Compiler Testing

Cristian Cadar

Slides by Alastair Donaldson

Software Reliability Course
Autumn 2016

Importance of compiler reliability

We rely on reliable compilers for:

- Day-to-day programming
- Source code analysis
- IR-level analysis

Are source- or IR-based verification techniques meaningful if the **compiler misbehaves**?

Compilers are complex pieces of software and contain bugs

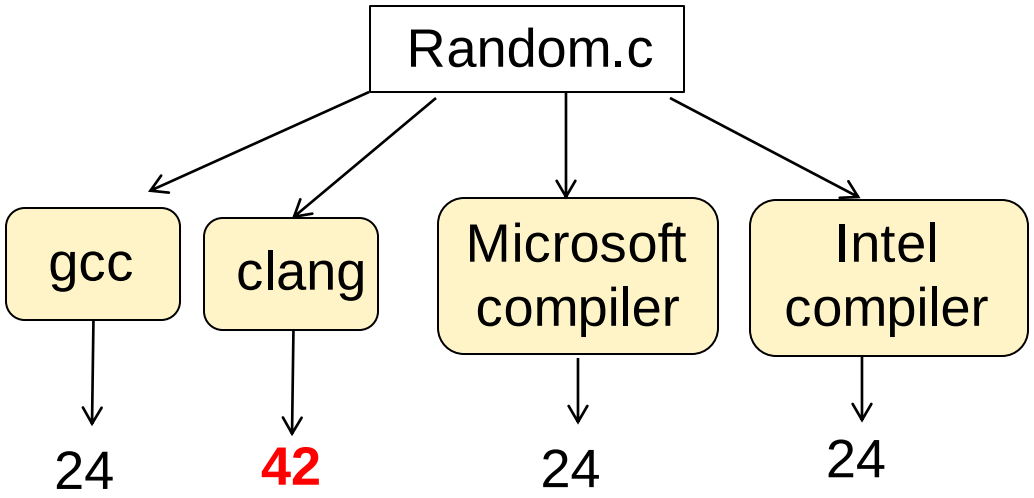
Let's look at two methods for testing them

Random differential testing

Generate random programs

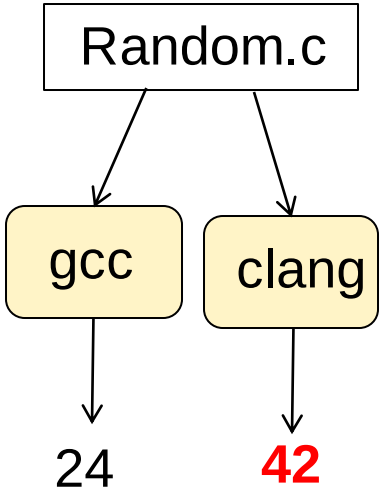
Try them with many compilers

Result mismatches indicate bugs



Pioneered by **Csmith**, University of Utah (PLDI'11)

Compiler testing and undefined behaviour



The mismatch is not erroneous if Random.c exercises **undefined behaviour**

If an execution exercises undefined behaviour, the entire execution has no meaning

Any result is acceptable

Compiler testing and undefined behaviour

Random differential testing requires programs that are **free from undefined behaviour**

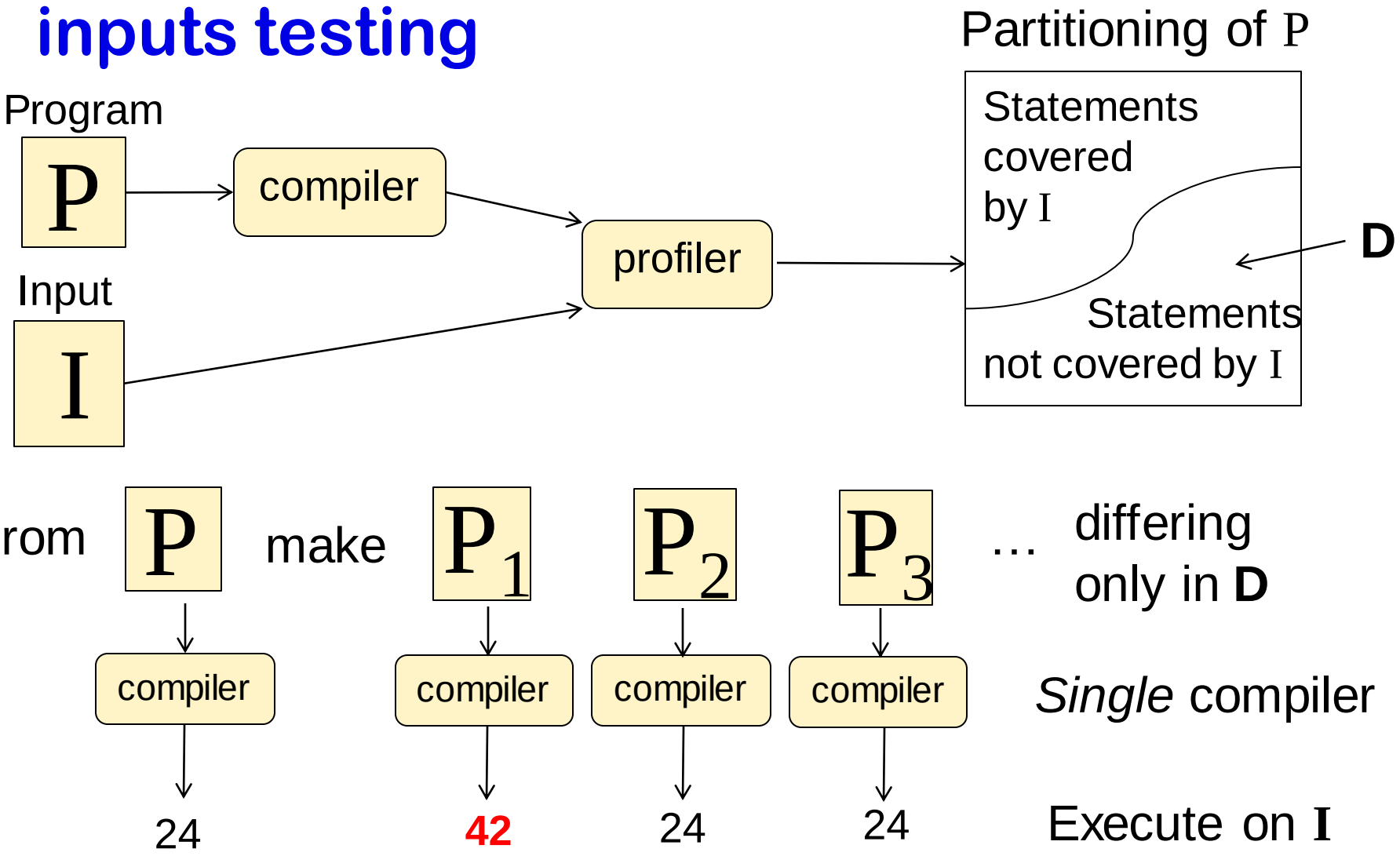
Csmith aims to guarantee this via careful generation, and "safe math" macros

E.g., instead of generating: `e1/e2` (`e1`, `e2` unsigned) generate `safe_div(e1, e2)` defined as follows:

```
#define safe_div(e1, e2) \
      ( (e2) == 0 ? (e1) : (e1) / (e2) )
```

Equivalence modulo inputs testing

Le et al. PLDI'14



Mismatches indicate **bugs**

An example of **metamorphic** testing

An open problem in compiler testing: floating-point

Optimisations in the presence of floating-point operations can change the results of programs

Compilers typically run in one of three modes:

- **Strict:** allow no such optimisations
- **Excess precision:** allow optimisations that increase the precision of operations
- **“Fast math”:** allow optimisations that assume algebraic laws that do not hold for floating-point

Excess precision example

Computing `a + b * c` in 32-bit floating-point usually involves:

- Computing `b * c`, rounding result to 32-bit value `t`
- Computing `a + t`, rounding result to 32-bit value

Many architectures support fused multiply-add instruction, `fma(a, b, c)`, which computes `a + b * c` in one step, with a **single** rounding

This can lead to **faster** and **more precise** computation

Transforming `a + b * c` to `fma(a, b, c)` can be a useful compiler optimisation

But it may change what the program computes!

“Fast math” example

Floating-point multiplication is not associative:
`(a*b)*c == a*(b*c)` does **not** hold in general

Other algebraic identities do not hold, e.g. we can have
`x + y == x` without `y` being 0

“Fast math” mode: compiler pretends floating-point operators **do** satisfy algebraic laws of real numbers

E.g., the compiler might optimise

```
x = x*x*x*x*x*x*x*x;
```

to

```
x *= x; x *= x; x *= x;
```

Faster, but for some values of `x` changes the result

Desirable in domains such as gaming where bit-precise results are not important

OpenCV Results

- Crosschecked 51 SIMD-optimized versions against their reference scalar implementations
  - Found mismatches in 10
- Most mismatches due to tricky FP-related issues:
  - Precision
  - Rounding
  - Associativity
  - Distributivity
  - NaN values

[EuroSys 2011]

OpenCV Results

Surprising find: min/max not commutative nor associative!

min(a,b) = a < b ? a : b

a < b (ordered) → always returns false if one of the operands is NaN

min(NaN, 5) = 5
min(5, NaN) = NaN

min(min(5, NaN), 100) = min(NaN, 100) = 100
min(5, min(NaN, 100)) = min(5, 100) = 5

Why are floating-point optimisations hard to test?

Random differential testing and EMI testing require checking whether results are **identical**

How can we distinguish between result differences due to:

- a compiler bug

vs.

- excess precision or “fast math” optimisations?

This is the subject of on-going work

Summary

Compilers bugs undermine program analysis efforts

Random differential testing and EMI testing are successful methods for testing compilers

Compiler testing is hard in the presence of floating-point