Random Testing for C and C++ Compilers with YARPGen

VSEVOLOD LIVINSKII, University of Utah and Intel Corporation, USA
DMITRY BABOKIN, Intel Corporation, USA
JOHN REGEHR, University of Utah, USA

Compilers should not crash and they should not miscompile applications. Random testing is an effective method for finding compiler bugs that have escaped other kinds of testing. This paper presents Yet Another Random Program Generator (YARPGen), a random test-case generator for C and C++ that we used to find and report more than 220 bugs in GCC, LLVM, and the Intel® C++ Compiler. Our research contributions include a method for generating expressive programs that avoid undefined behavior without using dynamic checks, and *generation policies*, a mechanism for increasing diversity of generated code and for triggering more optimizations. Generation policies decrease the testing time to find hard-to-trigger compiler bugs and, for the kinds of scalar optimizations YARPGen was designed to stress-test, increase the number of times these optimizations are applied by the compiler by an average of 20% for LLVM and 40% for GCC. We also created tools for automating most of the common tasks related to compiler fuzzing; these tools are also useful for fuzzers other than ours.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; **Source code generation**.

Additional Key Words and Phrases: compiler testing, compiler defect, automated testing, random testing, random program generation

ACM Reference Format:
Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (November 2020), 25 pages. https://doi.org/10.1145/3428264

1 INTRODUCTION

Compilers should not crash and they should not generate output that is not a faithful translation of their inputs. These goals are challenging to meet in the real world. For one thing, compilers often have significant code churn due to changing source languages, new target platforms to support, and additional requirements such as providing new optimizations. For another, since it is crucial for compilers to be as fast as possible, they use sophisticated, customized algorithms and data structures that are more difficult to get right than straightforward ones.

Although random test-case generation is an effective way to discover compiler bugs [Eide and Regehr 2008; Le et al. 2014, 2015; Nagai et al. 2012, 2013, 2014; Sun et al. 2016; Yang et al. 2011], experience shows that any given random test-case generator, testing a particular compiler, will eventually reach a saturation point [Amalfitano et al. 2015] where it finds very few new bugs. However, even after one test-case generator has reached this saturation point, the ability of a different generator to find bugs seems to be mostly unimpeded. For example, even after

Authors' addresses: Vsevolod Livinskii, vlivinsk@cs.utah.edu, University of Utah, Intel Corporation, USA; Dmitry Babokin, dmitry.y.babokin@intel.com, Intel Corporation, USA; John Regehr, regehr@cs.utah.edu, University of Utah, USA.

2475-1421/2020/11-ART196
https://doi.org/10.1145/3428264

Csmith [Yang et al. 2011] had reached apparent saturation on the then-current versions of GCC and LLVM, other techniques were able to discover large numbers of residual bugs [Le et al. 2014, 2015; Sun et al. 2016]. In other words, a test-case generator reaches a saturation point not because the compiler is bug-free, but rather because the generator contains biases that make it incapable of exercising specific parts of the compiler implementation.

This paper presents YARPGen, a new random test-case generator for C and C++ compilers that we created to explore several research questions:

- What bugs have been missed by previous compiler testing methods, including random testing?
- Is it possible to generate expressive random programs that are free of undefined behaviors without resorting to the kind of heavy-handed dynamic safety checks that Csmith used?
- Is it possible to build mechanisms into a generator that help it target different parts of an optimizer?

YARPGen is based on the idea that a generator should support *generation policies*: mechanisms for systematically altering the character of the emitted code, with the goal of making it possible for the generator to avoid, or at least delay, saturation. YARPGen's other main contribution is to emit code that is free of undefined behavior without resorting to the heavy-handed wrapper approach employed by Csmith [Yang et al. 2011] or staying within a narrow subset of the language, as generators such as Orange [Nagai et al. 2012, 2013, 2014] have done. So far, this generator has found 83 new bugs in GCC, 62 in LLVM, and 76 in the Intel® C++ Compiler.

The remainder of the paper is structured as follows. Section 2 briefly introduces random testing for C and C++ compilers. Section 3 describes our approach to random differential compiler testing. Section 4 presents the results of our testing campaign. Section 5 surveys previous work, and Section 6 concludes.

2 BACKGROUND

In this paper, we are concerned with two kinds of compiler bugs. First, the compiler might crash or otherwise terminate without generating object code. We call this an *internal compiler error*, or ICE. Second, the compiler may generate compiled object code and terminate normally, but the generated code does not perform the right computation. We call this a *wrong code bug*, or a *miscompilation*. Compilers can go wrong in other ways, such as looping forever, but in our experience, these failures are less common, and in any case, the testing campaign described in this paper did not pursue them.

2.1 Random Differential Testing

Like several previous compiler testing efforts [Ofenbeck et al. 2016; Yang et al. 2011], YARPGen is intended to be used for differential testing [McKeeman 1998]: it generates a program that follows the rules for C and C++ programs in such a fashion that when the random program is executed, it must produce a specific output. It turns out that YARPGen has enough information about the generated program that it could generate totally self-contained test cases that check their own computation against the correct result. A disadvantage of that approach is that it makes it very hard to use an automated test case reduction using a tool such as C-Reduce [Regehr et al. 2012], since many reduction steps would require a coordinated change to both the random computation and the expected answer. Differential testing avoids this problem. If independent compiler implementations often contained correlated bugs that caused them to return the same wrong answer, differential testing would not be a good idea. Empirically, this happens very rarely, and even if two compilers had the same bug, it would be noticed when a third compiler was added to the mix.

2.2 Undefined Behavior, Unspecified Behavior, and Implementation-Defined Behavior

A significant practical problem with using random testing to find miscompilation bugs is that C and C++ are *unsafe* programming languages: when a program executes an erroneous action such as touching heap storage that has been freed, the C or C++ implementation does not typically flag the violation by throwing an exception or terminating the program. Rather, the erroneous program may continue to execute, but with a corrupted memory state. Thus, randomly generated test cases that are erroneous are useless for differential testing. There are several hundred kinds of untrapped errors in C and C++ that are collectively referred to as *undefined behaviors* (UBs). All of these must be avoided in randomly generated code.

A related but less serious class of program behaviors is *unspecified behaviors* where the C or C++ implementation is allowed to choose one of several alternatives, but there is no requirement for consistency. An example is the order of evaluation of arguments to a function. Randomly generated programs can include unspecified behaviors, but their final answer should not depend on which behavior the compiler picked.

Finally, C and C++ have many *implementation-defined behaviors* where the compiler is allowed to make a choice, such as the range of values that may be stored in a variable of `int` type, but the behavior must be documented and consistent. Since it is impractical to avoid relying on implementation-defined behaviors in C and C++ code, programs generated by YARPGen do the same thing that basically all real C and C++ applications do: they rely on a subset of these behaviors that—for a given target platform—are reliably common across all modern implementations of these languages.

3 YARPGEN

This section describes how YARPGen can randomly generate loop-free C and C++ that is expressive and also statically and dynamically compliant with the rules specified in the language standards.

3.1 Avoiding Undefined Behavior

Programs produced by YARPGen must never execute operations that have undefined behavior. We can divide the undefined behaviors in C and C++ into those that are easy to avoid, and those that are hard to avoid.

The UBs that are easy to avoid generally correspond to surface-level, static properties of generated code. For example, the requirement that a pointer is never converted to a type other than an integer or pointer type is easy to avoid because YARPGen's type checker disallows code that would violate this constraint. Similarly, we avoid using uninitialized data by construction, since variables are initialized at their point of definition. Many undefined behaviors are avoided simply by not generating the program constructs that would lead them to happen. For example, the requirement that standard header files are not included while a macro with the same name as a keyword is defined is satisfied by not using the preprocessor in generated code. Similarly, restrictions on use of `setjmp` and `longjmp` cannot be violated because we do not generate those constructs at all.

UBs that are hard to avoid are ones such as integer overflows, that depend on dynamic characteristics of the execution of generated code. There are a number of possible strategies for achieving this goal:

- Generate tests in a fashion that is syntactically guaranteed to avoid UB, for example, by not generating arithmetic or pointers. Quest [Lindig 2005] used this method, which we rejected because it significantly limits the expressiveness of the generated code.

- Generate code without static constraints, inserting dynamic checks as needed to guard against undefined behaviors. Csmith [Yang et al. 2011] used this method, which we believe sacrifices expressiveness by obscuring potentially optimizable patterns behind ubiquitous safety checks.
- Generate programs without worrying about undefined behaviors, and then use a checking tool to discard those that are undefined. This has worked well when generated programs are quite small [Le et al. 2015], but YARPGen is intended to generate large programs to amortize compiler startup costs. The resulting programs would almost always be undefined.
- Run static analyses to conservatively avoid undefined behaviors during generation. Csmith used this strategy for pointer-related UBs and the Orange family of generators [Nagai et al. 2012, 2013, 2014] used it pervasively.

YARPGen uses the last strategy: it interleaves analysis and code generation in order to reliably and efficiently generate code that is free of UB, without unnecessarily sacrificing expressiveness. The strategy is to perform a very limited form of backtracking while generating expressions, converting operations that trigger undefined behavior into similar operations that are safe. This is described in detail below.

3.2 Generating Code

Creating the environment. First, YARPGen creates a collection of composite data types that will be used by the generated code, storing them into a *type table*. Types are created iteratively: the first array or struct can only contain elements with primitive types (signed and unsigned varieties of all integer types, pointers, and—for C++—bools), but subsequent structs and arrays can contain elements of previously generated types. Some structs contain bitfields. The rules that guide bitfield generation are different for C and C++. Specifically, in C they can only have `int` or `unsigned` type and their size should be less than the size of the base type. C++ lacks these limitations. YARPGen respects each language's rules when generating code in that language.

Next, YARPGen generates a set of global variables with primitive and composite types, storing them in a *symbol table*. Each global variable belongs to one of these categories:

- inputs: these variables are read but never written after initialization; their value is constant during test case execution
- outputs: these variables are written but never read by the test function
- mixed: these are both read and written

Input and output variables simplify UB-avoidance (Section 3.1). Some input variables are randomly `const`-qualified. When YARPGen is used to generate C++ code, some struct members are randomly given the `static` qualifier.

Creating statements. Each time YARPGen runs, it creates a set of—usually very large—functions. During generation, these functions are stored in a custom intermediate representation that supports static analysis and incremental construction.

The generation of each test function is top-down: a test function consists of a single top-level block, which contains a list of statements. Each statement does one of:

- Declares a new local variable of randomly chosen integer type, assigning it an initial value from a random arithmetic expression.
- Assigns the result of a random arithmetic expression into an existing variable.
- Begins a new conditional. Each conditional results in the generation of a random Boolean expression and one or two new blocks of statements. Once the block nesting depth reaches a configurable limit, generation of additional conditionals is disabled.

Creating expressions. YARPGen's expressions are generated top-down. They are free of side effects such as embedded assignment operators (e.g., `5 + (y = 3)`), function calls, and the `++` and `--` operators. Our motivation for omitting some of these language features is that they are lowered away early during compilation, whereas we are most interested in stress-testing middle-end compiler optimizations, which never see these language-level operators. A more detailed discussion of the limitations can be found in Section 3.6.

Each node in an expression is randomly one of:

- a unary, binary, or ternary operator
- a type cast
- a common subexpression (discussed in Section 3.3)
- a data element: a local or global variable, a struct member, an array element, a dereferenced pointer, or a constant

Recursive generation sometimes self-limits, for example, when a leaf node is selected randomly. Otherwise, when a configurable expression depth limit is reached, the generator forces a leaf to be selected.

Table 1. When an unsafe condition holds, YARPGen avoids undefined behavior by replacing the problematic operation. MIN and MAX are INT_MIN and INT_MAX for operations performed on `int`-typed values, or LONG_MIN and LONG_MAX for operations performed on `long`-typed values. (Due to the "usual arithmetic conversions" in C [International Organization for Standardization 2011] and C++ [International Organization for Standardization 2012], mathematical operations are never performed on `char`- or `short`-typed values.) MSB stands for Most Significant Bit. The "signed or unsigned" criteria are more complicated for shifts because the operands to a shift are not converted to a common type.

Operation	Unsafe condition	Signed or unsigned?	Replacement
-a	a == MIN	S	+a
a + b	a + b > MAX || a + b < MIN	S	a - b
a - b	a - b > MAX || a - b < MIN	S	a + b
a * b	a * b > MAX || a * b < MIN , where a != MIN && b != -1	S	a / b
a * b	a == MIN && b == -1	S	a - b
a / b	b == 0	S or U	a * b
a / b	a == MIN && b == -1	S	a - b
a % b	b == 0	S or U	a * b
a % b	a == MIN && b == -1	S	a - b
a << b	MIN < b < 0	a is U && b is S	a << (b + c), where c ∈ [-b; -b + bit_width(a))
a << b	MIN < b < 0	a is S && b is S	a << (b + c), where c ∈ [-b; -b + bit_width(a) - MSB(a))
a << b	b == MIN	a is U or S && b is S	a
a << b	b >= bit_width(a)	a is U && b is U or S	a << (b - c), where c ∈ (b - bit_width(a); b]
a << b	b>= bit_width(a)	a is S && b is U or S	a << (b - c), where c ∈ (b - bit_width(a) + MSB(a); b]
a >> b	MIN < b < 0	a is U or S && b is S	a >> (b + c), where c ∈ [-b; -b + bit_width(a))
a >> b	b == MIN	a is U or S && b is S	a
a >> b	b >= bit_width(a)	a is U or S && b is U or S	a >> (b - c) c ∈ (b - bit_width(a); b]
a >> b †	MIN < a < 0	a is S && b is U or S	(a + MAX) >> b
a >> b †	a == MIN	a is S && b is U or S	b

† implementation-defined behavior

Avoiding UB in expressions. An expression is a tree whose leaves are variables and constants. A randomly-generated expression is likely to execute one or more operations with undefined behavior when it is evaluated. YARPGen's procedure for avoiding UB works as follows.

First, the expression is type-checked in bottom-up fashion. This is done not to satisfy the C and C++ type checkers (every expression that we generate passes these checkers), but instead because the type of every subexpression needs to be known before we can analyze the flow of values through the expression. Second, again bottom-up, the values being processed by the expression are analyzed and expressions are rewritten as necessary to avoid UB.

Input variables are easy to analyze: they always hold their initial values. Output variables are also easy: they never appear in expressions. Mixed variables are updated by assignments, so their values must be tracked. Values are tracked concretely: YARPGen does not use a conservative abstract interpreter. This is possible because the generated code is loop-free (but also see Section 3.5).

When YARPGen detects an operation with undefined behavior, it performs a local fix, rewriting the operation into a form that does not trigger UB. The rewrites are shown in Table 1. For example, assume that the expression `-x` has the type `long` and `x` is a variable holding the value `LONG_MIN`. Since this expression is unsafe and must not be executed by a test program, YARPGen rewrites it as `+x`. We ensured that the rewrites in Table 1 eliminate UB in all cases by performing exhaustive testing of 4-bit values, and we also formally verified a subset of the rewrites at their full widths (32 and 64 bits) using the Z3 theorem prover. The verification script is available online[1].

The key invariant during this bottom-up process is that all subtrees of the node currently being analyzed are UB-free. Thus, when generation finishes, the entire expression is UB-free. This property can then be extended to other expressions by analyzing each expression in the order in which it will be evaluated during program execution. Conditionals do not present a problem for our approach because the analysis always knows which way a given conditional will go when the program executes.

This method for avoiding UB has several advantages over a more powerful backtracking scheme that could, for example, delete subtrees of expressions and regenerate them until UB-freedom is achieved. We implemented such a scheme and found that it usually requires multiple iterations before it succeeds, and that sometimes it paints itself into a corner and becomes stuck; one of the reasons that this happens is that parameter shuffling (Section 3.3) can significantly reduce the number of options available to YARPGen. It would be possible to extend the backtracking scheme to get unstuck by backtracking further (and perhaps all the way back to the start of program generation), but the implementation would be more involved. In contrast, YARPGen's rewrite rules are very efficient, never get stuck, and always succeed on the first try.

One might think that by generating programs whose behavior can be predicted statically, compiler testing would suffer because the programs would end up being too simple and would be mostly optimized away. This would only be the case if the compilers under test used link-time or whole-program optimization techniques. In other words, YARPGen relies on separate compilation to stop the compiler from propagating values between test initialization and test functions. If, in the future, compilers that we want to test make it difficult to disable link-time or whole-program optimizations, we may be forced to use a stronger optimization barrier, for example putting test code into a dynamically loaded library. We have tested this; it is trivial and does not require any changes to YARPGen itself, but rather only to the commands used to compile its output.

Lowering IR to the target language. When IR for the test functions is completely generated, it is lowered to either C or C++ and stored to a file. Lowering is a top-down recursive procedure. When lowering to C++, each array is randomly lowered to a C-style array, a `std::valarray`,

[1]https://github.com/intel/yarpgen/blob/v1/rewrites.py

a `std::vector`, or (for C++14 and newer) a `std::array`. There are also differences in array initialization across different versions of the C++ standard that must be taken into account.

Generating the test harness. Besides the file containing the test functions, YARPGen generates two additional files for each test case. First, a header containing declarations for all generated types, for the test functions, and for all global variables. Second, a driver file that defines the `main` function and defines (and optionally initializes) all global variables used by the test functions. The `main` function calls the test functions, computes a checksum by hashing values found in all output and mixed global variables (including struct fields and array elements), and then prints the checksum.

Given a particular set of implementation-defined behaviors, the C or C++ standard that is in effect completely determines the value of the checksum. In other words, all correct compilers, at all optimization levels, must generate executables that print the same checksum, which then serves as the key for differential testing. Any time changing the compiler or compiler options changes the checksum, we have found a compiler bug.

3.3 Generation Policies

When many random choices are drawn from the same distribution, randomly generated test cases can all end up looking somewhat alike. Our hypothesis was that this phenomenon reduces diversity in generated code and, in the long run, leads to less effective testing.

For YARPGen, we developed *generation policies* that systematically skew probability distributions in ways that are intended to cause certain optimizations to fire more often. Some distribution changes are applied in randomly chosen sub-regions of the generated code. Since different generation policies typically compose gracefully, YARPGen allows them to overlap arbitrarily, leading to additional diversity in code generation. Other generation policies apply to the entire test; in this case, to get diversity in tests, we rely on the fact that YARPGen will typically be run thousands or millions of times during a testing campaign.

These policies are drawn from our knowledge of how optimizing compilers work, and where bugs in them are likely to be found; it is not clear to us that this kind of insight can be automated in any meaningful fashion. We empirically evaluate the effectiveness of these generation policies in Section 4.

Arithmetic, logical, and bitwise contexts. An expression built from operators chosen from a uniform random distribution is less likely to contain dense clusters of, for example, bitwise operators, that would allow an optimization pass to perform rewrites such as these:

```
(x | c1) ^ (x | c2) → (x & c3) ^ c3 where c3 = c1 ^ c2
(x & c1) ^ (x & c2) → (x & (c1^c2))
(x & ~y) | (x ^ y) → (x ^ y)
```

These examples are from comments in LLVM's `reassociate` pass[2] and instruction simplification analysis.[3] The first transformation is intended to create a situation where common subexpression elimination can further optimize the code; the second and third are standard peephole optimizations.

YARPGen randomly chooses regions of code, or parts of expression trees, to use restricted subsets of mathematical operations, in order to more effectively trigger bugs in optimizations. These contexts are:

- additive: addition, subtraction, negation
- bitwise: binary negation, and, or, xor

[2]https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Transforms/Scalar/Reassociate.cpp#L1203-L1307
[3]https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Analysis/InstructionSimplify.cpp#L2138

- logical: and, or, not
- multiplicative: multiplication, division
- bitwise-shift: binary negation, and, or, xor, right/left shifts
- additive-multiplicative: addition, subtraction, negation, multiplication, division

An expression built using the bitwise-shift context might look like this:

```
a = ~b & ((c | d) ^ (e << 4));
```

Policies for constants. To ensure stress-testing of constant folding and related peephole optimizations, parts of expression trees are randomly selected to have all leaves as constants, or half of the leaves as constants. Code generated in the latter context looks like this:

```
a = (((-2147483647 - 1) ^ b) << (((-668224961 ^ 2147483647) + 1479258713) - 24)) |
     (((c + 2147483647) >> 8) << 8);
```

We found a bug in LLVM's bit-tracking dead code elimination pass (LLVM #33695) that we believe would have been very difficult to trigger without combining the shift-bitwise context with half of expression leaves being constants.

Since many peephole optimizations rely on special constants that are often small integers, YARPGen generates constants with small magnitude, and constants near values such as `INT_MAX`, more often than constants uniformly selected from the entire range. Transformations like this from LLVM's `instcombine` pass require constants of small magnitude, which would be very unlikely to occur if, for example, YARPGen selected constant values uniformly from $0..2^{64} - 1$.

```
(x + -1) - y → ~y + x
```

Other constants are generated to have contiguous blocks of zeroes and ones in their binary representations.

Finally, YARPGen maintains a buffer of previously-used constants, and then in all but logical context randomly reuses these constants (and also their negations and complements) in subsequent parts of expression trees. This allows us to trigger transformations such as these more frequently:[4]

```
(x | c2) + c → (x | c2) ^ c2 iff (c2 == -c)
((x | y) & c1) | (y & c2) → (x & c1) | y iff c1 == ~c2
((x + y) & c1) | (x & c2) → x + y iff c2 = ~c1 and
                              c2 is 0···01···1₂ and (y & c2) == 0
```

Common subexpression buffer. Common subexpression elimination, global value numbering, and related optimizations contain elaborate, bug-prone logic for factoring redundant computations out of code being compiled. To increase the chances of triggering these optimizations, YARPGen buffers previously-generated subexpressions and reuses them:

```
d = c + (128 * a >> (b % 13));
e = INT_MAX - (128 * a >> (b % 13));
```

Code like this would be unlikely to be generated by a sequence of independent random choices.

[4]https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Transforms/InstCombine/InstCombineAddSub.cpp#L906
https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Transforms/InstCombine/InstCombineAndOrXor.cpp#L2538
https://github.com/llvm/llvm-project/blob/release/10.x/llvm/lib/Analysis/InstructionSimplify.cpp#L2208-L2211

Parameter shuffling. Each kind of random choice made by YARPGen is controlled by a distribution. To avoid the "every test case is different, but they all look the same" phenomenon, YARPGen uses a parameter shuffling technique where, at startup time, random distributions are themselves randomly generated. So one test case might have 85% of its variables having `char` type, and the next might end up with only 3% of these. The insight behind shuffling is the same as the one behind swarm testing [Groce et al. 2012], but our implementation is finer-grained since it was designed into YARPGen instead of being retrofitted using command-line options. Parameters that are shuffled include those controlling:

- generation of compound types: base types for arrays, bit-field distribution in structs, whether to reuse existing compound types, etc.
- choices for arithmetic leaves: constant, variable, array element, structure member or pointer indirection
- where to put the results of assignments: local variable, mixed or output global variable, array element, struct member, pointer indirection
- array representation: C-style array, `std::valarray`, `std::vector`, `std::array`
- use of the subscript operator vs. `at(i)` to access array elements
- arithmetic operators: unary, binary, ternary
- kind of statements: variable definition, assignment, conditional
- what kind of special constants to generate, how often we reuse existing constants and what modifications to them will we apply
- how many common sub-expressions we create and how often we reuse them
- distribution of contexts

In total, there are 23 different parameters that are shuffled for each test. We believe this technique allows YARPGen to reach corner cases more effectively.

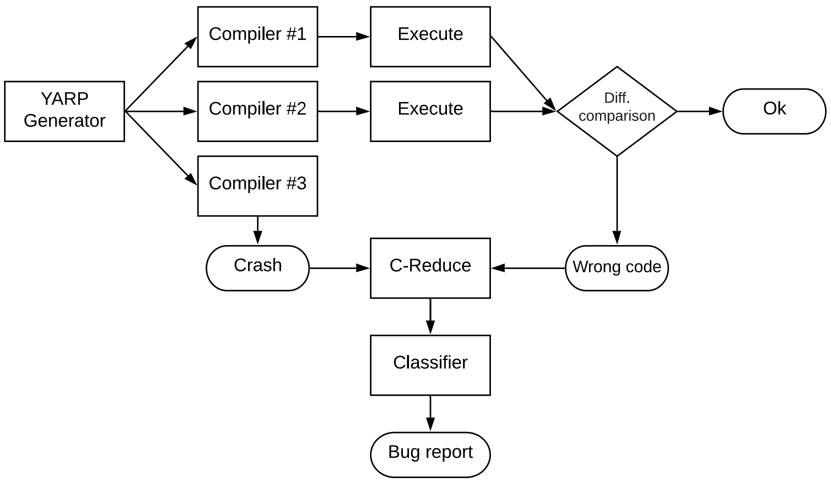

Fig. 1. Framework for differential random compiler testing

3.4 Automation

Figure 1 depicts the framework we developed for automating most of the important tasks associated with differential random compiler testing. This framework is not specific to YARPGen: we have also used it to run Csmith. Fewer than 10 lines of code needed to be changed to make this work.

On every core, the framework executes these steps in a loop:

(1) invokes YARPGen to create a test case,
(2) compiles the test using a configurable collection of compilers and compiler options, going to step (4) if any compiler crashes
(3) runs the resulting binaries, only continuing to the next step if behavioral differences in the output are detected,
(4) generates an "interestingness" test script for C-Reduce [Regehr et al. 2012], reduces the bug-triggering program, and
(5) runs a bug classifier that we developed, in an effort to cluster together all test cases that trigger the same bug.

This framework uses a configurable number of cores and requires essentially no user intervention in the case where no compiler bugs are found. When bugs are found, they are ready to be reported to compiler developers because they are reduced and come with a Makefile for reproducing them. The automation framework optionally runs generated code under UBSan, LLVM's undefined behavior sanitizer,[5] to check for bugs in the generator. It also takes care of cleaning up temporary files that are often left over when compilers crash, and supports optionally running failing tests multiple times, as a hedge against non-deterministic behavior (true non-determinism is rare in compilers, but we have seen incorrect bug reports because a test machine ran out of memory or disk space).

C-Reduce's transformations are syntactical in nature and tend to introduce undefined behavior during reduction. We rely on static and dynamic checking tools including compiler warnings, UBSan, ASan,[6] and tis-interpreter[7] to prevent undefined behaviors during test case reduction. (We do not rely on these tools to help YARPGen generate UB-free programs in the first place: there would be enormous overhead associated with invoking them incrementally on partially-constructed programs.)

Classifying crashes and wrong-code bugs. The bug classifier is crucial because we observed that the number of times bugs are triggered typically follows a power law: one bug might be triggered by 1% of test cases, another by 0.01% of test cases, and yet another by 0.0001% of test cases. A recent study [Böhme and Falk 2020] appears to support this observation. It is important to prevent frequently-occurring bugs that have not been fixed yet from creating a needle-in-the-haystack phenomenon, hiding bugs that are triggered only very rarely.

Bugs that cause the compiler to crash are relatively easy to classify because they typically emit a characteristic error message. When a good error message is not emitted, for example, because the compiler was terminated by the OS due to a protection violation, we could attempt to classify the bug by looking at the compiler's stack trace, but we have not yet done this.

Bugs that cause the compiler to emit incorrect code are more difficult to classify, and here we require some cooperation from the compiler.[8] The idea, first published by Whalley [1994], is to search over the sequence of transformations the compiler performs while optimizing a program. If leaving out a particular transformation makes the miscompilation disappear, then either that

[5]https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
[6]https://clang.llvm.org/docs/AddressSanitizer.html
[7]https://trust-in-soft.com/tis-interpreter
[8]https://llvm.org/docs/OptBisect.html

transformation is incorrect or else it ends up creating the conditions for a different, incorrect pass to introduce the miscompilation. Either way, this transformation is a useful fingerprint for the miscompilation, but it is not guaranteed to be a unique identifier for such errors.

3.5 Experimental Features

This section describes several features that we added to experimental versions of YARPGen. We include them in our description because we believe them to be interesting, and we used them to find compiler bugs. However, we wanted to describe them separately because they are, in general, less fleshed out and complete than are the features we describe elsewhere in this paper. None of these features were used in evaluation experiments, described in Section 4.

Floating point. Generating code that manipulates floating point (FP) values is not difficult. The primary challenge is not in generation, but rather in interpreting the results of differential testing: it is not uncommon for a compiler optimization to change the result of an FP computation, and yet compiler developers do not consider this to be a bug. Our partial solution to this problem is to limit FP values to a narrow range (0.5..1.0), to limit the number of operations in an arithmetic chain containing FP values, and to exclude floating point values from YARPGen's usual checksum mechanism. We instead dump the final values stored in FP variables to a file, and then perform differential testing using a threshold, rather than insisting on bit-for-bit equality. YARPGen supports the `float`, `double`, and `long double` types.

Vectors. The Clang C and C++ frontend for LLVM supports OpenCL vector extensions.[9] YARPGen has highly preliminary support for generation of short (2–16 element) vectors of the `int` and `unsigned` types. In order to be able to reuse YARPGen's UB avoidance mechanisms, we forced each vector to contain the same value in all elements, and disabled accesses to individual elements. We also disabled logical, comparison, and ternary operators for vector-typed values because the OpenCL semantics for these operations diverges from C and C++.

Loops. Extending YARPGen's machinery for avoiding undefined behaviors to support loops was an interesting challenge. Our solution uses a hybrid strategy. We test for the absence of undefined behavior in the first loop iteration and then ensure that the rest of the loop iterations operate on the same values as those seen in the first iteration. To make this happen, we disabled mixed variables (those that are both read and written), and we ensure that each array that is accessed inside a loop has the same value at all element positions. The loop iteration space is determined statically. Loop induction variables are treated specially: undefined behavior on them is avoided by construction, using the mechanism described in Section 3.1, and random accesses to induction variables from code in loop bodies are not allowed. Loop reduction variables (special-purpose variables used to, for example, sum up the values stored in an array) are treated in a similar fashion. Generated loops can contain regular conditional control flow, as well as the `break` and `continue` statements.

3.6 Limitations

YARPGen has a number of limitations including lack of support for: pointer arithmetic, function calls, enums, classes, preprocessor definitions, templates, lambdas, dynamic memory allocation, compound assignment operations, and side-effecting operators. Additionally, support for floating point values and loops is quite limited. Many of YARPGen's limitations—such as not generating code containing function calls—stem from its original goal of detecting bugs in scalar optimizations. Other limitations, such as not generating code containing strings or dynamic memory allocation, are implementation artifacts that we hope to lift in the future.

[9] https://clang.llvm.org/docs/LanguageExtensions.html#vectors-and-extended-vectors

3.7 Implementation and Deployment

YARPGen is about 8,600 lines of C++. Its testing framework (written in about 3,100 lines of Python) is responsible for running tests across many cores, reducing programs that trigger bugs, and classifying bugs. Our software is open source; the version described in this paper can be found at: https://github.com/intel/yarpgen/tree/v1 and the latest version can be found at: https://github.com/intel/yarpgen/

4 EVALUATION

4.1 Summary of a Testing Campaign

Over a two year period we used YARPGen to test the then-current versions of GCC, LLVM, and the Intel® C++ Compiler. In all cases for our main testing campaign, the compilers were generating code for various flavors of the x86-64 architecture. We also verified that YARPGen can be used to test compilers for ARM, but we didn't test it rigorously. Most of our testing was at the most commonly used optimization levels (e.g., `-O0`, `-O3`), but in some cases we also enabled non-standard compiler elements such as GCC's undefined behavior sanitizer[10] and LLVM's `NewGVN` pass.[11] The top-level results from this testing campaign are that we found:

- 83 previously-unknown bugs in GCC, 78 of which have been fixed so far. 32 were compiler crashes and 51 were wrong code bugs. 4 were in the front-end, 50 were in the middle-end, 11 were in the backend, and 18 of them are related to GCC's undefined behavior sanitizer. Some of these bugs are summarized in Table 2. The full list is available online[12].
- 62 previously-unknown bugs in LLVM, 49 of which have been fixed so far. 53 bugs were compiler crashes and 9 were wrong code bugs. One bug was in the front-end, 15 were in the middle-end, and 43 were in the backend (we could not categorize three bugs). Some of these bugs are summarized in Table 3. The full list is available online[13]. (This table contains one bug that the LLVM developers marked as DUPLICATE and four that were marked as WORKSFORME. Usually, we would not count such bugs as valid, but in these five cases we verified that these were real, previously unknown bugs first reported by us. In other words, the compiler developers labeled these bugs incorrectly.)
- 76 previously-unknown bugs in the Intel® C++ Compiler, 67 of which have been fixed so far. 30 bugs were compiler crashes and 46 bugs were wrong code. 35 were in scalar optimizations, 23 were in the backend, 17 were in the high-level optimization and vectorization framework, and one was in interprocedural optimizations.

Taken together, these results show that YARPGen is capable of finding defects in production-grade compilers that have already been heavily targeted by random test-case generators. Furthermore, most of these defects were deemed worth fixing by compiler developers.

Examples of reduced, bug-triggering programs. Listing 1 demonstrates the advantages of explicit common subexpression generation and special constants, while listing 2 shows the advantage of contexts.

Concurrently reported bugs. One way to gain confidence that a fuzzer is finding bugs that matter to real developers is to look for cases where a bug found by the fuzzer was independently re-reported by an application developer. About 10% of the LLVM bugs that we reported were independently

[10]https://developers.redhat.com/blog/2014/10/16/gcc-undefined-behavior-sanitizer-ubsan/
[11]https://lists.llvm.org/pipermail/llvm-dev/2016-November/107110.html
[12]https://github.com/intel/yarpgen/blob/v1/gcc-bug-summary.md
[13]https://github.com/intel/yarpgen/blob/v1/llvm-bug-summary.md

Table 2. YARPGen found 83 GCC bugs, this table shows 53 of them. An “ICE” is an internal compiler error, or a crash. We shortened some names of components and bug descriptions.

ID	Status	Type	Component	Description
83383	fixed	wrong code	tree-optimization	Wrong code with type conversions and ternary operators
83382	unconfirmed	wrong code	sanitizer	UBSAN tiggers false-positive warning
83252	fixed	wrong code	target	Wrong code with "-march=skylake-avx512 -O3"
83221	fixed	ICE	tree-optimization	qsort comparator not anti-commutative
82778	fixed	ICE	rtl-optimization	crash: insn does not satisfy its constraints
82576	NEW	ICE	rtl-optimization	sbitmap_vector_alloc() not ready for 64 bits
82413	fixed	ICE	c	-O0 crash (ICE in decompose, at tree.h:5179)
82381	fixed	ICE	tree-optimization	internal compiler error: qsort checking failed
82353	fixed	wrong code	sanitizer	runtime ubsan crash
82192	fixed	wrong code	rtl-optimization	gcc produces incorrect code with -O2 and bit-field
81987	fixed	ICE	tree-optimization	ICE in verify_ssa with -O3 -march=skylake-avx512
81814	fixed	wrong code	middle-end	Incorrect behaviour at -O0 (conditional operator)
81705	fixed	wrong code	middle-end	UBSAN: yet another false positive
81607	fixed	ICE	c++	Conditional operator: "type mismatch in shift expression"
81588	fixed	wrong code	tree-optimization	Wrong code at -O2
81556	fixed	wrong code	tree-optimization	Wrong code at -O2
81555	fixed	wrong code	tree-optimization	Wrong code at -O1
81553	fixed	ICE	rtl-optimization	ICE in immed_wide_int_const, at emit-rtl.c:607
81546	fixed	ICE	tree-optimization	ICE at -O3 during GIMPLE pass dom
81503	fixed	wrong code	tree-optimization	Wrong code at -O2
81488	fixed	wrong code	sanitizer	gcc goes off the limits allocating memory
81423	fixed	wrong code	rtl-optimization	Wrong code at -O2
81403	fixed	wrong code	tree-optimization	wrong code at -O3
81387	unconfirmed	wrong code	sanitizer	UBSAN consumes too much memory at -O2
81281	fixed	wrong code	sanitizer	UBSAN: false positive, dropped promotion to long type.
81162	fixed	wrong code	tree-optimization	UBSAN switch triggers incorrect optimization in SLSR
81148	fixed	wrong code	sanitizer	UBSAN: two more false positives
81097	fixed	wrong code	sanitizer	UBSAN: false positive for not existing negation operator
81088	fixed	wrong code	middle-end	UBSAN: false positive as a result of reassosiation
81065	fixed	wrong code	sanitizer	UBSAN: false positive as a result of distribution
80932	fixed	wrong code	sanitizer	UBSAN: false positive as a result of distribution
80875	fixed	ICE	sanitizer	UBSAN: compile time crash in fold_binary_loc
80800	fixed	wrong code	sanitizer	UBSAN: yet another false positive
80620	fixed	wrong code	tree-optimization	gcc produces wrong code with -O3
80597	fixed	ICE	ipa	internal compiler error: in compute_inline_parameters
80536	fixed	ICE	sanitizer	UBSAN: compile time segfault
80403	fixed	ICE	sanitizer	UBSAN: compile time crash
80386	fixed	wrong code	sanitizer	UBSAN: false positive - constant folding and reassosiation
80362	fixed	wrong code	middle-end	gcc miscompiles arithmetic with signed char
80350	fixed	wrong code	sanitizer	UBSAN changes code semantics
80349	fixed	ICE	sanitizer	UBSAN: compile time
80348	fixed	ICE	sanitizer	UBSAN: compile time crash in ubsan_instrument_division
80341	fixed	wrong code	middle-end	gcc miscompiles division of signed char
80297	fixed	ICE	c++	Compiler time crash: type mismatch in binary expression
80072	fixed	ICE	tree-optimization	ICE in gimple_build_assign_1
80067	fixed	ICE	sanitizer	ICE in fold_comparison with -fsanitize=undefined
80054	fixed	ICE	tree-optimization	ICE in verify_ssa
79399	fixed	ICE	middle-end	GCC fails to compile big source at -O0
78726	fixed	wrong code	middle-end	Incorrect unsigned arithmetic optimization
78720	fixed	wrong code	tree-optimization	Illegal instruction in generated code

Table 3. YARPGen found 62 LLVM bugs, this table shows 53 of them. An "ICE" is an internal compiler error, or a crash. We shortened some names of components and bug descriptions.

ID	Status	Type	Component	Description
35583	new	ICE	new bugs	NewGVN indeterministic crash
35272	fixed	ICE	new bugs	Assertion "Invalid sext node, dst <src!"
34959	new	wrong code	new bugs	Incorrect predication in SKX
34947	fixed	ICE	Backend: X86	-O0 crash
34934	new	ICE	Backend: X86	UNREACHABLE: "Unexpected request for libcall!"
34856	fixed	ICE	new bugs	Assertion: "Invalid constantexpr cast!"
34855	fixed	ICE	Backend: X86	Cannot select shl and srl for v2i64
34841	fixed	ICE	Scalar Opt	InstCombine - Assertion
34838	fixed	ICE	new bugs	InstCombine - Assertion
34837	fixed	ICE	new bugs	UNREACHABLE in DAGTypeLegalizer
34830	new	ICE	new bugs	"Cannot emit physreg copy instruction"
34828	fixed	ICE	Register Allocator	Assertion: "index out of bounds!"
34787	fixed	ICE	new bugs	Assertion: "Not a tree leaf"
34782	fixed	ICE	new bugs	Assertion: "Deleted edge still exists in the CFG!"
34381	fixed	wrong code	new bugs	Clang produces incorrect code at -O0
34377	new	wrong code	Backend: X86	Clang produces incorrect code with -O1 and higher
34137	fixed	ICE	Backend: X86	clang crash in llvm::SelectionDAG::Combine on -O0
33844	fixed	ICE	Backend: X86	Assertion: "loBit out of range"' failed with -O1
33828	fixed	ICE	Common CodeGen	Crash in X86DAGToDAGISel::RunSDNodeXForm
33765	fixed	ICE	new bugs	Inst. repl. in instcombine violates dominance relation
33695	fixed	wrong code	new bugs	Bit-Tracking Dead Code Elimination (BDCE) fails
33560	fixed	ICE	Backend: X86	"Cannot emit physreg copy instruction"
33442	new	wrong code	Frontend	UBSAN: right shift by negative value missed
32894	fixed	ICE	Backend: X86	ICE: "Cannot replace uses of with self"
32830	fixed	wrong code	new bugs	Clang produces incorrect code with -O2 and higher
32525	duplicate	ICE	Backend: X86	Assertion: "Number of nodes mismatch"
32515	fixed	ICE	Common CodeGen	Assertion" "DELETED_NODE in CSEMap!"
32345	fixed	ICE	Common CodeGen	Assertion: "Mask size mismatches value type size!"
32340	fixed	ICE	Common CodeGen	Assertion: "Deleted Node added to Worklist"
32329	fixed	ICE	new bugs	Silent failure in X86 DAG->DAG Inst Selection
32316	fixed	ICE	new bugs	Assertion: "Binary operator types must match!"
32284	fixed	ICE	new bugs	Assertion: "Invalid child # of SDNode!"
32256	fixed	ICE	new bugs	Assertion: "Cannot move between mask and h-reg"
32241	fixed	wrong code	new bugs	Incorrect result with -march=skx -O0 -m32
31306	fixed	ICE	Backend: X86	Compiler crash: Cannot select
31045	fixed	ICE	Backend: X86	Clang fails in insertDAGNode
31044	new	ICE	Backend: X86	Assertion: "Expect to extend 32-bit registers for LEA"
30813	fixed	ICE	New Bugs	Assertion: "MaskBits <= Size"
30783	fixed	ICE	New Bugs	Assertion: "replaceAllUses of value with new value"
30777	worksforme	ICE	New Bugs	Assertion: "Illegal cast to vector (wrong type or size)"
30775	fixed	ICE	Backend: X86	Assertion: "NodeToMatch was removed"
30693	fixed	ICE	Backend: X86	Segfault: alignment incorrect for relative addressing
30286	fixed	wrong code	new bugs	KNL bug at -O0
30256	fixed	ICE	new bugs	Assert in llvm::ReassociatePass::OptimizeAdd
29058	fixed	ICE	new bugs	Assert in llvm::SelectionDAG::Legalize()
28845	new	ICE	new bugs	Incorrect codegen for "store <2 x i48>"
28661	fixed	wrong code	new bugs	incorrect code for boolean expression at -O0
28313	fixed	ICE	New Bugs	Assertion: "isSCEVable(V->getType())"
28312	fixed	ICE	New Bugs	Assertion: "Res.getValueType() == N->getValueType(0)"
28301	fixed	ICE	New Bugs	Assertion: "Register is not used by this instruction!"

```c
#include <stdio.h>

unsigned short a = 41461;
unsigned short b = 3419;
int c = 0;

void foo() {
  if (a + b * ~(0 != 5))
    c = -~(b * ~(0 != 5)) + 2147483647;
}

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

Listing 1. GCC bug #81503, found on revision #250367. The correct result is 2147476810; the incorrectly optimized version returns -2147483648.

```c
func.c:
extern const long int a;
extern const long int b;
extern const long long int c;
extern long int d;

void foo() {
    d = ((c | b) ^ ~c) & (a | ~c);
}

/////////////////////////////////////////////////////

main.c:
#include <stdio.h>

extern void foo ();

const long int a = 6664195969892517546L;
const long int b = 1679594420099092216L;
const long long int c = 1515497576132836640LL;
long int d = -7475619439701949757L;

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

Listing 2. LLVM bug #32830, found on revision #301477. The correct result is -236191838907956185; the incorrectly optimized version returns -5292815780869864331.

discovered by projects such as SQLite (LLVM #35583), Chromium (LLVM #34830 and #33560), LibreOffice (LLVM #32830), Speex (LLVM #32515), and GROMACS (LLVM #30693). For GCC, only one bug was independently discovered, in Python 3.5.1 (GCC #71279).

This situation usually happened in two different cases. In the first case, we and application developers who test their product against the latest compiler version reported the same bug around the same time (usually within a couple of days from each other). In the second case, a bug that we had reported remained unfixed for some time (sometimes more than a year), and then, later on, application developers rediscovered it in the released compiler.

Finding a compiler bug using a fuzzer, rather than waiting for application developers to find it, is preferable for two important reasons. First, whereas our YARPGen-based workflow is directed entirely towards finding compiler bugs and has significant machinery aimed at making this task easier, finding a compiler bug that causes a large application to be miscompiled is extremely non-trivial, even for experienced developers. Second, test-case reduction for programs generated by YARPGen is automated and works very well. In contrast, extracting a minimal reproducer for a compiler bug from, for example, the Chromium sources is laborious. In practice, the reduced test cases that we report are much smaller than those reported by application developers.

Partial bug fixes. YARPGen is good at finding *partial bug fixes.* These occur when a compiler developer, for one reason or another, eliminates the symptoms of a compiler bug without fully addressing the root cause. This leads to the situation where the test case in the bug report no longer triggers the bug, but other test cases can still trigger it. Fuzzers seem to be particularly good at finding different ways to trigger errors, providing thorough testing of bug fixes. During our bug-finding campaign we discovered five LLVM bugs (#34381, #33695, #32284, #32256, #30775) and six GCC bugs (#83252, #81588, #80932, #80348, #71488, #70333) that were partially fixed.

Sanitizer bugs. YARPGen's testing system optionally uses the undefined behavior sanitizer supported by GCC and LLVM to ensure that generated tests are free from undefined behavior. Because YARPGen was designed to avoid all undefined behaviors, every alarm signaled by a sanitizer either indicates an error in YARPGen or in the sanitizer. We found two bugs in LLVM and 22 bugs in GCC that were in the undefined behavior sanitizer, or else were exposed due to using the sanitizer.

4.2 Evaluating the Direct Impact of Generation Policies

A major hypothesis behind YARPGen was that generation policies would make it more effective at finding compiler bugs. We conducted an experiment with the goal of confirming or rejecting this hypothesis. Our starting point was the set of LLVM crash bugs found by YARPGen which could be reliably identified by a specific error message printed by LLVM as it terminated. 22 bugs met this criterion; the rest of them had non-specific signatures, such as segmentation faults. We did not look at miscompilation bugs in this experiment because it is difficult to reliably determine that a particular miscompilation bug has been triggered (our bug classifier is useful, but not foolproof). All of the experimental features described in Section 3.5 were disabled for this experiment.

Out of the 22 bugs, we could not trigger 13 of them within 72 hours when using all threads on a 16-core (32 thread) Intel® Xeon® machine, using the latest version of YARPGen. We dropped these from the experiment—we deemed them impractical for our experiment due to their low probability of being triggered. For the remaining nine bugs, we ran YARPGen for 72 hours (on the same kind of Intel® Xeon® machine) in its standard configuration—where it uses generation policies—and also for 72 hours in a configuration where it does not use generation policies.

The results of this experiment are summarized in Table 4. Four of the bugs were not triggered at all without generation policies. Out of the remaining five bugs, two were triggered more often without generation policies and three were triggered more often with generation policies. We also

Table 4. Number of times nine selected LLVM crash bugs were found in 72 hours on a 32-thread machine, with and without generation policies (GP)

Bug ID	GP	No GP	GP found the bug more times?	GP is better at 95%?
27638	57029	67976	no	no
27873	23	2	yes	yes
29058	9	0	yes	yes
30256	21	0	yes	yes
30775	1	0	yes	no
32284	153	21	yes	yes
32316	5843	27922	no	no
32525	1	0	yes	no
33560	1853	1720	yes	yes

used a zero-mean test to see if the data support rejecting the null hypothesis "generation policies do not improve YARPGen's ability to trigger each bug" with 95% confidence. In summary, in four cases generation policies allowed a bug to be found that could not be found otherwise, whereas the reverse situation (a bug could be found without generation policies, but not with them) never occurred.

4.3 Evaluating an Indirect Impact of Generation Policies

Getting good statistical evidence about real compiler bugs is difficult, so we also investigated a less direct metric: the impact of generation policies on the number of times different optimizations were triggered, as measured using optimization counters supported by LLVM and GCC. An optimization counter simply records the number of times a particular optimization, such as common subexpression elimination, is used during a compilation. The basis for this experiment is that we believe (but have not experimentally confirmed) that triggering an optimization more times is better, because this will be more likely to uncover incorrect corner cases in the optimization. Optimization counters can be viewed as a kind of domain-specific code coverage metric: they are incremented at locations in the compiler code where experts believe the resulting information will be interesting or useful. For example, they are incremented when analysis can draw some conclusion or when optimization transformation is successfully applied. All of the experimental features described in Section 3.5 were disabled for this experiment.

We conducted an experiment where we ran YARPGen, with and without generation policies, for 30 repetitions of 48 minutes each (for 24 hours of total testing time) on a 32 core (64 thread) AMD Ryzen™ Threadripper machine. We performed this experiment separately for GCC and for LLVM, monitoring a collection of optimization counters that we had identified ahead of time as being the kind that we intend YARPGen to trigger often. (Other optimization counters, such as those that measure loop or floating point optimizations, would not have been relevant to this experiment.) We then used a zero-mean test to put each counter into one of three categories. First, those where we can be 95% confident that generation policies trigger the counter more times (i.e., generation policies improve testing capability). Second, those where we can be 95% confident that generation policies trigger the counter fewer times (i.e., generation policies worsen testing capability). Finally, a default category where neither of these conditions is true, corresponding to a null hypothesis where generation policies do not decisively make testing better or worse. The results are summarized in Table 5.

Table 5. Effect of generation policies (GP) on optimization counters

	Number of LLVM counters	Number of GCC counters
GP is better	94	67
GP is worse	108	27
Comparable	20	11
Total	222	105

For testing GCC, generation policies have a clear advantage (the benefit of using them outweighs disadvantages), whereas the LLVM results are mixed. We took a closer look at the counters in `instcombine` (LLVM's large collection of peephole optimizations). Many of these showed an increase, and those that decreased did not do so catastrophically. These results are shown in Table 6.

Table 6. Effect of generation policies on values of several LLVM optimization counters. "GP is better" means that a 95% confidence test was passed.

Counter	GP is better	Ratio of GP to no GP
instcombine.NumCombined	yes	1.04
instcombine.NumConstProp	no	0.78
instcombine.NumDeadInst	no	0.97
instcombine.NumDeadStore	yes	1.2
instcombine.NumExpand	yes	7.4
instcombine.NumFactor	yes	1.8
instcombine.NumReassoc	yes	1.8
instcombine.NumSel	yes	155.0
instcombine.NumSunkInst	no	0.95
instsimplify.NumExpand	yes	3.7
instsimplify.NumReassoc	yes	2.4
instsimplify.NumSimplified	no	0.76

Finally, for every optimization counter, we computed the ratio between the counter values with and without generation policies. The geometric mean of the resulting ratios is 1.2 for LLVM and 1.4 for GCC. This shows that overall, generation policies increase the number of optimizations triggered. The important thing, however, isn't a uniform increase but rather diversity across a large number of test cases. Therefore, even though generation policies decrease the number of times some optimizations are triggered, our goals can be met by simply turning off generation policies for some fraction of YARPGen's runs, ensuring that optimizations that end up being suppressed by generation policies get triggered enough times.

4.4 Performance of YARPGen

Figure 7 shows where CPU time is spent during a differential testing run. For this experiment we spent 24 hours testing GCC and LLVM, each at their `-O0` and `-O3` optimization levels, on a machine with two 28-core (56-thread) Intel® Xeon® Platinum 8180 processors. We used YARPGen in its standard, out-of-the-box configuration, with the experimental features described in Section 3.5 disabled. We measured CPU consumption from our Python test harness and consequently we did not measure how much CPU time the harness itself used. However, since it performs almost no real computation, we do not believe that it contributed significantly. Overall, YARPGen accounted

Table 7. Distribution of CPU time used during 24 hours of differential testing of GCC and LLVM, each at two optimization levels, on a large multicore system

tool	step	% of total CPU time
YARPGen	generation	4.98%
gcc -O0	compilation	19.54%
	execution	0.03%
gcc -O3	compilation	28.96%
	execution	0.03%
clang -O0	compilation	14.95%
	execution	0.03%
clang -O3	compilation	31.43%
	execution	0.03%

for less than 5% of the total CPU time; almost all of the rest of the CPU time was consumed by the compilers, which are particularly CPU-hungry when they are asked to optimize heavily.

4.5 YARPGen and Code Coverage

The last experiment we performed was to investigate what effect compiling programs generated by YARPGen had on code coverage of the open-source compilers. For both GCC and LLVM, we collected code coverage information from all combinations of:

(1) the suite of unit tests that come with the compiler,
(2) the C and C++ programs from SPEC CPU 2017, compiled at `-O3`, and
(3) random testing using YARPGen in its default configuration, invoking the compiler at `-O3`.

For both compilers, we got coverage information using a configure-time option provided by the compilers' respective build systems. All of the experimental YARPGen features described in Section 3.5 were disabled for this experiment.

Tables 8 and 9 summarize the results. A "region" is a unit of coverage that is similar to a basic block: multiple lines of C or C++ that do not contain control flow can belong to the same region. However, it is also possible for a single line of code containing internal control flow, such as `return x || y && z`, to contain multiple regions.[14]

Table 8. Coverage of LLVM source code

	Functions	Lines	Regions
YARPGen	28.53%	33.71%	24.78%
SPEC CPU 2017	36.14%	41.90%	33.68%
SPEC + YARPGen	36.57%	42.48%	34.34%
% change	+0.43%	+0.58%	+0.66%
unit test suite	84.33%	87.61%	78.87%
unit tests + YARPGen	84.34%	87.72%	79.04%
% change	+0.01%	+0.11%	+0.17%
unit tests + SPEC	84.36%	87.79%	79.15%
unit tests + SPEC + YARPGen	84.37%	87.83%	79.21%
% change	+0.01%	+0.04%	+0.06%

[14] https://clang.llvm.org/docs/SourceBasedCodeCoverage.html#interpreting-reports

Table 9. Coverage of GCC source code

	Functions	Lines	Branches
YARPGen	40.70%	34.41%	26.20%
SPEC CPU 2017	52.15%	45.18%	34.23%
SPEC + YARPGen	52.82%	47.39%	37.06%
% change	+0.67%	+2.21%	+2.83%
unit test suite	80.22%	78.28%	62.82%
unit tests + YARPGen	80.38%	79.31%	64.41%
% change	+0.16%	+1.03%	+1.59%
unit tests + SPEC	80.31%	78.59%	63.28%
unit tests + SPEC + YARPGen	80.44%	79.46%	64.65%
% change	+0.13%	+0.87%	+1.37%

Regardless of the coverage metric, it is clear that random testing using YARPGen does not improve coverage very much. (Our finding is compatible with numbers reported in Table 3 of the Csmith paper [Yang et al. 2011]). Similarly, the compilers are covered somewhat poorly by compiling SPEC CPU 2017. We believe that a reasonable takeaway from these results is that none of function, line, or region coverage is very good at capturing the kind of stress testing that is generated by a random program generator. It seems possible that a path-based coverage metric would capture this better.

5 RELATED WORK

Random testing has being used for almost 60 years [Sauder 1962]. A recent survey of compiler testing by Chen et al. [2020] contains a good summary of this field, and some interesting context is given by Marcozzi et al. [2019], who studied the character of miscompilation bugs found by fuzzers vs. those reported by compiler users.

In this section, we focus on the previous tools that are most closely related to YARPGen: randomized program generators. Another random testing technology, based on mutating existing test cases, less relevant.

5.1 Generative Fuzzers for C and C++ Compilers

McKeeman [1998] coined the term “differential testing” and created a family of generators called DDT that was extremely expressive—they could generate all valid C programs—but at the expense of including undefined behaviors. His work had a limited mechanism for avoiding some kinds of undefined behavior during test-case reduction. However, overall, it would not be suitable for finding miscompilation errors in modern compilers: the undefined operations would raise too many false positives.

Quest [Lindig 2005] was specifically designed to test calling conventions, and generated a great variety of interesting function call signatures. It avoided undefined behavior by simply not generating potentially dangerous constructs such as arithmetic expressions. Quest did not use differential testing, but rather generated self-checking programs using assertions. YARPGen and Quest are almost completely disjoint in terms of bug-finding power.

Eide and Regehr’s improved version of the existing Randprog tool used differential testing to look for miscompilation of volatile-qualified variables [Eide and Regehr 2008]. Their testing method relied on the invariant that the number of loads from or stores to a memory location corresponding to a volatile-qualified variable in C should not change across compilers or optimization levels.

The Csmith [Yang et al. 2011] program generator avoided some undefined behaviors using static analysis, but used wrapper functions to prevent unsafe arithmetic operations. The goal of entirely avoiding wrapper functions sets YARPGen apart from Csmith. However, Csmith was more expressive than YARPGen in certain respects; it supported reasonably expressive pointer arithmetic, for example.

The Orange [Nagai et al. 2012, 2013, 2014] family of random code generators was initially focused on finding compiler bugs, but their subsequent work shifted its focus towards missed optimization opportunities and other fuzzing techniques [Hashimoto and Ishiura 2016; Nakamura and Ishiura 2016]. The first Orange test case generator simply discarded expressions that contained undefined behavior [Nagai et al. 2012]; the second version added constants to shift expressions away from being undefined [Nagai et al. 2013]; and the third one added *operation flipping* [Nagai et al. 2014]: a scheme for replacing operations that lead to undefined behavior with a related operation that avoids the problem. YARPGen builds directly upon this previous work, but support many additional language features (control flow, structures, arrays, pointers, etc.). Orange tracked all of the values in the test during the generation, so at the end of the test it is able to create direct comparisons with expected value. This would make automated test case reduction difficult; YARPGen avoids this problem using differential testing.

Ldrgen [Barany 2018a,b] was designed to search for missed optimization opportunities. A motivating observation for this work was that test cases generated by Csmith often contain a lot of dead code that is eliminated by compiler, leading to a relatively small amount of machine code being emitted at the end. Ldrgen creates tests that are guaranteed to contain only live code using the Frama-C [Cuoq et al. 2012] static analyzer as a starting point. This generator does not avoid undefined behavior (except for limited attempts to avoid division by zero and oversized shifts). However, the generator does have good expressiveness and it generates code that is guaranteed to not be eliminated by the compiler.

A relatively recent trend is machine-learning-based test-case generation for compilers. DeepFuzz [Liu et al. 2019] targets C compilers; 82% of its outputs can be compiled and optimized. This effort was aimed at finding compiler crash bugs, and did not attempt to avoid undefined behaviors. A second effort, DeepSmith [Cummins et al. 2018], mainly targets OpenCL compilers. It generates tests that contain undefined and non-deterministic behavior, and then relies on third-party tools to filter out undesirable test cases that would have raised false positives.

The promise of machine-learning-based test-case generation is that by inferring the structure of the language to be generated, elaborate engineering efforts, of the kind that we did for YARPGen, can be avoided. However, it remains to be seen if the power of machine learning can be extended to deep semantic properties such as freedom from undefined behavior.

5.2 Generative Fuzzers for Other Languages

TreeFuzz [Patra and Pradel 2016] is based on probabilistic, generative models. It uses a set of examples to capture properties of the input language. The authors used this fuzzer to test JavaScript and HTML. Glade [Bastani et al. 2017] infers a target grammar from a set of examples and black-box access to a grammar parser. It was used to test Python, Ruby, and JavaScript. Dewey et al. [2015] used Constraint Logic Programming to test the Rust typechecker. To increase expressiveness, they used an idea that is related to swarm testing [Groce et al. 2012]: instead of creating one generator that uses all of the language features, they created different generators for selected subsets of the language.

5.3 Towards Diversity in Randomly Generated Test Cases

An early compiler fuzzer by Burgess and Saidi [1996] was designed with the goal of maximizing the number of optimization opportunities. Their Fortran generator explicitly generated common sub-expressions, linear induction variables, which are subject to strength reduction, and predefined patterns of arithmetic expressions. Other optimizations, such as copy propagation, constant folding, algebraic optimizations, code motion, and dead code elimination, were assumed to happen due to random nature of the generation process. This paper describes an idea that our generation policies build upon; they manually tuned probabilities in their generator in such a way that certain optimizations, such as constant folding and algebraic transformations, were triggered more frequently. YARPGen significantly extends this idea by supporting many more policies and by applying them automatically during the generation process (Section 3.3). The only shared feature between this previous work and YARPGen is the explicit generation of common subexpressions.

Swarm testing is an inexpensive way to improve test case diversity [Groce et al. 2012]. The idea is to randomly flip options in a generator that already exist for the purpose of disabling language features in generated test cases. In its original implementation, Swarm was designed as a stand-alone project that used Csmith command-line interface to alter probabilities of random decisions of each generated test. It could only alter global properties that were exposed by the developers externally. YARPGen has a similar built-in mechanism: parameter shuffling (Section 3.3). The motivation is the same: we want to increase diversity in generated tests by limiting or altering random choice decisions. Our implementation is significantly finer grained since we built generation policies into YARPGen from the start.

Adaptive Random Testing [Chen et al. 2010; Huang et al. 2019] is a technique for explicitly generating diverse test cases. The idea is that, to maximize test diversity, each new test case should be as dissimilar as possible to those that were already generated. This requires an explicit distance metric to be defined so that each new test can be chosen via distance maximization [Biagiola et al. 2019]. It was not clear to us whether there exists a sensible distance metric for the high-dimensional space occupied by C and C++ programs, and we did not pursue this direction in YARPGen.

5.4 Mutation-Based Compiler Fuzzing

Metamorphic testing is based on the idea that certain transformations, applied to a test case, should not change the behavior of that test case. *Equivalence modulo inputs* (EMI) is a family of techniques that exploited this idea to find (as far as we know) more compiler bugs than any other single technique. Orion [Le et al. 2014] pruned dead code from test cases and Athena [Le et al. 2015] had the capability of adding dead code. Hermes [Sun et al. 2016] extended this technique by adding the ability to modify live code in a way that did not change the result of the test case.

Mutation-based techniques have advantages and disadvantages with respect to the generation-based approach. The primary advantage of mutation is that since it builds on existing test cases, the expressiveness of the generated code is limited only by the expressiveness of the test cases that are used as the basis for mutation. The main disadvantage is that the generator is not stand-alone (for example, various EMI efforts used programs generated by Csmith as a starting point) and cannot easily guarantee the absence of undefined behavior in test cases. Our view is that the mutation-based and generation-based approaches to fuzzing are sufficiently different that they cannot be compared directly. Moreover, neither approach is likely to subsume the other: in practice, we need to employ both kinds of fuzzing.

6 CONCLUSION

YARPGen is effective at finding bugs in modern C and C++ compilers, even when they have been heavily tested by other random test-case generators. Over a two-year period, we found and reported 83 previously unreported bugs in GCC, 62 in LLVM, and 76 in the Intel® C++ Compiler. Many of these bugs have been fixed. Some of the bugs we found were independently rediscovered by the developers of large software projects, confirming that at least some of the bugs triggered by YARPGen also affect application code. Moreover, in cases where we reported a bug that was also reported by application developers, the reduced test cases in our bug reports were considerably smaller.

Our first main research contribution is generation policies: a mechanism for improving diversity in random programs. Generation policies also increase the number of times targeted optimizations are performed by LLVM by 20%, and by 40% for GCC, during a given amount of time devoted to random testing, increasing the likelihood that optimization bugs will be triggered. Another contribution is a mechanism for avoiding undefined behavior for relatively complex code including control flow, pointers, and arrays. Finally, YARPGen comes with a testing framework that automates essentially all tasks related to random compiler testing except for actually reporting the bugs.

ACKNOWLEDGMENTS

This material is based on work supported by a grant from the Intel® Corporation. The authors would like to thank GCC, LLVM, and Intel ® C++ Compiler development teams for their help and cooperation.

REFERENCES

Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. 2015. Exploiting the Saturation Effect in Automatic Random Testing of Android Applications. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems* (Florence, Italy) *(MOBILESoft '15)*. IEEE Press, 33—43.

Gergö Barany. 2018a. Finding Missed Compiler Optimizations by Differential Testing. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 82—92. https://doi.org/10.1145/3178372.3179521

Gergö Barany. 2018b. Liveness-Driven Random Program Generation. In *Logic-Based Program Synthesis and Transformation*, Fabio Fioravanti and John P. Gallagher (Eds.). Springer International Publishing, Cham, 112–127.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 95—110. https://doi.org/10.1145/3062341.3062349

Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-Based Web Test Generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 142—153. https://doi.org/10.1145/3338906.3338970

Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software*. 11.

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

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (Feb. 2020), 36 pages. https://doi.org/10.1145/3363562

Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive Random Testing: The ART of Test Case Diversity. *J. Syst. Softw.* 83, 1 (Jan. 2010), 60—66. https://doi.org/10.1016/j.jss.2009.02.022

Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 95—105. https://doi.org/10.1145/3213846.3213848

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods* (Thessaloniki, Greece) *(SEFM'12)*. Springer-Verlag, Berlin, Heidelberg, 233—247. https://doi.org/10.1007/978-3-642-33826-7_16

K. Dewey, J. Roesch, and B. Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 482—493.

Eric Eide and John Regehr. 2008. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM international conference on Embedded software*. 255–264.

Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 78—88. https://doi.org/10.1145/2338965.2336763

Atsushi Hashimoto and Nagisa Ishiura. 2016. Detecting arithmetic optimization opportunities for C compilers by randomly generated equivalent programs. *IPSJ Transactions on System LSI Design Methodology* 9 (2016), 21–29.

R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia. 2019. A Survey on Adaptive Random Testing. *IEEE Transactions on Software Engineering* (2019).

International Organization for Standardization 2011. *ISO/IEC 9899:201x: Programming Languages—C.* International Organization for Standardization. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

International Organization for Standardization 2012. *ISO/IEC N3337: Working Draft, Standard for Programming Language C++*. International Organization for Standardization. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216—226. https://doi.org/10.1145/2594291.2594334

Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386—399. https://doi.org/10.1145/2814270.2814319

Christian Lindig. 2005. Random Testing of C Calling Conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging* (Monterey, California, USA) *(AADEBUG'05)*. Association for Computing Machinery, New York, NY, USA, 3—12. https://doi.org/10.1145/1085130.1085132

Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1044–1051.

Michaël Marcozzi, Qiyi Tang, Alastair Donaldson, and Cristian Cadar. 2019. A Systematic Impact Study for Fuzzer-Found Compiler Bugs. *arXiv preprint arXiv:1902.09334* (2019).

William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (Dec. 1998), 100–107.

Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 48–53.

Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2013. Scaling up size and number of expressions in random testing of arithmetic optimization of C compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*. 88–93.

Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. *IPSJ Transactions on System LSI Design Methodology* 7 (2014), 91–100.

K. Nakamura and N. Ishiura. 2016. Random testing of C compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 676—679.

Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2016. RandIR: Differential Testing for Embedded Compilers. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala* (Amsterdam, Netherlands) *(SCALA 2016)*. Association for Computing Machinery, New York, NY, USA, 21—30. https://doi.org/10.1145/2998392.2998397

Jibesh Patra and Michael Pradel. 2016. Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data. *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664* (2016).

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335—346. https://doi.org/10.1145/2254064.2254104

Richard L. Sauder. 1962. A General Test Data Generator for COBOL. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference* (San Francisco, California) *(AIEE-IRE '62 (Spring))*. Association for Computing Machinery, New York, NY, USA, 317—323. https://doi.org/10.1145/1460833.1460869

Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 849—863. https://doi.org/10.1145/2983990.2984038

David B. Whalley. 1994. Automatic Isolation of Compiler Errors. *ACM Trans. Program. Lang. Syst.* 16, 5 (Sept. 1994), 1648—1659. https://doi.org/10.1145/186025.186103

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283—294. https://doi.org/10.1145/1993498.1993532