Symbolic Execution

Jeff Foster
University of Maryland, College Park
Introduction

• Static analysis is great
  ▪ Lots of interesting ideas and tools
  ▪ Commercial companies sell, use static analysis
  ▪ It all looks good on paper, and in papers

• But can developers use it?
  ▪ Our experience: Not easily
  ▪ Results in papers describe use by static analysis experts
  ▪ Commercial tools have a huge code mass to deal with developer confusion, false positives, warning management, etc
One Issue: Abstraction

• Abstraction lets us scale and model all possible runs
  ▪ But it also introduces conservatism
  ▪ *-sensitivities attempt to deal with this
    - * = flow-, context-, path-, field-, etc
  ▪ But they are never enough

• Static analysis abstraction ≠ developer abstraction
  ▪ Because the developer didn’t have them in mind
Symbolic Execution

- Testing works
  - But, each test only explores one possible execution
    - `assert(f(3) == 5)`
  - We hope test cases generalize, but no guarantees

- Symbolic execution generalizes testing
  - Allows *unknown* symbolic variables in evaluation
    - `y = α; assert(f(y) == 2*y-1);`
  - If execution path depends on unknown, conceptually *fork* symbolic executor
    - `int f(int x) { if (x > 0) then return 2*x - 1; else return 10; }`
Symbolic Execution Example

1. `int a = α, b = β, c = γ;`
2.     // symbolic
3. `int x = 0, y = 0, z = 0;`
4. `if (a) {
5.     x = -2;
6. }`
7. `if (b < 5) {
8.     if (!a && c) { y = 1; }
9.     z = 2;
10. }
11. assert(x+y+z!=3)
Insight

• Each symbolic execution path stands for *many* actually program runs
  - In fact, exactly the set of runs whose concrete values satisfy the path condition

• Thus, we can cover a lot more of the program’s execution space than testing can
Early work on symbolic execution


• James C. King. Symbolic execution and program testing. CACM, 19(7):385–394, 1976. (most cited)


The problem

- Computers were small (not much memory) and slow (not much processing power) then
  - Apple’s iPad 2 is as fast as a Cray-2 from the 1980’s

- Symbolic execution is potentially extremely expensive
  - Lots of possible program paths
  - Need to query solver a lot to decide which paths are feasible, which assertions could be false
  - Program state has many bits
Today

• Computers are much faster, memory is cheap
• There are very powerful SMT/SAT solvers today
  ▪ SMT = Satisfiability Modulo Theories = SAT++
  ▪ Can solve very large instances, very quickly
    - Lets us check assertions, prune infeasible paths
  ▪ We’ve used Z3, STP, and Yices
• Recent success: bug finding
  ▪ Heuristic search through space of possible executions
  ▪ Find really interesting bugs
Symbolic Execution for IMP

\[
\begin{align*}
  a &::= n \mid X \mid a_0+a_1 \mid a_0-a_1 \mid a_0 \times a_1 \\
  b &::= bv \mid a_0=a_1 \mid a_0\leq a_1 \mid \neg b \mid b_0 \land b_1 \mid b_0 \lor b_1 \\
  c &::= \text{skip} \mid X:=a \mid \text{goto } pc \mid \text{if } b \text{ then } pc \mid \text{assert } b \\
  p &::= c; \ldots; c
\end{align*}
\]

- \( n \in \mathbb{N} = \text{integers} \), \( X \in \text{Var} = \text{variables} \), \( bv \in \text{Bool} = \{\text{true, false}\} \)
- This is a typical way of presenting a language
  - Notice grammar is for ASTs
    - Not concerned about issues like ambiguity, associativity, precedence
- Syntax stratified into commands (\( c \)) and expressions (\( a,b \))
  - Expressions have no side effects
- No function calls (and no higher order functions)
Symbolic Executor

• (See .ml file)

• ...note: could also add counterexample generation code

• We built a pure symbolic executor
  ▪ It never actually runs the code
Path explosion

• Usually can’t run symbolic execution to exhaustion
  ■ Exponential in branching structure
    1. int a = α, b = β, c = γ; // symbolic
    2. if (a) ... else ...;
    3. if (b) ... else ...;
    4. if (c) ... else ...;

  - Ex: 3 variables, 8 program paths
  ■ Loops on symbolic variables even worse
    1. int a = α; // symbolic
    2. while (a) do ...;
    3.

  - Potentially $2^{31}$ paths through loop!
Search strategies

• Need to prioritize search
  ■ Try to steer search towards paths more likely to contain assertion failures
  ■ Only run for a certain length of time
    - So if we don’t find a bug/vulnerability within time budget, too bad

• Think of program execution as a dag
  ■ Nodes = program states
  ■ Edge(n1,n2) = can transition from state n1 to state n2

• Then we need some kind of graph exploration strategy
  ■ At each step, pick among all possible paths
Basic search

- Simplest ideas: algorithms 101
  - Depth-first search (DFS)
  - Breadth-first search (BFS)
  - Which of these did we implement?

- Potential drawbacks
  - Neither is guided by any higher-level knowledge
    - Probably a bad sign
  - DFS could easily get stuck in one part of the program
    - E.g., it could keep going around a loop over and over again
  - Of these two, BFS is a better choice
Randomness

• We don’t know a priori which paths to take, so adding some randomness seems like a good idea
  ▪ Idea 1: pick next path to explore uniformly at random (Random Path, RP)
  ▪ Idea 2: randomly restart search if haven’t hit anything interesting in a while
  ▪ Idea 3: when have equal priority paths to explore, choose next one at random
    - All of these are good ideas, and randomness is very effective

• One drawback: reproducibility
  ▪ Probably good to use psuedo-randomness based on seed, and then record which seed is picked
  ▪ (More important for symbolic execution implementers than users)
Coverage-guided heuristics

• Idea: Try to visit statements we haven’t seen before

• Approach
  ■ Score of statement = # times it’s been seen and how often
  ■ Pick next statement to explore that has lowest score

• Why might this work?
  ■ Errors are often in hard-to-reach parts of the program
  ■ This strategy tries to reach everywhere.

• Why might this not work?
  ■ Maybe never be able to get to a statement if proper precondition not set up

• KLEE = RP + coverage-guided
Generational search

- Hybrid of BFS and coverage-guided
- Generation 0: pick one program at random, run to completion
- Generation 1: take paths from gen 0, negate one branch condition on a path to yield a new path prefix, find a solution for that path prefix, and then take the resulting path
  - Note will semi-randomly assign to any variables not constrained by the path prefix
- Generation n: similar, but branching off gen n-1
- Also uses a coverage heuristic to pick priority
Combined search

- Run multiple searches at the same time
- Alternate between them
  - E.g., Fitnext

- Idea: no one-size-fits-all solution
  - Depends on conditions needed to exhibit bug
  - So will be as good as “best” solution, which a constant factor for wasting time with other algorithms
  - Could potentially use different algorithms to reach different parts of the program
SMT solver performance

• SAT solvers are at core of SMT solvers
  - In theory, could reduce all SMT queries to SAT queries
  - In practice, SMT and higher-level optimizations are critical

• Some examples
  - Simple identities \((x + 0 = x, x \times 0 = 0)\)
  - Theory of arrays \((\text{read}(42, \text{write}(42, x, A)) = x)\)
    - 42 = array index, A = array, x = element
  - Caching (memoize solver queries)
  - Remove useless variables
    - E.g., if trying to show path feasible, only the part of the path condition related to variables in guard are important
Libraries and native code

• At some point, symbolic execution will reach the “edges” of the application
  ▪ Library, system, or assembly code calls

• In some cases, could pull in that code also
  ▪ E.g., pull in libc and symbolically execute it
  ▪ But glibc is insanely complicated
    - Symbolic execution can easily get stuck in it
  ▪ ⇒ pull in a simpler version of libc, e.g., newlib
    - libc versions for embedded systems tend to be simpler

• In other cases, need to make models of code
  ▪ E.g., implement ramdisk to model kernel fs code
  ▪ This is a lot of work!
Concolic execution

• Also called *dynamic symbolic execution*

• Instrument the program to do symbolic execution as the program runs
  - I.e., shadow concrete program state with symbolic variables

• Explore one path, from start to completion, at a time
  - Thus, always have a concrete underlying value to rely on
Concretization

• Concolic execution makes it really easy to concretize
  ▪ Replace symbolic variables with concrete values that satisfy the path condition
    - Always have these around in concolic execution

• So, could actually do system calls
  ▪ But we lose symbolic-ness at such calls

• And can handle cases when conditions too complex for SMT solver
  ▪ But can do the same in pure symbolic system
Resurgence of symbolic execution

• Two key systems that triggered revival of this topic:
  - DART — Godefroid and Sen, PLDI 2005
    - Godefroid = model checking, formal systems background
  - EXE — Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006
    - Ganesh and Dill = SMT solver called “STP” (used in implementation)
      - Theory of arrays
    - Cadar and Engler = systems
Recent successes

• SAGE
  ▪ Microsoft internal tool
  ▪ Symbolic execution to find bugs in file parsers
    - E.g., JPEG, DOCX, PPT, etc
  ▪ Cluster of $n$ machines continually running SAGE

• KLEE
  ▪ Open source symbolic executor
  ▪ Runs on top of LLVM
  ▪ Has found lots of problems in open-source software
KLEE: Coverage for Coreutils

Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests ($L_{man}$) from KLEE tests ($L_{klee}$) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 9 applications, often significantly.
**KLEE: Coreutils crashes**

```
paste -d\  abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mknod -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\  abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

**Figure 7:** KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

Cadar, Dunbar, and Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008
Research tools at UMD

• Otter — symbolic executor for C
  ▪ Better library model than KLEE, support for multiprocess symbolic execution
• RubyX — symbolic executor for Ruby
• SymDroid — symbolic executor for Dalvik bytecode
Other symbolic executors

• Cloud9 — parallel symbolic execution, also supports threads
• Pex — symbolic execution for .NET
• jCUTE — symbolic execution for Java
• Java PathFinder — a model checker that also supports symbolic execution
Extra slides
## Results

<table>
<thead>
<tr>
<th></th>
<th>Inter-SDSE</th>
<th>Intra-SDSE</th>
<th>CCBSE(X) where X is</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>RP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>InterSDSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IntraSDSE</td>
<td></td>
</tr>
<tr>
<td>mkdir</td>
<td>34.7</td>
<td>19.7(10)</td>
<td>163.0 42.5</td>
<td>274.2</td>
</tr>
<tr>
<td>mkfifo</td>
<td>13.1</td>
<td>0.4</td>
<td>70.2 17.3</td>
<td>315.6</td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>∞</td>
<td>216.5 60.7</td>
<td>851.6</td>
</tr>
<tr>
<td>paste</td>
<td>12.6</td>
<td>0.5</td>
<td>26.0 0.5(1)</td>
<td>30.6</td>
</tr>
<tr>
<td>ptx</td>
<td>18.4</td>
<td>0.6(4)</td>
<td>24.2 0.7(1)</td>
<td>93.8</td>
</tr>
<tr>
<td>seq</td>
<td>12.1</td>
<td>0.4(1)</td>
<td>30.9 1.4</td>
<td>38.2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1891.0</td>
<td>7360.0</td>
<td>530.9 2424.8</td>
<td>3088.55</td>
</tr>
</tbody>
</table>

**Key:** Median | IQR | Outliers  | ∞ : time out

<table>
<thead>
<tr>
<th></th>
<th>Otter-KLEE</th>
<th>Otter-SAGE</th>
<th>Random Path</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pure</td>
<td>w/CCBSE</td>
<td>Pure</td>
</tr>
<tr>
<td>mkdir</td>
<td>168.9</td>
<td>124.7</td>
<td>365.3</td>
</tr>
<tr>
<td>mkfifo</td>
<td>41.7</td>
<td>38.2</td>
<td>77.6</td>
</tr>
<tr>
<td>mknod</td>
<td>174.8</td>
<td>93.1</td>
<td>108.5</td>
</tr>
<tr>
<td>paste</td>
<td>22.6</td>
<td>28.6</td>
<td>54.9</td>
</tr>
<tr>
<td>ptx</td>
<td>33.2</td>
<td>27.1</td>
<td>∞</td>
</tr>
<tr>
<td>seq</td>
<td>354.8</td>
<td>49.3</td>
<td>288.8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>795.8</td>
<td>360.9</td>
<td>4206.4</td>
</tr>
</tbody>
</table>
## Results

<table>
<thead>
<tr>
<th></th>
<th>Inter-SDSE</th>
<th>Intra-SDSE</th>
<th>CCBSE(X) where X is</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>RP</td>
<td></td>
</tr>
<tr>
<td>mkdir</td>
<td>34.7</td>
<td>19.7(10)</td>
<td>163.0 42.5</td>
<td>150.3</td>
</tr>
<tr>
<td>mkfifo</td>
<td>13.1</td>
<td>0.4</td>
<td>70.2 17.3</td>
<td>15.6(9)</td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>∞</td>
<td>216.5 60.7</td>
<td>4.2(8)</td>
</tr>
<tr>
<td>paste</td>
<td>32.6</td>
<td>0.5</td>
<td>26.0 0.5(1)</td>
<td>31.6 (3)</td>
</tr>
<tr>
<td>ptx</td>
<td>5.4</td>
<td>0.1(1)</td>
<td>24.2 0.7(1)</td>
<td>24.5 0.9(3)</td>
</tr>
<tr>
<td>seq</td>
<td>2.1</td>
<td>0.0</td>
<td>30.9 1.4</td>
<td>369.3</td>
</tr>
<tr>
<td>Total</td>
<td>18.8</td>
<td>0.5</td>
<td>530.9</td>
<td>3088.55</td>
</tr>
</tbody>
</table>

**Key:**

- Median
- SIQR (Outliers)
- ∞: time out

### Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution

### CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy

### w/CCBSE: mix with CCBSE(RP)

### RP: random-path

<table>
<thead>
<tr>
<th></th>
<th>Otter-KLEE</th>
<th>Otter-SAGE</th>
<th>Random Path</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w/CCBSE</td>
<td>Pure</td>
<td>w/CCBSE</td>
</tr>
<tr>
<td>mkdir</td>
<td>124.7</td>
<td>12.1(2)</td>
<td>1667.7</td>
</tr>
<tr>
<td>mkfifo</td>
<td>38.2</td>
<td>4.6</td>
<td>251.9</td>
</tr>
<tr>
<td>mknod</td>
<td>174.8</td>
<td>24.1</td>
<td>236.4</td>
</tr>
<tr>
<td>paste</td>
<td>22.6</td>
<td>0.5(4)</td>
<td>108.5</td>
</tr>
<tr>
<td>ptx</td>
<td>33.2</td>
<td>3.9</td>
<td>∞</td>
</tr>
<tr>
<td>seq</td>
<td>354.8</td>
<td>94.4(1)</td>
<td>∞</td>
</tr>
<tr>
<td>Total</td>
<td>795.8</td>
<td>360.9</td>
<td>4206.4</td>
</tr>
</tbody>
</table>

- Semi-interquartile range (SIQR) (seconds)
- # of outliers
- Fastest two times per row are highlighted
Results

Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy
w/CCBSE: mix with CCBSE(RP)  RP: random-path

<table>
<thead>
<tr>
<th></th>
<th>Inter-SDSE</th>
<th>Intra-SDSE</th>
<th>CCBSE(X) where X is</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>34.7 19.7(10)</td>
<td>∞</td>
<td>163.0 42.5</td>
<td>150.3 93.4 150.7 93.9 274.2 315.6(9)</td>
</tr>
<tr>
<td>mknod</td>
<td>13.1 0.4</td>
<td>∞</td>
<td>70.2 17.3</td>
<td>49.7 21.8 49.3 23.2(1)</td>
</tr>
<tr>
<td>paste</td>
<td>∞</td>
<td>∞</td>
<td>216.5 60.7</td>
<td>∞    851.6 554.2(8)</td>
</tr>
<tr>
<td>ptx</td>
<td>12.6 0.5 56.4 5.4</td>
<td>26.0 0.5(1) 31.0 4.8</td>
<td>32.1 4.0</td>
<td></td>
</tr>
<tr>
<td>seq</td>
<td>∞</td>
<td>∞</td>
<td>30.9 1.4</td>
<td>369.3 425.9(6) 391.8 411.1(6) 38.2 14.5(8)</td>
</tr>
</tbody>
</table>

Huge SIQRs, and many outliers...

<table>
<thead>
<tr>
<th></th>
<th>Otter-SAGE</th>
<th>Random Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>100.0 51.9</td>
<td>124.7 12.1(2)</td>
</tr>
<tr>
<td>mknod</td>
<td>41.7 5.2(1)</td>
<td>38.2 4.6</td>
</tr>
<tr>
<td>paste</td>
<td>22.6 0.5(4)</td>
<td>28.6 0.9(3)</td>
</tr>
<tr>
<td>ptx</td>
<td>33.2 3.9</td>
<td>27.1 2.7</td>
</tr>
<tr>
<td>seq</td>
<td>354.8 94.3(1)</td>
<td>49.3 5.1(1)</td>
</tr>
<tr>
<td>Total</td>
<td>795.8</td>
<td>360.9</td>
</tr>
</tbody>
</table>
**Results**

<table>
<thead>
<tr>
<th></th>
<th>Inter-SDSE</th>
<th>Intra-SDSE</th>
<th>CCBSE(X) where X is</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>34.7</td>
<td>19.7(10)</td>
<td>163.0 42.5</td>
<td></td>
</tr>
<tr>
<td>mkfifo</td>
<td>13.1</td>
<td>0.4</td>
<td>70.2 17.3</td>
<td></td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>∞</td>
<td>216.0 216.0</td>
<td></td>
</tr>
<tr>
<td>paste</td>
<td>12.6</td>
<td>0.5</td>
<td>26.0</td>
<td></td>
</tr>
<tr>
<td>ptx</td>
<td>18.4</td>
<td>0.6(4)</td>
<td>24.2</td>
<td></td>
</tr>
<tr>
<td>seq</td>
<td>12.1</td>
<td>0.4(1)</td>
<td>30.9</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>1891.1</td>
<td>7360.0</td>
<td>530.5</td>
<td></td>
</tr>
</tbody>
</table>

**CCBSE(X)**: call-chain-backward symbolic execution, with X as the forward strategy

w/CCBSE: mix with CCBSE(RP)  
RP: random-path

**Key:** Median | SIQR| Outliers | ∞: time out

---

**Inter(Intra)-SDSE**: inter(intra)-procedural shortest-distance symbolic execution

**Inter(Intra)-SDSE**: inter(intra)-procedural shortest-distance symbolic execution

**Distance heuristic works very well in practice**

**Using inter-procedural distance is crucial for SDSE**
## Results

**Inter(Intra)-SDSE**: inter(intra)-procedural shortest-distance symbolic execution  
**CCBSE(X)**: call-chain-backward symbolic execution, with X as the forward strategy  
**w/CCBSE**: mix with CCBSE(RP)  
**RP**: random-path  

<table>
<thead>
<tr>
<th></th>
<th>Inter-SDSE</th>
<th>Intra-SDSE</th>
<th>CCBSE(X) where X is</th>
<th>KLEE</th>
<th>SDSE timed out</th>
<th>CCBSE(RP) did not time out on mknod, and it has better overall performance than InterSDSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>34.7 19.7(10)</td>
<td>∞</td>
<td>163.0 42.5</td>
<td>150.3 93.4 150.7 93.9</td>
<td>∞</td>
<td>274.2 315.6(9)</td>
</tr>
<tr>
<td>mkfifo</td>
<td>13.1 0.4</td>
<td>∞</td>
<td>70.2 17.3</td>
<td>49.7 21.8 49.3 23.4(1)</td>
<td>851.6 554.2(8)</td>
<td>30.6 9.7(8)</td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>56.4 5.4</td>
<td>26.0 0.5(1)</td>
<td>31.0 4.8 32.1 4.0</td>
<td>93.8 81.7(7)</td>
<td>38.2 14.5(8)</td>
</tr>
<tr>
<td>paste</td>
<td>12.6 0.5</td>
<td>103.5 19.7(1)</td>
<td>24.2 0.7(1)</td>
<td>24.5 0.9(3) 24.1 1.1(2)</td>
<td>38.2 14.5(8)</td>
<td>38.2 14.5(8)</td>
</tr>
<tr>
<td>ptx</td>
<td>18.4 0.5</td>
<td>∞</td>
<td>30.9 1.4</td>
<td>369.3 425.9(6) 391.8 411.1(6)</td>
<td>38.2 14.5(8)</td>
<td>38.2 14.5(8)</td>
</tr>
<tr>
<td>seq</td>
<td>12.1 0.4</td>
<td>∞</td>
<td>30.9 1.4</td>
<td>369.3 425.9(6) 391.8 411.1(6)</td>
<td>38.2 14.5(8)</td>
<td>38.2 14.5(8)</td>
</tr>
<tr>
<td>Total</td>
<td>1891.5 7360.0</td>
<td>530.9 2424.8</td>
<td>2448.0</td>
<td>3088.55</td>
<td>274.2 315.6(9)</td>
<td>851.6 554.2(8)</td>
</tr>
</tbody>
</table>
Results

<table>
<thead>
<tr>
<th></th>
<th>Inter-Intra-SDSE</th>
<th>Intra-SDSE</th>
<th>CCBSE(X) where X is</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>RP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>InterSDSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IntraSDSE</td>
<td></td>
</tr>
<tr>
<td>mkdir</td>
<td>34.7</td>
<td>19.7(10)</td>
<td>∞</td>
<td>163.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mkfifo</td>
<td>13.1</td>
<td>0.4</td>
<td>∞</td>
<td>70.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>∞</td>
<td>216.5</td>
<td>∞</td>
</tr>
<tr>
<td>paste</td>
<td>12.6</td>
<td>56.4</td>
<td>26.0</td>
<td>31.0</td>
</tr>
<tr>
<td>ptx</td>
<td>18.4</td>
<td>103.5</td>
<td>24.2</td>
<td>24.1</td>
</tr>
<tr>
<td>seq</td>
<td>12.1</td>
<td>0.4(1)</td>
<td>30.9</td>
<td>369.3</td>
</tr>
<tr>
<td>Total</td>
<td>1891.0</td>
<td>7360.0</td>
<td>530.9</td>
<td>2424.8</td>
</tr>
</tbody>
</table>

**Inter(Intra)-SDSE**: inter(intra)-procedural shortest-distance symbolic execution

**CCBSE(X)**: call-chain-backward symbolic execution, with X as the forward strategy

**w/CCBSE**: mix with CCBSE(RP)

**RP**: random-path

**Key**: Median SIQR(Outliers) ∞ : time out

<table>
<thead>
<tr>
<th></th>
<th>Otter-KLEE</th>
<th>Otter-SAGE</th>
<th>Random Path</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pure</td>
<td>w/CCBSE</td>
<td>Pure</td>
</tr>
<tr>
<td>mkdir</td>
<td>168.9</td>
<td>31.0</td>
<td>124.7</td>
</tr>
<tr>
<td>mkfifo</td>
<td>41.7</td>
<td>5.2(1)</td>
<td>38.2</td>
</tr>
<tr>
<td>mknod</td>
<td>174.8</td>
<td>24.1</td>
<td>93.1</td>
</tr>
<tr>
<td>paste</td>
<td>22.6</td>
<td>0.5(4)</td>
<td>28.6</td>
</tr>
<tr>
<td>ptx</td>
<td>33.2</td>
<td>3.9</td>
<td>27.1</td>
</tr>
<tr>
<td>seq</td>
<td>354.8</td>
<td>94.3(1)</td>
<td>49.3</td>
</tr>
<tr>
<td>Total</td>
<td>795.8</td>
<td>360.9</td>
<td>4206.4</td>
</tr>
</tbody>
</table>

**Mixing** CCBSE with Otter-KLEE is good, even better than CCBSE and Otter-KLEE alone
### Results

Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution  
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy  
w/CCBSE: mix with CCBSE(RP)  
RP: random-path

<table>
<thead>
<tr>
<th></th>
<th>Inter-SDSE</th>
<th>Intra-SDSE</th>
<th>CCBSE(X) where X is</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>34.7</td>
<td>19.7(10)</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>mkfifo</td>
<td>13.1</td>
<td>0.4</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>0.5</td>
<td>56.4</td>
<td>38.2</td>
</tr>
<tr>
<td>paste</td>
<td>12.6</td>
<td>0.5</td>
<td>56.4</td>
<td>38.2</td>
</tr>
<tr>
<td>ptx</td>
<td>18.4</td>
<td>0.6(4)</td>
<td>103.5</td>
<td>14.5</td>
</tr>
<tr>
<td>seq</td>
<td>12.1</td>
<td>0.4(1)</td>
<td>56.9</td>
<td>14.5</td>
</tr>
<tr>
<td>Total</td>
<td>1891.0</td>
<td>7360.0</td>
<td>530.9</td>
<td>2424.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>InterSDSE</th>
<th>IntraSDSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>31.0</td>
<td>12.1(2)</td>
</tr>
<tr>
<td>mkfifo</td>
<td>5.2(1)</td>
<td>4.6</td>
</tr>
<tr>
<td>mknod</td>
<td>24.1</td>
<td>12.7</td>
</tr>
<tr>
<td>paste</td>
<td>0.5(4)</td>
<td>0.9(3)</td>
</tr>
<tr>
<td>ptx</td>
<td>3.9</td>
<td>2.7</td>
</tr>
<tr>
<td>seq</td>
<td>94.3(1)</td>
<td>5.1(1)</td>
</tr>
<tr>
<td>Total</td>
<td>795.8</td>
<td>360.9</td>
</tr>
</tbody>
</table>

KLEETree statistics:

<table>
<thead>
<tr>
<th></th>
<th>Median</th>
<th>SIQR(Outliers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>mkfifo</td>
<td>2.7(1)</td>
<td></td>
</tr>
<tr>
<td>mknod</td>
<td>10.7</td>
<td></td>
</tr>
<tr>
<td>paste</td>
<td>10.7</td>
<td></td>
</tr>
<tr>
<td>ptx</td>
<td>2.7</td>
<td></td>
</tr>
<tr>
<td>seq</td>
<td>10.7</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>288.8</td>
<td>170.8</td>
</tr>
</tbody>
</table>

**Key:** Median | SIQR(Outliers) | ∞: time out

**SDSE is fast in many cases, but can perform very poorly sometimes.**

**Mixing CCBSE with Otter-KLEE gives the best overall performance.**
Rubyx: Symbolic Execution for Rails

[Abstract Implementation]
Rails API

[Abstract Implementation]
Application

[Abstract Implementation]
Browser + Web Server

[Specifications]

Analysis Script

[Specifications, incl. roles]

Bad states reachable?
Programmable Specifications

• Rubyx provides four special operations
  ■ `fresh(name)` — returns a fresh symbolic variable
  ■ `assume(p)` — adds `p` to the path condition
  ■ `assert(p)` — checks that path condition implies `p`
  ■ `def invariant() p end` — maintains `p` as an invariant for all objects of the class

• In above, `p` can be any Ruby expression
  ■ As in Ruby, `false` and `nil` are false, everything else true

• Writing specs just like writing Ruby tests
  ■ And testing is heavily used in Ruby community
Example Specification

Only admin can modify database

```ruby
# send login request
response = Browser.exec(UserController, :login, fresh(:PARAMS))

# assume that login is successful
assume Browser.session[:id]

# send request to update user information
response = Browser.exec(UserController, :update, fresh(:PARAMS))

# assert that logged in user is admin
assert(User.admin?(Prin.sender)) if Db.modified?(User)
```
Generic Specifications

# No XSS: output sent to trusted users has been sanitized
assert (output.trust?) unless (Prin.receiver == Lattice.bot)

# Secrecy: output secrecy level is at most level of receiver
assert (Lattice.leq (output.secrecy?, Prin.receiver))

# No CSRF: messages must be sent from higher to low trust
# levels, and requests that change state must be POST requests
assert (Lattice.leq (Prin.receiver, Prin.sender)) if params[:post]
assert params[:post] if (Session.modifier? || Db.modified?)

# Authentication: the sender and receiver must be at least as
# trusted as the logged-in user
assert (Lattice.leq (session[Prin.Id], Prin.receiver))
assert (Lattice.leq (session[Prin.Id], Prin.sender)) if params[:post]
Results

- Analyzed 7 apps, found security vulnerabilities in all
- Common lack of understanding of CSRF, session replay
- Most vulnerabilities could be easily fixed

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pubmgr</td>
<td>✓</td>
<td>×(1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×(2)</td>
</tr>
<tr>
<td>coffee</td>
<td>? (2)</td>
<td>×(1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>depot</td>
<td>×(2)</td>
<td>×(10)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>chuckslist</td>
<td>✓</td>
<td>×(1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>× (-)</td>
</tr>
<tr>
<td>boxroom</td>
<td>✓</td>
<td>×(5)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>mystic</td>
<td>×(17)</td>
<td>×(8)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×r (6)</td>
</tr>
<tr>
<td>rtplan</td>
<td>✓</td>
<td>×(1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×r (2)</td>
</tr>
</tbody>
</table>

✓ = no vulner. found
× = vulner. found
? = potential vulner. found

(n) = n fixes
(-) = did not fix
r = replay attack