**SymSan**: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis

Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, Insik Shin
Concolic Execution: CONC(rete) + Symb(OLIC) execution

void foo(int x, int y) {
    z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            assert(0);
    }
}

Concrete State

Symbolic State

Testcase #1

Testcase #2
Overhead

\[ z = x \oplus y \]

Parsing:
- Interpretation, slow (Angr, KLEE)
- Instrumentation, faster! (QSYM, SymQEMU, SymCC, SymSan)

Locating:
- Reading Sym(x) and Sym(b) from symbolic state

Creating/ updating:
- Creating Sym(z), updating symbolic state

Locating/creating/updating have non-trivial overhead
Insight

• Concolic execution is a special form of dynamic data-flow analysis, so...
• it can be simply implemented on top of LLVM DFSan (highly-optimized)
Use SymSan

CC=symsan CXX=symsan make
Concolic Execution is forward data-flow analysis

<table>
<thead>
<tr>
<th></th>
<th>Generic Data-flow analysis</th>
<th>Concolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label Interpretation</td>
<td>Variables’ properties</td>
<td>Variables’ symbolic expressions</td>
</tr>
<tr>
<td>Label Introduction</td>
<td>How labels are introduced</td>
<td>Program inputs</td>
</tr>
<tr>
<td>Label Propagation</td>
<td>How labels are updated after executing an instruction</td>
<td>Compute symbolic expressions</td>
</tr>
<tr>
<td>Label Sinks</td>
<td>Where and how properties are used</td>
<td>Conditional branches. Update constraints</td>
</tr>
</tbody>
</table>

Var $\rightarrow$ Label (symbolic expression)
A label is (the index of) a symbolic expression

```c
struct dfsan_label_info {
    dfsan_label l1;  // symbolic sub-expression
    dfsan_label l2;  // symbolic sub-expression
    u64 op1;        // concrete operand
    u64 op2;        // concrete operand
    u16 op;         // opcode, using LLVM IR operations
    u16 size;       // size of the result
};
```

Union Table

- SymX
- SymY
- SymZ

```
... Lbl101  Lbl102  Lbl103 ...
... SymX   SymY   SymZ ...
```
Running example

$z = x + y$

Before Exec.

| Shadow Mem |  
| --- | --- |
| $x$ | 101  |
| $y$ | 102  |

After Exec.

| Shadow Mem |  
| --- | --- |
| $x$ | 101  |
| $y$ | 102  |
| $z$ | 103  |

Union Table (AST Table)

<table>
<thead>
<tr>
<th>...</th>
<th>Lbl101</th>
<th>Lbl102</th>
<th>Lbl103</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>SymX</td>
<td>SymY</td>
<td>UNINIT</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>...</th>
<th>Lbl101</th>
<th>Lbl102</th>
<th>Lbl103</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>SymX</td>
<td>SymY</td>
<td>L1:101 L2:102 OP: ADD</td>
<td>...</td>
</tr>
</tbody>
</table>
Why is SymSan faster?

<table>
<thead>
<tr>
<th></th>
<th>SymCC</th>
<th>SymSan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic expression</td>
<td>std::shared_ptr</td>
<td>A 32-bit integer</td>
</tr>
<tr>
<td>Shadow memory access</td>
<td>std::map, O(logn)</td>
<td>Direct mapping, O(1)</td>
</tr>
<tr>
<td>Symbolic expression</td>
<td>new/malloc()</td>
<td>atomic_fetch_inc()</td>
</tr>
<tr>
<td>allocation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arguments/Return value</td>
<td>std::array, multiple function</td>
<td>TLS, single MOV instruction</td>
</tr>
<tr>
<td>passing</td>
<td>calls</td>
<td></td>
</tr>
</tbody>
</table>
Efficiency (collecting constraints, no solving)

- Two orders of magnitude faster in constraints collecting
- Solves 2x more constraints given the same time budget
Memory Efficiency (no solving)

- Consumes one order of magnitude less memory
- Good for async-solving
Hybrid-fuzzing

- Fuzzbench (SymSan ranked #1 in average score)
Takeaways

- An efficient concolic executor, built on top of DFSan
- Doing source-based concolic execution? Try SymSan
  - near-optimal performance!
- World’s fastest 😊 if pairing with JIGSAW (Oakland’22)
Thank you for listening!

https://github.com/R-Fuzz/symsan