Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

FuzzFlesh

A multi-language compiler testing tool

Author:
Amber Gorzynski

Supervisor:
Prof. Alastair Donaldson

Submitted in partial fulfillment of the requirements for the MSc degree in Computing of Imperial College London

September 2023

Abstract

I create FuzzFlesh, a novel compiler testing tool with support for multiple languages. FuzzFlesh produces a set of executable test cases by fleshing randomly generated control flow graphs into programs in a target language. Each test case is constructed to have a deterministic execution path through the control flow graph. If the actual runtime path differs from the expected path, then a miscompilation bug may be responsible. This approach specifically targets control flow optimisations, an area of relative compiler complexity. FuzzFlesh and can be easily applied to a range of languages because the graph and path generation components are language-independent.

I test the approach on 4 different languages (LLVM IR, Java bytecode, CIL, C) and 10 compiler toolchains including a mix of well-used and more experimental toolchains (LLVM, GraalVM LLVM, GraalVM Java, HotSpot, CFR, Fernflower, Mono, ILSpy, Clang, GCC). Among these, FuzzFlesh found two bugs: one in each of the Java decompilers CFR and Fernflower. The Java decompilers are relatively experimental and must contend with control flow constraints. These results indicate that FuzzFlesh may be most effective when applied to toolchains that are capable of handling control flow restrictions.

I evaluate FuzzFlesh on LLVM IR, which has relatively accessible tooling. I use two metrics: (1) compiler code coverage and (2) mutation analysis, defined here as the ability of the approach to detect and kill synthetic bugs within the compiler. FuzzFlesh achieves reasonable performance on both metrics, given its simplicity, indicating that it can be a useful method for testing novel or early-stage languages or compiler toolchains. Further work could include adding language-specific features to increase the complexity of the programs, and testing toolchains that contain more complex control flow restrictions.

Contents

1 Introduction

1.1 Motivation

Compilers are essential software components. Like all other software, they suffer from bugs. Many testing tools exist, but the complexity of designing test approaches means that each tool tends to have particular strengths and limitations: they are good at detecting some types of bugs and not others. In addition, developing testing tools typically takes a significant amount of programming effort.

In this project I build on Klimis et al. [1] to develop a novel compiler testing approach called FuzzFlesh that complements existing compiler fuzzing methods. FuzzFlesh randomly generates two structures: (1) control flow graph 'skeletons' and (2) paths through each graph. It fleshes the graph-path combinations into test programs in a target language. Each program is equipped with a test oracle that verifies whether the actual runtime path through the graph matches the expected path. Any difference in the paths indicates a possible miscompilation. The self-checking nature of the test cases means that tests can be performed directly on a single compiler, in contrast to other testing methods that require multiple compiler versions.

A key strength of the approach is that most of the toolchain required can be implemented in a language-independent way. It can therefore be easily extended to multiple languages. The language-specific part of the toolchain does not take long for an experienced programmer to implement. This is in contrast to other tools, which require extensive language-specific engineering to produce test cases. The method could therefore be particularly useful for applications in novel languages and toolchains.

A further potential advantage of this method is its focus on exercising the control flow optimisations within compilers. While some existing compiler testing tools support a range of control flow constructs, this is not their central focus. And many tools have limited support for complex control flow because it makes other aspects of testing more difficult. These tools mainly exercise the parts of the compiler responsible for analysing language and optimisations that concern statement construction.

1.2 Objectives

This project aims to answer the following research questions:

RQ1 How effective is FuzzFlesh at identifying real compiler bugs?

RQ2 How thoroughly is FuzzFlesh able to test compiler optimisations?

RQ3 How does the efficacy of FuzzFlesh vary across languages and compiler toolchains?

RQ4 To what extent can FuzzFlesh identify bugs that are different from those identified by other approaches, and what are the drivers of any differences?

RQ5 How feasible is it to extend FuzzFlesh to multiple languages? What are the key constraints imposed by having a shared program skeleton generation component of the fuzzing toolchain?

1.3 Contributions

In this project I have answered the above research questions by:

- Creating a novel compiler testing toolchain that features shared program generation across multiple languages, based on control flow graph generation.
- Implementing the toolchain for four distinct languages that are characterised by unstructured control flow: LLVM IR, Java Bytecode, CIL, and C.[1]
- Evaluating the performance of the fuzzer using a range of metrics including fuzzing campaign outcomes for 10 compiler toolchains (LLVM, GraalVM LLVM, GraalVM Java, HotSpot, CFR, Fernflower, Mono, ILSpy, Clang, GCC), compiler code coverage, and mutation analysis.

FuzzFlesh was able to identify two distinct bugs within Java decompilers: one each in CFR and Fernflower. It was also able to achieve a reasonable level of code coverage and mutant kills in the LLVM compiler. FuzzFlesh may therefore be most useful for applications to compiler toolchains that must implement complex rules relating to control flow.

The remainder of this report is structured as follows:

- Section 2 provides an overview of the literature on compiler testing.
- Section 3 sets out the overall design of FuzzFlesh and discusses the language-independent components in detail.
- Section 4 discusses the implementation of the language-specific program fleshing approaches for each language.
- Section 5 describes each compiler toolchain that I tested during the project.
- Section 6 contains an evaluation of FuzzFlesh and reports the bugs in detail along with other evaluation metrics.
- Section 7 summarises the key findings with respect to each research question that is set out above in 1.2, and discusses areas of further work.

[1] In this context, 'unstructured' is defined as allowing unrestricted control flow. I note that C could be described as a structured language due to its support for hierarchically-organised constructs such as nested loops. However, for the purposes of this work I regard C to be unstructured because `goto` can be used to arbitrarily transfer control flow to any part of a program. This is in contrast to Java, for example, which imposes restrictions on the transfer of control.

2 Background

This section summarises the relevant literature on compiler testing. First, I describe the core functionality of compilers and the motivation for testing them. Next, I set out the general framework used for compiler testing. I then review and compare the main approaches to compiler testing.

2.1 Compilers and their bugs

Compilers are programs that translate code from a source language to a lower level target language. The compilation process consists of multiple connected phases, each of which performs a different task.

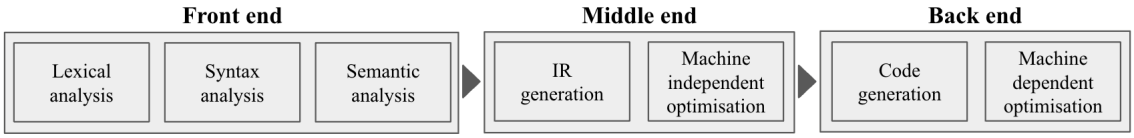


Figure 1: Compiler structure

The specific design varies across implementations. In general there are three passes:

- Front end (analysis): This contains lexical, syntactic, and semantic analysis. The purpose is to perform checks to ensure that the source code conforms to all language-specific requirements on syntax and semantics. For example, an important component of semantic analysis is type checking.
- Middle end (optimisation): Following analysis, the source code is translated to an intermediate representation that can be machine-independent (e.g. LLVM IR). Optimisations are performed with respect to some goals, such as faster or shorter code. The amount and nature of optimisations carried out varies extensively across compilers.
- Back end (code generation): Finally, the code generation phase translates the (optimised) intermediate representation to the target language. Depending on the compiler, this may be a machine-specific language, in which case machine-dependent optimisations may be performed.

Compilers can contain bugs at any phase. Bugs can be categorised as crash-inducing or miscompilation bugs. The latter is more insidious: the target code produced by the compiler is silently incorrect. This can cause unexpected problems in the program at runtime and can propagate to downstream applications, for example see Sun, Zhang and Su [2]. It is also difficult to detect because developers tend not to suspect the compiler as a source of bugs. Compiler testing is therefore an important and widely researched area. Chen et al. [3] provide a comprehensive review of current state-of-the-art compiler testing approaches, which covers 85 relevant papers. Note that in this review I focus on empirical compiler testing and do not cover formal verification. [2]

[2] It is difficult to formally verify compilers because both the inputs and outputs of compilers are arbitrarily long and complex. There is a lack of formal specification for exactly how compilers should operate, for example exactly which optimisation passes should be performed (and in what order).

2.2 Testing approach

Figure 2 shows the core components of compiler testing. The implementation of each component varies across test approaches. First, a set of 'test programs' is generated. Next, a test harness feeds the test programs to the compiler of interest and evaluates a 'test oracle' on the resulting output to determine whether the test has passed or failed. Test failure indicates the presence of a potential compiler bug. Finally, if a bug is detected, then a test case reducer is used to produce a minimised bug-triggering test case. This is necessary because the original test program may be large, which makes debugging difficult.

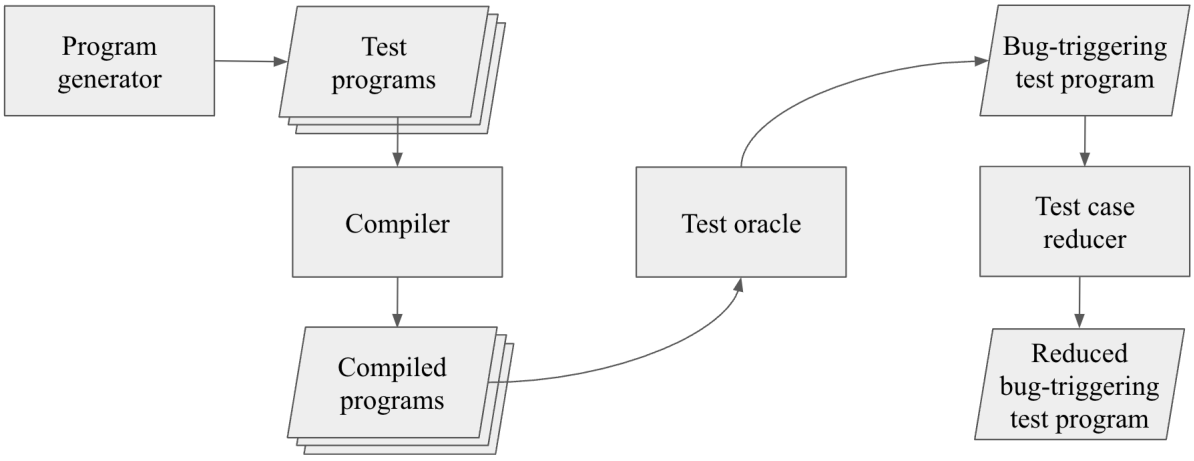


Figure 2: Compiler testing process

I discuss the key design features of compiler testing in Sections 2.3-2.7: test program generation and validity, test oracle construction, test diversity, test case reduction, and test evaluation.

2.3 Test program generation and validity

Program generation can take three approaches: create full programs from scratch, mutate existing full programs, or augment a skeleton program with code. The process typically includes some random component (fuzzing) to generate a large suite of tests in an automated way.

Ensuring validity of test programs is a key challenge in the production process. A test case is valid if it avoids language-specific undefined behaviour, for example maintaining type constraints throughout a strongly-typed program. An invalid program is of limited usefulness: at most it will test the initial analytical phases of the compiler. It may be useful for detecting some crashes, but it cannot detect miscompilation because the compiler will never reach the code generation stage for an invalid program.

Below I describe the main testing tools in terms of their approach to program generation and ensuring validity.

2.3.1 Program generation tools

Program generation typically involves using the AST of a language to compose and concatenate valid statements. Csmith [4] is a randomised test-case generation tool for C, which is

able to test for both compiler crashes and 'wrong-code' miscompilation bugs. Csmith has two key program generation features. First, the test-case programs are expressive: they exercise a variety of C language features including pointers, control-flow (`if/else, loops, goto`), and function calls. Second, test-case programs avoid undefined and unspecified behaviour through static and dynamic code-checking. This enables miscompilation testing, since any compilation 'wrong-code' bug is not caused by undefined behaviour. A limitation of Csmith is its complexity: extensive code is required to articulate the rules for avoiding undefined behaviour (the original tool is around 40k lines of C++).

YARPGen [5] is a C/C++ random test-case generator that is designed to improve upon Csmith in two areas. First, it adopts a simpler approach to ensuring program validity by using only static analysis during the code generation phase and avoiding dynamic checks. Second, it contains mechanisms to specifically target and stress-test the compiler's optimisation passes. Its key limitation is that the core tool does not support loops (unlike Csmith), thereby limiting the control flow complexity in the test-case programs.[3]

2.3.2 Program mutation tools

Program mutation takes an original program and edits it to produce a set of test programs. The nature of the mutation varies by tool. Equivalence Modulo Input (EMI) is a popular form of mutation testing. Its core idea is to take an existing real-world program and transform it by injecting or deleting code in such a way that the modified program is equivalent to the original program. Le, Afshari and Su [6] create a tool, Orion, that implements this approach by deleting 'dead code' that is never reached during runtime. The EMI approach is particularly good at testing a compiler's analysis and optimisation phases. This is because while the mutated programs are semantically equivalent to one another, they can differ significantly in terms of control flow structure, which in turn affects the optimisations that are implemented by the compiler.

Another advantage of program mutation exists with respect to ensuring program validity. The original program is free from undefined behaviour by design, and the mutations can also be designed to avoid undefined behaviour: Orion's mutations involve deleting dead code, which cannot introduce undefined behaviour. Donaldson et al. [7] create an EMI-based tool in which the code transformations are designed such that they are not capable of introducing undefined behaviour. Therefore there is no need for behaviour-checking.

CLsmith [8] and Orange4 [9] are program mutation tools that inject code rather than pruning it. CLsmith is developed from Csmith to test OpenCL compilers.[4] OpenCL programs typically do not contain any dynamically unreachable code, so the tool instead injects 'dead-by-construction' code, i.e. code which is constructed to be dynamically unreachable. Orange4 is a random test-case generator for C compilers, which repeatedly transforms an initial trivial program to produce new test cases. Orange4 injects features into reachable code (such as variable declarations) while ensuring that the overall program behaviour is unaffected. A key limitation of the Orange4 approach is the restrictions imposed on control flow. In order to reduce the complexity of ensuring no undefined behaviour is caused by code additions (given they are in 'live' code regions), loops are constructed such that the loop body will

[3] Note that some limited support for loops exists in the experimental parts of YARPGen's code, however this is still restricted.

[4] OpenCL is a multi-core programming model that allows programmers to utilise parallel processing capabilities of their machine by flagging data-parallelizable parts of a program.

execute at most once.

2.3.3 Skeleton fleshing

Fleshing is a more recent program generation method, and is most relevant for my project. The test case base is a skeleton program that requires a 'fleshing' step in which code is added to create an executable test program.

Klimis et al. [1] present a novel approach to compiler testing called 'control flow fleshing'. This approach involves generating interesting and valid control flow graphs for the SPIR-V GPU computing language. The graph creation step uses two methods: (1) mining CFGs from existing test suits, and (2) formal modelling of the SPIR-V language. They do not create a random generator, although they note that this is a further potential approach to CFG skeleton production. The graphs are then 'fleshed' to produce executable programs by writing a basic block of code for each graph vertex. CFG fleshing produces programs with interesting control flow and is therefore able to stress-test compiler optimisations that relate to control flow. This is a different approach to other tools, which focus on statement generation and tend to have limited control flow due to the added complexity of undefined beahviour checking. CFG validity checking is required because SPIR-V has strict restrictions on the structure of its control flow, although this is more straightforward than the undefined behaviour checking required for other tools such as Csmith.

Zhang, Sun and Su [10] develop a conceptually similar generation approach called Skeletal Program Enumeration (SPE). Rather than randomly generating or mutating large programs, they create a set of small syntactic skeletons based on the C/C++ AST with placeholders for variables. Placeholders in the skeletons are filled by exhaustively enumerating all possible variables $v \in V$. The SPE approach is limited because the resulting programs may contain undefined behaviour, which means only compiler crashes can be detected and not miscompilation bugs.

2.4 Test oracle construction

A test oracle is a mechanism for determining whether the compiler has exhibited any undesired behaviour. Compiler test oracle construction is challenging. In other areas of software testing it is typically straightforward to specify whether a test has passed or failed. For example, a unit test could assert whether a function has returned the expected value for the given inputs. However, detecting a miscompiled program requires a priori knowledge of the correct compilation. But the correct compilation can only be known by compiling the program. Therefore, a counterfactual for each compiled program is required that can be used to check for discrepancies. Strategies for constructing test oracles include differential testing and metamorphic testing.

2.4.1 Differential testing

Differential testing, introduced in McKeeman [11], involves executing a test case on two or more comparable compilation systems. The systems could be different compilers, different optimisations within the same compiler, or different versions of the same compiler. If the resulting output code differs or one compiler crashes, then the input test has exposed a potential bug.

Csmith and YARPGen both use differential testing on C and C++ compilers including different versions of GCC and LLVM. CLsmith uses both differential and metamorphic testing: the differential testing approach exploits the mutli-core nature of the language to compile on different OpenCL kernels and compare the results.

2.4.2 Metamorphic testing

Metamorphic testing uses metamorphic relations, which specify how changes in the input source code should affect the output code produced by the compiler [12]. The most commonly used relation is the equivalence relation. For example, a change to the input source code in a 'dead code' area should result in no change to the output code; the programs should be 'equivalent'. These relations can be used to form test oracles for mutated programs.

The advantage of metamorphic testing over differential testing is that it eliminates the requirement for multiple versions of the same compiler. Tools that use metamorphic testing include:

- Orion produces equivalent versions of the same program by pruning dead code. It checks equivalence by running the progams under some defined set of inputs and asserting that the output is equal across programs.
- Orange4 transforms programs by adding features in reachable parts of the code in a semantics-preserving way, for example adding a constant assignment.
- CLsmith injects 'dead-by-construction' code into programs. This is dynamically unreachable, for example inserting if-statements that will never evaluate to true at runtime.
- Spirv-fuzz applies a series of semantics-preserving transformations to a source program. The transformations include splitting code blocks, adding dead code, and adding loads / store statements.

Control flow fleshing [1] is of particular relevance to my project. Once CFGs are created as the program skeleton, a path through the graph is generated by a random traversal. The skeleton is 'fleshed' with code such that program execution should follow this path. The skeleton is also equipped with code to record the nodes that are actually visited during runtime. The test oracle is constructed by asserting whether the actual nodes visited during execution match the expected path. Any difference between the expected and actual path indicates a possible miscompilation bug.

2.5 Test diversity

Diversity of syntax, semantics, and control flow within and across test programs is necessary to test all parts of the compiler. Compilers have many configuration options including optimisation levels, source languages supported, and target platforms. The search space for bugs is therefore very large and difficult to search exhaustively. In addition, extending the search along one dimension can make another dimension more complex. For example, including a wider range of language features increases the difficulty of ensuring validity.

Groce [13] makes the case for diversity in software testing. Their core argument is that different types of fuzzers tend to be good at detecting different types of bugs. Therefore,

many different types of testing should be used, even if some of those methods are measurably 'worse' than others when compared at an aggregate level. The paper sets out three approaches to improving the level of testing diversity. First, varying test length can detect different types of bugs: for example, bugs relating to uninitialised values are more likely to be caught at shorter test lengths, whereas bugs relating to buffer overflows are more likely to be caught at longer test lengths. Second, swarm testing is a technique in which certain language features are either oversampled or omitted entirely from some tests. This decreases the diversity within a test, but increases the diversity across tests. The resulting set of behaviours across test cases is more diverse and can detect bugs that are triggered by repeated behaviour, or that can be 'suppressed' by some specific feature. Groce et al. [14] provide further detail on this method. Finally, ensemble methods are tools that employ multiple fuzzing techniques and therefore provide a diversity of approach.

McKeeman [11] also emphasises the importance of testing diversity by distinguishing between different 'quality levels' of tests, which refers to testing targeted at different levels of the compiler. For example, a 'low quality' test for a C compiler could be a random ASCII string: this is likely to pick up crash bugs in the syntactic level of the compiler, but is unlikely to probe the the optimisation and code generation level. Ideally, holistic testing should cover all levels.

Livinskii, Babokin and Regehr [5] also observe that fuzzers tend to reach a 'saturation point', after which they struggle to uncover further bugs. This is caused by the 'biases' within test-case generators that mean they are unable to explore and stress-test all aspects of the compiler. Therefore, there is always a need for novel and diverse testing approaches.

2.6 Test case reduction

Test case reduction is required to reduce a bug-triggering test case to a size that is conducive to debugging. Compiler developers request that bug reports are as specific as possible, while test programs can be arbitrarily long and complex. Test case reduction faces a similar challenge to program generation: it must avoid undefined behaviour.

Delta debugging [15] is an automated approach to test case reduction, given a passing test case and a corresponding failing test case. The delta debugging algorithm involves removing parts of the failing test case and checking whether the modified version still fails. This is repeated until a 'minimal test case' is found, which is defined as a test case in which removing any single input entity causes the failure to disappear. Once a passing test case and a minimal failing test case is obtained, isolating the difference between the cases helps to identify the source of the bug.

C-Reduce [16] is a test case reduction tool developed by the authors of CSmith. The paper presents three approaches to test case reduction. Two of the methods rely on Csmith to avoid undefined behaviour; however these methods tend to be unable to produce results that are small enough to be included in a bug report. The final method applies a series of reducing transformations to the bug-triggering test case and checks for program validity using third-party tools KCC and Frama-C. This approach is able to produce sufficiently small test cases.

Transformation-based program generation can provide a useful basis for test case reduction and deduplication of bug-triggering test cases. Donaldson et al. [7] design program transformations in the test case generation stage to be as simple and independent as possible.

Once a bug-triggering variant program is found, the test case is reduced by searching for a minimal subsequence of transformations that still trigger the bug. Deduplication is achieved by grouping similar transformations under 'types': bugs triggered by the same 'type' of transformation are considered more likely to be duplicates.

Ball and Horwitz [17] present an algorithm that can be used to reduce program control flow graphs. Program 'slicing' with respect to some program output involves removing the parts of the program that are unrelated; i.e. that can be removed without affecting the program calculation of that output. The method starts with the 'slicing point' (the program output of interest) and uses a Program Dependence Graph to trace all of the components of the graph that can influence that output. Other components are discarded. The challenge of unstructured control flow is that it is difficult to trace how statements such as jump and goto relate to specific outputs.

2.7 Test evaluation

Compilers typically have extensive test suites,[5] and are heavily used 'in the wild'. Therefore, one might expect that any bugs that could have a significant impact on real-world code would be quickly detected and reported by users, leaving only obscure bugs to be found by automated testing. But while undiscovered bugs may be obscure, they are not necessarily insignificant.

It is relatively easy to quantify the number of distinct bugs found by fuzzers. However, limited research exists on the real impact of such bugs. Marcozzi et al. [19] attempt to answer the question of whether fuzzing is useful by developing and implementing a methodology to evaluate the impact of historical fuzzer-found bugs on real-world code. The impact of a bug is defined as the extent to which it alters the semantics of a given application. The key findings of the paper are that bug-affected compiler code is typically encountered quite frequently when compiling real-word applications; however, the semantic impact of these bugs tends to be small in terms of the fraction of application functions affected. Interestingly, the semantic impact of user-found bugs was slightly lower than of fuzzer-found bugs, which suggests that bugs uncovered 'in the wild' are not necessarily more important than those identified by testing campaigns.

2.7.1 Coverage

Most papers summarising fuzzing campaigns report the headline number of bugs detected in each compiler under test. Code coverage metrics used to evaluate testing tools include incremental code coverage and path coverage:

- Incremental code coverage measures the percentage of compiler code that is executed during testing. The incremental effect of the testing tool is measured by comparing the code coverage of the compiler's own testing suite with and without the test tool. The delta is the incremental code coverage. For most tools, this tends to be fairly small,[6] which suggests that the tools are not covering significantly more code. However, despite this fact, these tools have been able to uncover a reasonable number of bugs. This suggests that incremental percentage covered may not be a useful measure for how the compiler is being stress-tested.

[5] For example, see the LLVM test suite [18]

[6] For example, YARPGen increases code coverage by around 0.01% to 2.83% across various coverage metrics

- Path coverage is an alternative metric [4, 5]. This metric is the number of paths through the compiler code executed as a proportion of all possible paths. It is difficult to measure because the number of paths through code is potentially infinite; neither paper attempts to quantify this.

2.7.2 Synthetic bug detection

Mutation analysis and testing are techniques used to evaluate and enhance the quality of test suites. Jia and Harman [20] provide a survey of the development of mutation testing. Mutants are synthetic bugs that are deliberately injected into a piece of source code. This can be done by applying mutation operators, which transform some of the code syntax. For example, a mutation operator could replace every arithmetic operator with a different arithmetic operator. The test suite is then run on the mutated code. A 'mutation score' is calculated, which is the number of mutants killed as a percentage of all mutants in the code. Test suites with a higher mutant score are considered to be more thorough. Andrews, Briand and Labiche [21] provide evidence that synthetic mutations are a useful proxy for real faults.

Madeyski et al. [22] describes a key challenge of mutation testing: the presence of equivalent mutants, which are semantically equivalent to the original code, and duplicated mutants, which are semantically equivalent to each other. Equivalent mutants increase the cost of mutation testing because tests are run on these mutants even though they are not 'interesting' in the sense of not representing distinct potential bugs. Hariri et al. [23] run mutation testing on LLVM to explore how compiler optimisations affect the cost and results of mutation testing. They find that the percentage of equivalent and duplicated mutants is higher at higher optimisation levels, which means that it is important to properly control for these when interpreting the overall mutation score of a test suite.

In addition to evaluating existing test suites, mutation testing can be used to enhance test suites by generating new tests. Mutants that are unkilled represent vulnerabilities: new tests can be created that are designed to kill those outstanding mutants, and added to the test suite. This is a pre-emptive form of testing that increases coverage before a real bug occurs. Mutants can also be used to assist debugging: automated tests can produce many failing test cases that are triggered by the same underlying bug. Holmes and Groce [24] present a method to identify which test cases are likely to be caused by the same underlying bug based on the observation that if two failures can be fixed by the same mutant, then it is likely that they are due to the same underlying bug.

3 FuzzFlesh design

In this section I describe the design of the FuzzFlesh tool. In Section 3.1 I set out an overview of the full toolchain and a brief description of each component. In Sections 3.2 - 3.6 I describe each component in further detail.

3.1 Overview

FuzzFlesh aims to test compilers for crash and wrong-code bugs. It generates test cases by creating CFG skeletons and 'fleshing' these into programs in a target language. It also creates a set of feasible paths through the graph, and a corresponding set of directions that the program should follow at each branch point in order to traverse a particular path. Each test program is equipped with code that records the actual path it takes at runtime. If there is any difference between the expected path and the actual path, then a miscompilation bug may be responsible. The comparison of the expected and actual path forms the test oracle, which is self-contained within every test case.

The graph- and path-generation parts of FuzzFlesh are language-independent. The outputs can be used to generate test programs in multiple target languages. The tool is thereby able to test a wide range of compiler toolchains.

Figure 3 shows the components of the FuzzFlesh toolchain: (1) Graph Generator, (2) Path Generator, (3) Flesher, and (4) Runner.

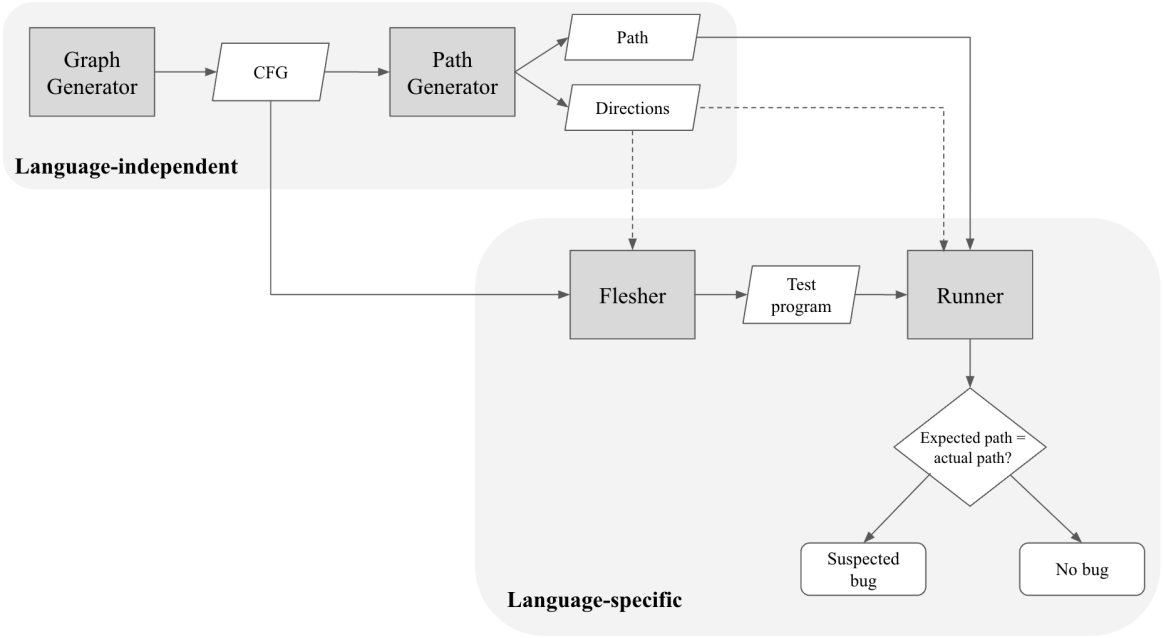


Figure 3: Proposed toolchain

The key components of FuzzFlesh are summarised below, with additional detail on each step in Sections 3.2 - 3.6

1. Generate graph: Generate a set of interesting and valid CFGs. An example CFG is shown in Figure 4, with a corresponding C program in Listing 1.

CFG definition: A CFG is a directed, rooted graph G with nodes N and edges E, where each n ∈ N represents a 'basic block' of a program, and each e ∈ E represents a possible path between nodes that the control flow of a program could take.

2. Generate a set of random (valid) paths through the CFG with a corresponding directions array. For example, a path through the illustrative example could be: [0, 1, 1, 1, 3] with directions array [1, 1, 1, 0]. The program control flow begins at node 0, evaluates the first condition to true and travels to node 1. It then evaluates the while condition to true and re-visits node 1 a further two times. Finally, the while condition evaluates to true and the control flow moves to node 3, which is the exit node.

3. Create a 'fleshed' program for each CFG. Fleshing refers to the translation from a graph structure to a program in a target language. This involves writing blocks of code for each vertex such that the resulting program follows the structure of the control flow graph. Each block contains the following components: the relevant branching condition (if any) for that block, with links to successor vertices; code to read a 'directions' array that guides the runtime control flow; and code to record each vertex visited during runtime in an 'output' array that is used during testing.

4. Wrap each pair (path, fleshed CFG) in a program that compares the 'expected path' (generated in step 2) with the 'actual path' recorded at runtime. This comparison provides the test oracle: any discrepancy between actual and expected path indicates a possible miscompilation error. For example, if we run our first random path and receive output [0, 2, 3] then a possible bug is flagged.

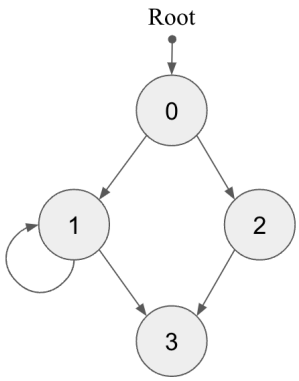


Figure 4: Example CFG

```
int main(){
        // node 0
        if(condition){
            // node 1
            while(condition) {}
        }
        else{
            //node 2
        }
        // node 3
        return 0;
}
```

Listing 1: Example CFG pseudocode

The graph and path generation steps are language-independent: the output is a set of data structures that can be used to create programs in a variety of languages. The program fleshing and test running steps are language-specific. In this project I have implemented four distinct language backends, which I discuss further in Section 4.

3.2 Graph generation

The graph generator is the first component of the fleshing tool. Its aim is to create a CFG that can be used as the basis for a program. The output is a directed graph data structure that contains nodes and directed edges.[7] Ideally, the set of generated graphs should be 'interesting' in that they cover a wide range of possible program structures and are able to trigger control-flow-based optimisations within compilers. The graph generator accepts a set of input parameters that configure the resulting graph shape. These are summarised in Table 1.

Parameter	Description	Type	Default
Graph size	Number of nodes to include in the graph	int	250
Generation approach	Generation approach to use	int	1
Annotations	Whether or not to include annotations	bool	True
N annotations	Number of annotations to include	int	$0.2 * size$
Minimum successors	Minimum number of successor nodes to add to each node when growing the graph	int	1
Maximum successors	Maximum number of successor nodes to add to each node when growing the graph	int	3

Table 1: Graph generation input parameters

I have implemented two graph generation approaches, each of which is adapted from graph generators provided in Python's NetworkX package [25]:

- Approach 1: Growing network generation. The graph generator grows a graph using a breadth-first algorithm. Each node is given a random number of successors, from 0 up to a maximum successor parameter. Once the number of nodes added to the graph is equal to the graph size, graph generation stops. This constructs a wide tree graph that contains a path from the entry node to at least one exit node, but does not have any jumps beyond immediate child nodes. To enhance the graph shape, it can be optionally annotated by adding edges to a randomly chosen number of nodes in the graph. The edges may jump forwards (from parent to a node further away from the root), backwards (from a parent to a node closer to the root), across the graph to a different branch, or to the node itself. Finally, exit nodes are optionally added to a randomly selected sample of nodes in the graph. This approach is adapted from Krapivsky and Redner [26]. The main adaptation is the addition of annotated edges, which I include to introduce cycles into the CFG so that important program constructs such as loops and branches are represented.
- Approach 2: Erdős-Rényi generation: The graph generator creates a set of nodes according to the graph size parameter. It then randomly connects them via directed

[7] The graph data structure uses Python's NetworkX package, which can be easily converted to/from .dot format.

edges. A single node is randomly selected as the root node, such that the graph has a single entry point. In addition, exit nodes are randomly added to a proportion of the nodes. This approach is adapted from Erdős and Rényi [27] approach to generating random graphs. The main adaptation is simplifying the node-connecting approach in order to generate the graphs more quickly.

Figure 5 shows illustrative examples of CFGs produced by each approach that have the same number of nodes. The blue edges in Approach 1 show the 'annotations' that are added to the underlying tree structure. Approach 1 produces graphs that contain longer chains of connected nodes, whereas Approach 2 produces more 'randomly' shaped graphs, typically with a smaller number of distinct nodes on any potential path. Approach 2 also produces graphs that contain more unreachable sub-graphs, because the graph is not grown out from the root node. For example, Approach 2 may create graphs where some nodes cannot be reached from the root node, whereas under Approach 1 every node can be reached from the root node under some set of directions. This difference in graph shapes may result in different compiler optimisations, for example to remove unreachable ('dead') code.

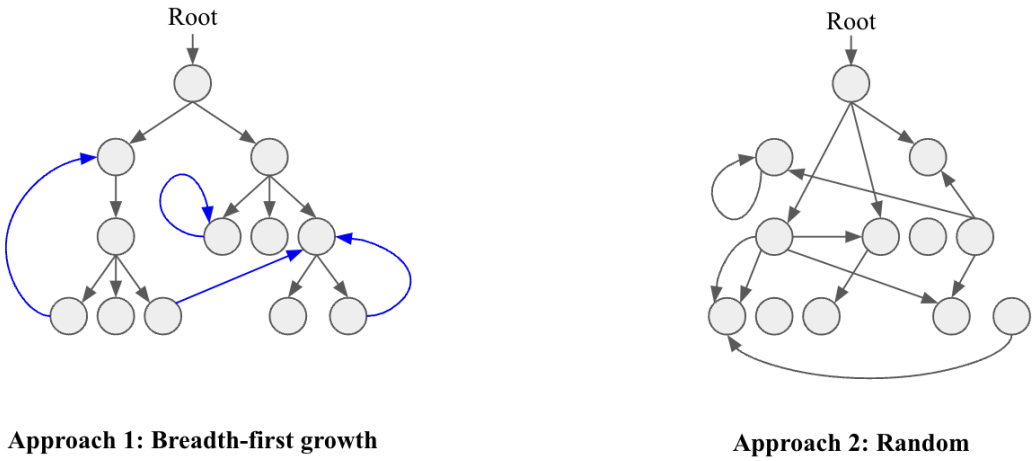


Figure 5: Illustrative examples of CFGs produced by each approach

The generated graphs must be a valid basis for programs, which is achieved by ensuring that they meet the following conditions:

- There must exist a single entry point to the graph. Most languages require that programs only have a single entry point.
- There must exist a path from the entry node to an exit node. Exit nodes are defined as having no child nodes. If no such path exists, then every execution of the graph will feature an infinite loop.
- Some languages have additional restrictions on control flow. For example, Java does not allow irreducible control flow. In this project, all language backends allow irreducible control flow, so the CFGs produced are valid. However, in future implementations it could be necessary to include an additional filtering step to remove CFGs that are invalid for a particular language.

An alternative graph generation approach would be to harvest them from the wild. Many tools exist to extract CFGs from existing programs.[8] Most of these tools output CFGs as a .dot

[8]For example Khronos provides a `spirv-cfg` tool that converts SPIR-V programs to GraphViz dot files, and

file, which is readable by the NetworkX Python package used to manipulate graph objects in this project. FuzzFlesh could be extended to take such graphs as inputs and either generate programs directly from them, or mutate them to form additional test cases.

The advantages of using real graphs as a base is that they may be more likely to produce test cases that are close to 'realistic' programs, and therefore may discover bugs that are more likely to be triggered in the wild. Combining graphs from different sources, or repeating them, would also provide a greater diversity of test cases. However, since the original program (presumably) did not trigger a bug, CFGs directly extracted from real programs may be less likely to find bugs than those that are randomly generated.

3.3 Path generation

The path generator performs a random traversal of each graph. It records the sequence of nodes visited on the path, and a corresponding directions array. Multiple paths can be generated for each graph, and they can be configured by a maximum length input parameter. Once this length is reached, the path generation algorithm identifies the shortest path to an exit node and adds this to the path. This stage of the toolchain is language-independent: paths are output as a data structure containing the path and the directions array.

An improvement to the path generation component would be to randomly instruct the path generator to identify opportunities to create many iterations of loops, for example where a node has an edge that returns to itself. This could be useful for triggering bugs that require repeated execution of the same action.

3.4 Program fleshing

At this stage, the toolchain becomes language-specific. The input is a graph data structure, and optionally a path. The output is a program in a particular language, which is designed to write the IDs of each node visited at runtime to an actual output array. This can be compared with the expected output to provide the test oracle.

The program output by this step is a test program with a single function containing the fleshed CFG. The function accepts input parameters for the directions array and an empty output array, which is filled during the program. The function is called from a separate wrapper program that passes the directions array and evaluates the output (discussed further in Section 3.5).

The general framework for program fleshing involves traversing the graph and 'fleshing' each node based on its characteristics. First, the start of the program is fleshed with the necessary code to begin the program, including the declaration of the function and data structures for the directions and output arrays. Next, the flesher traverses the CFG and fleshes each node with code that records the node ID in the output array so that the actual path taken through the program at runtime is recorded. In addition, the flesher checks the number of childeren that each node has and fleshes it as follows:

- Unconditional nodes with one child pass control directly to that child with the language-specific equivalent of a `goto` or `jump` statement.
- Conditional nodes with two or more children are fleshed with code to read the current direction from the directions array and evaluate a conditional statement based on this

LLVM provides an equivalent `dot-cfg` to extract CFGs from LLVM IR programs.

direction. Control is passed to the child node that is associated with the given direction. All of the languages I have implemented use branching `if/else` statements for two children, and `switch` statements for more than two children.

- Exit nodes with no children contain a return instruction that causes control to exit the function and return to the calling wrapper program. Test programs may have multiple exit nodes.

Fleshing is language-specific. I have therefore implemented it as distinct `ProgramGenerator` class for each language backend. However, all of these implementations share a common structure (described above). Within this program generation structure, each of the language backends contains several language-specific implementation options. These are described in Section 4.

As a future development of the tool, it would be possible to extract the overall structure as an abstract `ProgramGenerator` class containing the high-level algorithm for traversing the graph and fleshing each node based on the number of children it has. The advantage of this would be to provide a clear structure for future language implementations to follow. Within this structure, it would be possible to implement further language-specific program features. For example, to exercise a wider range of language features, each conditional node could be fleshed with either a `switch` or a set of `if/else` statements (rather than the current set up, in which the instruction is set based on the number of child nodes). Small language snippets featuring language-specific operators could also be added to nodes within this structure. However, it is possible that imposing an overall structure in this way reduces the flexibility of the program generation approach. For example, adding function calls would require a more complex approach to flesh the overall program and each called function.

3.5 Test runner

The test runner runs the test and evaluates the test oracle to determine whether a possible compiler bug is detected. This step is language-specific, and in some cases varies by compiler toolchain. For example, some languages have both static and JIT compilers, which require different compilation processes. These are discussed further in Section 5.

The test runner includes linking the generated test case program to a wrapper program, which performs the following functions:

- Reads the expected output and directions array from an input file, and declares the required data structures including the actual output array.
- Calls the test case function and passes the directions array and empty actual output array as parameters. The way in which the function is called can vary by compiler toolchain, for example the alternative implementations for static and JIT compiler are discussed in Section 4.
- Compares the expected and actual output arrays after the test case function has returned, and determines whether they are equal. Any necessary clean-up is performed here, for example memory deallocation in C++.

Due to the language- and compiler-specific nature of this step, there is unlikely to be any further potential for useful abstraction. It would theoretically be possible to create a meta-tool that took many options and deployed tests based on these. This could increase the overall throughput of tests. However, engineering fuzzing tools in this way tends to make them less

flexible in terms of adding new options. I have therefore constructed and maintained distinct runners for each language.

3.6 Test case reduction

I have not implemented an automated test case reducer in this project due to time constraints. However, in the future this would be an important feature to make the simplification and triage of bug-triggering test cases easier. I conducted manual test reduction for the two Java decompiler bugs found. Below I set out a systematic approach to test case reduction that is informed by my manual efforts.

In principle, a significant part of test case reduction could be implemented at the language-independent graph and path level. The overall test case reduction approach involves taking a bug-inducing test case, removing parts of it, and checking whether it is still bug-inducing. If it is, then the smaller test case is put through the same reducer to test whether it can be reduced further. If it is not, then the original test case is randomly reduced again to test whether a different reduction is possible.

As an illustrative example, suppose we have a program based on the CFG in Figure 6. The expected path through the graph is [0, 2, 1, 1, 1, 4, 6, 4, 6, 7], but the actual path at runtime is [0, 2, 4, 6, 7], which indicates that a bug is present. The program could be reduced in the following ways:

- Graph reduction: Segments of the graph could be removed, and the program refleshed. This could be guided by information from the program. Figure 7a shows the illustrative example where the graph has been reduced by removing all nodes that are never visited, and all edges from remaining nodes to the removed nodes. If this graph is no longer bug-triggering, then nodes could instead be removed incrementally from the original graph, starting with exit nodes. Alternatively, non-visited edges could be removed rather than non-visited nodes. When I manually reduced the Java bytecode test cases, I found that a minimum set of specific nodes and edges were required to trigger the bug, not all of which were visited at runtime.
- Path reduction: Parts of the path could be removed by identifying the divergence point in the path and simplifying the preceding or subsequent path. The simplification could be implemented by finding the shortest route from the root to the divergence point, or from the divergence point to an exit node. Figure 7b shows the example graph under path reduction. Path divergence occurs at path[2], and so the path reduction algorithm would retain path [0, 2, 1] and find the shortest route from path[3] (node 1) to the exit. This would give the full path: [0, 2, 1, 4, 5]. The motivation for retaining the full graph is that it may be necessary to trigger the compiler optimisation that contained the bug.
- Graph and path reduction: Parts of the graph *and* path could be removed. In the previous example, the path could be truncated *and* all non-visited nodes visited could be removed. This substantial type of reduction may be less likely to still be bug-triggering, however it would result in a faster reduction if it was successful.

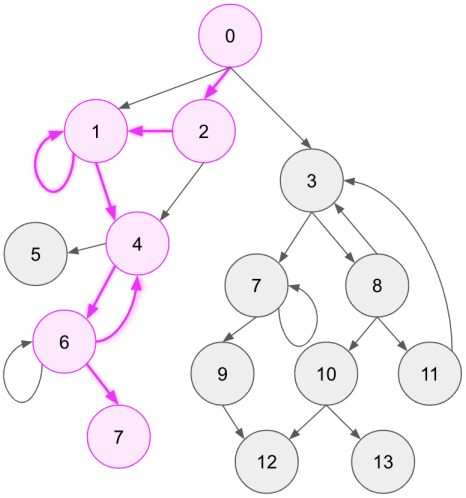
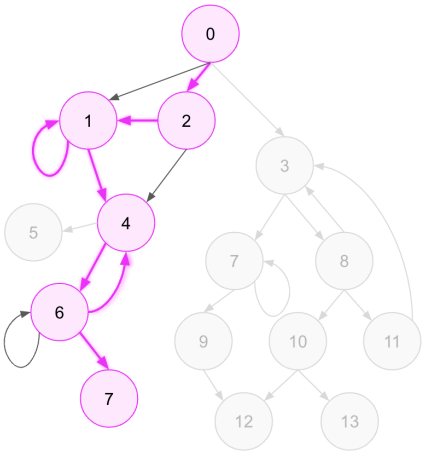
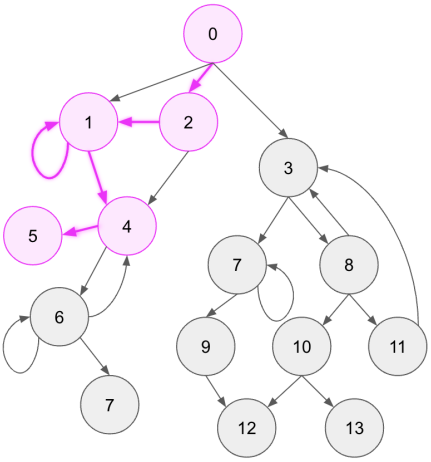


Figure 6: Example CFG
Expected path: [0, 2, 1, 1, 1, 4, 6, 4, 6, 7]
Actual path: [0, 2, 4, 6, 7]



(a) Reduce graph
Expected path: [0, 2, 1, 1, 1, 4, 6, 4, 6, 7]
Actual path: [0, 2, 4, 6, 7]



(b) Reduce path
Expected path: [0, 2, 1, 4, 5]
Actual path: [0, 2, 4, 6, 7]

Figure 7: Reduction approaches

4 Program fleshing

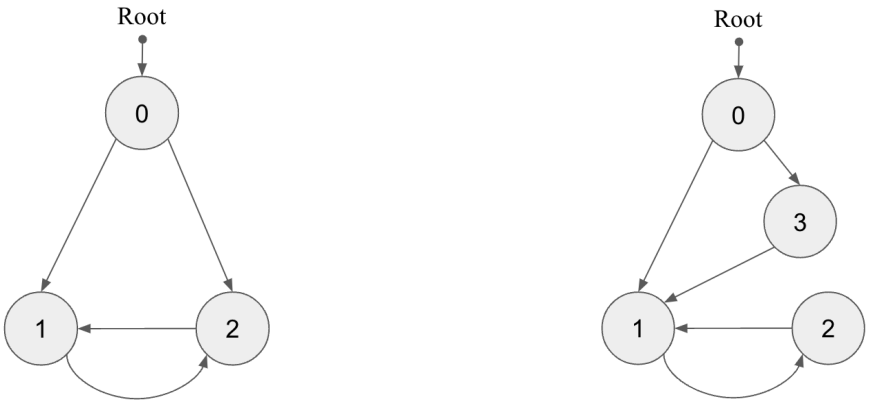
Section 4.1 gives an overview of the key features of the Flesher component of each language backend. First, I discuss the motivation for the choice of languages used in this project, and the general approach to the program generation within the FuzzFlesh toolchain. In Sections 4.2 - 4.5 I discuss the approach for each language in detail and show a simple example 'fleshed' node in each language.

4.1 Language selection

I have implemented four language backends for FuzzFlesh: LLVM IR, Java bytecode, Common Intermediate Language (CIL), and C. This demonstrates that the general FuzzFlesh framework can be successfully applied to a range of languages. The motivation for selecting languages is that they are all widely used languages, they cover a range of compiler toolchain options, and they include low-level (LLVM IR, Java bytecode, CIL) and high-level (C) languages.

It is beyond the scope of this project to give a formal definition of control flow within languages, however I briefly discuss here because it is relevant to several of the languages and compiler toolchains that I test. The CFG fleshing approach was previously successfully used to identify bugs in SPIR-V compilers. SPIR-V is a structured control flow graph based language. Constructs such as branches and loops are represented using basic blocks of code and are subject to a strict set of rules about nesting, entry, and exit. These rules are detailed in Section 2.1.1 of the Khronos SPIR-V language specification [28].

Structured languages do not allow irreducible control flow. Intuitively, irreducible occurs when a CFG exhibits a cycle that cannot be reduced to a single-entry, single-exit structure. It is possible to transform irreducible CFGs to reducible CFGs. This is demonstrated in Figure 8. However, this can result in programs that are very long [29]. In contrast, unstructured control flow graph languages (such as LLVM IR), do not impose rules about the path that control flow can take; they support unrestricted `goto` statements that can arbitrarily transfer control to any part of the program.



(a) Irreducible CFG **(b)** Reducible CFG

Figure 8: Example of irreducible-to-reducible CFG conversion adapted from [29]

Representing structured constructs as basic blocks in accordance with the rules in SPIR-V

was previously difficult because the rules were complex and unclear [1]. This may have resulted in compiler implementations that were not fully compliant, and therefore suffered from bugs.

LLVM IR, CIL, Java bytecode, and C all allow irreducible control flow. As a result, the FuzzFlesh language backends do not need to perform validity checks on the CFG inputs. In other language backends it may be necessary to implement a validity checker to ensure that valid programs are created.

4.1.1 Program options

Each language backend is implemented as a distinct `ProgramGenerator` class, as discussed in Section 3.4. These generators share a similar high-level structure, which in theory could be extracted as an abstract `ProgramGenerator` parent class. Within the fleshing structure, there are several language-specific implementation options. These are discussed in more detail in the language sections below.

One important option that is conceptually common across all languages is the treatment of the directions array. There are two conceptual options (each with multiple language-specific implementations):

- Statically unknown: The directions array can be passed to the test case function at runtime as a parameter from the wrapper program. In this case, the test case is fleshed to accept the directions array and read directions from it, but these directions are not known at compile-time. This creates a program in which every node in the graph is potentially dynamically reachable at run-time, given suitable inputs.

- Statically known: Alternatively, the directions array can be hard-coded into the function during the fleshing step. The directions are thereby known to the compiler at compile-time, and can be used to perform optimisations. For example, the compiler may be able to use this information to infer that some parts of the program are unreachable ('dead') and remove them as part of a dead code elimination optimisation.

These approaches may result in different compiler optimisations: the more information the compiler has at compile-time, the more optimisations it is likely to be able to make.

Further work could explore additional options for the treatment of the directions array, including node-specific directions arrays. In theory this gives the compiler the same amount of information for optimisations, but the compiler may find it easier to make inferences about each node since the information for that node is more concentrated. Additionally, a mixture of statically known and unknown directions could be used.

Beyond the directions array, an additional option for fleshing would be to randomly include additional language-specific features within each node. To minimise the requirement for undefined-behaviour checking, these language snippets could be designed to be self-contained and non-conflicting. However, any extensive language-specific content increases the development time, which may undermine the quick-implementation advantage of FuzzFlesh.

4.2 LLVM IR

LLVM Intermediate Representation (LLVM IR) [30] is a platform-independent intermediate representation language that is used throughout the LLVM compiler infrastructure project. LLVM is used as the compiler backend for many compiler toolchains, including `clang`/`clang++`, `rustc`, and `swiftc`.

LLVM IR has two important language characteristics in common with SPIR-V, which is the language that the CFG fleshing method was previously successfully applied to in Klimis et al.[1]

- LLVM IR and SPIR-V are **control flow graph** based languages: programs are represented as a series of non-branching basic blocks containing instructions. These basic blocks are connected by conditions, which form the edges of the graph.
- LLVM IR and SPIR-V use **Static Single Assignment (SSA)** form, which means that every variable is assigned exactly once. This simplifies variable properties: they cannot change value. As a result, SSA form simplifies optimisations that involve tracking variable dependencies through the program, such as dead code elimination and constant propagation. For example, constant propagation involves replacing variables with constant values in cases where these are known at compile-time. To illustrate: suppose a program initialises `int x1 = 3` and later initialises `int x5 = x1`. To apply constant propagation to `x5`, the compiler must check that `x1` is not reassigned in the intervening code. Under SSA, the value of `x1` cannot be changed. Therefore, no further checks are required.

LLVM IR was selected as the first language backend implementation for FuzzFlesh because it shares these language characteristics with SPIR-V. In addition, it is a widely-used language with an accessible compiler and associated tooling.

The main program fleshing option for LLVM IR test programs is the treatment of the directions array. I have implemented two options. First, the directions array is dynamically passed to the program at runtime. The advantage of this approach is that test throughput is higher: for n graphs and m paths, it is only necessary to compile n programs rather than $n \cdot m$ programs. However, fewer compiler optimisations are likely to be possible since the compiler does not have any information about which branches may be taken at runtime, nor which code is unreachable.

Second, the directions array is written in to the program at the fleshing stage, and is therefore known at compile-time. This gives the compiler more information to support optimisations. However, the test throughput is reduced due to additional compilation time since every (graph, path) combination must be statically compiled (as opposed to compiling a single graph and running it m times, once for each path).

Example fleshed node

Listing 2 shows the fleshed node 1 from the example CFG shown in Figure 4. Note that each variable is assigned only once under SSA form, which results in fairly verbose code. First, the node ID is stored in the output array. The correct position in the output array is tracked by a counter variable that is initialised to 0 at the beginning of the program and incremented after each write. Next, because this is a branching node, the directions are read from the `dirs` array using an index variable `dir_counter`, which is incremented after the direction is read. Finally, the direction is evaluated to give the branch condition. If the condition is true

(i.e. the current direction is equal to 0), then control returns to the beginning of node 1. Otherwise, control passes to node 3.

```
1:

; store node label in output array
%index_1 = load i32, i32* %counter
%output_1 = load i32*, i32** %output
%output_1_ptr = getelementptr inbounds i32, i32* %output_1, i32 %index_1
store i32 1, i32* %output_1_ptr

; increment counter
%temp_1_1 = add i32 %index_1, 1
store i32 %temp_1_1, i32* %counter

; get directions for node
%index_dir_1 = load i32, i32* %dir_counter
%dir_1 = sext i32 %index_dir_1 to i64
%dir_1_ptr = getelementptr inbounds [1 x i32], [1 x i32]* %dirs, i64 0, i64 %
    dir_1
%dir_1_value = load i32, i32* %dir_1_ptr

; increment directions counter
%temp_1_2 = add i32 %index_dir_1, 1
store i32 %temp_1_2, i32* %dir_counter

; branch
%condition_1 = icmp eq i32 %dir_1_value, 0
br i1 %condition_1, label %1, label %3
```

Listing 2: LLVM IR branching node with two children

4.3 Java bytecode

Java bytecode is the JVM platform-independent instruction set that is used within class files. Platform-specific JVMs load, verfiy, and execute bytecode class files using a JIT compiler. Java bytecode is a stack-based language, unlike LLVM IR or SPIR-V. While Java does not allow irreducible control flow, Java bytecode does. The bytecode instruction set [31] includes an unrestricted `goto` command that can be used to transfer control to anywhere in the program. This difference in restrictions allows Java bytecode more freedom to express and optimise code. For example, [32] contains an example where the `javac` compiler produces bytecode that 'may be surprising and counterintuitive to the Java developer... goto opcodes are used to help control flow'.

There is no native support for writing textual Java bytecode. Instead, I use Jasmin [33], which is an assembler for the JVM that converts textual bytecode to class files. Other frameworks for writing Java bytecode exist, however Jasmin appears to be the most minimal: its instruction set is identical to the JVM instruction set, and it does not provide any helper functionality to produce bytecode. I chose Jasmin as the simplest approach for the project to minimise the risk that my programs test the third-party assembler, rather than the JIT compiler.

There are two sets of options within the Java bytecode program flesher: the treatment of the directions array, and the way in which the test case is called from the wrapper.

Directions

The directions array can either be dynamically passed to the test case as a function parameter, or it can be hard-coded into the test program. If the directions array is unknown, then it is possible that the dynamic optimiser will not have enough information to perform some optimisations (since the next time the function is called, it is theoretically possible that a different directions array is passed).

Alternatively, if the directions array is known, this gives the JIT compiler more information to make inferences about the program structure, which can in turn enable a wider range of optimisations. Similarly to LLVM IR, the disadvantage of hard-coding the directions array within each test case is an increase in the number of programs that must be compiled to class files. However, a longer compile-time is less problematic for Java bytecode because optimisations are not performed at compile-time, which means the absolute time taken is shorter than for LLVM IR.

Wrapper

The second program fleshing option for Java bytecode is the way in which the test case is called from the wrapper program. I have implemented two approaches:

- Dynamic class loading: In this approach, I create a `TestCaseI` interface that includes a `testCase` function, which is implemented by all test case programs generated by the flesher. The Java wrapper program creates an instance of `TestCaseI` and uses reflection to load specific named test cases at runtime. The advantage of this approach is that the wrapper must only be compiled once. However, the use of reflection may limit the ability of the JIT compiler to perform optimisations.
- Static test case compilation: In this approach, I create a `TestCase` object in the wrapper and call a `testCase` member function. I compile the wrapper and test case separately for each test case by adding the test case to the Java classpath. This results in a longer compile time than the first approach since the wrapper must be compiled multiple times, however as noted above this does not have a large absolute impact due to the relatively fast Java compile time.

Example fleshed node

Listing 3 shows the fleshed node 1 from the example CFG shown in Figure 4. Java bytecode uses stack operations, and stores the output and directions arrays in local variables. For example, the output array is stored in local variable 2, and the output index is stored in local variable 3. The instructions `aload_2` and `iload_3` load the output array reference and index onto the stack; `sipush 1` and `iastore` 1 write the node number 1 to the output array. This is how the program keeps a record of which nodes were visited at runtime.

```
1  block_1:
2
3  ; store node label in output array
4  aload_2
5  iload_3
6  sipush 1
7  iastore
8
9  ; increment counter
10 iinc 3 1
11
12 ; get directions for node
13 aload_1
```

```
14 iload 4
15 iaload
16
17 ; increment directions counter
18 iinc 4 1
19
20 ; branch
21 ifeq block_1
22 goto block_3
```

Listing 3: Java bytecode branching node with two children

4.4 CIL

CIL, also known as MSIL, is a platform-independent intermediate representation language used within the Microsoft .NET framework to represent C#, F#, and Visual Basic. It is a stack-based language with a similar instruction set to Java bytecode. However, unlike Java, both CIL and C# allow the use of irreducible control flow.

The main program fleshing option is whether the directions array is passed to the program as a parameter or hard-coded within the test function body. The conceptual issues are similar to those found in Java bytecode (Section4.3) due to the similarity of the languages. However, in practice I found that CIL compilation took a long time due to the startup time for the .NET runtime environment. As a result, the additional compilation time required by the hard-coded directions array had a material impact on the test throughput.

Example fleshed node

Listing 4 shows the fleshed node 1 from the example CFG shown in Figure 4. CIL uses stack operations similarly to Java bytecode. Branching works by pushing the current direction onto the stack (in lines 18 - 20), and evaluating whether it is equal to zero (in lines 29 - 30). If the condition evaluates to false, then program control moves to node 3. Otherwise, control cycles back to the beginning of node 1.

```
1  block_1:
2
3  // store node label in output array
4  ldarg.2
5  ldind.ref
6  ldloc.1
7  ldc.i4 1
8  stelem.i4
9
10 // increment output counter
11 ldloc.1
12 ldc.i4.1
13 add
14 stloc.1
15
16
17 // push node direction
18 ldarg.1
19 ldloc.0
20 ldelem.i4
21
22 // increment directions counter
23 ldloc.0
```

```
24 ldc.i4.1
25 add
26 stloc.0
27
28 // branch
29 ldc.i4.0
30 ceq
31 brfalse block_2
32 br block_1
```

Listing 4: CIL branching node with two children

4.5 C

C is a high-level language. It allows irreducible control flow, which can be implemented via arbitrary `goto` statements. This language backend was the final implementation within the project. It took approximately only 6 hours to code and test, including several fleshing options. This demonstrates the ease with which the FuzzFlesh can be extended to additional languages and toolchains.

The main fleshing option in the C program is the treatment of the directions array. The additional language features available in C relative to low-level languages provide additional implementation options.

Similarly to the low-level languages, I implement a directions array that is passed to the test program as a parameter. In this context, it is unknown to the static compiler at compile-time, which limits the extent of possible optimisations.

There are multiple implementations of hard-coding the directions array into the test program. I use two slightly different options:

- Directions are known at compile time: I use an integer array that is initialised at the beginning of the program with the directions array values. This is analogous to the approach used in the low-level language implementations.
- Directions are known and unchangeable at compile time: I also implement a constant integer array, which provides the compiler with the information that this array will not be changed throughout the course of the program. From manual inspection of C-to-LLVM IR translation, `const` variables are sometimes translated differently and may therefore enable different levels of optimisation.

Example fleshed node

Listing 5 shows the fleshed node 1 from the example CFG shown in Figure 4. Since C is a high level language it is much easier to understand the statements, but the format is the same as the previous examples. The node ID is written to the output array at the current index, which is then incremented. The current direction is evaluated: if it is true, then control returns to the beginning of block 1. Otherwise, control moves to node 3.

```
1 block_1:
2
3 // store visited node to output array
4 actual_output[out_counter++] = 1;
5
6 if(directions[dir_counter++] == 0){
7     goto block_1;
```

```
8 }
9 else{
10     goto block_3;
11 }
```

Listing 5: C branching node with two children

5 Compiler toolchains

The Runner component of the toolchain calls the fleshed program from a wrapper program and executes it for a range of compiler toolchains.

I have tested most of the languages on multiple compiler toolchains. These can be categorised as static compilers, JIT compilers, and decompilers.

- Static compilers compile a program to machine code in full before execution. Optimisations are applied at this stage based on analysis of the program structure. Optimisations aim to increase the performance of a program, for example with respect to execution speed or program size. For example, a common optimisation is the removal of 'dead code' that can never be reached during program execution, e.g. anything that falls within a `while(false){...}` statement. This reduces the size of the code, and can simplify the program to enable more advanced optimisations.
- JIT compilers compile code to machine code during execution. Implementations vary, but optimisations are typically performed at runtime. The advantage of this is that optimisations can be based on runtime profiling information. To trigger optimisations, code must usually be executed multiple times so that profiling information is available. I therefore include a loop within the wrapper program that runs each test multiple times for JIT compiler testing, where the number of loop iterations is passed as a test parameter.
- Decompilers are available for many languages. The purpose of these tools is to reconstruct source code corresponding to compiled code. This can be useful for detecting security vulnerabilities, or program analysis in cases where the source code is unavailable. Decompilation is a difficult process because there can be multiple valid ways to decompile a given piece of code. For example, a low-level `switch` statement could be represented by a high-level `switch` statement or mutliple `if/else` statements. In addition, compiled code can be heavily altered through compiler optimisations, and information can be lost during compilation, for example variable names and code structure. This can make it difficult for the decompiler to reconstruct the original code.

5.1 LLVM IR

The LLVM project includes `clang/clang++`, which is the C/C++ compiler frontend for the LLVM compiler toolchain. I therefore use a wrapper program written in C++ to call the LLVM IR test case function. The motivation for using a different language for the wrapper is that C++ provides useful abstractions for reading and writing from/to files.

I have tested two LLVM compiler toolchains: a static compiler and a JIT compiler.

5.1.1 Static compilers

LLVM IR can be statically compiled using the LLVM toolchain. Optimisations are applied to the LLVM IR at this stage via the LLVM `opt` tool [34]. Examples of optimisations include `simplifycfg`, which eliminates 'dead' code (that is unreachable during runtime) and merges basic blocks that only have one predecessor or successor [35]. In addition, LLVM defines several pre-set optimisation levels: `O1`, `O2`, `O3`, `Oz`, and `Os`. Each of these involves applying a different set of optimisations. FuzzFlesh can either accept a specific optimisation, or choose a random default optimisation level.

5.1.2 JIT compilers

Several JIT compilers are available for LLVM IR, including the LLVM toolchain `lli` tool and GraalVM LLVM.[9] GraalVM LLVM uses a similar conceptual apporach to the Java HotSpot compiler which is described in detail in Section 5.2.1: it initially interprets LLVM bitcode, collects profiling information, and then dynamically compiles and optimises parts of the program that are 'hot'. The C++ wrapper for this toolchain therefore calls the LLVM IR test case function repeatedly in order to generate profiling information.

The preferred file format for GraalVM LLVM is a native executable with embedded bitcode.[10] GraalVM LLVM comes with a pre-built LLVM toolchain including `clang`. I found that it was necessary to produce the executables using this toolchain in order for GraalVM to accept the files without errors. This compilation process took a relatively long time to execute, and therefore limited the testing througput for this toolchain. I discuss this further in Section 6.1.

The C++ wrapper for this toolchain calls the LLVM IR test case function repeatedly in order to generate profiling information.

5.2 Java bytecode

The bytecode wrapper is implemented in Java. It calls the test case function repeatedly so that the JIT compiler has enough information to make profile-based optimisations. The number of function repetitions is an input parameter to the program.

5.2.1 JIT compilers

I test the HotSpot JIT compiler, which works as follows:[11] initially, when a program is run it is interpreted rather than compiled. Although interpretation is slower than executing compiled code, it is the quickest way to start up the program execution, and is suitable for short programs that only execute once. If the program runs for some time, then the compiler gathers profiling information to identify 'hot spots': parts of the program that are repeatedly executed. These hot spots are compiled using a tiered approach that implements increasingly aggressive optimisations based on real-time profiling data. Compiled code is cached so that it can be used in future function calls, while infrequently-used parts of the code continue to be interpreted. Optimisations can include method inlining and loop unrolling. Interestingly, the compiler may also de-optimise and re-optimise code. This can occur if the compiler has performed an optimisation based on an assumption that subsequently transpires to be false, for example if the program behaviour changes.

I also test the GraalVM JIT compiler, which operates in a similar way to the HotSpot JIT compiler [37].

[9] The GraalVM LLVM compiler is also called `lli`, but has slightly different functionality to the LLVM `lli` tool

[10] Although GraalVM documentation states that the GraalVM LLVM runtime can execute plain bitcode files, I found that it would not execute the bitcode files produced by FuzzFlesh, which LLVM's JIT compiler `lli` was able to execute.

[11] See [36] for more information on the HotSpot compiler.

5.2.2 Decompilers

In addition to JIT compilers, I also test several Java decompiler toolchains. These are somewhat 'experimental', and therefore are more likely to contain bugs or to be unable to handle some class files. I test two Java decompilers: Class File Reader (CFR) [38] and Fernflower [39]. I selected these decompilers based on Harrand et al. [40] and Mauthe et al. [41], which evaluate the performance of Java decompilers on a range of class files. CFG and Fernflower are two of the highest performing decompilers.

CFR is a Java decompiler written in Java 6, which can decompile modern Java features up to Java 14 and will attempt to decompile class files from other JVM languages into Java. It works by detecting known Java bytecode patterns from known compilers such as `javac`. Fernflower is the built-in decompiler used by IntelliJ. Both decompilers describe themselves as 'under development', and note that they will not be able to successfully decompile all class files. Nevertheless, they welcome bug reports and suggestions for improvement.

The testing process for the Java decompilers is as follows:

- First, I generate a Java bytecode program and compile it to a class file.
- Next, I use the decompiler to decompile the class file to a Java source program. At this stage, the decompiler may fail and throw an exception if it encounters Java bytecode that it is unable to process.
- If the decompilation was able to produce a Java source program, I re-compile it to a class file.
- Finally, I execute the de- and re-compiled program using the JVM. If the program crashes or does not give the expected runtime output, then a possible bug is flagged.

In some cases, it may be inappropriate to test a Java decompiler using test cases that were originally written in Java bytecode rather than Java. For example, it is possible that a Java bytecode program does not have a corresponding representation in Java due to differences in restrictions between the languages. However, all test cases used in this project can be compiled to class files and executed by a JVM, which suggests that a decompiler should be able to handle them. Additionally, since decompilers can be used for security purposes to inspect potentially malicious class files, it is useful to understand any areas of weakness in decompilation. At a minimum, the decompiler should provide an informative error message stating which part of the bytecode it was unable to process.

5.3 CIL

I implement a wrapper program in C#. Similarly to the Java wrapper, it calls the test case function repeatedly so that the JIT compiler has enough information to make profile-based optimisations. The number of function repetitions is an input parameter to the program.

I test the Mono compiler, which is a JIT compiler that performs profile-guided optimisations at runtime. In addition, I test the ILSpy C# decompiler.

5.3.1 JIT compilers

Mono [42] is an open source implementation of Microsof's .NET framework. It supports compilation on a range of architectures and operating systems including MacOS. Mono con-

sists of an interpreter and an optimising JIT compiler, similarly to HotSpot and GraalVM. Mono also supports a slower compilation engine based on LLVM,[12] which is slower to run but produces more optmised code. I found that the 'fast' compiler ran slowly on my machine, and so I did not test the slower Mono LLVM compiler during this project, but it could be an interesting area for further work.

5.3.2 Decompilers

ILSpy [43] is an open source .NET decompiler that decompiles .exe and .dll files to C#. Unlike the Java decompilers, it does not have to convert irreducible control flow to a reducible graph, since C# does not have the same control flow restrictions as Java. I use a similar de- and re-compilation process as set out in Section 5.2.2.

5.4 C

C is typically statically compiled. Many different compiler toolchains exist, which use a similar structure to the LLVM compiler toolchain described in Section 5.1. For example, different optimisation levels can be passed to the compiler at compile-time. I test `clang` and `gcc`, which are widely-used compiler toolchains.

5.4.1 Static compilers

`clang` is part of the Clang project [44], which is the C language family front-end for the LLVM project. Code compiled by `clang` is transformed to LLVM IR and optimised using the LLVM optimisations described in Section 5.1.1. It is possible to pass default optimisation levels to `clang`: `O1`, `O2`, `O3`, `Oz`, and `Os`. However, it is not possible to pass specific individual optimisations to `clang` as it is with the LLVM `opt` tool.

`gcc` is part of the GNU Compiler Collection [45]. Optimisation levels are the same as for `clang`. Detail on which specific optimisations are invoked at each level can be found at [46].

[12] See [42] for additional detail

6 Evaluation

In this Section I empirically evaluate FuzzFlesh. In Section 6.1 I describe the fuzzing campaign design and report the bugs identified. In Section 6.2 I evaluate the efficacy of FuzzFlesh on the LLVM toolchain to determine code coverage of the optimisation 'middle-end' of the compiler. In Section 6.3 I report performance in detecting synthetic bugs injected into the LLVM toolchain. In principle it would be possible to perform coverage and mutation analysis on other toolchains, however given time constraints and tool availability I have focused on LLVM.[13]

In Section 7 I use the evaluation results to address the research questions defined in Section 1.

6.1 Bug-finding experiments

FuzzFlesh was able to identify two possible bugs: one in the Java decompiler CFR, and another in the Java decompiler Fernflower. Java decompilers aim to reverse-engineer Java source code from class files, which is useful for security analysis and debugging in cases where the original source code is not available. See Section 5.2.2 for a more detailed description of these decompilers. I have reported both of these bugs. The CFR bug has been confirmed and a fix is in process, while the Fernflower potential bug is being investigated. Below I describe the experimental set-up across all toolchains, and then provide further detail on each bug.

6.1.1 Fuzzing campaign

I conducted fuzzing campaigns on a range of languages and compiler toolchains throughout this project, with the aim of identifying potential compiler crash- or wrong-code-bugs. Table 2 describes the full set of languages and toolchains tested.

Language	Toolchain	Type	Version	Architecture	OS
LLVM	LLVM	Static	18.0.0	arm64	macOS
	LLVM	Static	14.0.0	x86_64	Ubuntu 22.04
	GraalVM	JIT	15.0.6[14]	arm64	macOS
Java bytecode	HotSpot	JIT	17.0.8	x86_64	Ubuntu 22.04
	GraalVM	JIT	17.0.8+9.1	x86_64	Ubuntu 22.04
	CFR	Decompiler	0.152	x86_64	Ubuntu 22.04
	Fernflower	Decompiler	232.9559.62	x86_64	Ubuntu 22.04
CIL	Mono	JIT	6.12.0.199	arm64	macOS
	ILSpy	Decompiler	8.1.0.7455	arm64	macOS
C	clang	Static	14.0.0	x86_64	Ubuntu 22.04
	gcc	Static	11.4.0	x86_64	Ubuntu 22.04

Table 2: Tested toolchains

Table 3 reports the average test throughput and approximate total testing time for the main compiler toolchains and test configurations that I tested throughout the project. Throughput

[13] For example, JITWatch is an Oracle tool for observing which methods are being compiled in a program (rather than interpreted), and which optimisations are being applied.

[14] Oracle GraalVM Native 23.0.1

varied significantly for different toolchains. GraalVM LLVM and Mono were particularly slow: this may have been because they were running on a less powerful machine. I was limited on which machine I could run these toolchains on due to difficulties installing and compiling.[15] This machine configuration was mainly motivated by ease of implementation.

Language	Toolchain	OS	Directions	Throughput	Total test time
LLVM	LLVM	Ubuntu 22.04	Known	13,700	120
	LLVM	Ubuntu 22.04	Unknown	82,400	120
	GraalVM	macOS	Known	500	48
Java bytecode	HotSpot	Ubuntu 22.04	Known	3,750	120
	HotSpot	Ubuntu 22.04	Unknown	3,950	120
	CFR	Ubuntu 22.04	Known	1,000	48
	Fernflower	Ubuntu 22.04	Known	1,100	48
CIL	Mono	macOS	Unknown	250	48
	ILSpy	macOS	Unknown	250	48
C	Clang	Ubuntu 22.04	Known	2,000	60
	Gcc	Ubuntu 22.04	Known	450	60

Table 3: Throughput (tests per hour) and total test time (hours)

Within toolchains, as previously discussed the test configurations with known directions array have a lower throughput than those with an unknown directions array. For example, the LLVM testing throughput is around 6 times higher for unknown directions than known. This is due to the additional compilation time required for known directions.[16] The difference in throughput is much lower for known / unknown directions under the HotSpot compiler because programs are not statically compiled.

The different approximate time estimates for each compiler toolchain were in part driven by machine availability: I had access to multiple Ubuntu machines and was therefore able to conduct a longer testing campaign than those running on macOS. The shorter times for CFR and Fernflower reflect the fact that the range of tests that could run without triggering the bugs described below was more limited.

The difference in throughput and resource availability most severely affected GraalVM LLVM, which had a low throughput and only a single machine available for testing. As a result, it is likely that the fuzzing campaign that I conducted throughout this project was not sufficiently rigorous to thoroughly test it.

6.1.2 Bug 1: CFR nested switch

FuzzFlesh identified the first bug in CFR version 0.152. Listing 6 contains the main method from the bug-triggering test case, and Figure 9 shows the corresponding control flow graph. I manually reduced the test case from its original form, including removing the code relating to the directions array and output array since these were not necessary to trigger the bug. Decompiling this class leads to an error message that it cannot be decompiled.

[15] GraalVM LLVM in particular had difficulty accepting LLVM IR programs compiled in different ways, and so I limited testing to one macOS machine where it functioned most reliably.

[16] Since for m graphs and n paths, unknown directions only requires the compilation of m programs. Each path is passed to the program at runtime as a parameter. In contrast, known directions are hard-coded into the program, which results in $m{\cdot}n$ test programs that must be compiled.

```
.method public static main([Ljava/
    lang/String;)V
    .limit stack 2
block_0:
    bipush 1
    lookupswitch
        0: block_1
        1: block_2
        default : block_1
block_1:
    bipush 1
    lookupswitch
        0: block_3
        1: block_4
        default : block_3
block_2:
    return
block_3:
    return
block_4:
    return
.end method
```

Listing 6: CFR bug-triggering test

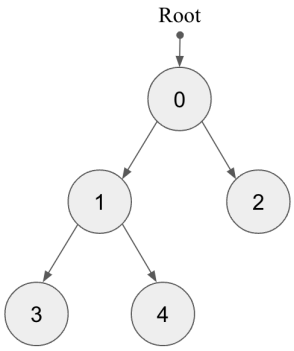


Figure 9: CFR bug-triggering CFG

I reported this bug and received a response from the CFR creator.[17] They explained that there were two underlying problems: first, 'an accounting failure caused by an unexpected rewrite' within the processing of the switch statements. Once this bug was fixed, CFR was able to decompile the test case but produced incorrect Java due to an additional bug. The source of the second, more serious, bug is the nested `lookupswitch` statements in blocks 0 and 1. Compiling Java with `javac` would result in the branches of the second switch being contained within the body of the first, whereas FuzzFlesh writes the blocks separately. This unexpected bytecode formulation confused CFR. Its usual approach in such cases is employ heuristics to untangle switch statements, however these did not work properly in this case.

The CFR creator commented he would expect that CFR should be able to cope with this program, and he is investigating the issue further. However, in general, CFR will likely not be able to decompile all class files that originate from Java bytecode perfectly, particularly those containing obfuscated control flow.

6.1.3 Bug 2: Fernflower irreducible control flow

FuzzFlesh identified a second bug in another decompiler, Fernflower, version 232.9559.62. Fernflower takes a different approach to decompilation than CFR: rather than looking for bytecode patterns emitted by known `javac` compilers, it attempts to determine whether the given class file can be represented in Java. It was therefore able to correctly decompile the class file corresponding to Listing 6.

However, it was not able to decompile the test case shown in Listing 7. The bug is still being investigated.[18] The source of the issue is likely to be that the CFG, shown in Figure 10, contains irreducible control flow, which is permitted in Java bytecode but not in Java. Irreducible control flow can be converted to reducible control flow using several techniques,

[17] https://github.com/leibnitz27/cfr/issues/348
[18] https://youtrack.jetbrains.com/issue/IDEA-331720

as noted in Section 4.1. Indeed, the removal of any edge, node or statement from the CFG resulted in a program that Fernflower was able to decompile, even those still containing irreducible control flow. For example, the decompiler worked correctly when I removed the statements on lines 12 and 13, which simply push and pop a constant to/from the stack.

```
1   .method public static main([Ljava
    /lang/String;)V
2       .limit stack 2
3   block_0:
4       bipush 2
5       ifeq block_1
6       goto block_3
7   block_1:
8       bipush 0
9       ifeq block_2
10      goto block_1
11  block_2:
12      bipush 2
13      pop
14      goto block_3
15  block_3:
16      bipush 0
17      lookupswitch
18          0: block_4
19          1: block_1
20          2: block_2
21          default : block_4
22  block_4:
23      return
24  .end method
25
```

Listing 7: Fernflower bug-triggering test

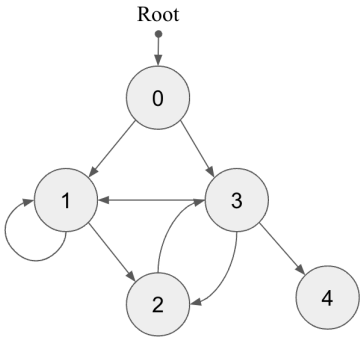


Figure 10: Fernflower bug-triggering CFG

The creator of Vineflower, another Java decompiler that is a fork of Fernflower, commented that they would expect Fernflower to be able to decompile this test case since part of its original purpose was to deobfuscate. The ability to deobfuscate is important if the decompiler is used for security purposes, such as examining class files from potentially malicious sources.

6.1.4 Bug discussion

Both bugs relate to a similar area: the decompilation of a language that allows irreducible control flow to a language that does not. While decompilers are generally not able to decompile every class file,[19] it is still useful to identify bugs so that the tools can be improved, particularly for use in security analysis.

The fleshing approach was also previously able to identify bugs in SPIR-V compilers. While SPIR-V compilers and Java decompilers clearly have many differences, they must both implement complex rules to handle control flow restrictions. The (de)compilers may struggle to implement these rules correctly: specifically, Java decompilers may struggle to implement efficient algorithms to convert irreducible-to-reducible control flow, even though it is theoretically possible. This is particularly the case for basic blocks that contain more complex statements, or when the CFG contains a more complex structure than the most simple irreducible cycle. Interestingly, FuzzFlesh did not find any bugs in ILSpy, which is the third

[19] For example, the CFR FAQ page states that it is likely that edge cases exist that it will struggle to deal with.

decompiler that I tested. However, the throughput for ILSpy tests was particularly low, and so I was not able to test it as thoroughly as the Java decompilers. ILSpy decompiles from a binary executable to C#, which allows irreducible control flow. This suggests that FuzzFlesh is most useful for testing compiler toolchains that must handle restrictions relating to control flow.

FuzzFlesh was not able to identify any bugs in the majority of compiler toolchains tested. This may be because they are relatively well-used, and so do not contain many bugs, or because the test programs produced are too simple to trigger bugs. I explore whether the test programs are able to reach into the optimising parts of compilers in Sections 6.2 and 6.3, and discuss potential areas for development in Section 7.

6.2 Compiler code coverage

Compiler code coverage measures the proportion of the compiler source code that is executed ('covered') during program compilation. Coverage metrics provide evidence on the thoroughness of the testing in terms of the extent to which FuzzFlesh is exercising different parts of the compiler. This information can be used to understand whether the test programs produced by FuzzFlesh are too simple to trigger optimisations. For example, if FuzzFlesh does not cover a particular area of an optimisation file, this indicates that the program format is not able to trigger that optimisation.

I evaluate the extent to which FuzzFlesh is able to cover compiler optimisations because this is the area that the approach is specifically targeted at.

6.2.1 Methodology

For the evaluation, I only consider the LLVM toolchain and do not test coverage of other languages or toolchains. This is because the process of instrumenting compiler source code with coverage is not straightforward.[20]

The methodology for estimating LLVM coverage is as follows:

- First, I compile LLVM with coverage instrumentation using `gcov` [47].
- Next, I use the instrumented LLVM to compile and optimise test cases. This produces coverage information for each code file within the compiler. The coverage information includes details of which specific lines and functions were executed during the test compilation.
- Finally, I process the raw information that is produced by `gcov` using `lcov`, which is a graphical report-producing front-end for `gcov`. Coverage can be calculated in aggregate over multiple tests to provide a picture of typical compiler coverage under repeated testing.

I use two comparators to evaluate FuzzFlesh's coverage:

[20] Building compilers with coverage involves instrumenting them using third-party tooling. LLVM has built-in coverage tooling (`llvm-cov`), however I was not able to build the `opt` tool using this because the memory requirements at the linking stage exceeded my available memory. The large memory requirements at the build stage arise because the instrumentation process bloats the compiler files. In addition, the third-party tooling does not reliably work with all computer configurations: I attempted several configurations before successfully building LLVM with coverage on Ubuntu 18.04.

- **Csmith** is a well-used random generator of C/C++ programs. The programs can be easily compiled through the LLVM toolchain. In addition, they support arbitrary `goto` statements which lead to interesting control flow.
- The **LLVM test suite** contains whole programs that are mainly written in C/C++. Whole program test outputs are compared to a reference output to ensure they are compiling correctly. LLVM also has unit and regression tests, which are more targetted pieces of LLVM IR code. Unit tests are primarily intended to test data structures, and so are not a particularly relevant comparator for FuzzFlesh. Regression tests aim to test the analysis and transform passes on LLVM IR, and so these would be a useful comparator. However, I was not able to build LLVM with coverage including the unit and regression tests due to memory constraints,[21] so I have not included these.

LLVM optimisations can be applied in multiple ways:

- The `opt` tool can be used to apply individual optimisations or combinations of optimisations through default levels: `O1`, `O2`, `O3`, `Os`, and `Oz`. `O1` - `O3` are increasingly aggressive optimisation levels, while `Os` optimises for output code size and `Oz` optimises for code speed. `opt` operates on LLVM IR (i.e. it is not possible to use `opt` to optimise a C program).
- `clang/clang++` can also be used to invoke optimisations through the same default levels as `opt`. LLVM IR code can be compiled direcly by `clang/clang++`, so it can be tested by the FuzzFlesh LLVM IR language backend as well as the C language backend. However, specific optimisations cannot be passed to `clang`.

LLVM optimisations are implemented as a series of 'Analysis' and 'Transform' passes that perform operations on a program [35]. Analysis passes gather information about the program to inform Transform passes, which apply specific transformations to the code. The optimisations are implemented as a set of C++ files contained within the LLVM code base. Transform pass files are grouped into sub-categories including AggressiveInstCombine, Scalar, and Vectorize. I measure the code coverage of FuzzFlesh, Csmith, and the LLVM test suite on the Transform passes, and report the results below.

6.2.2 Overall coverage results

I test two main configurations of invoking optimisations, one each through `clang` and `opt`. For all tests, I measure the code coverage from running 1,000 FuzzFlesh tests, 1,000 Csmith tests, and the LLVM whole-program test suite.[22]

`clang` results

Table 5 shows the code coverage results for the overall coverage of the LLVM compiler and overall coverage of a subset of transformation categories. These tests were compiled using `clang++` under the `O3` optimisation flag, which performs the most intensive optimisations. I compare the coverage results from FuzzFlesh tests produced using the C language backend,

[21] See footnote 20

[22] I note that this may not be a fully appropriate comparison because of the different test throughputs for FuzzFlesh and Csmith: if one testing approach takes much longer to compile then this may not be a fair comparison of their testing ability. An alternative approach would be to give each tool a time budget (e.g. of 6 hours) and measure coverage over that period. In addition, running the full LLVM test suite took much longer than either of the test approaches; however, the test suite is not a directly equivalent testing approach to random testing in any case.

Csmith tests (which are written in C), and the LLVM test suite (which is mainly written in C++).[23] I have not presented the results for any that have less than 5% line and function coverage CFGC, Csmith, and the LLVM test suite.[24]

	FuzzFlesh C		Csmith		LLVM tests	
	Line	Function	Line	Function	Line	Function
AggrInstCombine	31	75	63	96	91	100
Scalar	24	29	40	39	49	45
Utils	23	32	34	37	52	52
InstCombine	20	47	50	71	72	88
Vectorize	19	24	53	66	80	81
IPO	6	6	10	8	13	10
ObjCARC	5	10	5	10	5	10
Coroutines	1	4	1	4	2	6

Table 4: clang: LLVM optimisation coverage from invocation of clang on C/C++ programs (%)

Despite the relative simplicity of the fleshing approach, it is able to achieve a reasonable level of coverage for some transformation categories. For example, FuzzFlesh covers 31% of lines in the Vectorize Transform category, and 24% of lines in the Scalar category, which contains many important transformations including aggressive dead code elimination (ADCE) and transformations relating to loop optimisations.

Unsurprisingly, the FuzzFlesh has lower coverage than Csmith. This is expected because Csmith tests are designed to produce code that is far more syntactically rich than FuzzFlesh. For example, Csmith includes function calls, which correspondingly exercise the function inlining optimisation, while the fleshing approach does not include this language-specific feature. FuzzFlesh also does not include most of the operators and instructions of the language, which results in lower optimisation coverage. Both fuzzers have lower coverage than the LLVM test suite, which is expected because the LLVM test suite is carefully designed to cover the important parts of the compiler code base.

`opt` results

In addition to compiling tests using `clang`, I also measure the code coverage from running the LLVM `opt` tool on 1,000 FuzzFlesh LLVM IR tests and 1,000 Csmith tests under the `O3` optimisation flag. In order to ensure I am comparing a similar compilation process, I first convert the Csmith programs to LLVM IR using the `clang -emit-llvm` flag, and then invoke `opt` on the resulting IR programs. Table 5 shows the code coverage results for the overall coverage of the LLVM compiler and overall coverage of a subset of transformation categories.

The `opt` coverage results follow a similar pattern to the `clang` results: FuzzFlesh achieves a reasonable level of coverage, up to 14% of some overall Transform categories, but is lower than Csmith. Interestingly, optimising through `opt` rather than `clang` results in lower coverage for both the FuzzFlesh and Csmith tests. For example FuzzFlesh C tests achieve 24% coverage of Scalar Transforms when compiled through `clang`, but FuzzFlesh IR programs

[23] `clang` is also able to compile LLVM IR programs to executables, however this will of course not exercise the LLVM components that are concerned with parsing C/C++ syntax. This is not a significant issue because testing these parts of `clang` is not the focus of this project. Nevertheless, I report C tests compiled under `clang` here for a closer like-for-like comparison.

[24] This includes the Instrumentation and CFGuard categories, which are negligibly covered by all approaches.

	FuzzFlesh IR		Csmith IR	
	Line	Function	Line	Function
AggressiveInstCombine	14	58	61	96
Scalar	14	22	31	35
Utils	11	19	29	36
Vectorize	10	15	40	49
InstCombine	7	20	44	68
IPO	4	6	11	9
Coroutines	1	4	1	4
Instrumentation	0	1	0	1

Table 5: opt: LLVM optimisation coverage from invocation of opt on programs in LLVM IR format (%)

only achieve 14% coverage when `opt` is invoked directly. `clang` optimises by invoking the underlying LLVM `opt` tool, so it is somewhat surprising that invoking it directly (using the same `O3` optimisation level) leads to different amounts of optimisation coverage.

This difference could be caused by two factors. First, differences in the nature of optimisations called by `opt` and `clang`. This could occur because `clang` performs optimisations until some set of conditions is met, and these may differ from the conditions used by `opt`. Alternatively, the sequencing or set of optimisations involved the `clang O3` option is different to the LLVM `opt O3` option. This suggests that a fuzzing campaign with the objective of testing the optimisation files should focused on `clang` rather than `opt`.

	FuzzFlesh IR		FuzzFlesh C	
	Line	Function	Line	Function
Vectorize	18	23	19	24
Scalar	17	23	24	29
AggrInstCombine	14	58	31	75
Utils	12	20	23	32
InstCombine	7	20	20	47
ObjCARC	5	10	5	10
IPO	4	5	6	6
Coroutines	1	4	1	4

Table 6: clang: LLVM coverage from invocation of clang on programs in LLVM IR and C (%)

Second, the difference between the FuzzFlesh C and LLVM IR tests. Table 6 presents coverage results for FuzzFlesh IR programs compiled through `clang` and FuzzFlesh C programs compiled through `clang`. This shows that C programs generated by FuzzFlesh are able to achieve higher coverage of the LLVM optimisations than the IR programs. Both sets of programs were generated using the same CFG and path generation approach and so have similar structures. The main difference is therefore the language. This suggests that the C program may be converted to LLVM IR using a richer variety of LLVM IR syntax than I used in the FuzzFlesh program generator. This provides further evidence that syntactic richness is important for covering optimisations, and that a future development of FuzzFlesh should include language-specific options for implementing control flow structures.

Note that the coverage comparison in Table 5 is not entirely like-for-like: there is no way to

emit fully raw LLVM IR from C code. The `-emit-llvm` option performs very minor optimisations, and decorates all functions with a set of attributes that may affect future optimisations.[25] This means that testing `opt` using C tests that have been converted to LLVM IR may under-exercise the tool, which may be another reason for lower coverage.

6.2.3 File-level coverage results

More granular coverage results show that the fleshing approach can achieve reasonably high coverage of particular optimisation code files. Table 7 shows the LLVM files for which FuzzFlesh C achieves the highest line coverage when compiled using `clang`.

Transform	File	FuzzFlesh C	Csmith	LLVM tests
IPO	InferFunctionAttrs.cpp	100	100	100
Utils	SizeOpts.cpp	100	100	100
Scalar	SCCP.cpp	97	97	100
Utils	LoopRotationUtils.cpp	72	77	78
Utils	PromoteMemoryToRegister.cpp	72	82	86
IPO	CalledValuePropagation.cpp	69	75	83
Scalar	LoopDeletion.cpp	68	94	96
Utils	LoopSimplify.cpp	67	94	95
Scalar	ADCE.cpp	67	89	89
Utils	LCSSA.cpp	66	74	79
Utils	GlobalStatus.cpp	62	98	98
Utils	SCCPSolver.cpp	62	75	89
Scalar	DeadStoreElimination.cpp	61	81	92
IPO	FunctionAttrs.cpp	61	67	88
Scalar	LoopStrengthReduce.cpp	59	73	79

Table 7: Line coverage of a sample of LLVM optimisation files (%)

FuzzFlesh achieves high coverage for several optimisation files that relate to control flow, including LoopRotationUtils.cpp, which converts loops into do/while structures. The purpose of this transformation is to extract loop-invariant operations, such as loads (in IR), into the loop header so that they are not repeatedly executed within the loop body. Other CFG-specific optimisations include LoopDeletion.cpp and LoopSimplify.cpp. FuzzFlesh achieves a lower level of coverage than Csmith and the LLVM test suite across all files.

Table 8 reports the files for which FuzzFlesh IR achieves the highest line coverage under `opt`. This also shows relatively high coverage of CFG-specific optimisations. For example, the fleshing tests achieve 64% coverage for the LoopSimlify.cpp optimisation file, which ensures that nested loops are clearly separated.

FuzzFlesh C and IR are not able to cover more code than the LLVM test suite for any optimisation files. FuzzFlesh IR is able to achieve higher coverage than Csmith on one optimisation file, which is in **bold** in Table 8. FuzzFlesh does not cover some files well, particularly those relating to vectorization. Although FuzzFlesh achieves relatively high overall vectorization coverage, it is not able to cover some files that are well-covered by Csmith and the test suite.

[25] For example, using the `-emit-llvm` option produces LLVM IR with an `optnone` attribute applied to all fucntions, which prohibits any optimisations. Applying a further flag, `Xclang -disable-O0-optnone` removes the `optnone` attribute, but other attributes including `noinline`, which prohibits function inlining, remain.

Transform	File name	FuzzFlesh	Csmith	LLVM tests
Utils	SizeOpts.cpp	100	100	100
Scalar	SCCP.cpp	97	97	100
Utils	LCSSA.cpp	68	74	79
Scalar	ADCE.cpp	65	89	89
Utils	LoopSimplify.cpp	64	94	95
Utils	PromoteMemoryToRegister.cpp	64	82	86
IPO	InferFunctionAttrs.cpp	64	100	100
Scalar	LoopDeletion.cpp	59	94	96
Scalar	BDCE.cpp	58	100	100
Utils	LoopRotationUtils.cpp	56	77	78
Scalar	InstSimplifyPass.cpp	53	55	53
Scalar	**SimplifyCFGPass.cpp**	**53**	**46**	**62**
Utils	CanonicalizeFreezeInLoops.cpp	52	54	57
IPO	FunctionAttrs.cpp	52	67	88
Scalar	GVN.cpp	51	76	85

Table 8: opt: Line coverage of a sample of LLVM optimisation files (%)

For example, VPlan.cpp is not covered at all by FuzzFlesh C, but is 72% and 80% covered by Csmith and the LLVM tests respectively. Vectorization involves converting loops to vector operations. Csmith may express loops in a way that is more amenable to optimisation than the FuzzFlesh C or IR programs, which use a limited set of instructions to represent control flow constructs. For example, FuzzFlesh C uses `goto` to represent loops rather than `for`. This means that they do not have an explicitly represented increment counter, which is useful for implementing vectorization operations because it is clear to the compiler how many times a loop body will be executed. It is also useful for scalar optimisations such as loop unrolling, which involves repeating the loop body n times, where n is the number of increments. If the number of increments is not explicitly represented, then it is more difficult for the compiler to apply this optimisation.

6.2.4 Comparison of program fleshing approaches

I implemented several program fleshing approaches for LLVM IR, which are detailed in Section 4.2. The motivation for this was that providing the compiler with different amounts of information at compile-time could lead to different levels of optimisations. I measure compiler coverage under the two main program fleshing options for LLVM to test whether the different approaches result in different levels of coverage.

Table 9 reports the results for 1,000 FuzzFlesh tests in which the directions array is known at compile-time, and 1,000 FuzzFlesh tests in which the directions array is unknown. Coverage is slightly higher for the tests in which the directions array is known, which suggests that providing the compiler with this additional information allows it to perform additional optimisations. However, the incremental coverage is fairly small. A useful next step would be to explore the specific optimisation files further, specifically the conditions for particular optimisations to be triggered, to understand what kind of language-specific program fleshing options could induce greater optimisation.

	Directions known		Directions unknown	
	Line	Function	Line	Function
AggressiveInstCombine	14	58	14	58
Scalar	14	22	11	20
Utils	11	19	9	17
Vectorize	10	15	10	14
InstCombine	7	20	5	17
IPO	4	6	4	6
Coroutines	1	4	1	4
Instrumentation	0	1	0	1
CFGuard	0	0	0	0
ObjCARC	0	0	0	0

Table 9: LLVM coverage under different program fleshing options (%)

6.2.5 Code coverage limitations

Code coverage has some limitations as an evaluation metric. It only describes whether a line was executed, and does not include information on the relative importance of different lines, or whether tricky edge cases are covered. In addition, aggregate coverage over a set of tests does not provide information on the sequencing of line execution, which may matter for bug detection. For example, it may be necessary to execute lines 3, 5, and 7 in sequence to trigger a bug; if test A executes lines 3 and 7 and test B executes line 5, all of the bug-triggering lines are 'covered' but we have not tested the bug-triggering sequence.

Path coverage is a more advanced method that takes into account sequencing, but it is complex to implement and existing tools are less available than function and line coverage. A further useful metric to use would be differential coverage. This reports the number of lines that are covered by one test but not another. For example, one testing approach could achieve 70% coverage and the other could achieve 20%. If the latter approach covers a distinct set of functions or lines to the former, then it is a useful complementary testing approach.

6.3 Mutation analysis

Mutation analysis is the process of injecting synthetic bugs ('mutants') into a piece of code and checking whether they cause a test to fail. This can be used to evaluate how well a testing approach can detect and 'kill' mutants in the compiler code.

6.3.1 Methodology

I use the Dredd mutation analysis tool [48] to mutate the LLVM compiler toolchain. Dredd can mutate C++ code bases and provides functionality to track which mutants should be reachable by specific test cases. The mutation analysis process is as follows:

- **Build:** First, I build two instrumented versions of LLVM: one with mutant instrumentation to inject synthetic bugs into the compiler, and a second with mutant tracking instrumentation to keep track of which mutants a particular test should be able to detect. Dredd uses a compilation database produced when LLVM is initially built to determine where to inject mutants.

- **Mutate:** Next, I use Dredd to mutate a specific file within the LLVM code base. Once the file is mutated, I re-build LLVM with the mutation so that it is incorporated into the binaries. Dredd is able to mutate multiple files or libraries, however I only mutate individual files because the process of mutation increases the file size substantially. This can lead to difficulty at the re-build stage when the large files must be linked. Due to the time taken to mutate files, I only perform mutation analysis for a small subset of optimisation files.
- **Test:** Finally, I run the mutated `opt` on a set of FuzzFlesh and Csmith tests with the appropriate optimisation. For example, I mutate the file LoopSimplifyCFG.cpp within LLVM, and I optimise tests with the `simplifycfg` option. All mutants injected by Dredd are disabled by default, which means that they do not have an effect on the file. Mutants are enabled one at a time using an environment variable that specifies which mutant to activate. Each test is run repeatedly until either (a) all mutants that are theoretically detectable by the test are killed, or (b) some time limit expires. Mutant-killing results are collected from the full set of tests and aggregated to give the overall 'mutant score', which is the proportion of mutants in the file that were killed during testing.

6.3.2 Results

The mutation testing results are shown in Table 10. FuzzFlesh tests are able to detect many mutants within the optimisation files: 55% of mutants in the aggressive dead code elimination (ADCE.cpp) file are killed, and 43% of mutants in the simplify CFG (LoopSimplifyCFG.cpp) file are detected. Of the detected mutants, all are killed. These results provide further evidence that FuzzFlesh is an effective testing method that is able to detect bugs within the LLVM optimisation code. In addition to the coverage measures, the mutation testing result demonstrates that the tests are able to identify and kill bugs in the code, as opposed to simply 'covering' the compiler.

I also test the ability of Csmith to kill mutants as a comparator for FuzzFlesh. Similarly to the coverage analysis, Csmith kills a higher proportion of mutants than the FuzzFlesh tests. This is likely to be due to the greater syntactic richness and optimisation-triggering abilities of Csmith.

Transform	File	FuzzFlesh IR	Csmith
Scalar	ADCE.cpp	55	89
Utils	LCSSA.cpp	44	49
Scalar	LoopSimplifyCFG.cpp	43	85
Scalar	LoopUnroll.cpp	29	47
Utils	BasicBlocksUtils.cpp	26	39
Scalar	LoopRotation.cpp	24	29
Scalar	LoopUnrollAndJam.cpp	11	11

Table 10: Mutants detected and killed by FuzzFlesh and Csmith (% of total mutants in each file)

7 Conclusion and further work

FuzzFlesh is a multi-language compiler testing tool that has successfully identified two bugs in Java decompilers. It has demonstrated the feasibility of sharing a language-independent random program skeleton generator across several different language backends with a small amount of programming time. In this Section I discuss the key findings of this project with respect to each of the research questions that were introduced in Section 1.2, and set out areas of potential further work to address the limitations of FuzzFlesh.

7.1 Summary of findings

7.1.1 RQ1: How effective is FuzzFlesh at identifying compiler bugs?

FuzzFlesh was able to detect two potential bugs: one each in the Java decompilers CFR and Fernflower. I have reported both bugs and received confirmation of the CFR bug, while the Fernflower potential bug is still under investigation. The ultimate source of both bugs is likely to be the implementation of rules relating to control flow restrictions. This demonstrates that FuzzFlesh can be an effective method for identifying real bugs within compilers, particularly those relating to control flow.

However, FuzzFlesh was not able to identify any bugs in the majority of compiler toolchains tested, which suggests that in its present state it may not be an effective method for bug-catching.

7.1.2 RQ2: How thoroughly is FuzzFlesh able to cover compiler optimisations?

I evaluate the thoroughness of FuzzFlesh tests using two metrics: compiler code coverage and mutation analysis. Code coverage measures the amount of the compiler that the tests exercise. Mutation analysis measures the ability of the tests to detect synthetic bugs that are injected into the compiler. FuzzFlesh tests are able to achieve a reasonable level of compiler optimisation coverage given the simplicity of the approach, including over 50% line coverage of 28 LLVM optimisation files. FuzzFlesh is also able to detect and kill mutants (synthetic bugs) that are injected into LLVM compiler optimisation files, which provides further evidence that the approach can successfully identify compiler bugs.

7.1.3 RQ3: How does the efficacy of FuzzFlesh vary across languages and compiler toolchains?

All bugs identified were in Java decompilers. These have two salient characteristics compared to the other langauges and toolchains tested. First, they are relatively novel and experimental compared to well-established compilers such as `clang`, which means that they are less well-tested and therefore more likely to contain bugs. Second, while Java bytecode does not have any control flow restrictions, Java does. The decompiler must therefore convert the program from unstructured to structured, which is a complex process that can lead to errors (as was the case here). The conceptual approach implemented by FuzzFlesh was previously used to successfully identify bugs in SPIR-V compilers [1], which also involves strict control flow restrictions. This suggests that FuzzFlesh may be particularly well-suited to compiler toolchains that have complex control flow restrictions.

7.1.4 RQ4: To what extent can FuzzFlesh identify bugs that are different from those identified by other approaches, and what are the drivers of any differences?

The approach focuses on control flow, which results in test cases that have a different format to other approaches. For example the Java bytecode tests containing irreducible control flow would not be produced by a Java compiler fuzzer, yet they were able to uncover some interesting bugs.

Additionally, the ease of implementation for novel languages means that this approach could be a useful 'first-pass' compiler tester for novel languages and toolchains.

7.1.5 RQ5: How feasible is it to extend the compiler fuzzing method to multiple languages? What are the key constraints imposed by having a shared program-generation component of the fuzzing toolchain?

FuzzFlesh can be easily extended to multiple languages. I have demonstrated this by successfully applying the approach to four distinct languages: LLVM IR, Java bytecode, CIL, and C. Implementation to an additional language is expected to take approximately 8 - 12 hours of coding time for an experienced programmer, while language-specific program generation variants will of course take more time to implement. This method could therefore have useful applications in novel languages or compiler toolchains. For example, the method was effective in identifying bugs in Java decompilers, CFR and Fernflower, which are somewhat experimental tools.

Shared program-generation components did not impose any significant constraints on the languages that I tested. However, languages that involve greater control flow restrictions would require additional filtering stages because the CFG-generation step will produce invalid graphs. Producing valid CFGs, for example that do not contain irreducible control flow, may require significant additional work, or obtaining graphs from an alternative source.

7.2 Areas for further work

The obvious limitation of this method is that it has only found two bugs 'in the wild'. This could be because the toolchains are so well-tested that there exist few bugs, or because the tests are not sufficiently complex to trigger any existing bugs, particularly in well-used compilers. Areas for further research to address the limitations of FuzzFlesh are as follows:

- **Incorporation of additional language-specific features:** The method is able to cover relevant parts of the compiler and detect synthetic bugs. However, it may be that the kind of real-world-bugs likely to exist in compilers are not detectable by such simple programs. The tests leave most of the compiler code uncovered, and most mutants remain unkilled. This suggests that the approach may be achieving a fairly superficial level of coverage, and is not able to probe deeper into the compiler code where the real bugs might be lurking. A simple language-specific development would be to include a wider range of options for implementing control flow features, for example using a wider range of branching instructions. A further potential extension would be to combine the control flow structure with some randomly generated code snippets within each block (although this would require more complex behaviour testing). Another option would be to specifically encode optimisation-inducing control flow features, for example similarly to Livinskii, Babokin and Regehr [49].

- **Testing on structured control flow languages:** FuzzFlesh was successful at uncovering bugs in the Java decompilers and previously in SPIR-V, which is a structured control flow graph language that is subject to many rules around control flow handling. This suggests that the method may be particularly effective at uncovering bugs relating to control flow restrictions, where the compiler implementation of these rules may have gone awry. It would therefore be useful to test other structured languages, or tools that operate at the intersection between structured and unstructured languages e.g. compilers / decompilers that are converting unstructured to structured programs.
- **Wider range of program generation methods:** It is possible that the program generation methods used here do not produce 'interesting' enough program structures. Alternative methods of program generation could be explored, for example based on mutating CFGs that are extracted from real programs.
- **A more thorough fuzzing campaign:** In the project I tested many different languages and toolchains. Due to time constraints and low test throughput for some toolchains, it is possible that the number of programs tested was simply insufficient to locate bugs. For example, GraalVM LLVM testing had a very low throughput because the LLVM IR programs had to be constructed in a specific format using a version of `clang` that shipped with GraalVM, which took a long time. This means that I was not able to test it thoroughly in the time available. In general, newer toolchains may be less amenable to fuzzing due to slowness or inability to cope with unusual programs.

In addition, if the method is successful at identifying bugs, then a useful area of further work would be to develop a fleshing-based method for test case reduction, as discussed in Section 3.6. The method would be based on reducing the graph and paths, rather than reducing the program, which means that it could be shared across language backends.

References

[1] Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. Taking Back Control in an Intermediate Representation for GPU Computing. *Proceedings of the ACM on Programming Languages*, 7:1740 – 1769, 2023. URL https://doi.org/10.1145/3571253.

[2] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Towards understanding compiler bugs in GCC and LLVM. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 294 – 305, 2016. URL https://doi.org/10.1145/2931037.2931074.

[3] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A Survey of Compiler Testing. *ACM Computing Surveys*, 53:1 – 36, 2019. URL https://doi.org/10.1145/3363562.

[4] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283 – 294, 2011. URL https://doi.org/10.1145/1993498.1993532.

[5] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random Testing for C and C++ Compilers with YARPGen. *Proceedings of the ACM on Programming Languages*, 4:1 – 25, 2020. URL https://doi.org/10.1145/3428264.

[6] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216 – 226, 2014. URL https://doi.org/10.1145/2594291.2594334.

[7] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, André Perez Maselco Stefano Milizia, and Antoni Karpiński. Test-case reduction and deduplication almost for free with transformation-based compiler testing. *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1017 – 1032, 2021. URL https://doi.org/10.1145/3453483.3454092.

[8] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 65 – 76, 2015. URL https://doi.org/10.1145/2737924.2737986.

[9] Kazuhiro Nakamura and Nagisa Ishiura. Random testing of C compilers based on test program generation by equivalence transformation. *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 2016. URL https://ieeexplore.ieee.org/document/7804063.

[10] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. *ACM SIGPLAN Notices*, 52:247 – 361, 2017. URL https://doi.org/10.1145/3140587.3062379.

[11] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100 – 107, 1998. URL https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf.

[12] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: A new approach for generating test cases. *Technical REport HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong*. URL https://arxiv.org/abs/2002.12543.

[13] Alex Groce. Let a thousand flowers bloom: on the uses of diversity in software testing. *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 136 – 144, 2021. URL https://doi.org/10.1145/3486607.3486772.

[14] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, 2012. URL https://users.cs.utah.edu/~regehr/papers/swarm12.pdf.

[15] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28:183 – 200, 2002. URL https://doi.org/10.1109/32.988498.

[16] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. *ACM SIGPLAN Notices*, 46:335 – 346, 2012. URL https://doi.org/10.1145/2345156.2254104.

[17] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. *International Workshop on Automated and Algorithmic Debugging*, pages 206 – 222, 1993. URL https://link.springer.com/chapter/10.1007/BFb0019410.

[18] LLVM. LLVM test suite, . URL https://llvm.org/docs/TestingGuide.html. Accessed: 2023-01-09.

[19] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. Compiler Fuzzing: How Much Does It Matter? *Proceedings of the ACM on Programming Languages*, 3:1 – 29, 2019. URL https://doi.org/10.1145/3360581.

[20] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37:649 – 678, 2011. URL https://dl.acm.org/doi/10.1145/3575693.3575750.

[21] J.H. Andrews, J.H, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? *Publicationes Mathematicae Debrecen*, pages 290–297, 2005.

[22] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering*, 40(1): 23–42, 2014. doi: 10.1109/TSE.2013.44.

[23] Farah Hariri, August Shi, Hayes Converse, Sarfraz Khurshid, and Darko Marinov. Evaluating the Effects of Compiler Optimizations on Mutation Testing at the Compiler IR Level. *IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 105 – 115, 2016. doi: 10.1109/ISSRE.2016.51.

[24] Josie Holmes and Alex Groce. Using mutants to help developers distinguish and debug (compiler) faults. *Software Testing, Verification and Reliability*, 30, 2020. URL https://doi.org/10.1002/stvr.1727.

[25] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008. URL https://networkx.org/.

[26] P. L. Krapivsky and S. Redner. Organization of growing random networks. *Physical Review E*, 63(6), may 2001. doi: 10.1103/physreve.63.066123. URL https://doi.org/10.1103%2Fphysreve.63.066123.

[27] P. Erdös and A. Rényi. On random graphs i. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 402–411, 2005. doi: 10.1109/ICSE.2005.1553583.

[28] Khronos group. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html#_structured_control_flow. Accessed: 2023-01-09.

[29] Larry Carter, Jeanne Ferrante, and Clark Thomborson. Folklore confirmed: Reducible flow graphs are exponentially larger. *ACM SIGPLAN Notices*, 38, 10 2003. doi: 10.1145/640128.604141.

[30] LLVM. LLVM, . URL https://llvm.org/. Accessed: 2023-01-09.

[31] Oracle. Java Bytecode Instruction Set, . URL https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html. Accessed: 2023-01-09.

[32] Ted Neward. The Server Side. URL https://www.theserverside.com/news/1363881/The-Working-Developers-Guide-to-Java-Bytecode. Accessed: 2023-06-09.

[33] Jonathan Meyner and Daniel Reynaud. Jasmin. URL https://jasmin.sourceforge.net/. Accessed: 2023-01-09.

[34] LLVM. LLVM Opt, . URL https://llvm.org/docs/CommandGuide/opt.html. Accessed: 2023-01-09.

[35] LLVM. LLVM Passes, . URL https://llvm.org/docs/Passes.html. Accessed: 2023-01-09.

[36] Oracle. Oracle HotSpot, . URL https://www.oracle.com/java/technologies/whitepaper.html. Accessed: 2023-01-09.

[37] GraalVM. Graal Compiler. URL https://www.graalvm.org/latest/reference-manual/java/compiler/. Accessed: 2023-01-09.

[38] CFR. CFR. URL https://www.benf.org/other/cfr/. Accessed: 2023-01-09.

[39] IntelliJ. Fernflower. URL https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine/src/org/jetbrains/java/decompiler. Accessed: 2023-01-09.

[40] Nicolas Harrand, César Soto-Valero, Martin Monperrus, and Benoit Baudry. Java decompiler diversity and its application to meta-decompilation. *CoRR*, abs/2005.11315, 2020. URL https://arxiv.org/abs/2005.11315.

[41] Noah Mauthe, Ulf Kargén, and Nahid Shahmehri. A large-scale empirical study of android app decompilation. *2021 IEEE International Conference on Software Analysis, Evo-*

lution and Reengineering (SANER), pages 400–410, 2021. doi: 10.1109/SANER50967.2021.00044.

[42] Mono. Mono. URL https://www.mono-project.com/docs/advanced/mono-llvm/. Accessed: 2023-01-09.

[43] ILSpy. ILSpy. URL https://github.com/icsharpcode/ILSpy. Accessed: 2023-01-09.

[44] LLVM. Clang, . URL https://clang.llvm.org/. Accessed: 2023-01-09.

[45] GNU. GCC, . URL https://gcc.gnu.org/. Accessed: 2023-01-09.

[46] GNU. GCC Optimisations, . URL https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed: 2023-01-09.

[47] GNU. gcov, . URL https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. Accessed: 2023-01-09.

[48] Multicore Group, Imperial College London. Dredd. URL https://github.com/mc-imperial/dredd. Accessed: 2023-01-09.

[49] Vsevolod Livinskii, Dimitry Babokin, and John Regehr. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proceedings of the ACM on Programming Languages*, 7:1826 – 1847, 2023. URL https://doi.org/10.1145/3591295.