Find a Compiler Bug in 5 Minutes

Christian Lindig
Saarland University
Department of Computer Science
Saarbrücken, Germany
lindig@cs.uni-sb.de

ABSTRACT

In a C compiler, function calls are difficult to implement correctly because they must respect a platform-specific calling convention. But they are governed by a simple invariant: parameters passed to a function must be received unaltered. A violation of this invariant signals an inconsistency in a compiler. We automatically test the consistency of C compilers using randomly generated programs. An inconsistency manifests itself as an assertion failure when compiling and running the generated code. The generation of programs is type-directed and can be controlled by the user with composable random generators in about 100 lines of Lua. Lua is a scripting language built into our testing tool that drives program generation. Random testing is fully automatic, requires no specification, yet is comparable in effectiveness with specification-based testing from prior work. Using this method, we uncovered 13 new bugs in mature open-source and commercial C compilers.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability, Measurement, Experimentation, Languages

1. INTRODUCTION

C compilers have been around virtually forever and building them is so well understood that it is taught in compiler classes. The truth is, specifically function calls in C are difficult to implement correctly. The reason is that generated machine code must adhere to a strict regime of calling conventions. These are issued by hardware vendors to ensure interoperability between compilers. For each supported platform a compiler must implement such a calling convention, of which typical programs only exercise a small part. This makes the implementation of function calls a source for latent compiler bugs even in mature compilers.

For example, the code in Figure 1 on the following page uncovers a bug[1] in the GNU C compiler 3.3 that is part of Apple's MacOS X 10.3 operating system for the PowerPC. Function `main` passes the values of four global variables to variadic function `callee`, which checks that it indeed receives the right values. This fails for the fourth argument `i`:

```
$ gcc -O2 -o bug bug.c
$ ./bug
bug:23: failed assertion ‘y.f == i.f’
Abort trap
```

We found several compiler bugs of this kind using randomly generated programs in the style of Figure 1. These programs are designed to test a simple invariant: a value passed to or from a function must be received unaltered. We call this property *consistency* of functional calls, which must be guaranteed by a compiler.

Testing compilers for consistency is fully automatic since the generated programs encode the consistency tests themselves. Running such a program fails an embedded assertion in case of an inconsistency. Therefore, testing consistency *requires no specification or test oracle* and dramatically simplifies testing of C compilers compared to prior work (Bailey and Davidson, 1996).

Our generation of programs is type driven: a test case is constructed around the declaration of a function (Section 2). But to be effective, the statistical distribution of test cases must be controlled. Claessen and Hughes (2000) proposed composable random generators to test functional Haskell programs; we applied their idea and designed composable test-case generators for Lua (Ierusalimschy et al., 1996), a scripting language that drives our testing tool.

We claim the following contributions:

- *Composable random generators provide a concise way to control the distribution of test cases.* The generator for ANSI C is specified in about 100 lines of Lua (Section 3).
- *Random testing of calling conventions is effective.* We found 13 new bugs in production-quality compilers on

[1] Reported as GCC bug #18742.

```
 1  #include <stdarg.h>
 2  #include <assert.h>
 3  union  A {float a; double b;}
 4      c = { 52.54 };
 5  struct B {double d; int e;}
 6      h = { 78.01, 834 };
 7  union  C {short int f; char g;}
 8      i = { 68 };
 9  struct D {char j; double k;}
10      n = { 'c', 31.01 };
11  struct E {long long l; double m;}
12      o = { 167L, 17.2 };
13
14  union A
15  callee(struct D a, struct E b, ...)
16  {
17      va_list ap;
18      struct B x;
19      union  C y;
20      va_start (ap, b);
21      x = va_arg (ap, struct B); /* 3rd */
22      y = va_arg (ap, union C);  /* 4th */
23      assert (y.f == i.f);      /* fails */
24      va_end (ap);
25      return c;
26  }
27  int main( int argc, char **arg ) {
28      union A r;
29      r = callee (n, o, h, i);
30      return 0;
31  }
```

Figure 1: GCC 3.3 on MacOS X 10.3 passes `union C i` incorrectly to variadic function `callee`; the assertion in line 23 fails. The code was generated by our testing tool Quest and is slightly simplified for presentation. In a variadic function, extra arguments must be accessed using macro `va_arg`, which receives the argument's type and returns its value.

Unix systems (Section 4, Table 3). Finding such bugs takes typically a few minutes.

- *Manually coded C programs exercise only a small part of a calling convention.* This explains why users (and developers) haven't tripped over the bugs we found (Section 5).

We investigated random testing of calling convention with Quest, a new tool that generates programs in the style of Figure 1. We discuss prior and related work in Sections 6 and 7, and provide our conclusions in Section 8.

2. TEST GENERATION SCHEME

The generation of test cases is type-driven: we generate function declarations randomly and generate a test case for each declaration. For example, the declaration

```
char f(int, short*)
```

is enough to generate a function `f(int, short*)` and a function `void g(void)` that calls it, both shown in Figure 2.

```
int    x = 6362;   /* all random */
short *y = (short*) 6328282U;
char   z = 'q';
char f(int a, short* b) {
    assert(a == x);
    assert(b == y);
    return z;
}
void g(void) {
    char c;
    c = f(x,y);
    assert(c == z);
}
```

Figure 2: Code generation scheme for a function `char f(int, short*)`: function `g` calls `f` and passes values of global variables, which are checked by `f`; likewise for the return value of `f`.

Function `g` passes values to `f`, which checks that it receives the right values, and returns a value, which in turn is checked by `g`.

Given a declaration, we generate for each parameter and return type a global variable that we initialize with a random value. Function `g` passes these values to `f`, which uses assertions to check that each value it receives is indeed the one of the corresponding global variable. Likewise, to test the return value, `f` returns the value of a global variable, which is checked by `g`. Function `main` (not shown) finally calls `g`.

This code generation scheme extends to all simple and compound C types that can be passed to a function or be returned by it: floating-point types, pointers, arrays, structures, and unions. A pointer is treated as an unsigned integer; we only compare a pointer's value but do not allocate a value to point at. The value of a compound global variable like an array, structure, or union can be defined with a C initializer; this also works for a hierarchical type, like an array of structures.

Two compound values, like two structures, cannot be compared directly. We unfold them recursively and generate a sequence of assertions that compare values component by component.

The values we use to initialize global variables are selected randomly, without special attention to extremal values. This is sufficient since we don't apply any operations to them but test them for equality only.

Our Quest tool can generate variadic functions, whose best-known instance is `printf(char *fmt,...)`: such a function takes regular named arguments like `fmt`, plus extra arguments. A variadic function must access the unnamed extra arguments one by one using the `va_arg` macro from the ⟨`stdarg.h`⟩ header. This is demonstrated in the code in Figure 1.

Finally, Quest can emit the called function `f` (the callee) and the calling function `g` into two different files. This way they can be compiled by two different compilers to test their

consistency. For code generation this means that global variables and functions defined in one file must be declared `extern` in the other.

3. RANDOM DECLARATIONS

In principle it would be enough to generate function declarations (and from them programs) that use all legal ANSI-C features. But from a practical point of view it is desirable to control the statistical distribution of declarations: some compilers, like GCC, accept C programs beyond the ANSI standard, others support only a subset. Section 5 below also presents reasons why we believe that the distribution of test cases has a strong impact on their ability to uncover bugs. We therefore want distributions to be user-programmable and provide composable random generators as a solution (Claessen and Hughes, 2000). Readers not interested in the details of test-case generation and customization may safely skip this section.

The composition of test generators is expressed in Lua, a scripting language embedded in QUEST that drives the generation of test cases. Lua is a Pascal-like scripting language that is designed to be embedded into applications to make them scriptable by the user (Ierusalimschy et al., 1996; Ramsey, 2004). Lua's most prominent data type is *table*, an associative array that is used in Lua to model lists, arrays, and modules. QUEST, which is implemented in the ML dialect Objective Caml (Leroy et al., 2004), contains a Lua interpreter that provides random generators as Lua functions. By writing small Lua functions, a user can define generators, which in turn define test cases.

The sole rôle of Lua is to let the user control test case generation. The design of the test case generators itself is independent from Lua—we could have used any other scripting language, or none at all, if we had decided to provided only hard-coded generators.

A function declaration, which is the base for a test case, is characterized by three components: (1) a list of argument types, (2) a list of types for values passed as extra arguments to a variadic function, and (3) a return type. We generate test cases by using three generators, one for each component of a declaration.

A generator in general is a composition of simpler generators. The most basic generators provide a source for randomness; they are complemented by a set of generators for C types. These follow the abstract syntax for C types captured by the grammar c in Figure 3. Together they may be combined into generators for complex types, like a list of structures.

3.1 Type Generators

To characterize generators as they are available to the user in Lua, we give ML-style types to them: a generator that produces a C type has type c *gen*, a generator that produces values of type α has type α *gen*. The most basic generators produce the same simple C type in every run. These are listed at the top of Table 1: generator `char` produces type `char`, generator `long` type `long`, and so on.

A generator can take another generator as argument, for instance `unsigned`, which makes it a function of type c *gen* $\rightarrow$ c *gen*: `unsigned(char)` produces the type `unsigned char` from the generator for the `char` type. Likewise, a generator producing pointer types is obtained by applying `pointer` to *any* type generator.

The flexibility gained from composing complex generators from simple generators becomes more evident with arrays: the `array` generator takes two generators: one for types and one for numbers. The type generator determines the element type of the array, the number generator the length of the array. Every time the `array` generator is run, it runs the two generators as well. These could produce, for example, `unsigned char` and `3` in the first run, and `double*` and `2` in the second. This leads to two different array types: `unsigned char[3]` and `double*[2]`.

A generator for a structure (or union) type also takes two generators as arguments, one for the type and one for the number of members. Here however, the number generator is run first and returns how often the type generator is to be run to produce a list of member types. The type generator may return a totally different type in each run, which leads to a structure (or union) type with diverse member types.

$$
\begin{array}{rcl}
\textit{width} & ::= & \mathsf{char} \mid \mathsf{short} \mid \mathsf{int} \mid \\
 & & \mathsf{long} \mid \mathsf{long\ long} \\
\textit{fwidth} & ::= & \mathsf{float} \mid \mathsf{double} \mid \mathsf{long\ double} \\
\textit{sign} & ::= & \mathsf{signed} \mid \mathsf{unsigned} \\
\textit{member} & ::= & (\textit{name}, c) \\
c & ::= & \mathsf{void} \\
 & \mid & \mathsf{int}(\textit{sign}, \textit{width}) \\
 & \mid & \mathsf{float}(\textit{fwidth}) \\
 & \mid & \mathsf{array}(c, \textit{length}) \\
 & \mid & \mathsf{pointer}(c) \\
 & \mid & \mathsf{struct}(\textit{name}, \textit{member}, \ldots) \\
 & \mid & \mathsf{union}(\textit{name}, \textit{member}, \ldots)
\end{array}
$$

Figure 3: Abstract syntax for C types.

Generator	Generator Type	C Type
`char`	c *gen*	`char`
`long`	c *gen*	`long int`
`unsigned`	$c\ gen \rightarrow c\ gen$	unsigned type
`float`	c *gen*	`float`
`pointer`	$c\ gen \rightarrow c\ gen$	pointer type
`array`	$c\ gen \times num\ gen \rightarrow c\ gen$	`array`, n mem's
`structure`	$c\ gen \times num\ gen \rightarrow c\ gen$	`struct`, n mem's
`union`	$c\ gen \times num\ gen \rightarrow c\ gen$	`union`, n mem's

Table 1: Some generators for C types. They are available as Lua functions to compose high-level generators.

Generator	Generator Type	Value generated
`number`	$num \to num\ gen$	pick number from $0, \ldots, n-1$
`choose`	$num \times num \to num\ gen$	pick number from $m, \ldots, n$
`list`	$num\ gen \times \alpha\ gen \to (\alpha\ list)\ gen$	list with n members
`elements`	$\alpha\ list \to \alpha\ gen$	pick value from list
`oneof`	$(\alpha\ gen)\ list \to \alpha\ gen$	pick generator from list
`unit`	$\alpha \to \alpha\ gen$	constant generator
`iszero`	bool *gen*	indicate end of recursion
`smaller`	$(\alpha\ gen)\ list \to \alpha\ gen$	like `oneof`, limits recursion
`bind`	$\alpha\ gen \times (\alpha \to \beta\ gen) \to \beta\ gen$	monadic bind operator

Table 2: Primitive random generators; they are available as Lua functions.

3.2 Basic Generators

To build a generator whose output varies in every run we need some source of randomness. It is provided by the primitive generators shown in Table 2.

Generator `number` provides a random number, `choose` a number from an interval. Given a generator for a value, `list` creates a generator that produces a list of values whose length depends on a number generator. Given a list of values, `elements` produces a generator that picks one value at random in every run. This idea is raised to the next level with `oneof`: it takes a list of generators and picks a generator at every run.

3.3 The ANSI C Generator

The default generator for ANSI-compliant declarations in Quest is composed in about 100 lines of Lua code, from which we show a subset in Figure 4.

The test-case generator is defined by `ANSI.test`, which returns a table—denoted by curly braces—that binds three generators for arguments, extra arguments (for variadic functions), and the result type. Several simple generators for sizes and lengths are bound to names at the top of Figure 4.

The generator `ANSI.arg_` for argument types is *recursive*. It considers two cases: in the base case it returns an integer or float type, otherwise it selects with `smaller` from a list a generator that itself uses `ANSI.arg`. The two cases are necessary to ensure that recursion terminates.

3.4 Taming Recursion

The grammar for C types in Figure 3 is recursive, which leads to recursive generators. Without care, recursive generators could fail to terminate when run. To avoid this we limit the depth of recursion and therefore the size of a type.

The function that runs a generator takes two arguments: the generator, and the maximum recursion depth (which is 2 by default). The parameter for the depth is only passed behind the scene between generators and does not clutter their interface; three functions cooperate to ensure termination:

- Function `smaller` ($(c\ gen)\ list \to c\ gen$) takes a list of generators and selects one in every run. In addition, it decrements the depth before passing it to subordinated generators. Hence, only recursion going through `smaller` counts towards maximum the depth.
- Generator `iszero` (bool *gen*) generates true if the actual depth is zero.
- Function `bind` ($\alpha\ gen \times (\alpha \to \beta\ gen) \to \beta\ gen$) defines a generator and takes two arguments: a generator (like `iszero`) and a function. It passes the generated value to the function which returns a generator depending on the value. In our case, the generator returned by `ANSI.arg_` depends on the depth of the recursion.

```
ANSI.members    = choose(1,3)  -- for structs
ANSI.argc       = choose(1,10) -- argv length
ANSI.vargc      = choose(0,3)  -- var args
ANSI.array_size = choose(1,3)

function ANSI.arg_ (issimple)
    if issimple then
        return oneof { any_int, any_float }
    else
        return smaller -- like oneoef (c.f. 3.4)
        { any_int      -- signed/unsigned
        , any_float    -- all sizes
        , pointer(ANSI.arg)
        , array(ANSI.arg,ANSI.array_size)
        , struct(ANSI.arg,ANSI.members)
        , union(ANSI.arg,ANSI.members)
        }
    end
end
ANSI.arg = bind(iszero,ANSI.arg_)

function ANSI.test () return
    { args       = list(ANSI.argc,ANSI.arg)
    , varargs    = list(ANSI.vargc,ANSI.varg)
    , result     = ANSI.result
    }
end
```

Figure 4: The test generator for ANSI C in Lua. The generators `ANSI.result` for return types and `ANSI.varg` for extra arguments are omitted. Curly braces denote tables.

The generator abstraction is implemented as a monad. This

makes it easy to pass values like the depth behind the scene while maintaining full composability (Wadler, 1997). The monad also hides sources for member names like they are required for the implementation of `struct` and `union`.

4. EVALUATION

To evaluate the effectiveness of random testing we tried to find inconsistencies in function calls translated by Unix C compilers. We focused on production-quality commercial and open-source compilers but also included two compilers still under development. In particular:

- GCC, the GNU C compiler, including the experimental development version 4.0.0 (FSF, 2003).
- LCC, a retargetable ANSI C compiler (Fraser and Hanson, 1995).
- TCC, a small and fast ANSI C compiler for the Linux/ $x86$ platform (Bellard, 2004). This compiler is under early development.
- PathCC, a commercial compiler from PathScale Inc. with a special focus on performance and the generation of 64-bit code on the $x86$ architecture (PathScale, 2004).
- MipsPro, the ANSI C compiler for SGI workstations running the IRIX operating system (SGI, 1999).
- PGCC, a commercial compiler from with a special focus on performance on the $x86$ architecture (Portland Group, 2004).
- ICC, the commercial Intel C compiler for Linux on the $x86$ platform (Intel, 2004).

We tested these compilers on a number of platforms where the selection of platforms was influenced foremost by our access to them. As a consequence, we have tested more compilers on Linux/$x86$ than on any other platform.

Compilers GCC, ICC, and TCC support the C language beyond the ANSI standard. They support empty structures and unions, as well as arrays of length zero. When testing GCC, ICC, and TCC we generated programs that used these features. All other compilers we tested with ANSI-compliant code.

To find inconsistencies, we executed a loop for 15 minutes that generated code using QUEST, compiled it using the compiler under test, and ran it. Each generated program contained 20 test cases, that is, pairs of a calling and a called function. The loop was left immediately when either the compiler failed, or an inconsistency in the compiled code was found. As far as compilers supported them, we tried different optimization options (none, `-O`, `-O2`, `-O3`) but did not try options unrelated to optimization.

The essence of our test procedure fits in one long line of shell commands; we often used it to quickly test compilers for bugs:

```
while true; do
    quest > test.c      # generate test
    cc -o test test.c || break
    ./test || break     # run test
    echo -n .           # progress indicator
done
```

The infinite loop is left when the compiler fails, or, more likely, the program `test` raises an assertion failure. This would leave `test.c` as a test case to report to the developers. Most bugs showed up within a few minutes of testing, which inspired the title of the paper.

Table 3 shows our results. We found bugs[2] in all compilers except PGCC, and TCC. The 13 bugs that we found constitute two classes: 4 bugs that crashed a compiler and 9 bugs that showed up as inconsistencies. The three compiler crashes of GCC involved languages extensions, in particular the usage of empty structures. It is not clear what caused PathCC to crash.

We believe that the 3 crashes of GCC on Linux result from the same bug; we uncovered it with *different* test cases in *different* compiler versions. We could identify the bug by its distinctive error message from the register allocator. The same probably applies for one inconsistency found in GCC on MacOS X.

The 9 bugs that showed up as inconsistencies involve advanced function declarations: 4 bugs showed up in variadic functions, 2 bugs involved a `struct`- or `union`-typed parameter, and the bug where PathCC fails to pass a `float` the test function had the following declaration:

```
union A f(double a, union B b,
          struct C c, float d, struct E e)
```

Here we suspect that not the fourth parameter `float d` caused the problem but the other compound parameters.

4.1 Consistency Between Compilers

Two functions, one calling another, may be compiled by different compilers. When both compilers adhere to a platform-specific calling convention values should pass unaltered from one function to another. QUEST can generate appropriate test cases and we used this for a small experiment. We refrained from a more systematic test because they would be affected by the inconsistencies we had found already. It would be difficult to attribute an inconsistency to a particular compiler knowing that it had shown internal inconsistencies before.

We conducted two experiments with GCC 4.0.0, TCC 0.9.22, and LCC 4.2 on Linux/$x86$. We found that GCC and TCC agreed perfectly—we did not find any inconsistencies. On the other hand, GCC 4.0.0 and LCC 4.2 showed more inconsistencies than we expected from the one bug we found in LCC.

The inconsistencies between LCC and GCC could be explained with the lack of standards for calling conventions.

[2]The programs that uncovered these bugs are available at `http://www.cs.uni-sb.de/~lindig/quest/bugs/`

Compiler and Options	Platform	Symptoms and Comments
MipsPro 7.3.1.3m, `-O3`	Irix 6.5/MIPS	`struct` not passed correctly
GCC 2.95.3, `-O2`	SunOS 5.8/Sparc	`double` as var arg not passed correctly
GCC 2.95.4, `-O`	Linux/$x86$	compiler crashes, reported as bug #16819, fixed in GCC 3.4
GCC 3.2.2 `-O3`	Linux/$x86$	same as bug#16819?
GCC 3.3, none	Irix 6.5/MIPS	union as var arg not passed correctly, reported as bug #19268, fixed in GCC 3.4
GCC 3.3, none	MacOS X 10.3	involves GCC extension, reported as bug #18742, see Figure 1
GCC 3.3.3, `-O3`	Linux/$x86$	same as bug #16819?
GCC 4.0.0, none	MacOS X 10.3	same as bug #18742?
GCC 4.0.0	Linux/$x86$	no inconsistency found
LCC 4.2, none	Linux/$x86$	`double` var arg not passed correctly
PathCC 1.4, `-O2 -m32`	Linux/$x86$	`float` not passed correctly, bug reported, fixed in Release 2.0
PathCC 1.4, `-O2 -m32`	Linux/$x86$	`union` with `struct` not passed correctly, fixed in Release 2.0
PathCC 2.0, `-Ofast`	Linux/$x86$	compiler crashes with floating-point exception in "LNO" phase, reported as bug #5273, fixed in upcoming Release 2.1
ICC 8.1, none	Linux/$x86$	var arg not passed correctly, reported as bug #292019
TCC 0.9.22	Linux/$x86$	no inconsistency found
PGCC 5.2	Linux/$x86$	no inconsistency found

Table 3: Bugs found with Quest in compilers on Unix systems.

Architecture manuals like Intel (2003) typically specify a calling convention for simple values but omit any discussion how to pass structures or unions. Here, compiler writers are left on their own or are forced to reverse-engineer existing compilers.

5. NON-RANDOM C CODE

Surprised by the many inconsistencies we found we looked for some explanation. For this we analyzed the function definitions of real-world programs and programs in the GCC test suite; we compared them to 200 programs generated by QUEST.

- As a representative collection of real-world programs we looked at the SPEC CPU 2000 benchmark suite. The SPEC benchmark suite is a standardized set of programs to evaluate the performance of a computer's processor, memory architecture, and compilers (SPEC, 1999). The suite is intended to cover a range of typical compute-intensive applications and we looked specifically at 12 programs[3] written in C.
- The GCC 4.0.0 source code contains various test suites that are used during development of the compiler. We analyzed the `gcc.c-torture` test suite that contains 1,638 (short) C files; about 5% of them we could not analyze due to syntactical problems.

As a fairly coarse measure, we analyzed the programs statically using CIL, a framework for analyzing C programs (Necula et al., 2002). On Apple MacOS 10.3 we measured in particular for function definitions:

- the number of arguments;
- the number of variadic function definitions;
- the distribution of argument types;
- the distribution of return types.

The SPEC benchmark, the GCC test suite, and the QUEST-generated code contain 5,566, 5,055, and 4,200 functions, respectively. From these, the following fractions were variadic: 14 (or 0.3%) in SPEC, 137 (2.7%) in GCC, and a substantial 985 (23.5%) in QUEST-generated code.

Figure 5 summarizes the distribution of types and argument numbers for our subjects. In the SPEC benchmark suite over 80% of argument types are either `int` or a pointer, and over 95% of functions either return no value, an `int`, or a pointer. Types are similarly distributed in the GCC test suite; pointer and `int` dominate the argument types, and `void` and `int` the return types. Essentially no function in the SPEC or GCC suite returns or receives a structure or union. Even simple types like `char` or `float` are almost absent among return types. This is very different for QUEST-generated programs: structures, unions, and floating-point types constitute a sizeable part of the distribution.

The length of argument lists of programs in the SPEC and GCC test suites are heavily skewed towards functions with less than 3 arguments. There are, however, a curious 13.1% percent of functions with 6 arguments in the GCC test suite. The maximum argument length in the SPEC suite was 17, and 32 in the GCC suite. Again, the programs generated by QUEST show a much more even distribution with up to 10 arguments. (There are 47.6% of functions with declaration `void f()`: those call the function under test and are somewhat misleading.)

Currently QUEST does not generate functions with more than 10 parameters. We chose this limit because we know of no calling convention that puts more than 8 parameters

[3]The SPEC CPU 2000 suite contains 3 more programs written in C that we could not analyze using CIL: `perlbmk`, `vortex`, and `equake`.

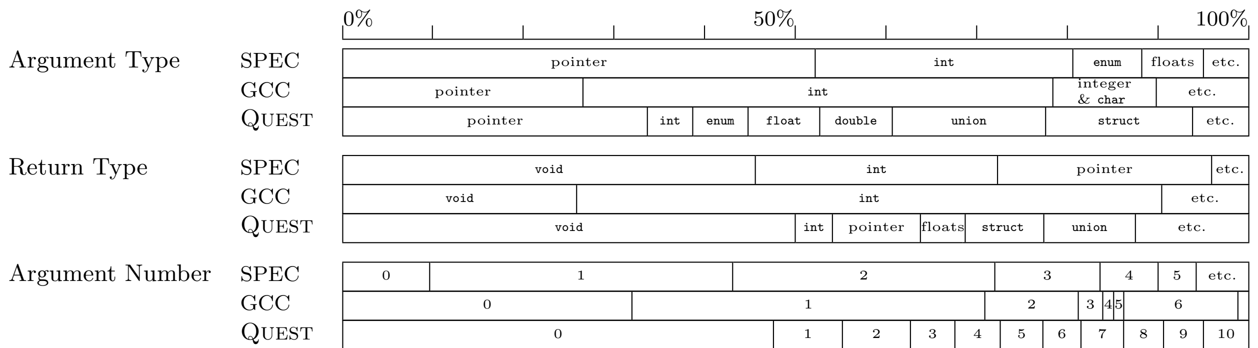

Figure 5: Distribution of declarations in C programs: types in argument positions, types in return position, and length of argument lists. Shown are the distributions for 12 programs from the SPEC CPU 2000 benchmark, for the GCC 4.0.0 test suite `gcc.c-torture`, and for code generated by Quest. Declarations generated by Quest are more varied than those found in the SPEC and GCC suites.

in registers. We thus are confident that 10 parameter suffice to exercise all registers used for parameter passing, as well as some stack positions. In any case, this could be easily changed by defining a different generator `ANSI.argc` in Figure 4.

Our results are based on static analysis and don't reflect how often a certain function is actually used in a program run. Theoretically, a statically rare declaration could dominate the dynamic execution. However, early experiments suggest that the statically dominant types in the SPEC benchmark are even more dominant at run time.

In the SPEC and the GCC test suite simple types and short argument lists are predominant. We suspect that they never execute a large part of a calling convention implementation in a compiler. Both users (good) and developers (bad) are thus unlikely to find latent bugs in the implementation of function calls. We believe that the decidedly wider spectrum of declarations generated by QUEST is responsible for uncovering bugs in otherwise mature compilers.

6. PRIOR WORK

Bailey and Davidson (1996) had previously tested the conformance of C function call implementations with calling conventions. Conformance with a calling convention implies consistency but requires a specification to test against. Bailey and Davidson's test methodology builds upon their formalization of calling conventions (Bailey and Davidson, 1995).

A calling convention is intended to ensure interoperability between compilers and libraries from different vendors. It details which registers and stack locations to use when passing parameters between functions. Calling conventions are defined in architecture manuals issued by hardware vendors, which makes them platform-dependent. They are typically informal and sometimes confusing to the degree that compilers failed even on their examples, as noted by Bailey and Davidson (1996). The especially arcane C calling conventions are the main difficulty to implement C function calls correctly.

Bailey and Davidson formalized calling conventions. Their

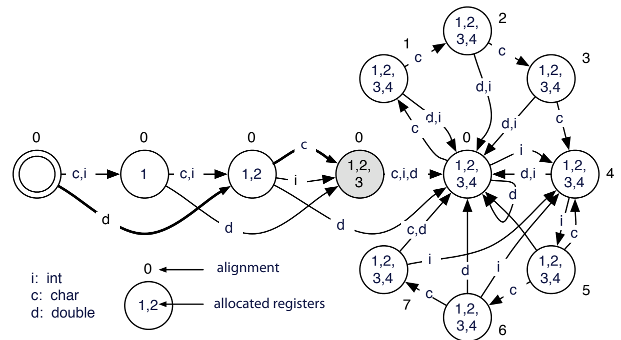

Figure 6: Automaton for a simple calling convention (Bailey and Davidson, 1995). Numbers inside nodes denote allocated registers, numbers next to them alignment. A parameter list `(double, char)` allocates registers 1 to 3 and no stack positions.

model is based on automata, so-called P-FSA; an automaton for a simplistic calling convention is shown in Figure 6. They derive test cases from such an automaton to test the conformance of a compiler with the modelled calling convention.

An automaton models the allocation of registers and stack locations for parameter passing. A state represents a set of allocated resources, a transition is labeled with a parameter type for which a resource is allocated in the next state. Since the number of function parameters are unbound, an automaton is infinite in principle. To make it finite, Bailey and Davidson track only register resources precisely. Stack locations, on the other hand, are grouped into equivalence classes based on their alignment: for example, all 8 byte-aligned stack locations are represented by the same node. Nodes for stack locations are often reachable from each other and lead to cyclic automata.

Every path in an automaton represents a declaration, each of which is a candidate for a test case. In the presence of cycles there are infinite many; looking at the finite many acyclic paths, Bailey and Davidson found more than 10^8 of them

on some RISC architectures—still too many to test. They devised a smart heuristic to derive a manageable number of test cases.

Bailey and Davidson tested the conformance of compilers on the MIPS platform, whose calling convention is notorious for being confusing and difficulty to implement. They found 9 bugs related to consistency and 13 bugs related to parameter passing between functions compiled by different compilers.

7. RELATED WORK

The test methodology of Bailey and Davidson (1996) validates the conformance of a function call implementation based on test cases derived from a model of a calling convention. The validation is almost as good as a verification since the method guarantees a high degree of coverage. As its main drawback it depends on a *target-specific* model or formal specification, which is not readily available and the reason why Bailey and Davidson's methodology wasn't widely adopted.

Random testing is target-independent and requires no specification. Of course, we cannot argue for exhaustiveness: by analogy, only random paths in Bailey-Davidson automaton would be used as test cases. We cannot verify the conformance with a calling convention directly. But testing external consistency against a conforming reference compiler could detect violations of a calling convention. Overall, the main attraction of random testing is simplicity.

Random testing in general is surrounded by a discussion of its effectiveness in comparison to other methods (Duran and Ntafos, 1984). It attracted a lot of theoretical attention with the goal to measure its effectiveness, or to derive upper bounds for remaining bugs after testing (Bernot et al., 1997; Chen and Yu, 1996; Tsoukalas et al., 1993).

QuickCheck by Claessen and Hughes (2000) introduced the idea of composable generators as first-class values. They use composable generators to test Haskell programs. We adopted this idea to create a domain-specific extension of Lua (Ierusalimschy et al., 1996) for the generation of C declarations.

Celentano et al. (1980) present a grammar-driven program generator that covers the entire input language for a compiler. As such, it has to cope with many more context-sensitive aspects than we do for C types. As the main difference, Quest-generated tests are self-evaluating because they test consistency of a compiler.

The correct translation of function calls is only a small part of a totally correct compiler. At least for optimizing compilers, verified correctness is still a research problem. A promising trend is proof-carrying code (Necula, 1997): instead of verifying the entire compiler, verify the results of an individual translation. For this a compiler augments the code it emits with annotations that can be verified before executing the code. The successful verification of annotations implies certain code characteristics. Because of its smaller code base, a verifier is easier to implement correctly than an optimizing compiler.

8. CONCLUSIONS

Random testing the consistency of function calls has revealed 13 new bugs in mature C compilers (see Table 3). This leads us to conclusions about both this specific property of C compilers, and our test method.

Despite their long history C compilers still contain bugs in the implementation of functions calls. These are provoked by arcane C calling conventions that compilers must implement to ensure interoperability with existing code. By their nature calling conventions are platform specific and thus a compiler cannot implement a general solution—the problem is therefore unlikely to vanish. This suggests an opportunity for a framework to implement calling conventions more easily by providing common abstractions, for example for the stack frame layout (Lindig and Ramsey, 2004).

Our analysis of typical C code shows that it is unlikely to trigger a remaining inconsistency in a compiler. All bugs that we found were in advanced aspects of calling conventions, like structures passed by value to a variadic function. These are almost absent in real-world code, but probably also in compiler test suites. We therefore suggest compiler developers to adopt our tool as a complement to existing regression tests.

Random testing of consistency is as effective as specification-based testing, yet easier; indeed it is so easy, that anyone could use it in a few lines of shell code to automatically find bugs in a compiler. The simplicity stems from a number of sources: First, testing for consistency requires no specification. Second, composable random generators are a powerful device to compose highly structured test data while still controlling their statistical distribution. And third, tests are self-evaluating: running a test is equivalent to running code that the program under test generated. This suggests that random testing of consistency is especially well suited for compilers and interpreters and is the main result of this paper.

An open problem is how to avoid finding the same bug again. Distinct failing test cases might be caused by the same bug, which is undecidable in general. It would be helpful to fingerprint the execution of test cases such that similar executions lead to similar fingerprints which could identify bugs.

An idea for future work is to tie Quest, a compiler, and the execution of generated code into a feedback loop. This would allow for the automatic minimization of a failed test using the Delta Debugging algorithm (Zeller, 2001): after a failing test was found, the test generator would try to find a minimal failing test case. Smaller test cases make it easier for developers to locate a bug.

Quest is open source; the code and Quest-generated test cases are are available from `http://www.st.cs.uni-sb.de/~lindig/src/quest/`.

Acknowledgements

Christopher Krauß conducted the static analysis of the SPEC benchmarks. Discussions with Stephan Neuhaus, Tom Zimmerman, and Andreas Rossberg helped to improve this paper.

References

Mark W. Bailey and Jack W. Davidson. A formal model and specification language for procedure calling conventions. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 298–310, January 1995.

Mark W. Bailey and Jack W. Davidson. Target-sensitive construction of diagnostic programs for procedure calling sequence generators. *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 31(5):249–257, May 1996.

Fabrice Bellard. *TCC – Tiny C Compiler's Unix Manual Page*, November 2004. URL `http://www.tinycc.org/`. Release 0.9.22. Manual part as part of the source code.

Gillis Bernot, Laurent Bouaziz, and Gall Gall. A theory of probabilistic functional testing. In *Proc. of the 1997 International Conference on Software Engineering*, pages 216–226, 1997.

Augusto Celentano, Stefano Crespi-Reghizzi, Pierluigi Della Vigna, Carlo Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software—Practice and Experience*, 10(11):897–918, November 1980.

Tsong Yueh Chen and Yuen-Tak Yu. On the expected number of failures detected by subdomain testing and random testing. *IEEE Transactions on Software Engineering*, 22 (2):109–119, February 1996.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279, September 18–21 2000.

Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

FSF. *GCC Internals Manual.* Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA, 2003. URL `http://gcc.gnu.org/`.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software — Practice and Experience*, 26(6): 635–652, 1996. URL `http://www.lua.org/`.

Intel. *IA-32 Intel Architecture Software Developers's Manual, Vol. 1.* Intel Corporation., P.O. Box 7641, Mt. Prospect, IL 60056, 2003.

Intel. *Intel C++ Compiler for Linux Systems User's Guide.* Intel Corporation, 2200 Mission College Blvd., Santa Clara, CA 95052, USA, June 2004. Release 8.1, Document Number 253254-031.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System 3.08: Documentation and User's Manual.* INRIA, 2004. Available from `http://caml.inria.fr`.

Christian Lindig and Norman Ramsey. Declarative composition of stack frames. In Evelyn Duesterwald, editor, *Proc. of the 14th International Conference on Compiler Construction*, number 2985 in Lecture Notes in Computer Science, pages 298–312. Springer, 2004.

George C. Necula. Proof-carrying code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 15–17, 1997.

George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of Conference on Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, 2002.

PathScale. *PathScale EKO Compiler Suite Release Notes.* PathScale, Inc, 477 N. Mathilda Avenue, Sunnyvale, CA 94085, USA, 2004. URL `http://www.pathscale.com/`. Release 1.4.

Portland Group. *PGI User's Guide.* The Portland Group Compiler Technology, 9150 SW Pioneer Ct, Suite H, Wilsonville, OR 97070, USA, June 2004. URL `http://www.pgroup.com/`. Release 5.2.

Norman Ramsey. Embedding an interpreted language using higher-order functions and types. *Journal of Functional Programming*, 2004. To appear. Preliminary version appeared on pages 6–14 of *Proc. of the ACM Workshop on Interpreters, Virtual Machines, and Emulators*, June 2003.

MIPSpro ANSI C Compiler Release Notes. SGI, 1500 Crittenden Lane, Mountain View, CA 94043, USA, 1999. URL `http://www.sgi.com/`. Release 7.3.

SPEC. SPEC CPU 2000 benchmark suite. Standard Performance Evaluation Corporation, December 1999. URL `http://www.spec.org/`. Version 1.0.

Markos Z. Tsoukalas, Joe W. Duran, and Ntafos Ntafos. On some reliability estimation problems in random and partition testing. *IEEE Transactions on Software Engineering*, 19(7):687–697, July 1993.

Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.

Andreas Zeller. Automated debugging: Are we close? *Computer*, 34(11):26–31, November 2001.