Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing

Timotej Kapus, Cristian Cadar

Department of Computing
Imperial College London
if (x > 2294967295) {
    assert(false);
}
printf("x: %u\n", x);
Symbolic execution

● Used in industry:
  ○ IntelliTest
  ○ SAGE
  ○ KLOVER
  ○ SPF
  ○ Apollo
● Active research field
Symbolic execution

```c
1  unsigned int x = 5;
2  int main() {
3      if (x > 2294967295) {
4          assert(false);
5      }
6      printf("x: %u\n",x);
7  }
```
1 unsigned int x = 5;
2 int main() {
3     make_symbolic(&x);
4     if (x > 2294967295) {
5         assert(false);
6     } else {
7         printf("x: %d\n", x);
8     }
9 }

Symbolic execution

x = *

TRUE

x > 2294967295

FALSE

x ≤ 2294967295

x > 2294967295

assert(false);

x ≤ 2294967295

printf("x: %d", x);

Assertion fail

x: 2
Many available open source
Complex pieces of software
  - Accurate interpreter or precise instrumentation
  - Accurate constraint solving
  - Constraint gathering
  - Scheduling
  - Effective optimizations such as caching, fast solving, etc.

Symbolic executors

CREST

Angr
Symbolic executors

- Many available open source
- Complex pieces of software
  - Accurate interpreter or precise instrumentation
  - Accurate constraint solving
  - Constraint gathering
  - Scheduling
  - Effective optimizations such as caching, fast solving, etc.
Bugs in symbolic executors

- Particularly bad
- Lead to false sense of security
- Examples:
  - Missing a branch
  - Exploring spurious branches

```c
unsigned int x = 5;
int main() {
    make_symbolic(&x);
    if(x > 2294967295) {
        assert(false);
    }
    printf("x: %u\n",x);
}
```
Differential testing of symbolic execution

Randomly generated program

Compile

Execute

Compare

Compile

Symbolically execute
Testing symbolic executors

- Compare two executions (native/symbolic) in 3 different modes:
  - **Concrete** - tests interpretation/instrumentation
  - **Single Path** - tests constraint gathering and solving
  - **Multi Path** - tests scheduling, test case generation
Concrete mode

1 unsigned int x = 5;
2 int main() {
3     if (x > 2294967295) {
4         assert(false);
5     }
6     printf("x: %u\n", x);
7 }
Single-Path mode

```c
1 unsigned int x = 5;
2 int main() {
3     make_symbolic(&x);
4     CONSTRAIN(x, 5);
5     if(x > 2294967295) {
6         assert(false);
7     }
8     printf("x: \%u\n",x);
9 }
```

x: 5

Assertion fail
Single-Path mode: Constrainers

```c
1 unsigned int x = 5;
2 int main() {
3     make_symbolic(&x);
4     CONSTRAIN(x, 5);
5     if(x > 2294967295) {
6         assert(false);
7     }
8     printf("x: %u\n", x);
9 }
```

if(x < 5) silent_exit(0);
if(x > 5) silent_exit(0);
Single-Path mode: Constrainers

```c
unsigned int x = 5;

int main() {
    make_symbolic(&x);

    CONSTRANIN(x, 5);

    if(x > 2294967295) {
        assert(false);
    }

    printf("x: %u\n", x);
}
```

**if(x != 5) silent_exit(0);**
Multi-Path mode

1 unsigned int x = 5;
2 int main() {
3     make_symbolic(&x);
4     if (x > 2294967295) {
5         assert(false);
6     }
7     printf("x: %u\n", x);
8 }

Test case: x = 7
Test case: x = 23

MATCH!
Multi-Path mode

```c
int x = 5;

void main() {
    make_symbolic(&x);
    if(x < 0)
        printf("x: %d", -x);
    else
        printf("x: %d", x);
}
```

Test case: $x = -7$
Test case: $x = 23$

- $x: 7$
- $x: -23$
- $x: 7$
- $x: 23$

16
Testing symbolic executors

- Built a pipeline
- Run experiments in batches
- Avoid bugs found in previous batches
Instrumentation supports

- **Csmith**
  - Random program generator
  - Found many bugs in compilers
  - Doesn’t generate programs with undefined behaviour

- **Instrumentation supports**:
  - Marking variables as symbolic
  - Oracles
  - Constraining
Versions correspond to mode:
- Concrete mode - native version
- Single path mode - single path version
- Multi path mode - multi path version
Oracles can check:
1. Executor doesn’t crash
2. Function call chain
3. Output (values of all global variables) matches
4. Coverage achieved on the random program
Finally:
- Gather mismatches
- Reduce interesting ones
- Report bugs
Case Studies

KLEE

- Main case study
  - Familiarity
  - Flexibility
- Built on top of LLVM
- Keeps all paths in memory

CREST

- Concolic execution
- Instrumentation instead of interpretation
- Doesn’t generate test cases

FuzzBALL

- Binary level executor
- Doesn’t generate test cases
KLEE summary of runs

Number of runs

- Concrete: 520930 runs, 116 total run time
- Single-Path: 168648 runs, 94 total run time
- Single-Path with coverage: 1992 runs, 51 total run time
- Multi-Path: 6625 runs, 72 total run time

Total run time (h)

Number of runs
- Concrete: 116 total run time
- Single-Path: 94 total run time
- Single-Path with coverage: 51 total run time
- Multi-Path: 72 total run time
Summary of bugs found

<table>
<thead>
<tr>
<th>Tool</th>
<th>Found</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>KLEE</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>Crest</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FuzzBALL</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
KLEE bugs by mode

- **Concrete mode**
- **Single-Path mode**
- **Single-Path and Multi-Path mode**
- **Multi-Path mode**
Crest bugs by mode

- Concrete mode
- Single-Path mode
FuzzBALL bugs by mode

- Single-Path mode

3
Bugs found by oracle

- **Crash**: KLEE 6, Crest 1, FuzzBALL 2
- **Output**: KLEE 4, Crest 2, FuzzBALL 1
- **Function call chain**: KLEE 3, Crest 0, FuzzBALL 0
- **Output & Function call chain**: KLEE 1, Crest 0, FuzzBALL 0
Example bug: *Crest*

```c
1  unsigned int a;
2  int main() {
3      make_symbolic(&a);
4      if(a > 2294967295) {
5          assert(false);
6      }
7      printf("a: %d\n",a);
8  }
```

<table>
<thead>
<tr>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: 6</td>
<td>a: 6</td>
</tr>
<tr>
<td>Assertion fail</td>
<td>a: 23</td>
</tr>
</tbody>
</table>
Example bug: *KLEE*

```c
1 int g_10 = 0;
2 int main() {
3     make_symbolic(&g_10);
4     do {
5         printf("loop\n");
6         g_10 &= 2;
7     } while(!((3 ^ g_10) / 1));
8 }
```

<table>
<thead>
<tr>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
</tbody>
</table>
| loop            | ...

The code snippet demonstrates a bug in the symbolic execution tool *KLEE*. The expected output is a loop that should terminate, but the actual output shows the loop indefinitely, indicating a potential issue in the tool's handling of symbolic values.
Example bug: FuzzBALL

1 unsigned int g_54 = 0;
2 unsigned int g_56 = 0;
3
4 void main ( void ) {
5     make_symbolic(&g_54);
6     CONSTRAIN(g_54, 0);
7     g_56 ^= 0 < g_54;
8     printf("g_56: %u\n", *(g_56));
9 }
Conclusions

- Developed techniques that test many aspects of symbolic executors
- Applied them to 3 different symbolic executors
- Total bugs found:
  - 14 in *KLEE*
  - 3 in *Crest*
  - 3 in *FuzzBALL*
## Constrainers

<table>
<thead>
<tr>
<th>Type</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;,&gt;$</td>
<td>$\neg (x &lt; v) \land \neg (x &gt; v)$</td>
</tr>
<tr>
<td>$\leq, \geq$</td>
<td>$x \leq v \land x \geq v$</td>
</tr>
<tr>
<td>range</td>
<td>$\neg (x \leq v - 2) \land \neg (x \geq v + 3) \land$  $\neg (x = v - 1) \land \neg (x = v + 1) \land \neg (x = v + 2)$</td>
</tr>
<tr>
<td>divisors</td>
<td>$\land_i \neg (x \mod d_i \neq 0) \land x &gt; 1 \land x \leq v$</td>
</tr>
</tbody>
</table>
KLEE bugs by oracle

- Crash
- Output
- Output & Function call chain
- Function call chain
CREST bug

- 14 in KLEE (9 fixed)
- 3 in Crest (1 fixed)
- 3 in FuzzBALL (3 fixed)

- found within first 5000 runs of a batch

```c
unsigned int a;
int main() {
    CrestUInt(&a);
    printf("a: \%d\n", a);
    if (a < 2294967295) {
        exit(0);
    }
}
```
Single-Path Mode

Compare native execution, with symbolic execution constrained to the exact same path as native execution.

```c
int x = 5;
makesymbolic(&x);
if (x < 5) silent_exit(0);
if (x > 5) silent_exit(0);
```
Symbolic execution

- Mark some inputs as symbolic
- Runs the program, while gathering constraints on the symbolic data
- Forks at branch points when both sides are feasible
- Upon hitting a terminal state (i.e. error), solves the gathered constraints, to produce an input leading the program to the same state
Configuration includes:

- Program generation options
  - size/complexity of the program
  - language features to use
- Compilation options
- Mode
- Oracles to use
Instrumentation supports

- Marking variables as symbolic
- Oracles
- Constraining
versions correspond to mode used:
● Concrete mode - native version
● Single path mode - single path version
● Multi path mode - multi path version
Oracles can check:
1. Executor doesn’t crash
2. Function call chain
3. Output (values of all global variables) matches
4. Coverage achieved on the program
Finally:

- Gather mismatches
- Reduce interesting ones
- Report bugs
```c
void foo(unsigned int x) {
    if (x > 2294967295) {
        assert(false);
    }
    printf("x: %u\n", x);
}
```

<table>
<thead>
<tr>
<th>Expected output</th>
<th>Actual output</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: 6</td>
<td>x: 6</td>
</tr>
<tr>
<td>Assertion fail</td>
<td>x: 23</td>
</tr>
</tbody>
</table>