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 augmenting it with operators taken from *mutation testing*, and program splicing. We conduct a controlled study to show that our hybrid approaches significantly improve fuzzing effectiveness qualitatively (consistently finding unique bugs that baseline approaches do not) and quantitatively (typically finding more unique bugs in the same time span, despite fewer program executions). Our easy-to-apply 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.

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, 11 pages. 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 without knowledge of a compiler flaw [2]. The literature on compiler testing is extensive [7].

As McKeeman's [17] widely cited paper suggests, one core approach to testing compilers is based on the generation of *random programs*. Csmith [23] is perhaps the most prominent example of this method. 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 compiler projects, even major ones. For instance, to our knowledge there is *no* useful tool for generating random Rust programs (none seems to be prominently featured in `rustc` testing). 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 knowledge of Rust, to randomly modify *a set of supplied Rust programs*. Similarly, the `solc` compiler, used for most smart contracts on the Ethereum blockchain, is fuzzed using methods similar to those used for Rust[1]; we call these approaches, based on mutating a starting set of programs, *no-fuss* fuzzing.

Most compiler projects, even large ones, do not have a team of spare random testing and compiler/language 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 a *grammar*, 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 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; e.g., forcing identifiers to be defined before they are used. Third, many interesting bugs can *only* be exposed by programs that do not satisfy a language's supposed grammar, due to differences between a formal grammar and the actual parser used in a compiler, or other subtle implementation details. Salls et al. [20] found that many bugs could not be discovered using a grammar-based generator. Finally, a usable grammar simply may not be available, especially as the tools will expect a grammar in a particular format (e.g. antlr4), and may add restrictions

foo, baz

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

[1] Creating a grammar-based fuzzer has been an open issue for Solidity since August of 2020 (https://github.com/ethereum/solidity/issues/9673).

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 least interesting bugs

Table 1. Compiler Fuzzing Techniques

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 [13], few compilers are actually extensively tested that way.

Unfortunately, "no-fuss" fuzzing must make use of off-the-shelf *fuzzing* tools, originally designed to find security vulnerabilities in inputs treated largely as byte-streams. No-fuzz fuzzing therefore suffers from two major drawbacks:

1. The methods used by fuzzers to mutate inputs tend to take code that exercises interesting compiler behavior, and transform it into code that is rejected by the parser. This is inefficient, and makes it almost impossible to find bugs requiring a sequence of subtle modifications.
2. Bugs are often found via very un-humanlike inputs.

Combined together, these problems tend to make most compiler fuzzing performed in practice inefficient in terms of finding bugs and prone to find less interesting 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 may be the *only* option available in practice to small compiler projects, improving the effectiveness of no-fuss fuzzing is an obvious way to practically improve compiler testing. Ideally, such improvements would not require *any* additional effort on the part of developers.

This paper proposes one such improvement, based on changing the way in which general-purpose fuzzers modify (mutate) inputs. We augment the set of primarily byte-based 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—like the ability to get through a parser. Figure 1 is one such input generated by our approach, yielding a syntacticaly well-formed program that triggers deeper behavior in the compiler's optimization routines.

We evaluate our technique on four real-world compilers, and show that it significantly improves the mean number of distinct compiler bugs detected. We have reported more than 100 previously undiscovered bugs, subsequently fixed, and received a bug bounty for our efforts. In the longest-running campaign, that targeting the `solc` compiler for Solidity code,

```
contract C {
  function fun_x () public {}
  function fun_y () public {}
  function f() public
  {
    int h=true?1:3;
  }
  function () r=true?fun_x:fun_y;
}
```

Figure 1. An example of an early crash-inducing Solidity program found with our approach (the bug was submitted and fixed). The combination of expressions and function delcarations trigger complex behavior in an optimization routine that attempts to deduplicate low level code blocks.

we were the first to report a large number of serious bugs, despite extensive fuzzing performed by the developers, OSS-Fuzz, and external contributors.

2 Mutation-Testing-Based Fuzzing

2.1 Mutation-Based Fuzzing

One use of the term "mutation" appears in the context of the "no-fuss" *mutation-based* fuzzing discussed above [16]. A fuzzer such as AFL operates by executing the program under test (here, the compiler) on inputs (initially those in a corpus of example programs), using instrumentation to determine code coverage for each executed input. The fuzzer then takes inputs that look interesting (e.g., uniquely cover some compiler code) and adds them to a *queue*. The basic loop repeatedly takes some input from the queue, *mutates* it by making some essentially random change (e.g., flipping a single bit, or removing a random chunk of bytes), then executes the new, mutated input under instrumentation, and adds the new input to the queue if it satisfies the fuzzer's coverage-based criteria for interesting inputs. 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 [4, 14, 16]. However, the inner fuzzing loop strategy is simple:

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 interesting, add it to the queue. Then repeat the process from step 1.

Inputs that crash the compiler in step 3 are reported as bugs. Using such a fuzzer is often extremely easy, requiring only 1) building the compiler with special instrumentation[2] and 2) finding a set of initial programs to use as a corpus.

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 [10] or following taint [8], which in the case of compilers tends to be ineffective, since the relationships are too complex to solve/follow. A second common approach, providing a *dictionary* of meaningful byte sequences in a language, is both burdensome on compiler developers and limited in effectiveness: a dictionary cannot, for example, help the fuzzer delete 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 [6, 15, 18] 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 the tests. Mutation testing is used in software testing research, occasionally in industry at-scale, and in some critical open-source work [1, 3, 19].

A mutation testing approach is defined by a set of mutation operators. Such operators 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 do not work on code that does not parse. However, recently there has been a proposal to perform mutation using purely syntactic operations, defined by a set of regular expressions [12]. Rather than taking a program, per se, this approach simply takes "code-like" text and produces a set of variants that 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 that property*. The resemblance to the mutations performed by mutation-based fuzzers is the core insight behind our approach.

2.3 Combining Both Forms of Mutation

The core of our approach is simply to add a new set of mutations to the repertoire of a mutation-based fuzzer. These mutations are either traditional mutation operators or inspired by traditional mutation operators, but with changes made to satisfy the needs of fuzzing. Unlike most changes made by mutation-based fuzzers, these mutations are likely to preserve the property that an 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. If most mutation operators tended to produce uninteresting code that doesn't even compile, mutation testing would not be of much use. Moreover, because our approach is based on the idea of a "universal" mutation tool [12], 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). Staying within the theme of preserving code structure, we further enable an augmentation of our approach where we decompose existing test programs into constituent fragments that are then used to synthesize and mutate new inputs at runtime.

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 our approach. For example, code that crashes a C or C++ compiler, but that includes extensive undefined behavior may well be ignored by developers. Csmith [23] devotes a great deal of effort to avoiding generating such code. 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. 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.

[2] With AFL this is fairly trivial, by using a drop-in compiler replacement, for C, C++, Rust; there are AFL variants for Go, Python, and other languages, as well. AFL can also 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 fuzz throughput is important.

3 Implementation

The heart of the implementation is to implement a set of mutation-testing operators so that a mutation-based fuzzer can apply them to inputs. There is a large literature on the selection of mutation operators for mutation testing, 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, and there is some suggestion that such approaches do not outperform random selection in any case [11]. We therefore 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.

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 a compiler to prune 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 for us: there will often not be a parser that the tool could use. However, as we discuss below, recent work on multi-language syntax transformation [22] 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.

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 2 shows part of the implementation. Most operators are implemented by choosing a string to find and a string to replace 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, and most scans terminate before scanning a large fragment of the input. 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.

```
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 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, " ", " ", "");
```

Figure 2. Part of the Fast String-Based Approximation

This approach is, as stated, 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, and very good average runtime. The time required is much closer to AFL's built in mutations than to techniques such as solving constraints, even a linear approximation [10], and is successful much, much, more often than solving constraints over compiler inputs, which are among the hardest conceivable for modern SMT solvers or linear approximations to handle. Figure 3b shows some sample transformations of inputs using this approach. Note 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.

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

(a) Original code

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

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

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

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

(b) Four mutations

Figure 3. Mutations of Simple Code

3.1.2 Smart Syntax-Aware Mutation. Our core approach uses fast mutation written in C that we added directly to 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 multi-year investment in tools, 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`[3] tool [22] 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, as regular expressions (and our string approach) tends to do. `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 for unsupported languages like custom DSLs. `Comby` is likely to perform "well-enough" for any language anyone is likely to want to input to a compiler.

[3]https://comby.dev

`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. The following figure illustrates the decomposition of an example Zig program.

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

decompose

`templates`

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

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

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

`fragments`

```
f(notfound);
```

```
uint256 arg
```

```
notfound
```

A "decompose" operation yields three `templates` in the left column (`$1` are placeholders for future substitution) and extracts three corresponding concrete `fragments`, shown in the right column. The "decompose" operation is done with `Comby`, which extracts concrete values inside parentheses, braces, and square brackets (patterns `($expr)`, `{$expr}`, `[$expr]` respectively). Note that our approach can be customized to extract *any* kind of syntactically significant pattern and 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 this in

Section 4) and is thus slow in applying on-the-fly mutations, and in that sense less appealing than the string-level mutation. Our results in Section 4 do show, however, that generative syntax-aware input manipulation demonstrates compelling utility despite incurring significant slowdown.[4]

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

Our full implementation is based on Google's released code for the AFL fuzzer (https://github.com/google/AFL), and available as an open source tool (that, to date, has 68 stars on GitHub, has been forked 7 times, and has external users who contact us).[5] The main change to AFL is the addition of code such as that shown in Figure 2. 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 stock AFL havoc mutation operators are applied; by default, this happens 25% of the time.

4 Evaluation

We ran a a controlled experiment to evaluate the effectiveness of two new strategies based on our approaches in Sections 3.1.1 and 3.1.2 to improve "no-fuss" compiler fuzzing. Our main goal is to answer to what extent these low-effort strategies demonstrate significant benefit in the domain of compiler fuzzing, and how they influence fuzzer behavior and performance.

4.1 Experimental Setup

We evaluate our approach on four compilers, for the languages Solidity, Move, Fe and Zig[6]. Solidity is a high-profile language for writing smart contracts on the Ethereum blockchain, and very widely used. Fe is an experimental statically-typed language for Ethereum smart contracts. Move is Facebook's smart contract language, developed for the industrial blockchain solution Diem[7]. Zig is an up-and-coming systems language operating in the same space as C, Rust, Nim, and other statically-typed languages with manual memory management. Move and Fe are implemented in Rust, Solidity is implemented in C++, and Zig is implemented in a mix of C++ and Zig itself.

Fuzzer configurations. We perform a comparative evaluation of four fuzzing configurations. The first configuration is stock `AFL` in "quick & dirty" mode, our baseline of comparison across all projects. Because our approach is integrated directly into stock `AFL`, we're comparing to the baseline *implementation* (a desirable practice for sound comparative fuzzer evaluations [4]). Although `AFL` continues to be a de facto industry standard for "no-fuss" fuzzing, numerous community-driven improvements have been made to the `AFL++` project, which can often outperform stock `AFL`. Therefore, for extra measure, we seek to compare our results to a second fuzzing configuration using the existing `AFL++` tool. We successfully configured `AFL++` for three of our four compilers (excluding Zig), and report our results in Section 4.2. Note, however, that since our technique is not implemented in `AFL++`, the comparison is incongruent, and potentially handicapped by orthogonal `AFL++` improvements that may stand to boost our approach.[8]

The third and fourth configurations we compare are both strategies based on our new approaches in Section 3. The third configuration applies purely string-level mutations as described in Section 3.1.1 with **75%** probability. The fourth configuration augments the pure string-level mutation strategy with syntax-aware mutation (Section 3.1.2), where our `AFL` has **33%** probability to request that the server generate a new input (using template splicing), **33%** probability to perform string-level mutation on the input, and **34%** probability to run `AFL` as usual.

We chose the ratios in our strategies with a best-effort method by running just a single 24 hour trial on a single project (Solidity) for various configurations (e.g., 90% string-level mutation, 75% pure syntax-aware mutation, and 25%-50% ratio). We found the 75% and 33%-33% strategies to be the best candidates to evaluate deeply over many hours of fuzzing. We note the especially appealing avenue of future work for devising optimal selections of these parameters (perhaps based on language attributes, or input corpora).

Fuzzing trials and duration. We ran 14 trials per fuzzing configuration for each compiler to control for variability and randomness. We ran four configurations for the Solidity, Move, and Fe compilers (where `AFL++` is included), and three configurations for Zig (`AFL++` excluded) for a total of 210 trials. A single trial comprises 24 hours of fuzzing on a single core, starting from the initial input corpus. We chose 24 hour trials because our intent is to answer whether we can observe (relatively immediate) effects of strategies that aim to surface deeper bugs. Our choice aligns with existing work that shows finding new vulnerabilities earlier during a fuzzing campaign is proportionally cheaper (more likely) than long-running campaigns [5]. I.e., if our strategies exhibit any significant competitive advantage, we expect it to manifest early in a controlled setting (within 24 hours). In aggregate, our experiments represent 210 days of fuzzing to demonstrate fuzzer performance for quickly surfacing

[4] We are actively improving the tooling architecture, a matter of engineering rather than limitations on the inherent merits of the approach.
[5] Link omitted for blinding.
[6] See solidityland.org, move-book.com, fe-lang.org and ziglang.org, respectively.
[7] diem.com
[8] Indeed, recent improvements, not available during our first implementation, compel us to implement our approach in `AFL++`.

bugs with our “no-fuss” enhancements on these compilers. Each project was fuzzed at an early commit before we had reported bugs to the upstream repository.

Input corpora and preprocessing All fuzzer trials ran over inputs derived from the project's own source tree. A summary is shown in Table 2. For example, the Solidity base corpus is 2,447 **Files** ending in `.sol` in the `test/libsolidity` subdirectory. For fuzzer trials using syntax-aware input generation, we decompose the base corpus into unique templates (**Templ.**) and concrete program fragments (**Frag.**). Because this process can yield very large (and therefore slow) inputs during generation, we remove all templates and fragments larger than 4KB. For Solidity, the base corpus decomposes into 9,308 templates and 7,651 concrete program fragments. The remaining project corpora are as follows:

Proj	Source	Files	Templ.	Frag.
Solidity	`.sol` test files	2,447	9,308	7,651
Move	all `.diem`	1,103	9,650	10,916
Fe	all `.fe`	126	253	153
Zig	compile-error tests	586	1,762	1,562

Table 2. Summary of input corpora for four compilers.

Creating **Templ.** and **Frag.** input components incurs some preprocessing per project. This time ranges from less than 2 minutes (for Zig) to 2 hours (for Move). Our setup does not attempt to adjust the 24 hour fuzzing time to compensate for this preprocessing, because in a real world setting this (relatevely small) preprocessing cost is greatly amortized over the time of a real fuzzing campaign that typically lasts much longer than 24 hours. Even in our experimental setup, this cost, which would account for for less than 5% of the fuzzing time on average for a single trial, is amortized over multiple 24 hour trials, and thus difficult to adjust for fairly. A benefit of this upfront per-project process is that our fuzzer trials using the 33% syntax-aware mutation strategy start fuzzing with an empty, “zero” program and rely entirely on the probability to potentially generate new combinations inputs from the **Templ.** and **Frag.** at runtime. This is architecturally distinct from our other `AFL` fuzzing configurations (both the baseline and pure string-level mutations) that run preprocessing bundled with AFL, filtering initial inputs (**Files**) based on code coverage (which may take seconds to minutes, or even longer, depending on corpus size) before fuzzing begins. In brief, preprocessing times in our configurations are not directly comparable, but it is reasonable to assume that the costs for all configurations converge to zero in realistic, long-running campaigns.

Hardware. Each trial ran on Ubuntu 18.04, on a single core of Intel Xeon Gold 6240 2.6 GHz CPUs, and with up to 30GB free RAM.

4.2 Results

Our main result is that our enhancements with string-level and syntax-aware mutation consistently uncover unique bugs missed by `AFL` and `AFL++`, often performing better and yielding a higher overall discovery of unique bugs. Which strategy is favorable varies per project, and in some cases, one of our strategies underperforms due to the “fast or smart?” tradeoff. We first give an overview of results followed by notable thematic observations.

Overview. Table 3 summarizes the fuzzing runs for each project and configuration pair. A row in the table represents the average number over 14 trials for a project in that configuration. `AFL-baseline` is stock `AFL`'s results. For projects Solidity, Move, and Fe, we include `AFL++`'s results. The `AFL++` version varies based on the version compatible with a project's commit at which we started fuzzing. All **Unique Bugs** for Solidity, Move, and Fe are determined by classifying crashing inputs by bug-fixing commits, except for at most 2 unique, yet unfixed Solidity bugs manually inspected by an expert author. We assume a single commit fixes a single bug, an assumption that is consistent with prior work [9, 21], general software development practice, and our manual inspection. I.e., **Unique bugs** is a precise ground truth of truly unique bugs. **Unique Bugs** for Zig are largely unfixed at the time of writing, and are instead comprehensively classified by an expert author manually inspecting every crashing input and its error message. Not reported in the table, a proportion of (potentially duplicative) crashing inputs for `Move` were found by all configurations that remain untriaged, because we could not find a bug-fixing commit, or because no fix exists yet.

In general, either our `text-mutation` or `splice-mutation`, or both, does better than existing tools. One of our strongest results shows that the hybrid `splice-mutation` strategy discovers roughly 8 more bugs than `AFL` (approximately 3× as many) on Solidity on average. For Solidity, the simple approach of `text-mutation` also performs well, discovering roughly 4 more bugs than AFL. Especially interesting, the predictability of `AFL` and `AFL++` bug discovery on Solidity varies highly, sometimes just finding a single unique bug (Min = 1) in a trial. The root cause is that both these fuzzers get “stuck” exploring a parser bug,[9] thinking that every new crash is interesting. Our approaches didn't fall prey to this pathological behavior, and always found at least 7 unique bugs, which is likely accounted for by variability in input mutation. Another strong result is the discovery of more than 2× as many unique bugs found via `splice-mutation` in Zig compared to other approaches. Somewhat similar to Solidity, `splice-mutation` on Zig overcomes a pathological behavior where usual `AFL` fuzzing finds extremely large inputs “interesting”, but they do not actually crash the compiler. This behavior affects all

[9] Commit 0b9c84 in `github.com/ethereum/solidity`.

Project	Configuration	Unique Bugs Avg	Unique Bugs Min	Unique Bugs Max	Avg Execs (Millions)	Avg Paths (K)	Avg Bitmap Cvg (%)	Compiles (K)	Compiles (%)
Solidity	`AFL-baseline`	3.79	1	6	35.6	12.1	54.32	2.94	20.2
	`AFL++ 3.15a`	5.21	1	8	57.9	8.8	20.59†	3.76	33.4
	`text-mutation`	7.57	7	9	30.3	14.2	55.65	5.48	32.7
	`splice-mutation`	11.79	7	14	16.0	16.8	57.32	5.24	31.1
Move	`AFL-baseline`	7.14	6	8	56.3	4.9	63.21	1.77	29.5
	`AFL++ 2.64c`	6.36	5	7	47.7	4.5	62.43	1.63	28.7
	`text-mutation`	8.43	7	9	61.5	6.0	63.28	2.38	33.2
	`splice-mutation`	6.00	5	8	7.2	5.0	63.16	1.21	23.8
Fe	`AFL-baseline`	6.57	5	7	24.3	3.5	27.93	0.55	14.8
	`AFL++ 2.64c`	6.50	5	8	22.8	3.4	27.76	0.48	13.2
	`text-mutation`	6.50	5	7	17.9	3.3	27.84	0.47	13.5
	`splice-mutation`	6.93	6	9	6.0	2.6	27.84	0.42	15.1
Zig	`AFL-baseline`	2.57	1	5	2.2	3.4	40.99	0.12	3.2
	`text-mutation`	2.36	0	4	2.1	3.3	40.96	0.13	3.3
	`splice-mutation`	7.64	0	13	1.3	3.9	41.84	0.27	7.1

Table 3. Main results of controlled experiment. We fuzzed each project for 14 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 fast string-based mutation operators (textual find-replace patterns) with a probability of 75% on every fuzzed input. Stock AFL manipulates the input the remaining 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) with probability 33%; and (3) uses stock AFL the remaining 34% of the time. †The instrumentation for a more recent version `3.15a` of `AFL++` differs from our baseline `AFL` version and is the likely reason for a difference in reported coverage %.

configurations (accounting for 0 unique bugs for some trials with `text-mutation` and `splice-mutation`) but is effectively suppressed on average by variation created with `splice-mutation`. We expand more on the qualitative appeal of our approaches below. On the whole, our weakest result reveals two related insights:

- One of our strategies may not perform better than stock approaches. This is the case for Fe, where `text-mutation` performs only on par with `AFL`. Here, `splice-mutation` is the only strategy to perform slightly better than other tools.
- One of our strategies may underperform compared to stock approaches. This is the case for Move, where `splice-mutation` performs worse than the baseline, whereas `text-mutation` performs best.

These observations highlight the potential *tradeoffs* of "fast or smart?" mutation. In the former case, simple mutations are not enough to enrich the search space and discover more bugs, but the smarter hybrid variety does perform marginally better, despite being more than 3 times *slower* than all other approaches by average number of executions (**Avg. Execs**). Conversely, the quick `text-mutation` approach does best in the Move project, where the `splice-mutation` hybrid variety is just too slow (almost 8× slower) to compensate for its "smart" benefit.

A look at quick and exclusive findings. With exception to Fe, the improvement of our best approach for each project is statistically significant (by Wilcoxon rank-sum test, $p < 0.05$). But rather than a statistical measure, which treats every unique bug as equal, perhaps the most compelling case for choosing "fast or smart" mutation enhancements lies in the qualitative properties of our approaches. Specifically, we found that our approaches tend to discover bugs that are exclusive to a particular strategy (requiring a rather unique alteration of inputs to find), or to identify unique, not-so-shallow bugs *quickly*. In the former case, for instance, looking into qualitative results of Fe revealed that the slightly better bug discovery of `splice-mutation` could be attributed to a *consistent* discovery of a Rust borrow error (triggered in all 14 trials) that was never discovered by any other tool.[10] In the latter case, we see that the hybrid approach on Solidity consistently discovers a particular contract bug[11] while it is never discovered by the `text-mutation` strategy. Yet notably, during the process of running long term

[10] Commit 3b977b in `github.com/ethereum/fe`.

[11] Report ommitted for anonymity

campaigns (Section 5), this same bug *was* eventually discovered by the pure `text-mutation` approach after some period longer than 24 hours. As another strong qualitative measure, our `splice-mutation` approach *exclusively* found a majority of 14 unique bugs out of a total 25 unique bugs across all approaches and trials. In these cases, evidence points to hybrid approaches leading to rapid and exclusive discoveries of unique bugs arising from the combination of simple and complex mutations on the input space.

Additionally, while finding only 1 additional bug or 1.5 additional bugs in some cases may, at first, seem like a modest gain, it is important to recall that in fuzzing, finding additional bugs is *extremely hard*. Böhme and Falk [5] show that "[W]ith twice the machines, we can find *all known* bugs in half the time. Yet, finding linearly *more* bugs in the same time requires exponentially more machines" [5]. That is, finding even one more bug may be very costly. Moreover, as our real-world results in Section 5 show, over time the ability to detect additional bugs adds up to a substantial number of new bugs detected, even for a compiler being aggressively fuzzed by numerous other techniques, using more computing power.

Properties of mutated inputs. We also examined the distinct number of *compiling* inputs in the final queue (the set of all interesting, non-crashing, inputs AFL produced). We investigated this because the number of paths found (cf. **Avg Paths**, Table 3) is somewhat uninformative for compilers, wherein general paths that expose peculiar parse errors are less interesting for testing the compiler's internals than paths involving different behavior in the stages of compilation after parsing. The most serious compiler bugs, with potential to produce *wrong code*, break invariants in the later stages of compilation, especially during complex optimizations. We therefore looked at the number and percentage of interesting inputs (interesting because of new coverage of some kind) that *actually compiled* (**Compiles** in Table 3) as a rough approximation of how much behavior triggering deeper stages of compilation the fuzzing strategies found. On average, our approaches tend to do better either in the absolute number of compiling bugs (K) or percentage of compiling bugs (%) when the values of other approaches are steady. For example, our approaches find thousands more such inputs in Solidity, while the percentage of compiling bugs of the total set varies little compared to `AFL++`. On the other hand, looking at the best-performing bug finder on Fe, `splice-mutation` generates a similar number of compilable programs to other tools, but with a slightly higher ratio. It is also interesting to take this number into consideration with the number of executions, particularly for the `splice-mutation` strategy. The relationship suggests that the "fast or smart" tradeoff surfaces in terms of compilable programs during fuzzing, i.e., chosen strategies may boost the absolute number or percentage of compiling programs and yield more bugs or higher quality bugs.

Project	Length	Total	✓	◷	–
Solidity	20mo 30d	71	69	2	9
Move	20d	14	12	2	0
Fe	9mo 6d	49	42	7	6
Zig	7d	2	1	1	0
All		136	123	13	15

Table 4. 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). **Total** is the number of true, unique bugs reported and acknowledged by maintainers (✓ Fixed + ◷ Unfixed bugs).

5 Non-Experimental Fuzzing Campaigns: Bugs Reported

In addition to our controlled experiments fuzzing a version of each compiler before we reported any bugs, we ran real fuzzing campaigns, updating the compiler versions as new commits were made, and adjusting our corpus to include new tests, of various durations, on each compiler. Two of these campaigns are ongoing (for Solidity and Fe) and have been well-supported and well-received by the compiler teams.

Perhaps the most important evidence of the real world effectiveness is that we fuzzed the Solidity compiler for over a year with our approach, and in that time reported a large number of otherwise unreported bugs that have been fixed; we submitted our most recent bug (as of this writing) on 11/2/21. Prior to and during our campaign, Solidity had been fuzzed heavily using AFL, using a dictionary, by the developers and by external contributors, and has been on OSS-Fuzz since the first quarter of 2019 (https://blog.soliditylang.org/2021/02/10/an-introduction-to-soliditys-fuzz-testing-approach/). While no grammar-based approach has been applied to Solidity as a whole, the Yul IL has been fuzzed using Google's libprotobug-mutator library (https://github.com/google/libprotobuf-mutator). Despite competing with these efforts, 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 not otherwise discovered. Because Solidity bug triage is very well supported, we can add an additional measure of the effectiveness of our approach in finding bugs that involve "realistic" code. After October of 2020, the Solidity team began adding a "should compile without error" label to submitted bugs that involved legal code the compiler rejects. Of the 38 bugs we submitted since that date, 15 (nearly 40%) have involved correct but rejected (via a crash) code. Such bugs are inherently harder to find and usually more interesting than those where a compiler crashes rather than reporting an error when given invalid code.

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. Fe developers received this effort warmly, and quickly made some changes to the Fe compiler to make AFL fuzzing more effective (by crashing when Fe caused the Yul backend to fail). Using our approach, we were able to provide the project with high-quality fuzzing very early in the lifetime of an experimental compiler project. We speculate that better "no-fuss" fuzzing could expose language corner cases early in the implementation of a compiler, avoiding having to make costly changes later, when more code depends on erroneous implementation assumptions or (even more disastrously) poor language design choices. Some of our bug reports (omitted for blinding) triggered lengthy discussions in the issue of a language or compiler design foundational decision. Due to the small size of the Fe team, 8 of our reported bugs have not been analyzed yet and confirmed as valid. The most recent of these was submitted (as of this writing) on 9/25/21. The Fe effort was sufficiently influential that it was invoked in discussions of the long-term strategies for building language-customized fuzzing for Fe.

At the time we fuzzed Facebook's `Move` compiler, the project had been fuzzing various components, but less so the compiler itself. The majority of bugs reported in the `Move` compiler were quickly confirmed and fixed, and developers expressed interest in incorporating our approach into their fuzzing CI.[12]

We ran a shorter, less-intensive campaign on the `Zig` compiler. The `Zig` compiler continues to be under heavy development, and a small team of maintainers are prioritizing efforts to rewrite the components where we found bugs.

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 [7].

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. [20], 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" [20]. 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*, though in practice the change is fairly small (also true of our approach when comby pre-processess a corpus).

7 Conclusions

Using automated methods to find bugs is an important part of modern compiler development. For a variety of reasons, the use of off-the-shelf fuzzers, especially AFL, is the most widely used such appproach. Mutation-based fuzzers, however, were originally used to test binary formats, and their heritage limits their effectiveness for fuzzing compilers. Most solutions to this problem either place significant burden on compiler developers, or are ineffective, or both. We show that using ideas from mutation testing it is possible to significantly improve AFL-based fuzzing of compilers without forcing developers to provide grammars or a dictionary. One of our two configurations performed best for all compilers we investigated, in experiments, sometimes dramatically so (finding more than twice as many bugs on average as the best version of un-augmented AFL). Using our approach we reported more than 100 previously unknown bugs, that have been fixed as a result, in important compiler projects.

[12] Link to public dialog omitted for anonymity.

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] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Softw.* 38, 3 (2021), 79–86.

[5] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 713–724. https://doi.org/10.1145/3368089.3409729

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

[7] 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.

[8] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[9] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*. 197–208. https://doi.org/10.1145/2499370.2462173

[10] 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.

[11] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. 2017. Mutation Reduction Strategies Considered Harmful. *IEEE Trans. Reliab.* 66, 3 (2017), 854–874. https://doi.org/10.1109/TR.2017.2705662

[12] 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

[13] 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.

[14] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[15] 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.

[16] 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

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

[18] 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.

[19] 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

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

[21] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic Crash Bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 612–622. https://doi.org/10.1145/3238147.3238200

[22] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 363–378. https://doi.org/10.1145/3314221.3314589

[23] 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.