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

Daniel Dunbar, Cristian Cadar, Dawson Engler
Stanford University
Agenda

- Symbolic Execution
- KLEE
- Environment Modeling
- Evaluation/Results
- Critique
Symbolic Execution

- Symbolic Input
- Path Conditions
  - (Constraint Set)
- Tests Case Generation
- Complexity
- Environment Dependencies
Agenda

- Symbolic Execution
- KLEE
- Environment Modeling
- Evaluation/Results
- Critique
Goal

- Hit all Executable Lines of Code (ELOC)
- Error Detection
  - Dangerous Operations
Usage

● Compile test code
  ○ LLVM compiler for GNU C

● Run KLEE on bytecode
  ○ Optionally specify the number, size, and type of Symbolic Input
  ○ Time Limit
  ○ File data and size
  ○ Failures on each program path
Architecture

- High Level
  - Operating System for symbolic processes \((state)\)
  - Interpreter

- Maps compiled instructions to constraints (bit level accuracy)
  - Does not support symbolic floating point, longjmp, threads, and assembly code
Architecture (Basic)

- **Interpreter Loop**
  - Selects one state and symbolically executes one instruction in state context
  - Repeats until no remaining states or user timeout reached

- **Storage Locations for States**
  - Expressions (Trees)
    - **Leaves** represent symbolic variables or constants
    - **Interior nodes** represent operations
  - Constant Expressions = *Concrete*

- **Conditional Branches**
  - Query constraint solver (STP) if provably true or false
    - Take provable branch
  - Otherwise clone state and explore both
Architecture (Basic)

- **Memory Access**
  - Maps every memory object to a distinct Constraint Solver (STP) array

- **Pointers**
  - If a pointer can reference N objects
  - KLEE clones state N times constraining pointer in each state

- **Errors**
  - Generate test using solved constraint and terminate state
State Representation

- **KLEE tracks all memory objects**
  - Implement Copy on Write (COW) at the object level
    - Sharing objects until change

- **Heap represent as immutable map**
  - Portions of map can also be shared amongst multiple states
  - Constant cloning time
  - Previous work (EXE) used one native process per state
Query Optimizations

- Expression Rewriting
- Constraint Set Simplification
- Implied Value Concretization
- Constraint Independence
- Counter-example Cache (Cex. Cache)
Expression Rewriting

- **Simple Arithmetic simplifications**
  - $x + 0 = x$

- **Strength reduction**
  - $x \times 2^n = x \ll n$

- **Linear simplification**
  - $2x - x = x$
Constraint Set Simplification

- **Constraint Set**
  - **Grows** as program runs
  - Constraints on the same variable tend to become more specific

Example:
1. Constraint Added: $x < 10$
2. New Constraint: $x = 5$
3. KLEE actively rewrites previous constraint: $5 < 10$
4. Constraint evaluates to true and is removed
Implied Value Concretization

- Simple algebraic expressions
  - Constraints can be evaluated before being added
    - Concrete

Example:
- $x + 1 = 10$
  - KLEE solves expression
  - Writes `concrete` value ($x = 9$) back to memory
Constraint Independence

- **Constraints**
  - Many constraints do not overlap
  - Divide constraints into independent subsets
    - Reduce constraints sent to query STP
    - KLEE explicitly tracks

Example:
- Given Constraint Set: \(\{i < j, j < 20, k > 0\}\)
- Query whether \(i = 20\)

Only first two constraints are needed
Counter-example Cache (Cex. Cache)

- **Cache**
  - Reduces redundant queries
  - Maps sets of constraints to counter examples (i.e. variable assignments)
  - Special sentinel for no solution
  - Custom data structure based on UBTree structure (Hoffmann and Hoehler)
    - Allows for subsets and supersets of constraint sets

- **Properties**
  - If the subset of a constraint set has no solution, then neither does the original constraint set
  - If the superset of a constraint set has a solution, then the solution also satisfies the original constraint set
  - When a subset of a constraint set has a solution, it is likely that this is also a solution for the original constraint set
Query Optimization Impact

**Figure 2:** The effect of KLEE's solver optimizations over time, showing they become more effective over time, as the caches fill and queries become more complicated. The number of executed instructions is normalized so that data can be aggregated across all applications.

**Table 1:** Performance comparison of KLEE's solver optimizations on COREUTLS. Each tool is run for 5 minutes without optimization, and rerun on the same workload with the given optimizations. The results are averaged across all applications.
State Scheduling

- **Random Path Selection**
  - Maintains binary tree for program path
    - **Leaves** represent current states
    - **Internal Nodes** represent execution forking
  - Branch points have equal probability for each branch
    - Properties
      - Favors higher branches to reach more code
      - Avoids starvation when fork bombs occur (creating new states)
    - Random selection has neither property
State Scheduling

- **Coverage-Optimized Search**
  - Heuristically computes weights for each state
    - Trying to select states likely to cover new code in immediate future
  - Randomly selects state according to weight
  - Heuristics include minimum distance to an uncovered instruction, the call stack of the state, and whether the state recently covered new code

- **Round Robin**
  - KLEE performs both Random Path Selection and Coverage-Optimized Search
  - Protects against stuck states
  - Same state pool (*interleaving*)
  - Limit defined by both max `instructions` and `time`
Agenda

- Symbolic Execution
- KLEE
- Environment Modeling
- Evaluation/Results
- Critique
Environment Modeling

● **Reading**
  ○ Should be able to return all values that could be possible from a read

● **Writing**
  ○ Data written should be reflected in subsequent reads

● **Modeling**
  ○ Uses models written in C to simulate the runtime environment
Modeling the File System

- **Operating on Files**
  - If a file is concrete and exists on the system then the corresponding system call is invoked.
  - Otherwise the operation is emulated through a symbolic file system.
    - Symbolic file system contains a single directory with N symbolic files.
    - N and size of each file specified by user.

- **Opening Files**
  - When a concrete file is passed into open(), the actual file will be pointed to.
  - When a symbolic file is passed into open(), the unconstrained file is matched to N symbolic files and one failure path.

- **Why System Call Level?**
  - Writing the models at C standard library level would be much more work and more likely to contain errors.
Failing System Calls

● **Unexpected Failures**
  ○ Writes could fail due to full disk
  ○ Hard to diagnose bugs for this

● **Catching Unexpected Failures**
  ○ Environmental failures simulated
  ○ System calls failed in a controlled way
  ○ Optional feature
Re-running Test Cases

- Generated test cases rerun through KLEE provided replay driver
- OS objects created through test cases having concrete values
- Program executed using concrete CL arguments from test case
- Ptrace used to allow system call failures return errors
Agenda

- Symbolic Execution
- KLEE
- Environment Modeling
- Evaluation/Results
- Critique
Evaluation

- In-depth Coverage Experiments
  - Coreutils
  - BUSYBOX

- Comparisons
  - Random testers
  - HiStar
Coverage Methodology

- **Coverage Metric**
  - Line coverage measured through gcov
  - Underrepresentation of actual coverage

- **Coverage measurement**
  - Test cases run on stand-alone version of each utility
  - Library code is ignored in results, included in raw size calculation
  - Uncalled functions and dead code are omitted
GNU Coreutils Analysis

- **Run Configuration**
  - GNU Coreutils averages around 3-4K ELOC per utility
  - KLEE run with 60 minute timeout

- **Line Coverage**
  - Mean: 90.9%, Median: 94.7%, Aggregate: 84%
  - 16 tools with 100% coverage

- **Test Cases**
  - Total: 3321 tests, Mean: 37, Median: 33, Max: 129
  - Average path length per test: 76, Median: 53, Max: 512

- **System Calls**
  - Finding system call failure paths increased coverage from 79.9% to 84.5%
Comparison to Dev Tests

● **Line Coverage**
  ○ KLEE beats developer test suite with 84.5% compared to 67.7% coverage
  ○ Average: 90.9% compared to 68.4%, Median: 94.7% compared to 72.5%
  ○ KLEE reached 16 tools with 100% compared to 1 with developer tests

● **Branch Coverage**
  ○ Overall: KLEE reached 76.9%, developers reached 56.5%

● **Methodology Difference**
  ○ KLEE tests for low level errors while developer tests validate output
Bugs Located

- **Old bugs**
  - KLEE discovered 3 bugs that existed since 1992
  - 7 other more recent bugs not found in older versions
  - KLEE test cases added to official developer testing suite

- **Bug Example**
  - `pr`, used to paginate files before printing
  - `chars_per_input_tab` and `chars_per_c` equal to tab width (T)
  - `width` is computed as `(T - input_position mod T)`
  - `input_position` can be negative due to backspacing
  - This can cause a memory overflow due to buffer allocation
Comparison to Random Tests

● Methodology
  ○ 15 random benchmarks using same command line as KLEE, max runtime of 65 minutes
  ○ Tests ran natively for each tool without gcov, then reran with gcov

● Results
  ○ Random tests generated 44% more tests without gcov overhead than with
  ○ Extra tests generated only caused 1% improvement in coverage
  ○ KLEE explored paths to termination only a few times slower than random
  ○ KLEE had less repeated branches improving runtime
Comparison to Random Tests
BUSYBOX Analysis & Results

● Overview
  ○ UNIX utilities made for embedded systems
  ○ Cut down functionality compared to Coreutils
  ○ Tests run in “coreutils” subdirectory with same methodology on 75 utilities

● Analysis Results
  ○ Overall: 90.5%, Mean: 93.5%, Median: 97.5%
  ○ 100% coverage on 31 utilities
  ○ Developer results: 44.8% overall line coverage

● Bugs Found
  ○ 21 BUSYBOX bugs found, 21 bugs found in MINIX, another utility collection
  ○ All bugs were memory errors
Equivalence Checks

- **Deep Correctness**
  - No approximations are made in KLEE analysis
  - False states solved are proven as false, for any condition expressed in C
  - Functional equivalence can be checked through `assert(f(x) == f'(x))`
  - If no path is found that violates this statement, then the functions are equivalent
  - If a path is found that violates this statement, a test case will be created showing so
  - 100% coverage required to guarantee equivalence

- **Example**
  - X and Y are made symbolic, assert used to check for differences
  - Path in `mod_opt` without division by 0 solves the constraint
  - Tautology found, equivalence proven

```c
1: unsigned mod_opt(unsigned x, unsigned y) {
2:     if((y & -y) == y) // power of two?
3:         return x & (y-1);
4:     else
5:         return x % y;
6: }
7: unsigned mod(unsigned x, unsigned y) {
8:     return x % y;
9: }
10: int main() {
11:     unsigned x,y;
12:     make_symbolic(&x, sizeoff(x));
13:     make_symbolic(&y, sizeoff(y));
14:     assert(mod(x,y) == mod_opt(x,y));
15:     return 0;
16: }
```
Equivalence Testing Experiment

● Overview
  ○ Deep-correctness checked by comparing 67 Coreutils tools to “equivalent” BUSYBOX tools

● Methodology
  ○ 2 tools have all global symbols renamed and linked
  ○ Both tools ran through same symbolic environment and have output compared
  ○ If a mismatch is detected, a test case will be generated showing so

● Results
  ○ comm, tee, cksum, split, tr, [, sum, tail, unexpand, split, and ls within a small subset of tools showing mismatches
  ○ comm and tee exhibited mismatches that were serious correctness errors

<table>
<thead>
<tr>
<th>Input</th>
<th>BUSYBOX</th>
<th>COREUTILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm t1.txt t2.txt</td>
<td>[does not show difference]</td>
<td>[shows difference]</td>
</tr>
<tr>
<td>tee -</td>
<td>[does not copy twice to stdout]</td>
<td>[does]</td>
</tr>
<tr>
<td>tee &quot;&quot; &lt;t1.txt</td>
<td>[infinite loop]</td>
<td>[terminates]</td>
</tr>
</tbody>
</table>
HiStar Kernel

- **Methodology**
  - Test driver utilized on user-mode kernel
  - Creates core kernel data structures with a single process having access to a single page of memory

- **Results**
  - When configured with a disk, KLEE can only trigger uncovered code with a large number of kernel objects present
  - KLEE had 17% more coverage than random tests with disk, 28.4% more without
  - A critical security bug was located in the 32-bit version of the kernel
Related Work

● **Differences in Symbolic Execution**
  ○ Other Symbolic Execution Systems are static and have no interaction with the running environment
  ○ Any external interaction requires concrete procedure call args
  ○ Rather than using heuristics, other frameworks test paths compositionally

● **Similarities in Symbolic Execution**
  ○ Other systems also implement optimizations prior to sending queries to the solver
  ○ Like Java PathFinder, tests generated by KLEE can be used to drive code ran through model checkers
  ○ Other systems have implemented deep correctness checking for applications such as network protocol implementations and PHP scripts
Conclusion

○ Goals
  ○ At time of publishing, KLEE’s long term goal is to consistently reach 90% coverage for arbitrary programs

○ Accomplishments
  ○ KLEE has generated significantly higher coverage than developer suites on real code
  ○ KLEE has been used to check 452 applications and found 56 serious bugs
Agenda

- Symbolic Execution
- KLEE
- Environment Modeling
- Evaluation/Results
- Critique
Critique

- Symbolic floating-point support, variable sizes for memory objects
  - Extensions have been created to support these, but they are still not part of the original tool
- Improve heuristic to handle complexity problem for symbolic execution
- Environment modeling requires implementation
Derivative Works

- **Cloud9**
  - Built on top of KLEE, adds parallel symbolic execution
- **KLEE-Float**
  - Extension of KLEE that enables floating-point symbolic execution
- **GKLEE**
  - Extension of KLEE for CUDA C/C++ programs
- **SemFix**
  - Utilizes KLEE, generalized program repair utility
Sources

- KLEE-Float:  https://srg.doc.ic.ac.uk/projects/klee-float/
- GKLEE:  http://formalverification.cs.utah.edu/GKLEE/
Thanks!