# No-Fuss Compiler Fuzzing


## Abstract

Developing a bug-free compiler is difficult; modern optimizing compilers are among the most complex software systems humans build. Fuzzing is one way to identify subtle compiler bugs that are hard to find with human-constructed tests. Grammar-based fuzzing, however, requires a grammar for a compiler's input language, and can miss bugs induced by code that does not actually satisfy the grammar the compiler *should* acccept. Grammar-based fuzzing also seldom uses advanced modern fuzzing techniques based on coverage feedback. However, modern mutation-based fuzzers are often ineffective for testing compilers because most inputs they generate do not even come close to getting past the parsing stage of compilation. This paper introduces a technique for taking a modern mutation-based fuzzer (AFL in our case, but the method is general) and adding mutation rules, based on operators used in mutation testing, to make such fuzzing more effective. We show that adding such mutations significantly improves fuzzing effectiveness. Our approach has allowed us to report more than 100 confirmed and fixed bugs in production compilers, and found a bug in the Solidity compiler that earned a security bounty. The most important feature of our approach is that for compilers written in C, C++, Go, Rust, or another language supported by AFL, the process of fuzzing is extremely low-effort for compiler developers. They essentially build their system with fuzzer instrumentation, point the fuzzer to a set of example programs that compile without error, and examine any crashes detected.



***CCS Concepts:*** • **Software and its engineering → Dynamic analysis**; **Software testing and debugging**.

***Keywords:*** fuzzing, compiler development, mutation testing

**ACM Reference Format:**
. 2022. No-Fuss Compiler Fuzzing. In ,. ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3486607.3486772





*foo, baz*





ACM ISBN 978-1-4503-9110-8/21/10…$15.00
https://doi.org/10.1145/3486607.3486772


## 1 Introduction

## 2 Mutation-Testing-Based Compiler Fuzzing

### 2.1 Mutation-Based Fuzzing

One use of the term "mutation" appears in the context of *mutation-based* fuzzing [5]. One can consider two basic kinds of compiler testing based on the generation of random input programs. One, in recent years paradigmatically expressed in the Csmith tool [8], works by using a grammar and deep knowledge of the language accepted by the compiler, to generate progams to test the compiler. This is sometimes called *generative* fuzzing. A second approach is to use an off-the-shelf fuzzer, such is used to find security vulnerabilities (e.g., the ubiquitous American Fuzzy Lop (afl) https://github.com/google/AFL), and a *corpus* of example programs, such as the set of regression tests for the compiler or a set of real-world programs. A fuzzer such as AFL operates by executing the program under test (here, the compiler) on inputs (initially those in the corpus), using instrumentation to determine code coverage in the compiler for each executed input. The fuzzer then takes inputs that look interesting and adds them to a *queue*. The basic loop is then to take some input from the queue, *mutate* it by making some essentially random change (e.g., flipping a single bit, or removing a random chunk of bytes), execute the new, mutated input under instrumentation, and add the new input to the queue if it seems "interesting" — typically, if it hits some kind of coverage target that has not previously been hit. The details of selecting inputs from the queue and determining how to mutate an input vary widely, and improving the effectiveness of this basic approach has been a major topic of recent software testing and security research. However, the basic strategy usually still fits into a simple basic model:

1. Select an input from the queue.
2. Mutate that input in order to obtain a new input.
3. Execute the new input, and if it is deemed interesting, add it to the queue.
4. Go back to the first step.

Any inputs that crash the compiler in step 3 are reported to the user. Using such a fuzzer is often extremely easy, involving no more work than 1) building the compiler with special instrumentation and 2) finding a set of initial programs to use as a corpus. Even compiling with instrumentation can be optional; some fuzzers (including AFL) can use QEMU to fuzz arbitrary binaries. However, for compilers, it is usually best if possible to rebuild the compiler, since QEMU-based execution is much slower, and compilers are slow enough to seriously degrade fuzzing throughput.

Our work focuses on improving step 2 of this process, in a way that is agnostic to how the details of the other aspects of fuzzing are implemented. In particular, the problem with most approaches to mutation in the literature, for compiler fuzzing, is that changes such as byte-level-transformations almost always take compiling programs that exercise interesting compiler behavior, and transform them into programs that don't make it past early stages of parsing. Alternative approaches to what are called "havoc"-style mutations tend to involve solving constraints or following taint, which in the case of compilers tends to be ineffective, since the relationships to be preserved are quite complex, and implemented in complex code. We propose a cheap way to produce a much larger number of useful, interesting mutations, without paying a price that makes fuzzing practically infeasible, or requiring any effort on the part of compiler developers.

### 2.2 Mutation Testing

A different use of the term "mutation" appears in the field of mutation testing. Mutation testing [3, 4, 6] is an approach to evaluating and improving software tests that works by introducing small syntactic changes into a program, under the assumption that if the original program was correct, then a program with slightly different semantics will be incorrect, and should be flagged as such by effective tests. Mutation testing is now widely used in software testing research, and is used to varying degrees in industry at-scale and for especially critical software development [1, 2, 7].

### 2.3 Limitations

The most important limitation for the mutation-testing-based approach is that if compiler *crashes* are mostly uninteresting, fuzzing of this kind will probably not be very useful. This applies, of course, to all AFL-style fuzzing, not just to fuzzing using the technique proposed in this paper. For example, C and C++ include a large variety of undefined behaviors. Code that crashes a C or C++ compiler, but that includes (unusual) undefined behavior may well be ignored by developers. Csmith [8] devotes a great deal of effort to avoiding generating code that falls outside the "ineresting" part of the language. On the other hand, many languages more recent than C and C++ attempt to provide a more "total" language where, while a program may be considered absurd by a human, fewer (or no) programs are undefined in the sense that C and C++ use the term. For example, smart contract languages such as those studied in this paper, generally aim to make all programs that compile well-defined, or at least minimize the problem to more managable cases such as order of evaluation of sub-expressions. Similarly, Rust code without use of `unsafe` or libraries using `unsafe` should be well-defined. For most more recent languages, and some older languages such as Java, a program that crashes the compiler is, in general, likely of interest to compiler developers. However, the proposed technique will be much more limited in effectiveness for C and C++ compilers.

## 3 Conclusions

## References


[1] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E. McKenney. 2017. Applying Mutation Analysis on Kernel Test Suites: an Experience Report. In *International Workshop on Mutation Analysis*. 110–115.

[2] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036

[3] Timothy Budd, Richard J. Lipton, Richard A DeMillo, and Frederick G Sayward. 1979. *Mutation analysis*. Yale University, Department of Computer Science.

[4] Richard J. Lipton, Richard A DeMillo, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

[5] V. Manes, H. Han, C. Han, s. cha, M. Egele, E. J. Schwartz, and M. Woo. 5555. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 01 (oct 5555), 1–1. https://doi.org/10.1109/TSE.2019.2946563

[6] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[7] Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (Gothenburg, Sweden) *(ICSE-SEIP '18)*. Association for Computing Machinery, New York, NY, USA, 163–171. https://doi.org/10.1145/3183519.3183521

[8] 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*. 283–294.