Compiler Bugs

"Finding and Understanding Bugs in C Compilers"

X. Yang, Y. Chen, E. Eide, J. Regehr

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2011

Example

```
1    int foo (void) {
2      signed char x = 1;
3      unsigned char y = 255;
4      return x > y;
5    }
```

Figure 1. We found a bug in the version of GCC that shipped with Ubuntu Linux 8.04.1 for x86. At all optimization levels it compiles this function to return 1; the correct result is 0. The Ubuntu compiler was heavily patched; the base version of GCC did not have this bug.

Results

For the past three years, we have used Csmith to discover bugs in C compilers. Our results are perhaps surprising in their extent: to date, we have found and reported more than 325 bugs in mainstream C compilers including GCC, LLVM, and commercial tools. Figure 1 shows a representative example. Every compiler that we have tested, including several that are routinely used to compile safety-critical embedded systems, has been crashed and also shown to silently miscompile valid inputs. As measured by the responses to our bug reports, the defects discovered by Csmith are important. Most of the bugs we have reported against GCC and LLVM have been fixed. Twenty-five of our reported GCC bugs have been classified as P1, the maximum, release-blocking priority for GCC defects. Our results suggest that fixed test suites—the main way that compilers are tested—are an inadequate mechanism for quality control.

Common Reasons

We claim that Csmith is an effective bug-finding tool in part because it generates tests that explore atypical combinations of C language features. Atypical code is *not* unimportant code, however; it is simply underrepresented in fixed compiler test suites. Developers who stray outside the well-tested paths that represent a compiler's "comfort zone"—for example by writing kernel code or embedded systems code, using esoteric compiler options, or automatically generating code—can encounter bugs quite frequently. This is a significant problem for complex systems. Wolfe [30], talking about independent software vendors (ISVs) says: "An ISV with a complex code can work around correctness, turn off the optimizer in one or two files, *and usually they have to do that for any of the compilers they use*" (emphasis ours).

Basic Idea

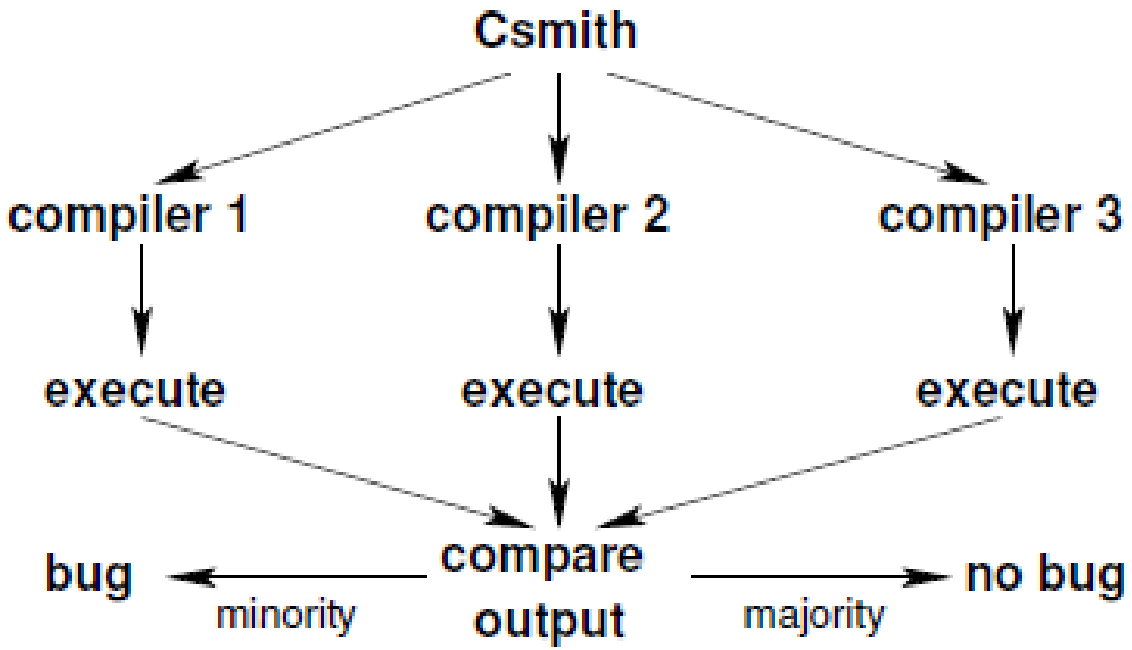

Figure 2. Finding bugs in three compilers using randomized differential testing

Randomly-Generated Programs

Csmith creates programs with the following features:

- function definitions, and global and local variable definitions
- most kinds of C expressions and statements
- control flow: `if/else`, function calls, `for` loops, `return`, `break`, `continue`, `goto`
- signed and unsigned integers of all standard widths
- arithmetic, logical, and bitwise operations on integers
- structs: nested, and with bit-fields
- arrays of and pointers to all supported types, including pointers and arrays
- the `const` and `volatile` type qualifiers, including at different levels of indirection for pointer-typed variables

More Examples of Bugs

3.7 Examples of Wrong-Code Bugs

This section characterizes a few of the bugs that were revealed by miscompilation of programs generated by Csmith. These bugs fit into a simple model in which optimizations are structured like this:

```
analysis
if (safety check) {
  transformation
}
```

An optimization can fail to be semantics-preserving if the analysis is wrong, if the safety check is insufficiently conservative, or if the transformation is incorrect. The most common root cause for bugs that we found was an incorrect safety check.

More Examples of Bugs

GCC Bug #1: wrong safety check[4] If `x` is variable and `c1` and `c2` are constants, the expression `(x/c1)!=c2` can be profitably rewritten as `(x-(c1*c2))>(c1-1)`, using unsigned arithmetic to avoid problems with negative values. Prior to performing the transformation, expressions such as `c1*c2` and `(c1*c2)+(c1-1)` are checked for overflow. If overflow occurs, further simplifications can be made; for example, `(x/1000000000)!=10` always evaluates to 0 when `x` is a 32-bit integer. GCC falsely detected overflow for some choices of constants. In the failure-inducing test case that we discovered, `(x/-1)!=1` was folded to 0. This expression should evaluate to 1 for many values of `x`, such as 0.

More Examples of Bugs

GCC Bug #2: wrong transformation[5] In C, if an argument of type `unsigned char` is passed to a function with a parameter of type `int`, the values seen inside the function should be in the range 0..255. We found a case in which a version of GCC inlined this kind of function call and then sign-extended the argument rather than zero-extending it, causing the function to see negative values of the parameter when the function was called with arguments in the range 128..255.

More Examples of Bugs

GCC Bug #3: wrong analysis[6] We found a bug that caused GCC to miscompile this code:

```
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 }
```

The generated code returned 1 instead of 0. The problem occurred when the compiler failed to recognize that p and q are aliases; this happened because q was mistakenly identified as a read-only memory location, which is defined not to alias a mutable location. The wrong not-alias fact caused the store in line 7 to be marked as dead so that a subsequent dead-store elimination pass removed it.

More Examples of Bugs

GCC Bug #4: wrong analysis[7] A version of GCC miscompiled this function:

```
1 int x = 4;
2 int y;
3
4 void foo (void) {
5   for (y = 1; y < 8; y += 7) {
6     int *p = &y;
7     *p = x;
8   }
9 }
```

When `foo` returns, `y` should be 11. A loop-optimization pass determined that a temporary variable representing `*p` was invariant with value `x+7` and hoisted it in front of the loop, while retaining a dataflow fact indicating that `x+7 == y+7`, a relationship that no longer held after code motion. This incorrect fact lead GCC to generate code leaving 8 in `y`, instead of 11.

More Examples of Bugs

LLVM Bug #1: wrong safety check[8] `(x==c1)||(x<c2)` can be simplified to `x<c2` when `c1` and `c2` are constants and `c1<c2`. An LLVM version incorrectly transformed `(x==0)||(x<-3)` to `x<-3`. LLVM did a comparison between 0 and −3 in the safety check for this optimization, but performed an unsigned comparison rather than a signed one, leading it to incorrectly determine that the transformation was safe.

LLVM Bug #2: wrong safety check[9] `(x|c1)==c2` evaluates to 0 if `c1` and `c2` are constants and `(c1&~c2)!=0`. In other words, if any bit that is set in `c1` is unset in `c2`, the original expression cannot be true. A version of LLVM contained a logic error in the safety check for this optimization, wrongly replacing this kind of expression with 0 even when `c1` was not a constant.

Testing a Verified Compiler

Testing CompCert CompCert [14] is a verified, optimizing compiler for a large subset of C; it targets PowerPC, ARM, and x86. We put significant effort into testing this compiler.

The first silent wrong-code error that we found in CompCert was due to a miscompilation of this function:

```
1    int bar (unsigned x) {
2      return -1 <= (1 && x);
3    }
```

CompCert 1.6 for PowerPC generates code returning 0, but the proper result is 1 because the comparison is signed. This bug and five others like it were in CompCert's unverified front-end code. Partly in response to these bug reports, the main CompCert developer expanded the verified portion of CompCert to include C's integer promotions and other tricky implicit casts.

Testing a Verified Compiler

The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.