KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

- Cristian Cadar, Daniel Dunbar, Dawson Engler
- 2008

EXE: Automatically Generating Inputs of Death

- Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, Dawson R. Engler
- 2006

- Presented by Oren Kishon
9/3/2014

KLEE (an improvement of EXE) is a tool for automatic test generation using symbolic execution. Klee runs as an interpreter on C code (llvm intermediate code, actually) to find bugs and to create tests for a future run of the program regardless of KLEE. It aims to reach high code-coverage, i.e. to cover as much of the flow-of-control paths of the program (all possibilities of "If A then B else C" flows). KLEE was tested on the Unix *Coreutils* program package and reached over 90% code coverage – more then the testing suite built manually over 15 years.
Symbolic executing is a software interpretation process in which all memory locations (variables, arrays) are assigned with symbolic expressions instead of concrete values. According to the program execution commands, the execution process also holds a mutable set of logical constraints over the symbolic values. The constraint set is mutated whenever the program branches (IF statement) according to the conditional path chosen. KLEE, in fact, forks itself to execute both paths. In termination points (or possibly on branching, for optimization) the set of constraints needs to be "solved" i.e. to produce concrete values for the symbols which hold the constraints so when used as input for the program, it will reach the same condition path. KLEE uses such a constraint solver – STP. STP converts software operations (arithmetic operations, bitwise operations, object referencing etc.) to logical operations on single bits. It forms CNF formulas and solves them using of-the-shelf SAT solver.
KLEE faces the challenges all symbolic executors face. One challenge is path-explosion. Since the number of possible paths grows exponentially with program length, full coverage cannot be reached and there is need for a path choosing algorithm. KLEE uses two kinds of searching algorithms for selecting the next path to execute when on branch. Another challenge is environment interfacing. In order for environment dependent variables could be executed symbolically KLEE developers have wrapped Linux's system calls to create a symbolic environment (for ex: a file pointer could be symbolic and not really change).
KLEE was executed on the 89 programs of *Coreutils*, an hour each. It created tests that cover in average more then 90% of lines of code per a tool, which is more then the coverage of existing test set built by the developers manually. It has found a few bugs that date up to 1992. KLEE was also used for equivalency testing: It compared the outputs of Coreutils tools against their matching tool in the Busybox package. It found many inconsistencies (Busybox is supposed to be exactly like Coreutils, but with a different implementation).

-

As an original Idea I first presented KleeNet (**KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment).** KleeNet is a symbolic executor based on KLEE, with two main additions: a Node executor, which executes a single process symbolically, but with added semantics of packet handling, and a network executor which executes symbolically the network state process – removal and creation of nodes, for instance.

The aim of the KleeNet example was to raise a discussion about the fact that in the sense of environment dependent variables, every system needs an ad-hoc development of a symbolic environment, much like KLEE creators developed for Linux's system calls (they added about 2500 lines of code for wrapping system calls). Coreutils don't use, for example, multithreading. In order to symbolically test multithreading there is need to to develop a symbolic scheduler simulator (another environment dependent variable).
For embedded systems important environment dependent variables are the CPU clock rate and the amount of power supplied. Symbolic executors for such systems will have to simulate different clock rates (which cause different hardware peripherals to act un-normally) and low power supply (to simulate battery dying). These variables can be simulated by software and perhaps be formalized as constraints.

During the discussion after the presentation the following points came up:

- About the discussion item whether line coverage is a good evaluation method: other evaluation methods were suggested such as path-depth histogram. It was concluded that there is no formal method known to be the best method for evaluating a test tool.
- About the discussion item wether KLEE's environment simulating is good enough: It was agreed that for single threaded programs its the best thing to do (it was pointed out the there is a KLEE version for multithreading). In addition, it is much better then many of KLEE's “component” tools who give up the environment challenge and in face of a decision they just concretize an environment dependent variable with random value.
- About the discussion item about the purpose of such comprehensive testing, which doesn't deal with the program correctness at all (doesn't do functional testing): It was agreed that this is not actually a discussion matter since the two types of testing complete each other: Symbolic execution is needed for finding bugs related to memory leaks and overruns, which are security related issues (they can be used by attackers).