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

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

Arindam Sharma∗
arindam.sharma@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 fuzzed-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.

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

ACM Reference Format:

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 LinFuzzer [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, 63]). 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, 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) 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, 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. 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. 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.
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 GRAMMAR-MINATOR [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- and dynamically-valid 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 (\textit{\S}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* (\textit{\S}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 (\textit{\S}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 \S4). 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 (\S4).

We first discuss the custom mutators employed by GRAYC, whose key objective is to produce statically-valid programs (\textit{\S}3.1). We then describe our *Enhancer* tool that allows GRAYC to be used for differential testing and compiler test suite augmentation (\textit{\S}3.2), and describe pertinent implementation details (\textit{\S}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:

```c
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:

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

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

```c
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:

```
typedef struct {
  unsigned w[3];
} Y;
Y arr[32];

int main() {
  int i = 0;
  for (i=0; i<32; ++i)
    arr[i].w[1] = i == 1;
  for (i=0; i<32; ++i)
    arr[i].w[1] = i == 1;
  if (i=1)
    abort();
  return 0;
}
```

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* recombiners the following two programs: $P_{\text{blue}}$ (the destination program) and $P_{\text{green}}$ (the source program).
We mark the lines used in the output programs in blue if they originate from \(P\) Combine-Functions and in green if they originate from \(P\) origminate from:

\[
\begin{align*}
\text{Program } P_{\text{blue}}: & \\
\text{int } \text{dest_func}(\text{int } x_{\text{dest}}, \text{int } y_{\text{dest}}) \{ \\
\text{int } b_{\text{dest}} = x_{\text{dest}} \text{ dest}; \\
\text{int } b_{\text{dest}} = b_{\text{dest}} + y_{\text{dest}}; \\
\text{return } b_{\text{dest}}; \\
\} \end{align*}
\]

\[
\begin{align*}
\text{Program } P_{\text{green}}: & \\
\text{int } \text{are}(); \\
\text{int } \text{source_func}(\text{int } j_{\text{src}}, \text{int } k_{\text{src}}) \{ \\
\text{return } m_{\text{src}}; \\
\} \\
\end{align*}
\]

The recombiner merges the body of \text{source_func} in \(P_{\text{green}}\) into the body of \text{dest_func} in \(P_{\text{blue}}\). 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 \text{COMBINE-FUNCTIONS} could output. We mark the lines used in the output programs in blue if they originate from \(P_{\text{blue}}\), and in green if they originate from \(P_{\text{green}}\).

\[
\begin{align*}
\text{Output program } P_1: & \\
\text{int } \text{dest_func}(\text{int } x_{\text{dest}}, \text{int } y_{\text{dest}}) \{ \\
\text{int } j_{\text{src}}; \\
\text{int } k_{\text{src}}; \\
\text{int } b_{\text{dest}} = x_{\text{dest}} \text{ dest}; \\
\text{int } b_{\text{dest}} = b_{\text{dest}} + y_{\text{dest}}; \\
\text{return } b_{\text{dest}}; \\
\} \end{align*}
\]

\[
\begin{align*}
\text{Output program } P_2: & \\
\text{int } \text{dest_func}(\text{int } x_{\text{dest}}, \text{int } y_{\text{dest}}) \{ \\
\text{int } j_{\text{src}}; \\
\text{int } k_{\text{src}}; \\
\text{int } b_{\text{dest}} = x_{\text{dest}} \text{ dest}; \\
\text{int } b_{\text{dest}} = b_{\text{dest}} + y_{\text{dest}}; \\
\text{return } b_{\text{dest}}; \\
\} \end{align*}
\]

\text{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: \text{COMBINE-FUNCTIONS} randomly selects one of the two return values (\(P_1\) uses the return statement from \(P_{\text{blue}}\), while \(P_2\) that from \(P_{\text{green}}\) and adds it as a single return statement of the merged function.

\textbf{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 \textit{conservative} mode to GrayC that applies mutations less aggressively with the aim of generating dynamically-valid programs with higher probability. In this mode, certain mutations 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 \textit{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 \textit{aggressive} mode both with respect to bug-finding ability and

\begin{table}[!ht]
\centering
\caption{GrayC’s code mutators and recombiners.}
\begin{tabular}{|c|c|c|l|}
\hline
\textbf{#} & \textbf{Type} & \textbf{Construct} & \textbf{Short Name} & \textbf{Description} \\
\hline
1 & Mutator & Statement & 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 & Mutator & Inject-Control-Flow & ADD-BREAK & 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 & Mutator & Delete-Expression & DELETE-EXPRESSION & Delete sub-expressions from a given expression in a corpus program. \\
5 & Mutator & Expand-Expression & EXPAND-EXPRESSION & Expand sub-expressions with other sub-expressions from the corpus program; e.g. in an assignment or loop condition. \\
6 & Mutator & Replace-Constant & 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 \(b\), making its body dead code. \\
7 & Mutator & Flip-Bit & REPLACE-DIGIT & Similar to Flip-Bit but on the number’s decimal representation: either flip the sign or change a single digit. \\
8 & Mutator & Change-Type & CHANGE-TYPE & Change the type of an expression (\textit{short}, \textit{long}, \textit{unsigned}, \textit{float}, etc.). \\
9 & Mutator & Replace-Unary-Operator & REPLACE-UNARY-OPERATOR & Replace unary operator with an assignment using the same variable; e.g. replace \(+\) in a \textit{for} statement to \(i+=2\) or \(i+=3\). \\
10 & Mutator & Flip-Operator & Flip-Operator & Replace one operator with another (arithmetic operators). \\
11 & Mutator & Replace-Function-Body & REPLACE-FUNCTION-BODY & Replace the body of a function with that of another function with the same number of arguments. \\
12 & Recombiner & Function & REPLACE-FUNCTION-BODY & Replace the body of a function with that of another function with the same number of arguments. \\
13 & Recombiner & Reference & Reference & Combine the body of a function with another function with the same number of arguments, either by concatenating bodies or interleaving their statements. \\
\hline
\end{tabular}
\end{table}
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. 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 FRAME-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.

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].
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 us to finding any compilation 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:

```c
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 \texttt{d.c[1]} with \texttt{d.c[1] || d.c[1]}, and then using Replace-By-Constant, which modified \texttt{d.c[1] || d.c[1]} to \texttt{d.c[268435456] \&\& \texttt{d.c[1]}}. During constant folding (middle-end), the decompression of \texttt{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] 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 11b. 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.

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 LanoFuzz 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.

---

7Ten 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.

8This refers to GrayC running its standard aggressive mode. We also performed our full set of controlled experiments using the conservative mode described in §1. As discussed in §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.

9LanoFuzz 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 nonduplicate 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).

### 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.\(^6\) 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

\(^6\)Csmith-generated programs are by construction compilable; however, some files hit our compilation timeout.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Programs/h</th>
<th>Statically-valid (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Csmith</td>
<td>1.144</td>
<td>99.96%</td>
</tr>
<tr>
<td>GrayC</td>
<td>2.906</td>
<td>99.47%</td>
</tr>
<tr>
<td>GrayC-Fragments-Fuzzing</td>
<td>4.152</td>
<td>99.08%</td>
</tr>
<tr>
<td>PolyGlot</td>
<td>641</td>
<td>91.26%</td>
</tr>
<tr>
<td>GrayC-No-Cov-Guidance</td>
<td>4.629</td>
<td>75.04%</td>
</tr>
<tr>
<td>RegExpMutator</td>
<td>1.392</td>
<td>19.21%</td>
</tr>
<tr>
<td>Clang-Fuzzer</td>
<td>1.183</td>
<td>1.55%</td>
</tr>
<tr>
<td>Grammarinator</td>
<td>5.391</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

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.
We measured coverage for (192,139 lines), followed by (323,250), 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.

5.3 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 gcov [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 (323,520), 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.
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.

**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 fuzzier-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


[79] 1109/APPCAS.2016.7804063