# Symbolic execution by compiling symbolic handling into binaries with SymCC and SymQEMU

…

2022 Frejus, meeting GDR/GT MFS

Aurélien Francillon

joint work with Sebastian Poeplau

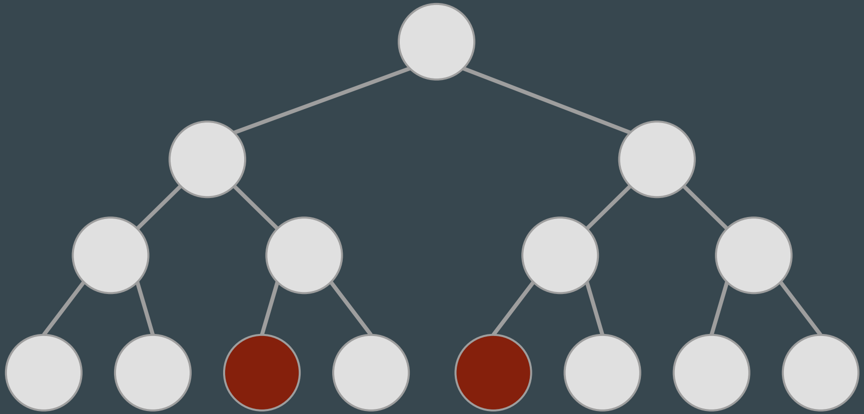
## Motivation

- Software is everywhere
- Software failures can cause lots of harm
    - Often annoying: can’t video-chat with parents, shipments delivered late
    - Sometimes severe: financial fraud, privacy infringement
    - Worst case, life-threatening: autonomous driving, airplane controls, medical equipment
- Need tools to find errors in software
    - Formal verification for critical software
    - (Knuth: “Beware of bugs in the above code; I have only proved it correct, not tried it.”)

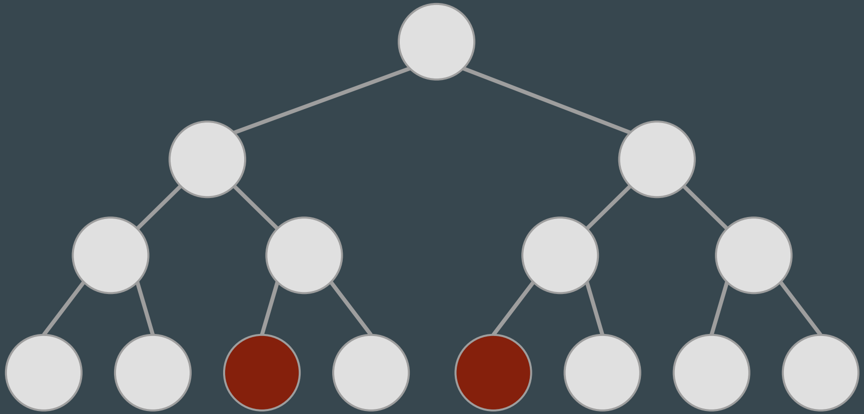
# Dynamic symbolic execution

Background

## Software testing

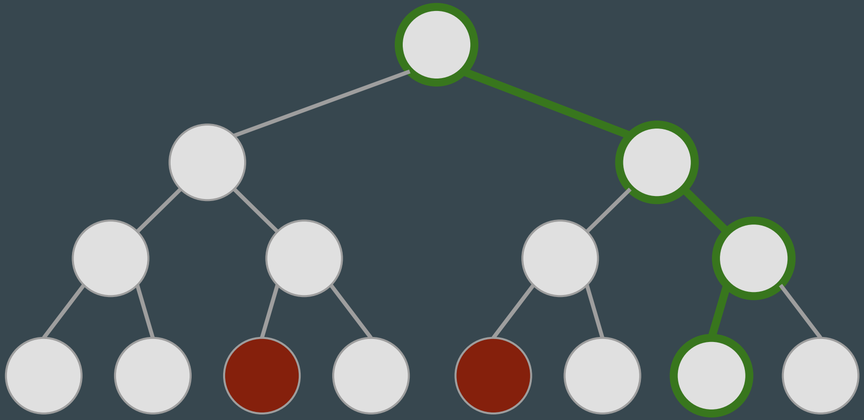


# Software testing



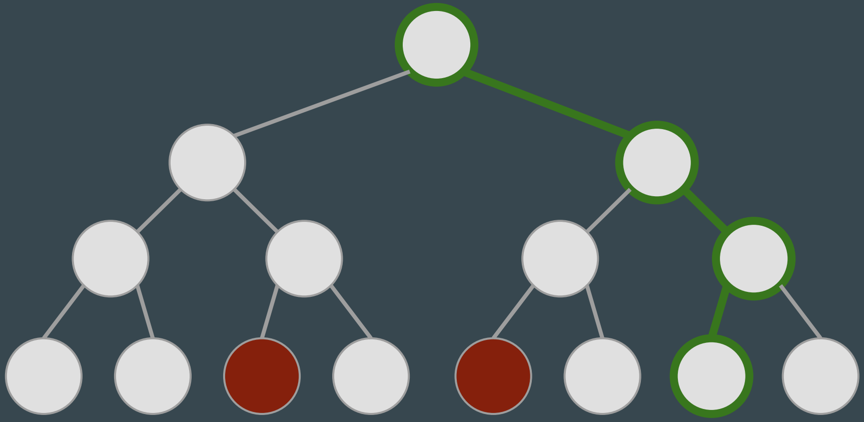
Example-based tests

## Software testing

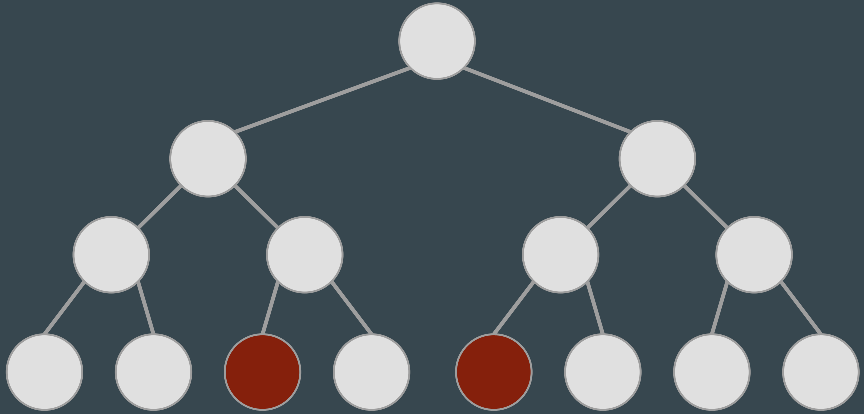


Example-based tests

# Software testing

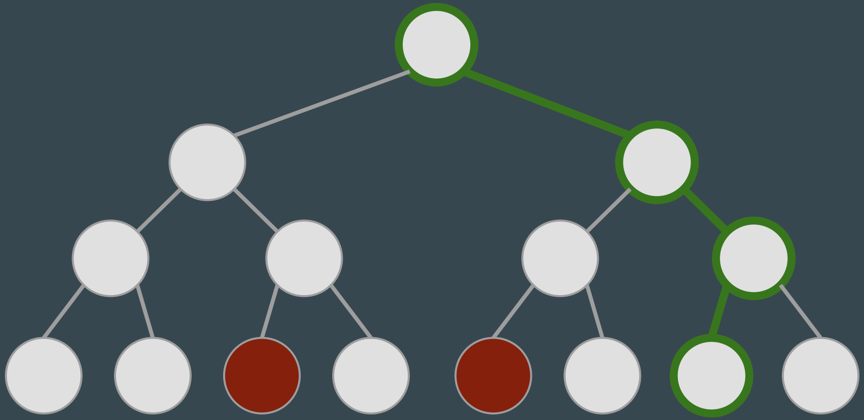


Example-based tests

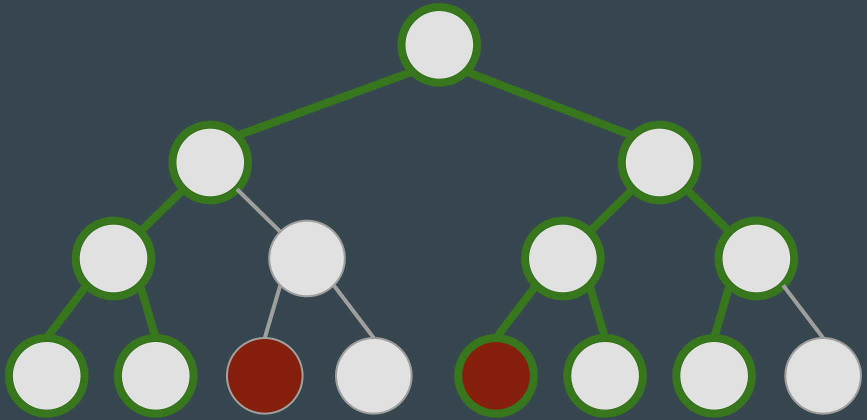


Fuzzing
(random inputs)

## Software testing

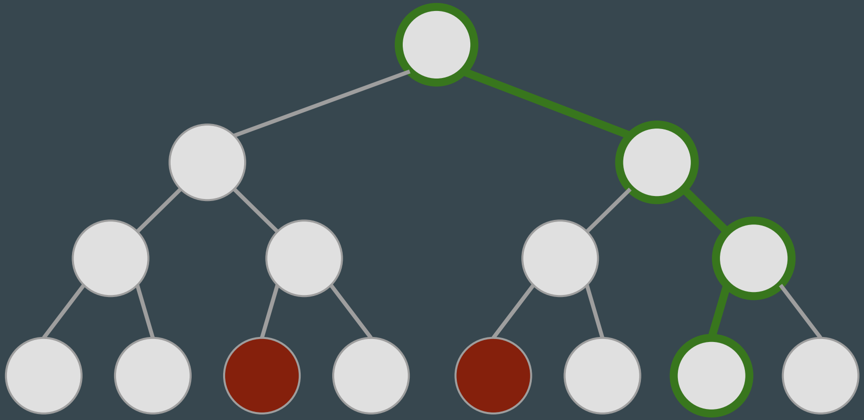


Example-based tests

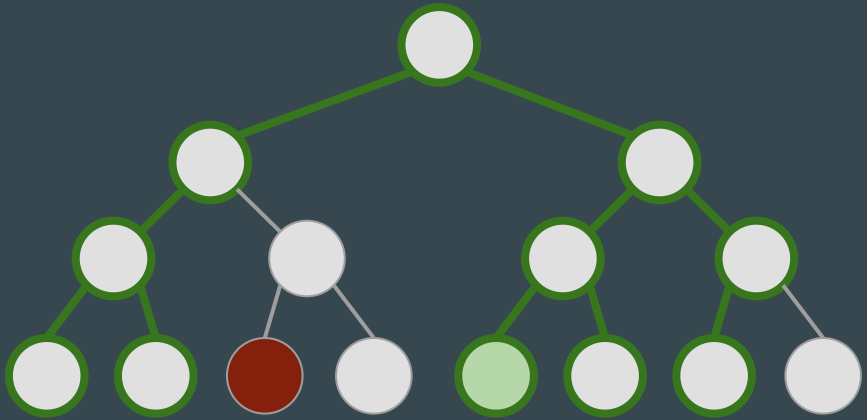


Fuzzing
(random inputs)

## Software testing



Example-based tests



Fuzzing
(random inputs)

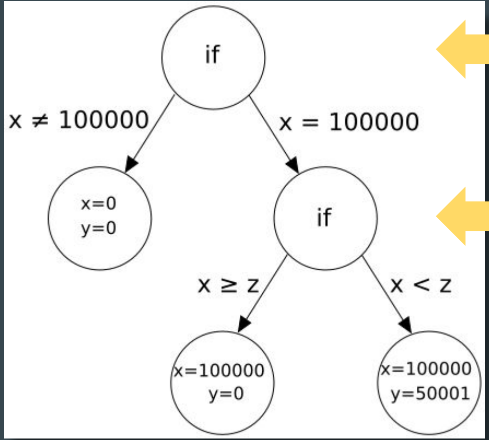
# Dynamic Symbolic Execution

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



# SMT solving

- Used for test case generation in symbolic execution
- “Satisfiability modulo theories”
  - → Can solve many different types of queries
  - → Based on heuristics for the Boolean satisfiability problem (SAT)
- TL;DR

  It’s expensive!

```
;; Integers x, y and z
(declare-const x (_ BitVec 32))
(declare-const y (_ BitVec 32))
(declare-const z (_ BitVec 32))

;; z == 2 * y
(assert (= z (bvmul y #x00000002)))

;; x == 100000
(assert (= x #x000186a0))

;; x < z
(assert (bvslt x z))

;; Solve!
(check-sat)
(get-model)
```

SMT query for one path in the example program

## Symbolic execution vs fuzzing

- Speed: fuzzers are faster
- Power: symbolic execution is more powerful
- Idea: combine the two!
    - → Often called “hybrid fuzzing”
    - → Successful research implementations: Driller [1], QSYM [2]
    - → <u>No need for correct symbolic execution !</u>
        - ■ Approximate results are still useful for fuzzing

[1] Stephens et al.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution, NDSS 2016
[2] Yun et al.: QSYM: A practical concolic execution engine tailored for hybrid fuzzing, USENIX 2018

## Research goal

Performance is a major challenge of symbolic execution.

Can we build a fast symbolic executor while staying flexible and conceptually simple?

## Design decisions in symbolic execution

- Where to start?
  - Source code contains useful information
  - In some cases, we may only have a binary
  - Often translated to intermediate representation
- What to do at branch points?
  - “Classic” approach: fork execution, follow both paths
  - Concolic execution: set a fixed input, follow its path, generate new inputs as we go

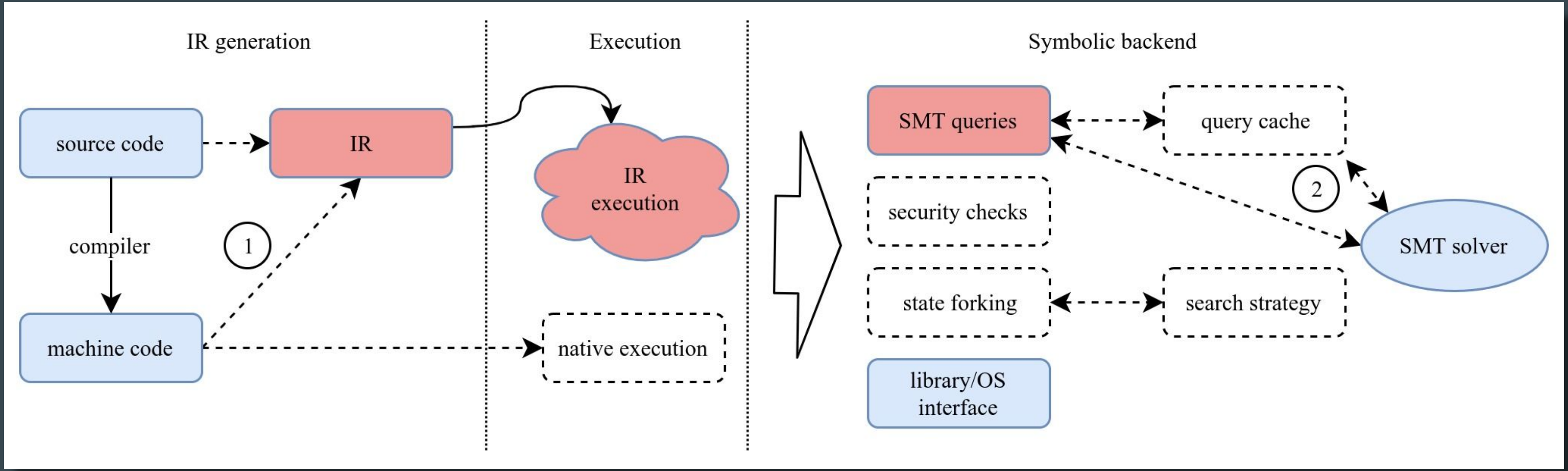
## Intermediate representation

```
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

# Design space



Previous work marked in the diagram:

① Kim et al.: Testing intermediate representations for binary analysis, ASE 2017

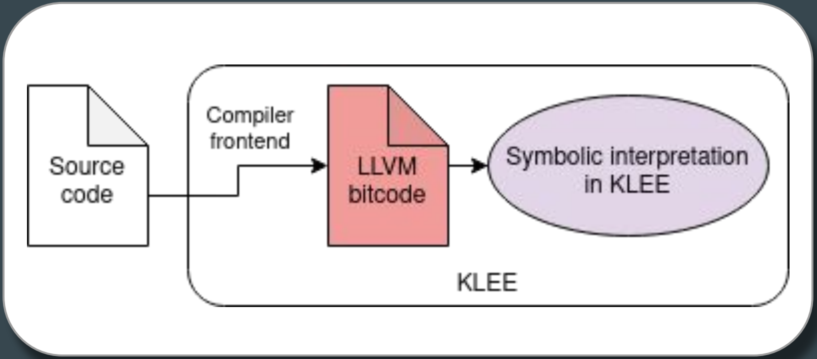
② Palikareva and Cadar: Multi-solver support in symbolic execution, CAV 2013 *and*
Liu et al.: A comparative study of incremental constraint solving approaches in symbolic execution, HVC 2014

## State of the art

- Where do existing implementations shine?
- What can we improve?

## KLEE

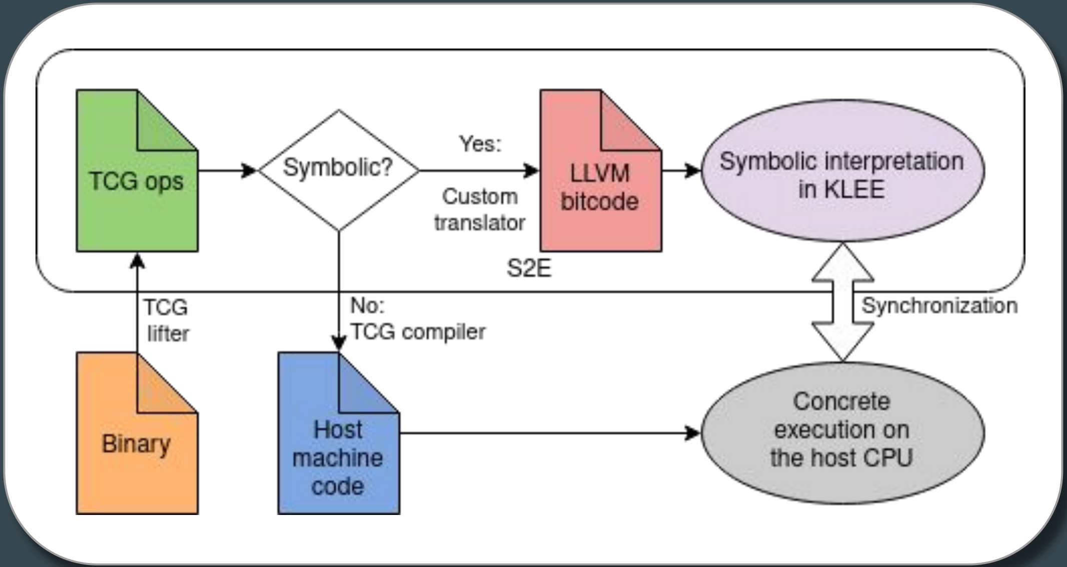
- One of the earliest “modern” implementations (Cadar et al., 2008)
- Compiler translates source code to LLVM bitcode
  - → Source code required
  - → Much simpler implementation than earlier systems because of IR
- Executes user-space programs



Cadar et al.:
KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs, OSDI 2008

## S2E

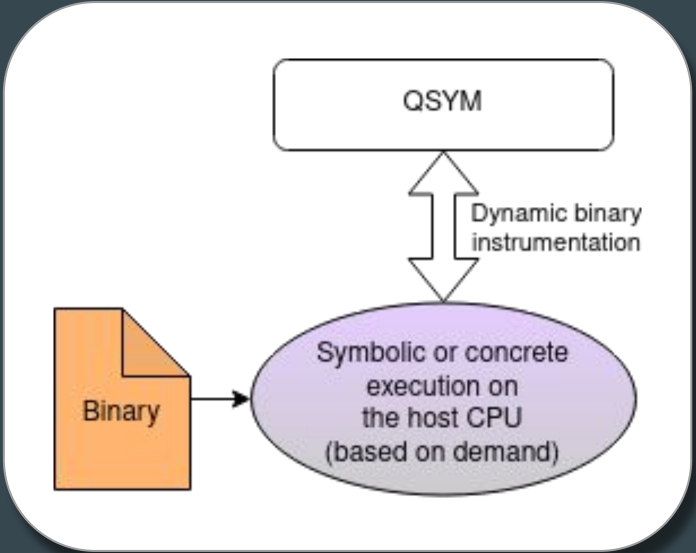
- Basic idea: QEMU + KLEE
    - QEMU’s TCG ops are lifted to LLVM bitcode
    - Bitcode is fed to KLEE
- Entire operating system inside
- Conceptually very flexible
    - Implemented for x86 only
- Highly complex



Chipounov et al.: Selective symbolic execution, HotDep 2009 *and*
The S2E platform: Design, implementation, and applications, ACM TOCS 2012

## QSYM

- Based on dynamic binary instrumentation
  - Intel Pin to insert symbolic handling at run time
  - Symbolic semantics at the x86 machine-code level
- High performance
- Supports binaries
- Conceptually simple (good!)
- Inflexible
  - Tied to the x86 instruction set
- Tedious implementation
  - Need to implement symbolic handling for *each x86 instruction*


QSYM
Dynamic binary instrumentation
Binary
Symbolic or concrete execution on the host CPU (based on demand)

Yun et al.: QSYM: A practical concolic execution engine tailored for hybrid fuzzing, USENIX 2018

## Popular implementations

| KLEE | S2E | QSYM | angr [1] |
|---|---|---|---|
| Source code to LLVM bitcode | Binary to LLVM bitcode via QEMU | No IR; execution of x86 machine code | Binary to VEX IR (Valgrind project) |
| Implemented in C++ | Implemented in C/C++ | Implemented in C++ | Implemented in Python |
| Forking | Forking | Concolic | Forking |
|  | Based on KLEE |  |  |

[1] Shoshitaishvili et al.: SoK: (State of) The art of war: Offensive techniques in binary analysis, S&P 2016

## Comparing implementations

Intermediate representation and its generation

## Research questions

- Does it matter whether we generate IR from source code or binaries? How?

- 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

## Setup

- Programs from DARPA Cyber Grand Challenge (CGC)
    - Designed around a simplified Linux (“DECREE”)
    - Source code available
    - Meant to be used as a test set for vulnerability detection (and exploit generation)

## Setup

- Programs from DARPA Cyber Grand Challenge (CGC)
  - Designed around a simplified Linux (“DECREE”)
  - Source code available
  - Meant to be used as a test set for vulnerability detection (and exploit generation)
- Concolic execution to avoid bias from different search strategies

## Setup

- Programs from DARPA Cyber Grand Challenge (CGC)
  - Designed around a simplified Linux (“DECREE”)
  - Source code available
  - Meant to be used as a test set for vulnerability detection (and exploit generation)
- Concolic execution to avoid bias from different search strategies
- Environment
  - Ubuntu 16.04
  - 24 GB of memory (mainly needed for angr)
  - 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

[1] Qu and Robinson: A case study of concolic testing tools and their limitations, ESEM 2011
[2] Xu et al.: Benchmarking the capability of symbolic execution tools with logic bombs, DSC 2018

## 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 [1, 2]

[1] Qu and Robinson: A case study of concolic testing tools and their limitations, ESEM 2011
[2] Xu et al.: Benchmarking the capability of symbolic execution tools with logic bombs, DSC 2018

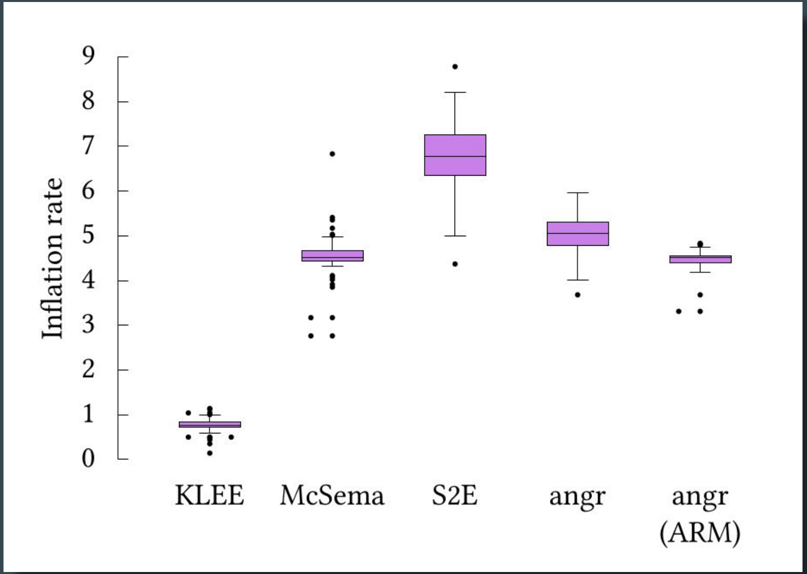
## Challenges

- 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 [1, 2]
- What's the significance?
    - Results are not representative for the set of all possible programs under test
    - We can still identify trends
    - But: scientific progress requires evaluation and comparison!
    - Need a methodology for comparing symbolic execution engines

[1] Qu and Robinson: A case study of concolic testing tools and their limitations, ESEM 2011
[2] Xu et al.: Benchmarking the capability of symbolic execution tools with logic bombs, DSC 2018

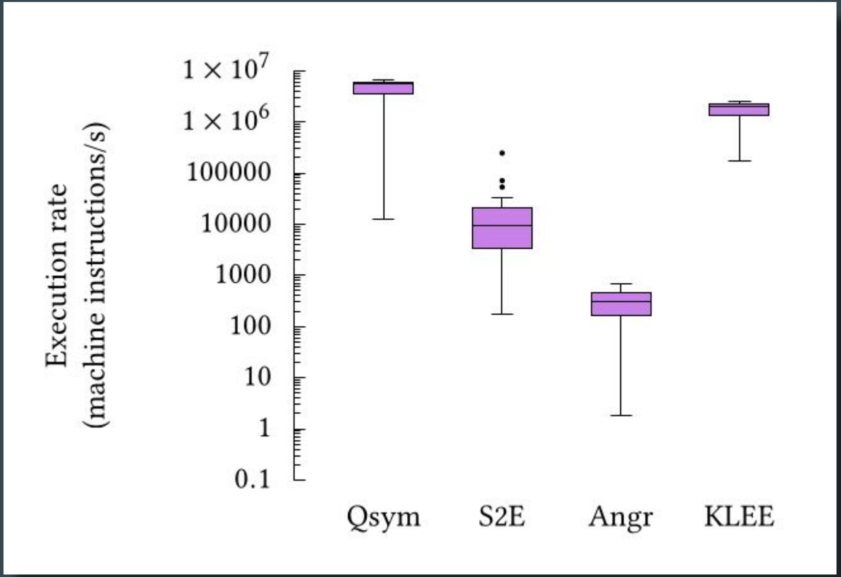
# Results: Code size

- 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
- Absence of IR seems beneficial

## Example: Query complexity

Queries generated for the C expression

data[3] == 55

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

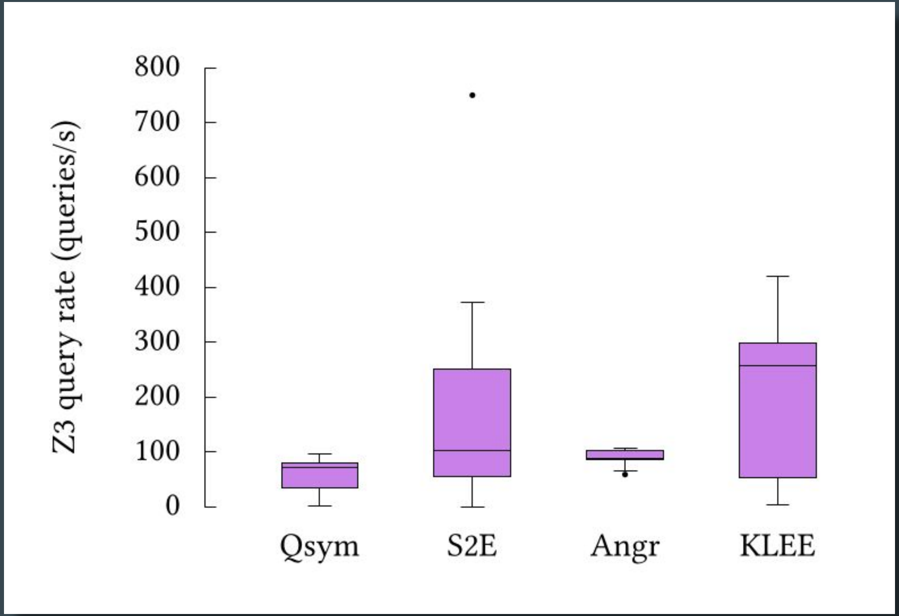
KLEE

```
(= (_ 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)))
```

S2E

# 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



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!

Poeplau and Francillon, USENIX 2020, Distinguished Paper Award

# 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: 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!)

## Remember the findings from the IR study?

For easy queries, generate IR from source code.

For fast execution, work on machine code directly.

## Evaluation

Two settings:

1. Benchmark experiments
2. Real-world targets

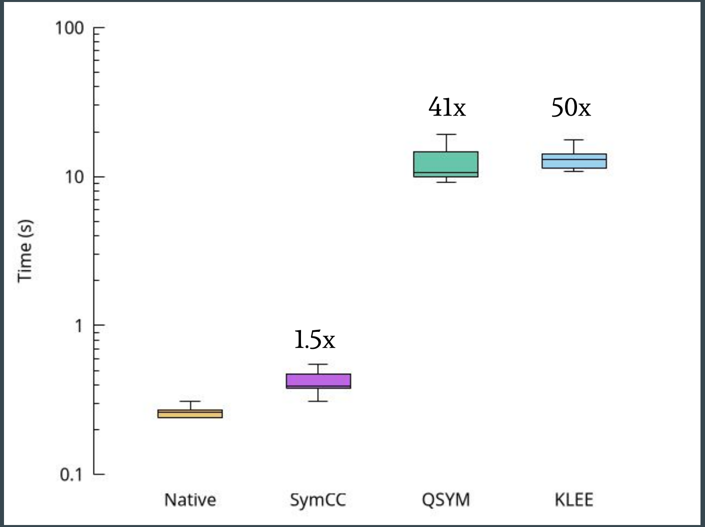
## Benchmark: Setup

- Goal: highly controlled environment
- Concolic execution on CGC programs
- 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

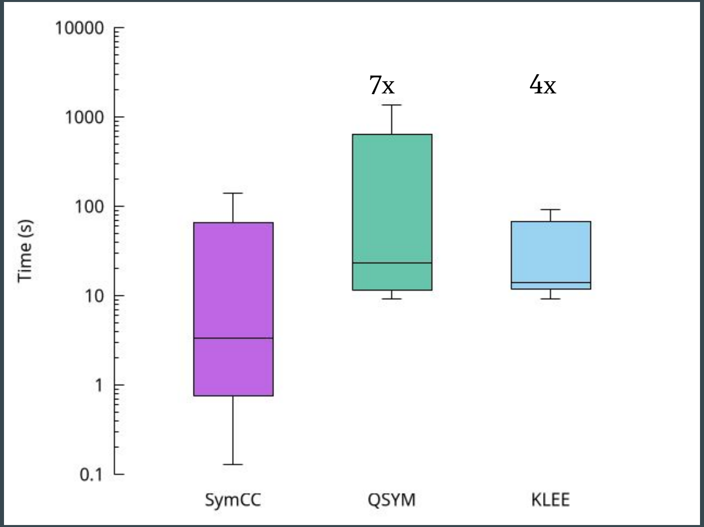
### Fully concrete

No symbolic input provided



### Concolic

Input data is made symbolic



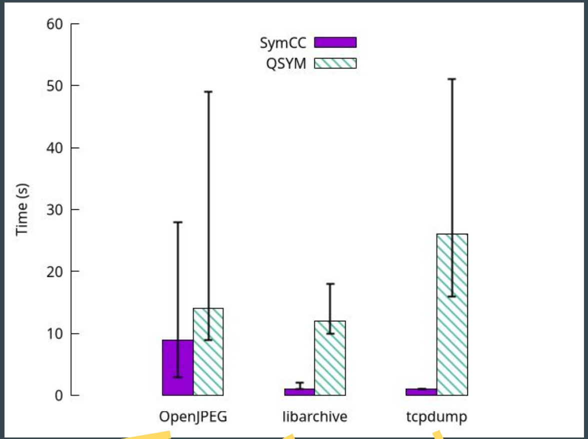
Coverage better than KLEE, similar to QSYM

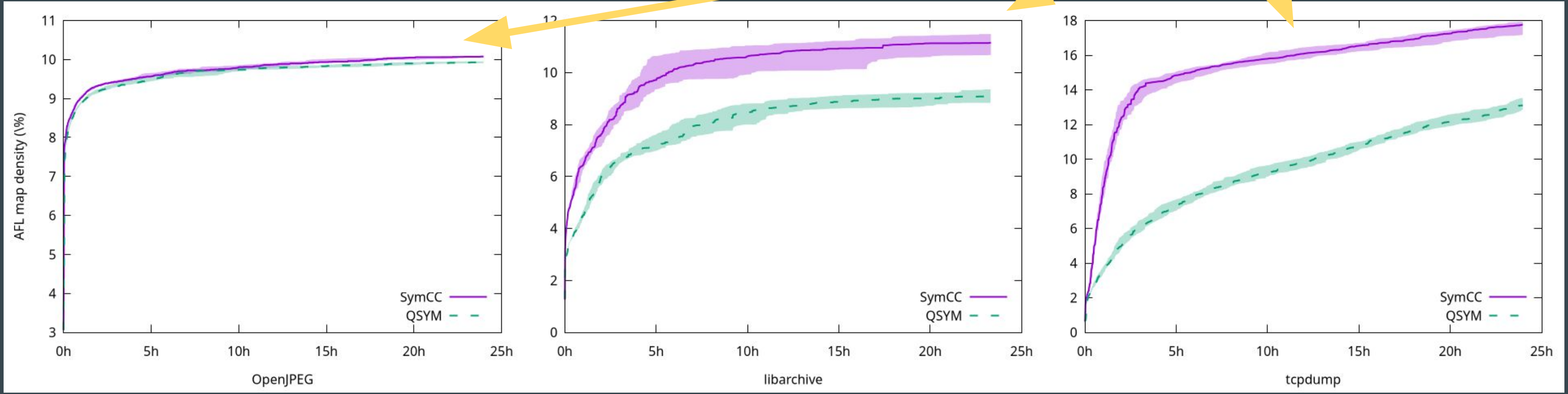
## Real-world targets: Setup

- Goal: show scalability to real-world software
- Popular open-source projects: OpenJPEG, libarchive, tcpdump
- Hybrid fuzzing: AFL and concolic execution with SymCC/QSYM
  - → 2 AFL processes, 1 SymCC/QSYM (like in QSYM's evaluation)
  - → Very simple integration strategy: run symbolic executor and AFL concurrently, exchange new inputs
- 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





## 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

## SymQEMU

Compilation-based symbolic execution for binaries

Poeplau and Francillon (NDSS 2021)

## Motivation

- SymCC is great, but it requires source code
- Why would you want to work without sources?
    - Proprietary dependencies
    - Security audits (e.g., firmware analysis)
    - Large projects with complex build systems, multiple source languages, etc.
- What’s wrong with existing solutions?
    - Low speed (e.g., angr)
    - Lack of flexibility (e.g., QSYM)
    - High complexity (e.g., S2E)

## Goals

- Speed!
    - Relatively uncontroversial…
- Architectural flexibility
    - Can't cross-compile without source code
    - Firmware analysis requires support for many CPU types
    - Analysis host may be different from target architecture
- Robustness
    - Don't want to write disassemblers ourselves
- Simplicity
    - Make a system that people can understand

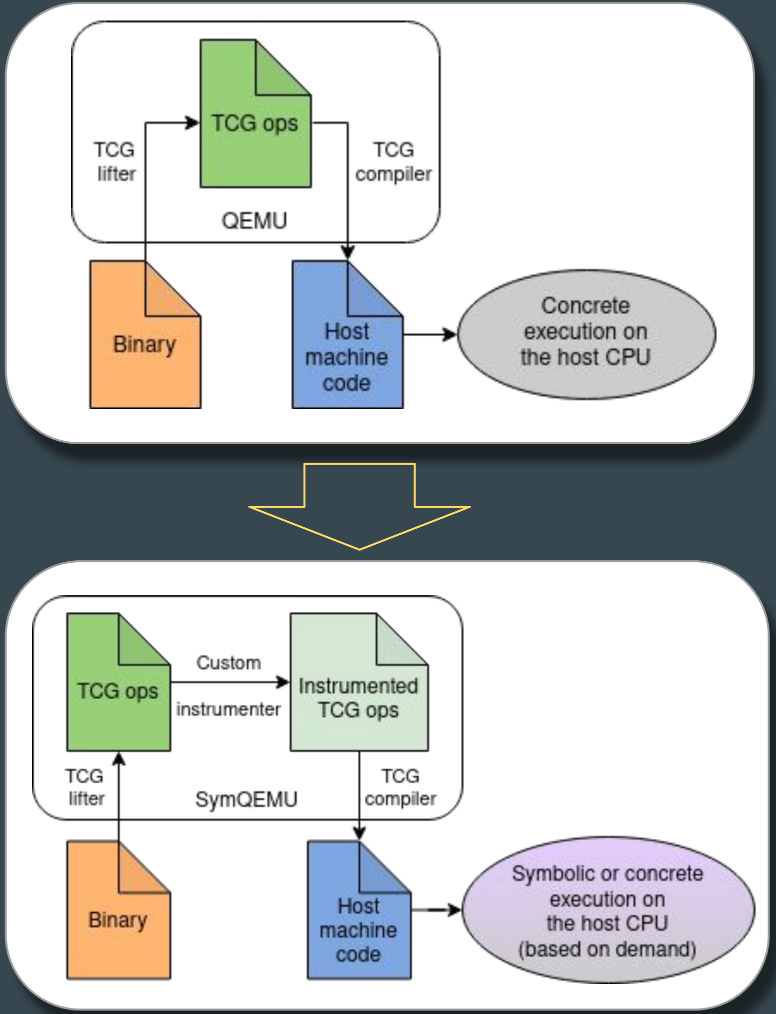
# SymQEMU

Design and implementation

- QEMU reliable and flexible
- Compilation-based symbolic execution is fast
- Approach: insert symbolic handling during binary translation

# SymQEMU: Implementation

- Modified QEMU
  - Insert symbolic handling during binary translation
  - Symbolic semantics at the level of TCG ops
- Simple implementation
  - Small instruction set
  - Backend reused from SymCC (i.e., QSYM)
- Flexibility
  - Inherited from QEMU
- High performance
  - To be demonstrated!



# Evaluation

Again, two settings:

1. Google FuzzBench
2. Whole-program analysis

## FuzzBench: Summary

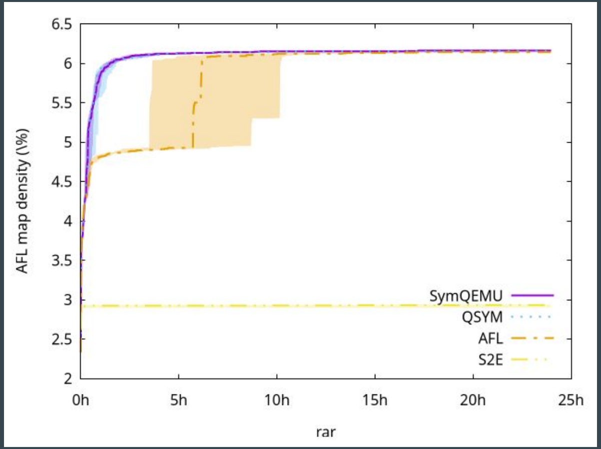
- Comparing SymQEMU with 12 fuzzers
- Hybrid fuzzing with AFL
- Second-highest score overall (without using source code)
- Outperformed all others on 3 out of 21 targets
- Better than pure AFL on 14 out of 21

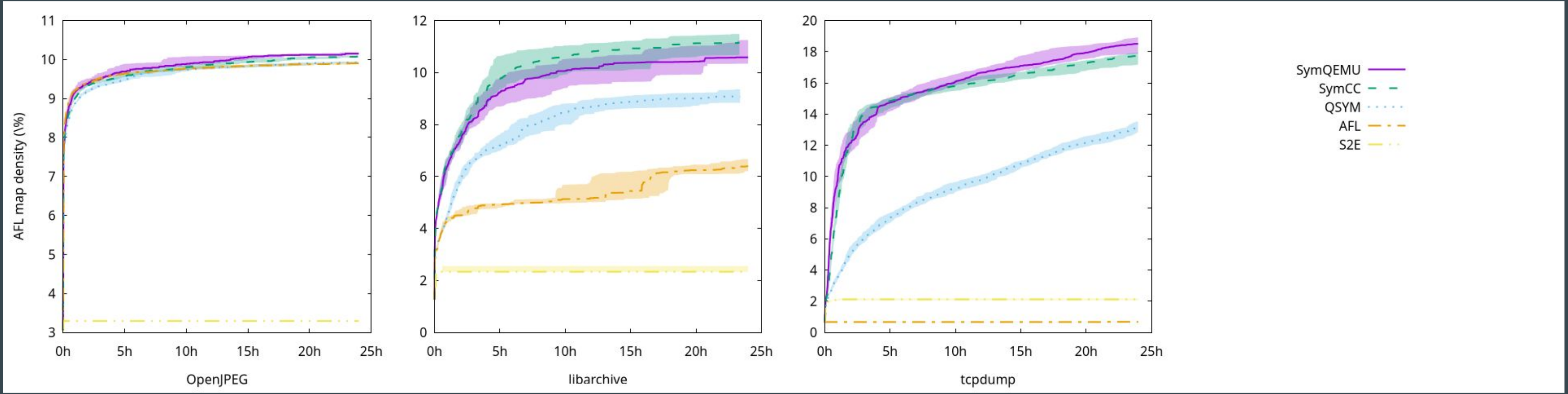
## Whole-program analysis: Setup

- Targets
  - Open source: OpenJPEG, libarchive, tcpdump (like SymCC evaluation)
  - Closed source: rar
- Systems under test
  - SymQEMU, QSYM, SymCC (open-source targets only): hybrid fuzzing with AFL
  - S2E: symbolic exploration with default search strategy
  - Pure AFL
  - 3 CPU cores for each configuration (like in SymCC evaluation)
- Intel Xeon Platinum 8260 CPU with 2GB of RAM *per core*
  - See the thesis for fineprint regarding S2E
- 24 hours, 30 iterations (~5 CPU core years)

# Whole-program analysis: Results

- SymQEMU significantly outperforms QSYM, S2E and pure AFL
- Performance comparable with SymCC (but without using source code)





## Compilation-based symbolic execution works on binaries as well.

SymQEMU inserts symbolic handling into binaries during dynamic binary translation
Significantly faster than state of the art, performance comparable with SymCC
Works on closed-source software

## Conclusion

# We have shown how to accelerate symbolic execution significantly.

Novel concept of compilation-based symbolic execution
Applicable to source code and binaries

Full results and software available online
(http://www.s3.eurecom.fr/tools/symbolic_execution/)

## Future work

Where to go next...

## Future work

- Execute higher-level abstractions for better reasoning capabilities (→ loop summaries [1], state merging [2])
- Solving
  - SMT solvers reason at the bit level - can we abstract? (→ formal-methods community)
  - Use incremental nature of solver queries
- Coordination
  - How best to combine symbolic execution with fuzzing? (→ PANGOLIN [3])

[1] Saxena et al.: Loop-Extended Symbolic Execution on Binary Programs, ISTA 2009
[2] Bucur et al.: Parallel symbolic execution for automated real-world software testing, EuroSys 2011
[3] Huang et al.: PANGOLIN: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction, S&P 2020

## Future work (continued)

- Exploration
  - Directed symbolic execution (→ KATCH [1], ParmeSan [2])
  - Data space vs control flow
    (→ MAGMA [3], “Measuring the coverage achieved by symbolic execution” by Cadar and Kapus)
- Evaluation
  - Code coverage (alone) may be the wrong metric
  - How to perform a fair comparison?
    (→ LAVA [4], MAGMA [3], “Evaluating fuzz testing” [5])

[1] Marinescu and Cadar: KATCH: High-coverage testing of software patches, FSE 2013
[2] Österlund et al.: ParmeSan: Sanitizer-guided graybox fuzzing, USENIX 2020
[3] Hazimeh et al.: MAGMA: A ground-truth fuzzing benchmark, 2020
[4] Dolan-Gavitt et al.: LAVA: Large-scale automated vulnerability addition, S&P 2016
[5] Klees et al.: Evaluating fuzz testing, CCS 2018

## Thank you

Questions?

Colloque gestion de crise et numérique : nouvelles menaces et nouvelles solutions

Jeudi 31 mars 2022 – de 9h00 à 16h15

Institut Mines-Télécom
19 place Marguerite Perey – 91120 Palaiseau

#colloqueimt

Colloque labellisé par

cnrs GDR Groupement de recherche Sécurité Informatique

## Additional material

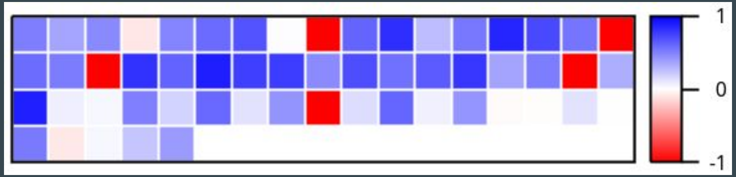
# SymCC benchmark: Coverage

## Approach

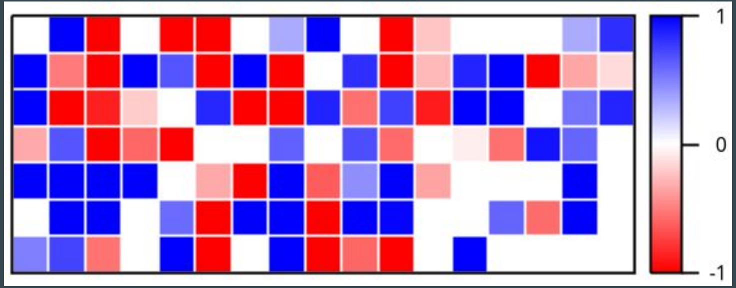
After concolic execution, measure edge coverage of newly generated inputs with afl-showmap.

## 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



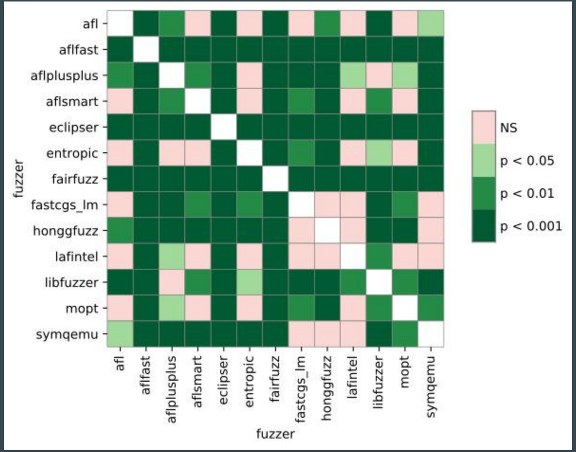
Comparison with QSYM (116 programs): SymCC is better on 47, worse on 40, and equal on 29

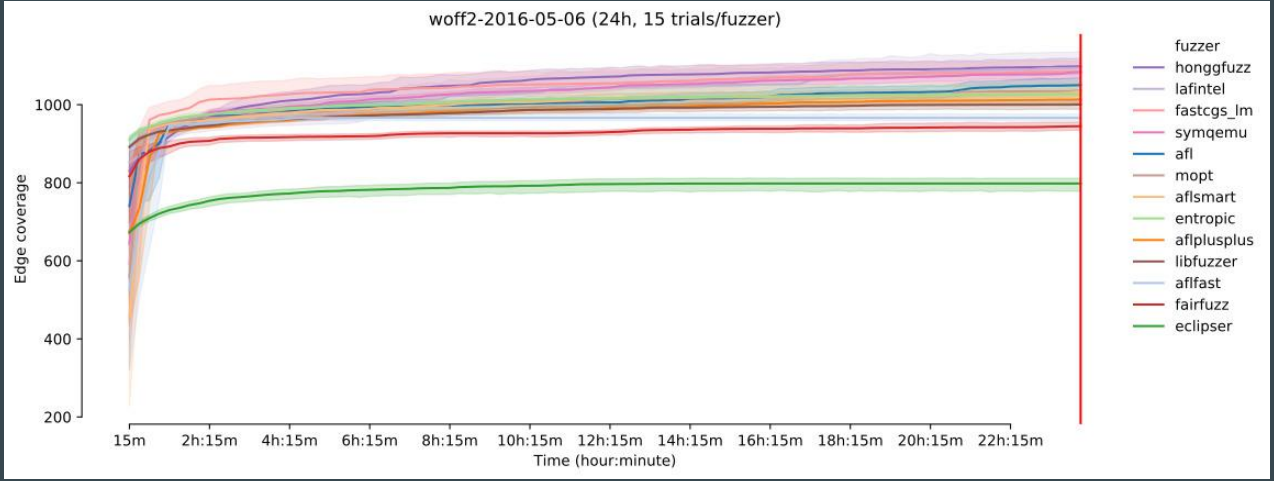
## FuzzBench: Setup

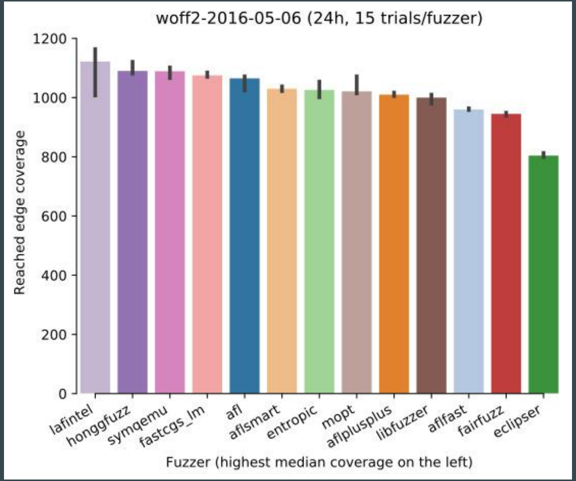
- Google FuzzBench: evaluation service for fuzzers
  - Tests fuzzers on open-source targets
  - 12 fuzzers, 21 targets, 24 hours, 15 iterations (~10 CPU core years)
  - Experiments performed by Google, resulting in a detailed report
- Integrating SymQEMU
  - Hybrid fuzzing with AFL (as before)
  - Packaged in a Docker image as required for FuzzBench

# FuzzBench: Results

- Second-highest score overall (without using source code)
- Outperformed all others on 3 out of 21 targets
- Better than pure AFL on 14 out of 21


afl
aflfast
aflplusplus
aflsmart
eclipser
entropic
fairfuzz
fastcgs_lm
honggfuzz
lafintel
libfuzzer
mopt
symqemu
fuzzer
NS
p < 0.05
p < 0.01
p < 0.001


woff2-2016-05-06 (24h, 15 trials/fuzzer)
Edge coverage
Time (hour:minute)
fuzzer
honggfuzz
lafintel
fastcgs_lm
symqemu
afl
mopt
aflsmart
entropic
aflplusplus
libfuzzer
aflfast
fairfuzz
eclipser


woff2-2016-05-06 (24h, 15 trials/fuzzer)
Reached edge coverage
Fuzzer (highest median coverage on the left)
lafintel
honggfuzz
symqemu
fastcgs_lm
afl
aflsmart
entropic
mopt
aflplusplus
libfuzzer
aflfast
fairfuzz
eclipser