Symbolic execution with SymCC: Don't interpret, compile!

•••

Sebastian Poeplau, Aurélien Francillon

Compiling symbolic-execution capabilities into executables

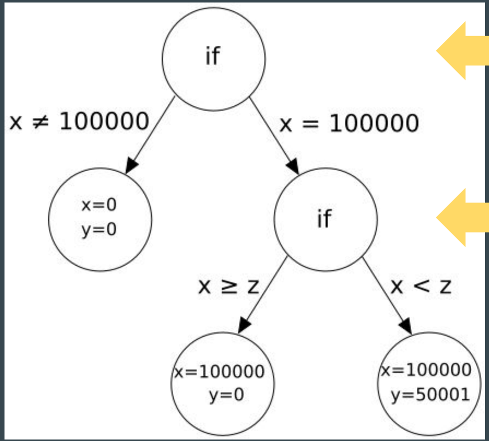
Recap: Symbolic Execution

Explore programs by keeping track of computations in terms of inputs

Target program

```
void f(int x, int y) {
  int z = 2*y;
  if (x == 100000) {
     if (x < z) {
        assert(0); /* error */
     }
  }
}
```

symbolic execution



Current approaches
(e.g., KLEE, S2E, angr)

Interpreter approach

Target program (bitcode)

```
define i32 @is_double(i32, i32) {
  %3 = shl nsw i32 %1, 1
  %4 = icmp eq i32 %3, %0
  %5 = zext i1 %4 to i32
  ret i32 %5
}
```

N times

Interpreter (e.g., KLEE, S2E, angr)

```
while (true) {
  auto instruction = getNextInstruction();
  switch (instruction.type) {
    // ...
    case SHL: {
      auto result = instruction.operand(0) <<
                         instruction.operand(1);
      auto resultExpr =
        buildLeftShift(instruction.operandExpr(0),
                             instruction.operandExpr(1));
      setResult(result, resultExpr);
      break;
    }
  }
}
```

SymCC
Compilation instead of interpretation

SymCC: Overview

Target program (bitcode)

```
define i32 @is_double(i32, i32) {
  %3 = shl nsw i32 %1, 1
  %4 = icmp eq i32 %3, %0
  %5 = zext i1 %4 to i32
  ret i32 %5
}
```

once

Instrumented target (bitcode)

```
define i32 @is_double(i32, i32) {
  %3 = call i8* @_sym_get_parameter_expression(i8 0)
  %4 = call i8* @_sym_get_parameter_expression(i8 1)
  %5 = call i8* @_sym_build_integer(i64 1)
  %6 = call i8* @_sym_build_shift_left(i8* %4, i8* %5)
  %7 = call i8* @_sym_build_equal(i8* %6, i8* %3)
  %8 = call i8* @_sym_build_bool_to_bits(i8* %7)

  %9 = shl nsw i32 %1, 1
  %10 = icmp eq i32 %9, %0
  %11 = zext i1 %10 to i32

  call void @_sym_set_return_expression(i8* %8)
  ret i32 %11
}
```

SymCC: Implementation

- Compiler pass and run-time library
- Pass inserts calls to the run-time library at compile time
  - → Built on top of LLVM
  - → Easily integrate with all LLVM-based compilers
  - → Independent of CPU architecture and source language
- Run-time library builds up symbolic expressions and calls the solver
  - → Two options for run-time library
  - → "Simple backend": wrapper around Z3, little optimization, good for debugging
  - → "QSYM backend": reuse expressions and solver infrastructure from QSYM (but NOT the instrumentation!)

QSYM is different

- Yun et al., USENIX Security 2018
- Based on dynamic binary instrumentation
  - → Rewrites binaries at run time using Intel Pin
  - → Inserts calls to functions that build symbolic expressions and interacts with a solver
- Strengths
  - → No interpreter: higher performance than interpreted systems
  - → Support for binaries
- But...
  - → Rewritten program is less efficient than compiled programs
  - → Binary level, i.e., need to implement symbolic handling for *each x86 instruction*

Recap

We compile symbolic-execution capabilities right into the binary.

- Most others interpret
- QSYM uses dynamic binary instrumentation

Evaluation
Benchmark and real-world targets

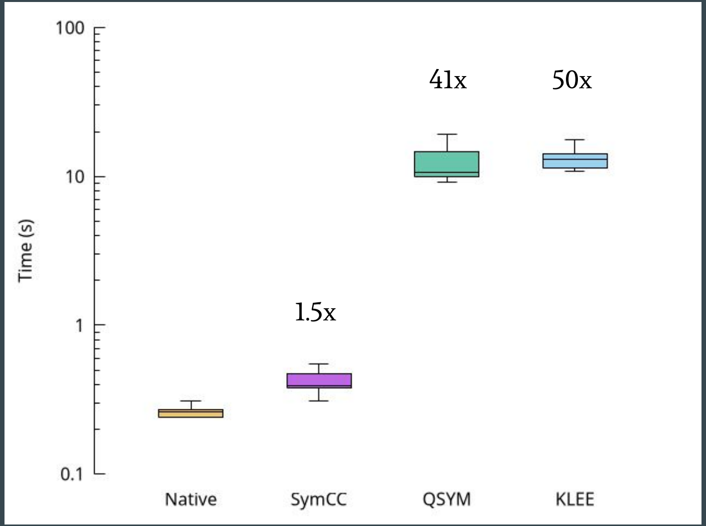
Benchmark: Setup

- Goal: highly controlled environment
- DARPA CGC programs
- Concolic execution with fixed inputs
  - → Fixed code paths
  - → Single execution with generation of new inputs
- Intel Core i7 CPU and 32GB of RAM
- 30 minutes for a single execution
  (regular, i.e. non-symbolic, execution takes milliseconds)
- Compared with KLEE and QSYM
  - → Excluded S2E: very similar to KLEE in aspects that matter here
  - → Excluded angr: not optimized for execution speed

Benchmark: Execution Speed

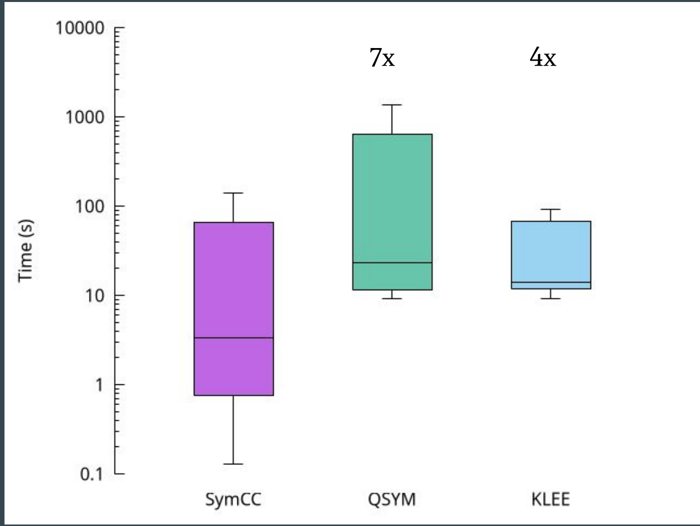
Fully concrete

No symbolic input provided



Concolic

Input data is made symbolic



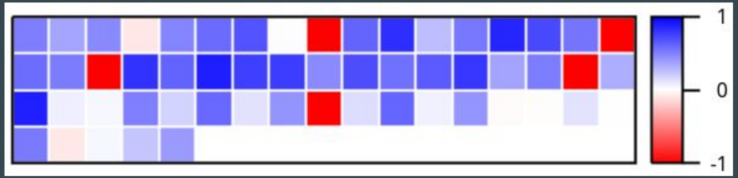
Benchmark: Coverage

Approach

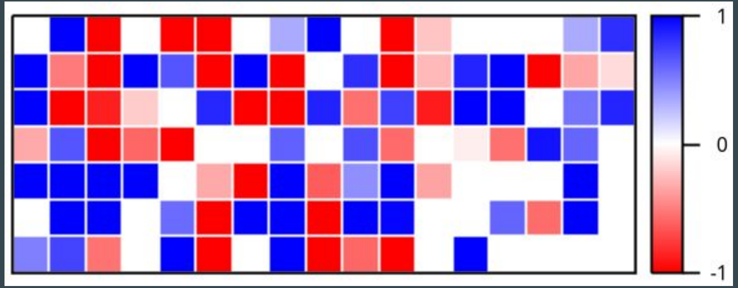
After concolic execution, measure edge coverage of newly generated inputs with afl-showmap.

Visualization

- Compare paths found by only one system
- More intense color: more unique paths
- Blue for SymCC, red for KLEE/QSYM



Comparison with KLEE (56 programs): SymCC is better on 46 and worse on 10



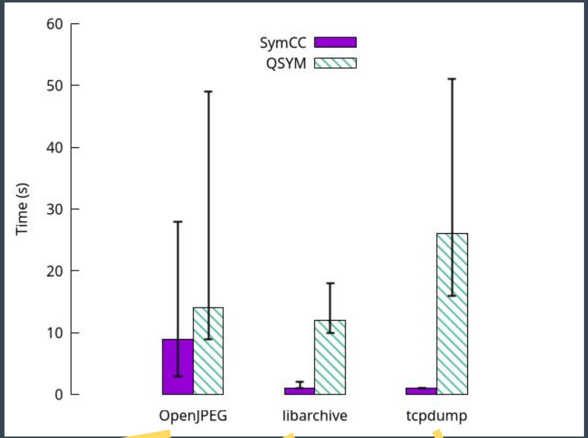
Comparison with QSYM (116 programs): SymCC is better on 47, worse on 40, and equal on 29

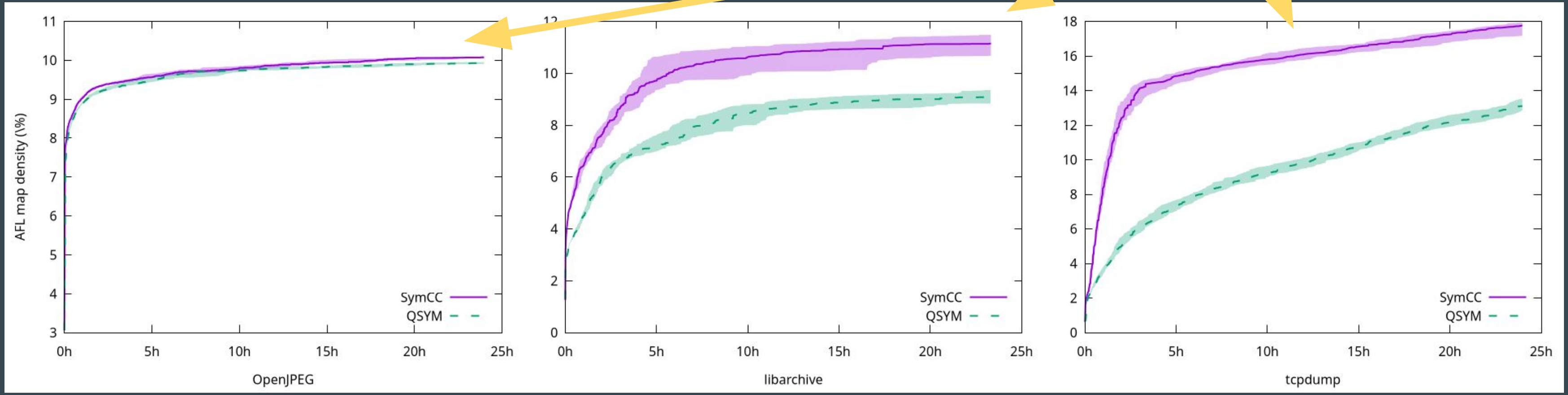
Real-world targets: Setup

- Goal: show scalability to real-world software
- Popular open-source projects: OpenJPEG, libarchive, tcpdump
- Hybrid fuzzing: AFL and concolic execution with SymCC/QSYM
    - → Same approach as Driller and QSYM
    - → 2 AFL processes, 1 SymCC/QSYM (like in QSYM's evaluation)
- Intel Xeon Platinum 8260 CPU with 2GB of RAM *per core*
- 24 hours, 30 iterations (→ roughly 17 CPU core months)
- Excluded KLEE: unsupported instructions in target programs

Real-world targets: Results

- Higher coverage than QSYM
- Statistically significant coverage difference (Mann-Whitney-U, $p < 0.0002$)
- Found 2 CVEs in OpenJPEG
- Speed advantage correlates with coverage gain





Conclusion

We have shown that compilation makes symbolic execution more efficient.

SymCC compiles symbolic-execution capabilities into binaries
Orders of magnitude faster than state of the art
Significantly more code coverage per time, 2 CVEs

Thank you!

sebastian.poeplau@eurecom.fr
aurelien.francillon@eurecom.fr

https://github.com/eurecom-s3/symcc
(code, docs, evaluation details)