KLEE: Effective Testing of Systems Programs

Cristian Cadar

Joint work with Daniel Dunbar and Dawson Engler

April 16th, 2009
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
Program Path

• **Program Path**
  – A path in the control flow of the program
    • Can start and end at any point
    • Appropriate for imperative programs

• **Feasible program path**
  – There exists an input that leads to the execution of this path

• **Infeasible program path**
  • No input that leads to the execution
Concrete vs. Symbolic Executions

• Real programs have many infeasible paths
  – Ineffective concrete testing

• Symbolic execution aims to find rare errors
Symbolic Testing Tools

- EFFIGY [King, IBM 76]
- PEX [MSR]
- SAGE [MSR]
- SATURN [Stanford]
- KLEE [Stanford]
- Java pathfinder [NASA]
- Bitscope [Berkeley]
- Cute [UIUC, Berkeley]
- Calysto [UBC]
Symbolic Exploration

- Execute a program on symbolic inputs
- Track set of values symbolically
- Update symbolic states when instructions are executed
- Whenever a branch is encountered check if the path is feasible using a theorem prover call
Symbolic Execution Tree

• The constructed symbolic execution paths
• Nodes
  – Symbolic Program States
• Edges
  – Potential Transitions
• Constructed during symbolic evaluation
• Each edge requires a theorem prover call
Simple Example

1) int x, y;
2) if (x > y) {
3)   x = x + y;
4)   y = x - y;
5)   x = x - y;
6)   if (x > y)
7)     assert false;
8)}
Challenge 1: Limitations of Theorem Provers

```c
foobar(int x, int y) {
    if (x * x * x > 0) {
        if (x > 0 && y == 10) {
            abort();
        }
    }
    else {
        if (x > 0 && y == 20) {
            abort;
        }
    }
}
```
1) FILE *fp;
2) fp = fopen("test.txt", "w");
3) if (fp) {
4)       struct stat buffer;
5)   if (stat ("text.txt", &buffer) != 0) {
6)       abort();
7)   }
8) }

Challenge 3: #Theorem prover calls

1) int i = 0;
2) while i < n {
   i = i + 1;
}
3) if (n == 10^6) {
4)    abort();
5) }

Concolic Testing
Concrete + Symbolic = Concolic

- Combine *concrete testing* (concrete execution) and *symbolic testing* (symbolic execution)
- Trade coverage (miss bugs) for scalability
- Reduce the number of theorem prover calls
- Reduce the complexity of path formulas
- Can cope with external calls
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Solve: \(2*y_0 == x_0\)
Solution: \(x_0 = 2, y_0 = 1\)
Concolic Testing Approach

int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

Concrete state:
- \( x = 2, y = 1, z = 2 \)

Symbolic state:
- \( x = x_0, y = y_0, z = 2*y_0 \)

Path condition:
- \( 2*y_0 == x_0 \)
Concolic Testing Approach

int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y + 10) {
            ERROR;
        }
    }
}
```

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

Path condition

Solve: \((2 \times y_0 = x_0) \land (x_0 > y_0 + 10)\)

Solution: \(x_0 = 30, y_0 = 15\)

\(2 \times y_0 = x_0\)

\(x_0 > y_0 + 10\)

\(x = 2, y = 1, z = 2\)

\(x = x_0, y = y_0, z = 2 \times y_0\)
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
Concolic Testing Approach

int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
The Concolic Testing Algorithm

Classify input variables into symbolic / concrete

Instrument to record symbolic vars and path conditions

Choose an arbitrary input

Execute the program

Symbolically re-execute the program

Negate the unexplored last path condition

Is there an input satisfying constraint
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;
    if (x == 1234) 
        return -x;
    return x;
}

Toy Example

x = *

x < 0  TRUE

x = 1234  FALSE
x ≥ 0

x = -2

x = 1234

x = 1234

x ≠ 1234

x = 3

test1.out

return -x

return -x

test2.out

return x

test3.out
KLEE Architecture

- C code
- LLVM
- LLVM bytecode
- SYMBOLIC ENVIRONMENT
- KLEE
- 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
Exponential Search Space

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
... 
... 
... if (x < 10) {
    ...
} 

x + y > 10 
z & -z = z 
x < 10 ?
```
Caching Solutions

- Static set of branches: lots of similar constraint sets

\[
2 * y < 100 \\
x > 3 \\
x + y > 10
\]

\[
2 * y < 100 \\
x + y > 10
\]

Eliminating constraints cannot invalidate solution

\[
2 * y < 100 \\
x > 3 \\
x + y > 10 \\
x < 10
\]

Adding constraints often does not invalidate solution

\[
x = 5 \\
y = 15
\]

UBTree data structure [Hoffman and Koehler, IJCAI '99]
Dramatic Speedup

Aggregated data over 73 applications

- Blue: Base
- Dotted blue: Irrelevant Constraint Elimination
- Dashed green: Caching
- Red: Irrelevant Constraint Elimination + Caching

Time (s) vs. Executed instructions (normalized)
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

- If all arguments are concrete, forward to OS

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

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

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

Coverage (ELOC %)

Apps sorted by KLEE coverage
Beats 15 Years of Manual Testing

<table>
<thead>
<tr>
<th>KLEE coverage</th>
<th>Manual coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>91%</td>
<td>68%</td>
</tr>
</tbody>
</table>
Busybox Suite for Embedded Devices

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

Coverage (ELOC %)

Apps sorted by KLEE coverage
Busybox – KLEE vs. Manual

Avg/utility

<table>
<thead>
<tr>
<th></th>
<th>KLEE</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage</td>
<td>94%</td>
<td>44%</td>
</tr>
</tbody>
</table>

Apps sorted by KLEE coverage - Manual coverage
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 Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>md5sum -c t1.txt</code></td>
<td>Calculate MD5 hash of a file</td>
</tr>
<tr>
<td><code>mkdir -Z a b</code></td>
<td>Create a directory with special permissions</td>
</tr>
<tr>
<td><code>mkfifo -Z a b</code></td>
<td>Create a named pipe with special permissions</td>
</tr>
<tr>
<td><code>mknod -Z a b p</code></td>
<td>Create a named pipe with special permissions</td>
</tr>
<tr>
<td><code>seq -f %0 1</code></td>
<td>Generate sequence of numbers</td>
</tr>
<tr>
<td><code>pr -e t2.txt</code></td>
<td>Print a file with escape characters</td>
</tr>
<tr>
<td><code>tac -r t3.txt t3.txt</code></td>
<td>Reverse sort a file</td>
</tr>
<tr>
<td><code>paste -d\&quot;abcdefghijklmnopqrstuvwxyz</code></td>
<td>Combine multiple files into one with a delimiter</td>
</tr>
<tr>
<td><code>ptx -F\&quot;abcdefghijklmnopqrstuvwxyz</code></td>
<td>Print a file with special characters</td>
</tr>
<tr>
<td><code>ptx x t4.txt</code></td>
<td>Print a file with special characters</td>
</tr>
</tbody>
</table>

### Files:

- **`t1.txt`**: `\t \tMD5(`
- **`t2.txt`**: `\bb\bb\bb\bb\bb\b\t`
- **`t3.txt`**: `\n`
- **`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 $\texttt{assert}(f(x) == f'(x))$
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 $\textit{IEEE Std.1003.1}$
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>
<tr>
<td>`t1.txt: a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>`t2.txt: b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(no newlines!)</td>
<td></td>
</tr>
</tbody>
</table>
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