KleeFL - Seeding Fuzzers With Symbolic Execution

Julian Fietkau
jfietkau@sec.t-labs.tu-berlin.de
Security in Telecommunications
Technische Universität Berlin

Bhargava Shastry
bshastry@sec.t-labs.tu-berlin.de
Security in Telecommunications
Technische Universität Berlin

Jean-Pierre Seifert
jpseifert@sec.t-labs.tu-berlin.de
Security in Telecommunications
Technische Universität Berlin

Introduction

Any software project has to face the problem of software bugs that causes it to crash, freeze or behave in unintended ways. These bugs make the application and hence its users vulnerable to a variety of threats and furthermore enable an opportunity for malicious users to bypass security mechanisms to obtain unauthorized privileges and data. Thus, different tools have been developed to detect such flaws autonomously. In the area of dynamic program analysis symbolic execution and fuzzing have become one of the most popular approaches for white-box testing. While either of them carry their special assets and drawbacks, all underlie theoretical limitations pretty much as complexity problems, e.g. by finding bugs in codes like this:

```
value = 100 / ( 42 - user_input )
```

To merge the capabilities of both approaches we will present a hybrid whitebox fuzzer, called KleeFL that combines 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 resulting 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 proved this novel idea against a simple example application as well as real-world applications.

Design

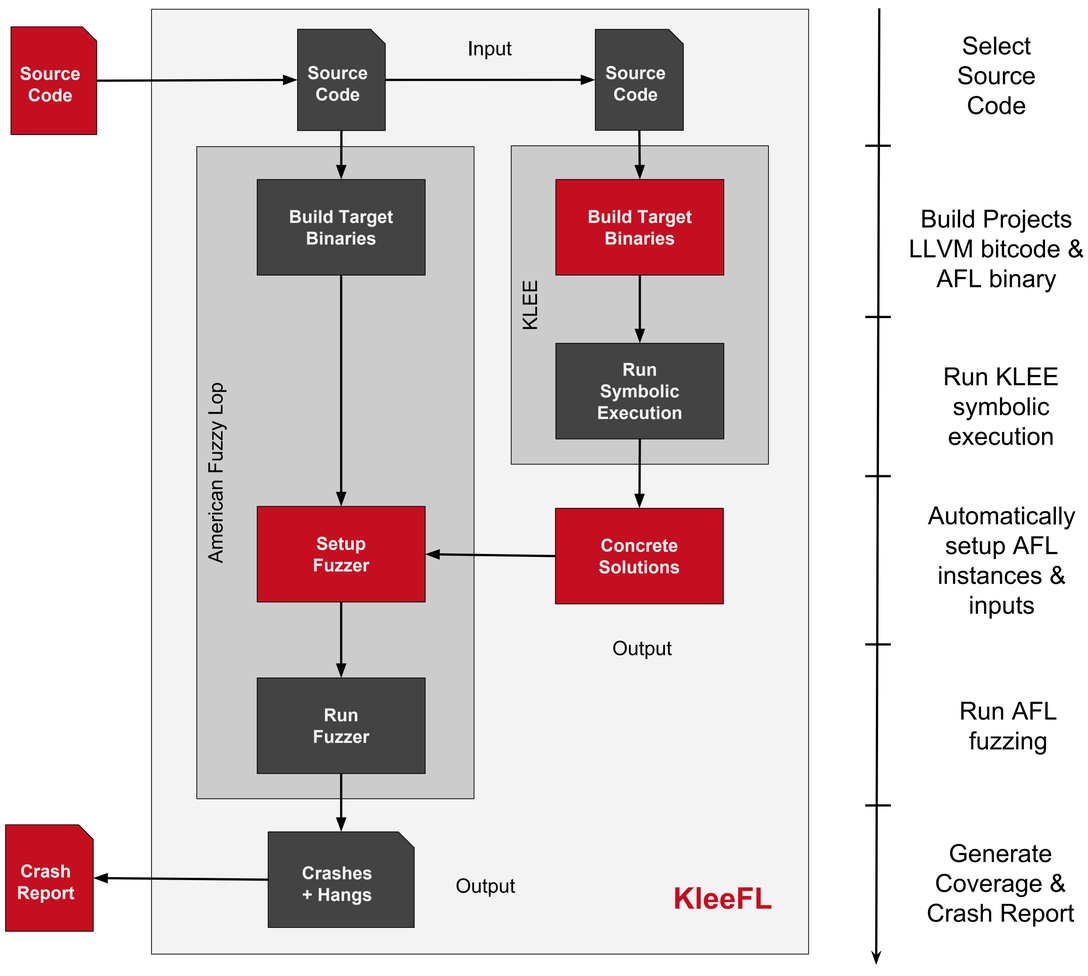

Figure 1: Design overview of KleeFL covering 6 consecutive phases

Evaluation

Our initial evaluation covers a simple example implementations which includes typical conditions and error types of a usual C/C++ application. Another evaluation was done on the cjpeg binary which is part of the libjpeg implementation. The results of this can be seen in the following table:

Metric	Simple Seed	Expert Seed	KleeFL
Avg. AFL cycles	0	1,49	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 min	17 × 20:20 min
Unique AFL 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 1: Summary of a real-world experiment with cjpeg binary from libjpeg

For a sound evaluation we set up a comparative number of fuzzer instances based on the number of meaningful seeds we discovered with KLEE. The symbolic execution engine was kept alive for approximately 20 minutes. Indeed this time has to be considered when comparing discovery time, while a manual setup requires also time to find a diverse set of input examples.

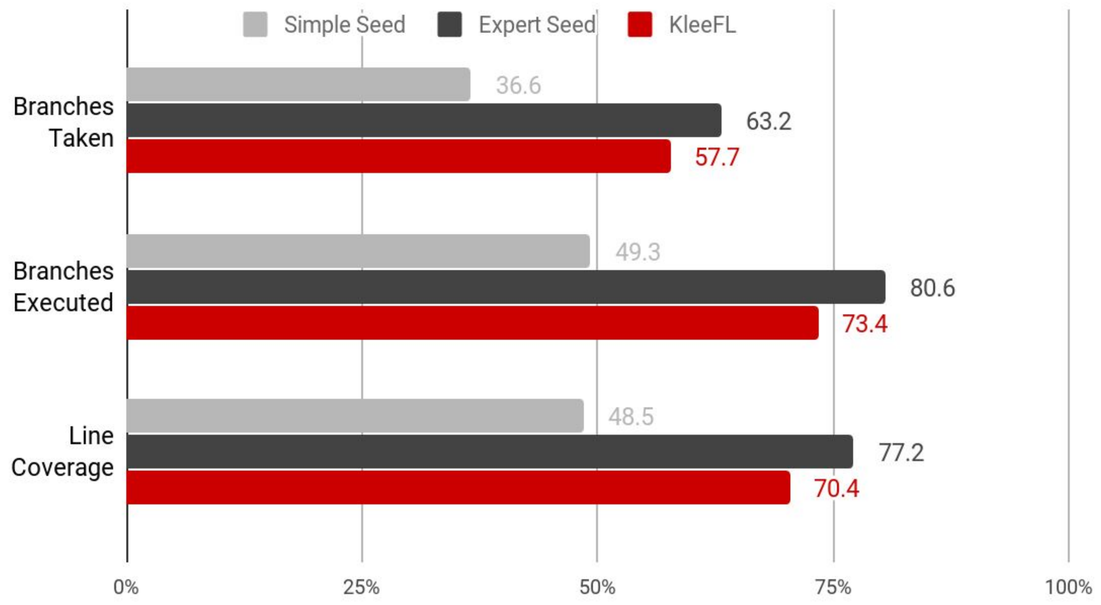

Figure 2: Code coverage results for the three chosen approaches

Crash	Description	Simple Seed	Expert Seed	KleeFL
jmemmgr.c:406	Div by Zero	09:11	03:19	00:01
rdppm.c:152	Segfault	14:27	12:28	08:54
rdppm.c:170	Segfault	16:40	09:10	07:30

Table 2: Discovery time summary of revealed bugs in cjpeg (min.)

References

1. M. Zalewski, “Symbolic execution in vuln research.” https://lcamtuf.blogspot.de/2015/02/symbolic-execution-in-vuln-research.html
2. D. E. Cristian Cadar, Daniel Dunbar, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” 2008.
3. M. Zalewski, “American fuzzy lop technical whitepaper.” http://lcamtuf.coredump.cx/afl/technical_details.txt, Accessed: 2016-04-24.
4. Git repository and additional resources: http://bit.ly/kleefl-git

Conclusion

+ Produce very good seeds for fuzzing in an fully automated manner
+ High code coverage comparable to expert setups
+ Automating a task which is normally done by human experts
+ Allows faster and deeper bug findings compared to simple setups

- Requires additional compilation into LLVM bitcode and runtime for KLEE
- Inherit natural limitations and complexity of symbolic execution