CSMITHEDGE: More Effective Compiler Testing by Handling Undefined Behaviour Less Conservatively

Empirical Software Engineering 27, 129 (2022)

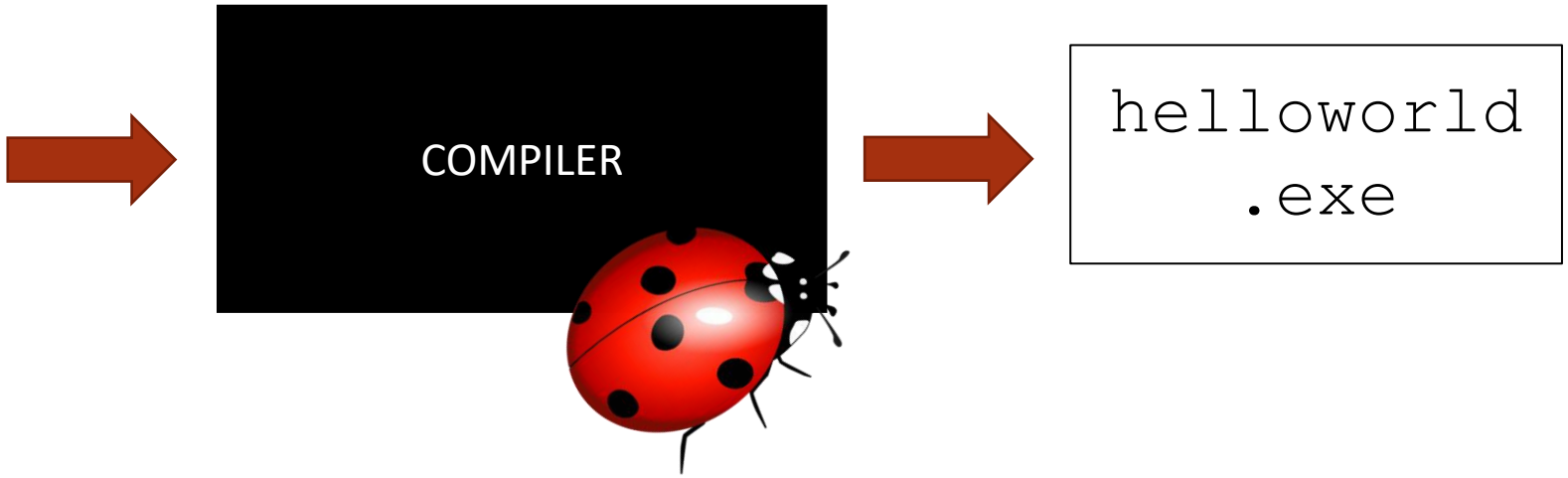
Karine Even-Mendoza, Cristian Cadar, Alastair F. Donaldson

Imperial College London

ASE JF 2022 - October 2022

Compiler correctness is extremely important

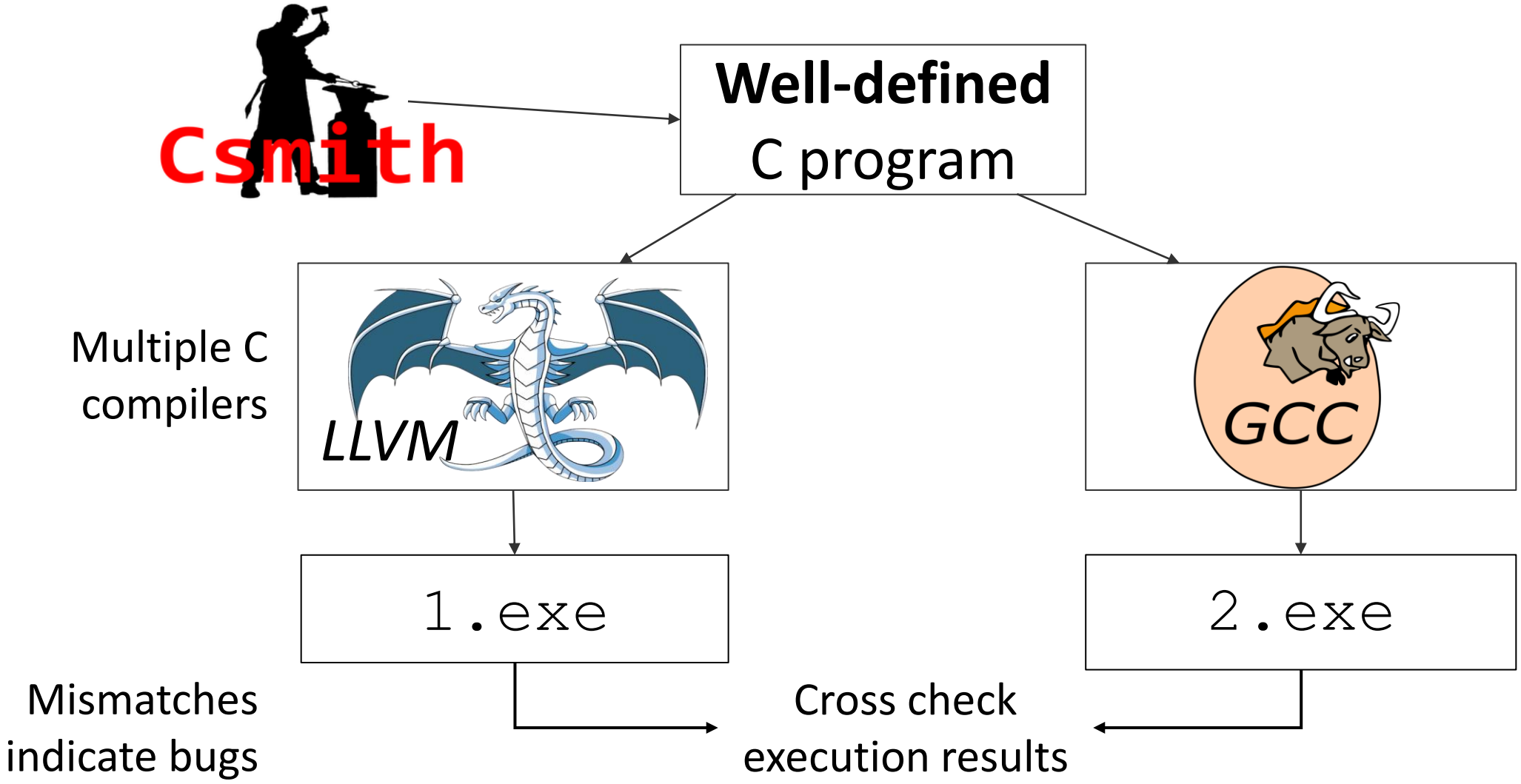
```c
#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```



(1) Crashes/hangs **or** (2) silently produces incorrect code

➔ broad impact on the quality of software

Differential compiler testing has been extremely effective!



Differential compiler testing has been extremely effective!

Csmith has found hundreds of bugs in GCC and LLVM

Csmith team won Most Influential PLDI 2011 Paper Award (at PLDI 2021)

But …

Compilers have become **immune** to Csmith

Prof John Regehr (Csmith research group lead) in 2019:

Similar story for other compiler fuzzing tools

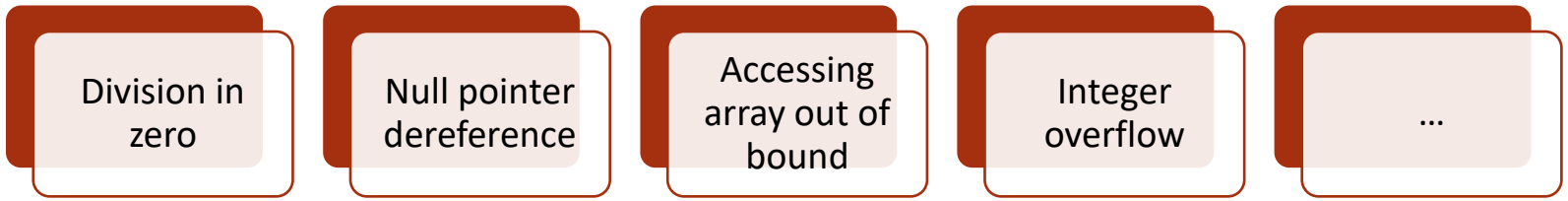
CsmithEdge: closer to the edge

- New fuzzer: compilers not yet immune to it but … takes long time to develop
- Idea: can we adapt **existing** fuzzers to find new bugs?

- **CsmithEdge** ➔ gets **closer to the edge** of the language semantics
  - By being less conservative about undefined behaviours
- **9** new bugs in C compilers + detected several old bugs
- **None** of these bugs can be found by regular Csmith!

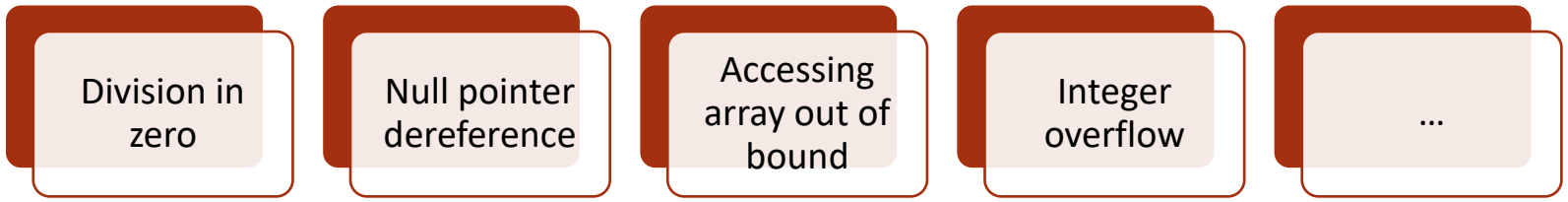
Fuzzing, compilers and undefined behaviours

- **Main challenge**: generating interesting + UB-free-programs
- **Undefined Behaviours** (UB)



Fuzzing, compilers and undefined behaviours

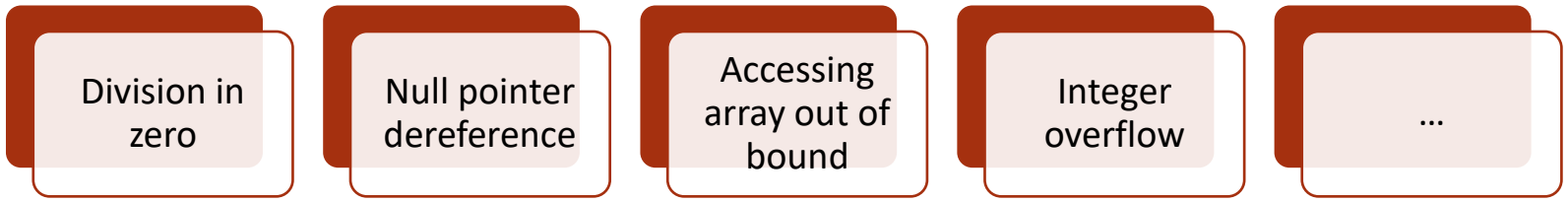
- **Main challenge**: generating interesting + UB-free-programs
- **Undefined Behaviours** (UB)



UB = behaviour that the does not respect the language specification and for which the International Standard imposes no requirements

Fuzzing, compilers and undefined behaviours

- **Main challenge**: generating interesting + UB-free-programs
- **Undefined Behaviours** (UB)



- **Programs with UB**: unpredictable result ➔ mismatches meaningless
  ➔ compiler developers specifically request not to file such reports

Fuzzing, compilers and undefined behaviours

- Csmith introduces constraints for UB-free program generation

- Example: avoid UB related to division in zero via “safe math” wrappers

a/b → $(b == 0)\,?\,a : a/b$

Unsafe division

Safe division wrapper

Fuzzing, compilers and undefined behaviours

- Csmith introduces constraints for UB-free program generation

- Example: avoid UB related to division in zero via “safe math” wrappers

$$\underbrace{a/b}_{\text{Unsafe division}} \longrightarrow \underbrace{(b == 0)\,?\,a : a/b}_{\text{Safe division wrapper}}$$

```c
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = s+i;
    t = t/i;
  }
  printf("Result: %d,%d\n", s,t);
}
```

Fuzzing, compilers and undefined behaviours

- Csmith introduces constraints for UB-free program generation

- Example: avoid UB related to division in zero via “safe math” wrappers

$$\underbrace{a/b}_{\text{Unsafe division}} \longrightarrow \underbrace{(b == 0)\,?\,a : a/b}_{\text{Safe division wrapper}}$$

```c
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = s+i;
    t = t/i;
  }
  printf("Result: %d,%d\n", s,t);
}
```

Fuzzing, compilers and undefined behaviours

- Csmith introduces constraints for UB-free program generation

- Example: avoid UB related to division in zero via “safe math” wrappers

a/b → $(b == 0)\,?\,a : a/b$

Unsafe division — Safe division wrapper

```
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = s+i;
    t = t/i;
  }
  printf("Result: %d,%d\n", s,t);
}
```

```
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = safe_add(s, i);
    t = safe_div(t, i);
  }
  printf("Result: %d,%d\n", s,t);
}
```

CsmithEdge – research hypothesis

Observation
Resulting program never contains certain expressions/statements

Problem
Some of the code optimizations in the compiler can be inapplicable

Hypothesis
Generation constraints limit the form of programs we can generate and thus the bugs we can find

CsmithEdge vs Csmith

- **Observation + Hypothesis ➔** found new bugs in GCC, LLVM and Visual Studio

```
int main(){
    const long ONE = 1L;
    long y = 0L;
    long x = ((long) (ONE || (y = 1L)) % 8L);
    printf("x = %ld, y = %ld\n", x, y);
}
```

➔ **Bug: violation of the short-circuiting** op. rule: if the first operand is sufficient to determine the overall result, then the second operand should not be evaluated, in case it commits side effects or exhibits UB.

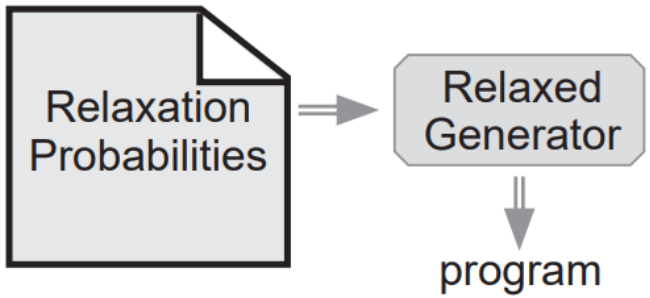
➔ Replace `safe_mode` with the operator itself

➔ Arithmetic operators **can** appear now outside the ternary operator

CsmithEdge: being less conservative

Modify Csmith to create more interesting programs by weaken constraints related to UB avoidance

(1) Weaken generation constraints

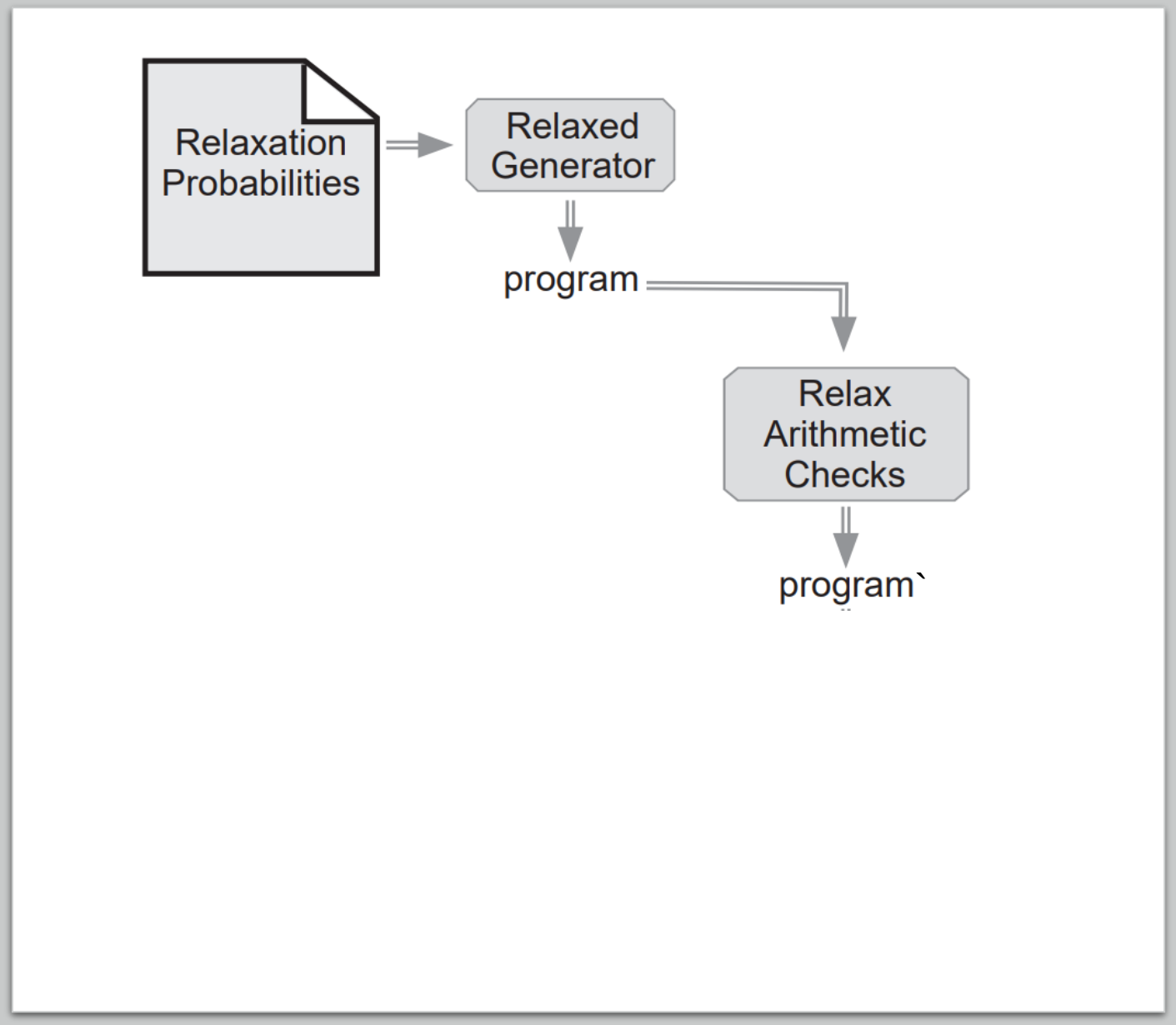


CsmithEdge: being less conservative

Modify Csmith to create more interesting programs by weaken constraints related to UB avoidance

(1) Weaken generation constraints

(2) Weaken post generation constraints

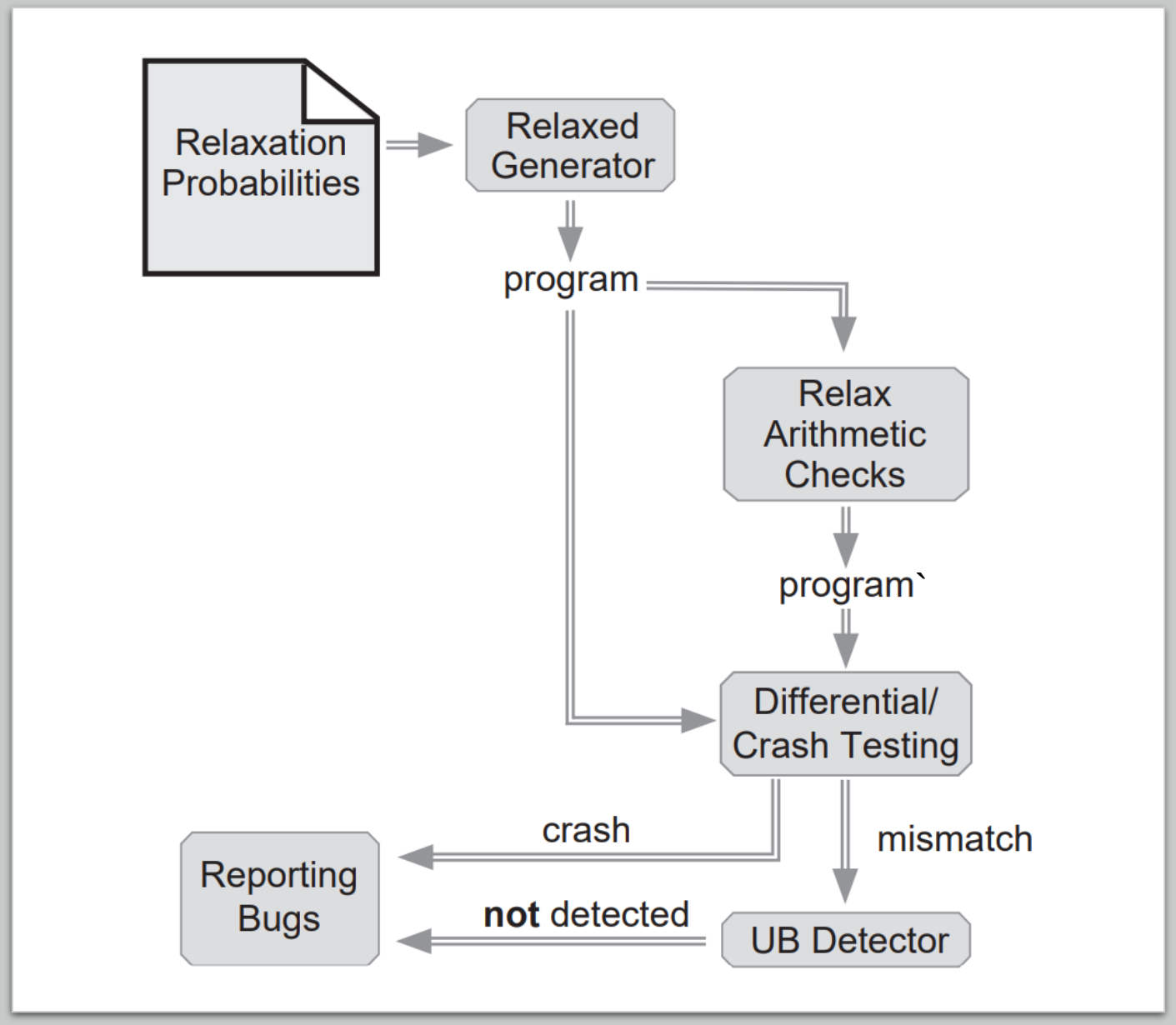


CsmithEdge: being less conservative

Modify Csmith to create more interesting programs by weaken constraints related to UB avoidance

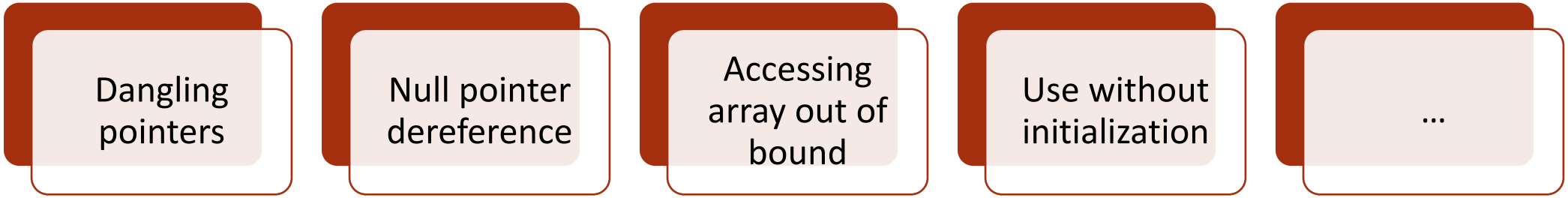
(1) Weaken generation constraints

(2) Weaken post generation constraints



CsmithEdge: weaken generation constraints

- These constraints guard against



- Use set of probabilities to decide separately per generated testcase:
  (1) a sub-set of constraint to weaken
  (2) The probabilities each of the selected constraint can be weaken

- **Example**: allow null pointer deference with 10% of the times (that is, enforce the constraint 90% of the times), and allow accessing array out of bound 23% of the times; the rest of the constraint are enforced all the time

CsmithEdge: weaken post generation constraints

- Post generation constraints: `safe_math` wrappers for arithmetic operators

- Given a testcase: CsmithEdge's dynamic analysis detects and replaces redundant `safe_math` uses with the corresponding arithmetic operator

```
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = safe_add(s, i);
    t = safe_div(t, i);
  }
  printf("Result: %d,%d\n", s,t);
}
```

Relax arithmetic checks

```
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = s+i;
    t = safe_div(t, i);
  }
  printf("Result: %d,%d\n", s,t);
}
```

Evaluation

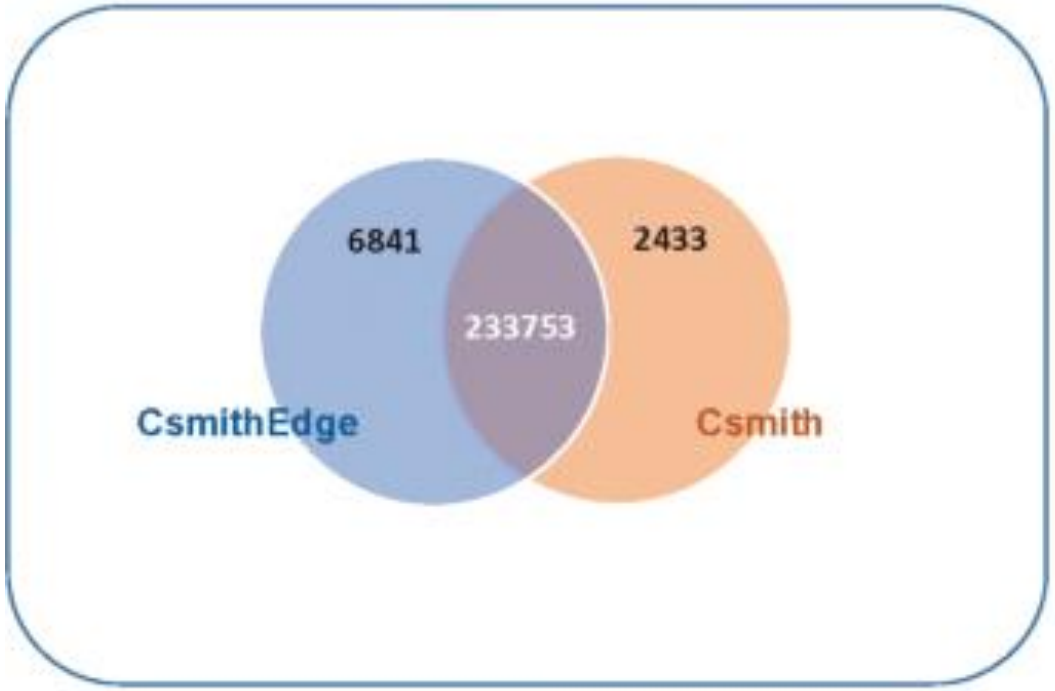
Six-month evaluation in the wild

- 7 new bugs in GCC, 1 new bug in LLVM, 1 new bug in Visual Studio, and several bugs in older versions
- Each of which required a different subset of relaxations

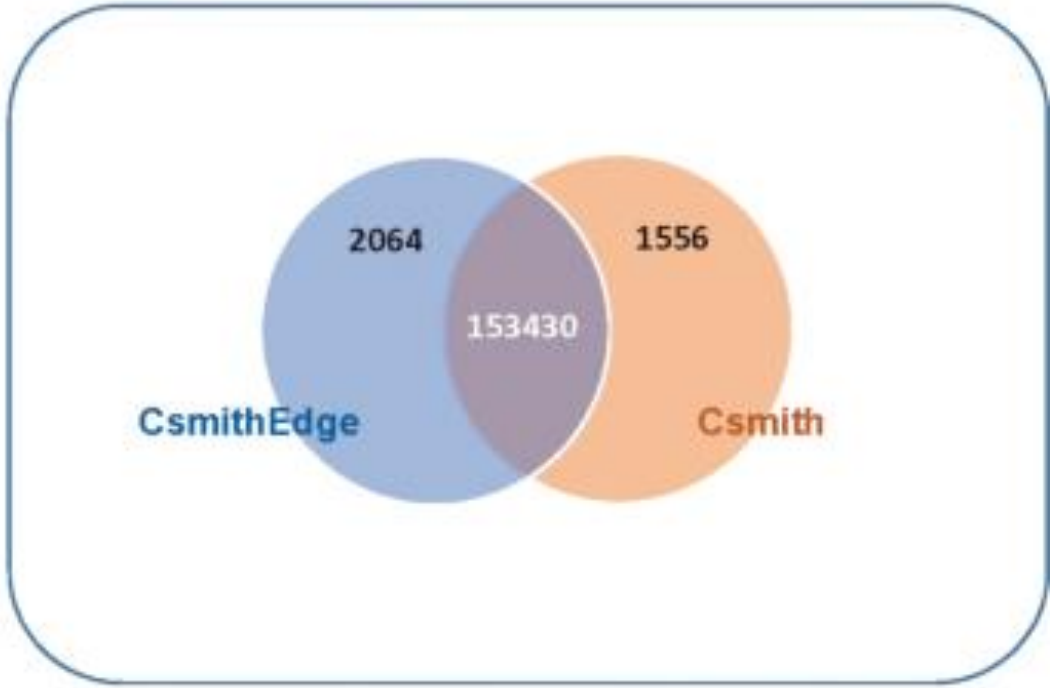
Throughput

- 1.6x overhead due to the use of sanitizers (50 s + lazy use of sanitizers)
- Depends on timeout settings and sanitizers ➔ full details in the paper!

Additional Coverage – 135 K programs

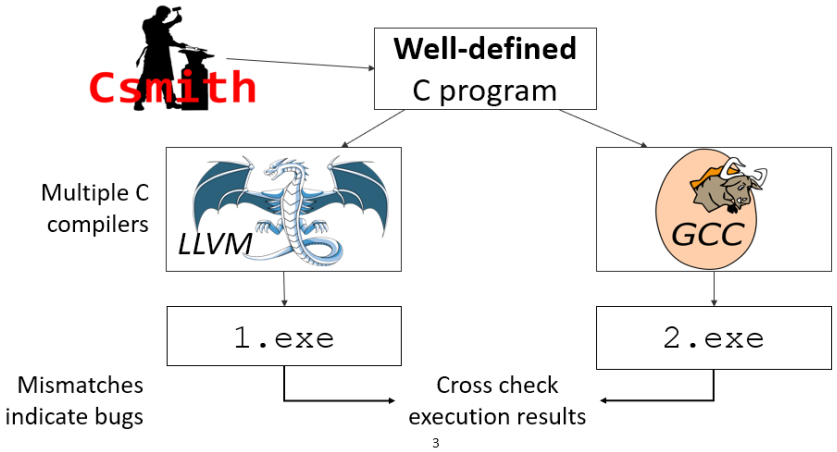


(c) GCC 10.2.1



(d) LLVM 11.0.0

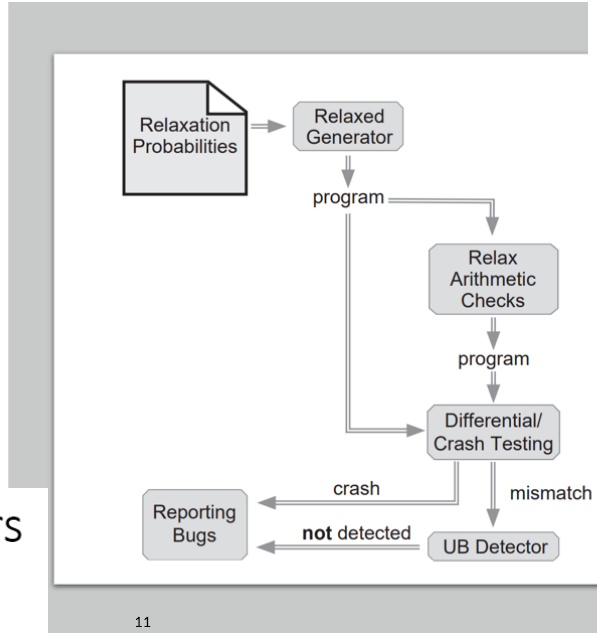
Differential compiler testing has been extremely effective!



- Modify Csmith to create more interesting programs by

(1) Weaken generation constraints

(2) Weaken post generation constraints



CsmithEdge: closer to the edge

- New fuzzer: compilers not yet immune to it but … takes long time to develop
- Idea: can we adapt **existing** fuzzers to find new bugs?

- **CsmithEdge** ➔ gets **closer to the edge** of the language semantics
  - By being less conservative about undefined behaviours
- **9** new bugs in C compilers + detected several old bugs
- **None** of these bugs can be found by regular Csmith!

Fuzzing, compilers and undefined behaviours

- Csmith introduces constraints for UB-free program generation
- Example: avoid UB related to division in zero via “safe math” wrappers

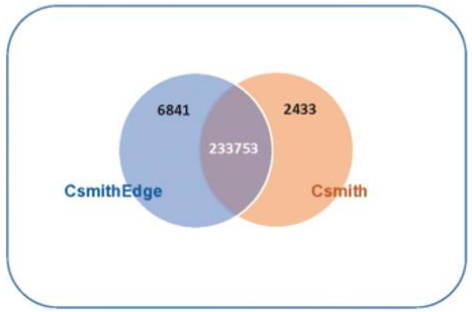
a/b → $(b == 0)\,?\,a : a/b$

Unsafe division — Safe division wrapper

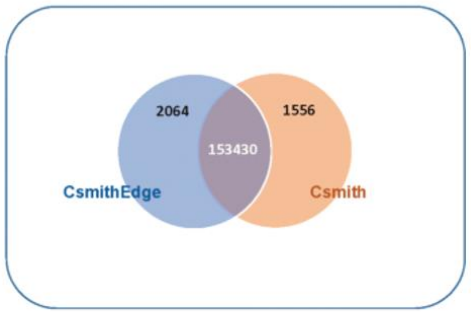
```
int main()
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = s+i;
    t = t/i;
  }
  printf("Result: %d,%d\n", s,t);
}
```

```
int main
{
  int s = 5;
  int t = 2147483646;
  for (int i = 8; i >= -8; i--) {
    s = safe_add(s, i);
    t = safe_div(t, i);
  }
  printf("Result: %d,%d\n", s,t);
}
```

Additional Coverage – 135 K programs



(c) GCC 10.2.1



(d) LLVM 11.0.0