KLEE
Symbolic Execution

How would you test this program?

```c
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y) {
        if (y == x+10) {
            abort();
        }
    }
}
```
Symbolic Execution

How would you test this program?

Exhaustively! $2^{64}$ inputs to try…
Symbolic Execution

How would you test this program?

Exhaustively! $2^{64}$ inputs to try…
Symbolic Execution

How would you test this program?

```
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
```

Randomly? $1/2^{64}$ chance that a random input crashes.
Symbolic Execution

How would you test this program?

```c
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y) {
        if (y == x+10) abort();
    }
}
```

Randomly? $1/2^{64}$ chance that a random input crashes.
Symbolic Execution

How would you test this program?

\[ z = 2x \]
\[ z = y \]
\[ y = x+10 \]

...\[ y = 2x \text{ and } 2x = x+10 \]
...\[ y = 20, \ x = 10 \]

Logically!

```c
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
```
Symbolic Execution

How would you test this program?

```c
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
```

Logically!
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
Symbolic Execution

```c
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
```
Symbolic Execution

```c
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y) {
        if (y == x + 10)
            abort();
    }
}
```
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
    no crash
}
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y) {
        if (y == x+10)
            abort();
    }
}

Symbolic Execution

\[
\begin{align*}
\text{z} & = 2\times x; \\
\text{z} & = 2\times x \\
\text{if (z == y)} & \quad \text{z = 2\times x \land z = y} \\
\text{if (y == x+10)} & \quad \text{z = 2\times x \land z \neq y} \\
\text{no crash} & \\
\text{no crash} & \\
\end{align*}
\]
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x + 10)
            abort();
}
Symbolic Execution

```c
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y) {
        if (y == x+10) abort();
    }
}
```
void test_me(int x, int y) {
    int z = 2*x;
    if (z == y)
        if (y == x+10)
            abort();
}
Symbolic Execution

Is the path satisfiable?
\[ z = 2^x \land z = y \land y = x+10 \]
Symbolic Execution

Is the path satisfiable?
\[ z = 2^*x \land z = y \land y = x+10 \]

\(\text{(declare-const } x \text{ Int)}\)
\(\text{(declare-const } y \text{ Int)}\)
\(\text{(declare-const } z \text{ Int)}\)

\(\text{(assert } (= z (* x 2)))\)
\(\text{(assert } (= z y))\)
\(\text{(assert } (= y (+ x 10)))\)

\(\text{(check-sat)}\)
\(\text{(get-model)}\)
Symbolic Execution

Is the path satisfiable?
\[ z = 2 \times x \land z = y \land y = x + 10 \]

\[
\begin{align*}
&(\text{declare-const } x \ \text{Int}) \\
&(\text{declare-const } y \ \text{Int}) \\
&(\text{declare-const } z \ \text{Int}) \\
&(\text{assert } (= z (* x 2))) \\
&(\text{assert } (= z y)) \\
&(\text{assert } (= y (+ x 10))) \\
&(\text{check-sat}) \\
&(\text{get-model})
\end{align*}
\]
Symbolic Execution

That’s great! But what about…

• Vagaries of a real language (C)
• Interaction with libraries
• Input/output (files, command line)

And then, make it fast enough to use.
Symbolic Execution

That's great! But what about…

- Vagaries of a real language (C)
- Interaction with libraries
- Input/output (files, command line)

And then, make it fast enough to use.
Symbolic Execution

That’s great! But what about…

• Vagaries of a real language (C)
• Interaction with libraries
• Input/output (files, command line)

And then, make it fast enough to use.
Symbolic Execution

That’s great! But what about…

• Vagaries of a real language (C)
• Interaction with libraries
• Input/output (files, command line)

And then, make it fast enough to use.

KLEE is symbolic execution that actually works.
KLEE Architecture

• An “operating system” for “symbolic processes”
• Actually: symbolic execution for LLVM bitcode
• “Forks” on every branch, to evaluate both sides

This all sounds like a terrible idea.
KLEE Architecture

To make it fast:

- Concretize instructions wherever possible
- Don’t fork for infeasible paths (ask an SMT solver)
- Don’t keep executing a path once it reaches an error
- Don’t model memory as a single flat array
  - Bad for SMT solvers — instead, model each object as a distinct array
  - Model each possibility in a points-to set as a different state
Compact State Representation

- Lots and lots of states! (up to 100k, 1GB RAM)
- Each state needs to track all memory objects in that state — but most memory objects are rarely changed
- Copy-on-write at object granularity
- Heap is an immutable map for sharing between states
  - And can be cloned in constant time when forking
Query Optimization

• Execution time dominated by constraint solving — so do as little constraint solving as possible

• Constraint Independence
  • Only include constraints from the current state if they affect the query being evaluated

• \{i < j, j < 20, k > 0\} and query i=20
Query Optimization

- Counter-example Cache
  - KLEE makes many redundant queries
  - Naive cache: just map each query to its result
  - Fancier cache: can index subsets and supersets of a query
    - If $A$ is unsatisfiable, then $A \land X$ is unsatisfiable
    - If $A \land X$ is satisfiable, then $A$ is satisfiable
    - If $A$ is satisfiable, its solution might also be a solution to $A \land X$ (and this is cheap to check)
Query Optimization

Number of queries reduced by 95%
Runtime reduced by 10x

<table>
<thead>
<tr>
<th>Optimizations</th>
<th>Queries</th>
<th>Time (s)</th>
<th>STP Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>13717</td>
<td>300</td>
<td>281</td>
</tr>
<tr>
<td>Independence</td>
<td>13717</td>
<td>166</td>
<td>148</td>
</tr>
<tr>
<td>Cex. Cache</td>
<td>8174</td>
<td>177</td>
<td>156</td>
</tr>
<tr>
<td>All</td>
<td>699</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>
State Scheduling

- The core of KLEE is a loop that chooses the next symbolic state to evaluate.
- Random Path Selection
  - State is a binary tree (nodes are forks).
  - Randomly select a path through the tree, and execute the node at the leaf.
- Why? Favors nodes high in the tree (more freedom), and avoids fork bombs from loops.
State Scheduling

• Coverage-Optimized Search
  • There is a heuristic.
  • Guides the search towards uncovered instructions
• These two strategies are applied in round-robin style
Environment Modeling

- Real programs run on real operating systems and use real files and stuff
- Files and other inputs could be symbolic
  - Need to model all system calls for symbolic inputs (read, write, stat, ...)
- Modeling system calls rather than libc makes implementation easier — can just compile some libc using our system call implementations
- Can model failing system calls, and provide replay for failing test cases (via ptrace)
Evaluation

• Ran KLEE over all GNU coreutils coverage versus coreutils test suite
**Evaluation**

- Can use symbolic execution to check equivalence of two implementations
- Compared Coreutils to Busybox

<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>
| cksum /             | "4294967295 0 /
| split /             | 
| tr [ 0 `<' 1 ]      | "/: Is a directory"
| sum -s <t1.txt       | [duplicates input on stdout]             |                                         |
| tail -2l            | "97 1 -"                                |                                         |
| unexpand -f          | [rejects]                                | [accepts]                                |
| split -              | [accepts]                                | [rejects]                                |
| ls --color-blah      | [accepts]                                | [rejects]                                |

$t1.txt$: a  
$t2.txt$: b
Evaluation

- Tested the HiStar kernel, executing a single process that executes up to three system calls

<table>
<thead>
<tr>
<th>Test</th>
<th>Random</th>
<th>KLEE</th>
<th>ELOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Disk</td>
<td>50.1%</td>
<td>67.1%</td>
<td>4617</td>
</tr>
<tr>
<td>No Disk</td>
<td>48.0%</td>
<td>76.4%</td>
<td>2662</td>
</tr>
</tbody>
</table>
Discussion

- Is coverage a good metric for measuring the quality of tests?
- KLEE’s not easy to use — where is the trade-off between wrangling KLEE and just writing tests?
  - SAGE — as an x86 symbolic execution engine — is more usable?
- Handling environment is hard — KLEE shoots for 100% accuracy, SAGE doesn’t. How important is it?
- Is it web scale? Is it Google scale?