Making No-Fuss Compiler Fuzzing Effective

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. Making No-Fuss Compiler Fuzzing Effective. In ,. ACM, New York, NY, USA, 9 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

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 [11].

As McKeeman's work suggests, one core approach to testing compilers is based on the generation of random programs. In recent years, the Csmith [15] project is perhaps the most prominent example of this method. However, builiding a tool such as Csmith 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, C. Building a tool like Csmith for a new programming language is not within the scope of most language or compiler development projects, even very prominent ones. 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 (and none seems to be used in `rustc` 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) by using a wrapper around 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 generator, but using methods similar to those used for Rust, again based on taking existing Solidity programs and randomly modifying them.

This approach (randomly mutating existing programs) is widely used by real-world compiler projects in part because *it is often very easy to apply*: we call it "no fuss" compiler fuzzing. 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 valid 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 probabilities of productions, etc., will be mostly uninteresting. Csmith is successful in part

Technique	Tool	Requirements from Developers	Weaknesses
Custom tool (e.g. Csmith)	Custom tool	None	Extremely labor-intensive, potentially years of work
Grammar-based	Grammar-based fuzzer	Usable grammar	Needs tuning, many bugs not in scope
"No-fuss" mutation-based	Off-the-shelf fuzzer (e.g., AFL)	Corpus of examples	Inefficient, has trouble hitting "deep" bugs; may focus on "bad" code that should not compile but causes crashes

Table 1. Compiler Fuzzing Techniques

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, many 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. Salls et al. [14] elaborate this weakness in a recent paper, finding that many bugs could not be found using a grammar-based generator. Finally, a usable grammar may not be available, especially as the tools will expect a grammar in a particular format, and may add restrictions on the structure of the grammar. In the early stages, many programming language projects lack a stable, well-defined grammar in any formal, standalone, notation. An ad-hoc "grammar" used by the compiler implementation may be the only grammar around. Thus, while grammar-based compiler testing has sometimes been extremely successful (e.g., the LangFuzz [8] approach), few compilers are actually extensively tested using purely grammar-based tools.

Therefore, many projects rely on the approach we call "no-fuss" fuzzing, which uses off-the-shelf *fuzzing* tools, originally designed to find security vulnerabilities in programs, to modify existing inputs (e.g. regression test 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." This makes such fuzzing inefficient, and makes it almost impossible for it to find bugs requiring numerous subtle modifications of corpus programs.
2. Second, when such fuzzers do find bugs, the bugs are often found 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. Table 1 summarizes the existing widely-used compiler fuzzing techniques and their weaknesses.

Given that "no-fuss" fuzzing is widely used in large projects and is basically 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 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. In the longest-running campaign, that targeting the `solc` compiler for Solidity code, we were the first to report a very large number of serious bugs, despite continuous, extensive fuzzing using un-modified AFL performed by the compiler developers and external contributors, over the same time frame.

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 [10], 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 [15], 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. A second common approach, providing a *dictionary* of meaningful byte sequences in a language, is both burdensome on compiler developers (though less so than providing a full grammar), and limited in effectiveness: a dictionary cannot, for example, help the fuzzer delete meaningful sub-units of code, such as statements or blocks.

We propose a novel way to produce a much larger number of useful, interesting mutations for source code, without paying an analytical price that makes fuzzing practically infeasible for compilers, and without requiring *any* additional 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, 9, 12] 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, 13].

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 [7]. 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 [7], the mutation operators used are generally language-agnostic, and useful for fuzzing any programming language that syntactically resembles common languages (under which we include not only C-like languages, but even LISP-like languages based on s-expressions).

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 [15] 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 Implementation and Example Operations

The heart of the implementation is to add a set of mutation operators to the repertoire of changes that a mutation-based fuzzer can apply to an input. There is a large literature on the selection of mutation operators to apply, but this literature focuses on identifying operators that help find holes in a testing effort. There is no reason to believe that this is particularly indicative of the operators that will be most useful in fuzzing. We therefore simply used a large number of operators that apply to a wide variety of programming languages, based on the set of operators provided by the Universal Mutator tool, ignoring any highly-language-specific operators.

3.1 Fast or Smart?

More important than the selection of the best mutation operators (which will likely vary considerably by target compiler) is a fundamental decision. A mutation testing tool can be highly intelligent, only applying operators in ways that should produce compiling code, based on a parse of the program to be mutated. Or, like the universal mutator, it can be "dumb" and apply rules without expensive analysis of the code, trusting the compiler to prune the resulting invalid mutants. Which approach is best for fuzzing is not obvious: on the one hand, all fuzzing (including generative) relies on executing very large numbers of inputs; most "random" inputs will be uninteresting, neither exposing a bug nor novel behavior to drive further exploration. Fuzzer throughput is a critical factor, and a "dumb" mutation strategy can produce modified inputs much more rapidly than a "smart" approach that must parse the input. On the other hand, if a shallower analysis during mutation production greatly decreases the probability that the mutated inputs will expose bugs or new behavior, the result is, effectively, slower fuzzing. If adding a parsing stage makes mutation generation take twice as long, but more than doubles the probability the input generated will be useful, it will be a net gain for in-practice fuzzing throughput.

Of course, at first glance, it would appear that "smart" strategies are not even possible in our context: there will often not be a parser that the mutation tool could use. However, as we discuss below, recent work on multi-language syntax transformation enables an approach that can use *syntax fragments* to provide a significant degree of intelligent mutation without specialized parsers for a compiler's input language, at the cost of additonal time required to synthesize inputs.

```
case 0: /* Semantic statement deletion */
    strncpy(original, "\n", MAX_MUTANT_CHANGE);
    strncpy(replacement, "\nif (0==1)\n", MAX_MUTANT_CHANGE);
    break;
case 1:
    strncpy(original, "(", MAX_MUTANT_CHANGE);
    strncpy(replacement, "(!", MAX_MUTANT_CHANGE);
    break;
case 2:
    strncpy(original, "==", MAX_MUTANT_CHANGE);
    strncpy(replacement, "!=", MAX_MUTANT_CHANGE);
    break;
...
case 53: /* Swap comma delimited things case 4 */
   delim_swap(out_buf, temp_len, &original, &replacement, pos,
                        ",", ",", ")");
    break;
case 54: /* Just delete a line */
    delim_replace(out_buf, temp_len, &original, &replacement,
                         pos, "\n", "\n", "");
    break;
case 55: /* Delete something like "const" case 1 */
    delim_replace(out_buf, temp_len, &original, &replacement,
                         pos, " ", " ", "");
    break;
```

Figure 1. Part of the Fast String-Based Approximation of Mutation Operators

3.1.1 Fast String-level Approximation of Mutation Operators. The core implementation of our technique is a text-based approximation of the regular expression based approach taken by the universal mutator. Rather than call the mutation tool, which is written in Python and relatively slow, we hand-crafted, using low-level C string libraries, approximations of the mutation operators for all languages (the "universal" rules from the universal mutator) and those for "C-like" languages. Figure 1 shows part of the implementation. Most operators are implemented by choosing a string to find and a string to repalce it with; the mutator finds a random occurrence of the `original` string and replaces it with the `replacement` string. Other operators require more involved string manipulation, e.g., removing a semicolon-delimited statement, or swapping function arguments. Critically, however, all operations involve only basic C string operations, and no more than 4 linear scans of the entire text to be mutated. The vast majority of operations require no more than one linear scan in the worst case. When an operation that is chosen cannot be applied (e.g., the string to be replaced is not present), another operation is attempted, up to a maximum number of tries.

This approach is, as stated, is fast. While slower than many built-in AFL mutations (obviously searching for strings is slower than flipping a randomly chosen bit, or incrementing a byte value), it has a fairly low upper bound on worst-case runtime. The time required is much closer to AFL's built in mutations than to techniques such as solving constraints, even a linear approximation [6]. Figure 2 shows some sample transformations of inputs using this approach. Notice that some of the mutations tend to delete code, potentially large amounts of code. This is critical for enabling the fuzzer engine to compose interesting inputs, in that the larger two inputs are, the more likely they will have, e.g., namespace conflicts that prevent merging them.

Original code:

```
...
 int bar(int x, int y, int z) {
   if (x < y)
      return foo(x, y, z);
   while (x < y) {
      x++;
      z = z * 2;
   }
}
```

Mutant 1:

```
...
   if (x == y)
      return foo(x, y, z);
```

Mutant 2:

```
...
   if (x < y)
      return foo(x, z, y);
```

Mutant 3:

```
...
   while (x < y) {
      x++;
      break;
```

Mutant 4:

```
...
   while (x < y) {
   }
```

Mutant 5:

```
...
   while (x < y) {
     x++;
   }
```

Figure 2. Mutations of Simple Code

3.1.2 Smart Syntax-Aware Mutation. Our core approach enables fast mutation written in C that we added directly in AFL's fuzzing hot loop. Our early success with this method prompted us to augment AFL further with even smarter mutations to find more bugs more quickly. The intuition behind these smarter mutations is to manipulate (typically larger) code fragments that are likely to be syntactically valid and yet trigger deeper buggy properties (past compiler parsing). Unlike transformtions in the core approach that approximate syntactically valid transformations on strings or lines, syntax-aware transformations seek to accurately modify syntax in well-parenthesized expressions or entire multiline code blocks (e.g., function or `for`-loop bodies). In general, manipulating a program's parse tree like this imposes exactly the kind of burden and complexity that small compiler projects can't support (defining a precise grammar, keeping tooling up to date as the grammar evolves). Even with multiyear investment in tools (like Csmith), effort necessarily goes into deeper focus on language-specific properties and semantics that may not generalize to other compilers.

Our approach combats this cost by using the Comby[1] tool to do syntax-aware code matching and transformation. Comby works by coarsely parsing a program, taking care to correctly interpret nested syntax of code (parentheses and braces), and avoids conflating this syntax with strings and comments like regular expressions would. Comby is not a fully-fledged parser for any language, but it *is* language-aware, in that it recognizes a small set of language-specific constructs (e.g., syntax to identify quoted strings or comments) to accurately parse code blocks. Comby supports over 50 languages and uses a generic parser yet-unsupported ones like custom DSLs.

Comby does not have C bindings, so we expose its transformation abilities as a server to our AFL fuzzer. We implemented a minimal HTTP client in the fuzzer's hot loop to request inputs. We use the Comby server in a mutation mode where it *generates* inputs from a decomposition of *templates* and *concrete fragments* obtained from the initial corpus of programs. We first preprocess all programs in the initial corpus to obtain this decomposition. For example, given the input program:

```
function f(uint256 arg) public {
    f(notfound);
}
```

We obtain three templates by substituting expressions inside well-nested parentheses and blocks with a placeholder $1:

```
function f(uint256 arg) public {$1}
```

```
function f($1) public {
    f(notfound);
}
```

```
function f(uint256 arg) public {
    f($1);
}
```

And three corresponding concrete fragments that are substituted for:

```
uint256 arg
f(notfound);
notfound
```

The decomposition is itself done by Comby, which extracts concrete values inside parentheses, braces, and square brackets (patterns `($expr)`, `{$expr}`, `[$expr]` respectively). Note that our approach can be customized to extract *any* kind of syntactically significant pattern correspondingly decompose the input program; we simply chose syntax that commonly delineates code blocks and expressions (i.e., parentheses and braces) since these exhibit interesting properties that preserve structure (multiline statements, well-formed expressions) that go beyond what string-level mutations identify.

We perform this decomposition for all programs in the corpus to obtain templates and concrete fragments, which we then deduplicate. Once fuzzing, the server generates an input program by selecting, uniformly at random, a template and up to 10 program fragments and then substitutes all locations in the template with program fragments. In essence, the server splices new inputs that are likely to compose syntactically well-formed programs.

In addition to generating inputs, the server can also apply syntax-aware Comby transformation rules, analogous to string-level mutation operators. In practice, our server architecture adds considerable overhead (we discuss in Evaluation) that is slow in applying on-the-fly mutations, and less appealing than string-level mutation in Section 3.1.1. Our results in Section 4 *do* show, however, that generative syntax-aware input manipulation demonstrates compelling utility despite incurring significant slowdown.[2]

3.1.3 P(havoc) + P(text) + P(splice) = 1.

Our full implementation is based on Google's released code for the AFL fuzzer, and available as an open source tool (that, to date, has 68 stars on GitHub and has been forked 7 times).[3] The main change to AFL is the addition of code such as that shown in Figure 1. The new version (the "AFL compiler fuzzer") can also call out to comby to generate mutants. Two new command line parameters to AFL control the use of these features: `-1` determines the probability to generate a mutant using the fast C string implementation (with a default value of 75%), and `-2` determines the probability to call comby to generate a mutant (with a default value of 0%). If these two parameters add up to less than 100%, the remainder of the time the usual AFL havoc mutation operators are applied.

4 Evaluation

4.1 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.

[1]https://comby.dev

[2]We are actively improving the tooling architecture, which is generally a matter of engineering rather than limitations on the inherent merits of our approach.

[3]Link omitted for blinding.

Project	Configuration	Unique Bugs Avg	Unique Bugs Min	Unique Bugs Max	Avg Execs (Millions)	Avg Paths (K)	Avg Bitmap Cvg (%)	Queue Size (K)	Queue Compiles	Queue Error
Solidity	`AFL-baseline`	3.69	1	6	35.8	12.0	54.34	14.5	20%	80%
	`AFL++ 3.15a`	5.63	1	10	56.9	8.8	20.58	11.2	33%	67%
	`text-mutation`	7.81	7	11	30.3	14.3	55.65	16.7	31%	69%
	`splice-mutation`	11.81	7	14	16.0	16.8	57.33	16.8	34%	66%
Move	`AFL-basline`	7.19	6	8	56.9	4.9	63.23			
	`AFL++ 2.54b`	????	?	?	????	???	?????			
	`text-mutation`	8.31	7	9	61.2	6.0	62.27			
	`splice-mutation`	6.06	5	7	7.2	5.0	63.18			
Fe	`AFL-baseline`	6.56	5	7	24.5	3.5	27.91			
	`AFL++ 2.64c`	6.44	5	8	22.6	3.4	27.76			
	`text-mutation`	6.50	5	7	17.9	3.3	27.84			
	`splice-mutation`	6.94	6	9	5.0	2.6	27.83			
Zig	`AFL-baseline`	????	?	??	2.2	3.3	40.99			
	`text-mutation`	????	?	??	2.1	3.3	40.95			
	`splice-mutation`	????	?	??	1.3	3.9	41.82			

Table 2. Main results. We fuzzed each project for 16 trials (24 hours per trial) in different configurations: `baseline-AFL`, `AF++`, `text-mutation`, and `splice-mutation`. `baseline-AFL` is stock `AFL`; `AFL++` is a community-driven effort that enhances 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 remaining 34%. TODO: summarize results once flush.

Project	Length	Total	✓	⏲	–	✗
Solidity			2 + ??	??	??	??
Move	???	14	12	2	0	0
Fe						
Zig	???	2	0	2	0	0
All	???	16	14	4	0	0

Table 3. Fuzzing campaign results for real world bugs. Table shows all reported bugs to project's upstream issue tracker and status. ✓ is **fixed** bugs. ⏲ is **confirmed but unfixed bugs**. – is **duplicate bug reports** (either reported duplicatively by us or another contributor). ✗ is **invalid** bug reports (deemed not a real bug, i.e., WONTFIX). **Total bugs** is the number of true, unique bugs reported and acknowledged by maintainers (✓ Fixed + ⏲ Unfixed bugs). Table to include Alex's bugs still. Alex to answer: what do want to put for "campaign length" here? Fuzz time, or time span of bug reports? Or both (and we add a column)?

5 Non-Experimental Fuzzing Campaigns: Bugs Reported

WE NEED A NICE TABLE OF BUG CATEGORIES FOR ALL THE CAMPAIGNS WE RAN, INCLUDING CAMPAIGN LENGTH

Perhaps the most important evidence of the effectiveness of this approach is that we applied it to fuzzing the Solidity compiler for over a year, and in that time reported XX bugs that have been fixed. Prior to and during our campaign, Solidity had been fuzzed heavily using AFL, by the developers and by external contributors. Despite competing with the internal fuzzing team of the project and other developers, and never devoting more resources to the fuzzing than 3-4 docker container hosted instances of our fuzzing tool, running on a high-end laptop, we believe that our campaign was the largest single source of fuzzing-discovered bugs in the compiler during our campaign. The campaign was awarded a security bounty of $1,000 USD in Ethereum for discovery of a bug with potential security implications (and, it was noted, for the general effectiveness of the fuzzing), and the Solidity team encouraged and aided our efforts, once it was clear that the approach was very useful in exposing subtle bugs.

A second long-term fuzzing effort was directed at the Fe language, a Rust/Python-like alternative to Solidity for writing Ethereum contracts. Fe is an experimental language, and the project has far fewer resources than Solidity to devote to testing. We worked with the Fe developers to make Fe crash in additional cases, and were able to provide them with high-quality fuzzing very early in the lifetime of an experimental compiler project. We believe this effort was very useful to the

Project	Bug kind	Ref	Time to fix
Solidity			
Move	incorrect struct type arity	#7401	1d
	broken ref type constraint	#7613	1d
	...		
Fe			

Table 4. Detailed bug exemplars. **Time to fix** is the time taken from report to fix being merged.

Fe team, based on their comments, and speculate that better "no-fuss" fuzzing could expose language corner cases early in the implementation of a compiler, avoiding costly changes when more code depends on erroneous assumptions, or poor language design choices.

We also ran shorter, less-intensive campaigns against other compilers.

6 Related Work

Research on compiler testing, as noted in the introduction, has been an important subfield overlapping compiler development and design and software engineering and testing, for many years. Chen et al. summarize much of this work in a recent survey [5].

To our knowledge, very little work has appeared targeting the problem this paper addresses: improving the ability of general-purpose fuzzers to find (interesting) bugs in compilers. The recent work of Salls et al. [14], however, specifically aims to improve general-purpose fuzzer performance on compilers and interpreters. Their approach, which they call "token-level fuzzing" essentially produces a hybrid level in between grammar-based generation and "byte-level" mutation-based fuzzing. The core of their idea is to replace the largely byte-level mutations of AFL etc. with mutations at the *token* level of a grammar. They summarize the idea as "valid tokens should be replaced with valid tokens" [14]. In a sense, this extends the idea of using a dictionary, but with important changes: token-level fuzzing *only* applies token-level, not byte-level mutations, but also adds the composition of multiple token additions and substitutions to the set of single-step mutations. Token-level fuzzing is an attractive and useful idea, somewhat orthogonal to our approach. However, unlike our approach, token-level fuzzing does not apply AFL's havoc operations, so some bugs are simply not possible to find using token-level fuzzing (e.g., ones involving injecting unprintable characters in strings, including our Solidity bug earning a security bounty). In this sense, token-level fuzzing has some of the limitations of grammar-based generation. Token-level fuzzing also provides little help to a fuzzer in deleting large chunks of code, since this often would require a very large number of token operations, though the approach does include a way to copy statements from one input to another. Finally, token-level fuzzing requires using a lexer to find all tokens in input seeds, and if tokens not in those seeds would be useful, developers must provide any additional tokens. This requires modifying the fuzzing workflow to add token pre-processing, and is no longer strictly *no-fuss*.

7 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, Chun-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] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the International Conference on Software Engineering*. 736–747.

[7] 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

[8] 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.

[9] 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.

[10] 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

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

[12] 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.

[13] 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

[14] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*.

[15] 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.