An evaluation of Symbolic Execution Systems and the benefits of compilation with SymCC

……

Aurélien Francillon, Sebastian Poeplau
EURECOM, Sophia Antipolis, France

Aurélien Francillon

Professor at Eurecom, Graduate school on the French Riviera

- System security
- Embedded devices security
- Wireless security

Sebastian Poeplau, PhD Student

- Loves software engineering and building tools for developers!
- Worked at Lastline and Zalendo before PhD
- About to complete PhD, joining startup “Code Intelligence”

Why looking at Symbolic execution (performance)?

Symbolic execution is a middle ground between formal methods and traditional testing

- Can, in theory, provide complete coverage
- And therefore prove absence of (some) bugs (categories)?

Symbolic execution was proposed in:

- “Symbolic Execution and Program Testing“, J. King, CACM, 1976

Many tools exist KLEE, S2E, Angr, Triton, BinSec... with different goals and properties

Performance improved a lot with constraint solvers improvements in recent years

Combining Fuzzing with Symbolic Execution

This was first propose in the “Driller” paper

“Driller: Augmenting Fuzzing Through Selective Symbolic Execution”, NDSS 2016

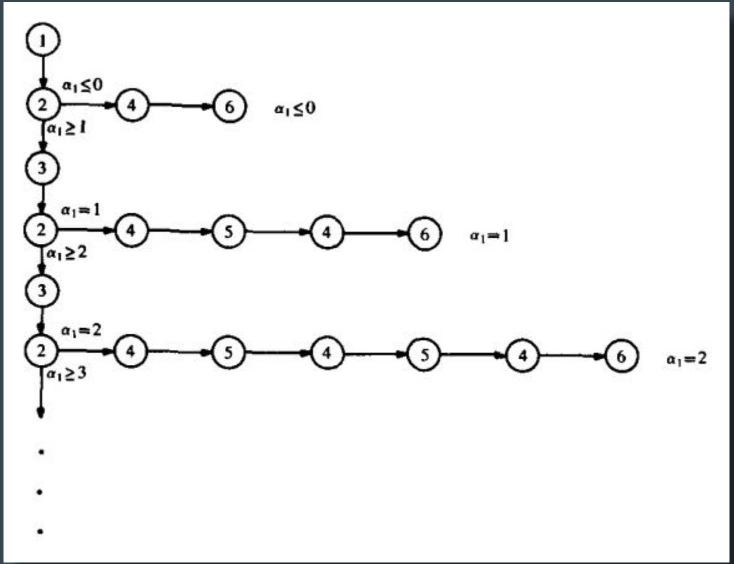
- Fuzzers are very fast but may miss some paths “if(X==0xDEADBEEF)”
- Symbolic execution alone is slow and gets stuck (state explosion, loops)

Combining Fuzzing and concolic execution: best of both worlds?

- Top 3 teams of the Cyber Grand Challenge used a combination of fuzzing and Symbolic Execution
- But performance problem with Symbolic execution engines...

Symbolic execution

- Trace computations in a program, building up symbolic formulas
- Solving symbolic expressions:
    - At branches to check if a branch is feasible
    - If a corruption (or fault) is detected, solve constraints and generate a test input



King: Symbolic Execution and Program Testing

Symbolic Execution

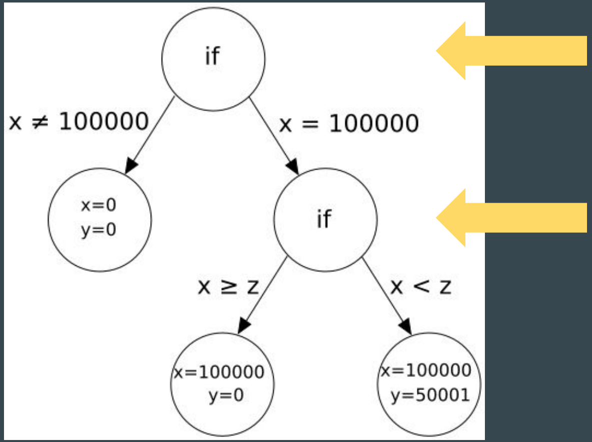
Explore programs by keeping track of computations in terms of inputs

When on one path only: “Concolic mode”

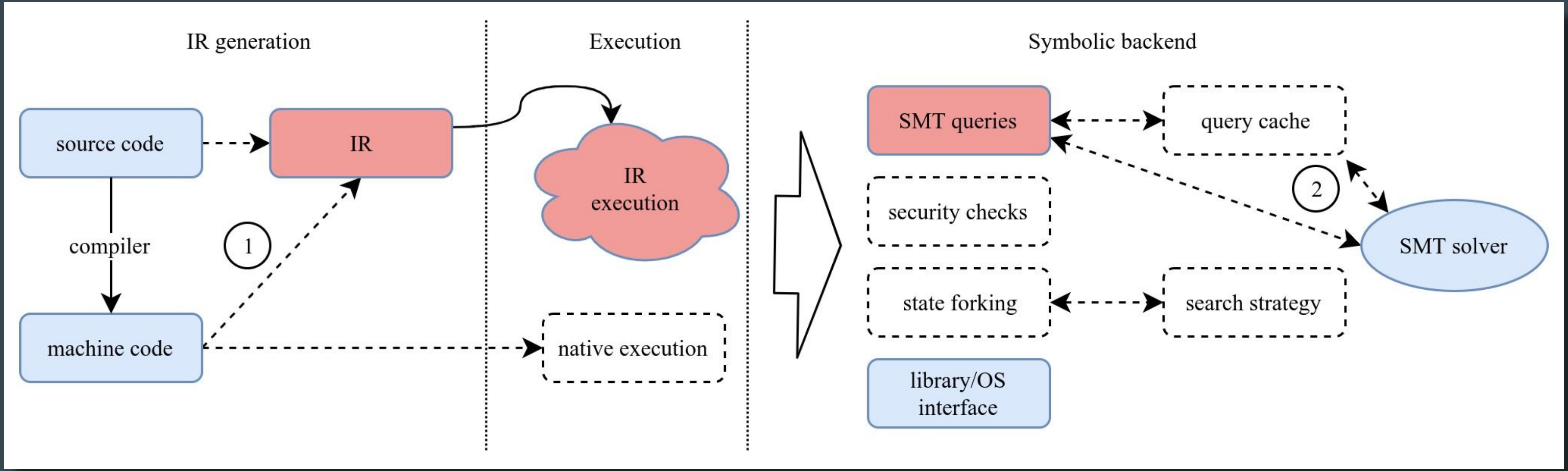
Target program

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

symbolic execution



Design space

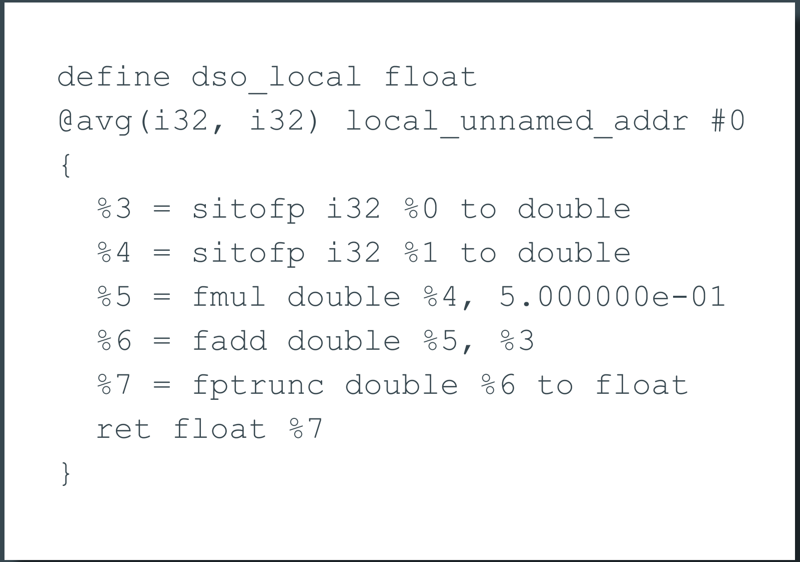


Previous work marked in the diagram:

① Kim et al.: Testing intermediate representations for binary analysis

② Palikareva and Cadar: Multi-solver support in symbolic execution
and Liu et al.: A comparative study of incremental constraint solving approaches in symbolic execution

Intermediate representation



```
define dso_local float
@avg(i32, i32) local_unnamed_addr #0
{
  %3 = sitofp i32 %0 to double
  %4 = sitofp i32 %1 to double
  %5 = fmul double %4, 5.000000e-01
  %6 = fadd double %5, %3
  %7 = fptrunc double %6 to float
  ret float %7
}
```

LLVM bitcode generated by Clang

- Abstract representation of a program
  - Often in static single assignment form (SSA)
  - Small instruction set
- Designed for different purposes
  - Compilers: LLVM bitcode
  - Dynamic instrumentation: VEX
  - Binary analysis: BIL, REIL
  - Many more; see Kim et al.: Testing Intermediate Representations for Binary Analysis

Research questions

- What is the impact of generating IR from source code or binaries?
- Is one IR more suitable than another? What about no IR?

Experiments

Code size

- How does IR generation impact code size?
- Estimate “information content” of IR

Execution speed

- How fast can we execute the IR?
- Crucial property according to Yun et al.

Query complexity

- How complex are the resulting SMT queries?
- Difficult queries slow down the analysis a lot

Implementations under analysis

KLEE	S2E	angr	Qsym
Source code to LLVM bitcode	Binary to LLVM bitcode via QEMU	Binary to VEX IR (Valgrind project)	No IR; execution of x86 machine code
Implemented in C++	Implemented in C/C++	Implemented in Python	Implemented in C++
No native execution	Binary translation through QEMU	Binary translation through Unicorn	Native execution via Intel Pin
	Based on KLEE		

Setup

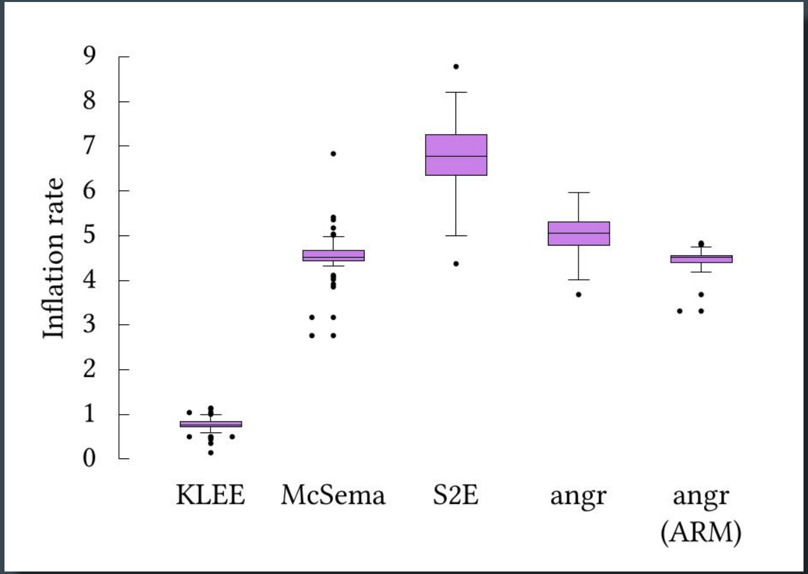
- Programs from DARPA Cyber Grand Challenge
  - Designed around a simple architecture (“DECREE”)
  - Source code available
  - Meant to be used as a test set for vulnerability detection (and exploit generation)
- Concolic execution
  - Follow the same fixed path in all engines
  - No bias from different exploration strategies
  - Path based on provided crashing inputs (“proofs of vulnerability”)
- Environment
  - Ubuntu 16.04, 24 GB of memory
  - 30 minutes per execution or solver run (whichever applies to the experiment)

Challenges

- We had to patch all engines
  - Add support for program particularities (e.g., support mmap in KLEE)
  - Insert measurement probes
- Still, only 24 out of 131 programs work in all four engines 😞
  - Unsupported instructions (e.g., floating-point arithmetic)
  - Excessive memory or CPU time consumption
  - Others concur: e.g., see Qu and Robinson, as well as Xu et al.
- Results are not fully representative of any possible program to test
  - But: scientific progress requires evaluation and comparison!
  - Need a methodology for comparing symbolic execution engines
  - We can still identify trends

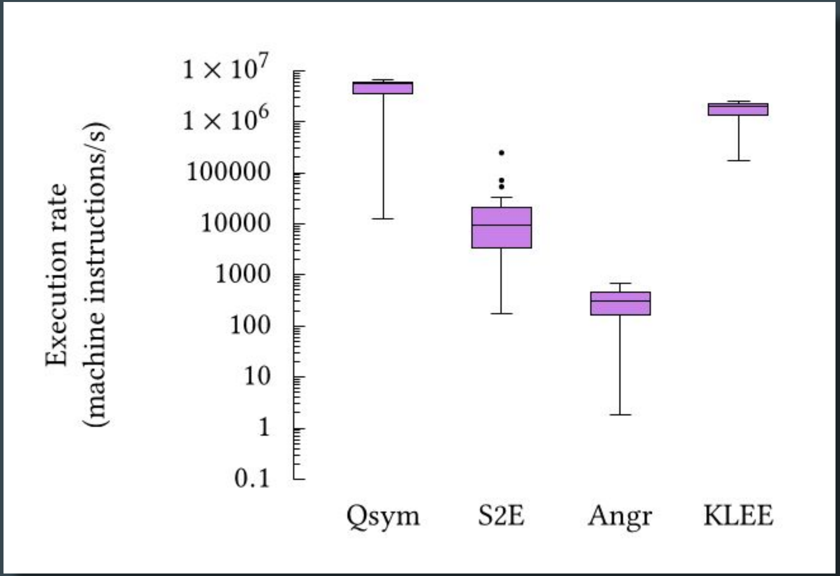
Results: Code size

- Measured *IR inflation rate*
  - Ratio between number of machine-code instructions and number of IR instructions
- Added two extra data points
  - McSema: lifter from binaries to LLVM bitcode
  - angr on ARM: apply angr's VEX translation to ARM machine code
- IR from source code is more concise
- S2E: problem with double translation?
  - Machine code → QEMU → LLVM bitcode



Inflation rate per IR generation mechanism
across 123 CGC programs and 106 coreutils binaries;
boxes contain 50% of the data points with the line marking the median, whiskers extend to 1.5 times the interquartile range, dots are outliers

Results: Execution speed



Execution speed of symbolically executed instructions across 24 CGC programs

- Measured *IR execution rate*
    - Symbolically executed instructions per unit of time
    - Normalized by average inflation rate
- Qsym unsurprisingly fastest
- angr: slow because of Python
- KLEE and S2E: same basis, but S2E executes less expressive IR
- Absence of IR seems beneficial

Example: Query complexity

Queries generated for the C expression

data[3] == 55

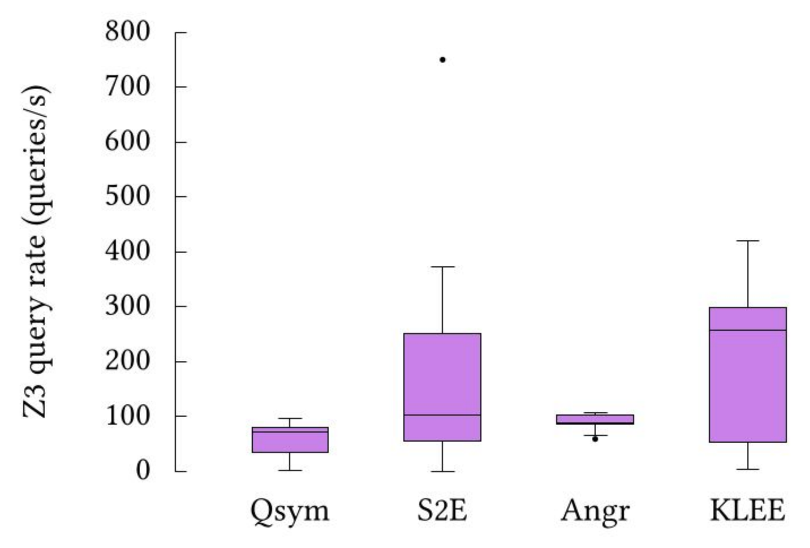
by KLEE (below) and S2E (right)

```
(= (_ bv55 8)
   ((_ extract 7 0)
    ((_ zero_extend 24)
     (select data (_ bv3 32)))))
```

```
(= (_ bv0 64)
   (bvand
    (bvadd
     ;; 0xFFFFFFFFFFFFFFC9
     (_ bv18446744073709551561 64)
     ((_ zero_extend 56)
      ((_ extract 7 0)
       (bvor
        (bvand
         ((_ zero_extend 56)
          (select data (_ bv3 32)))
         ;; 0x00000000000000FF
         (_ bv255 64))
        ;; 0xFFFF88000AFDC000
        (_ bv18446612132498620416 64))))))
    (_ bv255 64)))
```

Results: Query complexity

- Measured *query rate*
  - Number of solved queries per unit of time
- KLEE’s queries are simplest
  - Potentially because they are derived from high-level IR
- S2E gets close to KLEE
  - Internally based on KLEE
  - But different IR generation mechanism
- Is LLVM bitcode beneficial?



Query rates as a proxy for query complexity across across 23 CGC programs

Source vs binary

Research question 1

- Large impact on IR size, thus possibly on execution speed
- SMT queries derived from source are easier

Difference between IRs

Research question 2

- No observable difference between LLVM bitcode and VEX
- Fastest execution is achieved by using machine code directly

What did we find?

For easy queries, generate IR from source code.

For fast execution, work on machine code directly.

Limitations: small data set, effects of IR and IR generation are hard to isolate.

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

•••

Sebastian Poeplau, Aurélien Francillon

Distinguished paper award, Usenix Security 2020

EURECOM
Sophia Antipolis

Compiling symbolic-execution capabilities into executables

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*

Analysis process (QSYM)

attach via ptrace

Target process

Recap

We compile symbolic-execution capabilities right into the binary.

- Most others interpret
- QSYM uses dynamic binary instrumentation

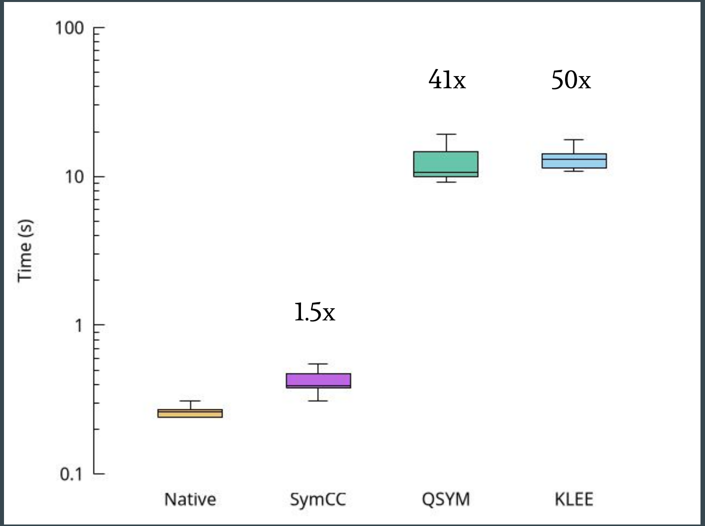
Evaluation

Benchmark and real-world targets

Benchmark: Execution Speed

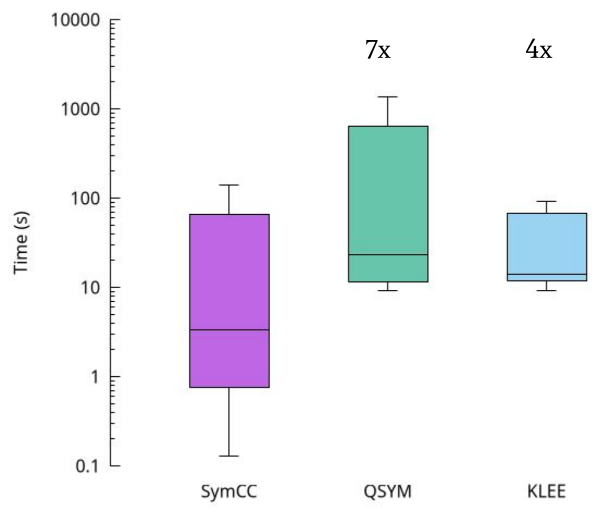
Fully concrete

No symbolic input provided



Concolic

Input data is made symbolic



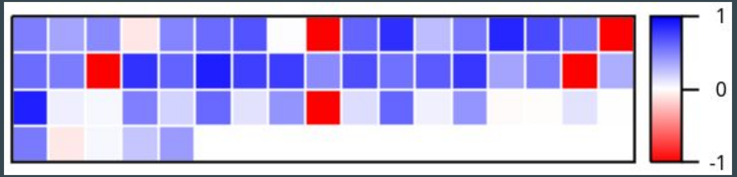
Benchmark: Coverage

Approach

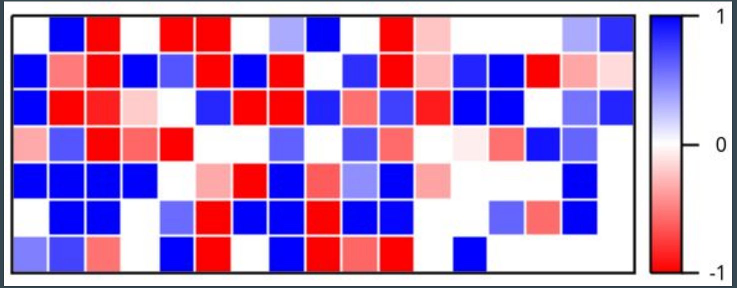
After concolic execution, measure edge coverage of newly generated inputs with afl-showmap.

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



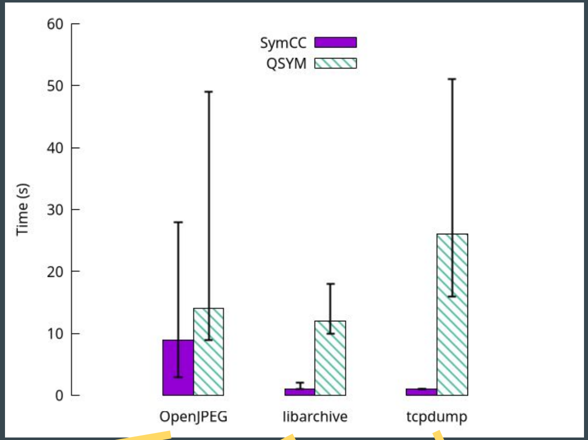
Comparison with QSYM (116 programs): SymCC is better on 47, worse on 40, and equal on 29

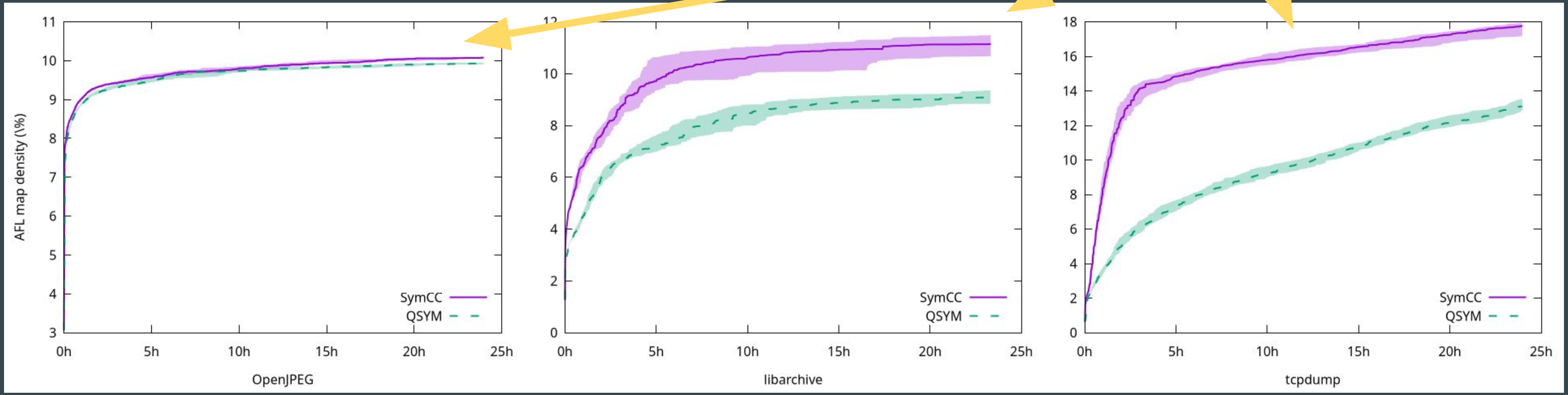
Real-world targets: Setup

- 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

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

Needs source code

- Often the case that source is available
- Binary code (libraries) just executed concretely

How to perform multipath exploration like Klee?

Thank you!

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

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