Comparison between Csmith and YarpGen

Molly Chen

xiaoou@iscas.ac.cn

2021.1.13

What is it ?

- Csmith and YarpGen
  - Compiler testing tool
  - Randomized test-case generation tools, for finding compiler bugs.
  - Developed by John Regehr et al.

    https://www.cs.utah.edu/~regehr/
  - Csmith (2011) only for C compiler
  - YarpGen (2020) for C and C++ compiler.
  - Predecessor: Randprog (1600 lines long, 2007)

Testing Method

- Random testing and differential testing

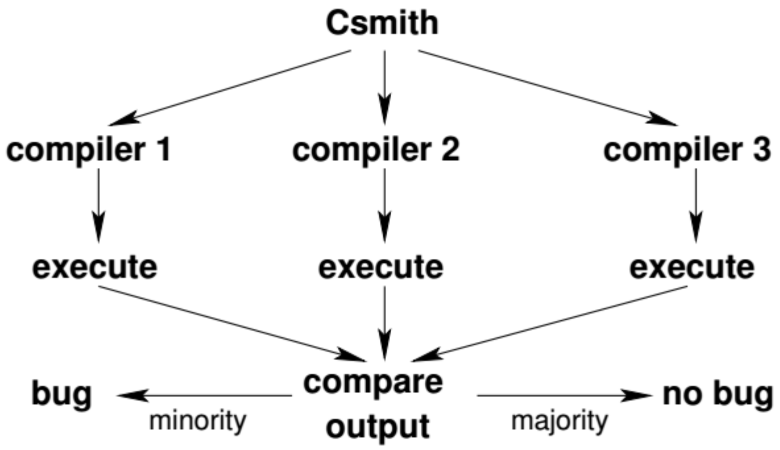


Figure 2. Finding bugs in three compilers using randomized differential testing

Picture from thesis *Finding and Understanding Bugs in C Compilers*

Outcome

- Csmith: report 325 bugs （25 GCC P1 bug）
- YarpGen: report more than 220 bugs

Design Goals

- Csmith:
  - be well formed and have a single meaning according to the C standard.
  - maximize expressiveness.

    (support many language features and combinations of features)

- YarpGen:
  - to avoid, or at least delay, saturation.

    (diversity, expressiveness, target specific optimization pass)

Advantage

- Csmith:
  - cover a large subset of C, avoiding the **undefined and unspecified behaviors**. (to ensuring single interpretation.)
  - C features supporting: complex control flow, data structures, pointers, arrays, structs.

    (cost: analysis and dynamic checks increase Csmith code size. 40k lines)

- YarpGen:
  - generating programs without using **dynamic checks**.
  - Using **generation policies** to target different parts of an optimizer.
  - Automated tools for compiler fuzzing. (Harness)
  - Do not support function call.

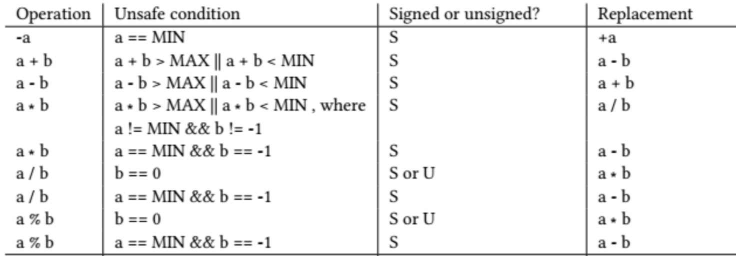
Randomly Generating Programs

- Csmith:
  - governed by a grammar for a subset of C.
  - maintains: a global environment, a local environment (for safety check)
  - top-down recursive generation with 6 steps.

- YarpGen:
  - Top-down recursive generation.
  - No function call. Top level is main block.
  - Lowering IR to the target language.

How to avoid Undefined and Unspecified Behaviors (UBs)

- Csmith:
  - Easy UBs: can be avoided structurally by generating programs in such a way that problems never arise.
  - Hard UBs: using static analysis and adding run-time checks to the generated code.
    - Static: to avoid use variables without initialization, initialize variables close to where they are declared.
    - Run-time check: null pointer checks.
- YarpGen:
  - easy UBs: same with Csmith
  - Hard UBs:
    - It interleaves analysis and code generation. While generating, convert operations that trigger undefined behavior into similar operations that are safe.

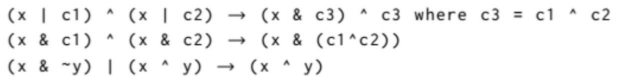


Operation	Unsafe condition	Signed or unsigned?	Replacement
-a	a == MIN	S	+a
a + b	a + b > MAX || a + b < MIN	S	a - b
a - b	a - b > MAX || a - b < MIN	S	a + b
a ⋆ b	a ⋆ b > MAX || a ⋆ b < MIN , where a != MIN && b != -1	S	a / b
a ⋆ b	a == MIN && b == -1	S	a - b
a / b	b == 0	S or U	a ⋆ b
a / b	a == MIN && b == -1	S	a - b
a % b	b == 0	S or U	a ⋆ b
a % b	a == MIN && b == -1	S	a - b

Generation Policies

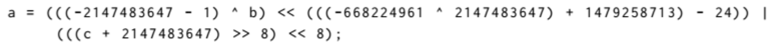
- Csmith: don’t have

- YarpGen:
    - Arithmetic, logical, and bitwise contexts



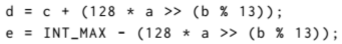
```
(x | c1) ^ (x | c2) → (x & c3) ^ c3 where c3 = c1 ^ c2
(x & c1) ^ (x & c2) → (x & (c1^c2))
(x & ~y) | (x ^ y) → (x ^ y)
```

    - Policies for constants



```
a = (((-2147483647 - 1) ^ b) << (((-668224961 ^ 2147483647) + 1479258713) - 24)) |
     (((c + 2147483647) >> 8) << 8);
```

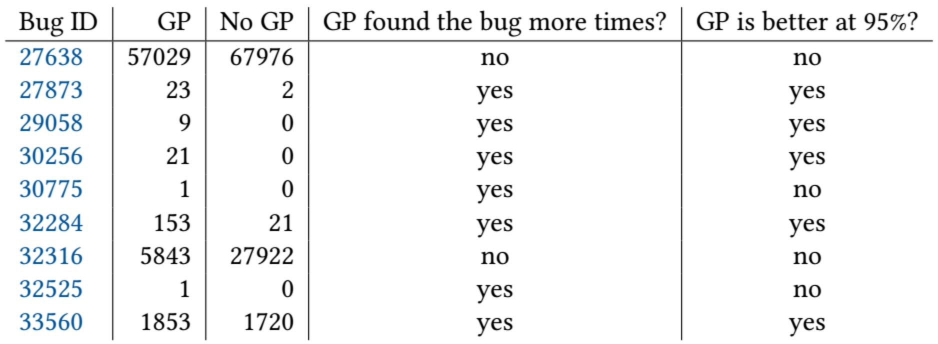
    - Common subexpression buffer



```
d = c + (128 * a >> (b % 13));
e = INT_MAX - (128 * a >> (b % 13));
```

    - Parameter shuffling

    - **Evaluating the impact of generation policies.**



Bug ID	GP	No GP	GP found the bug more times?	GP is better at 95%?
27638	57029	67976	no	no
27873	23	2	yes	yes
29058	9	0	yes	yes
30256	21	0	yes	yes
30775	1	0	yes	no
32284	153	21	yes	yes
32316	5843	27922	no	no
32525	1	0	yes	no
33560	1853	1720	yes	yes

Thank you