Improving Program Testing and Understanding via Symbolic Execution

Kin-Keung Ma

PhD Dissertation Defense
December 9th, 2011
Motivation

- Every year, billions of dollars are lost due to software system failures.
  - More than a third of this cost could be avoided if better software testing was performed.
- However, software testing comes with great cost.
  - Typically, half of the man-hours of a software project is dedicated to software testing.

A dilemma?
“Never send a human to do a machine's job.”

— Agent Smith
Prior Work on Symbolic Execution

- A huge body of work has studied designing automated solutions for program testing.
- **Symbolic execution**
  - Proposed in 1970s; has recently become practical
  - Systematically explores execution paths
- **Existing symbolic executors**: KLEE, SAGE, JPF-SE, …
- **Applications**
  - Automated test generation
  - Program verification
Symbolic execution can be improved to

1. Solve the line reachability problem effectively using directed search strategies; and

2. Help understanding configurable software systems by incorporating symbolic execution with coverage analyses.
Contributions

- **Otter, a symbolic execution framework for C programs**
  - Design and implementation
  - Comparison with KLEE, SAGE and JPF-SE

- **Directed symbolic execution**
  - Strategies to generate tests that drive program executions to particular line targets
  - Evaluate strategies on real-world programs

- **Use Otter to analyze behaviors of configurable software**
  - Exhaustively enumerate all paths enabled by config options
  - Track code coverage for each path; derive useful information
Symbolic Execution: An Example

```c
int a = α; // symbolic
if (a<0) {
    if (a==0)
        return 0;
    else
        return -1;
} else {
    if (a>100)
        return 1;
    else
        return 100/a;
}
```
Symbolic Execution: An Example

```c
int a = α; // symbolic
if (a<0) {
    if (a==0)
        return 0;
    else
        return -1;
} else {
    if (a>100)
        return 1;
    else
        return 100/a;
}
```

![Decision tree for symbolic execution example]
Symbolic Execution: An Example

```c
int a = α; // symbolic
if (a<0) {
    if (a==0)
        return 0;
    else
        return -1;
} else {
    if (a>100)
        return 1;
    else
        return 100/a;
}
```

"Path condition"

- **Green path:** \(α \geq 0 \land α \leq 100\)
- **Red path:** \(α < 0 \land α = 0\)
Symbolic Execution: An Example

```c
int a = α; // symbolic
if (a<0) {
    if (a==0)
        return 0;
    else
        return -1;
} else {
    if (a>100)
        return 1;
    else
        return 100/a;
}
```

![Symbolic Execution Diagram](image)
Symbolic Execution: An Example

```c
int a = α; // symbolic
if (a<0) {
    if (a==0)
        return 0;
    else
        return -1;
} else {
    if (a>100)
        return 1;
    else
        return 100/a;
}
```

"Is the divisor zero?"
Otter, a Symbolic Executor for C

- Written in OCaml
- Use the C intermediate language (CIL) infrastructure to transform a C program into a high-level representation
  - Remove redundant constructs (e.g., for(;;) converted to while)
- Use STP as the constraint solver
- State = counter + memory + stack + path condition
  - Faithfully reproducing C's memory model
- Otter manipulates states
  - Keeps a pool of unexplored states
  - Each iteration Otter’s scheduler picks a state to explore
  - State → [symbolic execution] → states/errors
  - Adds states into the pool
Interact with STP, an SMT Solver

- STP solves satisfiability of logic formulae
- Consult STP whenever there are symbolic values in the guard
- Query: is \( e \) true under path condition \( c \)?

<table>
<thead>
<tr>
<th>( e \land c )</th>
<th>( \neg e \land c ) satisfiable</th>
<th>( \neg e \land c ) unsatisfiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>satisfiable</td>
<td>unknown (Otter forks)</td>
<td>true</td>
</tr>
<tr>
<td>unsatisfiable</td>
<td>false</td>
<td>infeasible</td>
</tr>
</tbody>
</table>
Search Strategies

- In many cases, infeasible to explore all paths in a program
- Guide the search based on strategies, e.g.,
  - Random-path: probability of exploring a path with length \( n \sim 1/2^n \)
  - KLEE [OSDI08]: random-path + bias towards uncovered code
  - SAGE [NDSS08]: coverage-guided generational search
- Otter enables convenient extension of strategies
  - Replicate strategies used by KLEE and SAGE
  - New strategies, will be discussed in a moment
Compared to Prior Symbolic Executors

- **KLEE**
  - Uses **LLVM** to transform C programs to bytecode
  - Limited strategies

- **JPF-SE**
  - Java Pathfinder’s symbolic execution extension
  - **Depth-first search**
    - Mostly used to enumerate all paths, i.e., strategies don’t matter
  - Only enumerate paths of **bounded** depth
Compared to Concolic Testing

- **Concrete + symbolic**
- DART, CUTE, SAGE, Pex, …
- **Algorithm:**
  1. Start with a random input; run the program concretely
  2. Collect symbolic constraints along the execution path
  3. Negate one constraint; generate a new input
     - The new input will drive the program to a new path

- **Good**
  - Concrete execution is faster
  - No need to engineer an interpreter

- **Bad**
  - Duplicate work when enumerating all paths
  - Limited strategies
    - Mostly about choosing which one constraint to negate
Directed Symbolic Execution

Joint work with Khoo Yit Phang, Jeffrey Foster and Michael Hicks
Motivation

- Line reachability problem
  - Given a target line in the program, can we find an input that drives the program to that line?
  - Equivalent to: find an input that causes the program to enter a particular state expressed as a conditional
    - E.g., `if(i>MAX) assert(0);`

- Applications
  - Reproduce the error in a bug report
    - Target line from stack trace
  - Triage errors reported by static analysis
    - Target line from error report: detect false positives
Contributions

- Existing strategies: random-path (RP), KLEE, SAGE
  - Aimed at maximizing coverage
- DSE: guide the search towards a target line
  - Shortest-Distance Symbolic Execution (SDSE)
    - Bias the search using inter-procedural CFG-distances to target lines
  - Call-Chain-Backward Symbolic Execution (CCBSE)
    - Start symbolic execution in the function containing the target line;
    - Set the function as target, and repeat this backward up the call chain, until we find a feasible path from the start of the program.
- Mixing CCBSE with a forward strategy (Mix-CCBSE)
  - A forward strategy is any of RP, KLEE, SAGE, SDSE, etc.
Shortest-Distance Symbolic Execution (SDSE)
Shortest-Distance Symbolic Execution (SDSE): Example

```c
int main(void) {
    int argc; char argv[MAX_ARGC][1];
    symbolic(&argc); symbolic(&argv);
    int i, n = 0, b[4] = { 0, 0, 0, 0 };
    for (i = 0; i < argc; i++) {
        if (*argv[i] == 'b') {
            b[n++] = 1; /* potential buf. overflow */
        } else
            foo(); /* some expensive function */
    }
    while (1) {
        if (getchar()) /* get symbolic input */
            /* ...do something... */;
    }
    return 0;
}
```

First 5 iterations:
Shortest-Distance Symbolic Execution (SDSE): Example

```c
int main(void) {
  int argc; char argv[MAX_ARGC][1];
symbolic(&argc); symbolic(&argv);
  int i, n = 0, b[4] = { 0, 0, 0, 0 }; for (i = 0; i < argc; i++) {
    if (*argv[i] == 'b') {
      b[n++] = 1; /* potential buf. overflow */
    } else
      foo(); /* some expensive function */
  }
  while (1) {
    if (getchar()) /* get symbolic input */
      /* ...do something... */;
  }
  return 0;
}
```

Program input, made symbolic

First 5 iterations:

entry

- argc=0 argv[0] = 'b' argv[0] ≠ 'b'
- argc=1 argv[1] = 'b' argv[1] ≠ 'b'

buffer overflow!
Shortest-Distance Symbolic Execution (SDSE): Example

```c
int main(void) {
    int argc; char argv[MAX_ARGC][1];
symbolic(&argc); symbolic(&argv);
    int i, n = 0, b[4] = { 0, 0, 0, 0 };
    for (i = 0; i < argc; i++) {
        if (*argv[i] == 'b') {
            b[n++] = 1; /* potential buf. overflow */
        } else {
            foo(); /* some expensive function */
        }
    }
    while (1) {
        if (getchar()) /* get symbolic input */
            /* ...do something... */;
    }
    return 0;
}
```

b is a buffer of size 4. n is the next position in b to write into.
Shortest-Distance Symbolic Execution (SDSE): Example

```c
int main(void) {
    int argc; char argv[MAX_ARGC][1];
    symbolic(&argc); symbolic(&argv);
    int i, n = 0, b[4] = { 0, 0, 0, 0 };
    for (i = 0; i < argc; i++) {
        if (*argv[i] == 'b') {
            b[n++] = 1; /* potential buf. overflow */
        } else
            foo(); /* some expensive function */
    }
    while (1) {
        if (getchar()) /* get symbolic input */
            /* ...do something... */;
    }
    return 0;
}
```

A for-loop iterating over `argv` for 5 iterations:
Shortest-Distance Symbolic Execution (SDSE): Example

```c
int main(void) {
    int argc; char argv[MAX_ARGC][1];
symbolic(&argc); symbolic(&argv);
    int i, n = 0, b[4] = { 0, 0, 0, 0 };
    for (i = 0; i < argc; i++) {
        if (*argv[i] == 'b') {
            b[n++] = 1; /* potential buf. overflow */
        } else
            foo(); /* some expensive function */
    }

    while (1) {
        if (getchar()) /* get symbolic input */
            /* ...do something... */;
    }
    return 0;
}
```

If the argument is a letter b, put a 1 into b.
Shortest-Distance Symbolic Execution (SDSE): Example

```c
int main(void) {
    int argc; char argv[MAX_ARGC][1];
    symbolic(&argc); symbolic(&argv);
    int i, n = 0, b[4] = { 0, 0, 0, 0 };
    for (i = 0; i < argc; i++) {
        if (*argv[i] == 'b') {
            b[n++] = 1; /* potential buf. overflow */
        } else
            foo(); /* some expensive function */
    }

    while (1) {
        if (getchar()) /* get symbolic input */
            /* ...do something... */;
    }
    return 0;
}
```

If the argument is NOT a letter b, run `foo()` which is expensive.
Shortest-Distance Symbolic Execution (SDSE): Example

```c
int main(void) {
    int argc; char argv[MAX_ARGC][1];
symbolic(&argc); symbolic(&argv);
    int i, n = 0, b[4] = { 0, 0, 0, 0 }; 
    for (i = 0; i < argc; i++) {
        if (*argv[i] == 'b') {
            b[n++] = 1; /* potential buf. overflow */
        } else
            foo(); /* some expensive function */
    }

    while (1) {
        if (getchar()) /* get symbolic input */ /* ...do something... */;
    }
    return 0;
}
```

First 5 iterations:

- `argc=0` `argv[0]='b'` `argv[0]≠'b'`
- `argc=1` `argv[1]='b'` `argv[1]≠'b'`

an infinite loop, doing work

buffer overflow!
Shortest-Distance Symbolic Execution (SDSE): Example

```
1 int main(void) {
2     int argc; char argv[MAX_ARGC][1];
3     symbolic(&argc); symbolic(&argv);
4     int i, n = 0, b[4] = { 0, 0, 0, 0 };
5     for (i = 0; i < argc; i++) {
6         if (*.argv[i] == 'b') {
7             b[n++] = 1; /* potential buf. overflow */
8         } else
9             foo(); /* some expensive function */
10     }
11     while (1) {
12         if (getchar()) /* get symbolic input */
13             /* ...do something... */;
14     }
15     return 0;
16 }
```

buffer overflow occurs when \( n \geq 4 \)

6 iterations:
A conditional to detect buffer overflow. `assert(0)` is the target.
Shortest-Distance Symbolic Execution (SDSE): Example

```c
int main(void) {
    int argc; char argv[MAX_ARGC][1];
    symbolic(&argc); symbolic(&argv);
    int i, n = 0, b[4] = { 0, 0, 0, 0 }; 
    for (i = 0; i < argc; i++) {
        if (*argv[i] == 'b') {
            if (n >= 4)
                assert(0);
            b[n++] = 1; /* potential buf. overflow */
        } else
            foo(); /* some expensive function */
    }
    while (1) {
        if (getchar()) /* get symbolic input */
            /* ...do something... */;
    }
    return 0;
}
```

First 5 iterations:

- Green path is the shortest path
- Buffer overflow!
Shortest-Distance Symbolic Execution (SDSE): Example

```c
int main(void) {
    int argc; char argv[MAX_ARGC][1];
    symbolic(&argc); symbolic(&argv);
    int i, n = 0, b[4] = { 0, 0, 0, 0 };
    for (i = 0; i < argc; i++) {
        if (*argv[i] == 'b') {
            if (n >= 4)
                assert(0);
            b[n++] = 1; /* potential buf. overflow */
        } else
            foo(); /* some expensive function */
    }
    while (1) {
        if (getchar()) /* get symbolic input */
            /* …do something… */;
    }
    return 0;
}
```

First 5 iterations:

- Green path is the shortest path
- Distance = 2
- Buffer overflow!
Shortest-Distance Symbolic Execution (SDSE): Distance Calculation

1. Build a CFG for each function
2. Build an inter-procedural CFG by splitting function calls into call and return nodes, and connecting them to function entries and exits.
Shortest-Distance Symbolic Execution (SDSE): Distance Calculation

3. Compute shortest distances of paths with matching calls and returns
Invalid path, because path enters bar from one call site and returns to another call site.
Variants of SDSE

- **SDSE-intra**
  - Intra-procedural SDSE
  - Target is not in the current function $\Rightarrow$ distance $= \infty$

- **SDSE-pr**
  - Probabilistic SDSE
  - Explore a path with probability $\propto \frac{1}{(its\ distance\ to\ target)}$

- **RR(RP,SDSE)**
  - Round-robin of random-path and SDSE
  - $RR(X_1, X_2, ...)$: circularly use one strategy at a time
  - Avoid getting stuck by mixing with a random strategy

- **Batched versions of these variants**
  - $B(X)$: keep exploring the same path if it doesn’tfork
    - Up to a number of steps
  - Avoid recalculating SDSE too often
  - But may overrun on paths that are not interesting
Call-Chain-Backward Symbolic Execution (CCBSE)
Call-Chain-Backward Symbolic Execution (CCBSE): Idea
Call-Chain-Backward Symbolic Execution (CCBSE): Idea

Start at function (foo) containing the target. Run forward symbolic execution using some strategy X to find the target.
Once found, go back to its caller (bar). Run forward symbolic execution to find the call to foo.
Call-Chain-Backward Symbolic Execution (CCBSE): Idea

Before searching thru foo, check if we can continue along the known path-to-target in foo.
If realizable, we immediately have a path-to-target from bar by stitching the two paths. We don’t need to explore foo again.
Call-Chain-Backward Symbolic Execution (CCBSE): Idea

Repeat this process until we go back to main.

- Underlying forward strategy as a parameter of CCBSE: \textit{CCBSE}(X)
- Paths as summaries: can reuse the symbolic executor machinery
Call-Chain-Backward Symbolic Execution (CCBSE): Requirement

- Require starting symbolic execution at any function
- Initialize function parameters and global variables
  - Primitive types: purely symbolic
  - Pointers: one of
    - NULL
    - A single object of the base type
    - For primitive base types, an array of objects of the base type
      - To model things like a C-string (char*)
- Lazy initialization for arbitrarily recursive data structures
  - A pointer is not initialized until program “uses” it
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```c
void main() {
    int m, n, i;
    symbolic(&m); symbolic(&n);
    for (i=0; i<1000; i++)
        if (m == i) f(m, n);
}

void f(int m, int n) {
    int i, a, sum=0;
    for (i=0; i<6; i++)
        a = n%2;
        if (a) sum += a+1;
        n/=2;
    }
while(1) {
    if (sum==0 && m==7)
        assert(0);
    }
```
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```c
1    void main() {
2        int m, n, i;
3        symbolic(&m); symbolic(&n);
4        for (i=0; i<1000; i++)
5            if (m == i) f(m, n);
6    }
7    void f(int m, int n) {
8        int i, a, sum=0;
9        for (i=0; i<6; i++) {
10           a = n%2;
11           if (a) sum += a+1;
12           n/=2;
13        }
14    while(1) {
15        if (sum==0 & m==7)
16               assert(0);
17    }
18    }
```

m, n are input
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```c
void main() {
    int m, n, i;
    symbolic(&m); symbolic(&n);
    for (i=0; i<1000; i++)
        if (m == i) f(m, n);
}

void f(int m, int n)
int i, a, sum=0;
    for (i=0; i<6; i++)
        a = n%2;
        if (a) sum += a+1;
    n/=2;
}
while(1) {
    if (sum==0 && m==7)
        assert(0);
}
```

\[ f(m, n) \text{ is called for all } m \text{ in } [0,1000) \]
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```c
void main() {
  int m, n, i;
  symbolic(&m); symbolic(&n);
  for (i=0; i<1000; i++)
    if (m == i) f(m, n);
}

void f(int m, int n) {
  int i, a, sum=0;
  for (i=0; i<6; i++) {
    a = n%2;
    if (a) sum += a+1;
    n /= 2;
  }
  while(1) {
    if (sum==0 && m==7)
      assert(0);
  }
}
```

Entry:
```
m==0 \rightarrow m==1 \rightarrow \cdots \rightarrow m==999
```

Function Call:
```
f(m, n)
```

Example Calculation:
```
sum+=a_0+1 \quad sum+=a_1+1 \quad sum+=a_5+1
```

Calculate the sum of 6 least significant bits of n.
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```c
void main() {
    int m, n, i;
    symbolic(&m); symbolic(&n);
    for (i=0; i<1000; i++)
        if (m == i) f(m, n);
}
void f(int m, int n) {
    int i, a, sum=0;
    for (i=0; i<6; i++)
        a = n%2;
    if (a) sum += a+1;
    n /= 2;
}
while(1) {
    if (sum==0 && m==7)
        assert(0);
}
```
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```c
void main() {
    int m, n, i;
    symbolic(&m); symbolic(&n);
    for (i=0; i<1000; i++)
        if (m == i) f(m, n);
}
void f(int m, int n) {
    int i, a, sum=0;
    for (i=0; i<6; i++) {
        a = n%2;
        if (a) sum += a+1;
        n /= 2;
    }
    while(1) {
        if (sum==0 && m==7)
            assert(0);
    }
}
```

Failing condition 1: m==7; directly determined by m in f(m,n)
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```c
void main() {
    int m, n, i;
    symbolic(&m); symbolic(&n);
    for (i=0; i<1000; i++)
        if (m == i) f(m, n);
}
void f(int m, int n) {
    int i, a, sum=0;
    for (i=0; i<6; i++) {
        a = n%2;
        if (a) sum += a+1;
        n /= 2;
    }
    while(1) {
        if (sum==0 && m==7)
            assert(0);
    }
}
```

The only path among $2^6$ paths that has $\text{sum}==0$: the true branch at line 11 is never taken.

Failing condition 2: $\text{sum}==0$; indirectly determined by $n$ in $f(m,n)$
Every time \( f(m, n) \) is called, CCBSE first checks if the green path is realizable. It’s always realizable since \( n \) is unconstrained. When \( m == 7 \), the assertion is reachable. CCBSE avoids re-exploring the \( 2^6 \) paths in \( f \).
Mixing CCBSE with Forward Strategies

- Simultaneously run
  - CCBSE
  - Forward search from main
- Try merging when the forward search reaches a function that has a path to target found by CCBSE
- Best case: better than forward/CCBSE alone due to smaller search spaces
- Worst case: linear to the runtime of forward search (plus overhead)
Mixing CCBSE with Forward Strategies

- **Simultaneously** run
  - CCBSE
  - Forward search from `main`
- **Try merging** when the forward search reaches a function that has a path to target found by CCBSE
- Best case: better than forward/CCBSE alone due to smaller search spaces
- Worst case: linear to the runtime of forward search (plus overhead)
Mixing CCBSE with Forward Strategies

- Simultaneously run
  - CCBSE
  - Forward search from `main`
- Try merging when the forward search reaches a function that has a path to target found by CCBSE
- Best case: better than forward/CCBSE alone due to smaller search spaces
- Worst case: linear to the runtime of forward search (plus overhead)
Mixing CCBSE with Forward Strategies

- **Simultaneously** run
  - CCBSE
  - Forward search from `main`
- **Try merging** when the forward search reaches a function that has a path to target found by CCBSE
- Best case: better than forward/CCBSE alone due to smaller search spaces
- Worst case: linear to the runtime of forward search (plus overhead)
Evaluation

- Compare different strategies
  - SDSE, SDSE-pr, SDSE-intra, RR(RP,SDSE), and their batched ver.
  - CCBSE with RP and SDSE
  - RP, OtterKLEE and OtterSAGE
    - With and without mixing with CCBSE(RP)
  - KLEE, as a reference

- Benchmark: 10 bugs in 9 programs from Coreutils 6.10
  - Targets are bugs previously reported by KLEE

- Each test is run 21 times using different seeds
  - To measure the effect of randomness
### Results

<table>
<thead>
<tr>
<th>Program</th>
<th>SDSE</th>
<th>SDSE-pr</th>
<th>SDSE-intra</th>
<th>B(SDSE)</th>
<th>B(SDSE-pr)</th>
<th>RR(RP,SDSE)</th>
<th>B(RR(RP,SDSE))</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>35.4</td>
<td>232.4</td>
<td>244.8</td>
<td>127.7</td>
<td>159.4</td>
<td>∞</td>
<td>∞</td>
<td>667.0</td>
</tr>
<tr>
<td>mkfifo</td>
<td>23.2</td>
<td>1,051.5</td>
<td>237.9</td>
<td>21.6</td>
<td>25.1</td>
<td>21.1</td>
<td>21.5</td>
<td>656.5</td>
</tr>
<tr>
<td>mknoed</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1,218.0</td>
<td>33.8</td>
</tr>
<tr>
<td>paste</td>
<td>18.9</td>
<td>57.0</td>
<td>23.7</td>
<td>21.6</td>
<td>25.1</td>
<td>21.1</td>
<td>21.5</td>
<td>51.6</td>
</tr>
<tr>
<td>seq</td>
<td>574.5</td>
<td>42.3</td>
<td>31.6</td>
<td>407.6</td>
<td>41.0</td>
<td>1,731.4</td>
<td>674.5</td>
<td>313.4</td>
</tr>
<tr>
<td>ptx</td>
<td>439.0</td>
<td>47.5</td>
<td>31.6</td>
<td>407.6</td>
<td>41.0</td>
<td>1,731.4</td>
<td>674.5</td>
<td>∞</td>
</tr>
<tr>
<td>ptx2</td>
<td>1,729.6</td>
<td>∞</td>
<td>26.9</td>
<td>∞</td>
<td>∞</td>
<td>293.7</td>
<td>293.8</td>
<td>∞</td>
</tr>
<tr>
<td>md5sum</td>
<td>25.6</td>
<td>297.8</td>
<td>529.4</td>
<td>316.4</td>
<td>131.7</td>
<td>102.0</td>
<td>9.9</td>
<td>∞</td>
</tr>
<tr>
<td>tac</td>
<td>∞</td>
<td>∞</td>
<td>3,943.9</td>
<td>∞</td>
<td>∞</td>
<td>3,045.0</td>
<td>3,045.0</td>
<td>∞</td>
</tr>
<tr>
<td>pr</td>
<td>953.9</td>
<td>1,418.4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Total</td>
<td>12,800.1</td>
<td>21,230.7</td>
<td>24,823.4</td>
<td>15,381.3</td>
<td>14,626.0</td>
<td>13,584.9</td>
<td>11,644.8</td>
<td>21,522.3</td>
</tr>
</tbody>
</table>

### CCBSE w/ X = SDSE/RR

<table>
<thead>
<tr>
<th>Program</th>
<th>CCBSE w/ X = SDSE</th>
<th>Pure</th>
<th>RP</th>
<th>CCBSE w/ X = SDSE</th>
<th>Pure</th>
<th>RP</th>
<th>CCBSE w/ X = SDSE</th>
<th>Pure</th>
<th>RP</th>
<th>CCBSE w/ X = SDSE</th>
<th>Pure</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>∞</td>
<td>148.4</td>
<td>31.8(4)</td>
<td>199.6</td>
<td>32.8(5)</td>
<td>152.4</td>
<td>32.8(5)</td>
<td>337.1</td>
<td>32.8(5)</td>
<td>389.8</td>
<td>32.8(5)</td>
<td>143.9</td>
</tr>
<tr>
<td>mkfifo</td>
<td>25.7</td>
<td>12.1</td>
<td>25.7</td>
<td>12.1</td>
<td>25.7</td>
<td>12.1</td>
<td>25.7</td>
<td>12.1</td>
<td>25.7</td>
<td>12.1</td>
<td>25.7</td>
<td>12.1</td>
</tr>
<tr>
<td>mknoed</td>
<td>∞</td>
<td>199.1</td>
<td>31.8(4)</td>
<td>182.2</td>
<td>32.8(5)</td>
<td>122.3</td>
<td>32.8(5)</td>
<td>116.4</td>
<td>32.8(5)</td>
<td>126.5</td>
<td>32.8(5)</td>
<td>205.8</td>
</tr>
<tr>
<td>paste</td>
<td>22.8</td>
<td>12.1</td>
<td>22.8</td>
<td>12.1</td>
<td>22.8</td>
<td>12.1</td>
<td>22.8</td>
<td>12.1</td>
<td>22.8</td>
<td>12.1</td>
<td>22.8</td>
<td>12.1</td>
</tr>
<tr>
<td>seq</td>
<td>1,791.9</td>
<td>∞</td>
<td>1,791.9</td>
<td>∞</td>
<td>1,791.9</td>
<td>∞</td>
<td>1,791.9</td>
<td>∞</td>
<td>1,791.9</td>
<td>∞</td>
<td>1,791.9</td>
<td>∞</td>
</tr>
<tr>
<td>ptx</td>
<td>1,010.4</td>
<td>502.5</td>
<td>101.0</td>
<td>101.0</td>
<td>101.0</td>
<td>101.0</td>
<td>101.0</td>
<td>101.0</td>
<td>101.0</td>
<td>101.0</td>
<td>101.0</td>
<td>101.0</td>
</tr>
<tr>
<td>ptx2</td>
<td>∞</td>
<td>665.1</td>
<td>35.8(5)</td>
<td>735.5</td>
<td>35.8(5)</td>
<td>1,062.3</td>
<td>35.8(5)</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>md5sum</td>
<td>36.0</td>
<td>11.8</td>
<td>36.0</td>
<td>11.8</td>
<td>36.0</td>
<td>11.8</td>
<td>36.0</td>
<td>11.8</td>
<td>36.0</td>
<td>11.8</td>
<td>36.0</td>
<td>11.8</td>
</tr>
<tr>
<td>tac</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>pr</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Total</td>
<td>22,686.8</td>
<td>17,813.5</td>
<td>16,249.7</td>
<td>17,617.1</td>
<td>20,709.2</td>
<td>19,984.6</td>
<td>13,414.5</td>
<td>14,961.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results

SDSE: shortest-distance symbolic execution; -pr: probabilistic; -intra: intraprocedural
B(X): batched X  RR(X,Y): round-robin X and Y
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>Program</th>
<th>SDSE</th>
<th>SDSE-pr</th>
<th>SDSE-intra</th>
<th>B(SDSE)</th>
<th>B(SDSE-pr)</th>
<th>RR(RP,SDSE)</th>
<th>B(RR(RP,SDSE))</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>35.4</td>
<td>232.4</td>
<td>424.9</td>
<td>127.7</td>
<td>221.0</td>
<td>159.4</td>
<td>320.3</td>
<td>∞</td>
</tr>
<tr>
<td>mkfifo</td>
<td>23.2</td>
<td>1,051.5</td>
<td>1,051.5</td>
<td>22.1</td>
<td>92.4</td>
<td>243.4</td>
<td>414.5</td>
<td>∞</td>
</tr>
<tr>
<td>mkmod</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>paste</td>
<td>18.9</td>
<td>57.0</td>
<td>23.7</td>
<td>21.6</td>
<td>24.8</td>
<td>25.1</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>seq</td>
<td>574.5</td>
<td>42.3</td>
<td>974.8</td>
<td>31.6</td>
<td>37.4</td>
<td>33.7</td>
<td>25.3</td>
<td>∞</td>
</tr>
<tr>
<td>ptx</td>
<td>439.0</td>
<td>47.5</td>
<td>974.8</td>
<td>31.6</td>
<td>37.4</td>
<td>33.7</td>
<td>25.3</td>
<td>∞</td>
</tr>
<tr>
<td>ptx2</td>
<td>1,729.6</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>md5sum</td>
<td>25.6</td>
<td>26.9</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>tac</td>
<td>953.9</td>
<td>3,943.9</td>
<td>5,824.6</td>
<td>31.5</td>
<td>33.7</td>
<td>33.7</td>
<td>23.3</td>
<td>∞</td>
</tr>
<tr>
<td>pr</td>
<td>4,628.4</td>
<td>124,244.7</td>
<td>17,617.1</td>
<td>20,700.5</td>
<td>23,714.5</td>
<td>14,961.9</td>
<td>9,200.5</td>
<td>∞</td>
</tr>
<tr>
<td>Total</td>
<td>12,800.1</td>
<td>124,244.7</td>
<td>17,617.1</td>
<td>20,700.5</td>
<td>23,714.5</td>
<td>14,961.9</td>
<td>9,200.5</td>
<td>121,522.3</td>
</tr>
</tbody>
</table>

Key: Median SIQR(Outliers) ∞: time out

Semi-interquartile range (SIQR) (seconds)
Median (seconds)
# of outliers

Fastest two times per row are highlighted

Time out: 1,800s (tac, pr: 7,200s)
Results

<table>
<thead>
<tr>
<th>Program</th>
<th>SDSE</th>
<th>SDSE-pr</th>
<th>SDSE-intra</th>
<th>B(SDSE)</th>
<th>B(SDSE-pr)</th>
<th>RR(RP,SDSE)</th>
<th>B(RR(RP,SDSE))</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>35.4</td>
<td>232.4</td>
<td>424.9</td>
<td>127.7</td>
<td>159.4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>mkefio</td>
<td>23.2</td>
<td>1,051.5</td>
<td>∞</td>
<td>22.1</td>
<td>93.4</td>
<td>451.6</td>
<td>258.1</td>
<td>667.0</td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>57.0</td>
<td>23.7</td>
<td>21.6</td>
<td>25.1</td>
<td>21.1</td>
<td>21.5</td>
<td>33.8</td>
</tr>
<tr>
<td>paste</td>
<td>18.9</td>
<td>57.0</td>
<td>23.7</td>
<td>21.6</td>
<td>25.1</td>
<td>21.1</td>
<td>21.5</td>
<td>33.8</td>
</tr>
<tr>
<td>seq</td>
<td>574.5</td>
<td>42.3</td>
<td>407.6</td>
<td>41.0</td>
<td>1,731.4</td>
<td>674.5</td>
<td>51.6</td>
<td>313.4</td>
</tr>
<tr>
<td>ptx</td>
<td>439.0</td>
<td>47.5</td>
<td>974.8</td>
<td>31.6</td>
<td>122.9</td>
<td>97.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ptx2</td>
<td>1,729.6</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>293.7</td>
<td>290.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>md5sum</td>
<td>25.6</td>
<td>∞</td>
<td>21.8</td>
<td>∞</td>
<td>33.7</td>
<td>23.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tac</td>
<td>∞</td>
<td>21.8</td>
<td>∞</td>
<td>21.8</td>
<td>33.7</td>
<td>23.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pr</td>
<td>953.9</td>
<td>121.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>12,800.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Huge SIQRs (∞ means the 75th %tile timed out)

Huge SIQRs (154s), and many outliers (5)
Results

SDSE: shortest-distance symbolic execution; -pr: probabilistic; -intra: intraprocedural
B(X): batched X  RR(X,Y): round-robin X and Y
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>Program</th>
<th>SDSE</th>
<th>SDSE-pr</th>
<th>SDSE-intra</th>
<th>B(SDSE)</th>
<th>B(SDSE-pr)</th>
<th>RR(RP,SDSE)</th>
<th>B(RR(RP,SDSE))</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>35.4</td>
<td>232.4</td>
<td>424.9</td>
<td>127.7</td>
<td>159.4</td>
<td>∞</td>
<td>∞</td>
<td>667.0</td>
</tr>
<tr>
<td>mkfifo</td>
<td>23.2</td>
<td>1,051.5</td>
<td>∞</td>
<td>22.1</td>
<td>93.4</td>
<td>451.6</td>
<td>258.1</td>
<td>656.5</td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1,218.0</td>
<td>33.8</td>
</tr>
<tr>
<td>paste</td>
<td>18.9</td>
<td>57.0</td>
<td>23.7</td>
<td>21.6</td>
<td>25.1</td>
<td>21.5</td>
<td>21.5</td>
<td>19.1</td>
</tr>
<tr>
<td>seq</td>
<td>574.5</td>
<td>42.3</td>
<td>∞</td>
<td>407.6</td>
<td>41.0</td>
<td>1,731.4</td>
<td>674.5</td>
<td>51.6</td>
</tr>
<tr>
<td>ptx</td>
<td>439.0</td>
<td>47.5</td>
<td>974.8</td>
<td>31.6</td>
<td>37.4</td>
<td>122.9</td>
<td>97.2</td>
<td>313.4</td>
</tr>
<tr>
<td>ptx2</td>
<td>1,729.6</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>293.7</td>
<td>239.8</td>
<td>∞</td>
</tr>
<tr>
<td>md5sum</td>
<td>25.6</td>
<td>∞</td>
<td>∞</td>
<td>26.9</td>
<td>∞</td>
<td>31.5</td>
<td>33.7</td>
<td>∞</td>
</tr>
<tr>
<td>tac</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5,824.6</td>
<td>102.0</td>
<td>∞</td>
</tr>
<tr>
<td>pr</td>
<td>953.9</td>
<td>∞</td>
<td>∞</td>
<td>3,943.9</td>
<td>3,045.0</td>
<td>11,644.8</td>
<td>14,961.9</td>
<td>∞</td>
</tr>
<tr>
<td>Total</td>
<td>12,800.1</td>
<td>21,230.7</td>
<td>24,823.4</td>
<td>15,381.3</td>
<td>14,626.0</td>
<td>13,584.9</td>
<td>11,644.8</td>
<td>21,522.3</td>
</tr>
</tbody>
</table>

Using interprocedural distance is crucial for SDSE

Distance heuristic works very well in practice
### Results

<table>
<thead>
<tr>
<th>Program</th>
<th>SDSE</th>
<th>SDSE-pr</th>
<th>SDSE-intra</th>
<th>B(SDSE)</th>
<th>B(SDSE-pr)</th>
<th>RR(RP,SDSE)</th>
<th>B(RR(RP,SDSE))</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>35.4</td>
<td>232.4</td>
<td>424.9</td>
<td>127.7</td>
<td>159.4</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>mkfifo</td>
<td>23.2</td>
<td>1,051.5</td>
<td>∞</td>
<td>22.1</td>
<td>93.4</td>
<td>451.6</td>
<td>258.1</td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1,218.0</td>
<td>667.0</td>
</tr>
<tr>
<td>paste</td>
<td>18.9</td>
<td>57.0</td>
<td>23.7</td>
<td>21.6</td>
<td>25.1</td>
<td>21.5</td>
<td>21.5</td>
</tr>
<tr>
<td>seq</td>
<td>574.5</td>
<td>42.3</td>
<td>∞</td>
<td>407.6</td>
<td>41.0</td>
<td>1,731.4</td>
<td>674.5</td>
</tr>
<tr>
<td>ptx</td>
<td>439.0</td>
<td>47.5</td>
<td>974.8</td>
<td>31.6</td>
<td>37.4</td>
<td>122.9</td>
<td>97.2</td>
</tr>
<tr>
<td>ptx2</td>
<td>1,729.6</td>
<td>∞</td>
<td>∞</td>
<td>26.9</td>
<td>31.5</td>
<td>33.7</td>
<td>33.7</td>
</tr>
<tr>
<td>md5sum</td>
<td>25.6</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5,824.6</td>
<td>1,130.6</td>
<td>1,062.3</td>
</tr>
<tr>
<td>tac</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>3,943.9</td>
<td>3,045.0</td>
<td>102.0</td>
<td>9.9</td>
</tr>
<tr>
<td>pr</td>
<td>953.9</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Total**: 12,800.1  21,230.7  24,823.4  15,381.3  14,626.0  13,584.9  11,644.8  21,522.3

**KLEE**

<table>
<thead>
<tr>
<th>Program</th>
<th>Median</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>∞</td>
<td>667.0</td>
</tr>
<tr>
<td>mkfifo</td>
<td>656.5</td>
<td>33.8</td>
</tr>
<tr>
<td>mknod</td>
<td>51.6</td>
<td>13.2</td>
</tr>
<tr>
<td>paste</td>
<td>313.4</td>
<td>∞</td>
</tr>
<tr>
<td>seq</td>
<td>51.6</td>
<td>33.8</td>
</tr>
<tr>
<td>ptx</td>
<td>313.4</td>
<td>∞</td>
</tr>
<tr>
<td>ptx2</td>
<td>313.4</td>
<td>∞</td>
</tr>
<tr>
<td>md5sum</td>
<td>313.4</td>
<td>∞</td>
</tr>
<tr>
<td>tac</td>
<td>313.4</td>
<td>∞</td>
</tr>
<tr>
<td>pr</td>
<td>313.4</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Total**: 6,462.6  14,961.9

**Key:** Median[SDSE(Outlier)] ∞: time out

---

**SDSE**: shortest-distance symbolic execution; **-pr**: probabilistic; **-intra**: intraprocedural

**B(X)**: batched X  **RR(X,Y)**: round-robin X and Y

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

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

---

**Batching can hurt SDSE’s effectiveness**
Results

<table>
<thead>
<tr>
<th>Program</th>
<th>SDSE</th>
<th>SDSE-pr</th>
<th>SDSE-intra</th>
<th>B(SDSE)</th>
<th>B(SDSE-pr)</th>
<th>RR(RP,SDSE)</th>
<th>B(RR(RP,SDSE))</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>35.4</td>
<td>∞</td>
<td>232.4</td>
<td>424.9</td>
<td>127.7</td>
<td>159.4</td>
<td>322.5</td>
<td>∞</td>
</tr>
<tr>
<td>mkfifo</td>
<td>23.2</td>
<td>0.9(3)</td>
<td>1,051.5</td>
<td>∞</td>
<td>22.1</td>
<td>93.4</td>
<td>451.6</td>
<td>258.1</td>
</tr>
<tr>
<td>mkdir</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>paste</td>
<td>18.9</td>
<td>1.4(3)</td>
<td>85.9</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>seq</td>
<td>574.5</td>
<td>98.4(3)</td>
<td>159.9</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>ptx</td>
<td>439.0</td>
<td>31.3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>ptx2</td>
<td>1,729.6</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>md5sum</td>
<td>25.6</td>
<td>1.7(3)</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>3,943.9</td>
<td>131.7</td>
<td>102.0</td>
</tr>
<tr>
<td>tac</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>pr</td>
<td>953.9</td>
<td>141.9(3)</td>
<td>3,943.9</td>
<td>131.7</td>
<td>102.0</td>
<td>11,644.8</td>
<td>21,522.3</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>12,800.1</td>
<td>21,280.7</td>
<td>24,823.9</td>
<td>15,381.1</td>
<td>4,626.0</td>
<td>13,584.9</td>
<td>11,644.8</td>
<td>21,522.3</td>
</tr>
</tbody>
</table>

**Mixing CCBSE with forward strategy X can be better than X alone**

<table>
<thead>
<tr>
<th>Program</th>
<th>CCBSE w/ X=</th>
<th>OtterKLEE</th>
<th>OtterSAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>∞</td>
<td>148.4</td>
<td>337.1</td>
</tr>
<tr>
<td>mkfifo</td>
<td>25.7</td>
<td>62.2</td>
<td>108.4</td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>199.1</td>
<td>116.4</td>
</tr>
<tr>
<td>paste</td>
<td>22.8</td>
<td>27.9</td>
<td>24.5</td>
</tr>
<tr>
<td>seq</td>
<td>1,791.9</td>
<td>407.1</td>
<td>341.7</td>
</tr>
<tr>
<td>ptx</td>
<td>1,010.4</td>
<td>103.8</td>
<td>79.0</td>
</tr>
<tr>
<td>ptx2</td>
<td>∞</td>
<td>665.1</td>
<td>399.6</td>
</tr>
<tr>
<td>md5sum</td>
<td>36.0</td>
<td>73.5</td>
<td>610.4</td>
</tr>
<tr>
<td>tac</td>
<td>∞</td>
<td>4,826.6</td>
<td>3,165.7</td>
</tr>
<tr>
<td>pr</td>
<td>∞</td>
<td>5,729.5</td>
<td>6,462.6</td>
</tr>
<tr>
<td>Total</td>
<td>22,686.8</td>
<td>16,249.7</td>
<td>19,984.6</td>
</tr>
</tbody>
</table>

Key: Median [Min-Max], ∞ : time out
### Results

**SDSE**: shortest-distance symbolic execution; **-pr**: probabilistic; **-intra**: intraprocedural  
**B(X)**: batched X  
**RR(X,Y)**: round-robin X and Y  
**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>Program</th>
<th>SDSE</th>
<th>SDSE-pr</th>
<th>SDSE-intra</th>
<th>B(SDSE)</th>
<th>B(SDSE-pr)</th>
<th>RR(RP,SDSE)</th>
<th>B(RR(RP,SDSE))</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>35.4</td>
<td>∞</td>
<td>232.4</td>
<td>424.9</td>
<td>127.7</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>mknifo</td>
<td>23.2</td>
<td>1.4(9)</td>
<td>1,051.5</td>
<td>∞</td>
<td>22.1</td>
<td>93.3</td>
<td>451.6</td>
<td>258.1</td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>paste</td>
<td>18.9</td>
<td>1.4(5)</td>
<td>57.9</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>seq</td>
<td>574.5</td>
<td>106(4)</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>ptx</td>
<td>439.0</td>
<td>311.2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>ptx2</td>
<td>1,729.6</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>md5sum</td>
<td>25.6</td>
<td>1.7(9)</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>3,943.9</td>
<td>984.2</td>
<td>∞</td>
</tr>
<tr>
<td>tac</td>
<td>953.9</td>
<td>1,414(5)</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>pr</td>
<td>∞</td>
<td>1,414(5)</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>12,800.1</td>
<td>21,230.7</td>
<td>24,823.4</td>
<td>15,381.0</td>
<td>4,626.0</td>
<td>13,584.9</td>
<td>11,644.8</td>
<td>21,522.3</td>
</tr>
</tbody>
</table>

**Key:**  
- Median $$\text{SQRT}(\text{Outlier})$$  
- ∞: time out

**Mixing CCBSE with forward strategy X can be better than CCBSE and X alone.**
Results

<table>
<thead>
<tr>
<th>Program</th>
<th>SDSE</th>
<th>SDSE-pr</th>
<th>SDSE-intra</th>
<th>B(SDSE)</th>
<th>B(SDSE-pr)</th>
<th>RR(RP,SDSE)</th>
<th>B(RR(RP,SDSE))</th>
<th>KLEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir</td>
<td>35.4</td>
<td>232.4</td>
<td>424.9</td>
<td>127.7</td>
<td>159.4</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>mkfifo</td>
<td>23.2</td>
<td>1,051.5</td>
<td>∞</td>
<td>21.1</td>
<td>93.4</td>
<td>451.6</td>
<td>258.1</td>
<td>667.0</td>
</tr>
<tr>
<td>mknod</td>
<td>∞</td>
<td>57.0</td>
<td>23.7</td>
<td>21.6</td>
<td>21.1</td>
<td>21.5</td>
<td>33.8</td>
<td></td>
</tr>
<tr>
<td>paste</td>
<td>18.9</td>
<td>57.3</td>
<td>23.7</td>
<td>21.6</td>
<td>21.1</td>
<td>21.5</td>
<td>51.6</td>
<td></td>
</tr>
<tr>
<td>seq</td>
<td>574.5</td>
<td>42.3</td>
<td>31.6</td>
<td>293.7</td>
<td>239.8</td>
<td>313.4</td>
<td>313.4</td>
<td></td>
</tr>
<tr>
<td>ptx</td>
<td>439.0</td>
<td>47.5</td>
<td>31.6</td>
<td>293.7</td>
<td>239.8</td>
<td>313.4</td>
<td>313.4</td>
<td></td>
</tr>
<tr>
<td>ptx2</td>
<td>1,729.6</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>31.5</td>
<td>33.7</td>
<td>33.7</td>
<td></td>
</tr>
<tr>
<td>md5sum</td>
<td>25.6</td>
<td>∞</td>
<td>∞</td>
<td>5,824.6</td>
<td>131.7</td>
<td>102.0</td>
<td>9.9</td>
<td></td>
</tr>
<tr>
<td>tac</td>
<td>953.9</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>pr</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>3,943.9</td>
<td>3,045.0</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>12,800.1</td>
<td>21,230.7</td>
<td>24,823.4</td>
<td>15,381.3</td>
<td>13,584.9</td>
<td>11,644.8</td>
<td>21,522.3</td>
<td></td>
</tr>
</tbody>
</table>

**Key:** Median | Speedup | Time-out

**B(RR(RP,SDSE)) performs the best overall, and it finishes in reasonable amount of time for most benchmark programs.**
Multi-target DSE

- Given **a set of line targets**, find inputs that drive the program execution to **as many targets as possible**
  - Within a time limit

- **Application**: improve code coverage
  - **Targets**: lines **not covered** by the program’s test suite
  - Corner cases where tests are hard to derive

- **Extending SDSE to solve multi-target DSE**
  - **SDSE**: shortest distance to **any one target**
  - **SDSE-pr**: probability \( \propto \frac{1}{(\text{shortest distance to any one target})} \)
  - **SDSE-rr**: shortest distance to **one target at a time**
  - **Ph(OtterKLEE,SDSE,3)**
    - Run OtterKLEE first; when coverage rate slows down, switch to SDSE
    - Only consider batched versions of these variants
      - Batching is more likely to help in longer runs
Multi-target DSE: Line-Target Coverage

- Same experimental setup as single-target
  - Targets: lines not covered by Coreutils’ test suite
  - Time limit: 2 hours; median of 5 runs
- Avg%: average line-target coverage over all benchmark programs
- Agg%: aggregated line-target coverage
  - # covered lines in all programs / # line targets in all programs

<table>
<thead>
<tr>
<th>Program</th>
<th>B(SDSE)</th>
<th>B(SDSE-pr)</th>
<th>B(SDSE-rr)</th>
<th>B(RR(RP, SDSE))</th>
<th>B(RR(RP, SDSE-rr))</th>
<th>B(Ph(OtterKLEE, SDSE,3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg %</td>
<td>51.9</td>
<td>45.2</td>
<td>55.5</td>
<td>63.3</td>
<td>63.1</td>
<td>50.9</td>
</tr>
<tr>
<td>Agg %</td>
<td>31.0</td>
<td>29.6</td>
<td>45.5</td>
<td>48.2</td>
<td>46.5</td>
<td>41.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>CCBSE(RP)</th>
<th>OtterKLEE</th>
<th>OtterSAGE</th>
<th>RP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Pure</td>
<td>w/CCBSE</td>
<td>Pure</td>
<td>w/CCBSE</td>
</tr>
<tr>
<td>Avg %</td>
<td>62.2</td>
<td>60.6</td>
<td>59.8</td>
<td>61.0</td>
</tr>
<tr>
<td>Agg %</td>
<td>48.3</td>
<td>45.4</td>
<td>49.2</td>
<td>47.9</td>
</tr>
</tbody>
</table>

SDSE: shortest-distance symbolic execution; -pr: probabilistic
B(X): batched X  RR(X,Y): round-robin X and Y  Ph(X,Y,r): run X first, then Y, based on factor r
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy
w/CCBSE: mix with CCBSE(RP)  RP: random-path
Multi-target DSE: Coverage over Time
Multi-target DSE: Coverage over Time

Normalized coverage over time:
Run all benchmark programs in parallel. At time t, if strategy X covers some target from program P, X’s coverage at time t is increased by 1/(total # target in P)
Multi-target DSE: Coverage over Time

Pure SDSE strategies didn’t perform well
Multi-target DSE: Coverage over Time

Undirected strategies start off great, but slow down after a while.
Multi-target DSE: Coverage over Time

Mix-CCBSEs perform better than their forward-only constituents
Multi-target DSE: Coverage over Time

B(RR(RP,SDSE-rr)) starts off slower, but catches up and performs better than others most of the time.
Summary

- Directed symbolic execution
  - Guide the search towards a target line

- Our contributions:
  - Shortest-Distance Symbolic Execution (SDSE)
  - Call-Chain-Backward Symbolic Execution (CCBSE)
  - Mixing CCBSE with a forward strategy (Mix-CCBSE)

- Single-target DSE
  - Each directed strategy has its strength and weakness
  - B(RR(RP,SDSE)) performs the best overall

- Multi-target DSE
  - B(RR(RP,SDSE-rr)) starts off slightly slower than undirected strategies, but catches up and performs the best
Using Symbolic Execution to Understand Behavior in Configurable Software Systems

Joint work with Elnatan Reisner, Charles Song, Jeffrey Foster and Adam Porter
Motivation

- Sometimes it is feasible to enumerate all execution paths
  - Small (finite) symbolic input space
- Example: configurable software
  - Use configuration options to switch on/off program features
  - Options often boolean, or take values from a finite set
    - Small (finite) symbolic input space!
Use Otter to Find Per-Path Coverage

- Studied vsftpd, ngIRCd and grep
  - Each has run-time configuration options
- Choose a set of options to be symbolic
- Use Otter to enumerate all paths enabled by these options
  - Under their test suites
- Track per-path coverage information
  - Consider line, block, edge, and condition coverage
- Support further analysis
Advantages of using Otter

- Naively enumerate all configurations is infeasible
- Concolic testers are bad at enumerating all paths
- KLEE’s use of LLVM to transform C programs to bytecode may confuse coverage tracking at source code level
- JPF-SE appears unable to scale to software systems of the size of those we studied (~10kloc each)
Empirical Study

- Ran Otter on 40 machines for 2 weeks
- \# combinations = test cases × \# configurations
- \# paths: across all test cases
- \# paths <<<< \# combinations!
- Each path explored by Otter corresponds to many combinations
  - Each path only “looks” at a subset of options

<table>
<thead>
<tr>
<th>Subject Programs</th>
<th>vsftpd</th>
<th>ngIRCd</th>
<th>grep</th>
</tr>
</thead>
<tbody>
<tr>
<td># combinations</td>
<td>137,438,953,472</td>
<td>61,834,752</td>
<td>66,650,112</td>
</tr>
<tr>
<td># paths</td>
<td>30,304</td>
<td>53,205</td>
<td>625,181</td>
</tr>
</tbody>
</table>
Conclusions

- Otter, a symbolic execution framework for C programs
  - Useful in improving program testing and understanding
- Directed symbolic execution
  - Solve single-target and multi-target line reachability problems
  - Mixing SDSE strategies with RP performs well
  - Good performance of directed strategies
- Use Otter to analyze behavior of configurable software
  - Enumerating all paths enabled by config options is practical
  - Enumerating all combinations of options is not
Otter is available for download!

https://bitbucket.org/khooyp/otter
Thank You