Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. In this paper we present our compiler-testing tool and the results of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the undefined and unspecified behaviors that would destroy its ability to automatically find wrong-code bugs. Our second contribution is a collection of qualitative and quantitative results about the bugs we have found in open-source C compilers.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; D.3.2 [*Programming Languages*]: Language Classifications—C; D.3.4 [*Programming Languages*]: Processors—compilers

General Terms Languages, Reliability

Keywords compiler testing, compiler defect, automated testing, random testing, random program generation

1. Introduction

The theory of compilation is well developed, and there are compiler frameworks in which many optimizations have been proved correct. Nevertheless, the practical art of compiler construction involves a morass of trade-offs between compilation speed, code quality, code debuggability, compiler modularity, compiler retargetability, and other goals. It should be no surprise that optimizing compilers—like all complex software systems—contain bugs.

Miscompilations often happen because optimization safety checks are inadequate, static analyses are unsound, or transformations are flawed. These bugs are out of reach for current and future automated program-verification tools because the specifications that need to be checked were never written down in a precise way, if they were written down at all. Where verification is impractical, however, other methods for improving compiler quality can succeed. This paper reports our experience in using testing to make C compilers better.

The definitive version was published in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, Jun. 2011, http://doi.acm.org/10.1145/1993498.1993532

```
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.

We created Csmith, a randomized test-case generator that supports compiler bug-hunting using differential testing. Csmith generates a C program; a test harness then compiles the program using several compilers, runs the executables, and compares the outputs. Although this compiler-testing approach has been used before [6, 16, 23], Csmith's test-generation techniques substantially advance the state of the art by generating random programs that are expressive—containing complex code using many C language features—while also ensuring that every generated program has a single interpretation. To have a unique interpretation, a program must not execute any of the 191 kinds of undefined behavior, nor depend on any of the 52 kinds of unspecified behavior, that are described in the C99 standard.

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.

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). As another example, the front

page of the Web site for GMP, the GNU Multiple Precision Arithmetic Library, states, "Most problems with compiling GMP these days are due to problems not in GMP, but with the compiler."

Improving the correctness of C compilers is a worthy goal: C code is part of the trusted computing base for almost every modern computer system including mission-critical financial servers and life-critical pacemaker firmware. Large-scale source-code verification efforts such as the seL4 OS kernel [12] and Airbus's verification of fly-by-wire software [24] can be undermined by an incorrect C compiler. The need for correct compilers is amplified because operating systems are almost always written in C and because C is used as a portable assembly language. It is targeted by code generators from a wide variety of high-level languages including Matlab/Simulink, which is used to generate code for industrial control systems.

Despite recent advances in compiler verification, testing is still needed. First, a verified compiler is only as good as its specification of the source and target language semantics, and these specifications are themselves complex and error-prone. Second, formal verification seldom provides end-to-end guarantees: "details" such as parsers, libraries, and file I/O usually remain in the trusted computing base. This second point is illustrated by our experience in testing CompCert [14], a verified C compiler. Using Csmith, we found previously unknown bugs in unproved parts of CompCert—bugs that cause this compiler to silently produce incorrect code.

Our goal was to discover serious, previously unknown bugs:

- in mainstream C compilers like GCC and LLVM;
- that manifest when compiling core language constructs such as arithmetic, arrays, loops, and function calls;
- targeting ubiquitous architectures such as x86 and x86-64; and
- using mundane optimization flags such as –O and –O2.

This paper reports our experience in achieving this goal. Our first contribution is to advance the state of the art in compiler test-case generation, finding—as far as we know—many more previously unknown compiler bugs than any similar effort has found. Our second contribution is to qualitatively and quantitatively characterize the bugs found by Csmith: What do they look like? In what parts of the compilers are they primarily found? How are they distributed across a range of compiler versions?

2. Csmith

Csmith began as a fork of Randprog [27], an existing random C program generator about 1,600 lines long. In earlier work, we extended and adapted Randprog to find bugs in C compilers' translation of accesses to volatile-qualified objects [6], resulting in a 7,000-line program. Our previous paper showed that in many cases, these bugs could be worked around by turning volatile-object accesses into calls to helper functions. The key observation was this: while the rules regarding the addition, elimination, and reordering of accesses to volatile objects are not at all like the rules governing ordinary variable accesses in C, they are almost identical to the rules governing function calls.

For some test programs generated by Randprog, our rewriting procedure was insufficient to correct a defect that we had found in the C compiler. Our hypothesis was that this was always due to "regular" compiler bugs not related to the volatile qualifier. To investigate these compiler defects, we shifted our research emphasis toward looking for generic wrong-code bugs. We turned Randprog into Csmith, a 40,000-line C++ program for randomly generating C programs. Compared to Randprog, Csmith can generate C programs that utilize a much wider range of C features including complex control flow and data structures such as pointers, arrays, and structs. Most of Csmith's complexity arises from the requirement that it interleave static analysis with code generation in order to produce meaningful test cases, as described below.

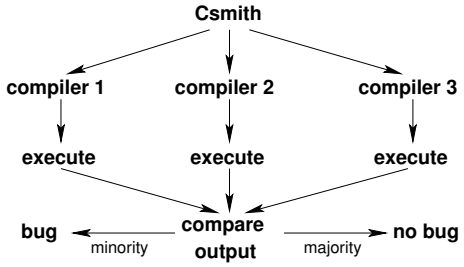

Figure 2. Finding bugs in three compilers using randomized differential testing

2.1 Randomized Differential Testing using Csmith

Random testing [9], also called fuzzing [17], is a black-box testing method in which test inputs are generated randomly. Randomized differential testing [16] has the advantage that no oracle for test results is needed. It exploits the idea that if one has multiple, deterministic implementations of the same specification, all implementations must produce the same result from the same valid input. When two implementations produce different outputs, one of them must be faulty. Given three or more implementations, a tester can use voting to heuristically determine which implementations are wrong. Figure 2 shows how we use these ideas to find compiler bugs.

2.2 Design Goals

Csmith has two main design goals. First and most important, every generated program must be well formed and have a single meaning according to the C standard. The meaning of a C program is the sequence of side effects it performs. The principal side effect of a Csmith-generated program is to print a value summarizing the computation performed by the program.[1] This value is a checksum of the program's non-pointer global variables at the end of the program's execution. Thus, if changing the compiler or compiler options causes the checksum emitted by a Csmith-generated program to change, a compiler bug has been found.

The C99 language [11] has 191 *undefined behaviors*—e.g., dereferencing a null pointer or overflowing a signed integer—that destroy the meaning of a program. It also has 52 *unspecified behaviors*—e.g., the order of evaluation of arguments to a function—where a compiler may choose from a set of options with no requirement that the choice be made consistently. Programs emitted by Csmith must avoid all of these behaviors or, in certain cases such as argument-evaluation order, be independent of the choices that will be made by the compiler. Many undefined and unspecified behaviors can be avoided structurally by generating programs in such a way that problems never arise. However, a number of important undefined and unspecified behaviors are not easy to avoid in a structural fashion. In these cases, Csmith solves the problem using static analysis and by adding run-time checks to the generated code. Section 2.4 describes the hazards that Csmith must avoid and its strategies for avoiding them.

Csmith's second design goal is to maximize expressiveness subject to constraints imposed by the first goal. An "expressive" generator supports many language features and combinations of features. Our hypothesis is that expressiveness is correlated with bug-finding power.

[1] Accesses to volatile objects are also side effects as described in the C standard. We do not discuss these "secondary" side effects of Csmith-generated programs further in this paper.

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

The most important language features not currently supported by Csmith are strings, dynamic memory allocation, floating-point types, unions, recursion, and function pointers. We plan to add some of these features to future versions of our tool.

2.3 Randomly Generating Programs

The shape of a program generated by Csmith is governed by a grammar for a subset of C. A program is a collection of type, variable, and function definitions; a function body is a block; a block contains a list of declarations and a list of statements; and a statement is an expression, control-flow construct (e.g., `if`, `return`, `goto`, or `for`), assignment, or block. Assignments are modeled as statements—not expressions—which reflects the most common idiom for assignments in C code. We leverage our grammar to produce other idiomatic code as well: in particular, we include a statement kind that represents a loop iterating over an array. The grammar is implemented by a collection of hand-coded C++ classes.

Csmith maintains a *global environment* that holds top-level definitions: i.e., types, global variables, and functions. The global environment is extended as new entities are defined during program generation. To hold information relevant to the current program-generation point, Csmith also maintains a *local environment* with three primary kinds of information. First, the local environment describes the current call chain, supporting context-sensitive pointer analysis. Second, it contains effect information describing objects that may have been read or written since (1) the start of the current function, (2) the start of the current statement, and (3) the previous sequence point.[2] Third, the local environment carries points-to facts about all in-scope pointers. These elements and their roles in program generation are further described in Section 2.4.

Csmith begins by randomly creating a collection of struct type declarations. For each, it randomly decides on a number of members and the type of each member. The type of a member may be a (possibly qualified) integral type, a bit-field, or a previously generated struct type.

After the preliminary step of producing type definitions, Csmith begins to generate C program code. Csmith generates a program top-down, starting from a single function called by `main`. Each step of the program generator involves the following sub-steps:

1. Csmith randomly selects an allowable production from its grammar for the current program point. To make the choice, it consults a probability table and a filter function specific to the current point: there is a table/filter pair for statements, another for expressions, and so on. The table assigns a probability to each of the alternatives, where the sum of the probabilities is one. After choosing a production from the table, Csmith executes the filter, which decides if the choice is acceptable in the current context. Filters enforce basic semantic restrictions (e.g., `continue` can only appear within a loop), user-controllable limits (e.g., maximum statement depth and number of functions), and other user-controllable options. If the filter rejects the selected production, Csmith simply loops back, making selections from the table until the filter succeeds.
2. If the selected production requires a target—e.g., a variable or function—then the generator randomly selects an appropriate target or defines a new one. In essence, Csmith dynamically constructs a probability table for the potential targets and includes an option to create a new target. Function and variable definitions are thus created "on demand" at the time that Csmith decides to refer to them.
3. If the selected production allows the generator to select a type, Csmith randomly chooses one. Depending on the current context, the choice may be restricted (e.g., while generating the operands of an integral-typed expression) or unrestricted (e.g., while generating the types of parameters to a new function). Random choices are guided by the grammar, probability tables, and filters as already described.
4. If the selected production is nonterminal, the generator recurses. It calls a function to generate the program fragment that corresponds to the nonterminal production. More generally, Csmith recurses for each nonterminal element of the current production: e.g., for each subcomponent of a compound statement, or for each parameter in a function call.
5. Csmith executes a collection of dataflow transfer functions. It passes the points-to facts from the local environment to the transfer functions, which produce a new set of points-to facts. Csmith updates the local environment with these facts.
6. Csmith executes a collection of safety checks. If the checks succeed, the new code fragment is committed to the generated program. Otherwise, the fragment is dropped and any changes to the local environment are rolled back.

When Csmith creates a call to a new function—one whose body does not yet exist—generation of the current function is suspended until the new function is finished. Thus, when the top-level function has been completely generated, Csmith is finished. At that point it pretty-prints all of the randomly generated definitions in an appropriate order: types, globals, prototypes, and functions. Finally, Csmith outputs a `main` function. The `main` function calls the top-level randomly generated function, computes a checksum of the non-pointer global variables, prints the checksum, and exits.

2.4 Safety Mechanisms

Table 1 lists the mechanisms that Csmith uses to avoid generating C programs that execute undefined behaviors or depend on unspecified behaviors. This section provides additional detail about the hazards that Csmith must avoid and its strategies for avoiding them.

Integer safety More and more, compilers are aggressively exploiting the undefined nature of integer behaviors such as signed overflow and shift-past-bitwidth. For example, recent versions of Intel CC, GCC, and LLVM evaluate `(x+1)>x` to 1 while also evaluating `(INT_MAX+1)` to `INT_MIN`. In another example, discovered by the authors of Google's Native Client software [3], routine refactoring of C code caused the expression `1<<32` to be evaluated on a

[2] As explained in Section 3.8 of the C FAQ [25], "A sequence point is a point in time at which the dust has settled and all side effects which have been seen so far are guaranteed to be complete. The sequence points listed in the C standard are at the end of the evaluation of a full expression (a full expression is an expression statement, or any other expression which is not a subexpression within any larger expression); at the `||`, `&&`, `?:`, and comma operators; and at a function call (after the evaluation of all the arguments, and just before the actual call)."

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

Table 1. Summary of Csmith's strategies for avoiding undefined and unspecified behaviors. When both a code-generation-time and code-execution-time solution are listed, Csmith uses both.

platform with 32-bit integers. The compiler exploited this undefined behavior to turn a sandboxing safety check into a nop.

To keep Csmith-generated programs from executing integer undefined behaviors, we implemented a family of wrapper functions for arithmetic operators whose (promoted) operands might overflow. This was not difficult, but had a few tricky aspects. For example, the C99 standard does not explicitly identify the evaluation of `INT_MIN%-1` as being an undefined behavior, but most compilers treat it as such. The C99 standard also has very restrictive semantics for signed left-shift: it is illegal (for implementations using 2's complement integers) to shift a 1-bit into or past the sign bit. Thus, evaluating `1<<31` destroys the meaning of a C99 program on a platform with 32-bit ints.

Several safe math libraries for C that we examined themselves execute operations with undefined behavior while performing checks. Apparently, avoiding such behavior is indeed a tricky business.

Type safety The aspect of C's type system that required the most care was *qualifier safety*: ensuring that `const` and `volatile` qualifiers attached to pointers at various levels of indirection are not removed by implicit casts. Accessing a const- or volatile-qualified object through a non-qualified pointer results in undefined behavior.

Pointer safety Null-pointer dereferences are easy to avoid using dynamic checks. There is, on the other hand, no portable run-time method for detecting references to a function-scoped variable whose lifetime has ended. (Hacks involving the stack pointer are not robust under inlining.) Although there are obvious ways to structurally avoid this problem, such as using a type system to ensure that a pointer to a function-scoped variable never outlives the function, we judged this kind of strategy to be too restrictive. Instead, Csmith freely permits pointers to local variables to escape (e.g., into global variables) but uses a whole-program pointer analysis to ensure that such pointers are not dereferenced or used in comparisons once they become invalid.

Csmith's pointer analysis is flow sensitive, field sensitive, context sensitive, path insensitive, and array-element insensitive. A points-to fact is an explicit set of locations that may be referenced, and may include two special elements: the null pointer and the invalid (out-of-scope) pointer. Points-to sets containing a single element serve as must-alias facts unless the pointed-to object is an array element. Because Csmith does not generate programs that use the heap, assigning names to storage locations is trivial.

Effect safety The C99 standard states that "[t]he order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified." Also, undefined behavior occurs if "[b]etween two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored."

To avoid these problems, Csmith uses its pointer analysis to perform a conservative interprocedural analysis and determine the *effect* of every expression, statement, and function that it generates. An effect consists of two sets: locations that may be read and locations that may be written. Csmith ensures that no location is both read and written, or written more than once, between any pair of sequence points. As a special case, in an assignment, a location can be read on the RHS and also written on the LHS.

Effects are computed, and effect safety guaranteed, incrementally. At each sequence point, Csmith resets the current effect (i.e., may-read and may-write sets). As fragments of code are generated, Csmith tests if the new code has a read/write or write/write conflict with the current effect. If a conflict is detected, the new code is thrown away and the process restarts. For example, if Csmith is generating an expression `p + func()` and it happens that `func` may modify `p`, the call to `func` is discarded and a new subexpression is generated. If there is no conflict, the read and write sets are updated and the process continues. Probabilistic progress is guaranteed: by design, Csmith always has a non-zero chance of generating code that introduces no new conflicts, such as a constant expression.

Array safety Csmith uses several methods to ensure that array indices are in bounds. First, it generates index variables that are modified only in the "increment" parts of `for` loops and whose values never exceed the bounds of the arrays being indexed. Second, variables with arbitrary value are forced to be in bounds using the modulo operator. Finally, as needed, Csmith emits explicit checks against array lengths.

Initializer safety A C program must not use an uninitialized function-scoped variable. For the most part, initializer safety is easy to ensure structurally by initializing variables close to where they are declared. Gotos introduce the possibility that initializers may be jumped over; Csmith solves this by forbidding gotos from spanning initialization code.

2.5 Efficient Global Safety

Csmith never commits to a code fragment unless it has been shown to be safe. However, loops and function calls threaten to invalidate previously validated code. For example, consider the following code, in which Csmith has just added the loop back-edge at line 7.

```
1   int i;
2   int *p = &i;
3   while (...) {
4     *p = 3;
5     ...
6     p = 0;
7   }
```

The assignment through `p` at line 4 was safe when it was generated. However, the newly added line 7 makes line 4 unsafe, due to the back-edge carrying a null-valued `p`.

One solution to this problem is to be conservative: run the whole-program dataflow analysis before committing any new statement to the program. This is not efficient. We therefore restrict the analysis to local scope except when function calls and loops are involved. For a function call, the callee is re-analyzed at each call site immediately.

Csmith uses a different strategy for loops. This is because so many statements are inside loops, and the extra calls to the dataflow analysis add substantial overhead to the code generator. Csmith's strategy is to optimistically generate code that is *locally safe*. Local safety includes running a single step of the dataflow engine (which reaches a sound result when generating code not inside any loop).

The global fixpoint analysis is run when a loop is closed by adding its back-edge. If Csmith finds that the program contains unsafe statements, it deletes code starting from the tail of the loop until the program becomes globally safe. This strategy is about three times faster than pessimistically running the global dataflow analysis before adding every piece of code.

2.6 Design Trade-offs

Allow implementation-defined behavior An ideally portable test program would be "strictly conforming" to the C language standard. This means that the program's output would be independent of all unspecified and unspecified behaviors and, in addition, be independent of any *implementation-defined behavior*. C99 has 114 kinds of implementation-defined behavior, and they have pervasive impact on the behavior of real C programs. For example, the result of performing a bitwise operation on a signed integer is implementation-defined, and operands to arithmetic operations are implicitly cast to `int` (which has implementation-defined width) before performing the operation. We believe it is impossible to generate realistically expressive C code that retains a single interpretation across all possible choices of implementation-defined behaviors.

Programs generated by Csmith do not generate the same output across compilers that differ in (1) the width and representation of integers, (2) behavior when casting to a signed integer type when the value cannot be represented in an object of the target type, and (3) the results of bitwise operations on signed integers. In practice there is not much diversity in how C implementations define these behaviors. For mainstream desktop and embedded targets, there are roughly three equivalence classes of compiler targets: those where `int` is 32 bits and `long` is 64 bits (e.g., x86-64), those where `int` and `long` are 32 bits (e.g., x86, ARM, and PowerPC), and those where `int` is 16 bits and `long` is 32 bits (e.g., MSP430 and AVR). Using Csmith, we can perform differential testing within an equivalence class but not across classes.

No ground truth Csmith's programs are not self-checking: we are unable to predict their outputs without running them. This is not a problem when we use Csmith for randomized differential testing.

We have never seen an "interesting" split vote where randomized differential testing of a collection of C compilers fails to produce a clear consensus answer, nor have we seen any cases in which a majority of tested compilers produces the same incorrect result. (We would catch the problem by hand as part of verifying the failure-inducing program.) In fact, we have not seen even two unrelated compilers produce the same incorrect output for a Csmith-generated test case. It therefore seems unlikely that all compilers under test would produce the same incorrect output for a test case. Of course, if that did happen we would not detect that problem; this is an inherent limitation of differential testing without an oracle. In summary, despite the fact that Knight and Leveson [13] found a substantial number of correlated errors in an experiment on N-version programming, Csmith has yielded no evidence of correlated failures among unrelated C compilers. Our hypothesis is that the observed lack of correlation stems from the fact that most compiler bugs are in passes that operate on an intermediate representation and there is substantial diversity among IRs.

No guarantee of termination It is not difficult to generate random programs that always terminate. However, we judged that this would limit Csmith's expressiveness too much: for example, it would force loops to be highly structured. Additionally, always-terminating tests cannot find compiler bugs that wrongfully terminate a non-terminating program. (We have found bugs of this kind.) About 10% of the programs generated by Csmith are (apparently) non-terminating. In practice, during testing, they are easy to deal with using timeouts.

Target middle-end bugs Commercial test suites for C compilers [1, 19, 20] are primarily aimed at checking standards conformance. Csmith, on the other hand, is mainly intended to find bugs in the parts of a compiler that perform transformations on an intermediate representation—the so-called "middle end" of a compiler. As a result, we have found large numbers of middle-end bugs missed by existing testing techniques (Section 3.6). At the same time, Csmith is rather poor at finding gaps in standards conformance. For example, it makes no attempt to test a compiler's handling of trigraphs, long identifier names, or variadic functions.

Targeting the middle end has several aspects. First, all generated programs pass the lexer, parser, and typechecker. Second, we performed substantial manual tuning of the 80 probabilities that govern Csmith's random choices. Our goal was to make the generated programs "look right"—to contain a balanced mix of arithmetic and bitwise operations, of references to scalars and aggregates, of loops and straight-line code, of single-level and multi-level indirections, and so on. Third, Csmith specifically generates idiomatic code (e.g., loops that access all elements of an array) to stress-test parts of the compiler we believe to be error-prone. Fourth, we designed Csmith with an eye toward generating programs that exercise the constructs of a compiler's intermediate representation, and we decided to avoid generating source-level diversity that is unlikely to improve the "coverage" of a compiler's intermediate representations. For example, since additional levels of parentheses around expressions are stripped away early in the compilation process, we do not generate them, nor do we generate all of C's syntactic loop forms since they are typically all lowered to the same IR constructs. Finally, Csmith was designed to be fast enough that it can generate programs that are a few tens of thousands of lines long in a few seconds. Large programs are preferred because (empirically—see Section 3.3) they find more bugs. In summary, many aspects of Csmith's design and implementation were informed by our understanding of how modern compilers work and how they break.

3. Results

We conducted five experiments using Csmith, our random program generator. This section summarizes our findings.

Our first experiment was uncontrolled and unstructured: over a three-year period, we opportunistically found and reported bugs in a variety of C compilers. We found bugs in all the compilers we tested—hundreds of defects, many classified as high-priority bugs. (§3.1)

In the second experiment, we compiled and ran one million random programs using several years' worth of versions of GCC and LLVM, to understand how their robustness is evolving over time. As measured by our tests over the programs that Csmith produces, the quality of both compilers is generally improving. (§3.2)

Third, we evaluated Csmith's bug-finding power as a function of the size of the generated C programs. The largest number of bugs is found at a surprisingly large program size: about 81 KB. (§3.3)

Fourth, we compared Csmith's bug-finding power to that of four previous random C program generators. Over a week, Csmith was able to find significantly more distinct compiler crash errors than previous program generators could. (§3.4)

Finally, we investigated the effect of testing random programs on branch, function, and line coverage of the GCC and LLVM source code. We found that these metrics did not significantly improve when we added randomly generated programs to the compilers' existing test suites. Nevertheless, as shown by our other results, Csmith-generated programs allowed us to discover bugs that are missed by the compilers' standard test suites. (§3.5)

We conclude the presentation of results by analyzing some of the bugs we found in GCC and LLVM. (§3.6, §3.7)

	GCC	LLVM
Crash	2	10
Wrong code	2	9
Total	4	19

Table 2. Crash and wrong-code bugs found by Csmith that manifest when compiler optimizations are disabled (i.e., when the –O0 command-line option is used)

3.1 Opportunistic Bug Finding

We reported bugs to 11 different C compiler development teams. Five of these compilers (GCC, LLVM, CIL, TCC, and Open64) were open source and five were commercial products. The eleventh, CompCert, is publicly available but not open source.

What kinds of bugs are there? It is useful to distinguish between errors whose symptoms manifest at compile time and those that only manifest when the compiler's output is executed. Compile-time bugs that we see include assertion violations or other internal compiler errors; involuntary compiler termination due to memory-safety problems; and cases in which the compiler exhausts the RAM or CPU time allocated to it. We say that a compile-time *crash error* has occurred whenever the compiler process exits with a status other than zero or fails to produce executable output. Errors that manifest at run time include the computation of a wrong result; a crash or other abnormal termination of the generated code; termination of a program that should have executed forever; and non-termination of a program that should have terminated. We refer to these run-time problems as *wrong-code errors*. A *silent wrong-code error* is one that occurs in a program that was produced without any sort of warning from the compiler; i.e., the compiler silently miscompiled the test program.

Experience with commercial compilers There exist many more commercial C compilers than we could easily test. The ones we chose to study are fairly popular and were produced by what we believe are some of the strongest C compiler development teams. Csmith found wrong-code errors and crash errors in each of these tools within a few hours of testing.

Because we are not paying customers, and because our findings represent potential bad publicity, we did not receive a warm response from any commercial compiler vendor. Thus, for the most part, we simply tested these compilers until we found a few crash errors and a few wrong-code errors, reported them, and moved on.

Experience with open-source compilers For several reasons, the bulk of our testing effort went towards GCC and LLVM. First and most important, compiler testing is inherently interactive: we require feedback from the development team in the form of bug fixes. Bugs that occur with high probability can mask tricky, one-in-a-million bugs; thus, testing proceeds most smoothly when we can help developers rapidly destroy the easy bugs. Both the GCC and LLVM teams were responsive to our bug reports. The LLVM team in particular fixed bugs quickly, often within a few hours and usually within a week. The second reason we prefer dealing with open-source compilers is that their development process is transparent: we can watch the mailing lists, participate in discussions, and see fixes as they are committed. Third, we want to help harden the open-source development tools that we and many others use daily.

So far we have reported 79 GCC bugs and 202 LLVM bugs—the latter figure represents about 2% of all LLVM bug reports. Most of our reported bugs have been fixed, and twenty-five of the GCC bugs were marked by developers as P1: the maximum, release-blocking priority for a bug. To date, we have reported 325 in total across all tested compilers (GCC, LLVM, and others).

An error that occurs at the lowest level of optimization is pernicious because it defeats the conventional wisdom that compiler bugs can be avoided by turning off the optimizer. Table 2 counts these kinds of bugs, causing both crash and wrong-code errors, that we found using Csmith.

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.

The second CompCert problem we found was illustrated by two bugs that resulted in generation of code like this:

```
stwu r1, -44432(r1)
```

Here, a large PowerPC stack frame is being allocated. The problem is that the 16-bit displacement field is overflowed. CompCert's PPC semantics failed to specify a constraint on the width of this immediate value, on the assumption that the assembler would catch out-of-range values. In fact, this is what happened. We also found a handful of crash errors in CompCert.

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.

3.2 Quantitative Comparison of GCC and LLVM Versions

Figure 3 shows the results of an experiment in which we compiled and ran 1,000,000 randomly generated programs using LLVM 1.9–2.8, GCC 3.[0–4].0, and GCC 4.[0–5].0. Every program was compiled at –O0, –O1, –O2, –Os, and –O3. A test case was considered valid if every compiler terminated (successfully or otherwise) within five minutes and if every compiled random program terminated (correctly or otherwise) within five seconds. All compilers targeted x86. Running these tests took about 1.5 weeks on 20 machines in the Utah Emulab testbed [28]. Each machine had one quad-core Intel Xeon E5530 processor running at 2.4 GHz.

Compile-time failures The top row of graphs in Figure 3 shows the observed rate of crash errors. (Note that the y-axes of these graphs are logarithmic.) These graphs also indicate the number of crash bugs that were fixed in response to our bug reports. Both compilers became at least three orders of magnitude less "crashy" over the range of versions covered in this experiment. The GCC results appear to tell a nice story: the 3.x release series increases in quality, the 4.0.0 release regresses because it represents a major change to GCC's internals, and then quality again starts to improve.

The middle row of graphs in Figure 3 shows the number of *distinct assertion failures* in LLVM and the number of *distinct internal compiler errors* in GCC induced by our tests. These are the numbers of code locations in LLVM and GCC at which an internal

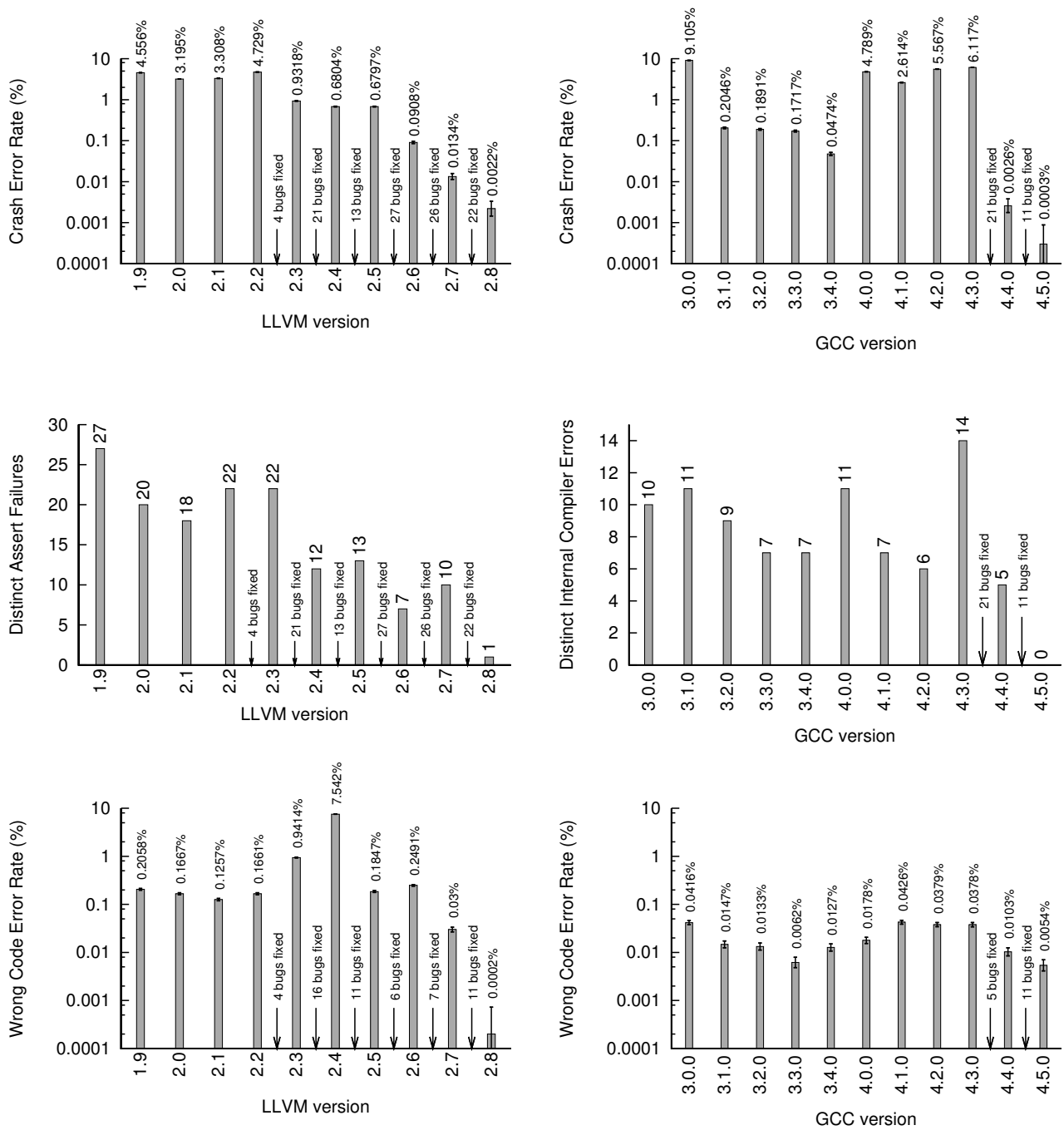

Figure 3. Distinct crash errors found, and rates of crash and wrong-code errors, from recent LLVM and GCC versions

consistency check failed. These graphs conservatively estimate the number of distinct failures in these compilers, since we encountered many segmentation faults caused by use of free memory, null-pointer dereferences, and similar problems. We did not include these faults in our graphed results due to the difficulty of mapping crashes back to distinct causes.

It is not clear which of these two metrics of crashiness is preferable. The rate of crashes is easy to game: we can make it arbitrarily high by biasing Csmith to generate code triggering known bugs, and compiler writers can reduce it to zero by eliminating error messages and always returning a "success" status code to the operating system. The number of distinct crashes, on the other hand, suffers from the drawback that it depends on the quantity and style of assertions in the compiler under test. Although GCC has more total assertions than LLVM, LLVM has a higher density: about one assertion per 100 lines of code, compared to one in 250 for GCC.

Run-time failures The bottom pair of graphs in Figure 3 shows the rate of wrong-code errors in our experiment. Unfortunately, we

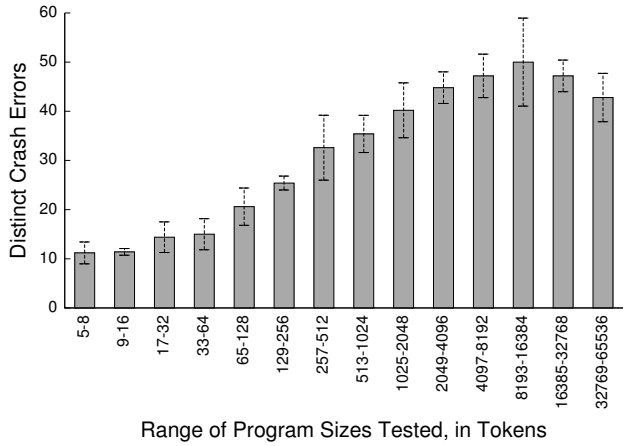

Figure 4. Number of distinct crash errors found in 24 hours of testing with Csmith-generated programs in a given size range

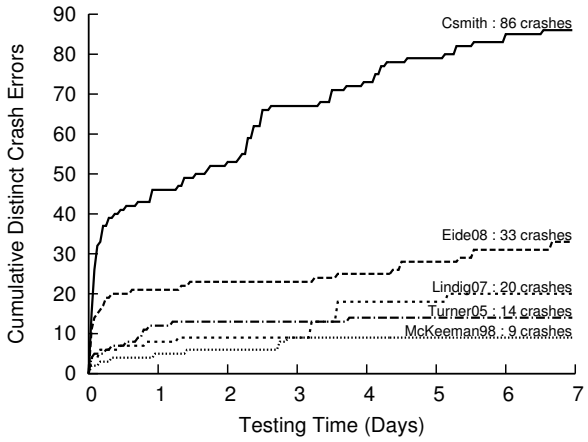

Figure 5. Comparison of the ability of five random program generators to find distinct crash errors

can only report the rate of errors, and not the number of bugs causing them, because we do not know how to automatically map failing tests back to the bugs that cause them. These graphs also indicate the number of wrong-code bugs that were fixed in response to our bug reports.

3.3 Bug-Finding Performance as a Function of Test-Case Size

There are many ways in which a random test-case generator might be "tuned" for particular goals, e.g., to focus on certain kinds of compiler defects. We performed an experiment to answer this question: given the goal of finding many defects quickly, should one configure Csmith to generate small programs or large ones? Other factors being equal, small test cases are preferable because they are closer to being reportable to compiler developers.

Using the same compilers and optimization options that we used for the experiments in Section 3.2, we ran our testing process multiple times. For each run we selected a size range for test inputs, configured Csmith to generate programs in that range,[3] executed the test process for 24 hours, and counted the distinct crash errors found. We repeated this for various ranges of test-input sizes.

Figure 4 shows that the rate of crash-error detection varies significantly as a function of the sizes of the test programs produced by Csmith. The greatest number of distinct crash errors is found by programs containing 8 K–16 K tokens: these programs averaged 81 KB before preprocessing. The confidence intervals are at 95% and were computed based on five repetitions.

We hypothesize that larger test cases expose more compiler errors for two reasons. First, throughput is increased because compiler start-up costs are better amortized. Second, the combinatorial explosion of feature interactions within a single large test case works in Csmith's favor. The decrease in bug-finding power at the largest sizes appears to come from algorithms—in Csmith and in the compilers—that have superlinear running time.

3.4 Bug-Finding Performance Compared to Other Tools

To evaluate Csmith's ability to find bugs, we compared it to four other random program generators: the two versions of Randprog described in Section 2 and two others described in Section 5. We ran each generator in its default configuration on one of five identical and otherwise-idle machines, using one CPU on each host. Each generator repeatedly produced programs that we compiled and tested using the same compilers and optimization options that were used for the experiments in Section 3.2. Figure 5 plots the cumulative number of distinct crash errors found by these program generators during the one-week test. Csmith significantly outperforms the other tools.

[3] Although we can tune Csmith to prefer generating larger or smaller output, it lacks the ability to construct a test case of a specific size on demand. We ran this experiment by precomputing seeds to Csmith's random-number generator that cause it to generate programs of the sizes we desired.

		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

Table 3. Augmenting the GCC and LLVM test suites with 10,000 randomly generated programs did not improve code coverage much

3.5 Code Coverage

Because we find many bugs, we hypothesized that randomly generated programs exercise large parts of the compilers that were not covered by existing test suites. To test this, we enabled code-coverage monitoring in GCC and LLVM. We then used each compiler to build its own test suite, and also to build its test suite plus 10,000 Csmith-generated programs. Table 3 shows that the incremental coverage due to Csmith is so small as to be a negative result. Our best guess is that these metrics are too shallow to capture Csmith's effects, and that we would generate useful additional coverage in terms of deeper metrics such as path or value coverage.

3.6 Where Are the Bugs?

Table 4 characterizes the GCC and LLVM bugs we found by compiler part. Tables 5 and 6 show the ten buggiest files in LLVM and GCC as measured by our experiment in Section 3.1. Most of the bugs we found in GCC were in the middle end: the machine-independent optimizers. LLVM is a younger compiler and our testing shook out some front-end and back-end bugs that would probably not be present in a more mature software base.

	GCC	LLVM
Front end	0	10
Middle end	49	75
Back end	17	74
Unclassified	13	43
Total	79	202

Table 4. Distribution of bugs across compiler stages. A bug is unclassified either because it has not yet been fixed or the developer who fixed the bug did not indicate what files were changed.

C File Name	Purpose	Wrong-Code Bugs	Crash Bugs
fold-const	constant folding	3	6
combine	instruction combining	1	5
tree-ssa-pre	partial redundancy elim.	0	4
tree-vrp	variable range propagation	0	4
tree-ssa-dce	dead code elimination	0	3
tree-ssa-reassoc	arithmetic expr. reassociation	0	2
reload1	register reloading	1	1
tree-ssa-loop-niter	loop iteration counting	1	1
dse	dead store elimination	2	0
tree-scalar-evolution	scalar evolution	2	0
Other (15 files)	n/a	19	24
Total (25 files)	n/a	29	50

Table 5. Top ten buggy files in GCC

C++ File Name	Purpose	Wrong-Code Bugs	Crash Bugs
Instruction-Combining	mid-level instruction combining	9	6
SimpleRegister-Coalescing	register coalescing	1	10
DAGCombiner	instruction combining	5	3
LoopUnswitch	loop unswitching	1	4
LICM	loop invariant code motion	0	5
LoopStrength-Reduce	loop strength reduction	1	3
FastISel	fast instruction selection	1	3
llvm-convert	GCC-LLVM IR conversion	0	4
ExprConstant	constant folding	2	2
JumpThreading	jump threading	0	4
Other (72 files)	n/a	46	92
Total (82 files)	n/a	66	136

Table 6. Top ten buggy files in LLVM

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.

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.

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.

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.

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.

[4] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42721`

[5] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43438`

[6] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42952`

[7] `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43360`

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.

LLVM Bug #3: wrong safety check[10] "Narrowing" is a strength-reduction optimization that can be applied to loads when only part of an object is needed, or to stores where only part of an object is modified. For example, at the level of the abstract machine this code loads and stores an `unsigned int`:

```
1 unsigned y;
2
3 void bar (void) {
4   y |= 255;
5 }
```

Optimizing compilers for x86 may translate `bar` into the following code, which loads nothing and stores a single byte:

```
bar:
  movb $-1, y
  ret
```

We found a case in which LLVM attempted to perform an analogous narrowing operation, but a logic error caused the safety check to succeed even when a different store modified the object prior to the store that was the target of the narrowing transformation.

LLVM Bug #4: wrong analysis[11] This code should print "5":

```
1 void foo (void) {
2   int x;
3   for (x = 0; x < 5; x++) {
4       if (x) continue;
5       if (x) break;
6   }
7   printf("%d", x);
8 }
```

LLVM's scalar evolution analysis computes properties of loop induction variables, including the maximum number of iterations. Line 5 of the program above caused this analysis to mistakenly conclude that `x` was 1 after the loop executed.

4. Discussion

Are we finding bugs that matter? One might suspect that random testing finds bugs that do not matter in practice. Undoubtedly this happens sometimes, but in a number of instances we have direct confirmation that Csmith is finding bugs that matter, because bugs that we have found and reported have been independently rediscovered and re-reported by application developers. By a very conservative estimate—counting only the times that a compiler developer explicitly labeled a wrong-code bug report as a duplicate of one of ours—this has happened eight times: four times for GCC and four for LLVM. We also have indirect confirmation that our bugs matter. The developers of open-source compilers fixed almost all of the bugs that we reported, and the GCC development team marked 25 of our bugs as P1: the maximum, release-blocking priority.

Creating reportable bugs Reporting compiler crash bugs is easy, but reporting wrong-code bugs is harder. Compiler developers will (rightfully) ignore a wrong-code bug report that is based on a large random program. Rather, a bug report must be accompanied by compelling evidence that a bug exists; in most cases the best evidence is a small test case that is obviously miscompiled. Delta debugging [31] automates test-case reduction, but all existing variants that are intended for reducing C programs—such as hierarchical delta debugging [18] and Wilkerson's implementation [29]—introduce undefined behavior. The resulting programs are small but useless. To avoid undefined behavior during reduction, we rely on compiler warnings, dynamic checkers, and manual test-case reduction. There is substantial room for improvement.

The relationship between testing and verification As our CompCert results make plain, verification does not obviate testing, but rather complements it. Testing can provide end-to-end evidence that numerous paths through a system work properly. Verification, on the other hand, typically focuses on a narrow slice of a stack of tools, and the parts outside the slice remain in the trusted computing base. There does not yet appear to be a nuanced understanding of the kinds of testing, and the amount of testing effort, that are rendered unnecessary by artifacts like CompCert [14] and seL4 [12].

Toward realistic, correct compilers Compilers must support rapid development to cope with new optimizations, new source languages, and new target architectures. Generated code often needs to be resource-efficient to support application developers' goals. Finally, compilers should generate correct code. Meeting even two of these goals is challenging, and it is not clear how to meet all three in a single tool. There seem to be four paths forward.

Compiler verification. Although it is difficult to imagine a verified compiler for C++0x, due to the immense complexity of the draft standard, CompCert is an existence proof that a verified, optimizing C compiler is within reach. However, the burden of verification is significant. CompCert still lacks a number of useful C features and few mainstream compiler developers have the formal verification skills that are needed to add new language features and optimization passes. On the other hand, projects such as XCERT [26] may dramatically lower the bar for working on verified compilation.

Compiler simplicity. For non-bottleneck applications, compiler optimization adds little end-user value. It would seem possible to take a simple compiler such as TCC [2], which does not optimize across statement boundaries, and validate it through code inspections, heavy use, and other techniques. At present, however, TCC is much buggier than more heavily-used compilers such as GCC and LLVM.

Compiler testing. We hypothesize that it is possible to gain high confidence in a complex compiler like GCC by choosing a fixed configuration, disabling optimization passes whose effects are significantly non-local, and performing "just enough testing." A test plan would be sufficient if all code paths through the compiler that are used to compile an application of interest had been tested. Clearly, a sophisticated way to abstract over paths is needed.

Equivalence checking. If equivalence checkers for machine code [7] could scale to large programs, verified compilers would be largely unnecessary because one compiler's output could be proved equivalent to another's. Although these tools are not likely to scale up to multi-megabyte applications anytime soon, it should

[8] `http://llvm.org/bugs/show_bug.cgi?id=2844`

[9] `http://llvm.org/bugs/show_bug.cgi?id=7750`

[10] `http://llvm.org/bugs/show_bug.cgi?id=7833`

[11] `http://llvm.org/bugs/show_bug.cgi?id=7845`

be possible to automatically partition applications into smaller parts so that equivalence checking can be done piecewise.

Future work Augmenting Csmith with white-box testing techniques, where the structure of the tested system is taken into account in a first-class way, would be productive. This will be difficult for several reasons. First, we anticipate substantial challenges in integrating the necessary constraint-solving machinery with Csmith's existing logic for generating valid C programs. It is possible that we will need to start over, next time engineering a version of Csmith in which all constraints are explicit and declarative, rather than being buried in a small mountain of C++. Second, the inverse problems that must be solved to generate an input become prohibitively difficult when inputs pass through a parser, particularly if the parser contains hash tables. Godefroid et al. [8] showed a way to solve this problem by integrating a constraint solver with a grammar for the language being generated. However, due to its non-local pointer and effect analyses, the validity decision problem for programs in the subset of C that Csmith generates is far harder than the question of whether a program can be generated by the JavaScript grammar used by Godefroid et al.

5. Related Work

Compilers have been tested using randomized methods for nearly 50 years. Boujarwah and Saleh [4] gave a good survey in 1997. In 1962, Sauder [22] tested the correctness of COBOL compilers by placing random variables in programs' data sections. In 1970, Hanford [10] used a PL/1 grammar to drive the generation of random programs. The grammar was extensible and was augmented by "syntax generators" that could be used, for example, to ensure that variables were declared before being used. In 1972, Purdom [21] used a syntax-directed method to generate test sentences for a parser. He gave an efficient algorithm for generating short sentences from a context-free grammar such that each production of the grammar was used at least once, and he tested LR(1) parsers using this technique.

Burgess and Saidi [5] designed an automatic generator of test cases for FORTRAN compilers. The tests were designed to be self-checking and to contain features that optimizing compilers were known to exploit. In order to predict test cases' results, the code generator restricted assignment statements to be executed only once during the execution of the sub-program or main program. These tests found four bugs in two FORTRAN 77 compilers.

In 1998, McKeeman [16] coined the term "differential testing." His work resulted in DDT, a family of program generators that conform to the C standard at various levels, from level 1 (random characters) to level 7 (generated code is "model conforming," incorporating some high-level structure). DDT is more expressive than Csmith (DDT is capable of generating all legal C programs) and it was used to find numerous bugs in C compilers. To our knowledge, McKeeman's paper contains the first acknowledgment that it is important to avoid undefined behavior in generated C programs used for compiler testing. However, DDT avoided only a small subset of all undefined behaviors, and only then during test-case reduction, not during normal testing. Thus, it is not a suitable basis for automatic bug-finding.

Lindig [15] used randomly generated C programs to find several compiler bugs related to calling conventions. His tool, called Quest, was specially targeted: rather than generating code with control flow and arithmetic, Quest generates code that creates complex data structures, loads them with constant values, and passes them to a function where assertions check the received values. Because its tests are self-checking, Quest is not based on differential testing. Self-checking tests are convenient, but the drawback is that Quest is far less expressive than Csmith. Lindig used Quest to test GCC, LCC, ICC, and a few other compilers and found 13 bugs.

Sheridan [23] also used a random generator to find bugs in C compilers. A script rotated through a list of constants of the principal arithmetic types, producing a source file that applied various operators to pairs of constants. This tool found two bugs in GCC, one bug in SUSE Linux's version of GCC, and five bugs in CodeSourcery's version of GCC for ARM. Sheridan's tool produces self-checking tests. However, it is less expressive than Csmith and it fails to avoid undefined behavior such as signed overflow.

Zhao et al. [32] created an automated program generator for testing an embedded C++ compiler. Their tool allows a general test requirement, such as which optimization to test, to be specified in a script. The generator constructs a program template based on the test requirement and uses it to drive further code generation. Zhao et al. used GCC as the reference to check the compiler under test. They reported greatly improved statement coverage in the tested modules and found several new compiler bugs.

6. Conclusion

Using randomized differential testing, we found and reported hundreds of previously unknown bugs in widely used C compilers, both commercial and open source. Many of the bugs we found cause a compiler to emit incorrect code without any warning. Most of our reported defects have been fixed, meaning that compiler implementers found them important enough to track down, and 25 of the bugs we reported against GCC were classified as release-blocking. All of this evidence suggests that there is substantial room for improvement in the state of the art for compiler quality assurance.

To create a random program generator with high bug-finding power, the key problem we solved was the expressive generation of C programs that are free of undefined behavior and independent of unspecified behavior. Csmith, our program generator, uses both static analysis and dynamic checks to avoid these hazards.

The return on investment from random testing is good. Our rough estimate—including faculty, staff, and student salaries, machines purchased, and university overhead—is that each of the more than 325 bugs we reported cost less than $1,000 to find. The incremental cost of a new bug that we find today is much lower.

Software Csmith is open source and available for download at `http://embed.cs.utah.edu/csmith/`.

Acknowledgments

The authors would like to thank Bruce Childers, David Coppit, Chucky Ellison, Robby Findler, David Gay, Casey Klein, Gerwin Klein, Chris Lattner, Sorin Lerner, Xavier Leroy, Bill McKeeman, Diego Novillo, Alastair Reid, Julian Seward, Zach Tatlock, our shepherd Atanas Rountev, and the anonymous reviewers for their invaluable feedback on drafts of this paper. We also thank Hans Boehm, Xavier Leroy, Michael Norrish, Bryan Turner, and the GCC and LLVM development teams for their technical assistance in various aspects of our work.

This research was primarily supported by an award from DARPA's Computer Science Study Group.

References

[1] ACE Associated Computer Experts. SuperTest C/C++ compiler test and validation suite. `http://www.ace.nl/compiler/supertest.html`.

[2] F. Bellard. TCC: Tiny C compiler, ver. 0.9.25, May 2009. `http://bellard.org/tcc/`.

[3] C. L. Biffle. Undefined behavior in Google NaCl, Jan. 2010. `http://code.google.com/p/nativeclient/issues/detail?id=245`.

[4] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.

[5] C. J. Burgess and M. Saidi. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology*, 38(2):111–119, 1996.

[6] E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *Proc. EMSOFT*, pages 255–264, Oct. 2008.

[7] X. Feng and A. J. Hu. Cutpoints for formal equivalence verification of embedded software. In *Proc. EMSOFT*, pages 307–316, Sept. 2005.

[8] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proc. PLDI*, pages 206–215, June 2008.

[9] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley, second edition, 2001.

[10] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, Dec. 1970.

[11] International Organization for Standardization. *ISO/IEC 9899:TC2: Programming Languages—C*, May 2005. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[12] G. Klein et al. seL4: Formal verification of an OS kernel. In *Proc. SOSP*, pages 207–220, Oct. 2009.

[13] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Software Eng.*, 12(1):96–109, Jan. 1986.

[14] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[15] C. Lindig. Random testing of C calling conventions. In *Proc. AADEBUG*, pages 3–12, Sept. 2005.

[16] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, Dec. 1998.

[17] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[18] G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In *Proc. ICSE*, pages 142–151, May 2006.

[19] Perennial, Inc. ACVS ANSI/ISO/FIPS-160 C validation suite, ver. 4.5, Jan. 1998. `http://www.peren.com/pages/acvs_set.htm`.

[20] Plum Hall, Inc. The Plum Hall validation suite for C. `http://www.plumhall.com/stec.html`.

[21] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[22] R. L. Sauder. A general test data generator for COBOL. In *AFIPS Joint Computer Conferences*, pages 317–323, May 1962.

[23] F. Sheridan. Practical testing of a C99 compiler using output comparison. *Software—Practice and Experience*, 37(14):1475–1488, Nov. 2007.

[24] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *Proc. FM*, pages 532–546, Nov. 2009.

[25] S. Summit. comp.lang.c frequently asked questions. `http://c-faq.com/`.

[26] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *Proc. PLDI*, pages 111–121, June 2010.

[27] B. Turner. Random Program Generator, Jan. 2007. `http://sites.google.com/site/brturn2/randomcprogramgenerator`.

[28] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.

[29] D. S. Wilkerson. Delta ver. 2006.08.03, Aug. 2006. `http://delta.tigris.org/`.

[30] M. Wolfe. How compilers and tools differ for embedded systems. In *Proc. CASES*, Sept. 2005. Keynote address. `http://www.pgroup.com/lit/articles/pgi_article_cases.pdf`.

[31] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, Feb. 2002.

[32] C. Zhao et al. Automated test program generation for an industrial optimizing compiler. In *Proc. ICSE Workshop on Automation of Software Test*, pages 36–43, May 2009.