KLEE: Effective Testing of Systems Programs

Cristian Cadar

Joint work with Daniel Dunbar and Dawson Engler
Writing Systems Code Is Hard

- Code complexity
  - Tricky control flow
  - Complex dependencies
  - Abusive use of pointer operations

- Environmental dependencies
  - Code has to anticipate all possible interactions
  - Including malicious ones
KLEE
[OSDI 2008, Best Paper Award]

• Based on symbolic execution and constraint solving techniques

• Automatically generates high coverage test suites
  – Over 90% on average on ~160 user-level apps

• Finds deep bugs in complex systems programs
  – Including higher-level correctness ones
int bad_abs(int x) {
    if (x < 0) {
        return -x;
    } else if (x == 1234) {
        return -x;
    } else {
        return x;
    }
}
KLEE Architecture

C code → LLVM → LLVM bytecode

SYMBOIC ENVIRONMENT → KLEE

x \geq 0
x \neq 1234

x = 3

Constraint Solver (STP)

x = -2
x = 1234
x = 3
Outline

• Motivation
• Example and Basic Architecture
• Scalability Challenges
• Experimental Evaluation
Three Big Challenges

• Motivation
• Example and Basic Architecture

• Scalability Challenges
  – Exponential number of paths
  – Expensive constraint solving
  – Interaction with environment

• Experimental Evaluation
Naïve exploration can easily get “stuck”
Use search heuristics:
• **Coverage-optimized search**
  – Select path closest to an uncovered instruction
  – Favor paths that recently hit new code
• **Random path search**
  – See [KLEE – OSDI’08]
Three Big Challenges

• Motivation
• Example and Basic Architecture
• Scalability Challenges
  – Exponential number of paths
  – Expensive constraint solving
  – Interaction with environment
• Experimental Evaluation
Constraint Solving

• Dominates runtime
  – Inherently expensive (NP-complete)
  – Invoked at every branch

• Two simple and effective optimizations
  – Eliminating irrelevant constraints
  – Caching solutions
    • Dramatic speedup on our benchmarks
Eliminating Irrelevant Constraints

• In practice, each branch usually depends on a small number of variables

```plaintext
... x + y > 10
...
if (x < 10) {
    ...
    ...
}
```
Caching Solutions

- Static set of branches: lots of similar constraint sets

\[
\begin{align*}
2 \times y &< 100 \\
x &> 3 \\
x + y &> 10
\end{align*}
\]

\[
\begin{align*}
x = 5 \\
y = 15
\end{align*}
\]

Eliminating constraints cannot invalidate solution

\[
\begin{align*}
2 \times y &< 100 \\
x + y &> 10
\end{align*}
\]

\[
\begin{align*}
x = 5 \\
y = 15
\end{align*}
\]

Adding constraints often does not invalidate solution
Dramatic Speedup

Aggregated data over 73 applications

- Base
- Irrelevant Constraint Elimination
- Caching
- Irrelevant Constraint Elimination + Caching
Three Big Challenges

- Motivation
- Example and Basic Architecture
- **Scalability Challenges**
  - Exponential number of paths
  - Expensive constraint solving
  - Interaction with environment
- Experimental Evaluation
Environment: Calling Out Into OS

```c
int fd = open("t.txt", O_RDONLY);
```

- If all arguments are concrete, forward to OS

```c
int fd = open(sym_str, O_RDONLY);
```

- Otherwise, provide *models* that can handle symbolic files
  - Goal is to explore all possible *legal* interactions with the environment
Environmental Modeling

```c
// actual implementation: ~50 LOC
ssize_t read(int fd, void *buf, size_t count) {
    exe_file_t *f = get_file(fd);
    ...
    memcpy(buf, f->contents + f->off, count)
    f->off += count;
    ...
}
```

- Plain C code run by KLEE
  - Users can extend/replace environment w/o any knowledge of KLEE internals
- Currently: effective support for symbolic command line arguments, files, links, pipes, ttys, environment vars
Does KLEE work?

• Motivation
• Example and Basic Architecture
• Scalability Challenges

• Evaluation
  – Coverage results
  – Bug finding
  – Crosschecking
GNU Coreutils Suite

• Core user-level apps installed on many UNIX systems
• 89 stand-alone (i.e. excluding wrappers) apps (v6.10)
  – File system management: `ls`, `mkdir`, `chmod`, etc.
  – Management of system properties: `hostname`, `printenv`, etc.
  – Text file processing: `sort`, `wc`, `od`, etc.
  – …

Variety of functions, different authors, intensive interaction with environment

Heavily tested, mature code
Coreutils ELOC (incl. called lib)

Executable Lines of Code (ELOC)

Number of applications

<table>
<thead>
<tr>
<th>ELOC Range</th>
<th>Number of Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-3000</td>
<td>5</td>
</tr>
<tr>
<td>3000-4000</td>
<td>53</td>
</tr>
<tr>
<td>4000-5000</td>
<td>16</td>
</tr>
<tr>
<td>5000-6000</td>
<td>6</td>
</tr>
<tr>
<td>6000-7000</td>
<td>4</td>
</tr>
<tr>
<td>7000-8000</td>
<td>1</td>
</tr>
<tr>
<td>8000-9000</td>
<td>3</td>
</tr>
<tr>
<td>9000-10000</td>
<td>2</td>
</tr>
</tbody>
</table>
Methodology

- Fully automatic runs
- Run KLEE one hour per utility, generate test cases
- Run test cases on uninstrumented version of utility
- Measure line coverage using `gcov`
  - Coverage measurements not inflated by potential bugs in our tool
High Line Coverage
(Coreutils, non-lib, 1h/utility = 89 h)

Overall: 84%, Average 91%, Median 95%

Apps sorted by KLEE coverage
Beats 15 Years of Manual Testing

**Avg/utility**

<table>
<thead>
<tr>
<th>KLEE</th>
<th>91%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>68%</td>
</tr>
</tbody>
</table>

Manual tests also check correctness
Busybox Suite for Embedded Devices

Overall: 91%, Average 94%, Median 98%

Coverage (ELOC %)

Apps sorted by KLEE coverage
Busybox – KLEE vs. Manual

KLEE coverage - Manual coverage

Apps sorted by KLEE coverage - Manual coverage

Avg/utility

<table>
<thead>
<tr>
<th>KLEE</th>
<th>94%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>44%</td>
</tr>
</tbody>
</table>
Does KLEE work?

- Motivation
- Example and Basic Architecture
- Scalability Challenges
- Evaluation
  - Coverage results
  - Bug finding
  - Crosschecking
GNU Coreutils Bugs

• Ten crash bugs
  – More crash bugs than approx last three years combined
  – KLEE generates actual command lines exposing crashes
Ten command lines of death

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>md5sum -c t1.txt</code></td>
<td>Calculate MD5 of t1.txt</td>
<td><code>pr -e t2.txt</code></td>
<td>Print t2.txt in escape code format</td>
</tr>
<tr>
<td><code>mkdir -Z a b</code></td>
<td>Create directories a and b</td>
<td><code>tac -r t3.txt t3.txt</code></td>
<td>Reverse line order of t3.txt</td>
</tr>
<tr>
<td><code>mkfifo -Z a b</code></td>
<td>Create named pipes a and b</td>
<td><code>paste -d\nabcdefghijklmnopqrstuvwxyz</code></td>
<td>Paste text in pipes a and b</td>
</tr>
<tr>
<td><code>mknod -Z a b p</code></td>
<td>Create named devices a and b and p</td>
<td><code>ptx -F\nabcdefghijklmnopqrstuvwxyz</code></td>
<td>Print text in pipes a and b</td>
</tr>
<tr>
<td><code>seq -f %0 1</code></td>
<td>Generate sequence from 0 to 1</td>
<td><code>ptx x t4.txt</code></td>
<td>Append to t4.txt</td>
</tr>
</tbody>
</table>

**t1.txt:**
```
t
```
**t2.txt:**
```
b
```
**t3.txt:**
```

**t4.txt:**
```
A
```
Does KLEE work?

• Motivation
• Example and Basic Architecture
• Scalability Challenges
• Evaluation
  – Coverage results
  – Bug finding
  – Crosschecking
Finding Correctness Bugs

- KLEE can prove asserts on a per path basis
  - Constraints have no approximations
  - An assert is just a branch, and KLEE proves feasibility/infeasibility of each branch it reaches
  - If KLEE determines infeasibility of false side of assert, the assert was proven on the current path
Crosschecking

Assume \( f(x) \) and \( f'(x) \) implement the same interface
1. Make input \( x \) symbolic
2. Run KLEE on \( \text{assert}(f(x) == f'(x)) \)
3. For each explored path:
   a) KLEE terminates w/o error: paths are equivalent
   b) KLEE terminates w/ error: mismatch found

Coreutils vs. Busybox:
1. UNIX utilities should conform to \( IEEE \text{ Std.} 1003.1 \)
2. Crosschecked pairs of Coreutils and Busybox apps
3. Verified paths, found mismatches
## Mismatches Found

<table>
<thead>
<tr>
<th>Input</th>
<th>Busybox</th>
<th>Coreutils</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tee &quot;&quot; &lt;t1.txt</code></td>
<td>[infinite loop]</td>
<td>[terminates]</td>
</tr>
<tr>
<td><code>tee -</code></td>
<td>[copies once to stdout]</td>
<td>[copies twice]</td>
</tr>
<tr>
<td><code>comm t1.txt t2.txt</code></td>
<td>[doesn’t show diff]</td>
<td>[shows diff]</td>
</tr>
<tr>
<td><code>cksum /</code></td>
<td>&quot;4294967295 0 /&quot;</td>
<td>&quot;/: Is a directory&quot;</td>
</tr>
<tr>
<td><code>split /</code></td>
<td>&quot;/: Is a directory&quot;</td>
<td></td>
</tr>
<tr>
<td><code>tr</code></td>
<td>[duplicates input]</td>
<td>&quot;missing operand&quot;</td>
</tr>
<tr>
<td><code>[ 0 &quot;&lt;&quot; 1 ]</code></td>
<td></td>
<td>&quot;binary op. expected&quot;</td>
</tr>
<tr>
<td><code>tail -2l</code></td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
<tr>
<td><code>unexpand -f</code></td>
<td>[accepts]</td>
<td>[rejects]</td>
</tr>
<tr>
<td><code>split -</code></td>
<td>[rejects]</td>
<td>[accepts]</td>
</tr>
</tbody>
</table>

`t1.txt: a  t2.txt: b  (no newlines!)`
Related Work

Very active area of research. E.g.:

- EGT / EXE / KLEE [Stanford]
- DART [Bell Labs]
- CUTE [UIUC]
- SAGE, Pex [MSR Redmond]
- Vigilante [MSR Cambridge]
- BitScope [Berkeley/CMU]
- CatchConv [Berkeley]
- JPF [NASA Ames]

KLEE
- Hundred distinct benchmarks
- Extensive coverage numbers
- Symbolic crosschecking
- Environment support
KLEE
Effective Testing of Systems Programs

• KLEE can effectively:
  – Generate high coverage test suites
    • Over 90% on average on ~160 user-level applications
  – Find deep bugs in complex software
    • Including higher-level correctness bugs, via crosschecking