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, 5 pages. https://doi.org/10.1145/3486607.3486772

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

foo, baz

© 2022 Association for Computing Machinery.

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

1 Introduction

Compilers are notoriously hard to test, and modern optimizing compilers tend to contain many subtle bugs. Compiler bugs can have serious consequences, including potentially the introduction of security vulnerabilities that cannot be detected by human or static analysis without knowledge of the compiler flaw [2]. The literature on compiler testing is extensive [5] and goes back to such foundational papers as McKeeman's introduction of the idea of differential testing in the context of random testing of compilers [10].

As McKeeman's work suggests, one core approach to testing compilers is based on the generation of random programs. In recent years, the Csmith [13] project is perhaps the most prominent example of this method. However, builiding a tool such as Csmith itself is a heroic effort, requiring considerable expertise and development time. Csmith itself is over 30KLOC, much of it complex and with a lengthy development history. Csmith is focused on a single, albeit extremely important, language. Building a tool like Csmith for a new programming language is not within the scope of most language or compiler development projects. Even for such a highly visible language/compiler project as Rust/rustc, to our knowledge there is *no* useful tool for generating random Rust programs used in the compiler testing. As far as we can tell, Rust is primarily (or perhaps) only fuzzed at the whole language level (https://github.com/dwrensha/fuzz-rustc/blob/master/fuzz_target.rs) using libFuzzer, a tool with no special knowledge of Rust syntax or semantics, to randomly modify a set of supplied Rust programs. Similarly, the `solc` compiler, which essentially defines the Solidity language used to write most smart contracts for the Ethereum blockchain, is not fuzzed using a Csmith-like sophisticated generator, but using methods similar to those used for Rust (which will be discussed in more detail below), based on taking existing Solidity programs and randomly modifying them.

This approach (randomly mutating existing programs) is widely used by real-world compiler projects because *it is relatively easy to apply*. Most compiler projects, even large ones, do not have a team of random testing experts available, so the construction of Csmith-like tools is out of the question. This means that the only way to generate programs from scratch is to use a tool that takes as input the *grammar* of a language and generates random outputs satisfying the grammar. However, such an approach has multiple problems. First, in many cases the programs produced by a grammar, without extensive attention to tuning the tool, will be mostly uninteresting. Csmith is successful in part because of the use of numerous heuristics to generate interesting code. Second, the grammar of a language alone

seldom provides guidance in avoiding simple errors that cause programs to be rejected without exploring interesting compiler behavior; a BNF grammar generally will not, for example, force identifiers to be defined before they are used. Third, some interesting bugs can only be exposed by programs that may not satisfy a language's grammar, in theory, due to differences between a formal grammar and the actual parser used in a compiler, or other subtle implementation details. Finally, a usable grammar may not be available. In the early stages, many programming language projects lack a stable, well-defined grammar. The grammar used by a compiler implementation may be the only grammar around, but not be in a form suitable for use by a grammar-based generation tool. Thus, while grammar-based compiler testing has sometimes been extremely successful (e.g., the LangFuzz [7] approach), few compilers are actually tested using purely grammar-based tools.

Therefore, many projects rely on an approach we will define in more detail below, which uses off-the-shelf *fuzzing* tools, originally designed to find security vulnerabilities in programs to modify existing inputs (programs) rather than generate programs from scratch. This approach has found many subtle compiler bugs, but suffers from two major drawbacks:

1. First, in most cases the methods used by such programs to modify (mutate) inputs tend to take code that exercises interesting compiler behavior, and transform it into code that is rejected in the early stages of parsing. That is, in most cases, general-purpose fuzzers tend to take code and turn it into "non-code."
2. Second, when such fuzzers do find bugs, the bugs are often founded in particularly un-humanlike inputs, such as code containing non-printable bytes. Bugs are often at the "crash the parser" level rather than deeper semantic levels of the compiler.

Combined together, these problems make most compiler fuzzing performed in practice, even on major projects, both inefficient in terms of finding bugs and perhaps prone to find bugs that are not the most important and interesting compiler bugs.

Given that this kind of fuzzing is the only option available in practice to small compiler projects that cannot devote resources to tuning a grammar-based fuzzer (or perhaps even supporting an always-up-to-date grammar), improving the effectiveness of such compiler fuzzing is an obvious way to practically improve compiler testing. Ideally, such improvements would not require *any* additional effort or change the workflow of existing compiler fuzzing setups (other than changing the fuzz tool to be used).

This paper proposes one such improvement, based on changing the way in which general-purpose fuzzers modify (mutate) inputs. We augment the set of changes made by such tools with a large number of modifications drawn from the domain of *mutation testing*, which only modifies code in ways likely to preserve desirable properties such as the ability to get through a parser. We evaluate our technique on several real-world compilers, and show that it dramatically improves the mean number of distinct compiler bugs detected, and moreover produces a much larger set of distinct, successfully compiling, inputs that explore compiler behavior. As a result of our approach, which is available as an easy-to-use tool based on the widely used AFL fuzzer, we have reported more than 100 bugs in important real-world compilers, the great majority of which have been confirmed and fixed, and received a bug bounty for our efforts.

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 [9], the primary random testing approach used by many compiler projects, as discussed above. Again, we note that there are two basic kinds of compiler testing based on the generation of random inputs to a compiler. One, in recent years paradigmatically expressed in the Csmith tool [13], works by using a grammar and/or deep knowledge of the language accepted by the compiler, to generate progams to test the compiler. This is sometimes called *generative* fuzzing. Generative fuzzing can be very effective, but often requires expert tuning of a large, sophisticated tool, and at minimum requires having a suitable usable grammar for the language of the compiler. In practice, many compiler projects do not employ generative fuzzing for practical reasons.

A second approach, and the only approach widely used in many major compiler projects, is to use an off-the-shelf fuzzer, such as is used to find security vulnerabilities (e.g., the ubiquitous American Fuzzy Lop (afl) https://github.com/google/AFL) or libFuzzer, 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 [4, 8, 11] 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, 3, 12].

A mutation testing approach is defined by a set of mutation operations. Such operations vary widely in the literature, though a few, such as deleting a small portion of code (such as a statement) or replacing arithmetic and relational operations (e.g., changing + to - or == to <=), are very widely used. Most mutation testing tools parse the code to be mutated, and many do not work on code that does not parse. However, recently there has been a proposal to perform mutation testing using truly purely syntactic operations, defined by a set of regular expressions implemention operations [6]. Rather than taking a program, per se, this approach simply takes "code-like" text and produces a set of variants that, if the original text is compiling source code, will include most common mutations. The essence of this approach to mutation testing, which can be applied to "any language," is essentially a transformation from arbitrary bytes to arbitrary bytes that, if the original bytes are "code-like" will tend to preserve the property of being "code-like."

2.3 Combining Both Forms of Mutation

Our approach is, in essence, simple. We add a set of mutations to the repertoire of a mutation-based fuzzer, for use in compiler fuzzing. These mutations are either traditional mutation operators from the mutation testing domain or inspired by traditional mutation operators, but with changes made to satisfy the needs of fuzzing. The key point is that, unlike most changes made by mutation-based fuzzers, these mutations are likely to take interesting code inputs and preserve the property, e.g., that the input will get through a parser or trigger interesting optimizations. The tendency to preserve such properties is natural, since the basis of mutation testing is to take an existing program and produce a set of new, similar programs, by applying mutation operators. If most mutation operators tended to produce uninteresting code that doesn't even compile, mutation testing would not be of use to anyone. Moreover, because our approach is based on the idea of a "universal" mutation tool [6], the mutation operators used are generally language-agnostic, and useful for fuzzing any programming language, or at least a wide variety of real-world languages.

2.4 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 [13] 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` should not crash the compiler, and any such

crashes indicate possible bugs in the Rust compiler or type system. 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 Evaluation

3.1 Setup

Corpus. For solidity, all `.sol` files in `test/libsolidity`. For Move, all `.diem` files in the repository. For `Fe`, all `.fe` files in repository. AFL starts of preprocessing inputs based on coverage and ignores uninteresting ones. say how many for each proj

For the template splicing technique, we start off with a noop input and gradually generate `(template, fragment)` pairs, synthesized on-demand. After generating pairs,wWe removed all large inputs in the `(template, fragment)` corpus (> 4KB). say how many of these.

4 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] Scott Bauer, Pascal Cuoq, and John Regehr. 2017. *POC||GTFO*. No Starch Press, Chapter Compiler Bug Backdoors.

[3] 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

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

[5] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[6] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An Extensible, Regular-expression-based Tool for Multi-language Mutant Generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 25–28. https://doi.org/10.1145/3183440.3183485

[7] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.

[8] 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.

[9] 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

[10] William McKeeman. 1998. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation* 10(1) (1998), 100–107.

[11] 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.

[12] 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

[13] 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.

Project	Configuration	Unique Bugs			Avg Execs	Avg Paths	Avg Bitmap
		Avg	Min	Max	(Millions)	(K)	Cvg (%)
Solidity	`ground-truth`	3.69	1	6	35.8	12.0	54.34
	`text-mutation`	8.19	7	11	30.3	14.3	55.65
	`splice-mutation`	9.47	7	13	16.0	16.8	57.33
Move	`ground-truth`	????	?	??	56.9	4.9	63.23
	`text-mutation`	????	?	??	61.2	6.0	62.27
	`splice-mutation`	????	?	??	7.2	5.0	63.18
Zig	`ground-truth`	????	?	??	????	???	?????
	`text-mutation`	????	?	??	????	???	?????
	`splice-mutation`	????	?	??	????	???	?????

Table 1. Main results. We fuzzed each project for 16 trials (24 hours per trial) in three different configurations: `ground-truth`, `text-mutation`, and `splice-mutation`. `ground-truth` is stock `AFL`. `text-mutation` applies mutation operators (textual find-replace patterns) with a probability of 75% on every fuzzed input. Sock AFL manipulates the input witht the remainder, 25% of the time. `splice-mutation` is a hybrid approach that (1) applies mutation operators as in `text-mutation` with probability 33%; (2) synthesizes a syntax-aware input with template (splice) 33% of the time, and (3) uses stock AFL for the remainding 34%. TODO: summarize results once flush.