Finding and Understanding Bugs in C Compilers

By

Xuejun Yang, Yang Chen, Eric Eide, John Regehr

Presented by:

Tony Xiao

Motivation

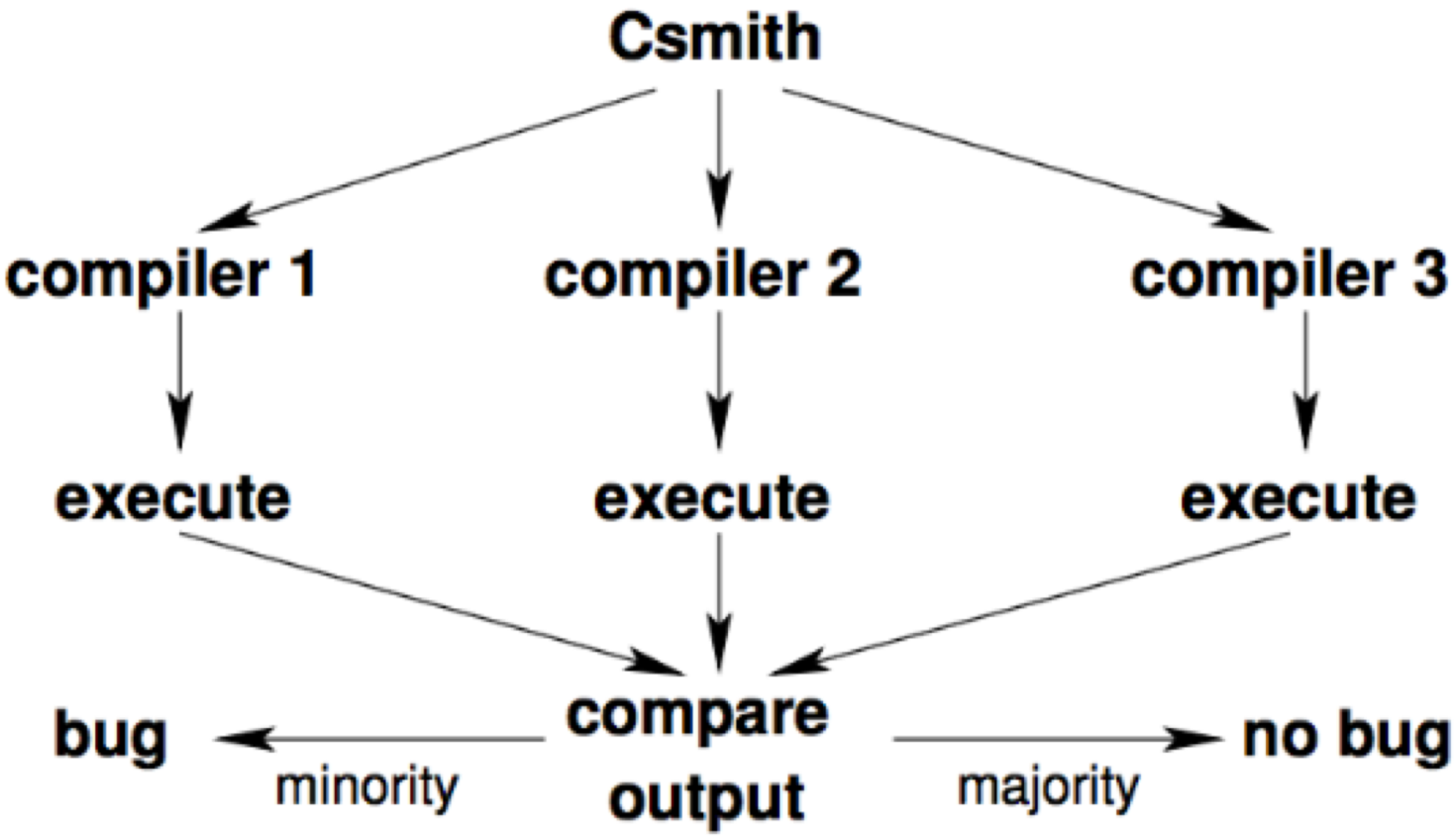
- Code for various safety critical embedded software, operating systems, servers, etc. is written in C.
- If a bug exists in a C compiler it could generate incorrect code which could have severe implications especially in critical applications.
- A bug in the compiler could also prevent the compiler from compiling perfectly valid code leading to wasted time of the application developer spent in trying to debug the issue.

Csmith

- It is a tool that helps uncover bugs in compilers through random testing (also known as fuzzing).
- Found more than 325 bugs, of which 25 bugs were assigned P1 priority

A 40,000 lines C++ program to generate random C programs

Bug Hunting


Csmith
compiler 1
compiler 2
compiler 3
execute
execute
execute
compare
output
bug
minority
no bug
majority

Example of Wrong Safety Check

(x == c1) || (x < c2)

x < c2

when c1 and c2 are constants and c1 < c2.

(x == 0) || (x < -3)

x < -3

LLVM did an unsigned comparison

Example of Wrong Analysis

```
1:  static int g[1];
2:  static int *p = &g[0];
3:  static int *q = &g[0];
4:
5:  int foo (void) {
6:      g[0] = 1;
7:      *p = 0;
8:      *p = *q;
9:      return g[0];
10: }
```

Result = 0

GCC = 1

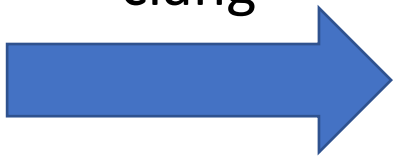
Design Goals of Csmith

1. **Every randomly generated program must be well formed and have a single interpretation based on the C standard.(i.e. avoid undefined behavior)**
2. Maximize "expressiveness". Expressiveness is the idea that the generated programs should use a wide variety and combinations of language features.

Example of Undefined Behavior

```
#include <cstdlib>

typedef int (*Function)();

static Function Do;

static int EraseAll() {
  return system("rm -rf /");
}

void NeverCalled() {
  Do = EraseAll;
}

int main() {
  return Do();
}
```

clang



```
main:
        movl    $.L.str, %edi
        jmp     system

.L.str:
        .asciz  "rm -rf /"
```

Ref to [Krister Walfridsson's blog]

Design Goals of Csmith

1. Every randomly generated program must be well formed and have a single interpretation based on the C standard.
2. **Maximize "expressiveness". Expressiveness is the idea that the generated programs should use a wide variety and combinations of language features.**

High Level Steps for Program Generation

Preliminary Step: Randomly generate a bunch of struct declarations

Generates a top-level function which will be called by main later

- Select an allowable production from grammar for the current program point
  - Consult probability table and then filter function
- Select target (variable or function)
  - dynamic probability table of potential targets
- Select a type
  - Restricted or unrestricted
- Nonterminal recursion
- Update local environment with points-to facts
- Safety checks
  - commit or rollback

Grammar used for Program Generation

- PROGRAM ::= <type-def-list><var-def-list><func-def-list>
- func-def-list ::= func-def <func-def-list>
- func-def ::= type func-name { block }
- block ::= <declaration-list> <statement-list>
- statement-list ::= statement <statement-list>
- statement ::= expression | control-flow | assignment | block
- control-flow ::= if … else| return | goto | for

Safety Mechanisms

Problem	Code-Generation-Time Solution	Code-Execution-Time Solution
use without initialization	explicit initializers, avoid jumping over initializers	—
qualifier mismatch	static analysis	—
infinite recursion	disallow recursion	—
signed integer overflow	bounded loop vars	safe math wrappers
OOB array access	bounded loop vars	force index in bounds
unspecified eval. order of function arguments	effect analysis	—
R/W and W/W conflicts betw. sequence points	effect analysis	—
access to out-of-scope stack variable	pointer analysis	—
null pointer dereference	pointer analysis	null pointer checks

Integer Safety

The safety problem of integers comes from undefined behaviors (UB) such as signed overflow:

```
int signedOverflow(int x) {
return x+1 > x; // either true or UB due to signed overflow }
```

and shift-past-bitwidth:

```
int shiftPastBitwidth() {
return 1 << 32; // UB when evaluated on 32 bit platform }
```

Pointer Safety

The first kind of pointer safety problem is null pointer dereference.

```
int a = 10;
void nullDereference(int *p) {
 *p = a; // cause execption if p is NULL
}
```

This can be easily avoided by dynamic checks.

```
int a = 10;
void safeDereference(int *p) {
if (p != NULL) { *p = a; } }
```

However, there is no reliable method to identify an invalid pointer that points to a function-scoped variable.

```
void invalidDereference(int *p) {
 int a = 10;
if( p != NULL){
        *p = a; } } // outside this function, we cannot dereference or compare p with other pointer // before it becomes valid again!
```

Global Safety

```
void incrementallyGeneratedUnsafeProgram() {
    int *p = &i;
    while (...) {
        *p = 3;
        p = 0; // unsafe because of the back-edge
    }
}
```

Design Trade-offs

- Allow implementation defined behaviour
- No ground truth
- No guarantee of termination (10% of programs generated by Csmith are non-terminating)
- Target middle-end bugs

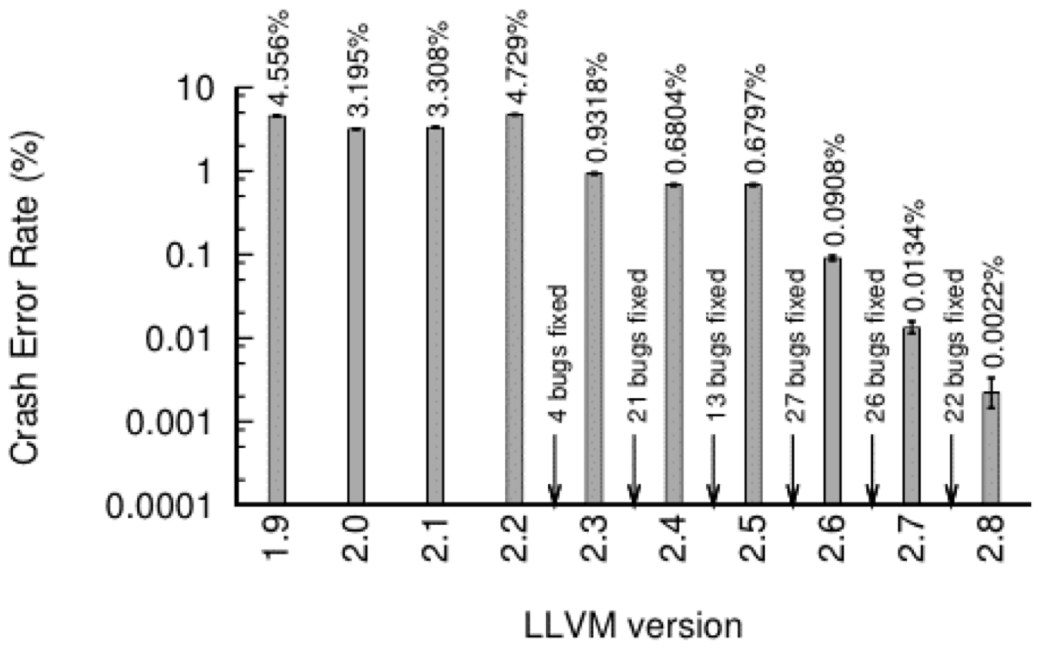
Experiments

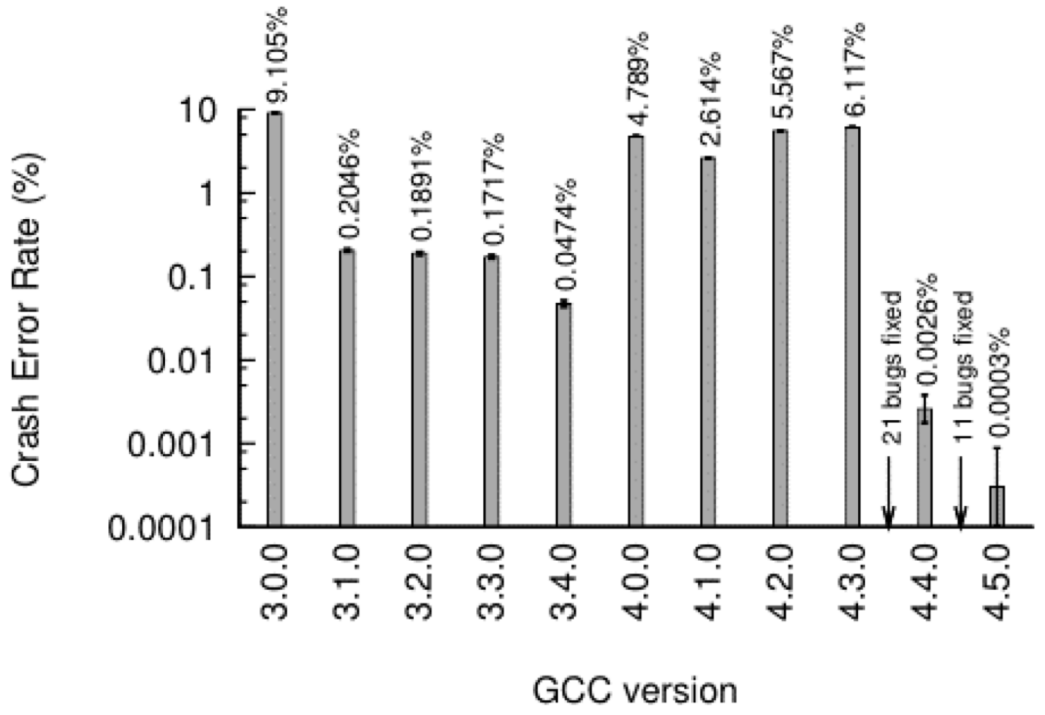
1. Unstructured & uncontrolled experiments over a period of 3 years where the authors used Csmith to find bugs in a variety of C compilers.
2. Compiled & ran one million random programs (generated by Csmith) using different versions of GCC and LLVM at optimization levels –O0, –O1, –O2, –Os, and –O3.
3. Examine Csmith's bug finding power as a function of the size of the random program.
4. Compare Csmith's performance to four other random C program generators.
5. Effect of random programs on branch, function & line coverage of GCC and LLVM source code.

Results

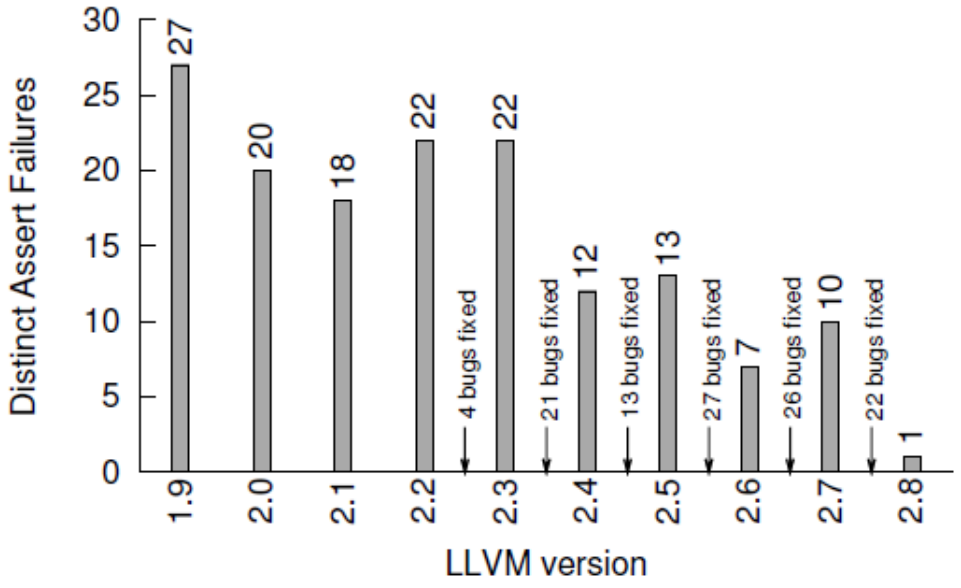
- Found bugs in 11 C compilers (5 were open source, 5 were commercial and the 11th, CompCert is publicly available but not open source).
- Two types of bugs – Compile time crash error, wrong code error.
- Experience with commercial compiler teams was not all that good.
- Focus was mainly on GCC and LLVMs.
- 202 LLVM bugs, 79 GCC bugs (out of which 25 were marked as highest priority).
- A total of 325 bugs found (in gcc, llvm and other commercial compilers).

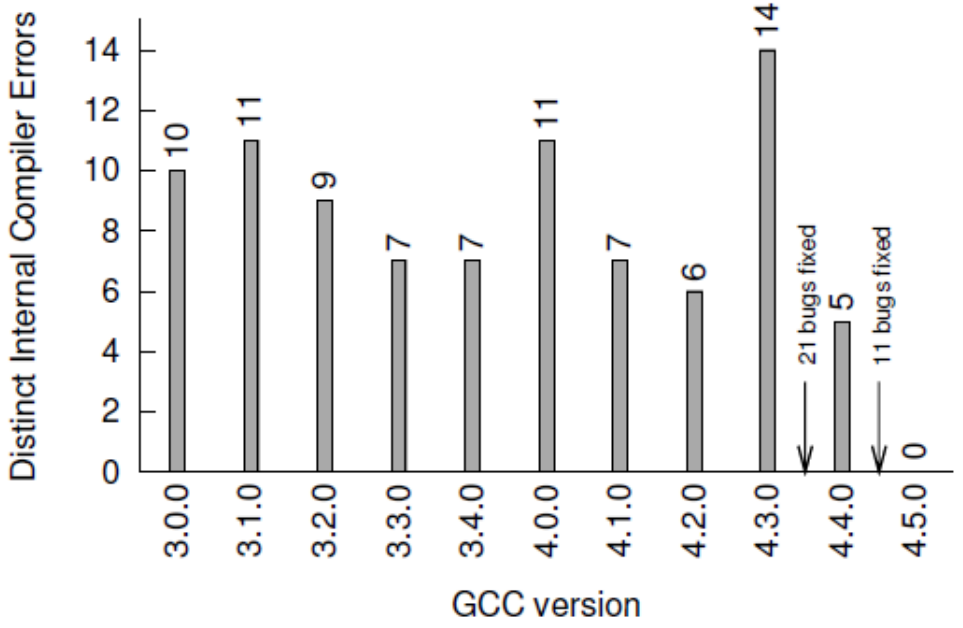
Experiment with Different Versions of the Same Compiler



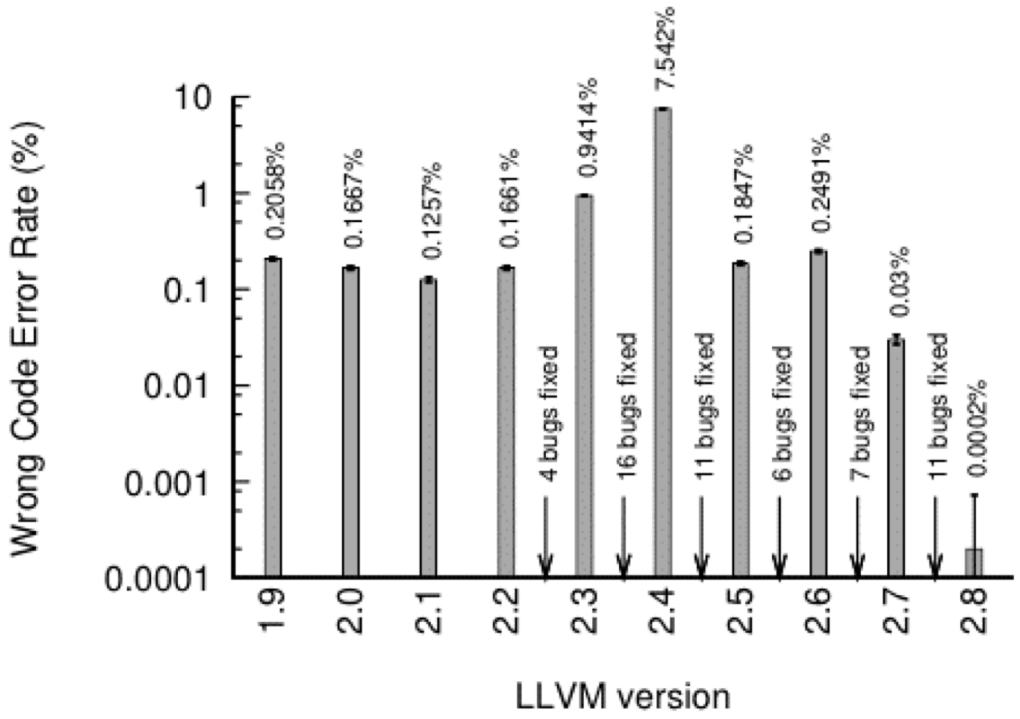


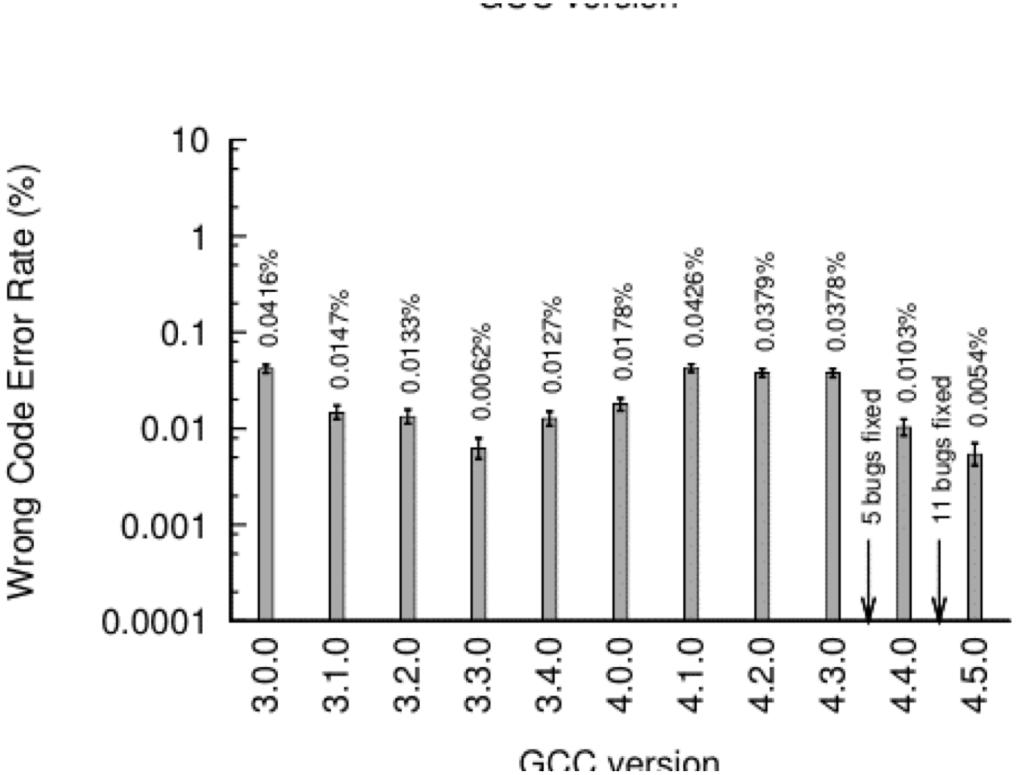
Experiment with Different Versions of the Same Compiler



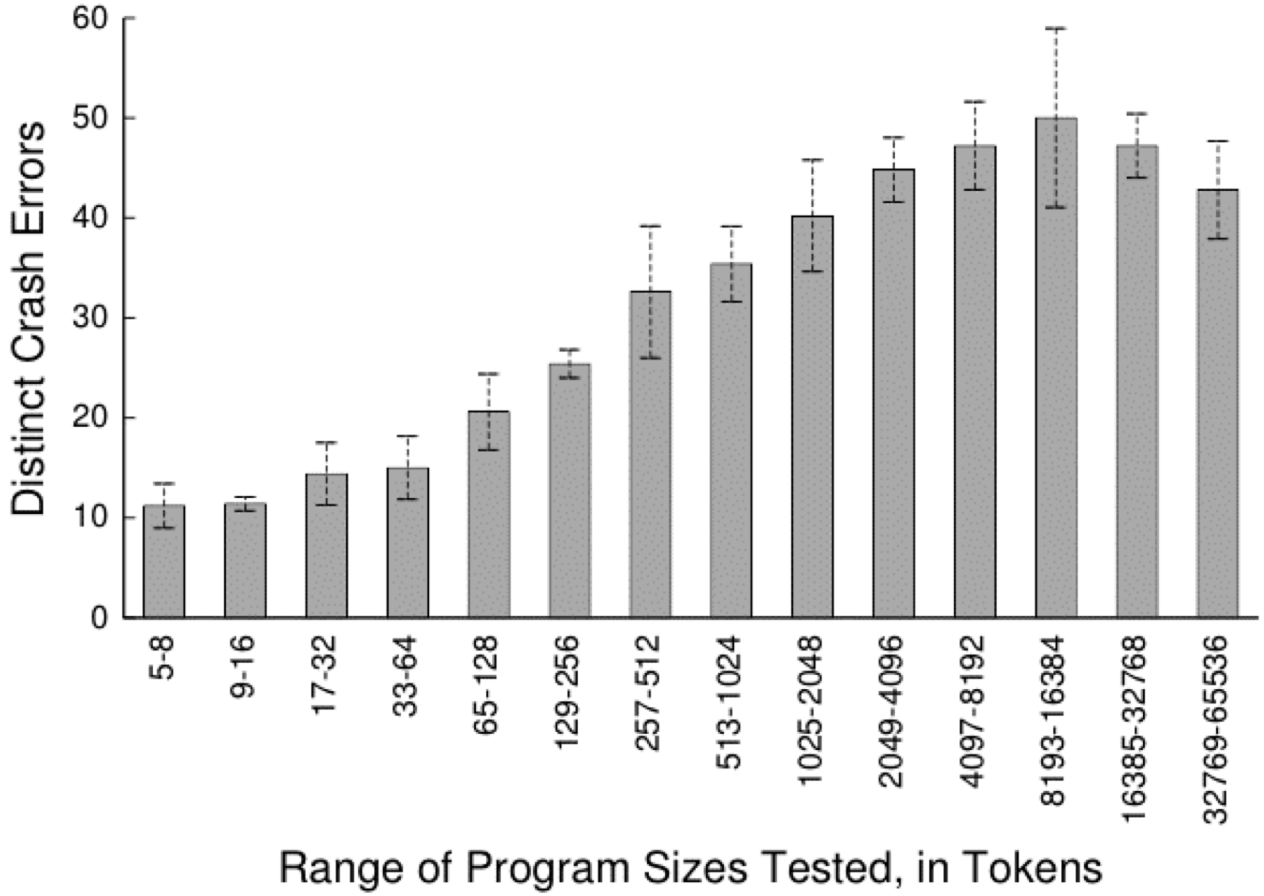


Experiment with Different Versions of the Same Compiler

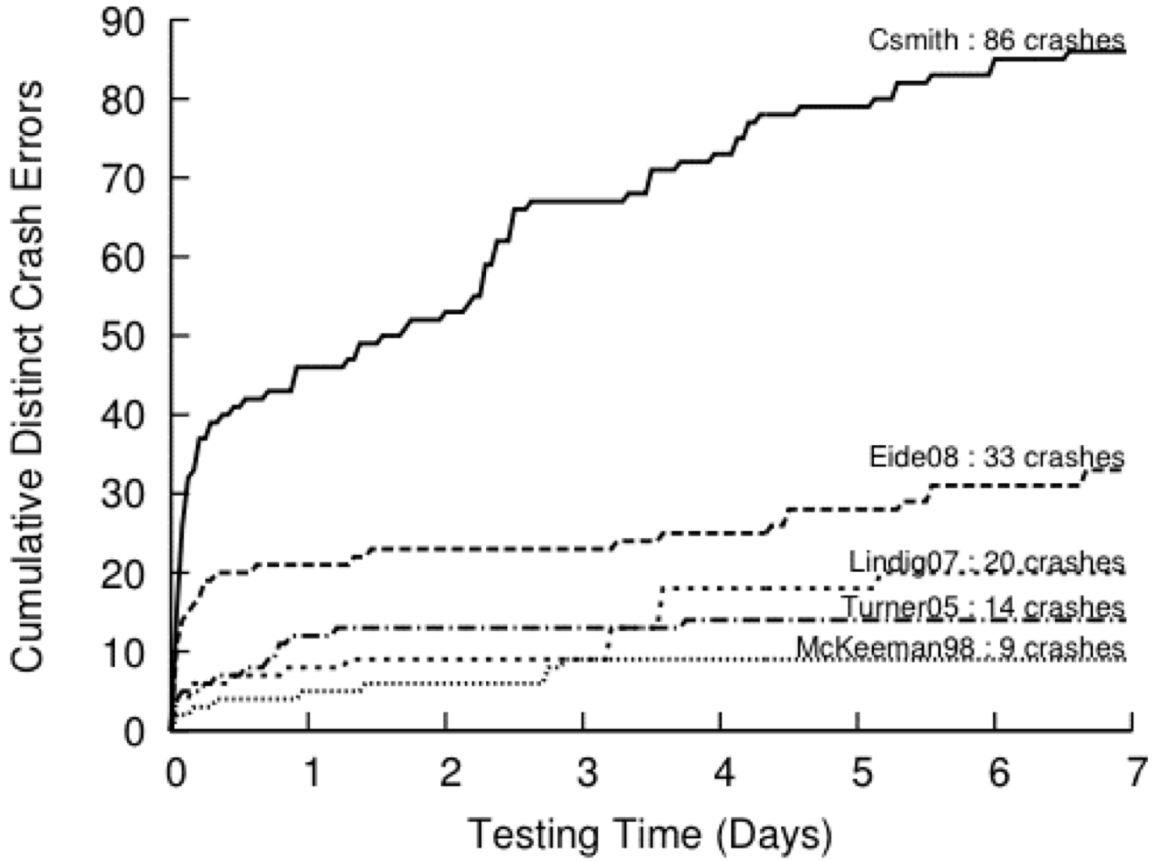




Number of Bugs and Generated Program Size



Comparison of Csmith & other Random C Program Generators



Effect of Random Programs on Code Coverage

		Line Coverage	Function Coverage	Branch Coverage
GCC	make check-c	75.13%	82.23%	46.26%
	make check-c & random	75.58%	82.41%	47.11%
	% change	+0.45%	+0.13%	+0.85%
	absolute change	+1,482	+33	+4,471
Clang	make test	74.54%	72.90%	59.22%
	make test & random	74.69%	72.95%	59.48%
	% change	+0.15%	+0.05%	+0.26%
	absolute change	+655	+74	+926

Discussion:

- The authors do not come up with a good explanation for the code coverage issue. What might be the reason?
- Tests that are randomly generated will never be like tests that are created by humans. Does this mean that this kind of testing is still useful?

Thank you!