Hardening LLVM with Random Testing

Xuejun Yang, Yang Chen

Eric Eide, John Regehr

{jxyang, chenyang, eeide, regehr}@cs.utah.edu

School of Computing
University of Utah

A LLVM Crash Bug

```
…
int * p[2];
int i;
for (...) {
   p[i] = &i;
   …
}
```

- LLVM crashed due to an over-aggressive assertion in Loop Invariant Code Motion in a 2.8 revision

- For some reason it thought the address-taken variable “i” must not be part of expression on LHS

Some LLVM Wrong Code Bugs

- A 2.4 revision simplifies "(a > 13) & (a == 15)" into "(a > 13)“

- A 2.8 revision folds "((x == x) | (y & z)) == 1” into 0
  - thought sub-expression (y & z) is constant integer

- A 2.8 revision reduces this loop into “i = 1”

```
for (i = 0; i < 5; i++)
{
    if (i) continue;
    if (i) break;
}
```

Tally of Bugs

From March, 2008 to present:

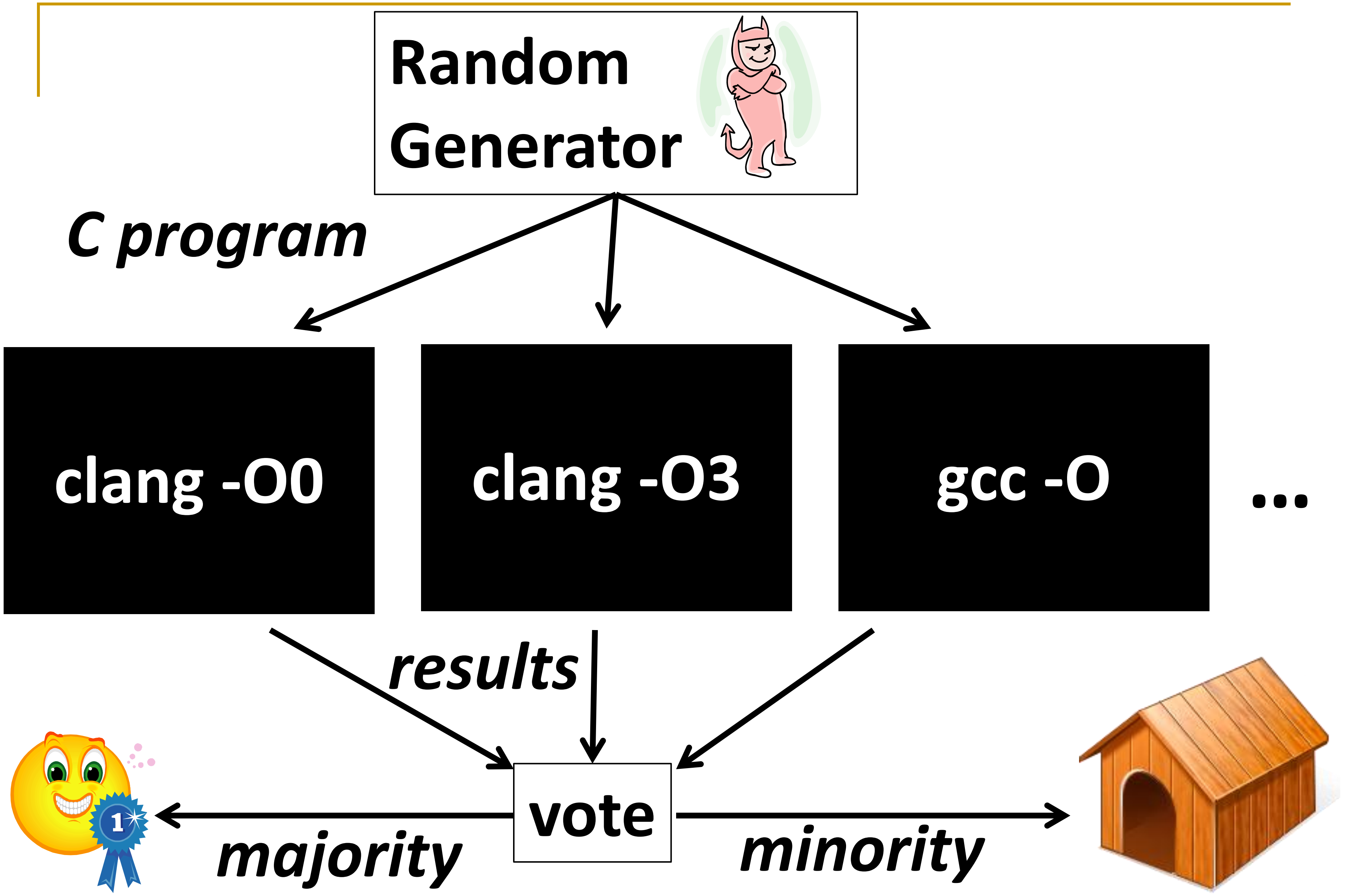
- 170 bugs fixed + 3 reported but not yet fixed
  - 57 wrong code bugs
  - 116 crash bugs

- Account for 2.6% of total valid bugs filed against LLVM in that period

Goal

- Use random differential testing to find LLVM bugs!

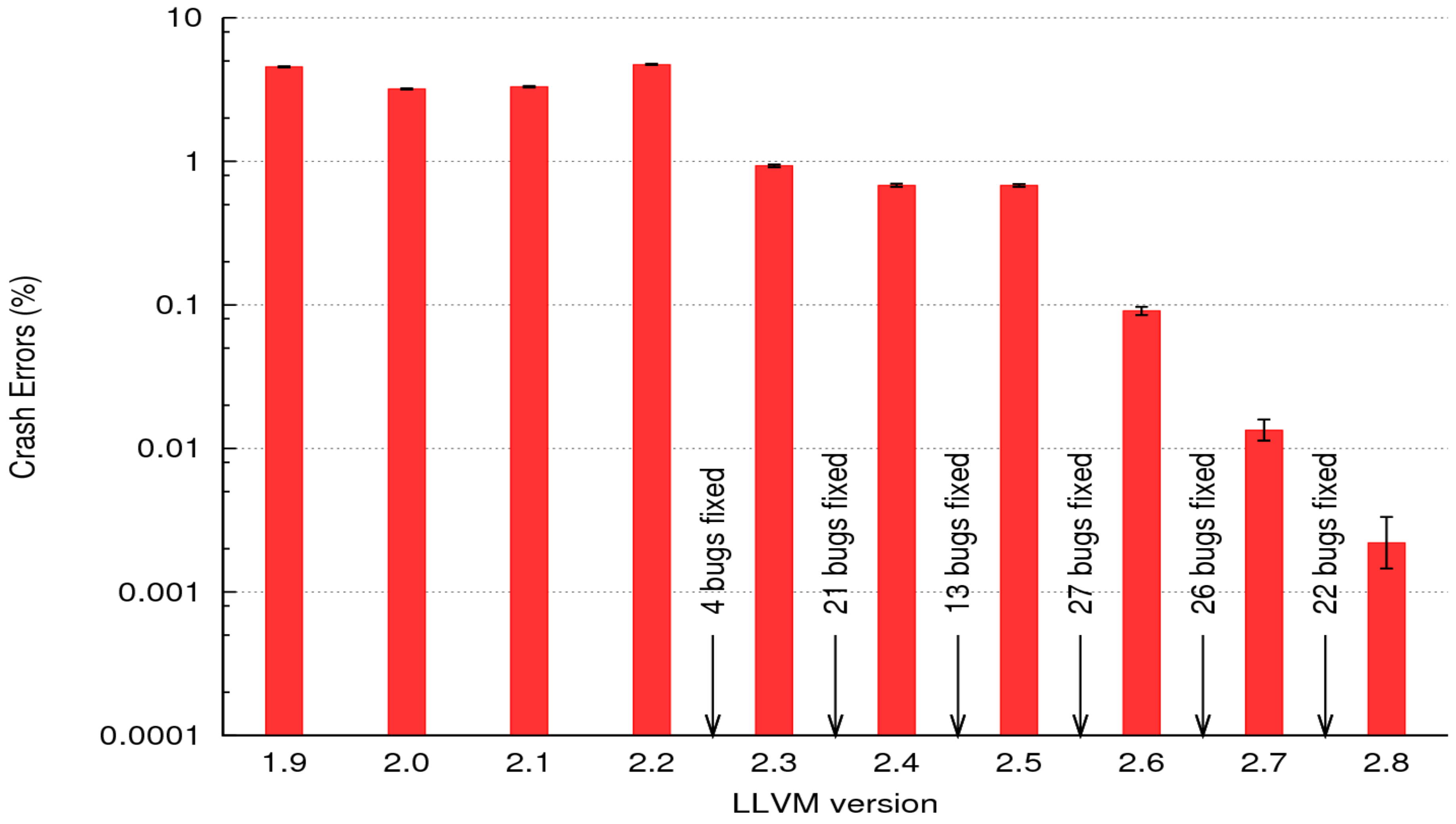
  - ✓ Aimed at deep optimization bugs
  - ✓ Systematically
    - ✓ Find bugs,
    - ✓ Report bugs
  - ✓ Automate the process as much as we can
  - ✓ **Ultimate goal: improve LLVM quality**


Random Generator
C program
clang -O0
clang -O3
gcc -O
...
results
vote
majority
minority

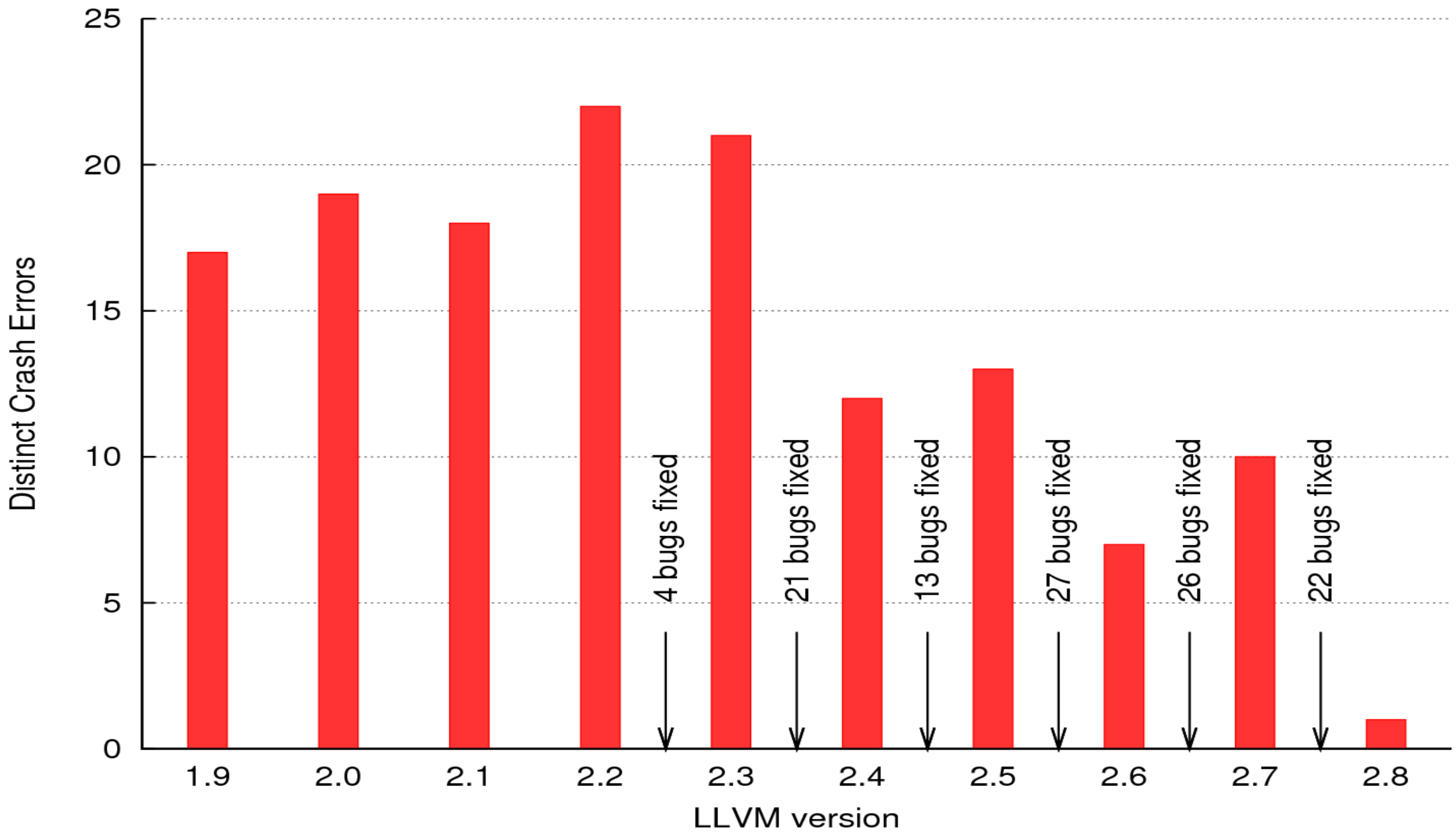
1. What we do
2. What we learned
3. How we do it

An Experiment

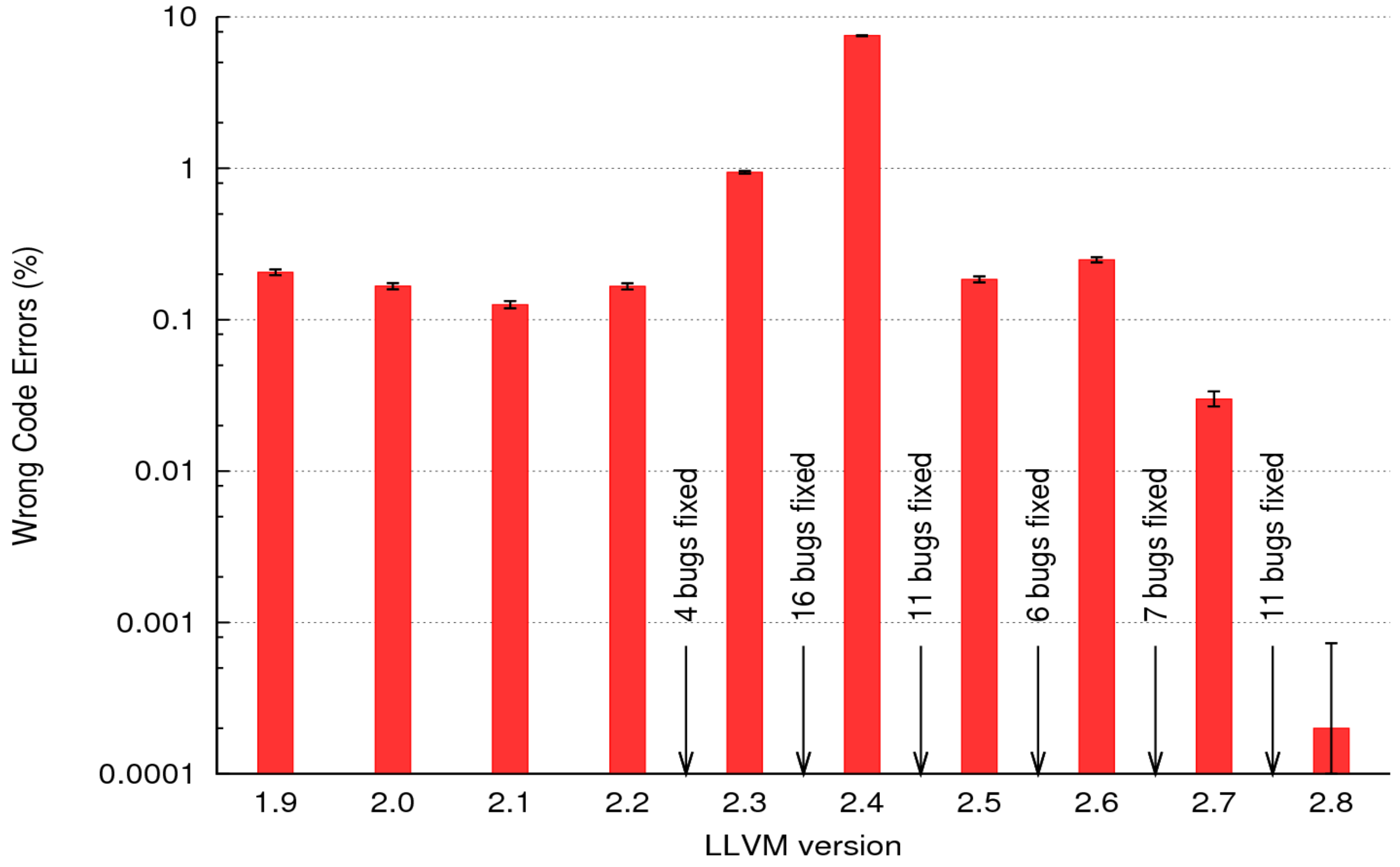
- Compiled and ran 1,000,000 random programs
- Using LLVM 1.9 – 2.8
- -O0, -O1, -O2, -Os, -O3
- x86 only



Crash Error is the percentage of crashes triggered by 1,000,000 random test cases

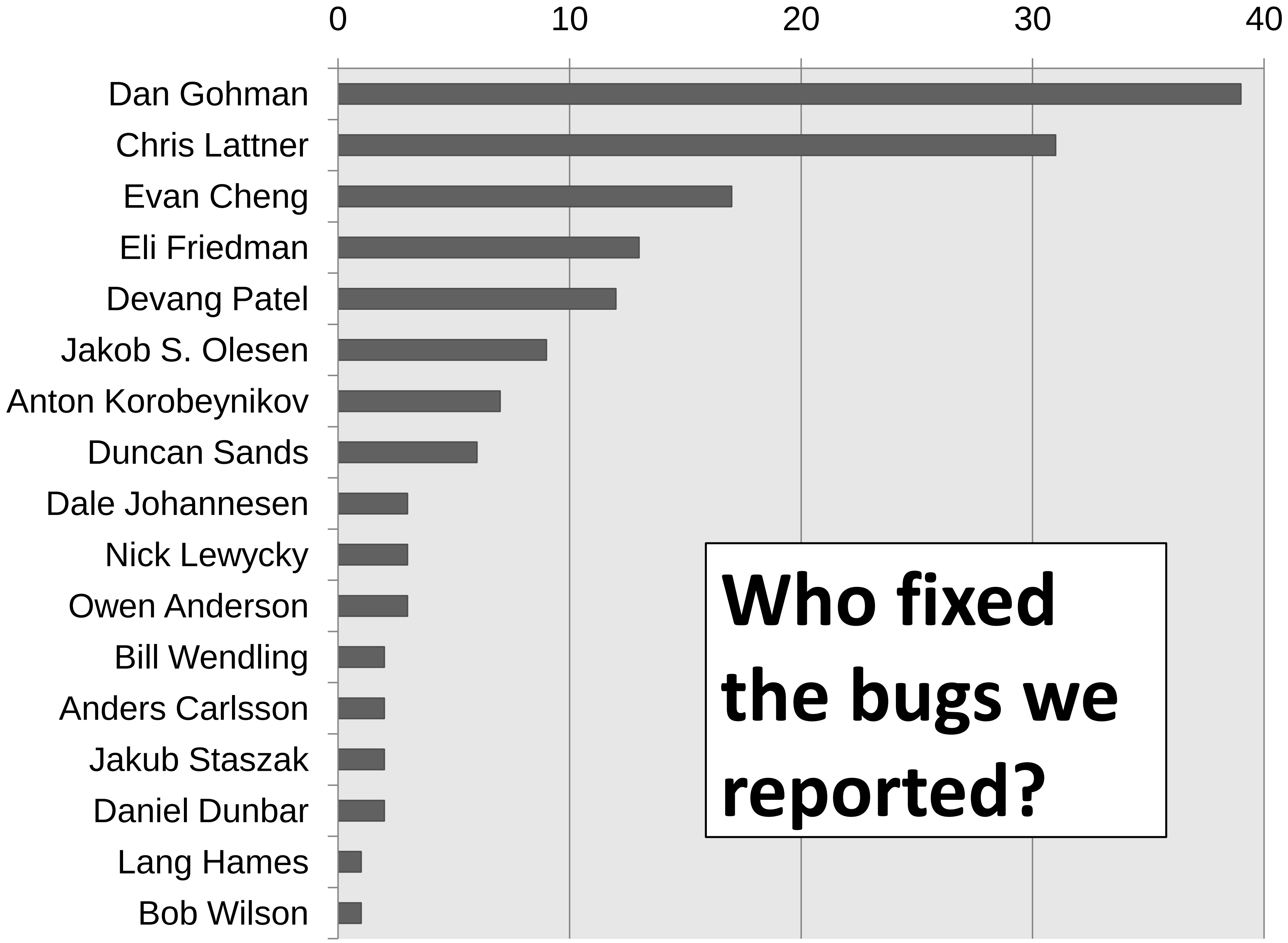


Distinct crash Errors is the number of distinct crashes triggered by 1,000,000 random test cases



Wrong Code Error Rate is the percentage of wrong code errors found by 1,000,000 random test cases

Who fixed the bugs we reported?



Bug Distribution Across Stages

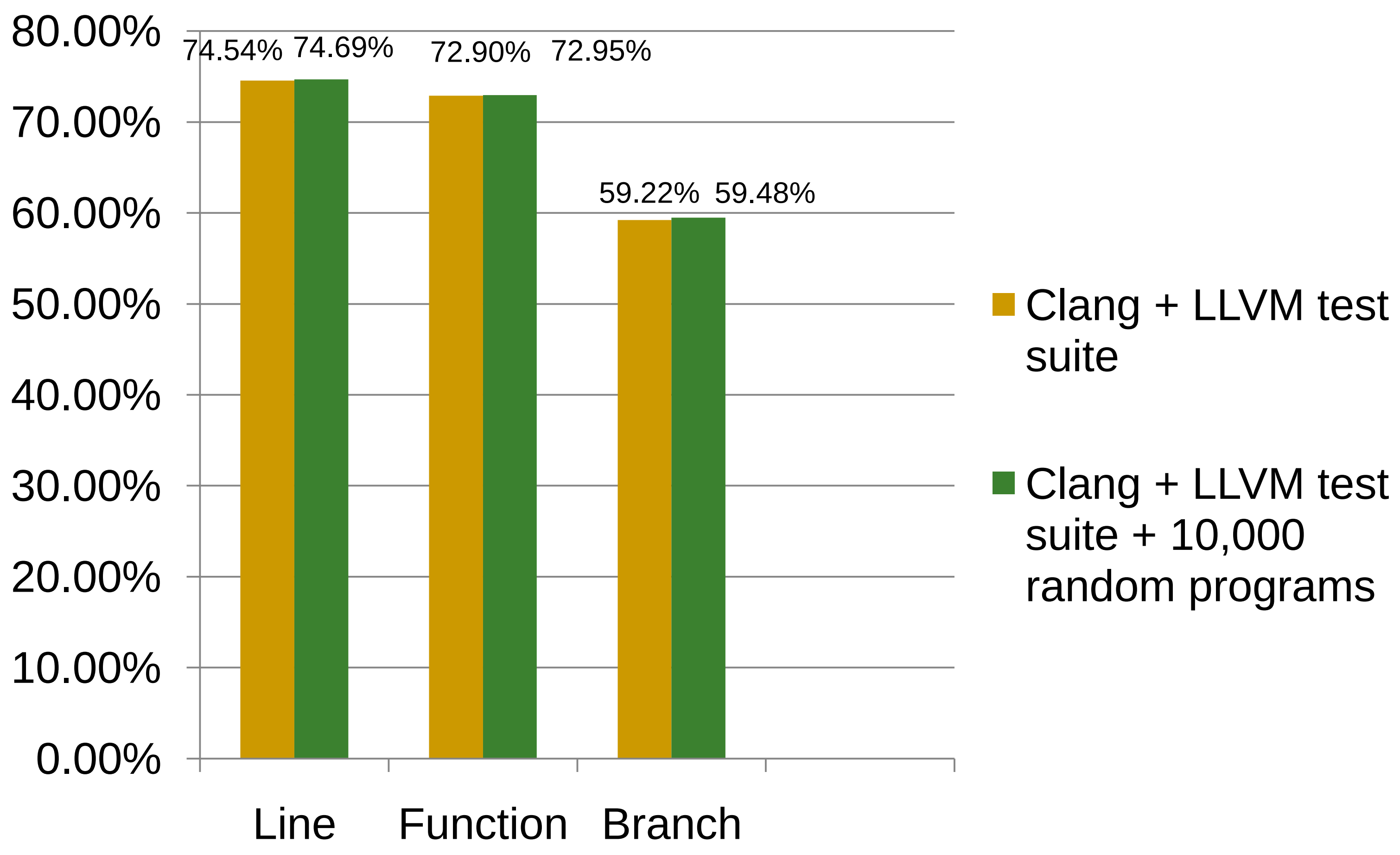
frontend	10
middle-end	61
backend	67
Can’t identify	35
total	173

- Classification is based on bugzilla records and/or source files committed for bug fixes
- Source files under Clang or gcc are considered frontend
- Source files under Analysis and Transforms are considered middle-end
- Source files under CodeGen and Target are considered backend

Bug Distribution Across Files

Cpp file	**Bugs found**
InstructionCombining	13
SimpleRegisterCoalescing	10
DAGCombiner	6
LoopUnswitch	5
LoopStrengthReduce	4
JumpThreading	4
LICM	4
FastISel	4
llvm-convert	4
ExprConstant	4
X86InstrInfo	3
X86InstrInfo.td	3
X86ISelDAGToDAG	3
BranchFolding	3
… (and many more)	

Increased Coverage by Random Programs



1. **What we do**
2. **What we learned**
3. **How we do it**

Beyond a Random Generator

- Ability to create random programs with unambiguous meanings
    - ✓ undefined behaviors
    - ✓ unspecified behaviors

- Ability to check the meaning of a C program is preserved by the compiler
    - ✓ Summarize the "meaning"
        - the checksum of global variables
    - ✓ Test harness: automatically detect checksum discrepancies

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

$ clang -O2 foo.c -o foo
$ ./foo
1
```

Not a Bug #2

```
int a;

void bar (int a, int b) {
}

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
```

Two Goals We Must Balance

- Be expressive on code generation

  - is easy if you don't care about undefined / unspecified behavior

- Avoid undefined behaviors and not depend on unspecified behaviors

  - is easy if you don't care about expressiveness

Expressive code that avoids undefined / unspecified behavior is hard

Design Compromises

- Implementation-defined behavior is allowed
  - Avoiding it is too restrictive
  - We cannot do differential testing of x86 Clang vs. MSP430 Clang
- No ground truth
  - If all compilers generate the same wrong answer, we'll never know
  - Generating self-checking random code restricts expressiveness
- No attempt to generate terminating programs
  - Test harness uses timeouts
  - In practice ~10% of random programs don't terminate within a few seconds

Supported features:

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

Not supported:

- Side-effecting expressions
- Comma operator
- Interesting type casts
- Strings
- Unions
- Floating point
- Nontrivial C++
- Nonlocal jumps
- Varargs
- Recursive functions
- Function pointers
- Dynamic memory alloc

Avoid Undefined Behaviors With “Extra” Code

- Use before initialization: declare all variables with explicit initializers
- Array OOB access: force all indices within bound using modulus
- Null pointer deference: check nullness before dereference

Undefined Integer 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

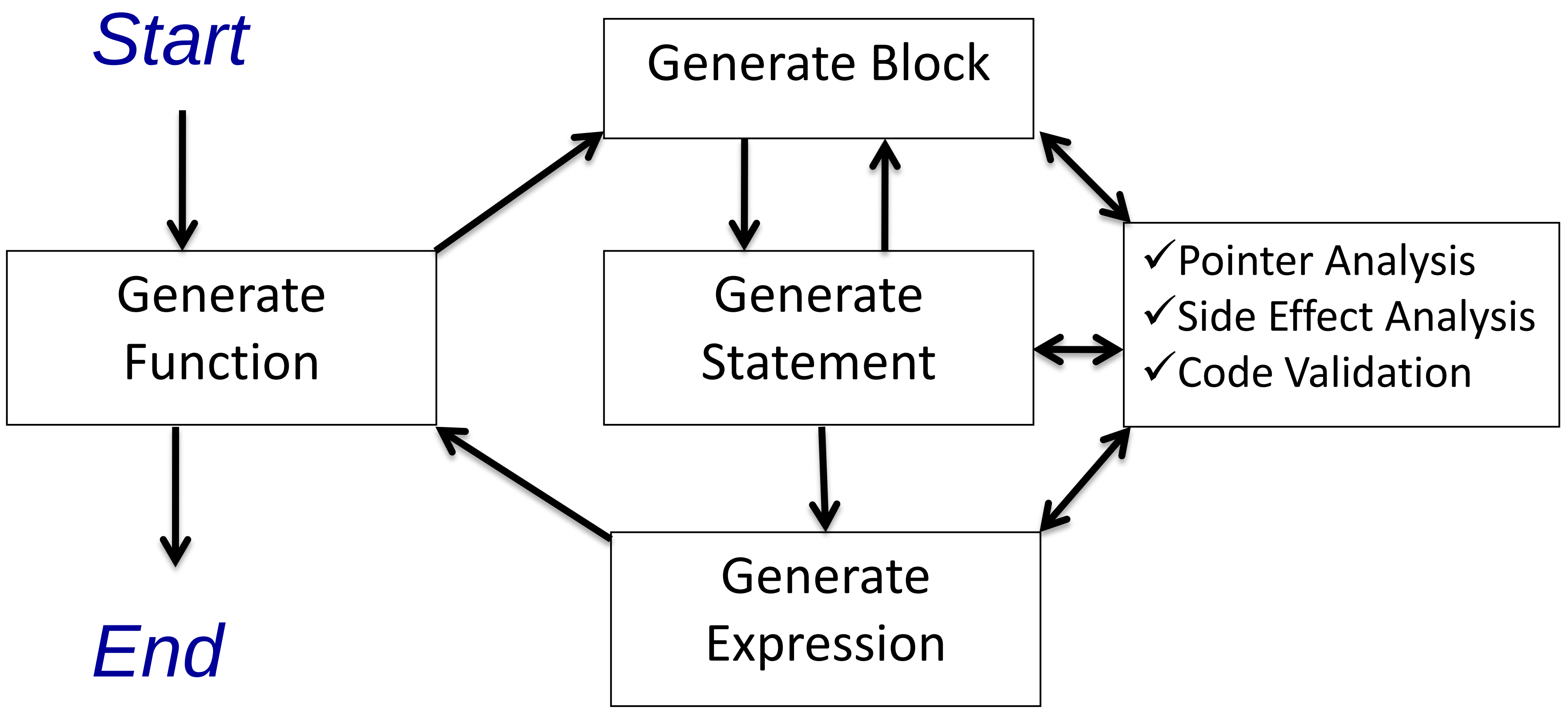
```
int safe_signed_left_shift (int si1, int si2) {
  if (si1 < 0 ||
      si2 < 0 ||
      si2 >= sizeof(int) ||
      si1 > (INT_MAX >> si2)){
    return si1;
  } else {
    return si1 << si2;
  }
}
```

Avoid Undefined/Unspecified Behaviors Statically

Some undefined / unspecified are too difficult to avoid structurally

- Dereference dangling pointers
- Order of evaluation problem:
  -*Two expressions writes to same variable*

Code Generator Augmented with Static Analyzer



Future work

- For us:
    - Create a turnkey system :
        - Test harness needs a partial rewrite (7000 lines of Perl)
    - Improve test case reduction
    - Support more C features
    - Support more languages and formats (including IR)

- For you:
    - Please keep fixing bugs we report
    - Tell us where random testing might be useful

Conclusion

- Random testing is powerful
  - Quickly found bugs in all compilers we tested
  - Found deep optimization bugs unlikely to be uncovered by other means
  - Can generate small test cases pinpoint the problem
  - But has limitations:
    - Never know when to stop testing
    - Coverage is not broad enough

- Compilers need random testing
  - Fixed test suite is not enough

Thank you …

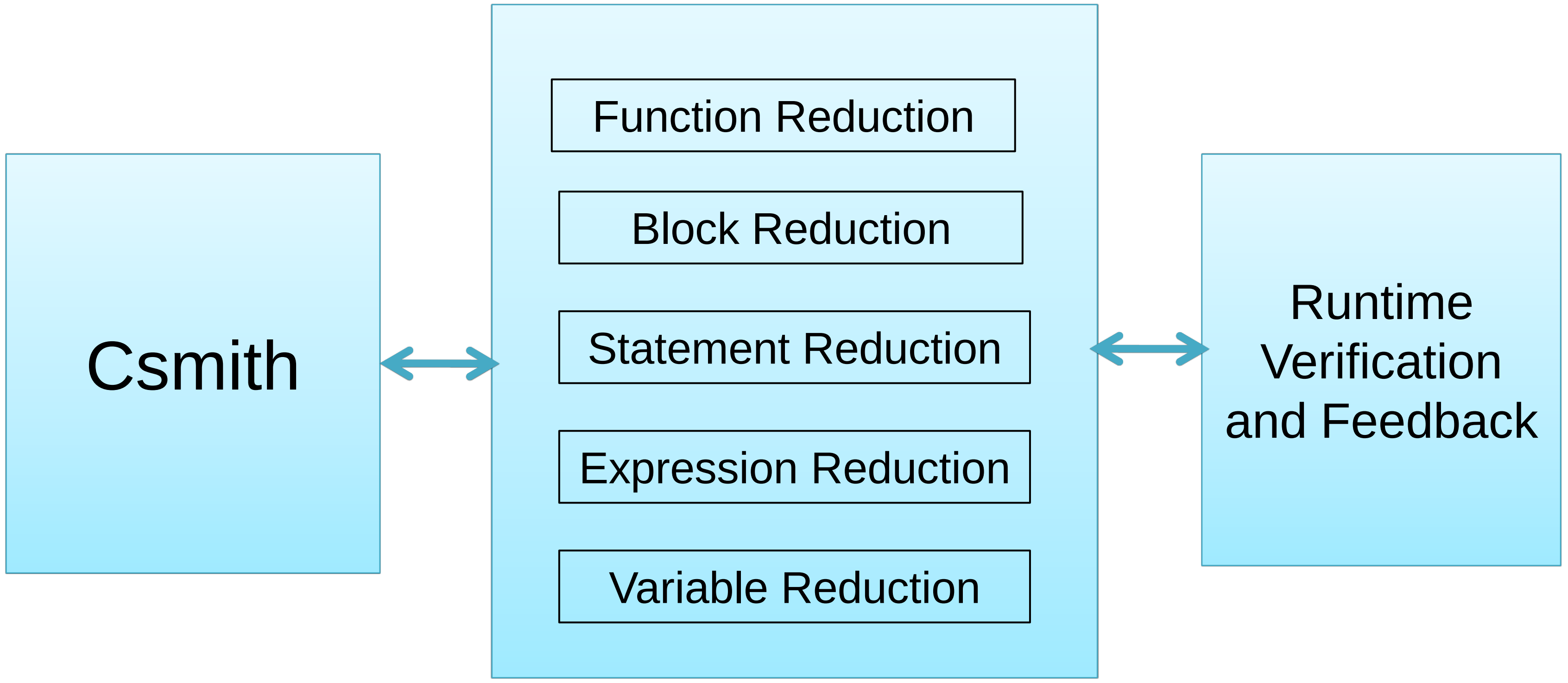
And thank Qualcomm for sponsoring my trip!

Questions?

Test Case Reduction

- Is Necessary for
  - Bug reporting
  - Bug fixing
  - Allow causal users find and report bugs using our tool
- Existing approach: 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

Hierarchical Reduction



✓ Initial results show reduced size of failure-inducing test cases by **93%** on average in a few minutes