Symbolic Execution with KLEE/LLVM

Jan Weicker

June 09, 2015

1 Symbolic Execution

Symbolic execution is a way of testing the validity of programs without the need to find all possible concrete values for the given variables. It uses path-to-path reasoning in its approach which, in most cases, represents a performance advantage over the input-by-input method used in other program analyses. This approach is, however, limited heavily by the exponential growth of the number of different paths in larger programs, making it viable only for relatively short code sections. Handling code that interacts with the environment, such as user inputs or accesses by the OS or the network, is also difficult.

KLEE tries to make use of the advantages of symbolic execution while working around its weaknesses.

2 KLEE

2.1 Features

KLEE operates on LLVM assembly language as created by the LLVM compiler. It interprets instructions and maps them to its own symbolic processes called "states". Each of these states has a register file, stack, heap, program counter and path condition. KLEE's core of operation is an interpreter loop handling the created states, symbolically executing instructions of the given program based on the state's path condition. Once all states have terminated, the loop is closed and the execution complete. The path conditions of finished states are translated into constraints which are then solved by KLEE's constraint solver to create the concrete input values needed to reach

the state's exit point in the actual program. KLEE uses its own constraint solver, KLEAVER, which is based on the STP (Simple Theorem Prover) constraint solver.

2.2 Handling the Weaknesses

KLEE uses a variety of methods aimed at reducing the negative impact symbolic execution's inherent weaknesses have on its performance.

2.2.1 Compact State Representation

Compact state representation is used to reduce the strain put on the system by the large number of states created during testing. While the actual number of generated states is not reduced in this step, the per-state resource requirements are. This is done by sharing memory and portions of the heap structures between the states.

2.2.2 Query Optimization

Query optimization techniques are used to reduce the load on the constraint solver by either simplifying expressions or outright eliminating queries before they are handled by STP. The techniques used by KLEE include:
Expression Rewriting: Rewrites expressions in the constraints into simpler ones.
Constraint Set Simplification: Simplifies constraints from the constraint set when new constraints add new knowledge about the variables in question.
Implied Value Concretization: Once a variable has a concrete value along a certain path, this value is written back to memory and further accesses to this variable return a constant expression.
Constraint Independance: Independent constraints can be split up in separate constraint sets. A query accessing a certain variable may then only need the constraints from one of the sets.
Counter-Example Cache: A cache that stores counter-examples (variable assigments) that make certain queries redundant.

2.2.3 State Scheduling

State scheduling determines which state to run at each instruction of the given code. KLEE uses the following two methods:
Random Path Selection orders the paths taken by previous states in a binary tree. States are randomly selected by starting at the root and following

a random branch at each node. This favors states "high up" in the tree which still have smaller constraint sets and can thus more easily access as-yet unreached code, and also prevents "fork bombing" when a part of the code starts rapidly creating new states.
Coverage-Optimized Search tries to favor and select states that are more likely to uncover as-yet unreached code.

2.2.4 Environment Modeling

When a program contains interaction from outside its code (e.g. system calls), the interpreter needs to still be able to simulate all the possible variables such an interaction may introduce into the flow. KLEE does this with models that imitate the behaviour of various system calls and creates constraints accordingly.

3 Examples