# 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

31st USENIX SECURITY SYMPOSIUM
AUGUST 11, 2022

# 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);
}
```

2*y0 != x0

x: 10  y: 6  Testcase #1

2*y0 == x0 && x0 > y0 + 5

x: 20  y: 10  Testcase #2

Concrete State

x -> 10
y -> 5

→

x -> 10
y -> 5
z -> 10

Symbolic State

x -> x0
y -> y0

→

x -> x0
y -> y0
z -> 2*y0

# 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

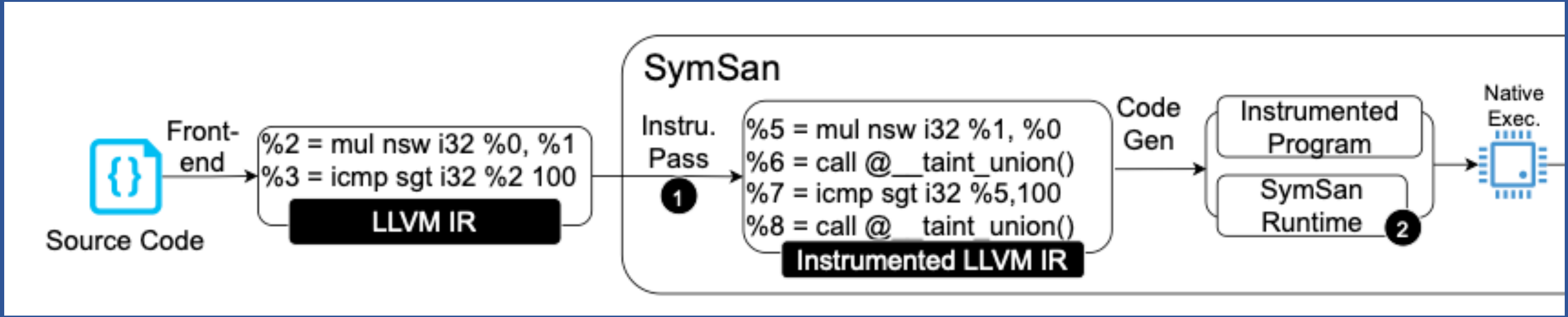
Locating/creating/updating have non-trivial overhead

# Insight

- Concolic execution is a special from 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
```



```
%2 = mul nsw i32 %0, %1
%3 = icmp sgt i32 %2 100
```

```
%5 = mul nsw i32 %1, %0
%6 = call @__taint_union()
%7 = icmp sgt i32 %5,100
%8 = call @__taint_union()
```

# Concolic Execution is forward data-flow analysis

| | **Generic Data-flow analysis** | **Concolic Execution** |
|---|---|---|
| Label Interpretation | Variables' properties | Variables' symbolic expressions |
| Label Introduction | How labels are introduced | Program inputs |
| Label Propagation | How labels are updated after executing an instruction | Compute symbolic expressions |
| Label Sinks | Where and how properties are used | Conditional branches. Update constraints |

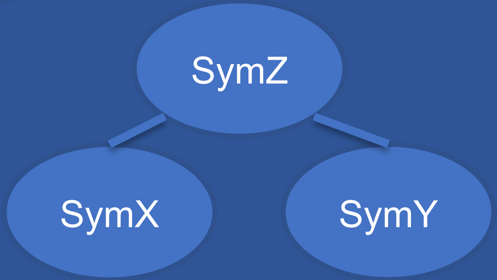
Var -> Label (symbolic expression)

# A label is (the index of) a symbolic expression

```
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
};
```

| … | Lbl101 | Lbl102 | Lbl103 | … |
|---|---|---|---|---|
| … | SymX | SymY | SymZ | … |

Union Table



# Running example

$$z = x + y$$

Shadow Mem

Union Table ( AST Table )

Before Exec.

| | |
|---|---|
| x | 101 |
| y | 102 |

| … | Lbl101 | Lbl102 | Lbl103 | … |
|---|---|---|---|---|
| … | SymX | SymY | UNINIT | … |

After Exec.

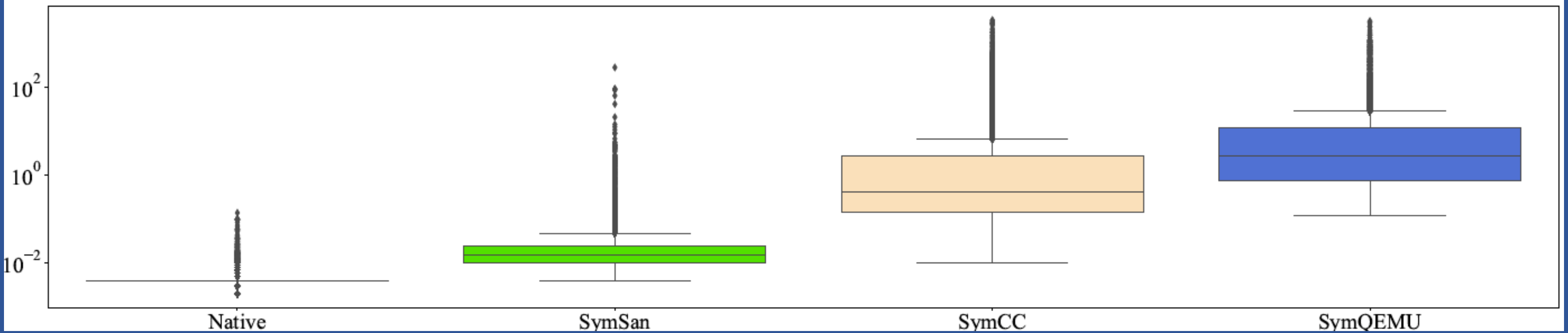
| | |
|---|---|
| x | 101 |
| y | 102 |
| z | 103 |

| … | Lbl101 | Lbl102 | Lbl103 | … |
|---|---|---|---|---|
| … | SymX | SymY | L1:101<br>L2:102<br>OP: ADD | … |

# Why is SymSan faster?

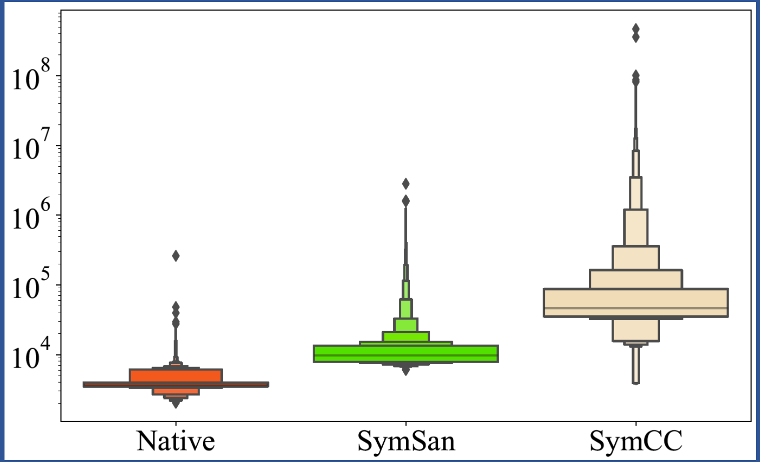
| | **SymCC** | **SymSan** |
|---|---|---|
| Symbolic expression | std::shared_ptr | A 32-bit integer |
| Shadow memory access | std::map, O(logn) | Direct mapping, O(1) |
| Symbolic expression allocation | new/malloc() | atomic_fetch_inc() |
| Arguments/Return value passing | std::array, multiple function calls | TLS, single MOV instruction |

# Efficiency (collecting constraints, no solving)



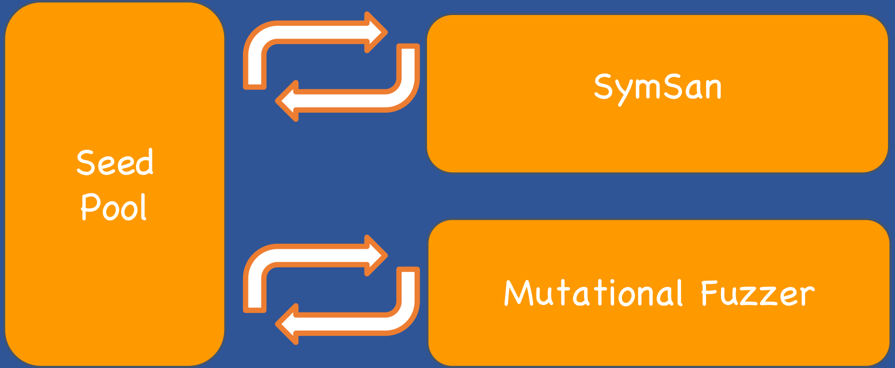
- Two orders of magnitude faster in constraints collecting
- Solves 2x more constraints given the same time budget

# 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

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

ARTIFACT EVALUATED
usenix ASSOCIATION
FUNCTIONAL