Systematic comparison of symbolic execution systems

Intermediate representation and its generation

...

Sebastian Poeplau, Aurélien Francillon
EURECOM, Sophia Antipolis, France

(to appear at ACSAC 2019)

Agenda

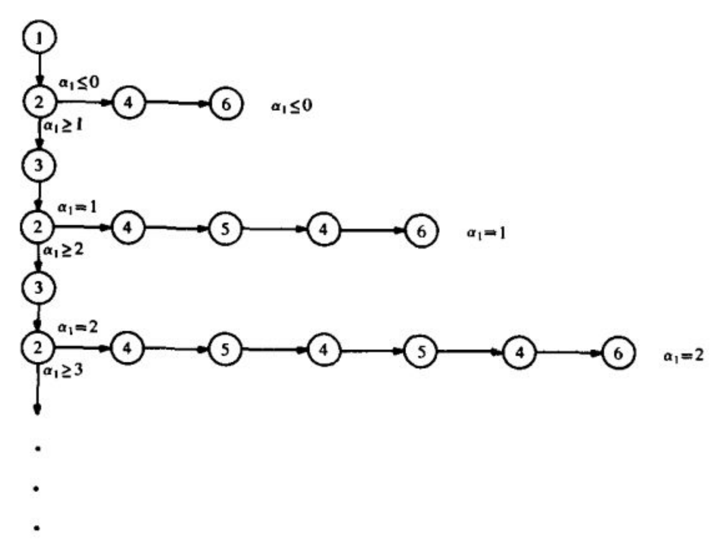
1. Background
2. Our study
   a. Systems under analysis
   b. Experimental setup
   c. Results
3. Discussion
4. Conclusion

Photo by Hope House Press - Leather Diary Studio on Unsplash

Background

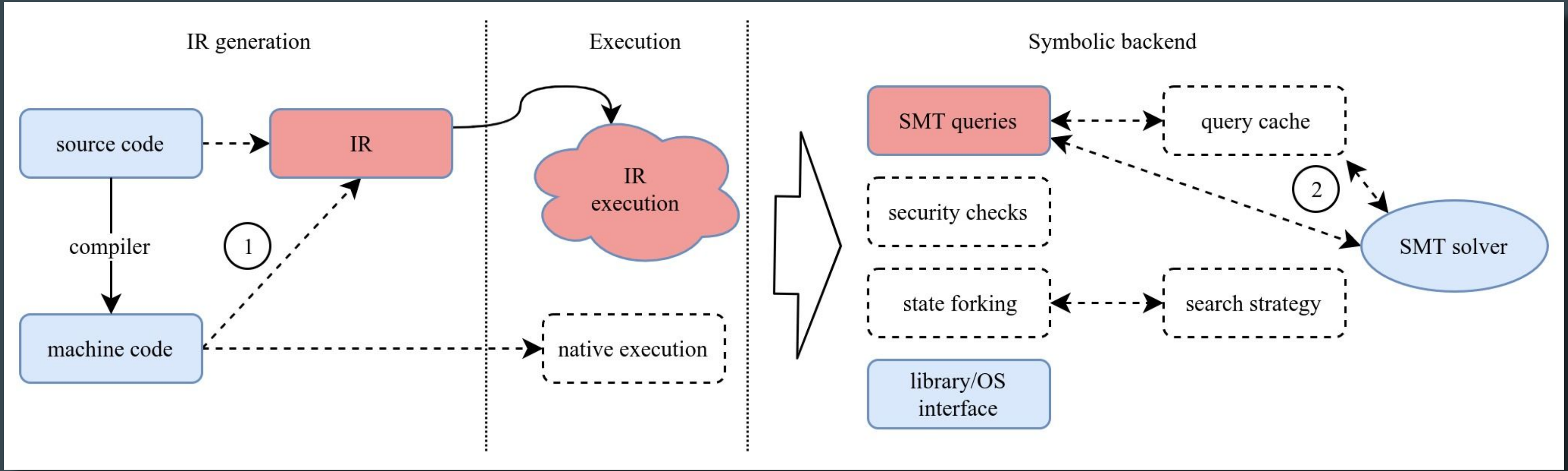
Symbolic execution

- Trace computations in a program, building up symbolic formulas
- At points of interest (e.g., branches), generate new inputs:
    - Substitute desired value into symbolic expression
    - Solve for the program input
- Many different implementations



King: Symbolic Execution and Program Testing

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

Our study

Intermediate representations are commonplace in symbolic execution.

But which one is best?

What is their impact in the first place?

We conducted a systematic study; work to be published at ACSAC 2019.

SMT solving

- “Satisfiability modulo theories”
  - SAT solver unites several theory solvers
  - Most interesting theory for us: bit vectors
  - Popular implementation: Z3 (MS Research)
- SAT: Boolean satisfiability problem
  - Known to be NP-complete
  - Good heuristics make many instances tractable
- Used for test case generation in symbolic execution

```
;; Integers x, k1 and k2
(declare-const x (_ BitVec 32))
(declare-const k1 (_ BitVec 32))
(declare-const k2 (_ BitVec 32))

;; ...all smaller than 50...
(assert (bvule x #x00000032))
(assert (bvule k1 #x00000032))
(assert (bvule k2 #x00000032))

;; ...and x is divisible by 6 and 7.
(assert (not (= x #x00000000)))
(assert (= x (bvmul k1 #x00000006)))
(assert (= x (bvmul k2 #x00000007)))

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

Example SMT query for Z3

Our study

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?

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		

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

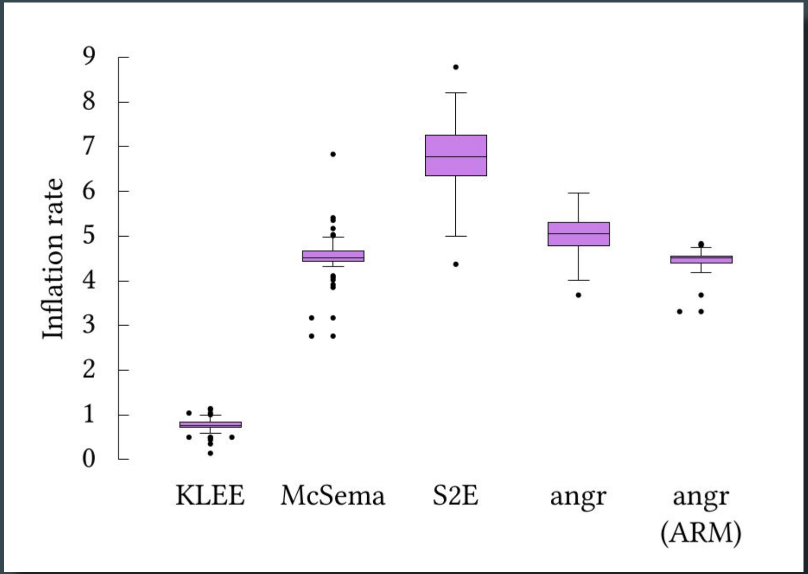
- Programs from DARPA Cyber Grand Challenge
  - Designed around a simple architecture ("DECREE")
  - Source code available
  - Meant to be used as a test set for vulnerability detection (and exploit generation)
- Concolic execution
  - Follow the same fixed path in all engines
  - No bias from different 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.
- Is there still value in our study?
  - Results are not representative for the set of all possible programs under test
  - But: scientific progress requires evaluation and comparison!
  - Need a methodology for comparing symbolic execution engines
  - We can still identify trends

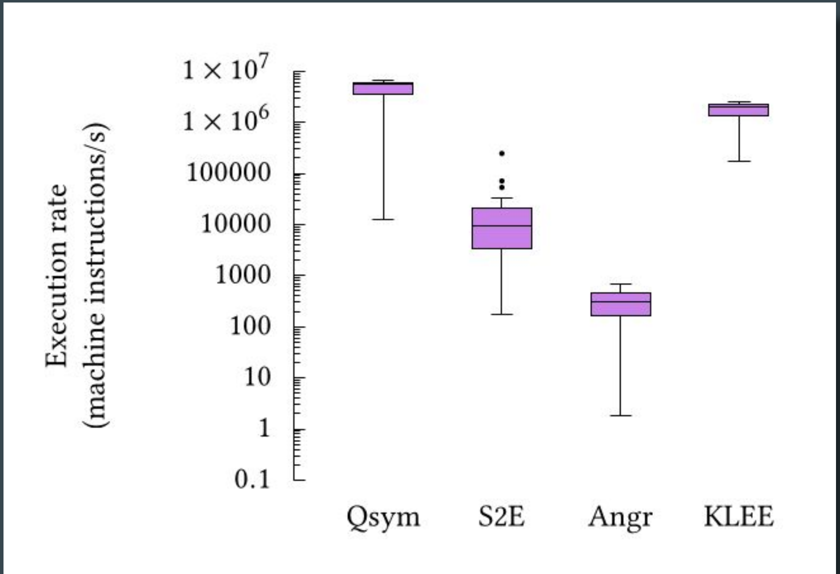
Results: Code size

- 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

stdin[3] == 55

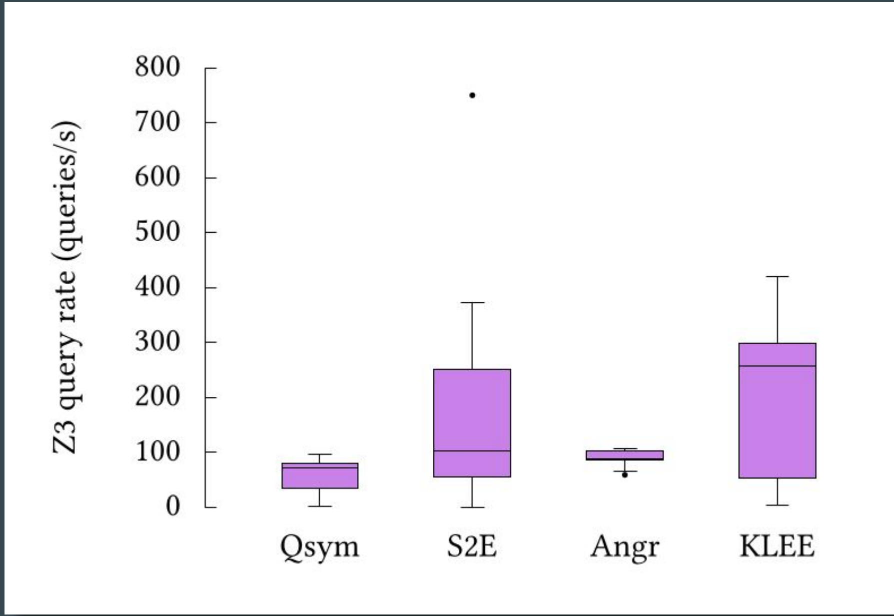
by KLEE (below) and S2E (right)

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

```
(=  (_ bv0 64)
   (bvand
    (bvadd
     ;; 0xFFFFFFFFFFFFFFC9
     (_ bv18446744073709551561 64)
     ((_ zero_extend 56)
      ((_ extract 7 0)
       (bvor
        (bvand
         ((_ zero_extend 56)
          (select stdin (_ 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

Discussion

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

Remark: Implementation language

- Independent of the choice of IR, but with a large impact on the overall result
- Implementation language influences the possible use cases
  - Python makes angr flexible for scripting and interactive exploration but is too slow for batch processing
  - C++ enables Qsym, KLEE and S2E to execute fast but limits extensibility
- Other factors, e.g., development speed, maintainability

Conclusion

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.

What's next?

- Assess the *quality* of generated test cases, not just the speed of generation
    - Interesting properties: effect on code coverage, similarity to existing test cases, directedness
- Find out what makes queries hard for SMT solvers
    - Some operations known to be tough (e.g., division of bit vectors)
    - Effect of compiler optimizations?
    - Goal: produce "solver-friendly" queries
- End-to-end comparison of symbolic execution engines
    - Compare from input to output, i.e., from program under test to discovered bugs
    - Many sources of bias
    - Large differences in implementation

Thank you!

{sebastian.poeplau, aurelien.francillon}@eurecom.fr

http://www.s3.eurecom.fr/tools/symbolic_execution/