stable and homogenous environment including our experiments.

The Vagrant box is available at: `https://owncloud.sec.t-labs.tu-berlin.de/owncloud/public.php?service=files&t=2c28a437ebb75a33ee127dd4ea827726`

For building the Project we are using the following tools and versions:

- American Fuzzy Lop, Version 2.05b,
  Available at: `http://lcamtuf.coredump.cx/afl/`
- KLEE, Version 1.1.0.,
  Available at: `https://github.com/klee/klee.git`
- Klee-uclibc, Version from Dec 12, 2013,
  Available at: `https://github.com/klee/klee-uclibc.git`
- Minisat SAT solver, Version 2.0,
  Available at: `https://github.com/stp/minisat`
- Zcov coverage reporting tool, Version from Jun 15, 2015,
  Available at: `https://github.com/ddunbar/zcov`
- Whole-program-llvm for building LLVM bitcode, Latest version from Nov 4, 2015, Available at: `https://github.com/travitch/whole-program-llvm`
- GDB exploitable plugin, Version from Dec 31, 2015,
  Available at: `https://github.com/jfoote/exploitable`
- Vagrant development environment, Version 1.8.1,
  Available at: `https://www.vagrantup.com/`
- And many other basic tools like gdb, xxd, gcc, clang, etc.

The latest version of KleeFL can be found at:
`https://gitlab.sec.t-labs.tu-berlin.de/jfietkau/Kleefl`

4 Implementation

As already mentioned the implementation of KleeFL is divided in multiple tools regarding to their purpose. In the following section we will explain their functionality and their intention sorted by chronological usage in KleeFL.

kleefl_init {clean}

kleefl_init will setup a directory structure required by KleeFL. In particular it will create the default run scripts to utilize KLEE and AFL in later steps. With an additional argument the project structure can be cleaned.

kleefl_build {make, make afl, make klee, clean}

kleefl_build contains scripts to build the target project source for KLEE and AFL. The llvm bitcode required by KLEE will be build with the wllvm compiler wrapper which simplify efforts by building complete C/C++ projects with clang / llvm compilers. After a successful build the scripts tries to extract llvm bitcode from any executable file discovered in the target project folder. According to the developers of KLEE, one of the core problems of KLEE is compiling and linking of various software projects to llvm bitcode, because tools like ar, libtool, and ld, do not in general understand LLVM bitcode [6]. By using wllvm a lot of these issues seems to be avoided. Nevertheless additional build scripts, tips and discussions about that can be found in their tutorials.
The instrumented binary required for fuzzing with AFL can be build with the compilers provided by the AFL tool suite like afl-gcc, afl-g++, afl-clang or afl-clang++. Until yet we only support default make & cmake projects with **kleefl_build**. All additional configuration has to be added manually based on the build instructions provided by the developers.

kleefl_pick binary

After building is done **kleefl_pick** will search for the dedicated AFL instrumented binary as well as the llvm bitcode version of a given application name. These files will be copied into the corresponding KleeFL project subdirectories, in particular fuzz/app and klee/app.bc.

/klee/run_klee.sh

After preparing KleeFL so far, we are ready for symbolic execution of the target binary. We provide a default start script for KLEE which should cover most basic application calls right out of the box. In the following section we will explain some of the configuration possibilities of KLEE which can easily be added or modified on the run script.

Argument	Description
–libc=uclibc	Allow external calls to a handed library
–posix-runtime	Support symbolic command line arguments
–sym-args min max size	Apply symbolic arguments with given size
–sym-files min max size	Apply symbolic files with size & quantity
	(Default file names in KLEE A, B, C, ...)
–only-output-states-covering-new	KLEE report only new discoveries
–optimize	Remove dead code from llvm-bitcode
–ignore-solver-failures	Continue execution even if solver crashes
–search=heuristic	select a special heuristic

Table 1: Overview of useful KLEE arguments

Available Heuristics covering: Depth-First Search, Random State Search, Random Path Selection or Non Uniform Random Search. Also interleaving heuristics in round-robin fashion are possible. The default heuristics is Random Path Selection interleaved with Non Uniform Random Search. More Information about heuristics can be found here: `https://klee.github.io/docs/options/`

Depending on the binary and KLEEs configuration, the symbolic execution can take a lot of time. And even worse the execution can also be aborted because of some heavy errors. Nevertheless KLEE has mostly discovered some interesting solutions even if we cancel the execution after some minutes. Changing the configuration and try again will often solve the issues. Keep in mind that partial solutions are always possible for concolic execution.
Finally, after running KLEE the symbolic execution findings can be observed in klee/klee-lastest/. This directory includes some statistics and information about the symbolic execution run and furthermore a set of files for each test case generated by KLEE. Especially the .ktest files is important here, because they contain all the concretized inputs generated by KLEE for each solution.

kleefl_prepare_fuzzing

After running KLEE the provided tool **kleefl_prepare_fuzzing** is able to generate valid AFL inputs and create interesting AFL calls which are added to the AFL start script. This tool is one of the core developments of KleeFL and is able to sort and setup crashing and non-crashing inputs and can additionally detect execution timeouts. Besides that, we have to group, filter and sort all discovered application argument calls to generate only meaningful AFL calls. This includes discarding any trivial call that will not use the given file input, because we only would waste resources here by fuzzing.

/fuzz/run_afl.sh

Afterwards the generated **run_afl.sh** can be launched to finally start the AFL instances. The fuzzers will run infinite until stopped. For good results it is appreciated to run them for multiple cycles which can take minutes, days or years depending on the application complexity. The higher we keep the cycles, the lower will be the probability to find new internal states of the program. [7]
A simple AFL instance can be started like this:

```
afl-fuzz -i input -o output app @@
```

The input directory has to contain the input seed, which should be best a minimum set of valid input files with maximum variety. The output directory will contain all the results and some statistics and information about the fuzzer instances. The argument @@ will be replaced by AFL with the fuzzed input file. Different configurations of AFL are available. The most important to know regarding KleeFL are sync mode, crash mode and dict mode, which are explained further in the following table:

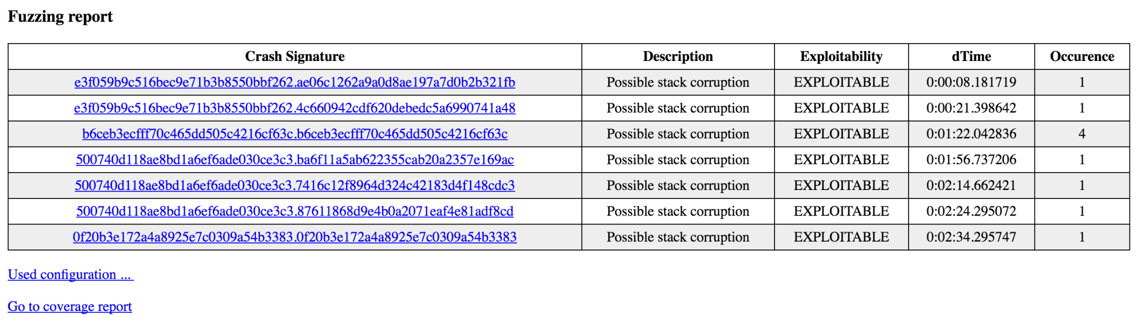
Example command	Description
afl-fuzz -i in -o out -M fuzzer0 ./app -f @@ afl-fuzz -i in -o out -S fuzzer1 ./app -f @@ afl-fuzz -i in -o out -S fuzzer2 ./app -f @@ afl-fuzz -i in -o out -S fuzzer3 ./app -f @@	Using synchronisation mode the master fuzzer (-M) will sync all slaves (-S) result in one common output directory. This way the instances are able to share paths, crash and workload knowledge.
afl-fuzz -C -i crashing -o out ./app -f @@	Running AFL with -C enables the peruvian rabbit mode, which tries to create new unique crashes while using already known crashes.
afl-fuzz -i in -o out -x dict ./app -f @@	While using -x AFL will parse files in the given directory to build an additional dictionary for fuzzing containing special values and other syntax elements.

Table 2: Overview of useful AFL arguments

After fuzzing the discovered crashes has to be analyzed. AFL will consider many crashes as unique, while further analysis will show they often just differ on covered code sections and finally caused by the same errors. Another problem we were facing is the lack of good automated analysis tools for AFL crash findings and coverage applicable to KleeFL. Beyond that, AFL provide some good statistics already, but they seem inconsistent often or can not be easily accumulated while using the synchronisation mode. Thats why we create our own tools.

kleefl_crash_inspector fuzz/sync_dir

kleefl_crash_inspector will iterate all AFL findings in a given synchronisation directory. The discovered crashes and further information about each crash will be collected like: time of first discovery, occurrence, command line arguments, called signal handler and provided input file. Additional information will be added by running gdb on each crashing call supported by gdb plugin gdb-exploitable. This way we are able to provide information like: error classifications (exploitable or not), error descriptions (stack corruption, etc.) and crash signatures which provide us a good way to relate crashing inputs to unique bugs. Finally, the **kleefl_crash_inspector** will generate a html report including a crash overview, detailed subpages for each crash and nevertheless a configuration overview about the unique fuzzer setup.



Fuzzing report

Crash Signature	Description	Exploitability	dTime	Occurence
e3f059b9c516bec9e71b3b8550bbf262.ae06c1262a9a0d8ae197a7d0b2b321fb	Possible stack corruption	EXPLOITABLE	0:00:08.181719	1
e3f059b9c516bec9e71b3b8550bbf262.4c660942cdf620debedc5a6990741a48	Possible stack corruption	EXPLOITABLE	0:00:21.398642	1
b6ceb3ecfff70c465dd505c4216cf63c.b6ceb3ecfff70c465dd505c4216cf63c	Possible stack corruption	EXPLOITABLE	0:01:22.042836	4
500740d118ae8bd1a6ef6ade030ce3c3.ba6f11a5ab622355cab20a2357e169ac	Possible stack corruption	EXPLOITABLE	0:01:56.737206	1
500740d118ae8bd1a6ef6ade030ce3c3.7416c12f8964d324c42183d4f148cdc3	Possible stack corruption	EXPLOITABLE	0:02:14.662421	1
500740d118ae8bd1a6ef6ade030ce3c3.87611868d9e4b0a2071eaf4e81adf8cd	Possible stack corruption	EXPLOITABLE	0:02:24.295072	1
0f20b3e172a4a8925e7c0309a54b3383.0f20b3e172a4a8925e7c0309a54b3383	Possible stack corruption	EXPLOITABLE	0:02:34.295747	1

Used configuration ...

Go to coverage report

Figure 2: Web report covering crashes, used configuration and coverage

kleefl_cov_inspector {make, binary fuzz/sync_dir}

Latest for experimenting the **kleefl_cov_inspector** will be a necessary utility to measure coverage information for the fuzzed binary. Therefore, we also implemented gcov support which requires building the target project with special flags enabled, one more time, like:

```
CFLAGS='-g -fprofile-arcs -ftest-coverage' LDFLAGS='-fprofile-arcs'
```

Any binary build this way will emit coverage information in special .gcda files on each run, saved in the source directory. These files can be collected and summarized easily with tools like zcov afterwards. Zcov is able to build a html report from the generated coverage.zcov file, which can be added to our html report. Therefore, **kleefl_cov_inspector** provide a tool to build cmake projects with special flags, collect coverage information from a given AFL synchronisation directory and finally build the coverage report about the run. A screenshot of an example coverage report can be seen in Appendix A.

5 Evaluation

To evaluate our implementation we created two experiments to examine that KleeFL is working as aspected and to finally prove the hypothesis: symbolic execution can supplement fuzzing process and yield better results. Therefore, we consider three approaches for each experiment:

1. Blunt performance
   We instrument the binary as usual for AFL and provide some basic input information. All fuzzer calls should be as simple as possible, without any special arguments and information a user has to figure out from the application manual or documentation. The same apply to the inputs, they shall only be valid input files without further knowledge about input file structure, syntax, keywords or any kind of special values or test cases. This run mostly exists to prove that AFL is quite ineffective without good seeds. In theory we expect here low code coverage and hence low error detection.

2. Expert performance
   In expert mode we instrument the binary as usual for AFL and provide good input information. Any fuzzer call should differ compared to the others to yield high coverage by running any function accessible via special arguments. Therefore, we will pick carefully hand-crafted application calls based on the manual / help function. Additionally the file inputs shall include also knowledge about file syntax and special keyword or values. However even an expert can not build the ultimate test case. This mode shall represent the professional practice and could be imagined as such as an penetration tester would set up the fuzzer. From prior experience we expect here good coverage and hence good error detection results.

3. KleeFL performance
   For KleeFL we instrument the binary as usual for AFL and also for KLEE. After we did the symbolic execution we only provide symbolic execution outputs to AFL as seeds, exactly like described so far.

5.1 Example Experiment

For experimental purposes we additionally developed a simple example application to test the functionality and nevertheless the performance of KleeFL. This application is just a simple file parser with no further purpose including some heavy bugs, hidden in some conditional branches. This means that only when a user provide some special keywords like ”value”, ”stack” or ”buffr” with decent arguments attached the appropriate code section will be executed. Image the general functionality like described in the pseudocode snippet:

```
buffer[] =[0, 1, ..., 256]
Read file provided by -f [filename] flag
    For each line in file do:
        If line start with 'value':
            value = integer(chars until newline)
            compute calc = 100 / (42 - value)
        If line start with 'stack':
            copy input line to buffer[32]
        If line start with 'buffr':
            value = integer(chars until newline)
            read value from buffer[value]
            calc = 100 / (42 - buffer[value])
        continue;
```

Listing 1: Pseudocode listing of the example application

For this experiment each setup includes two AFL instances, all of them with -f argument. For the blunt run we provided a valid input file without special keywords that trigger interesting code sections. As a result of this we can clearly see very low coverage and zero detected crashes in Table 3.

Metric	Blunt	Expert	KleeFL
Avg. cycles	20	7	29,5
Total executions	2.589.839	2.255.029	2.594.096
Avg. exec. per sec.	2227	2187	2246
Instances · runtime	2 · 10:03 min	2 · 10:10 min	2 · 10:05 min
Seed size	284 bytes	672 bytes	693 bytes
Total AFL crashes	0	12	10
Unique crashes	0	2	4
AFL paths max.	19	41	45
Branches Taken	43,8%	50,0%	59,4%
Branches Executed	75,0%	75,0%	81,2%
Line Coverage	37,8%	54,1%	73,0%

Table 3: Summary of the initial example experiment results

For the expert run we additionally provide special keywords, but not all. Here AFL accomplish higher coverage and find some bugs. But even by providing different keywords like ”value500”, the fuzzer is not be able to find the crash triggered by ”value42”. The code section that includes the segfault by reading the buffer will also not be located by the fuzzer because of the missing input keyword ”buffr” we withhold by purpose. The KleeFL run finally uses the default KLEE setup for round about 10 min, while applying Depth-First Search heuristic for better reproducibility. Afterwards we discovered at least 4 bugs while fuzzing, which include all intended crashes except one. Furthermore the fuzzer accomplished the maximum code coverage, as shown in the detailed coverage report. The only missing code sections just cover instructions for printing error messages and the file not found case. Keep in mind here that most application calls yielding error messages are indeed discovered by KLEE, but will be dismissed to optimize fuzzing performance if they don't access input files on a run. Another feature of KleeFL is that it will hence setup a special AFL instance (peruvian rabbit mode) that tries to create more unique crashes, based on already discovered crashes by KLEE. Because KLEE found the division by zero bug on its preceding run the setup includes one instance of this, which artificially increase the amount of cycles done.

Crash signature	Description	Blunt	Expert	KleeFL
e3f059b9c516bec9 e71b3b8550bbf262	Segfault example.c:45	-	00:01	00:08
500740d118ae8bd1 a6ef6ade030ce3c3	Segfault example.c:55	-	00:00	01:56
b6ceb3ecfff70c46 5dd505c4216cf63c	Div by Zero example.c:57	-	-	01:22
0f20b3e172a4a892 5e7c0309a54b3383	Segfault example.c:72	-	-	02:34

Table 4: Summary of discovered crashes for the example binary

5.2 libjpeg Experiment

As an real-world example we analyzed libjpeg with KleeFL. One of the most relevant findings we discovered are contained in the results of analysing the cjpeg binary. This tool is a JPEG compressor that support multiple image format filetypes. As before we setup three experiments for comparison reasons under equal conditions.

Metric	Blunt	Expert	KleeFL
Avg. cycles	0	1,47	0,20
Total executions	5.748.669	6.268.119	6.876.369
Avg. exec. per sec.	226,30	324,18	319,26
Instances · runtime	17 · 20:05 min	17 · 20:05	17 · 20:20
Seed size	1043	1043	932
Total AFL crashes	30	46	111
Unique crashes	4	4	5
AFL paths max.	438	463	522
Branches Taken	36,6%	63,2%	57,7%
Branches Executed	49,3%	80,6%	73,4%
Line Coverage	48,5	77,2%	70,4%

Table 5: Summary of the real-world experiment with cjpeg binary

For KLEE we used the default configuration from KleeFL and keep it alive for round about 20 minutes. The blunt and the expert run got the same input file (1 JPEG-, 1 BMP-image), which are offered as good examples by AFL. The most important finding here is, that KleeFL can detect more crashes and find each of them faster compared to expert setup of the fuzzer. Even though KleeFL has a slightly lower coverage at all.

Crash signature	Description	Blunt	Expert	KleeFL
6d5053f8e1cee042 a801c7606cd1cf50	Div by Zero jmemmgr.c:406	09:11	03:19	00:01
1ced7550443439c4 cb9a8653598799af	Segfault rdppm.c:152	14:27	12:28	08:54
e7dd39f3987f52be 6e51d02f709e518d	Segfault rdppm.c:170	16:40	09:10	07:30
48fd8bdb8bf6f27d f71c82e38a1f8e54	Segfault rdppm.c:171	-	15:25	10:48
852602b4581e303a d334795239d24cb1	Segfault rdppm.c:172	-	-	11:08

Table 6: Summary of discovered crashes on cjpeg covering the different approaches (Floating Point Exception at jmemmgr.c:406 was rediscovered with a second signature by blunt run. Note that 3 crashes probably yield by the same bug and just affect different lines)

5.3 Comparability

A important point for analysis in general is comparability, what we will discuss in the following section. The best results one could achieve with AFL is by keep it running for multiple cycles or at least for a minimum of one cycle. In terms of AFL a cycles is defined as "count of queue passes" which means the fuzzer iterate all discovered inputs and tries some basic modification. Thereby each cycles done lowers the chance of new discoveries. [8]
For reasons of feasibility we only provide a small amount of runtime for each AFL instance. In case of our first experiment we provided 10 minutes for each setup and accomplish a high amount of cycles, because the application is very simple. In case of our second experiment we provided 20 minutes for each fuzzer. Here only some simple application calls finished several cycles, while most of them get canceled before finishing the first one. Because AFL perform an array of deterministic fuzzing steps initially and we keep them running for a similar amount of time and total executions, we trust in good comparability here. Additionally we have considered to feed a equal amount of seeding bytes for each experiment to ensure a similar workload for the instances. In general each experiment has the same amount of instances and is running in the same virtual environment configuration we prepared for KleeFL.
Like the AFL solution even KLEE runs will mostly be incomplete, because of the computational complexity we already discussed. But fortunately there is no problem in interrupt KLEE while running and get at least a partial solution. To additionally

reduce the burden to KLEE we also set up some computational limitation to skip overly calculations and accept more likely incomplete solutions. Another problem we have to face is that KLEE seems running best with random path heuristics. This is good for daily work because you can circumvent solver crashes by restarting KLEE. Any new run can possibly deliver better results because KLEE will go other ways through the application and discover therefore another set of path constrains. On the other Hand this is bad for reproducibility because even the same setup will produce varying results. Alternatively we can use a non-random heuristic such as Depth-First Search, while this will yield more solvability problems.

6 Conclusion / Discussion

Finally, we will discuss the main benefits and drawbacks of KleeFL. In this project we could verify that KleeFL, even on its early stage of development, will already produce very good seeds for fuzzing. By further analysis of the experimental results we see a comparative high level of code coverage like produced when running an expert setup. This means that argument discovery as well as file input selection done by KLEE are already capable of automating processes which are normally done by humans.
If we keep a closer look at the experiment we can see that AFL is only capable to discover problems relating to its input knowledge. And even if we provide well chosen inputs that reveal more code sections, it will keep AFL busy some time until its can discover a critical input, like shown in the initial experiment. In general, this means we need to supply very good example inputs, which ideally cover all possible cases and hence reveal any conditional branch. This is difficult for humans as well as for computers. However, without chances are high that the fuzzer omit the branch and never be able to discover bugs included.
Another fact we extract is that KleeFL will spot more and probably also faster crashes. In case of the libjpeg experiment we found all crashes earlier in general, but we need to take into account that previous symbolic execution also takes time. By revealing more crashes, even for already known ones, we further extend the information collection for finding the root cause. This is desired by application testers because more information provided yield higher chances to patch the error decently and improve also testing the sufficiency of the patch afterwards.
The biggest drawbacks of KleeFL are inherited by natural KLEE limitations. One of the smaller drawback is anyway that we have to additionally compile the target application to llvm bitcode, which seems solved by wllvm so far. Another disadvantage to face is that KLEE's performance is highly depending on the application complexity. Symbolic execution in general has to struggle with the path explosion problem, which describes the fact that the number of feasible paths in a program grows exponentially. This can be solved in practice by accepting also partial and incomplete solutions, like already featured by KLEE. More computational limitations can be found by trying to solve non-deterministic operations or applications with a certain degree of computational complexity often used in encryption or hash algorithms. Another very practical context is that applications typically interact with their environment. Therefore, KLEE supports the POSIX environment model

by contributing its own ulibc that is already very good, but unfortunately does not cover all, for example pthreads. Additionally KLEE was developed using LLVM 2.9 and has only experimental support for LLVM 3.4. While using this, we discovered that newer features like intrinsic function are not covered by KLEE, too.

7 Future Work

In this last section we want to summarize some major ideas we have for further development of KleeFL. After getting good seeds for fuzzing with KleeFL, we consider an advanced solution where also the fuzzer can seed new KLEE instances by back annotation. This could result into something like a feedback loop, where new discoveries would be announced between both systems. Another idea is to provide a solutions for testing single libraries by integrating other fuzzers like libfuzz and similar.
For the environment model problem we already found a very promising solution that tries to model a whole POSIX OS environment for symbolic execution called Cloud9 [9]. Many further experiments could be done one a more mature base after porting KleeFL to the Cloud9 environment. Finally, even targeting more real-world examples to further prove the practical relevance of seeding fuzzers with symbolic execution is worth more research.

References

[1] M. Zalewski, "Symbolic execution in vuln research." https://lcamtuf.blogspot.de/2015/02/symbolic-execution-in-vuln-research.html. Accessed: 2016-04-24.

[2] M. Zalewski, "American fuzzy lop." http://lcamtuf.coredump.cx/afl/. Accessed: 2016-04-24.

[3] D. E. Cristian Cadar, Daniel Dunbar, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," 2008.

[4] J. Weicker, "Symbolic Execution with KLEE/LLVM," June 2015.

[5] D. M. Patrice Godefroid, Michael Y. Levin, "Automated Whitebox Fuzz Testing," 2012.

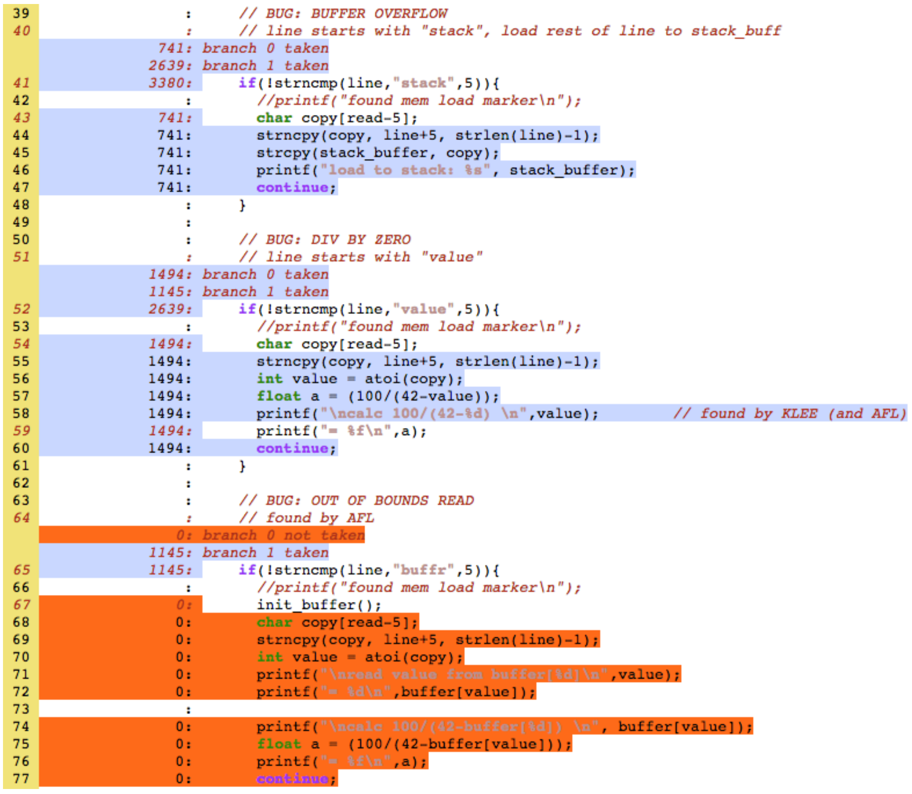
[6] D. E. Cristian Cadar, Daniel Dunbar, "Tutorial on how to use klee to test gnu coreutils." http://klee.github.io/tutorials/testing-coreutils/. Accessed: 2016-04-24.

[7] M. Zalewski, "American fuzzy lop technical whitepaper." http://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2016-04-24.

[8] M. Zalewski, "American fuzzy lop status screen information." http://lcamtuf.coredump.cx/afl/status_screen.txt. Accessed: 2016-04-24.

[9] C. Z. V. K. Stefan Bucur, George Candea, "Cloud9 automated software testing at scale." http://lcamtuf.coredump.cx/afl/status_screen.txt. Accessed: 2016-04-24.

Appendix A



```
                    :      // BUG: BUFFER OVERFLOW
                    :      // line starts with "stack", load rest of line to stack_buff
      741: branch 0 taken
     2639: branch 1 taken
     3380:     if(!strncmp(line,"stack",5)){
                    :        //printf("found mem load marker\n");
                741:       char copy[read-5];
                741:       strncpy(copy, line+5, strlen(line)-1);
                741:       strcpy(stack_buffer, copy);
                741:       printf("load to stack: %s", stack_buffer);
                741:       continue;
                    :      }
                    :
                    :      // BUG: DIV BY ZERO
                    :      // line starts with "value"
     1494: branch 0 taken
     1145: branch 1 taken
     2639:     if(!strncmp(line,"value",5)){
                    :        //printf("found mem load marker\n");
               1494:       char copy[read-5];
               1494:       strncpy(copy, line+5, strlen(line)-1);
               1494:       int value = atoi(copy);
               1494:       float a = (100/(42-value));
               1494:       printf("\ncalc 100/(42-%d) \n",value);          // found by KLEE (and AFL)
               1494:       printf("= %f\n",a);
               1494:       continue;
                    :      }
                    :
                    :      // BUG: OUT OF BOUNDS READ
                    :      // found by AFL
        0: branch 0 not taken
     1145: branch 1 taken
     1145:     if(!strncmp(line,"buffr",5)){
                    :        //printf("found mem load marker\n");
                  0:       init_buffer();
                  0:       char copy[read-5];
                  0:       strncpy(copy, line+5, strlen(line)-1);
                  0:       int value = atoi(copy);
                  0:       printf("\nread value from buffer[%d]\n",value);
                  0:       printf("= %d\n",buffer[value]);
                    :
                  0:       printf("\ncalc 100/(42-buffer[%d]) \n", buffer[value]);
                  0:       float a = (100/(42-buffer[value]));
                  0:       printf("= %f\n",a);
                  0:       continue;
```

Figure 3: Zcov report of example expert run showing covered (blue) and uncovered (red) code sections