GrayC: Greybox Fuzzing of Compilers and Analysers for C

Karine Even-Mendoza*†
karine.even_mendoza@kcl.ac.uk
Department of Informatics, King's College London
London, United Kingdom

Arindam Sharma*
arindam.sharma@imperial.ac.uk
Department of Computing, Imperial College London
London, United Kingdom

Alastair F. Donaldson
alastair.donaldson@imperial.ac.uk
Department of Computing, Imperial College London
London, United Kingdom

Cristian Cadar
c.cadar@imperial.ac.uk
Department of Computing, Imperial College London
London, United Kingdom

ABSTRACT

Fuzzing of compilers and code analysers has led to a large number of bugs being found and fixed in widely-used frameworks such as LLVM, GCC and Frama-C. Most such fuzzing techniques have taken a blackbox approach, with compilers and code analysers starting to become relatively immune to such fuzzers.

We propose a coverage-directed, mutation-based approach for fuzzing C compilers and code analysers, inspired by the success of this type of *greybox fuzzing* in other application domains. The main challenge of applying mutation-based fuzzing in this context is that naive mutations are likely to generate programs that do not compile. Such programs are not useful for finding deep bugs that affect optimisation, analysis, and code generation routines.

We have designed a novel greybox fuzzer for C compilers and analysers by developing a new set of mutations to target common C constructs, and transforming fuzzed programs so that they produce meaningful output, allowing differential testing to be used as a test oracle, and paving the way for fuzzer-generated programs to be integrated into compiler and code analyser regression test suites.

We have implemented our approach in GrayC, a new open-source LibFuzzer-based tool, and present experiments showing that it provides more coverage on the middle- and back-end stages of compilers and analysers compared to other mutation-based approaches, including Clang-Fuzzer, PolyGlot, and a technique similar to LangFuzz.

We have used GrayC to identify 30 confirmed compiler and code analyser bugs: 25 previously unknown bugs (with 22 of them already fixed in response to our reports) and 5 confirmed bugs reported independently shortly before we found them. A further 3 bug reports are under investigation. Apart from the results above, we have contributed 24 simplified versions of coverage-enhancing test cases produced by GrayC to the Clang/LLVM test suite, targeting 78 previously uncovered functions in the LLVM codebase.

*Both authors contributed equally to this research.
†A major part of this work was done as an Imperial College London employee.

ISSTA '23, July 17–21, 2023, Seattle, WA, United States
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598130

CCS CONCEPTS

• **Software and its engineering → Compilers**; *Maintaining software*; **Software testing and debugging**.

KEYWORDS

Greybox fuzzing, compilers, program analysers, code mutators, LibFuzzer, Clang, LLVM, GCC, MSVC, Frama-C

ACM Reference Format:
Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598130

1 INTRODUCTION

Over the last decade or so, randomised compiler testing, often termed *compiler fuzzing*, has seen an explosion of interest, with compiler fuzzers leading to the finding and fixing of thousands of bugs in C compilers such as Clang/LLVM and GCC [65, 82, 105], as well as in compilers for other languages such as OpenCL [67], OpenGL [17], SQL [92] and Verilog [57]. Similar efforts have been proposed for testing code analysers, leading to the discovery of bugs in popular frameworks such as model checkers, static analysers and symbolic executors [16, 61, 63].

During roughly the same period, fuzzing has revolutionised the field of software testing. However, most compiler fuzzers operate very differently from mainstream general-purpose fuzzers, such as AFL [106] and LibFuzzer [75], which are *coverage-directed* and *mutation-based*. Taking inspiration from genetic algorithms, such general-purpose fuzzers synthesise new inputs by mutating existing ones, and use coverage feedback as a fitness function: inputs that yield new coverage of the software under test are prioritised for further mutation. Due to their use of coverage information, these fuzzers are often termed *greybox*. Such fuzzers are equipped with built-in mutation operators that are very simple, involving byte-level transformations such as adding, removing or changing individual bytes. In contrast, most compiler and code analyser fuzzers either generate programs from scratch (e.g. [16, 68, 105]) or transform existing programs (e.g. [17, 65]). In either case, they are *blackbox*: their execution is not guided by information about coverage of the compiler codebase.

The main reason greybox fuzzing is hard to apply effectively to compilers,[1] particularly those for statically-typed languages, is that naive code mutations tend to produce *invalid* programs: programs that either do not conform to the language's syntax or disobey the language's static semantic rules (e.g. calling functions with inappropriately-typed arguments in C). Starting with a valid input that exercises a compiler all the way from lexing to analysis and/or code generation, naive greybox fuzzing (using byte-level mutations) is likely to produce a large stream of invalid programs that are rejected by the compiler's lexer, parser or type checker. Such invalid inputs can help find edge cases where the compiler crashes instead of gracefully rejecting a malformed program, but cannot find deeper errors in the compiler's middle- and back-ends, where the vast majority of optimisations are performed (the middle-end being responsible for platform-independent optimisations and the back-end for code generation and optimisations specific to the target architecture).

In contrast, blackbox grammar-based compiler fuzzers can be designed to emit valid programs by construction, allowing them to detect middle- and back-end bugs: crashes, or (when used in conjunction with a pseudo-oracle such as differential testing [78]) miscompilations (where the compiler emits incorrect object code). But despite these appealing properties, blackbox compiler fuzzers are prone to problems of *immunity*: once they have enabled the finding and fixing of a substantial number of bugs in a compiler, they tend to be unable to generate programs that trigger further bugs [86]. Lacking feedback, the fuzzers have no way of adapting their generation strategy to find more bugs.

This leads to an interesting research challenge which we address in this paper: how to devise greybox compiler fuzzing techniques that yield valid programs capable of detecting deep compiler bugs, and that can enhance the regression test suites of mature compilers.

Mutation-based approaches have been very successful in the context of dynamic languages such as JavaScript: LangFuzz [59] is a pioneering work in this space which found critical bugs in JavaScript and PHP interpreters, and more recent efforts, such as Superion [103] for JavaScript and XML and Nautilus [1] for JavaScript, Lua, PHP and Ruby, have added coverage-guidance. However, code mutations are less likely to result in invalid programs for dynamic languages, and front-end bugs are often equally valuable in the context of web security.

For statically-typed languages like C, preliminary steps towards mutations that have some chance of preserving static validity include the use of keyword dictionaries [52, 75], protobuf descriptions of programming language structure [96], and regular expressions and partial grammars for recognising common programming language-like features [53, 55, 101]. However, such methods still produce a high rate of invalid programs. For example, the LLVM project's Clang-Proto-Fuzzer tool, which relies on a protobuf description of a fragment of C/C++, was abandoned because it only found obscure front-end crash bugs that developers were reluctant to fix [94]; a presentation on the work reports *"Bugs are being fixed too slow (if at all)"* [96]. Indeed, we reported several front-end crash bugs triggered by invalid programs produced via naive mutation methods, and found they were not received positively by developers, either being closed as "won't fix", or ignored (see §5.4). A recent tool, PolyGlot [11], for generic language processor testing pays special attention to improving the likelihood that the test programs it creates are valid, yet achieves only limited coverage on the middle- and back-end compiler components, restricting its ability to find bugs in C compilers mainly to front-end crashes (see §5 for more details).

Our contribution. In an attempt to get the best of both worlds—the validity guarantees associated with grammar-based blackbox compiler fuzzing and the targeted search offered by a greybox approach—we present GrayC,[2] a greybox fuzzer for C compilers. The key innovation of GrayC is the use of *semantics-aware* mutation operators: mutation operators that preserve validity of the input program (including the typing rules of the language) with high probability.[3] These mutations work at the abstract syntax tree (AST) level, and include mutations that modify individual programs, as well as mutations that combine elements of multiple programs. The programs generated via semantics-aware mutation exercise the compiler codebase end-to-end, and can be used to find crashes deep in optimisation passes.

Rather than directly applying coverage-directed fuzzing to each compiler of interest, GrayC takes a "fuzzing by proxy" approach, akin to that taken in recent work on fuzzing instruction set simulators [56] and deployed CPUs [97]. We run coverage-directed fuzzing with GrayC's semantics-aware mutators on a particular compiler under test (compiled with suitable coverage instrumentation), collecting all the test programs that are generated during the fuzzing process. We then feed this *output corpus* to a range of different compilers under test, operating at various optimisation levels, to see whether they induce compiler crashes. This workflow, summarised in Figure 1, has the advantage that only the compiler used for generation of the output corpus needs to be compiled in a manner suitable for greybox fuzzing. The compilers and analysers subsequently tested using the output corpus can be arbitrary binaries, allowing closed-source compilers (e.g. MSVC) and tools not written in C/C++, to be tested (e.g. the Frama-C analyser [15], one of the experimental subjects in this paper, is written in OCaml).

Overview of results. We have used GrayC (at various stages of development) to test the Clang, GCC and MSVC compilers and the Frama-C code analyser. This led to us finding 30 confirmed bugs: 25 previously unknown compiler and analyser bugs, out of which 22 have already been fixed in response to our reports and a further 5 bugs that turned out to have already been reported by other users.[4] Importantly, of these 30 bugs, 22 are middle- or back-end bugs that can only be triggered by *valid* programs. It is due to a very high percentage of the programs that GrayC generates being valid that our technique was able to find these bugs; this is in contrast to other techniques that apply mutation-based fuzzing to C compilers.

[1]For succinctness, we will use the term *compilers* to refer to both compilers and code analysers, unless we make the distinction explicit.

[2]Pronounced "Grace", GrayC is a pun on greybox fuzzing for C, at the same time paying homage to compiler pioneer Grace Hopper.

[3]As discussed further in §3.1, there are strong practical reasons for tolerating a suitably low rate of invalid programs.

[4]Our reports of a further 3 bugs found by GrayC are waiting investigation.

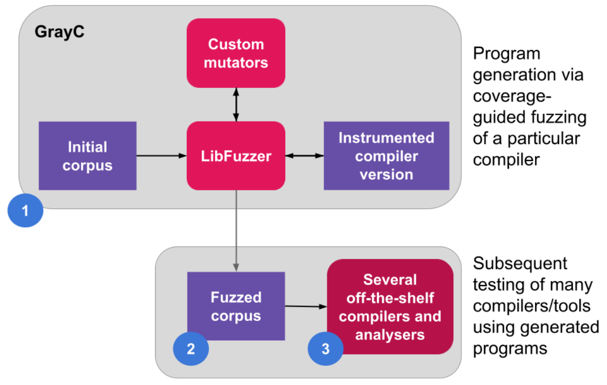
GrayC
Custom mutators
Initial corpus
LibFuzzer
Instrumented compiler version
Program generation via coverage-guided fuzzing of a particular compiler
1
Fuzzed corpus
Several off-the-shelf compilers and analysers
2
3
Subsequent testing of many compilers/tools using generated programs

Figure 1: Overview of greybox fuzzing with GrayC.

In parallel, we also performed extensive testing using the state-of-the-art blackbox fuzzer CSmith [105], and were unable to find *any* of the bugs that GrayC could find. This provides evidence that greybox compiler fuzzing has the potential to find bugs in compilers that have already been subjected to extensive blackbox fuzzing.

We also present a set of controlled experiments comparing the semantics-aware mutators of GrayC with naive byte-level mutation (via Clang-Fuzzer [13]), grammar-based fuzzing (via Grammarinator [58]), fragment-based fuzzing (via a tool similar to LangFuzz [59]), regular expression-based mutation (via Universal Mutator [53] and afl-compiler-fuzzer [55]), and a greybox approach for generic language processor testing (via PolyGlot [11]). Our results show that GrayC provides better coverage of middle- and back-end compiler components, and is able to find crashes in these components that are not found when the other methods are used, since they tend to generate invalid programs.

Finally, we have demonstrated how GrayC can have impact beyond just finding bugs by using it as the basis for contributing new tests to the LLVM test suite. By combining GrayC with an off-the-shelf test case reducer, and designing a novel tool, enhanCer, to equip reduced programs with a test oracle, we produced a set of small, well-defined programs that achieve coverage of particular LLVM optimisations that is not achieved by the LLVM test suite. We contributed these test cases back to the LLVM project to improve the coverage of regression testing, and the developers reviewed and accepted the test cases.

In summary, our main contributions are:

(1) A technique for coverage-directed mutation-based greybox compiler fuzzing that yields valid programs thanks to *semantics-aware mutators* specially designed for static languages;
(2) The implementation of this idea in a greybox compiler fuzzer, GrayC, which uses fuzzing by proxy to generate C programs that can be used as inputs to a range of compilers and analysers under test;
(3) A large testing campaign and experimental evaluation showing that GrayC finds more bugs and achieves higher coverage than other mutation-based approaches, and can generate programs that enhance the regression test suites of mature compilers.

2 BACKGROUND

This section provides an overview of the main concepts (§2.1) and tools (§2.2) necessary to understand our technique

2.1 Compiler Bugs and Program Validity

Our primary focus in this work is on *crash* bugs, where the compiler aborts unexpectedly. Specifically, we are interested in finding crashes deep in a compiler's codebase (e.g. in the optimiser or code generator). For this purpose, we distinguish between statically-valid and statically-invalid programs. Essentially, *statically-valid* programs are those that should be expected to compile according to the language specification, without reference to any particular compiler. Therefore, statically-valid programs are more likely to exercise deep parts of the compiler than statically-invalid programs.

We say that a program is *dynamically-valid* if it produces a well-defined deterministic result when executed. In particular, it must not trigger undefined behaviour (such as an out-of-bounds access, or division by zero) at runtime. Dynamically-valid programs can be used to find miscompilations via differential testing and enhance compiler test suites.

The focus of GrayC is on producing statically-valid programs but, as discussed in §3.2, sanitizer tools can be used to filter programs that are dynamically-invalid; we have used this to restrict attention to programs that might be useful for test suite augmentation (see §4.3).

2.2 LibFuzzer and Clang-Fuzzer

LibFuzzer [66] is a greybox in-process mutation-based fuzzing engine. It treats test cases as sequences of bytes, and the user must write a *fuzz target* function that uses a given byte sequence to invoke their system under test (SUT) in a meaningful way. LibFuzzer is fully integrated with the LLVM [64] infrastructure; using it requires using a special compilation flag.

Starting from a user-provided initial corpus, LibFuzzer produces new tests by mutating existing ones. By default, this is achieved using a set of byte-level mutations. If a mutated test results in new coverage, it is fed back into the corpus for future mutation. This process runs iteratively while the engine keeps track of any tests that cause the SUT to crash.

LibFuzzer provides an API that allows a *custom mutator* to be provided: a function that accepts an existing input as a sequence of bytes, and returns a mutated version of the input. The function can use domain-specific logic to interpret the input sequence of bytes according to the application domain of the system under test, and thus perform a semantically-meaningful mutation.

Clang-Fuzzer [13] allows fuzzing of the Clang compiler using LibFuzzer, by providing a fuzz target that interprets a sequence of bytes as text and feeds this text to Clang. Clang-Fuzzer uses LibFuzzer's built-in byte-level mutations, so the mutated programs that it generates are very unlikely to be statically-valid C/C++ programs. As described in detail in §3, our GrayC tool augments the Clang-Fuzzer fuzz target with a custom mutator that parses an input into an AST and performs semantics-aware, AST-level mutations, returning the mutated program as a string. This leads to a high rate of statically-and dynamically-valid programs.

3 GRAYC

The GrayC approach involves using mutation-based fuzzing as a program generation technique, and then using the generated programs to test compilers and analysers.

The high-level flow of GrayC is sketched in Figure 1. Starting with an initial corpus of valid test programs, GrayC uses LibFuzzer to perform coverage-guided mutation-based fuzzing (① in Figure 1). The fuzz target of Clang-Fuzzer is used to exercise the Clang/LLVM codebase, and our semantics-aware mutators are provided as LibFuzzer custom mutators, to maximise the chance that mutated programs are statically-valid. Unconventionally, the purpose of this stage is not to find bugs, but rather to generate a large corpus of diverse test programs, which are saved to an external directory, called the *fuzzed corpus* (②). The programs in the fuzzed corpus can then be used for deep testing of a range of off-the-shelf compilers (at various optimisation levels) and code analysers (③), which do not need to be compiled in a special manner; in fact they may be closed-source (this allowed us to find bugs in a proprietary compiler from Microsoft, see §4). The idea of this "fuzzing by proxy" approach is that coverage-guided fuzzing on a particular compiler of interest should lead to programs that are interesting and diverse, and thus useful for testing C compilers and analysis tools in general. This is supported by the bugs we have found using GrayC, affecting a range of targets (§4).

We first discuss the custom mutators employed by GrayC, whose key objective is to produce statically-valid programs (§3.1). We then describe our enhanCer tool that allows GrayC to be used for differential testing and compiler test suite augmentation (§3.2), and describe pertinent implementation details (§3.3).

3.1 Custom Mutators

Our custom mutators are *semantics-aware*, which enables them to generate statically-valid programs. GrayC receives—from LibFuzzer—a program to transform. It parses the program into an AST, and then selects, uniformly at random, a transformation and an appropriate AST node at which to apply the transformation. GrayC is based on the Clang LibTooling framework [12], which facilitates type-aware mutations by giving access to a fine-grained, typed AST for the program being mutated. This allows GrayC's custom mutators to have additional checks based on, but not limited to, types, variable names and scopes.

The transformations are summarised in Table 1, and are categorised into *mutations*, which take individual programs as input, and *recombiners*, which work on two programs, the second program selected from the corpus uniformly at random.

Mutators (lines 1–11 in Table 1). A mutator takes as input a program and transforms it based on a certain template. GrayC's mutators can add new statements, as well as edit or delete expressions and statements. For instance, Inject-Control-Flow adds a `break`, `continue` or `return` statement, Replace-By-Constant replaces an arithmetic expression by a constant (e.g. replacing `a=(a+1)%7;` with `a=6;`) and Change-Type changes the type of an expression (via explicit casting).

Using two examples, we illustrate how Delete-Statement works in isolation, and together with Duplicate-Statement.

Example 1. Consider this simple example:

```
for (int i=0; i<5; i++) {
  i+=2; printf("itr: \%d", i);
}
```

The Delete-Statement mutator acting on the for-loop block can either remove a statement:

```
for (int i=0; i<5; i++) {
  printf("itr: \%d", i);
}
```

or replace a block with the empty statement (via two consecutive applications):

```
for (int i=0; i<5; i++) {
  ;
}
```

Example 2. GrayC applies a series of mutators to the original program on the left (a program from the Clang/LLVM test suite) to synthesise the program on the right:

```
 1 typedef struct {
 2   unsigned w[3];
 3 } Y;
 4 Y arr[32];
 5 int main() {
 6   int i=0;
 7   unsigned x=0;
 8   for (i=0; i<32; ++i)
 9     arr[i].w[1]=i == 1;
10   for (i=0; i<32; ++i)
11     x+=arr[1].w[1];
12   if (x!=32)
13     abort();
14   return 0;
15 }
```

```
 1 typedef struct {
 2   unsigned w[3];
 3 } Y;
 4 Y arr[32];
 5 int main() {
 6   int i = 0;
 7   unsigned x = 0;
 8   for(i=0; i<32; ++i)
 9   for(i=0; i<32; ++i)
10     x+=arr[1].w[1];
11    x+=arr[1].w[1];
12   if (x!=32)
13     abort ();
14   return 0;
15 }
```

To do so, GrayC invokes: (i) Delete-Statement, to remove the inner statement of the first loop (in blue: left-program, line 9), and (ii) Duplicate-Statement, to duplicate the inner statement of the second loop (in green: left-program, line 11 to right-program, lines 10–11). The two separate loops in the original program have now converted to a nested loop in the fuzzed program due to the deletion of line 9 via two different Delete-Statement mutations: replacing the inner statement with the empty statement, and then also removing the empty statement. The Duplicate-Statement mutation can occur before, in-between or after the two Delete-Statement mutations.

Recombiners (lines 12–13 in Table 1). A recombiner takes as input two programs—a source program and a destination program—and transforms the destination program by adding parts of the source program. To allow for increased code diversity, the source programs can be picked from a larger set compared to the original corpus provided to LibFuzzer. GrayC's recombiners can then replace the body of a function with the body of another function from a different program, or combine the bodies of two functions from two different programs. We use a careful renaming scheme to work around name clashes between variables and functions in the source and destination programs.

Example 3. We illustrate how Combine-Functions recombines the following two programs: P_{blue} (the destination program) and P_{green} (the source program).

Table 1: GrayC's code mutators and recombiners.

#	Type	Construct	Short Name	Description
1			Duplicate-Statement	Duplicate a statement within the same block excluding variable declarations.
2	Mutator	Statement	Delete-Statement	Delete a non-declaration statement; randomly decide whether to keep the semicolon.
3			Inject-Control-Flow	Add a `break`, `continue` or `return` statement inside a loop. The statement is guarded by a condition based on an auxiliary loop counter so that it is only invoked on certain iterations.
4			Delete-Expression	Delete sub-expressions from a given expression in a corpus program.
5			Expand-Expression	Expand sub-expression with other sub-expressions from the corpus program; e.g. in an assignment or loop condition.
6			Replace-By-Constant	Replace an expression with a random valid constant expression of the same data type; e.g. replace a condition in a `while` to `0`, making its body dead code.
7	Mutator	Expression	Flip-Bit	Flip a bit in a constant expression.
8			Replace-Digit	Similar to Flip-Bit but on the number's decimal representation: either flip the sign or change a single digit.
9			Change-Type	Change the type of an expression (`short`, `long`, `unsigned`, `float`, etc.).
10			Replace-Unary-Operator	Replace unary operator with an assignment using the same variable; e.g. replace `i++` in a `for` statement to `i=i+2` or `i=i-3`.
11			Flip-Operator	Replace one operator with another (arithmetic operators).
12	Recombiner	Function	Replace-Function-Body	Replace the body of a function with that of another function with the same number of arguments.
13			Combine-Functions	Combine the body of a function with another function with the same number of arguments, either by concatenating bodies or interleaving their statements.

Program P_{blue}:

```
1 int dest_func(int x_dest
      , int y_dest) {
2   int b_dest=x_dest*y_dest;
3   b_dest=b_dest+5;
4   return b_dest;
5 }
6 int main() {
7   int ret=dest_func(6,7);
8   return ret;
9 }
```

+

Program P_{green}:

```
1 int a=0;
2 int source_func(int
      j_src, int k_src) {
3   int m_src=j_src+k_src;
4   return m_src;
5 }
6 int main() {
7   int ret=source_func(2,3);
8   return a;
9 }
```

The recombiner merges the body of `source_func` in P_{green} into the body of `dest_func` in P_{blue}. There are several options to merge the bodies of these functions. The programs P_1 and P_2 below are two of the possible programs that Combine-Functions could output. We mark the lines used in the output programs in blue if they originate from P_{blue}, and in green if they originate from P_{green}.

Output program P_1:

```
1  int dest_func(int x_dest
       , int y_dest) {
2    int j_src=x_dest;
3    int k_src=y_dest;
4    int m_src=j_src+k_src;
5    int b_dest=x_dest+y_dest;
6    b_dest=b_dest+5;
7    return b_dest;
8  }
9  int main() {
10   int ret=dest_func(6,7);
11   return ret;
12 }
```

Output program P_2:

```
1  int dest_func(int x_dest
       , int y_dest) {
2    int j_src=x_dest;
3    int k_src=y_dest;
4    int m_src=j_src+k_src;
5    int b_dest=x_dest+y_dest;
6    b_dest=b_dest+5;
7    return m_src;
8  }
9  int main() {
10   int ret=dest_func(6,7);
11   return ret;
12 }
```

Combine-Functions combines functions with the same number of arguments, and the first thing it does is to initialise the variables corresponding to the function arguments of the source function with the values of the arguments in the destination function (lines 2–4 in P_1 and P_2). The return statement is handled separately: Combine-Functions randomly selects one of the two return values (P_1 uses the return statement from P_{blue}, while P_2 that from P_{green}) and adds it as a single return statement of the merged function.

Aggressiveness. Recall from §2.1 that we make a distinction between statically- and dynamically-valid programs. The main objective of GrayC is to generate programs that, with high probability, turn out to be statically-valid. Such programs are suitable for finding compiler crash bugs, which is our main use case for GrayC. As discussed further in §3.2, dynamic analysis tools can be used to filter out programs that they observe to be dynamically-invalid, such that (modulo limitations of available dynamic analysers) the programs that pass this filtering step can be used to augment compiler regression test suites, or used for differential testing to search for miscompilation bugs.

We experimented with adding a special *conservative* mode to GrayC that applies mutations less aggressively with the aim of generating dynamically-valid programs with higher probability. In this mode, certain mutators behave in a more restricted fashion. For example, with respect to Table 1, Replace-By-Constant adds checks to avoid undefined behaviour based on the constant's location, e.g. the replaced constant should be non-negative if used as an array index, while Delete-Expression avoids selecting expressions using pointers, to reduce the chances of memory-related undefined behaviour. Because the Csmith program generator is designed to produce code that is free from undefined behaviour [105], we added another mutator in *conservative* mode, Add-Csmith-Block. This uses Csmith (configured to limit expression complexity and use no global variables, user-defined types or memory allocations) to generate a program with a single function, and pulls a block from the function into a corpus program.

Unfortunately, despite these efforts, we concluded—based on extensive experimental evaluation—that the *conservative* mode of GrayC did not pay off. We found it to be inferior to GrayC's standard *aggressive* mode both with respect to bug-finding ability and compiler code coverage. Furthermore, the overhead of performing additional checks and invoking Csmith made GrayC operate more slowly in this mode, so that although the chances of each generated program being dynamically-valid were higher, the slower rate of program generation overall meant that running GrayC in its *aggressive* mode and using sanitizers to filter out dynamically-invalid programs proved to be a more economical source of dynamically-valid programs.

3.2 enhanCer

To make the GrayC-generated programs suitable for differential testing and compiler test suite augmentation, we designed a new post-processing tool, enhanCer, that transforms these programs to produce a single output. That way, the output of the enhanced program can easily be compared during differential testing or added to an expected output file.

Inspired by the way Csmith [105] programs are designed, the single output hashes all the global variables in the program.[5] In addition, enhanCer: (1) adds to the global hash value all the strings printed by the program during execution, and (2) replaces any termination function, such as `abort` and `exit`, by an operation that adds to the global hash a unique string representing the termination function, and then replaces the operation by a `return` statement. The reason for which we eliminate termination functions is to ensure that the global hash is always printed at the end of a program execution. (Note that Csmith programs never contain calls to such functions by design, but in our case we start from existing programs that might contain them.) Finally, enhanCer includes other transformations, such as ensuring that the signature of `main` is always `int main(void)`.

Recall that programs produced by GrayC are not guaranteed to be dynamically-valid. Furthermore, it is possible that eliminating termination functions might introduce undefined behaviour to programs that were previously dynamically-valid. To guard against this, after transforming a given program, enhanCer invokes sanitizers to detect undefined behaviour. Programs that turn out to be dynamically-invalid are then discarded, so that they do not confound differential testing or lead to the possibility of programs that exhibit undefined behaviour being added to the set of end-to-end tests in a compiler's regression test suite.

3.3 Implementation Details

Our implementation is divided into several parts: GrayC, enhanCer, and a set of Bash and Python scripts for crash and differential testing. We make use of LLVM 12.0.1, with our mutators implemented on top of Clang-Fuzzer/LibTooling.

To detect undefined behaviour, enhanCer invokes Frama-C [15], an open-source industrial-strength framework dedicated to the formal analysis of C programs, and the Clang/LLVM compiler sanitizers: AddressSanitizer [95], a dynamic analysis tool to detect invalid memory accesses, MemorySanitizer [98] to detect uninitialised memory accesses, and UndefinedBehaviorSanitizer [99] to detect a wide range of undefined behaviours.

[5]If the program already produces a single output, enhanCer makes no changes.

4 USING GRAYC IN THE WILD

We divide our evaluation into two parts: a long-term fuzzing campaign used to find compiler and analyser bugs (presented in this section) and a series of controlled experiments designed to better understand the strengths and weaknesses of GrayC compared to other approaches (presented in §5).

During the development of GrayC, we applied it to several versions of a number of compilers and analysers. Our fuzzing campaigns (§4.1) led to the discovery of 30 confirmed bugs (§4.2), with another 3 bug reports still under investigation, and the contribution of 24 programs to the Clang/LLVM test suite (§4.3).

In parallel, we applied Csmith [105] continuously over a period of six months to look for bugs in GCC 11 and Clang-13 on x86_64. It did not find any bugs, adding weight to our hypothesis that compilers eventually become immune to blackbox fuzzing approaches [86].

Our artifact [18] contains data associated with these experiments, including links to bug reports and relevant logs.

4.1 Experimental Setup

We now summarise how we approached our open-ended fuzzing campaigns, with the aim of finding previously-unknown bugs in compilers and analysers.

Initial Corpus. GrayC's initial corpus was a collection of single-file programs from various sources: automatically-generated programs, compiler test suites, and C tutorials. In addition, we used Csmith to create a set of automatically-generated programs. We minimised the set of Csmith programs using C-Reduce [87] to have at least one reduced and dynamically-valid program covering each function in the fuzzed compiler that was covered by the original set of programs.

Fuzzing Campaigns. We applied our tool throughout its development, running it occasionally during 2021 and 2022. Overall, we estimate that we ran GrayC on various compiler versions for a total of several weeks. Each fuzzing campaign ran until it reached a time or disk space limit, or no new coverage was achieved for some time; as this was a series of long-running experiments, the details of these limits varied. Similarly to Clang-Fuzzer, GrayC terminates the fuzzing process when the mutation attempt fails 100 times.

Compilers and Analysers Tested. We tested recent versions of LLVM (`10,11,12,13,14` and 15), GCC (`10,11,12` and 13) and the code analyser Frama-C (`21,22,23` and 24) on Ubuntu Linux 18.04 LTS x86_64. We found bugs in GCC and LLVM on Linux by compiling each mutated program with and without sanitizer flags and using each of the standard `-O0`, `-O1`, `-O2`, `-O3`, and `-Os` optimisation levels. We also conducted a short evaluation on Windows with a small set of mutated programs generated on Linux to test the Microsoft Visual Studio Compiler (MSVC `19.28.29915`) with the `/Od` (no optimisations) and `/O2` (maximise speed) optimisation settings.

During our fuzzing campaigns, we used C-Reduce, the LLVM sanitizers and Frama-C as part of investigating the bugs that we found. This led to us to report 11 additional bugs in these tools as a by-product of our work [2–6, 24–26, 70, 72, 73].

Table 2: Compiler and code analyser bugs found by GrayC.

	Previously-unknown		Independently-reported	
	Confirmed	Fixed	Confirmed	Fixed
GCC	9	8	3	3
LLVM	2	2	1	0
MSVC	3	1	0	0
Frama-C	11	11	1	1
TOTAL	25	22	5	4

Table 3: Number of confirmed compiler and code analyser bugs found by GrayC in each high-level component.

	Front-end	Middle-/Back-end
GCC	2	10
LLVM	1	2
MSVC	3	0
Frama-C	2	10
TOTAL	8	22

4.2 Bugs Found

Table 2 gives an overview of the compiler and code analyser discovered. GrayC found 30 confirmed bugs [19–23, 27–33, 35–46, 69, 74, 79–81, 88]: 25 previously unknown (out of which 22 bugs have already been fixed in response to our reports), and 5 bugs confirmed and/or fixed independently shortly before we found them. Additionally, 3 bug reports (not included in Table 2) are pending investigation [89–91].

Table 3 classifies these bugs into those occurring in the front-end and those occurring in the middle- or back-end. Most of the bugs found by GrayC are in the middle- or back-end, demonstrating its ability to find deep bugs. The front-end bugs could in principle be found using more naive mutation approaches. However, the fact that GrayC generates statically-valid programs means that these bugs are taken seriously and fixed by developers; by contrast, front-end bugs triggered by statically-invalid programs are often left unaddressed by developers (see §5.4).

All the bugs we found are crash bugs, except for two which cause the compiler under test to hang. In particular, our use of GrayC plus enhanCer did not lead to us finding any miscompilation bugs using differential testing. However, we were successful in using GrayC plus enhanCer to generate coverage-enhancing test cases that have been accepted into the LLVM test suite (see §4.3).

To give a flavour of the kind of bug reports produced by GrayC, we now discuss one of them.

ICE (Internal Compiler Error) in GCC during constant folding optimisation. The following program fuzzed by GrayC led to an ICE in GCC 11 and GCC 12:

```
struct a d;
struct a {
  int b;
  int c[]
} main() {
  d.c[268435456] || d.c[1];
}
```

This program was obtained using Expand-Expression, which replaced `d.c[1]` with `d.c[1] || d.c[1]`, and then using Replace-By-Constant, which modified `d.c[1] || d.c[1]` to `d.c[268435456] || d.c[1]`. During constant folding (middle-end), the decomposition of `d.c[268435456]` triggered the bug; this was fixed by adding extra checks.

4.3 Compiler Test Case Contributions

We used GrayC's ability to generate dynamically-valid programs, with the help of enhanCer, to improve the LLVM test suite. In particular, we contributed test programs generated by GrayC which increase the function coverage achieved by the LLVM test suite.

Once we identified such programs, we transformed and reduced them further using enhanCer and C-Reduce and manually cleaned them up. So far, 16 of these programs were accepted into the LLVM test suite [83, 84][6] and 8 of these programs are under review [85]. These tests targeted 78 previously uncovered functions in `Transforms`, `IR`, `AST` and other parts of `clang lib`. All contributed test cases are available at [18, 49, 60].

5 CONTROLLED EXPERIMENTS

We next compare GrayC with other fuzzing methods, using controlled experiments.

5.1 Experimental Setup

Tools. We consider the following tools in our evaluation:

(1) **GrayC.** The tool introduced in this paper.[7]
(2) **GrayC-No-Cov-Guidance.** Fuzzing with no coverage guidance to assess a main claim of the paper, which is that coverage guidance is useful. We adapted GrayC to work without coverage guidance but with all its available mutators.
(3) **GrayC-Fragments-Fuzzing.** We adapted GrayC to run without coverage guidance, and only use code fragment injection (namely the Add-Csmith-Block mutator described in §3.1). This is the closest we can get to what LangFuzz[8] does: it is not coverage-guided, and only uses code fragment injection mutation [59].
(4) **Clang-Fuzzer.** Default Clang-Fuzzer [13, 66] (see §2).
(5) **Csmith.** Default Csmith [105] (see §1).
(6) **Grammarinator.** Default Grammarinator (v19.3) [50, 58]: a general purpose grammar-based open-source fuzzer.
(7) **PolyGlot.** The tool is taken from the artifact associated with the paper [11]: PolyGlot is a general-purpose AFL-based fuzzer that aims to generate statically-valid programs via a semantic error-fixing mechanism.

[6]Ten of the contributed tests were generated during the very early stages of the project, using prototype tools that predated GrayC and enhanCer in their current forms.
[7]This refers to GrayC running its standard *aggressive* mode. We also performed our full set of controlled experiments using the *conservative* mode described in §3.1. As discussed in §3.1, these experiments revealed that the *conservative* mode performed substantially worse than the *aggressive* mode. Due to the large number of other approaches we compare with experimentally, we omit results for the *conservative* mode in this section, to simplify figures and associated discussion. However, our artifact includes the means to reproduce the *conservative* mode experiments in addition to the other experiments we describe.
[8]LangFuzz was applied to JavaScript and PHP interpreters, and the tool is not publicly available for direct comparison.

(8) **RegExpMutator.** A LibFuzzer-based tool that uses Universal Mutator [53], a regexp-based mutator, instead of LibFuzzer's default mutator.

We have also experimented with ideas from a recent paper on "no-fuss compiler fuzzing" [55], by assembling a LibFuzzer-based tool that replaces the default mutators with those of afl-compiler-fuzzer [102]. These mutators are taken from mutation testing and program splicing and also rely on regular expressions. However, we were unsuccessful in generating a meaningful number of non-duplicate programs in our experiments, as the mutators only rarely trigger on our program corpus.

Implementation notes. To avoid coverage guidance, GrayC-No-Cov-Guidance and GrayC-Fragments-Fuzzing are not based on LibFuzzer. Instead, they are based on a simple script that repeatedly picks a program from the working corpus at random, applies the relevant mutators, and writes the mutated program back to the working corpus.

We implemented RegExpMutator by invoking the Universal Mutator tool as an external Python process. This leads to a variety of mutated programs being generated, of which one is chosen at random. We note that this is a rather inefficient way to perform regex-based mutation, and that by re-implementing the logic of the Universal Mutator in C++, it would likely be possible to achieve higher throughput.

Collecting test programs. We used each tool to construct a corpus of test programs for subsequent offline testing and coverage analysis of various compilers and analysers. For the tools that are coverage-guided, this is an example of "fuzzing by proxy" (see §3): the corpus of programs that arise during coverage-guided fuzzing of a particular system under tests is saved, and then used to test a number of different systems under test.

We allocated 24 h per tool for program collection, and to account for variance we repeated the collection process 10 times per tool. This resulted in 10 sets of generated programs per tool. The throughput and coverage results reported in §5.2 and §5.3 are averages over 10 sets.

For the tools that require an initial corpus (all tools except Csmith and Grammarinator), we assembled an initial corpus as described in §4.1. Our corpus snapshot for these controlled experiments contains 1,767 dynamically-valid single-file programs. The reader can consult our artifact for full details on the programs included. For the coverage-guided tools, fuzzing was performed against LLVM 12.0.1.

For Clang-Fuzzer, saving *all* mutated programs proved impractical: Clang-Fuzzer generates approximately one million programs every 24 h, with many duplicate programs having no effect on coverage. We considered filtering duplicates during or after fuzzing, which either reduced the tool efficiency (when spending time detecting duplicates) or led to excessively long post-processing times. As a result, for Clang-Fuzzer we decided to only save the mutated programs for which Clang-Fuzzer reports extra coverage (i.e. Clang-Fuzzer's default settings).

We now discuss our results with respect to throughput and static validity rate (§5.2), coverage (§5.3), and bug finding (§5.4).

Table 4: Average throughput (across 10 repetitions, over 24 h) by each tool and the percentage which are statically-valid. Numbers refer to unique programs after filtering duplicates.

	Programs/h	Statically-valid (%)
Csmith	1,144	99.96%
GrayC	2,906	99.47%
GrayC-Fragments-Fuzzing	4,152	99.08%
PolyGlot	641	91.26%
GrayC-No-Cov-Guidance	4,629	75.04%
RegExpMutator	1,392	19.21%
Clang-Fuzzer	1,183	1.55%
Grammarinator	5,391	0.0%

5.2 Throughput and Static Validity Rate

The key metric when comparing fuzzers is their bug-finding ability—and coverage as a proxy for that. However, it is instructive to interpret data on coverage and bug-finding ability in the context of the throughput achieved by each fuzzer. We present results related to throughput, with a particular emphasis on how it evolves over time and how many statically-valid programs are generated.

Throughput. The second column of Table 4 shows, for each tool, the average throughput over 24 h of fuzzing. GrayC's throughput is somewhere in the middle. The mutation-based black-box fuzzers (Grammarinator, GrayC-No-Cov-Guidance and GrayC-Fragments-Fuzzing) have the highest overall throughput, with Grammarinator on top. At the other end of the spectrum, PolyGlot has the lowest overall throughput, with Clang-Fuzzer second to last (however, recall from §5.1 that we capture only a subset of the programs that Clang-Fuzzer generates, because otherwise its throughput rate would be too high to be manageable). After a closer inspection, we found that PolyGlot and Clang-Fuzzer have the highest throughput in the beginning, but this decreases significantly, falling into the last places by the fourth and the eighth hour of fuzzing, respectively. This declining trend (shared in various degrees by all LibFuzzer-based tools) is mostly due to the corpus reduction functionality in LibFuzzer, which consumes more time as the corpus grows, leaving less time for program mutation.

Static validity rate. The last column of Table 4 shows, for each tool, the percentage of generated programs that are statically-valid. We consider a program to be statically-valid if it is compiled successfully by GCC 11.1.0 while imposing a compilation timeout of 45 s and a stack limit of 4 MB (we impose a stack limit because compiler crashes caused by programs with large stack allocations are usually due to resource exhaustion rather than compiler bugs).

Over 99% of the programs generated by Csmith, GrayC and GrayC-fragments-fuzzing are statically-valid.[9] PolyGlot achieved a high compilation rate of 91.26% with the initial corpus in this evaluation, much higher than originally reported with PolyGlot's initial corpus, which was a mixture of statically valid and invalid programs [11]. GrayC-No-Cov-Guidance's lack of coverage guidance resulted in a significantly lower compilation rate of 75.04%. We suspect this is because without coverage guidance, similar statically-invalid programs that cover the same front-end code do not get

[9] Csmith-generated programs are by construction compilable; however, some files hit our compilation timeout.

de-prioritised. Only 19.21% programs compile for RegExpMutator and only 1.55% for Clang-Fuzzer. None of the Grammarinator programs generated during this evaluation passes compilation.

5.3 Coverage

We measured coverage for GCC 12 on Ubuntu 18.04 LTS x86_64 and LLVM 13 on Ubuntu 20.04 LTS x86_64. We compiled the generated programs with `-O3`, to exercise a large number of optimisations, and we imposed a timeout of 50 s for compiling a program. We used the Gcov-based tool `gfauto` [51] to generate the coverage results in a human-readable format.

We compare GrayC with other mutation-based tools, which all start from an initial corpus. Including Csmith and Grammarinator in this comparison would be unfair, as they are generation-based tools that cannot benefit from the coverage of an initial corpus. Nevertheless, we measured coverage for Csmith and Grammarinator as well, and in both cases the coverage is smaller than the one for our initial corpus, with Grammarinator achieving particularly low coverage (with essentially no coverage in the middle- and back-end, given that all the generated programs are statically-invalid).

Results. Figure 2 (best viewed in colour) shows the line coverage achieved in GCC (left) and LLVM (right) by the mutation-based tools, plotting the mean and standard error over the hourly sampled rate per tool. In addition, we show the coverage achieved by the initial corpus, from which all these tools benefit. (Note that in the beginning, the coverage achieved by all tools is that of the initial corpus.) The hourly sampled rate of coverage per tool and the calculation of the average mean and standard error from the raw data are available in our artifact [18].

GCC Coverage: Figure 2 shows that GrayC achieves the highest coverage, with 348,362 lines covered after 24 h of fuzzing. GrayC-No-Cov-Guidance is in second place (345,386 lines), followed by Clang-Fuzzer (324,101), PolyGlot (323,770), RegExpMutator (323,250), GrayC-Fragments-Fuzzing (315,287) and the initial corpus (314,467).

LLVM Coverage: Figure 2 shows that for LLVM, it is GrayC-No-Cov-Guidance which achieves the highest overall coverage (192,139 lines), followed by Clang-Fuzzer (191,305), GrayC (190,781), PolyGlot (186,620), RegExpMutator (186,308), GrayC-Fragments-Fuzzing (181,360 lines) and the initial corpus (180,551).

We believe Clang-Fuzzer and GrayC-No-Cov-Guidance achieve higher overall coverage in LLVM because (an older version of) LLVM is the compiler used for analysing and parsing programs during program generation. It is likely that Clang-Fuzzer's statically-invalid programs achieve substantial coverage of error-handling code in the front-end, which remain unchanged in the newer version of LLVM against which we measure coverage. Indeed, as discussed next, most of the coverage achieved by Clang-Fuzzer is in the front-end, while GrayC exercises the more challenging middle- and back-end. These two factors likely have a similar effect in GrayC-No-Cov-Guidance, which also generates a large number of statically-invalid programs.

Middle- and Back-End Coverage in LLVM: Recall that a key design goal of GrayC is to produce programs that are statically-valid, in order to exercise the middle- and back-end components of compilers and analysers. Thus, for LLVM, we also measured the coverage achieved by each fuzzing tool in the middle- and back-end of the compiler, based on a best-effort classification of LLVM source directories into front-end, middle-end and back-end components. Our hypothesis was that because GrayC is effective at generating diverse valid programs, it would achieve better coverage of the middle- and back-end components compared to other techniques.

Figure 3 shows the middle- and back-end coverage achieved by each tool after 24 h of fuzzing. We plot the mean and standard error over the hourly sampled rate per tool, while making available the raw data in our artifact [18]. GrayC achieves the highest coverage (middle-end: 66,914 lines, back-end: 71,053 lines), followed by GrayC-No-Cov-Guidance (66,594 and 70,553), PolyGlot (64,455 and 67,621), Clang-Fuzzer (63,367 and 67,651), RegExpMutator (63,269 and 67,323), GrayC-Fragments-Fuzzing (62,469 and 66,742), and the initial corpus (62,441 and 66,738).

Unlike for the overall coverage results, Clang-Fuzzer performs significantly worse than GrayC here, because it mostly generates statically-invalid programs that are rejected by the front-end. For similar reasons, the coverage difference between GrayC configurations and the rest of the fuzzers (RegExpMutator and PolyGlot) is more pronounced.

5.4 Bug Finding

To better understand the bug finding abilities of each tool, we used the sets of programs gathered via our 24 h fuzzing runs to test LLVM 12 and GCC 12 with optimisation levels `-O0`, `-O1`, `-O2`, `-O3` and `-Os`, and Frama-C-24 on Ubuntu 18.04 LTS x86_64.

We imposed a per-program timeout of 45 s for compilation and 200 s for Frama-C analysis.

We used the following process to de-duplicate the crashes triggered by these programs, to identify a set of unique bugs discovered by each fuzzing tool. First, we bucketed the crashes based on error messages printed by the compiler/analyser, e.g. "internal compiler error: tree check: expected array_type". We then searched the bug trackers of LLVM, GCC and Frama-C to look for an existing bug report corresponding to each bucket. Where we could find no related report, we checked whether the crash still manifested with the latest version of the compiler/analyser. This was the case for all but one crash. In these cases, we filed a new bug report and awaited feedback from developers. In a few cases, crashes that appeared to be due to distinct bugs (based on bucketing) turned out (according to developer feedback) to be due to the same underlying bug. One issue we reported to Frama-C was closed as "won't fix"; we discarded crashes corresponding to this bug from further consideration. All other bugs were confirmed by developers. Our complete set of unique bugs comprises the bugs we found that were already reported, plus the new bugs we reported (excluding the one that was closed as "won't fix"), plus the remaining bug that must have been independently fixed.

Results. Table 5 summarises the number of distinct middle-end and front-end bugs found by each fuzzing tool (none of the bugs found was classified as back-end bugs). GrayC is the most successful at finding middle-end bugs, with GrayC-No-Cov-Guidance also succeeding at finding such bugs. The middle-end bugs found by Clang-Fuzzer and RegExpMutator are bugs in the analysis component of Frama-C; these tools did not find any middle-end

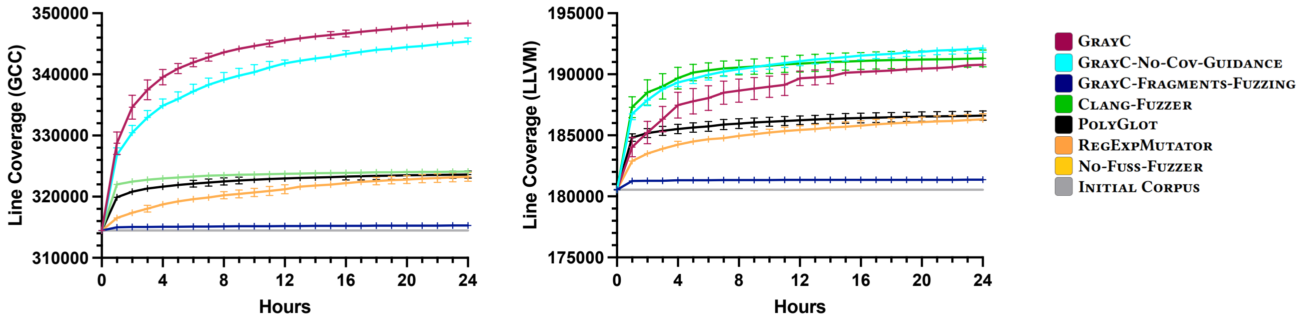

Figure 2: GCC and LLVM line coverage over 24 h of fuzzing.

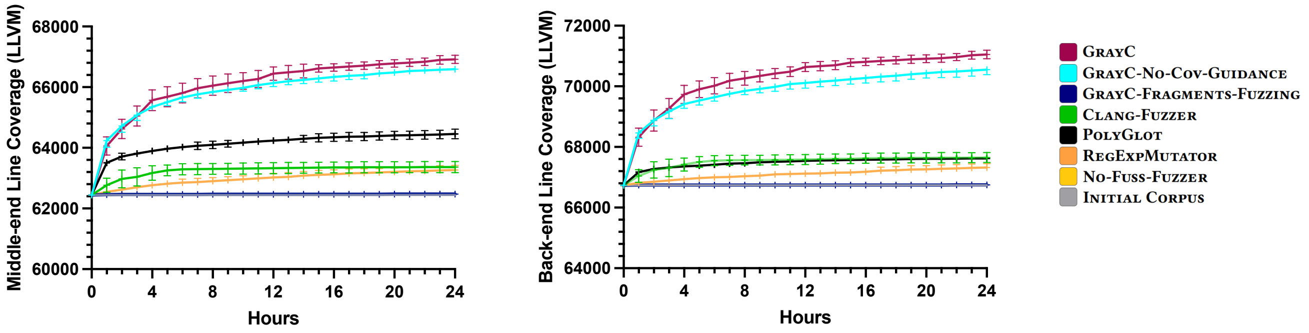

Figure 3: LLVM middle- and back-end coverage over 24 h of fuzzing.

Table 5: Confirmed unique bugs found by each tool during 24 h of fuzzing (union over 10 repetitions) in the middle- and front-end components.

Tool	Component		Fix Rate	Bug Report References
	Middle	Front		
GrayC	6	-	100%	[21, 30, 37, 38, 40, 46]
GrayC-No-Cov-Guidance	4	-	100%	[30, 38, 40, 46]
RegExpMutator	2	1	67%	[29, 30, 34]
Clang-Fuzzer	1	4	60%	[29, 34, 47, 48, 71]
PolyGlot	-	1	0%	Not reproducible
Csmith	-	-	0%	None
Grammarinator	-	-	0%	None
GrayC-Fragments-Fuzzing	-	-	0%	None

compiler bugs. The other fuzzing tools either found no bugs, or only front-end bugs.

In response to one of the front-end crashes in GCC found by both RegExpMutator and Clang-Fuzzer, triggered by statically-invalid programs, the developers responded: *"fuzzing source is going to turn up a lot of error-recovery cases - while somewhat interesting they will inevitably be [a] very low priority since GCC has mechanisms to present the user with a nicer error message..."* [47]. In LLVM, Clang-Fuzzer identified an incomplete program that led to a compiler hang and PolyGlot found a statically-invalid program that triggered a front-end ICE when parsing array types.

The low fix rate associated with front-end bugs (Table 5, "Fix Rate" column), the negative remarks and lack of action in relation to most of these somewhat pathological bugs which are triggered by statically-invalid programs, supports our hypothesis that for greybox fuzzing to work well in the domain of optimising compilers, mutation operators that yield statically-valid programs, such as those incorporated in GrayC, are essential.

Csmith, Grammarinator and GrayC-Fragments-Fuzzing did not find bugs during the controlled experiment. As discussed in §4, we did not find any bugs during a long-running testing campaign using Csmith; hence, it is unsurprising that Csmith did not uncover any bugs during this controlled experiment. We note that Frama-C has been extensively tested using Csmith in the past [16]. Grammarinator detected no bugs, probably due to its extremely low compilation rate and the fact the mature ahead-of-time compilers' front-ends have already been heavily tested. Similarly, GrayC-Fragments-Fuzzing's poor coverage delta (from the initial corpus) in both LLVM and GCC can explain these results.

6 RELATED WORK

As discussed in §1, randomised testing techniques have been successful in finding bugs in compilers for a range of languages, with a recent major focus on C (e.g. [65, 82, 105]), but also on other languages such as OpenCL [67], OpenGL [17], SQL [92] and Verilog [57]. There has also been significant work on applying similar techniques to testing code analysers (see e.g. [7, 16, 61, 63]).

Randomised compiler testing techniques mainly work by cross-checking multiple compilers (e.g. [57, 67, 105], a form of differential testing [54, 78], or checking expected equivalences between programs (e.g. [17, 65]), a form of metamorphic testing [9, 93]. We refer the reader to a survey for an overview of state-of-the-art techniques [8] and to a recent paper for a discussion of the importance of fuzzer-found bugs [77]. The main difference between these existing works and ours is that GrayC employs *greybox* fuzzing.

In §1 we have already discussed mutation-based fuzzing techniques in the context of dynamic languages such as JavaScript, particularly the pioneering work on LangFuzz [59] and more recent work on Superion [103] and Nautilus [1]. The recent PolyGlot technique [11] caters for *generic* language processor fuzzing, and is applicable to both dynamic and static languages, including C. Our evaluation against a variant of GrayC resembling LangFuzz (since the LangFuzz tool is unavailable) and against PolyGlot demonstrates the advantages of our approach.

A similar language-agnostic work is on "no-fuss fuzzing" of compilers [55], which investigates applying AFL-based greybox fuzzing to compilers for a number of smart contract languages and the Zig language [107]. Instead of building per-language custom mutators, this work investigates using regular expression based mutation, based on (a partial re-implementation of) the Universal Mutator tool [53], and mutation based on approximate parsing of input programs using simple features common to many languages, such as balanced parentheses [101]. The authors of [55] remark that their approach is geared towards languages that aim to be *total*, so that the compiler should behave gracefully for any input, and they explicitly comment that such approaches are less likely to be useful in the context of C/C++ compilers. This is supported by our experiments in §5.1, using a Universal Mutator-based LibFuzzer custom mutator.

An avenue for generating test cases via grammar-based techniques is explored in [100] in the context of "little languages". They reported their observations based on the evaluation with 61 single-pass student compilers and a grammar-based technique. Their findings have suggested the value of having an automated test case generation technique conjunctly used with the developer-populated test suite. Kifetew et al. [62] applied a stochastic context-free grammar to generate *valid sentences* and achieve high system-level branch coverage. They experimentally compared their suggestion of combining genetic programming with probabilities learned from corpora versus a semi-manual alternative approach of applying grammar annotations with genetic operators (when using a corpus for learning is not affordable) on six open-source Java systems containing hundreds of thousands of lines at most. Neither [100] nor [62] have evaluated a system equivalent in scale to mature C compilers with massive codebases: GCC has about 15 million lines of code [104], while LLVM is even larger [76].

GrayC builds on the (very basic) Clang-Fuzzer tool [13], which provides a fuzz target for Clang and uses LibFuzzer's default byte-level mutators. Our experimental results showed that, due to the naivety of byte-level mutators, Clang-Fuzzer is ineffective at finding deep compiler bugs. We attempted to compare with Clang-Proto-Fuzzer [14], an extension of Clang-Fuzzer that features partially semantics-aware mutators based on a protobuf description of a fragment of C++, but found that this project is no longer maintained and is not currently in a usable state. A presentation on the work already reported that developers have not been responsive to the bugs that it found (see §1).

An approach to differential testing of Java Virtual Machine (JVM) implementations also takes a coverage-guided approach [10]. Unlike our work, this approach does not focus on mutations that produce valid programs; in fact, the focus is on looking for discrepancies where one JVM accepts a class file, while another rejects it as being malformed.

7 CONCLUSION AND FUTURE WORK

We have presented the design of our coverage-directed mutation-based compiler fuzzing approach and its implementation, GrayC. Our evaluation demonstrates that GrayC can achieve better coverage of the middle- and back-end components of compiler codebases compared with other mutation-based approaches, leading to the discovery of numerous previously unknown bugs and to the contribution of new tests to the Clang/LLVM test suite. Future work will focus on revisiting the *conservative* mode of the tool which, as discussed in §3.1 turned out to perform poorly in terms of bug-finding ability and code coverage compared with GrayC's standard *aggressive* mode, and improving GrayC's facilities for potentially finding miscompilation bugs, e.g. by augmenting GrayC with mutations inspired by particular compiler optimisations of interest.

8 DATA AVAILABILITY

GrayC, enhanCer and the experimental infrastructure, data, and results are available as open source at [18, 49, 60].

9 ACKNOWLEDGEMENTS

This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141) and from the UK Engineering and Physical Sciences Research Council through grants EP/R011605/1 and EP/R006865/1.

REFERENCES

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proc. of the 26th Network and Distributed System Security Symposium (NDSS'19)* (San Diego, CA, USA). https://doi.org/10.14722/ndss.2019.23412

[2] C-Reduce Bug - clang delta (found as a by-product of fuzzing). Date Confirmed and Fixed Jan. 16, 2022. https://www.flux.utah.edu/listarchives/creduce-bugs/msg00555.html.

[3] C-Reduce Bug - clang delta (found as a by-product of fuzzing). Date Confirmed and Fixed Jan. 4, 2022. https://www.flux.utah.edu/listarchives/creduce-bugs/msg00553.html.

[4] C-Reduce Bug - clang delta (found as a by-product of fuzzing). Date Confirmed Jun. 7, 2021 and Fixed Jun. 20, 2021. https://www.flux.utah.edu/listarchives/creduce-bugs/msg00537.html.

[5] C-Reduce Bug - clang delta (found as a by-product of fuzzing). Date Confirmed November 2, 2022. https://www.flux.utah.edu/listarchives/creduce-bugs/msg00563.html.

[6] C-Reduce Bug - clang delta (found as a by-product of fuzzing). Date Reported Dec. 17, 2021. https://www.flux.utah.edu/listarchives/creduce-bugs/msg00551.html.

[7] Cristian Cadar and Alastair Donaldson. 2016. Analysing the Program Analyser. In *Proc. of the 38th International Conference on Software Engineering, New Ideas and Emerging Results (ICSE NIER'16)* (Austin, TX, USA).

[8] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* 53, 1 (2020), 4:1–4:36. https://doi.org/10.1145/3363562

[9] T.Y. Chen, S.C. Cheung, and S.M. Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report HKUST-CS98-01. Hong Kong University of Science and Technology.

[10] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proc. of the*

Conference on Programing Language Design and Implementation (PLDI'16) (Santa Barbara, CA, USA). https://doi.org/10.1145/2908080.2908095

[11] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2022. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'22)* (San Francisco, CA, USA). https://doi.org/10.1109/SP40001.2021.00071

[12] Clang LibTooling 2023. LibTooling. https://clang.llvm.org/docs/LibTooling.html.

[13] clangfuzzer [n. d.]. clang-fuzzer. https://github.com/llvm/llvm-project/tree/main/clang/tools/clang-fuzzer.

[14] clangprotofuzzer [n. d.]. clang-proto-fuzzer. https://llvm.org/docs/FuzzingLLVM.html#clang-proto-fuzzer.

[15] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: A software analysis perspective. In *Proc. of the 10th International Conference on Software Engineering and Formal Methods (SEFM'12)* (Thessaloniki, Greece). https://doi.org/10.1007/s00165-014-0326-7

[16] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *Proc. of the 4th International Conference on NASA Formal Methods (NFM'12)* (Norfolk, VA, USA).

[17] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, OOPSLA (2017), 93:1–93:29. https://doi.org/10.1145/3133917

[18] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. Artifact of GrayC: Greybox Fuzzing of Compilers and Analysers for C. https://doi.org/10.5281/zenodo.7976254. Zenodo.

[19] Frama-C Bug - Eva plugin. Date Confirmed Mar. 13, 2022 and Closed and fixed Jul. 11, 2022. https://git.frama-c.com/pub/frama-c/-/issues/2595.

[20] Frama-C Bug - Eva plugin. Date Confirmed May 10, 2022 and Fixed Jun. 10, 2022. https://git.frama-c.com/pub/frama-c/-/issues/2610.

[21] Frama-C Bug - Eva plugin. Date Confirmed Nov. 8, 2021 and Fixed Sept. 15, 2022. https://git.frama-c.com/pub/frama-c/-/issues/2585.

[22] Frama-C Bug - Eva plugin, kernel, abstract interpretation. Date Confirmed and Fixed Jun. 10, 2021. https://git.frama-c.com/pub/frama-c/-/issues/2563.

[23] Frama-C Bug - Front-end. Date Confirmed Oct. 14, 2021 and Date Fixed Dec. 3, 2021. https://git.frama-c.com/pub/frama-c/-/issues/2576.

[24] Frama-C Bug - Front-end (found as a by-product of fuzzing). Date Confirmed May 28, 2022 and Fixed Oct. 20, 2021. https://git.frama-c.com/pub/frama-c/-/issues/2559.

[25] Frama-C Bug - Front-end (found as a by-product of fuzzing). Date Confirmed Sept. 14, 2021 and Fixed Jul. 11, 2022. https://git.frama-c.com/pub/frama-c/-/issues/2573.

[26] Frama-C Bug - Front-end (found as a by-product of fuzzing). Date Confirmed Sept. 16, 2021 and Fixed Oct. 20, 2021. https://git.frama-c.com/pub/frama-c/-/issues/2574.

[27] Frama-C Bug - Kernel. Date Confirmed Apr. 20, 2021 and Fixed Apr. 30, 2021. https://git.frama-c.com/pub/frama-c/-/issues/2551.

[28] Frama-C Bug - Kernel. Date Confirmed Apr. 6, 2021 and Fixed Oct. 13, 2021. https://git.frama-c.com/pub/frama-c/-/issues/2550.

[29] Frama-C Bug - kernel. Date Confirmed Jan. 11, 2022 and Fixed Jul. 11, 2022. https://git.frama-c.com/pub/frama-c/-/issues/2592.

[30] Frama-C Bug - kernel, abstract interpretation. Date Confirmed and Fixed Jan. 24, 2022. https://git.frama-c.com/pub/frama-c/-/issues/2588.

[31] Frama-C Bug - kernel, abstract interpretation. Date Confirmed May 18, 2021 and Fixed May 21, 2021. https://git.frama-c.com/pub/frama-c/-/issues/2556.

[32] Frama-C Bug - kernel, Front-end. Date Confirmed Jan. 10, 2022 and Fixed Feb. 9, 2022. https://git.frama-c.com/pub/frama-c/-/issues/2590.

[33] Frama-C Bug - Parsing, EVA-plugin. Date Confirmed May 18, 2021 and Fixed May 21, 2021. https://git.frama-c.com/pub/frama-c/-/issues/2555.

[34] GCC Bug. Date Reported Aug. 6, 2016. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=72825.

[35] GCC Bug - Front-end. Date Confirmed Apr. 9, 2021 and Fixed Apr. 22, 2021. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99990.

[36] GCC Bug - Front-end. Date Confirmed Aug. 8, 2022 and Fixed Nov. 21, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=106560.

[37] GCC Bug - ipa. Date Confirmed Dec. 23, 2021 and Fixed Apr. 20, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=103818.

[38] GCC Bug - Middle-end. Date Confirmed Dec. 22, 2021 and Fixed Jan. 24, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=103813.

[39] GCC Bug - Middle-end. Date Confirmed May 02, 2022 and Fixed May 27, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=104402.

[40] GCC Bug - Middle-end (reported independently before we found it). Date Confirmed Mar. 20, 2018 and Fixed Apr. 14, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=84964.

[41] GCC Bug - Middle-end (reported independently shortly before we found it). Date Confirmed Nov. 18, 2022 and Fixed Nov. 19, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=103314.

[42] GCC Bug - rtl-optimization, middle-end. Date Reported Jun. 9, 2022 and Confirmed May 17, 2023. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=105910.

[43] GCC Bug - Tree optimization. Date Confirmed and Fixed Apr. 12, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=105232.

[44] GCC Bug - Tree optimization. Date Confirmed and Fixed Jun. 10, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=107170.

[45] GCC Bug - Tree optimization. Date Confirmed Dec. 23, 2021 and Fixed Jan. 5, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=103816.

[46] GCC Bug - Tree-optimization (reported independently before we found it). Date Confirmed Jul. 27, 2021 and Fixed Mar. 23, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101636.

[47] GCC Bug: incomplete program (several duplicate reports exist). Date Reported Aug. 28, 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=106764.

[48] GCC Bug (several related reports exist). Date Reported May 11, 2021, and Fixed 15 November 2022. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100525.

[49] GitHub. 2023. Git Repository of GrayC. https://github.com/srg-imperial/GrayC.

[50] GitHub. Date Accessed December 31, 2022. Git Repository of Grammarinator. https://github.com/renatahodovan/grammarinator.git.

[51] GitHub. Date Accessed March 23, 2022. Git Repository of gfauto. https://github.com/google/graphicsfuzz.git.

[52] Google. 2020. AFL dictionaries. https://github.com/google/AFL/blob/master/dictionaries/README.dictionaries.

[53] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. 2018. An Extensible, Regular-Expression-Based Tool for Multi-language Mutant Generation. In *Proc. of the 40th International Conference on Software Engineering (ICSE'18)* (Gothenburg, Sweden). https://doi.org/10.1145/3183440.3183485

[54] Alex Groce, Gerard J. Holzmann, and Rajeev Joshi. 2007. Randomized Differential Testing as a Prelude to Formal Verification. In *Proc. of the 29th International Conference on Software Engineering (ICSE'07)* (Minneapolis, MN, USA). https://doi.org/10.1109/ICSE.2007.68

[55] Alex Groce, Rijnard van Tonder, Goutamkumar Tulajappa Kalburgi, and Claire Le Goues. 2022. Making No-Fuss Compiler Fuzzing Effective. In *Proc. of the 31st International Conference on Compiler Construction (CC'22)* (Seoul, Korea). https://doi.org/10.1145/3497776.3517765

[56] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2019. Verifying Instruction Set Simulators using Coverage-guided Fuzzing*. In *Proc. of the 22nd Design, Automation & Test in Europe Conference & Exhibition (DATE'19)* (Florence, Italy). IEEE, 360–365. https://doi.org/10.23919/DATE.2019.8714912

[57] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proc. of the 28th International Symposium on Field-Programmable Gate Arrays (FPGA'20)*. ACM/SIGDA, 277–287. https://doi.org/10.1145/3373087.3375310

[58] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-Based Open Source Fuzzer. In *Proc. of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST'18)* (Lake Buena Vista, FL, USA). https://doi.org/10.1145/3278186.3278193

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

[60] GrayC Homepage. Date Accessed May 23, 2022. https://srg.doc.ic.ac.uk/projects/grayc/.

[61] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proc. of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17)* (Urbana-Champaign, IL, USA).

[62] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. 2017. Generating Valid Grammar-Based Test Inputs by Means of Genetic Programming and Annotated Grammars. *Empirical Softw. Engg.* 22, 2 (apr 2017), 928–961. https://doi.org/10.1007/s10664-015-9422-4

[63] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'19)* (Beijing, China).

[64] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, CA, USA). https://doi.org/0.1109/CGO.2004.1281665

[65] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'14)* (Edinburgh, UK). https://doi.org/10.1145/2594291.2594334

[66] LibFuzzer 2022. http://llvm.org/docs/LibFuzzer.html.

[67] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'15)* (Portland, OR, USA). https://doi.org/10.1145/2737924.2737986

[68] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. In *Proc. of the ACM on Programming Languages (OOPSLA'20)* (Chicago, IL, USA). https://doi.org/10.1145/3428264

[69] LLVM Bug - Arrays. Date Reported Jun. 9, 2021 and Closed Jan. 7, 2022. https://github.com/llvm/llvm-project/issues/49983.

[70] LLVM Bug - Clang codegen (found as a by-product of fuzzing). Date Confirmed Jan. 15, 2022. https://github.com/llvm/llvm-project/issues/53105.

[71] LLVM Bug - Clang Front-end. Date Reported May 6, 2022. https://github.com/llvm/llvm-project/issues/55312.

[72] LLVM Bug - compiler-rt:ubsan (found as a by-product of fuzzing). Date Confirmed Jan. 16, 2022. https://github.com/llvm/llvm-project/issues/51421.

[73] LLVM Bug - IR (found as a by-product of fuzzing). Date Reported Jul. 5, 2021. https://github.com/llvm/llvm-project/issues/50332.

[74] LLVM Bug - Union declaration. Date Reported Jun. 10, 2021 and Closed Jan. 13, 2022. https://github.com/llvm/llvm-project/issues/49993.

[75] LLVM Project. Date Accessed July 21, 2022. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[76] LLVM website [n. d.]. LLVM website. http://llvm.org/.

[77] Michaël Marcozzi, Qiyi Tang, Alastair Donaldson, and Cristian Cadar. 2019. Compiler Fuzzing: How Much Does It Matter?. In *Proc. of the ACM on Programming Languages (OOPSLA'19)* (Athens, Greece). https://doi.org/10.1145/3360581

[78] W. M. McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10 (1998), 100–107. Issue 1.

[79] MSVC Bug - CppCompiler, Front-end. Date Confirmed May 20, 2021. https://developercommunity.visualstudio.com/t/internal-compiler-error-when-compiling-program-wit/1427557.

[80] MSVC Bug - CppCompiler, Front-end. Date Confirmed May 20, 2021 and Closed Nov. 24, 2021. https://developercommunity.visualstudio.com/t/internal-compiler-error-when-compiling-program-wit/1427553.

[81] MSVC Bug - CppCompiler, Front-end. Date Confirmed May 20, 2021 and Fixed Nov. 9, 2021. https://developercommunity.visualstudio.com/t/syntactically-invalid-c-program-causes-microsoft-c/1427550.

[82] Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random testing of C compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. https://doi.org/10.1109/APCCAS.2016.7804063

[83] Phabricator-LLVM. 2020. Requests D88931 and D97686. https://reviews.llvm.org/D88931. Date Approved March 3, 2021.

[84] Phabricator-LLVM. 2022. Request D118234. https://reviews.llvm.org/D118234. Date Approved October 11, 2022.

[85] Phabricator-LLVM. 2023. Requests D142638 and D150857. https://reviews.llvm.org/D150857. Under review: date January 26, 2023 (re-open: May 18, 2023).

[86] John Regehr. 2020. The Saturation Effect in Fuzzing. https://blog.regehr.org/archives/1796.

[87] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'12)* (Beijing, China). https://doi.org/10.1145/2254064.2254104

[88] LLVM Bug - Front-end (reported independently before we found it). Date Confirmed Jan. 26, 2022. https://github.com/llvm/llvm-project/issues/49081.

[89] LLVM Bug - ASan (reported independently before we found it). Date Reported Feb. 20, 2021. https://github.com/llvm/llvm-project/issues/48633.

[90] LLVM Bug - Front-end (reported independently before we found it). Date Reported Jun. 26, 2021. https://github.com/llvm/llvm-project/issues/50222.

[91] LLVM Bug - Front-end (reported independently before we found it). Date Reported Nov. 12, 2015 and Fixed on early 2022. https://github.com/llvm/llvm-project/issues/25871.

[92] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-optimizing Reference Engine Construction. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'20)* (Online). https://doi.org/10.1145/3368089.3409710

[93] Sergio Segura, Gordon Fraser, Ana Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. (2016).

[94] Kostya Serebryany. 2022. Personal communication.

[95] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)* (Boston, MA, USA). https://doi.org/10.5555/2342821.2342849

[96] Kostya Serebryany, Vitaly Buka, and Matt Morehouse. 2017. Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator. In *2017 US LLVM Developers' Meeting*. https://llvm.org/devmtg/2017-10/slides/.

[97] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. 2021. SiliFuzz: Fuzzing CPUs by proxy. *CoRR* abs/2110.11519 (2021). arXiv:2110.11519

[98] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proc. of the International Symposium on Code Generation and Optimization (CGO'15)* (San Francisco, CA, USA). https://doi.org/10.1109/CGO.2015.7054186

[99] UBSan 2017. Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[100] Phillip van Heerden, Moeketsi Raselimo, Konstantinos Sagonas, and Bernd Fischer. 2020. Grammar-Based Testing for Little Languages: An Experience Report with Student Compilers. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 253–269. https://doi.org/10.1145/3426425.3426946

[101] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight multi-language syntax transformation with parser parser combinators. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'19)* (Phoenix, AZ, USA). https://doi.org/10.1145/3314221.3314589

[102] Rijnard van Tonder and Alex Groce. 2022. Making No-Fuss Compiler Fuzzing Effective: CC 2022 Artifact (0.1.0). https://doi.org/10.5281/zenodo.5982794. Zenodo.

[103] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *Proc. of the 41st International Conference on Software Engineering (ICSE'19)* (Montreal, Canada). https://doi.org/10.1109/ICSE.2019.00081

[104] Wikipedia: GNU Compiler Collection. Date Accessed May 18, 2022. https://en.wikipedia.org/wiki/GNU_Compiler_Collection#cite_note-loc-4.

[105] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11)* (San Jose, CA, USA). https://doi.org/10.1145/1993498.1993532

[106] Michal Zalewski. [n. d.]. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[107] Zig Software Foundation. Date Accessed September 1, 2022. Zig programming language. https://ziglang.org/.

Received 2023-02-16; accepted 2023-05-03