Exposing Difficult Compiler Bugs With Random Testing

John Regehr, Xuejun Yang, Yang Chen, Eric Eide
University of Utah

- Found serious wrong-code bugs in all C compilers we've tested
  - Including GCC
  - Including expensive commercial compilers
  - Including 11 bugs in a research compiler that was proved to be correct
  - 287 bugs reported so far
    - Counting crash and wrong-code bugs

```c
static int x;
static int *volatile z = &x;
static int foo (int *y) {
  return *y;
}
int main (void) {
  *z = 1;
  printf ("%d\n", foo(&x));
  return 0;
}
```

- Should print “1”
- GCC r164319 at -O2 on x86-64 prints “0”

```c
int foo (void) {
    signed char x = 1;
    unsigned char y = 255;
    return x > y;
}
```

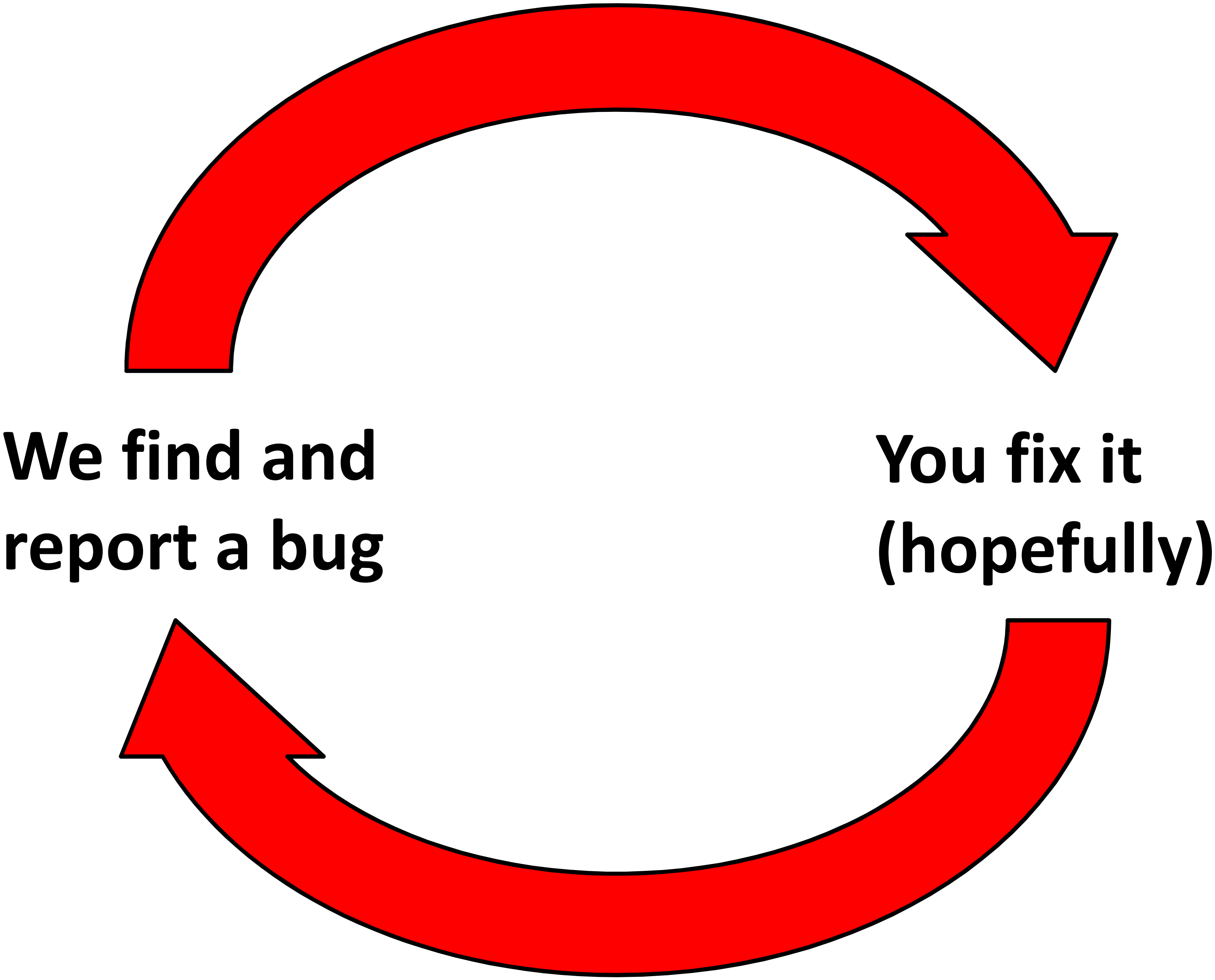
- **Should return 0**
- **GCC 4.2.3 from Ubuntu Hardy (8.04) for x86 returns 1 at all optimization levels**

```c
const volatile int x;
volatile int y;

void foo(void) {
  for (y=0; y>10; y++)
    {
      int z = x;
    }
}
```

```asm
foo: movl   $0, y
     movl   x, %eax
     jmp    .L3
.L2: movl   y, %eax
     incl   %eax
     movl   %eax, y
.L3: movl   y, %eax
     cmpl   $10, %eax
     jg     .L3
     ret
```

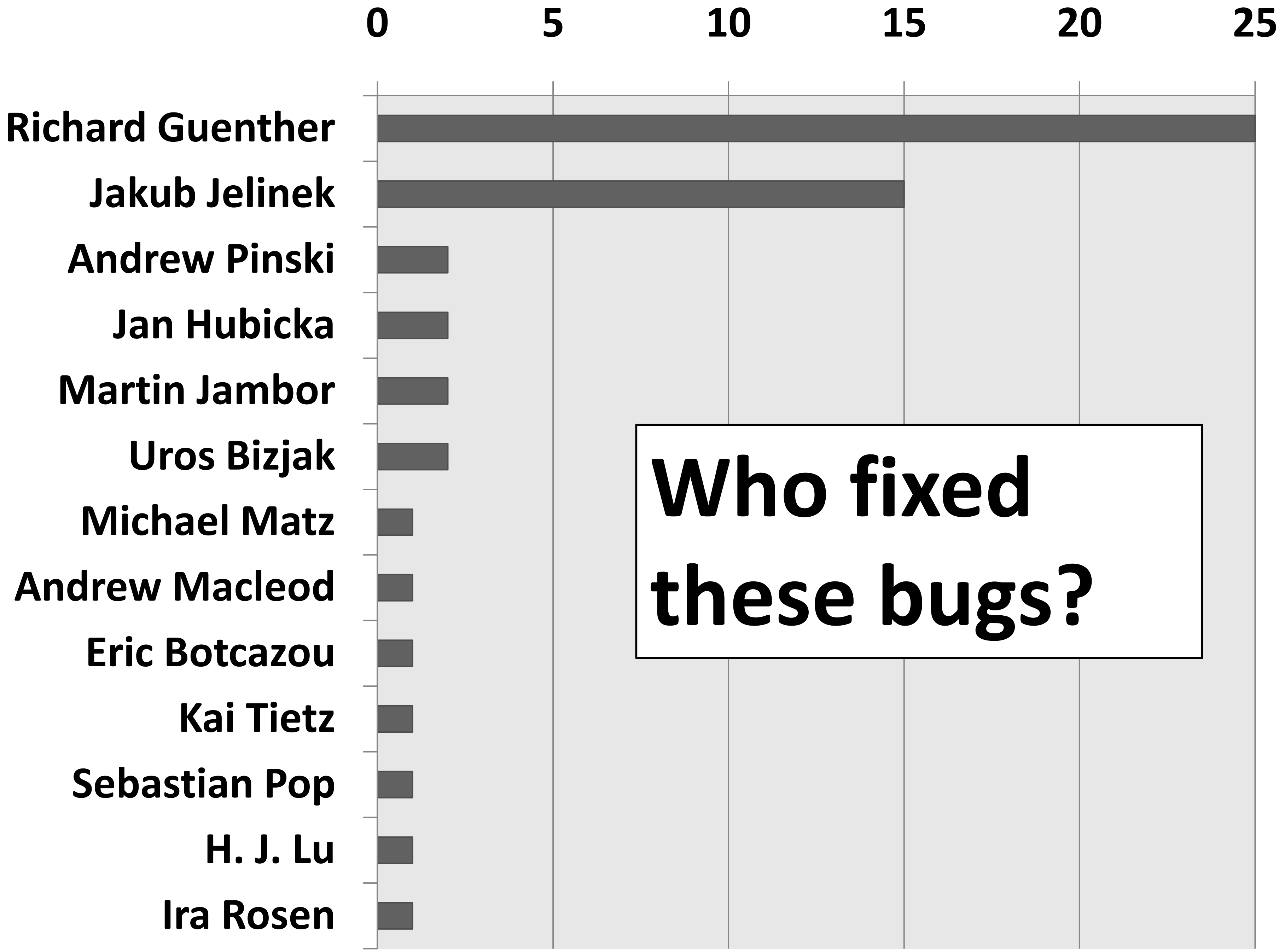
- **GCC 4.3.0 -Os for x86**


We find and report a bug
You fix it (hopefully)

55 bugs fixed so far + a few reported but not yet fixed

20 of these bugs were P1

Goal: Harden GCC by finding and killing difficult optimizer bugs


0
5
10
15
20
25
Richard Guenther
Jakub Jelinek
Andrew Pinski
Jan Hubicka
Martin Jambor
Uros Bizjak
Michael Matz
Andrew Macleod
Eric Botcazou
Kai Tietz
Sebastian Pop
H. J. Lu
Ira Rosen
Who fixed these bugs?

What Kind of Bugs?

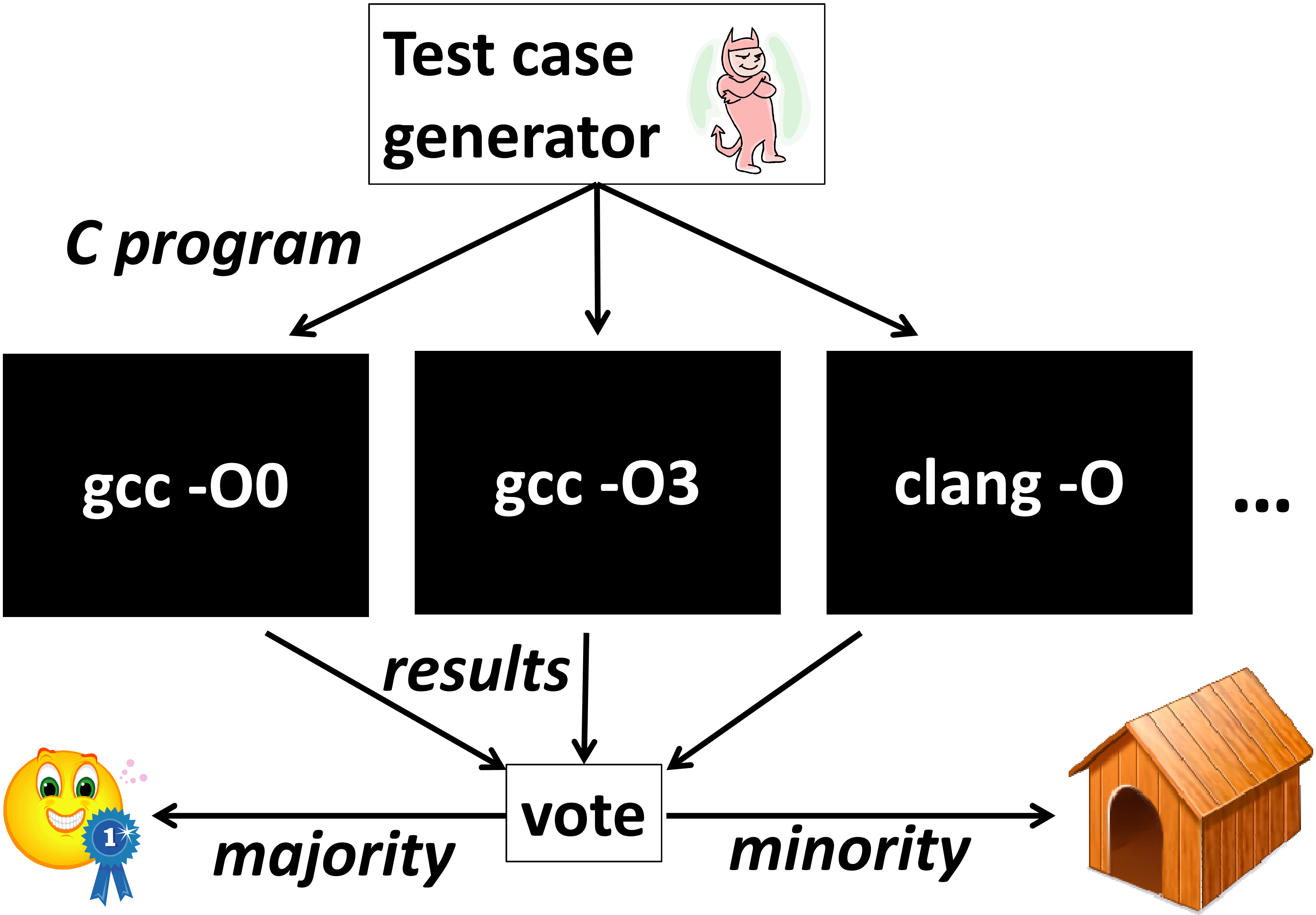
- Compiler crash or ICE
- Compiler generates code that…
  - Crashes
  - Computes wrong value
  - Wrongfully terminates
  - Wrongfully fails to terminate
  - Accesses a volatile wrong number of times

1. What we do
2. $\boxed{\text{How we do it}}$
3. What we learned
4. What still needs to happen


INT
MAIN()
{ ...

```
if ((((l_421 || (safe_lshift_func_uint8_t_u_u (l_421, 0xABE574F6L))) && (func_77(func_38((l_424 >=
    l_425), g_394, g_30.f0), func_8(l_408, g_345[2], g_7, (*g_165), l_421), (*l_400), func_8(((*g_349) !=
    (*g_349)), (l_426 != (*l_400)), (safe_lshift_func_int16_t_s_u ((**g_349), 0xD5C55EF8L)), 0x0B1F0B62L,
    g_95), (safe_add_func_uint32_t_u_u ((*g_165), l_431))) ^ ((safe_rshift_func_uint8_t_u_s (((*g_165)
    >= (**g_349)), (safe_mul_func_int8_t_s_s ((*g_165), l_421)))) <= func_77((*g_129), g_95, 1L, l_408,
    (*l_400))))) | (*l_400))) {
  struct S0 *l_443 = &g_30;
  (*l_400) = ((safe                                            ), (**g_349)))
    && l_425);
   l_447 ^= (safe_s                                            g_129,
    (*g_129), l_40
   (*l_446) = func_                                            ));
} else {
  const uint32_t l_
  l_448 = (*g_186)
  (*l_400) = (0L &
  (*l_400) = func_77((*g_31), ((*g_165) && 6L), l_426, func_77((*l_441), (safe_lshift_func_uint16_t_u_u
    ((((safe_mul_func_int16_t_s_s ((**g_349), (*g_165))) | ((*g_165) > l_426)) < (0 != (*g_129))), (&l_431
    == &l_408))), (l_453 == &l_407), func_77(func_38((*l_400), (safe_mod_func_uint16_t_u_u ((l_420 <
    (*g_165)), func_77((*l_441), l_456, (*l_446), (*l_448), g_345[5]))), g_345[4]), g_287,
    (func_77((*g_129), l_421, (l_424 & (**g_349)), ((*l_453) != (*g_129)), 0x6D4CA97DL) ==
    (safe_div_func_int64_t_s_s (-1L, func_77((*g_129), l_459, l_447, (*l_446), l_459)))), g_95, g_19),
    l_420), (*l_446));
}
```

A test program:
- Does lots of random stuff
- Checksums its globals
- Prints checksum and exits



Test case generator
C program
gcc -O0
gcc -O3
clang -O
...
results
vote
majority
minority

Test Case Generator

- Driven by
  - Random search
  - Depth first search
- Based on
  - Grammar for subset of C
  - Analyses to ensure test case validity

Not a Bug #1

```
int foo (int x) {
  return (x+1) > x;
}

int main (void) {
  printf ("%d\n",
          foo (INT_MAX));
  return 0;
}
```

```
$ gcc -O1 foo.c -o foo
$ ./foo
0
$ gcc -O2 foo.c -o foo
$ ./foo
1
```

Not a Bug #2

```c
int a;

void bar (int x, int y) {
}

int main (void) {
  bar (a=1, a=2);
  printf ("%d\n", a);
  return 0;
}
```

```
$ gcc -O bar.c -o bar
$ ./bar
1
$ clang -O bar.c -o bar
$ ./bar
2
```

Not a Bug #3

```c
int main (void) {
  long a = -1;
  unsigned b = 1;
  printf ("%d\n", a > b);
  return 0;
}
```

```
$ gcc -m64 baz.c -o baz
$ ./baz
0
$ gcc -m32 baz.c -o baz
$ ./baz
1
```

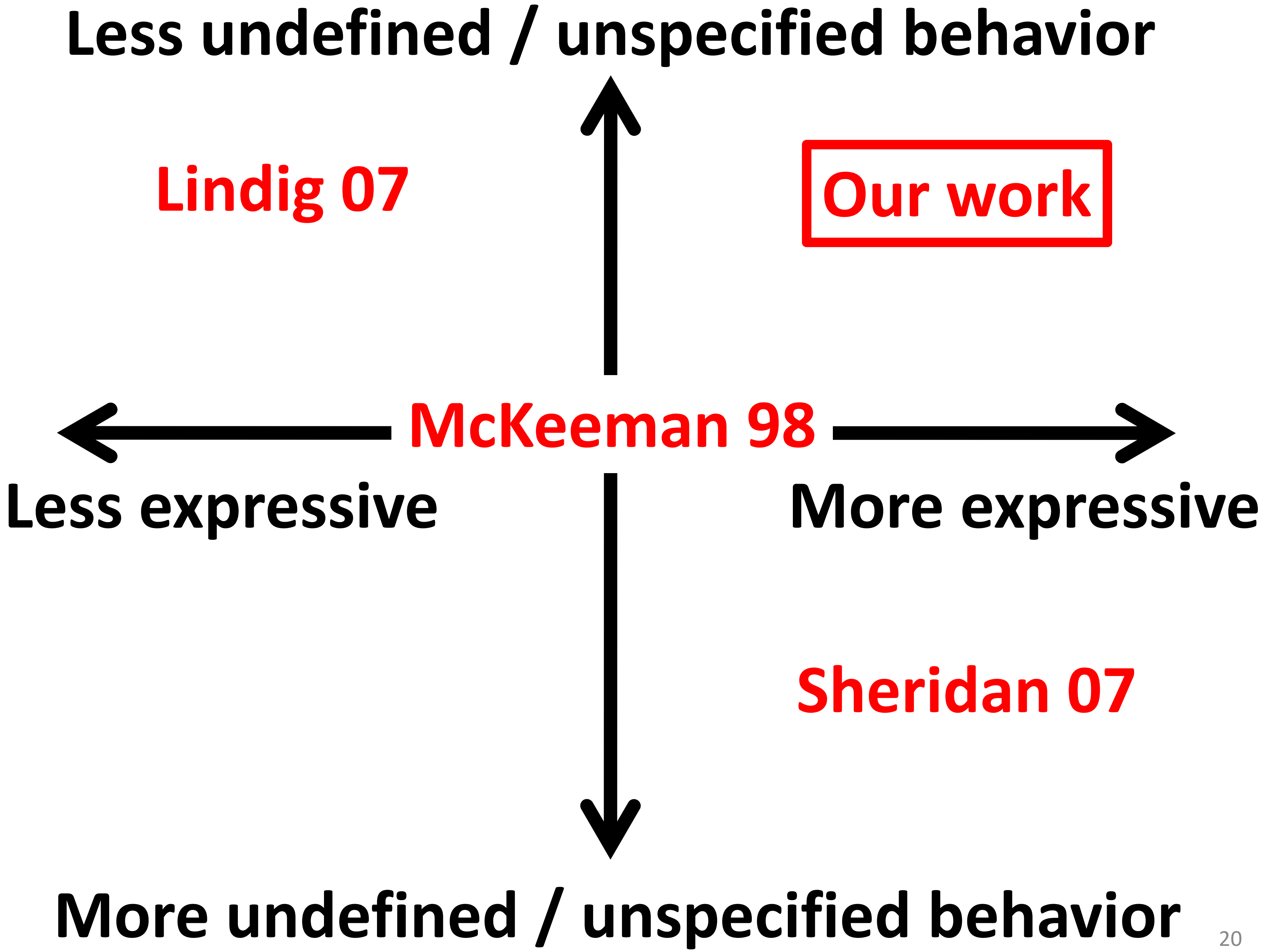
- **Key property for automated compiler testing:**
  - C standard gives each test case a unique meaning
  - Results differ → COMPILER BUG

- **Test cases must not…**
  - Execute undefined behavior (191 kinds)
  - Rely on unspecified behavior (52 kinds)

- **Expressive code generation is easy**
  - **If you don't care about undefined behavior**

- **Avoiding undefined behavior is easy**
  - **If you don't care about expressiveness**

- **Expressive code that avoids undefined / unspecified behavior is hard**


Less undefined / unspecified behavior
Lindig 07
Our work
McKeeman 98
Less expressive
More expressive
Sheridan 07
More undefined / unspecified behavior

Avoiding Undefined and Unspecified Behaviors

- Offline avoidance is too difficult
  - E.g. ensuring in-bounds array access
- Online avoidance is too inefficient
  - E.g. ensuring validity of pointer to stack
- Solution: Combine static analysis and dynamic checks

Order of Evaluation Problems

- Problem: Order of evaluation of function arguments is unspecified
- E.g.

```
foo(bar(),baz())
```

- Where bar() and baz() both modify some variable

Order of Evaluation Problems

- **Solution:**
  - **Compute conservative read and write set for each function**
    - **Interprocedural analysis**
    - **Including read/write through pointers**
  - **In between sequence points, never invoke functions where read and write sets conflict**

Integer Undefined Behaviors

- **Problem: These are undefined in C**
  - **Divide by zero**
  - **INT_MIN % -1**
    - **Debatable in C99 standard but undefined in practice**
  - **Shift by negative, shift past bitwidth**
  - **Signed overflow**
  - **Etc.**

Undefined Integer Behaviors

- **Solution: Wrap all potentially undefined operations**

```
int safe_signed_sub (int si1, int si2) {
  if (((si1^si2) & (((si1^((si1^si2)
        & (1 << (sizeof(int)*CHAR_BIT-1))))-si2)^si2))
        < 0) {
    return 0;
  } else {
    return si1 - si2;
  }
}
```

Pointer Problems

- **Problem: Undefined pointer behaviors**
  - **Null pointer deref**
  - **Deref pointer into dead stack frame**
  - **Create or use out of bounds pointer**

Pointer Problems

- **Solution:**
  - Some dynamic checks
    - `if (ptr) { … }`
  - Some static analysis
    - Track alias set for each pointer to ensure validity
    - Avoid casting away qualifiers

SUPPORTED

- Arithmetic, logical, and bit operations
- Loops
- Conditionals
- Function calls
- Const and volatile
- Structs
- Pointers and arrays
- Goto
- Break, continue
- Bitfields

UNSUPPORTED

- Comma operator
- Interesting type casts
- Strings
- Unions
- Floating point
- Nonlocal jumps
- Varargs
- Recursive functions
- Function pointers
- Malloc / free

Design Compromise #1

- Implementation-defined behavior is allowed
  - Avoiding it is too restrictive
- Cannot do differential testing of e.g. x86 GCC vs. AVR GCC
  - Fine in practice

Design Compromise #2

- No ground truth
  - If all compilers generate the same wrong answer, we'll never know
- We could write a C interpreter
  - No reason to think ours would be better than anyone else's
  - Not worth it

Design Compromise #3

- **No attempt to generate terminating programs**
  - **Test harness uses timeouts**
  - **In practice ~10% of random programs don’t terminate within a few seconds**

Design Compromise #4

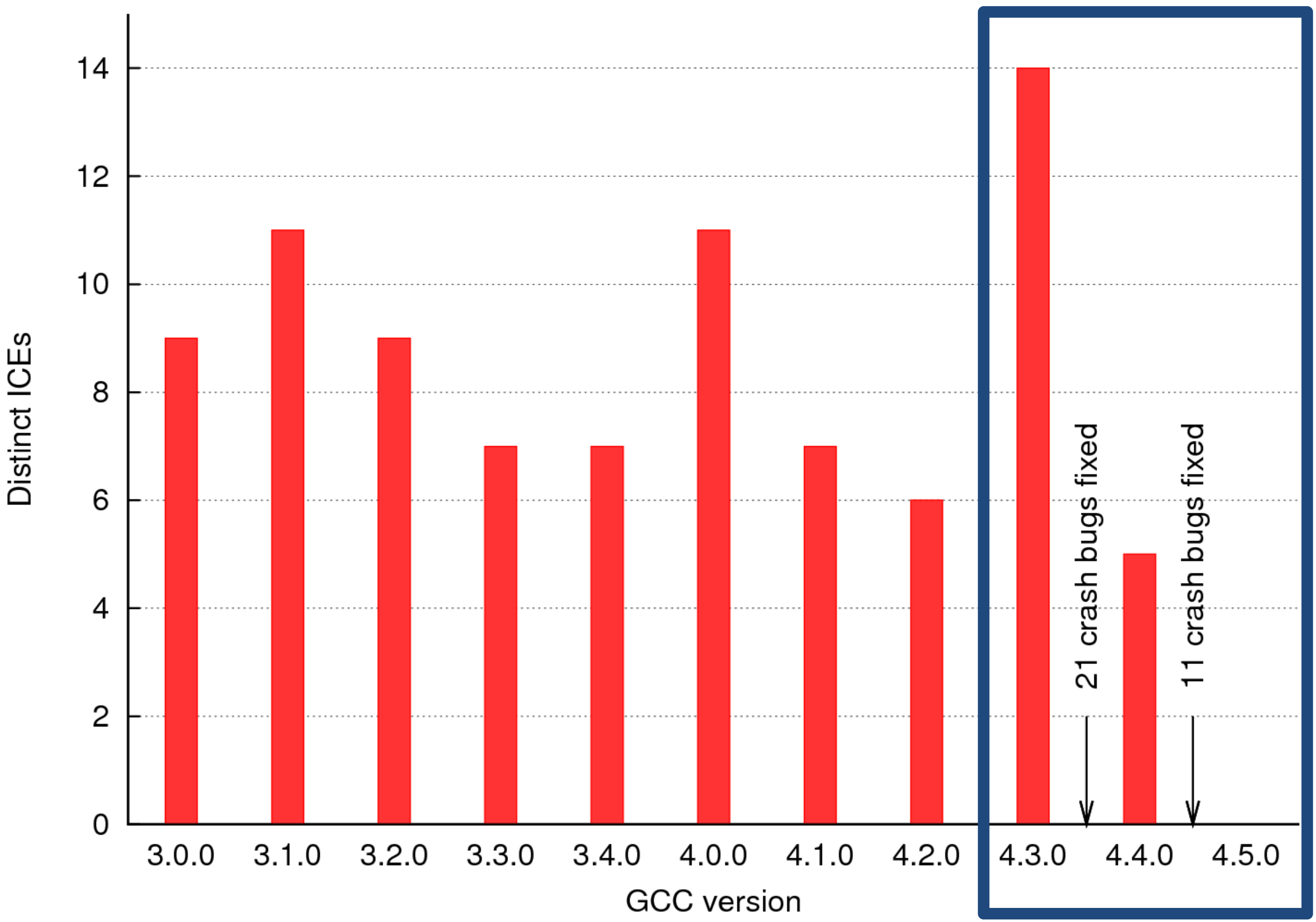
- Not aiming for coverage of the C standard
  - E.g. exceeding max identifier length
  - Existing test suites do a good job
- Goal is to find deep optimizer bugs
  - Existing test suites are insufficient

1. What we do
2. How we do it
3. What we learned
4. What still needs to happen

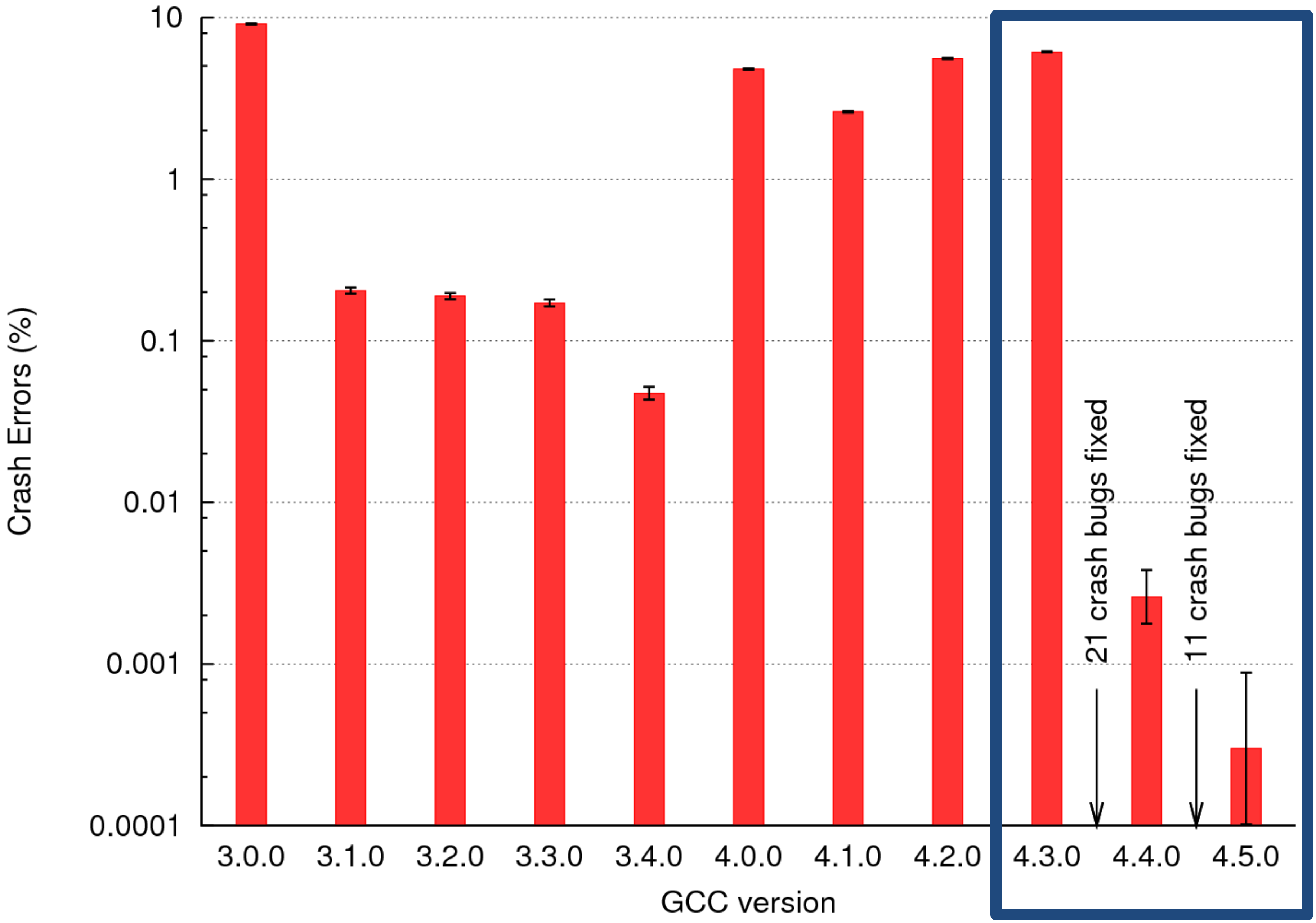
- **As expected: Higher optimization levels are buggier**
- **But sometimes a compiler is wrong…**
  - **Only at -O0**
  - **Consistently at all optimization levels**
  - **Because it was itself miscompiled**
  - **Because a system library function is wrong**
  - **Non-deterministically**
    - **Due to HW faults, ASLR, ???**

An Experiment

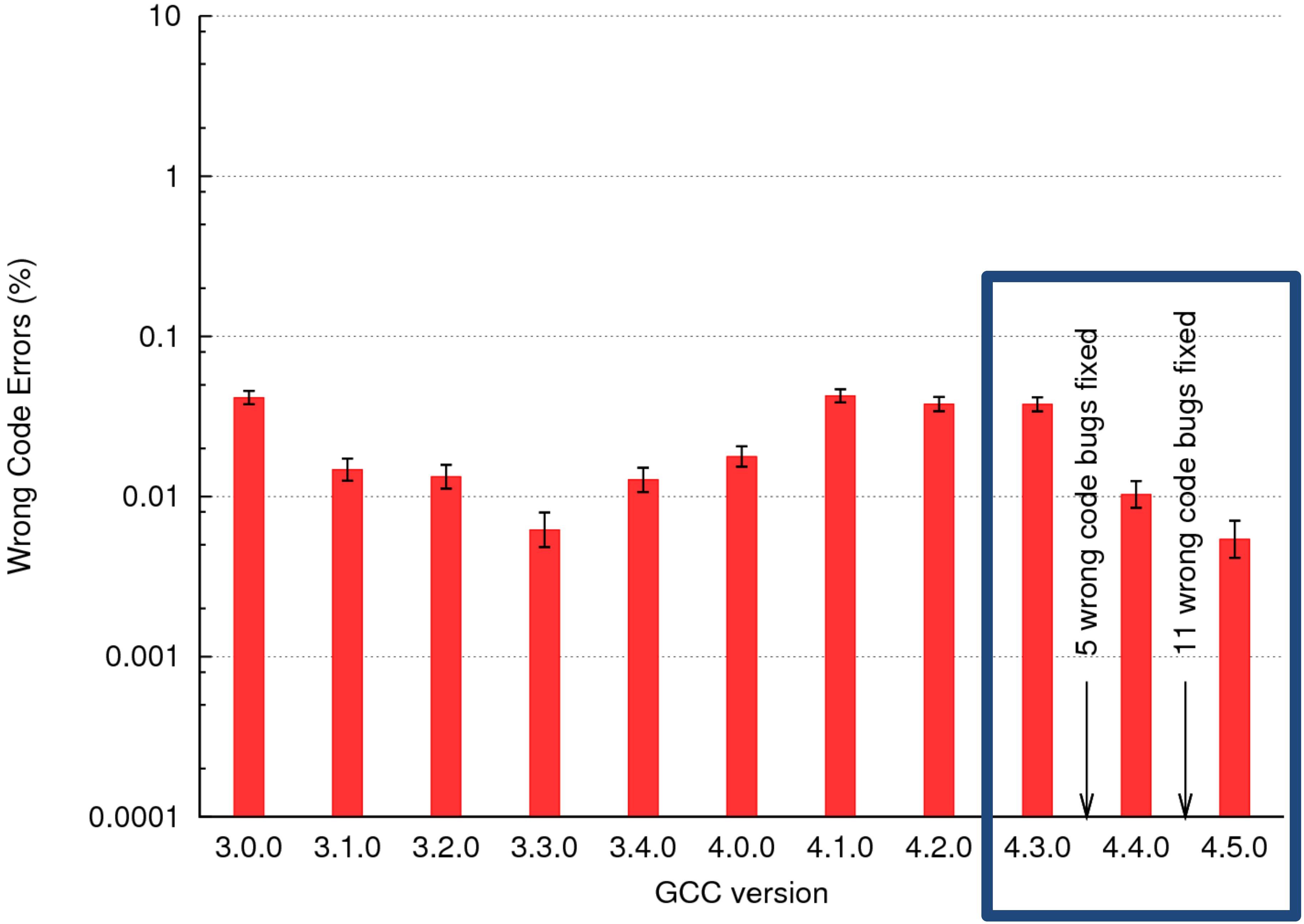
- Compiled and ran 1,000,000 random programs
- Using GCC 3.[0-4].0 and 4.[0-5].0
- -O0, -O1, -O2, -Os, -O3
- x86 only



Distinct ICEs
0
2
4
6
8
10
12
14
21 crash bugs fixed
11 crash bugs fixed
3.0.0
3.1.0
3.2.0
3.3.0
3.4.0
4.0.0
4.1.0
4.2.0
4.3.0
4.4.0
4.5.0
GCC version



10
1
0.1
0.01
0.001
0.0001
Crash Errors (%)
3.0.0
3.1.0
3.2.0
3.3.0
3.4.0
4.0.0
4.1.0
4.2.0
4.3.0
4.4.0
4.5.0
GCC version
21 crash bugs fixed
11 crash bugs fixed



10
1
0.1
0.01
0.001
0.0001
Wrong Code Errors (%)
5 wrong code bugs fixed
11 wrong code bugs fixed
3.0.0
3.1.0
3.2.0
3.3.0
3.4.0
4.0.0
4.1.0
4.2.0
4.3.0
4.4.0
4.5.0
GCC version

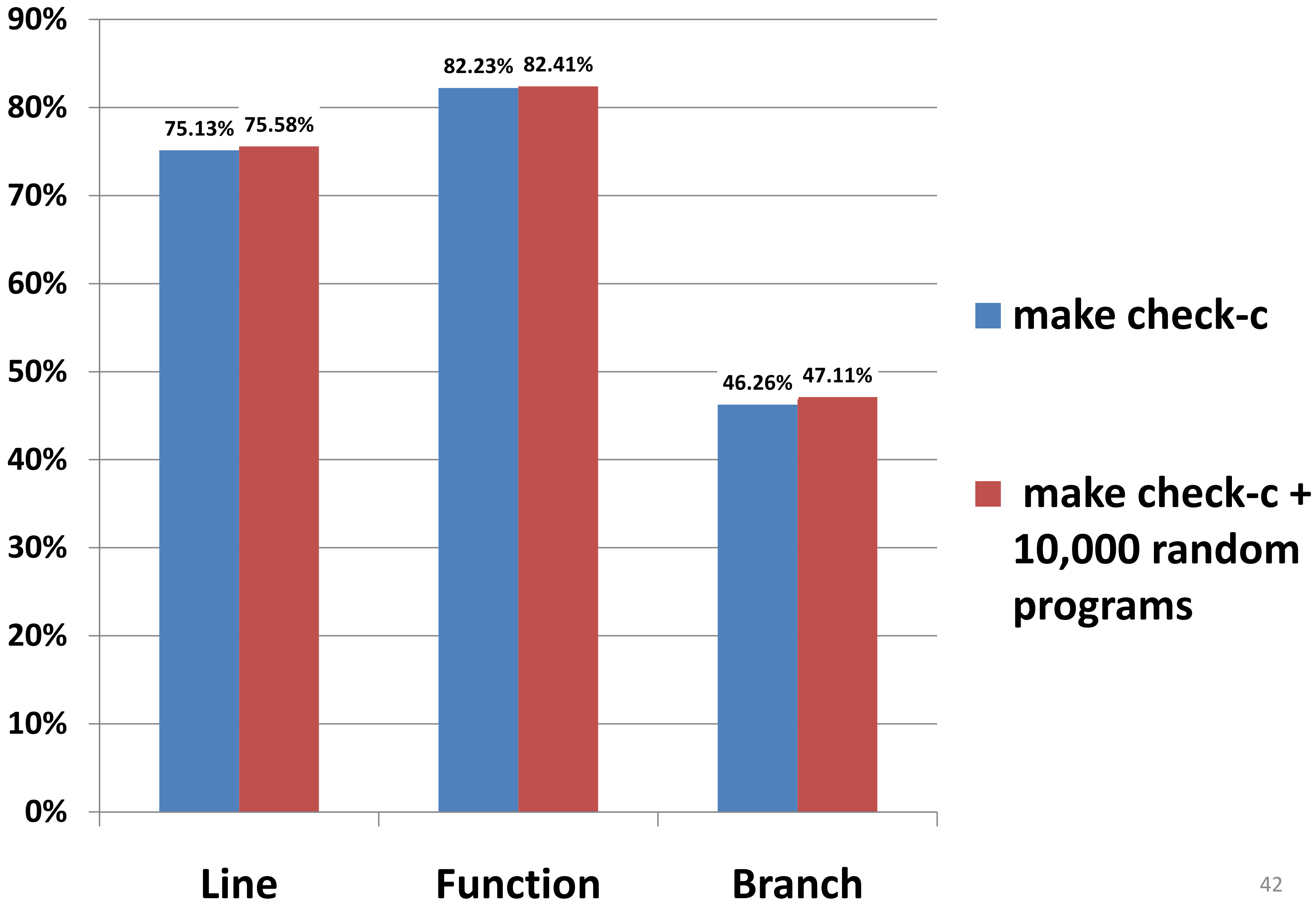
- **Fixing bugs we reported is correlated with reduction in observed error rate**
- **But is there causation?**
  - **Not enough information**
  - **This is not a controlled experiment – many bugs fixed besides the ones we reported**

Do These Bugs Matter?

- **How often do regular GCC users hit the kind of bugs we find?**
  - Several bugs we reported were subsequently re-reported by application developers
  - We sometimes find known bugs
  - But overall, not enough evidence

File	# of wrong code bugs	# of crash bugs
fold-const.c	3	6
combine.c	1	4
tree-ssa-pre.c	0	4
tree-vrp.c	0	4
tree-ssa-dce.c	0	3
tree-ssa-reassoc.c	0	2
reload1.c	1	1
tree-ssa-loop-niter.c	1	1
dse.c	2	0
tree-scalar-evolution.c	2	0
Other (12 files)	13	18
Total (22 files)	23	43

Coverage of GCC Code



1. What we do
2. How we do it
3. What we learned
4. What still needs to happen

- **We’ve only reported bugs for…**
  - A few of GCC’s platforms
  - The most basic compiler options
  - About 2 years’ worth of GCC versions
  - A subset of C
- **A lot of work remains to be done**
  - Can we push some random testing out into the community?

- **Can a casual user find and report compiler bugs using our tool?**
- **Need to…**
  - **Run the test harness – EASY**
  - **Run CPU emulators for testing cross compilers – EASY**
  - **Create reduced test cases – EASY (for ICEs)**
  - **Figure out if bugs are reported yet – EASY (for ICEs)**

- **However…**
  - **Creating reduced test cases for wrong code bugs is hard**
  - **Figuring out if a wrong code bug was already reported is hard**

- **Automation is needed**

- **Delta debugging is obvious way to reduce size of failure-inducing tests**

- **Delta debugging == Repeatedly remove part of the program and see if it remains interesting**
  - **Works well for crash bugs**
  - **Works poorly for wrong code bugs**

- **Problem: Throwing away part of a program may introduce undefined behavior**

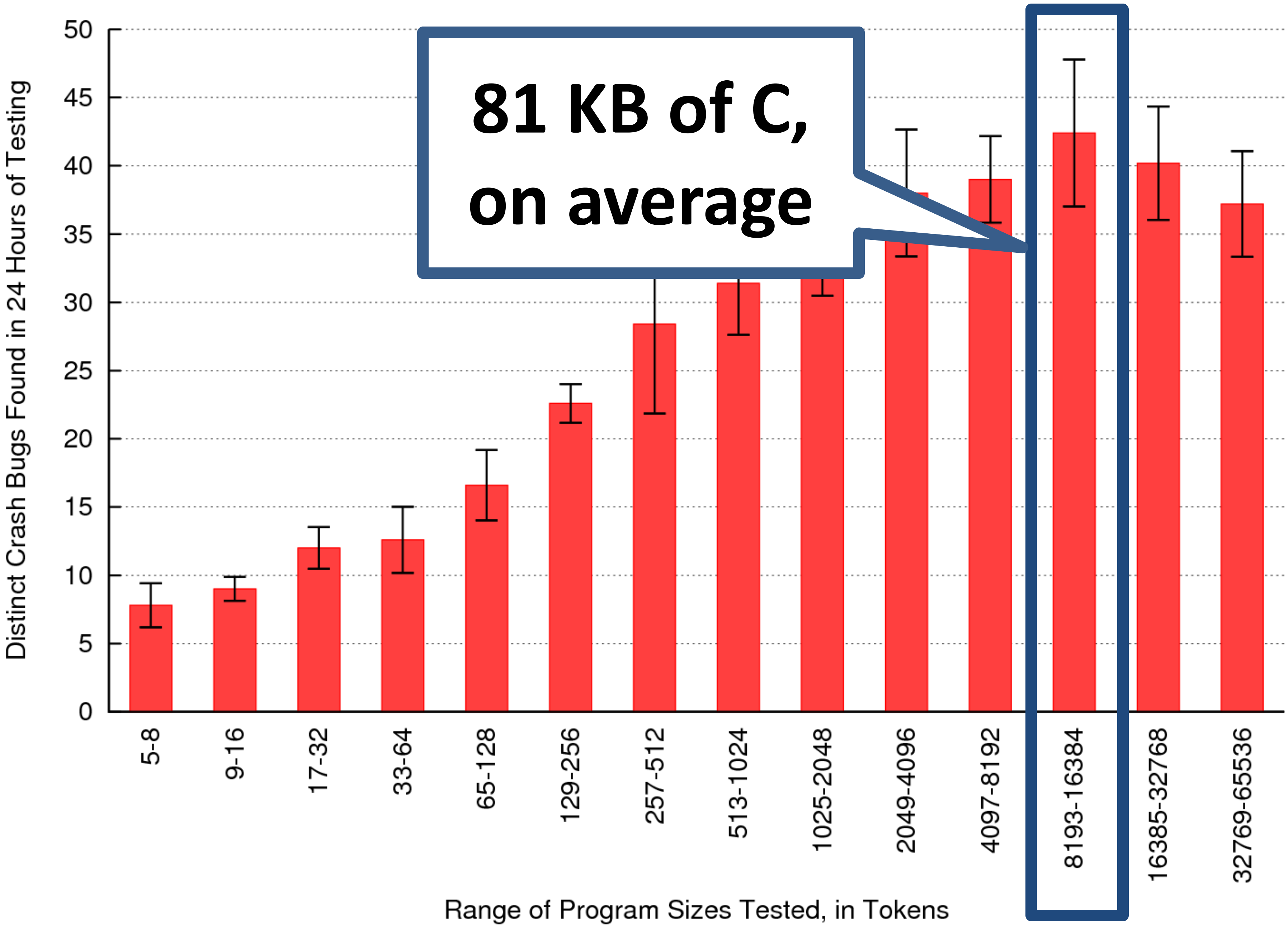
- **Example:**

```
int foo (void) {
  int x;
  x   ;
  return x;
}
```

Oops!

Possible Solutions

1. Generate small random programs
2. Detect undefined and unspecified behavior during reduction
3. Use the test case generator to reduce program size


81 KB of C, on average
Distinct Crash Bugs Found in 24 Hours of Testing
50
45
40
35
30
25
20
15
10
5
0
5-8
9-16
17-32
33-64
65-128
129-256
257-512
513-1024
1025-2048
2049-4096
4097-8192
8193-16384
16385-32768
32769-65536
Range of Program Sizes Tested, in Tokens

Possible Solutions

1. ~~Genera[illegible] programs~~
2. ~~De[illegible]undefined and un[illegible]ecified beh[illegible] during redu[illegible]~~
3. $\boxed{\text{Use the test case generator to reduce program size}}$

- **Prototype reduces size of failure-inducing test cases by 93%**
  - **Averaged over 33 wrong code bugs in GCC and LLVM**
  - **Takes a few minutes to reduce a program**
- **But given a few hours, a skilled human can do quite a bit better**

- **What if manual and automated test case reduction fails?**
  - If we cannot create a small testcase for a failure, we don't report the bug
    - Small ≈ 15 lines
  - This happens, but infrequently
  - Are we bad at testcase reduction or are there compiler bugs that only trigger on complex inputs?

- **What if an overnight run finds 500 programs that trigger wrong code bugs?**
  - **Did we just find one compiler bug or 500?**
- **If we can't answer this, we have to report 1 bug at a time**
  - **This is what we currently do**
  - **Need a way to do "bug triage"**

- **Idea for bug triage:**
  - Binary search on GCC versions to find the revision causing the bug
  - Same rev → likely same bug
  - Different rev → inconclusive!
    - Too often, bug was introduced earlier
    - Latent until exposed by some other change
- **Could also search over passes**
- **Any other ideas?**

- **TODO for us: Create a turnkey tester**
  - **Test harness needs a partial rewrite**
    - **7000 lines of Perl…**
  - **Testcase reducer needs improvement**

- **TODO for you: Please keep fixing bugs we report**
  - **Even volatile bugs**

One Last Idea

- Currently, compiler certification for critical systems is a bad joke
- Can we certify a version of GCC by
  - Restricting the set of optimization passes
  - Selecting a simple target (Thumb2 maybe)
  - Freeze features and fix bugs for a while
  - Perform near-exhaustive whitebox testing
    - Test paths in the compiler that matter

Conclusion #1

- Random testing is powerful
- But has drawbacks
  - Never know when to stop testing
  - Tuning probabilities is hard
  - Generating expressive output that is still correct is hard
  - Our generator is very C specific

Conclusion #2

- Fixed test suites are not enough
  - We find bugs other testing misses
  - We can auto-generate reduced testcases

Conclusion #3

- Our work is the most extensive fuzz attack on compilers to date
  - Quickly finds bugs in every compiler we've tested

- Compilers need random testing

Code Coverage Backup Slide

- **make check-c**
  - Lines : 75.13% (246876 / 328609)
  - Functions : 82.23% (15292 / 18596)
  - Branches : 46.26% (243658 / 526724)

- **make check-c + 10,000 random programs**
  - Lines : 75.58% (248358 / 328609)
  - Functions : 82.41% (15325 / 18596)
  - Branches : 47.11% (248129 / 526724)