Constraint-Based Testing for Floating-Point Code: Challenges and Opportunities

Cristian Cadar, Daniel Schemmel
Software Testing

• Software has bugs

• Many different bug finding tools exist
  – Regression suites, fuzzing, verification, static analysis, compiler sanitizers, …

• This talk: Tools using SMT constraints
  – Example: Symbolic Execution

• Floating point arithmetic is finicky…
Very Quick: Symbolic Execution

```c
if (x * y == 42) {
    printf("Welcome!\n");
} else {
    abort();
}
```
Floating Point SMT Theory

- The obvious solution: Just use a floating point SMT theory!
  - E.g.: QF_ABVFP instead of QF_ABV

- Mapping from program to constraints similar as for bitvectors

- We implemented this approach for KLEE

- It is very slow
Floating Point SMT Theory Performance Experiment

- Three theories
  - Integers
  - Bitvectors equivalent to int64_t
  - Floating Point Numbers equivalent to double

- Three constraints for each theory
  - $x + y = z$
  - $x \cdot x = y$
  - $x \neq y$
Benchmark 1: ./simple.py --mode int
  Time (mean ± σ): 146.2 ms ± 4.7 ms  [User: 124.3 ms, System: 21.1 ms]
  Range (min ... max): 140.2 ms ... 155.5 ms  19 runs

Benchmark 2: ./simple.py --mode bv64
  Time (mean ± σ): 189.8 ms ± 5.7 ms  [User: 161.2 ms, System: 27.7 ms]
  Range (min ... max): 183.6 ms ... 204.4 ms  14 runs

Benchmark 3: ./simple.py --mode fp64
  Time (mean ± σ): 714.7 ms ± 7.0 ms  [User: 676.5 ms, System: 35.7 ms]
  Range (min ... max): 707.2 ms ... 730.0 ms  10 runs
Floating Point SMT Theory Performance Experiment

- Let's add just one more condition...
  - $x + y = z$
  - $x \cdot x = y$
  - $x \neq y$
  - $x > 1$
Floating Point SMT Theory Performance Experiment

**Benchmark 1:** ./still-simple.py --mode int

- **Time (mean ± σ):** 146.9 ms ± 3.6 ms
- **Range (min ... max):** 139.8 ms ... 153.9 ms

[User: 125.5 ms, System: 20.5 ms] 19 runs

**Benchmark 2:** ./still-simple.py --mode bv64

- **Time (mean ± σ):** 197.6 ms ± 6.5 ms
- **Range (min ... max):** 190.4 ms ... 210.3 ms

[User: 165.2 ms, System: 31.6 ms] 14 runs

**Benchmark 3:** ./still-simple.py --mode fp64

- **Time (mean ± σ):** 2.084 s ± 0.029 s
- **Range (min ... max):** 2.044 s ... 2.128 s

[User: 2.038 s, System: 0.039 s] 10 runs
Approximate Solutions

• Any technique is incomplete or imprecise for non-trivial programs
  - Symbolic execution, fuzzing: Path explosion
  - Model checking: State explosion
  - Static analysis: Imprecise (has false positives)
  - Verification: Writing proofs infeasible for arbitrary programs

• Maybe an approximate solution is good enough?
Fixed Point Approximation of Floating Point Numbers

• The floating point theory can be lowered to the bitvector theory
  − Softfloat libraries do the same thing in the concrete world

• Structurally simpler queries are usually easier to solve

• Use a simplified lowering not exactly capturing IEEE 754 semantics

• Many programs don’t really use the full range of a double anyway…

• Can we just use a fixed point number instead?
## Fixed Point Performance Experiment

**Benchmark 1**: ./still-simple.py --mode int
- Time (mean ± σ): 146.9 ms ± 3.6 ms
- Range (min ... max): 139.8 ms ... 153.9 ms
- [User: 125.5 ms, System: 20.5 ms]
- 19 runs

**Benchmark 2**: ./still-simple.py --mode bv64
- Time (mean ± σ): 197.6 ms ± 6.5 ms
- Range (min ... max): 190.4 ms ... 210.3 ms
- [User: 165.2 ms, System: 31.6 ms]
- 14 runs

**Benchmark 3**: ./still-simple.py --mode fp64
- Time (mean ± σ): 2.084 s ± 0.029 s
- Range (min ... max): 2.044 s ... 2.128 s
- [User: 2.038 s, System: 0.039 s]
- 10 runs

**Benchmark 4**: ./still-simple.py --mode fix64
- Time (mean ± σ): 504.7 ms ± 5.9 ms
- Range (min ... max): 498.2 ms ... 513.7 ms
- [User: 459.2 ms, System: 43.7 ms]
- 10 runs
Fixed Point Approximation of Floating Point Numbers

• Advantages:
  - Can grant massive speedups

• Challenges:
  - Simplifying too far gives bogus results (basically degrades to a slow fuzzer)
  - Applicability in the real world

• Ongoing work – input appreciated!
  - How many bits for fixed point representation?
  - Overflow or wraparound behavior?
Summary

- Floating point arithmetic is a problem for constraint-based approaches
- SMT solving is very slow
- Fuzzing can quickly generate satisfying assignments
- Using simpler, approximate number representations can potentially speed up analysis at the cost of precision