FuzzFlesh: Randomised Testing of Decompilers Via Control Flow Graph-based Program Generation

Amber Gorzynski
Imperial College London, UK

Alastair F. Donaldson
Imperial College London, UK

Abstract

Decompilation is the process of translating compiled code into high-level code. Control flow recovery is a challenging part of the process. 'Misdecompilations' can occur, whereby the decompiled code does not accurately represent the semantics of the compiled code, despite it being syntactically valid. This is problematic because it can mislead users who are trying to reason about the program.

We present CFG-based program generation: a novel approach to randomised testing that aims to improve the control flow recovery of decompilers. CFG-based program generation involves randomly generating control flow graphs (CFGs) and paths through each graph. Inspired by prior work in the domain of GPU computing, (CFG, path) pairs are 'fleshed' into test programs. Each program is decompiled and recompiled. The test oracle verifies whether the actual runtime path through the graph matches the expected path. Any difference in the execution paths after recompilation indicates a possible misdecompilation. A key benefit of this approach is that it is largely independent of the source and target languages in question because it is focused on control flow. The approach is therefore applicable to numerous decompilation settings. The trade-off resulting from the focus on control flow is that misdecompilation bugs that do not relate to control flow (e.g. bugs that involve specific arithmetic operations) are out of scope.

We have implemented this approach in FuzzFlesh, an open-source randomised testing tool. FuzzFlesh can be easily configured to target a variety of low-level languages and decompiler toolchains because most of the CFG and path generation process is language-independent. At present, FuzzFlesh supports testing decompilation of Java bytecode, .NET assembly and x86 machine code. In addition to program generation, FuzzFlesh also includes an automated test-case reducer that operates on the CFG rather than the low-level program, which means that it can be applied to any of the target languages.

We present a large experimental campaign applying FuzzFlesh to a variety of decompilers, leading to the discovery of 12 previously-unknown bugs across two language formats, six of which have been fixed. We present experiments comparing our generic FuzzFlesh tool to two state-of-the-art decompiler testing tools targeted at specific languages. As expected, the coverage our generic FuzzFlesh tool achieves on a given decompiler is lower than the coverage achieved by a tool specifically designed for the input format of that decompiler. However, due to its focus on control flow, FuzzFlesh is able to cover sections of control flow recovery code that the targeted tools cannot reach, and identify control flow related bugs that the targeted tools miss.

2012 ACM Subject Classification Security and privacy → Software reverse engineering; Software and its engineering → Software testing and debugging; Software and its engineering → Compilers

Keywords and phrases Decompiler, Reverse Engineering, Control Flow, Software Testing, Fuzzing

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.30

Funding This work was supported by EPSRC grant EP/R006865/1 and gift funding from Google.

1 Introduction

In this work we present a new randomised testing approach for automatically finding *misdecompilation* bugs in decompilers.

Decompilers aim to reconstruct source code corresponding to compiled code. This can be useful for detecting security vulnerabilities in binaries [1, 17, 54, 28, 60, 48], source code recovery [18], or to enable program analysis in cases where the source code is unavailable [45, 55, 52, 20, 24].

Decompilation is a difficult process because (a) some information is lost during compilation and (b) restoring high-level code structure is technically difficult, particularly when the code has been heavily altered through compiler optimisations. Control flow recovery is especially challenging because there are often multiple valid ways to structure a given control flow graph (CFG) using high-level constructs (such as `while` or `if/else`) and the process of deciding which structure to use can be a source of errors. In addition, attempts by the decompiler to make the output code more readable (for example by eliminating as many `goto` statements as possible) rely on complex algorithms [6, 65, 25, 4].

In this paper, we define *misdecompilation* as any semantic deviation between the original low-level code and its decompiled counterpart. Misdecompilations mislead analysts, wasting their time. They are more serious than decompiler crashes or failures to decompile because there is no indication to the user that the output is incorrect.The consequences of misdecompilations can be severe: misunderstandings arising from a misdecompilation could cause a security analyst to overlook a vulnerability or mis-classify a piece of malware. For example, Votipka et al. [59] find that malware engineers prefer working with decompiled code for readability, but must switch to (harder-to-read) disassembly when they suspect a misdecompilation has occurred. Source code recovery can also be complicated by the presence of misdecompilation bugs [18].

In recognition of the importance of reliable decompilers, recent work has used randomised testing to search for decompiler bugs in a proactive manner [41, 71, 43]. However, these only target one language format each and do not focus specifically on control flow.

We build on Klimis et al. [34] to develop a novel decompiler fuzzing approach called CFG-based program generation. It has two key features:

CFG-based program generation targets control flow recovery. The approach aims to thoroughly test the control flow recovery components of decompilers. Beyond control flow, our programs are designed to use limited low-level language features, so that the only source of complexity is the control flow. This restriction to a limited set of features is important, because in practice each decompiler has an informal 'scope', which is the set of programs it *should* be able to decompile. Programs that make intricate use of specialised low-level language features risk being outside of the scope of a decompiler, which may not support the reconstruction of these features. These programs are not useful as test cases because (a) they will not be recompilable, and therefore cannot detect misdecompilation bugs, and (b) will not be interesting as crash- or failure-inducing test cases to developers, who generally focus on in-scope supported language features. Prior work has devoted considerable attention to addressing the problem of recompilability, for example through post-processing the decompiled output [41, 71]. By producing programs that are syntactically simple, our approach sidesteps this problem.

The trade-off associated with our restriction to the control flow-related parts of low-level languages is that the programs generated by our approach are unlikely to trigger bugs related to detailed language-specific features (such as bugs in the handling of arithmetic operations).

CFG-based program generation is easily retargeted to various low-level languages. All mainstream programming languages have some notion of control flow. CFGs generated by

our approach can therefore be 'fleshed' into many different target languages, without the need to provide full grammars to the tool or perform extensive language-specific engineering. We are not aware of any other decompiler testing techniques that can be easily retargeted to work with distinct low-level languages. Prior decompiler testing tools have been highly customised for particular languages, requiring a great deal of expertise to build and typically using existing specialised program generators for the target language [43]. Our approach drastically reduces this manual effort with respect to the testing of control flow recovery.

In more detail, CFG-based program generation involves randomly generating CFGs according to graph generation algorithms. Random paths through each CFG are also generated; these represent the expected runtime execution path. Each (CFG, path) pair is 'fleshed' into a program that (a) guarantees that a particular path through the program should be traversed, and (b) checks that this path really *is* followed when the program executes. This check forms the test oracle: each program contains a self-checking assertion that fails if there is any difference between the actual and expected path.

We have implemented this approach in FuzzFlesh, a new open source decompiler testing tool. Figure 1 illustrates how we use FuzzFlesh to test decompilers. First, we **generate** a CFG and path, which we **flesh** into a low-level program (for example a Java bytecode class). We use the decompiler under test to **decompile** the program. We then **recompile** the program. Finally, we **execute** the recompiled program and assert that the actual runtime path matches the expected path. Any differences indicate a possible misdecompilation.

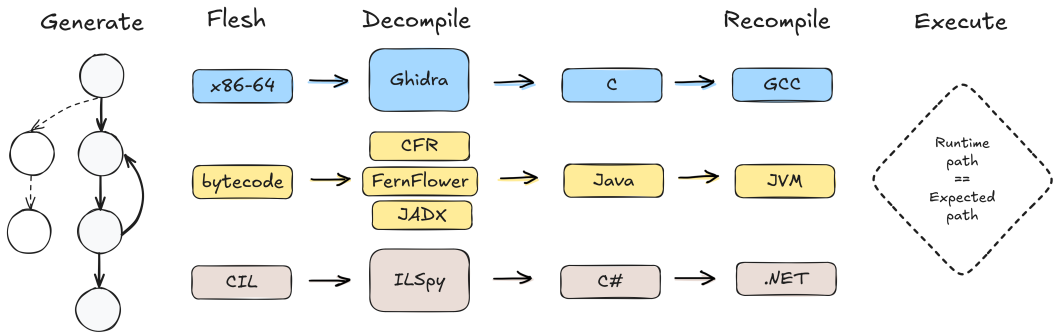

Figure 1 Workflow of CFG-based program generation as implemented by FuzzFlesh

FuzzFlesh also features a test case reducer that operates on the CFG and path to automatically reduce bug-triggering programs to a manageable size. The reduction process retains the "expected vs. actual path" test oracle, which assists in pinpointing the source of the misdecompilation.

Most of the program generation and reduction toolchain required can be implemented in a language-independent way, which means that it can be easily extended to multiple languages. Based on our experience implementing support for four languages, we estimate that implementing an additional language would only take approximately 8 hours for an experienced programmer (see Section 6.1). This is in contrast to other program generators used by decompiler testing tools, which require extensive language-specific engineering to produce test cases. For example, JavaFuzzer [66] and Csmith [68] are used by other tools and can only produce Java and C programs respectively. Such custom program generation tools require a large amount of effort to produce for each language. FuzzFlesh is therefore particularly useful for quickly testing a range of decompiler toolchains across different languages.

Currently, FuzzFlesh supports the generation of low-level programs in three different formats: Java bytecode, x86-64 machine code, and .NET assembly. We have used FuzzFlesh to test three Java decompilers (CFR, FernFlower, JADX), one binary decompiler (Ghidra), and one C# decompiler (ILSpy). We compare the performance of FuzzFlesh to a binary decompiler testing tool (DecFuzzer) and a Java decompiler testing tool (JD-Tester).

Our evaluation aims to answer the following research questions about CFG-based program generation, in the context of the FuzzFlesh tool:

RQ1	How effective is FuzzFlesh at identifying decompiler bugs?
RQ2	How does the efficacy of FuzzFlesh vary across languages and decompiler toolchains?
RQ3	How thoroughly is FuzzFlesh able to test the control flow recovery components of decompilers in comparison to other decompiler testing approaches?
RQ4	To what extent can FuzzFlesh identify bugs that are different from those identified by other decompiler testing approaches?

Our results show that FuzzFlesh is effective: it has found 12 previously-unknown bugs in four decompilers across two languages, of which eight have been confirmed (with six already fixed) in response to our bug reports. Ten of the bugs are directly related to control flow recovery. Six of the total bugs are misdecompilations, while the remaining six are refusals to decompile, whereby the decompiler refuses to produce a program in the target language due to an internal failure.[1] FuzzFlesh achieves reasonable code coverage of decompilers despite the limited syntax and language features used within its programs. As expected, due to its focus on control flow, the coverage FuzzFlesh achieves on any particular decompiler is lower than that achieved by language-specific decompiler testing tools. Nevertheless, FuzzFlesh is able to cover code that other tools cannot reach. In particular, FuzzFlesh identifies bugs within control flow recovery code that is not covered by other tools.

In summary, our main contributions are:

- A novel program generation technique that specifically targets control flow recovery in decompilers by generating CFG data structures. The programs are designed to be syntactically simple and therefore easily recompilable.
- An implementation of this technique, FuzzFlesh, which comprises: a CFG and path generating component, three different program 'fleshing' modules that target x86, Java bytecode, and .NET assembly, a program reducer that operates on the CFG and path data structures, and a test harness for automating the decompiler testing workflow.
- A large-scale testing campaign that identified 12 bugs by applying FuzzFlesh to five decompilers covering three distinct (low, high) language pairs: CFR, FernFlower, JADX, Ghidra, and ILSpy.
- An evaluation of FuzzFlesh featuring a comparison to two other recent decompiler testing approaches, DecFuzzer and JD-Tester, showing that FuzzFlesh offers complementary bug-finding abilities.

Availability. FuzzFlesh is open-source and available on GitHub [21]. An artifact is also published along with this paper [22]. This includes a copy of FuzzFlesh and all scripts and instructions required to reproduce our results.

[1] See Section 2.2 for further detail on decompiler problem categorisation.

2 Background

In this section we explain the necessary prerequisites for understanding the CFG-based program generation approach. Further related work is described in Section 7.

2.1 Decompilation

Decompilers are available for many languages [5, 30, 50, 29]. In general, decompilers are 'best effort': they may detect and reject low-level programs that are overly complex. Even when decompilation is attempted, decompilers typically cannot produce code that is syntactically identical to the original source code [16, 64]. Nevertheless, in order to be useful, decompilers must be free from serious errors. In particular, when a decompiler *does* deem a given low-level program in scope for decompilation, the high-level program that the decompiler emits *must* respect the semantics of the low-level program.

Decompilation involves several phases that reverse the compilation process. Implementations vary, but the general approach is as follows [13]. First, the executable binary is lifted to an intermediate representation (IR) using a process similar to disassembly. Next, the CFG, variables, and types are recovered from the IR. Further phases of recovery may take place, notably control flow structure recovery that recovers high-level syntactic control flow structures from the CFG (e.g. `if`/`else`, `for`, `while` and `do-while` in C-like languages). Other optimisations such as dead code elimination and variable renaming may be performed. Finally, code generation outputs a decompiled program in the source code language.

All phases of decompilation are challenging because information is irretrievably lost during compilation, including variable names (unless debug information has been inserted by the compiler), types, and high-level programming abstractions. Control flow structure recovery is particularly important and has been much researched. Early structuring algorithms attempt to match node configurations against a codified set of high-level schema [6]. However, this technique struggles when it encounters unexpected configurations. As a fallback, a decompiler may handle such cases by using `goto` statements. However, the use of `goto` can lead to code that is difficult to read, and many high-level languages (notably Java) do not feature an unrestricted `goto` statement or similar. Alternative structuring algorithms can be used to attempt to eliminate [65, 25] or minimise [4] the need for `goto` statements, but correctly implementing these algorithms is technically challenging. Control flow recovery becomes difficult when the program's control flow graph features *irreducible* structures [2]—cycles in the graph that have multiple entry points—but the target language does not support arbitrary `goto` statements, because irreducible control flow does not map naturally to structured programming language constructs.

2.2 Decompiler bug categories

Two categories of decompiler problems are unequivocally bugs:

- **Decompiler crash:** The decompiler fails to produce a program in the target language, and crashes, hangs, or otherwise fails ungracefully.
- **Misdecompilation:** The decompiler produces a program in the target language that can be successfully recompiled, but it is semantically inconsistent with the original program.

Two other categories of problems are sometimes, but not always, indicative of bugs:

- **Refusal to decompile:** The decompiler refuses to produce a program in the target language, failing gracefully with an error message. This may be indicative of a bug in the

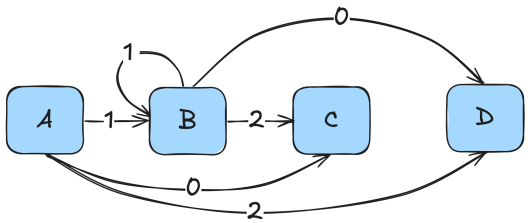

Figure 2 Example CFG generated by FuzzFlesh with directions `[0]` and expected output `[A,C]`

decompiler, but alternatively may be because the input program does not fall within the scope of programs the decompiler aims to handle.

- **Recompilation failure:** The decompiler produces a program in the target language that cannot be successfully recompiled, due to syntax or typing problems. Whether such problems constitute decompiler bugs depends on whether the decompiler has been designed to produce code that is expected to be directly recompilable.

Our approach aims to find misdecompilation bugs, which are of prime interest because they may mislead users. As a by-product, our approach has the potential to find decompiler crash bugs, which may be frustrating for users but are less serious than misdecompilations. It can also find 'refusal to decompile' issues that may be useful to decompiler developers in understanding and assessing the limits of their tools.

While some works address the detection of recompilation failures [71], this is not our focus because many decompilers do not aim to produce code that is directly recompilable. A distinction exists between *semantically correct* and *recompilable* code: the decompiler output may be semantically correct in that it is an accurate representation of the original code, but it may need significant formatting adjustments in order to recompile without errors. For example, Ghidra [50] sometimes excludes function declarations or necessary `#include` statements. This does not pose a problem for the typical user who visually analyses the decompiled output. We therefore do not include this category of failure because it conflates bugs and expected limitations of the decompiler.[2]

3 CFG-based Program Generation and the FuzzFlesh Tool

We now describe our CFG-based program generation approach in detail and explain how we have put it into practice in the FuzzFlesh tool.

We start with an overview of how the approach works using an example graph that was able to identify a misdecompilation bug in CFR [5], a tool that decompiles Java bytecode into Java source code. We then provide further details of each component of the approach.

CFG-based program generation involves randomly generating CFG skeleton data structures and then 'fleshing' them into programs in a target low-level language. Figure 2 shows an example CFG that might be generated. Each node represents a basic block in the low-level program that will ultimately be generated, and each edge represents a jump in the control flow between two nodes. The outgoing edges of a node are numbered with consecutive integers starting from 0. A CFG has a single entry point, in this case node A.

[2] We do automatically apply minor fixes to programs to support recompilability, which are discussed in Section 4. We do not report these issues as bugs.

```
test_program() {
    int[] directions = {0};
    int[] actual_output;

    int dir_index = 0;
    int out_index = 0;


block_A:
  actual_output[out_index++] = 0;
  switch(directions[dir_index++]) {
    case 0: goto block_C;
    case 1: goto block_B;
    case 2: goto block_D;
  }
```

```
block_B:
  actual_output[out_index++] = 1;
  switch(directions[dir_index++]) {
    case 0: goto block_D;
    case 1: goto block_B;
    case 2 : goto block_C;
  }

block_C:
  actual_output[out_index++] = 2;
  return;

block_D:
  actual_output[out_index++] = 3;
  return;
}
```

Figure 3 Example pseudocode corresponding to the CFG in Figure 2

A path through the CFG is also randomly chosen, and a corresponding sequence of edge labels is determined; we call this sequence the *directions* that must be taken for the path to be followed. For example, suppose the path $[A, C]$ through the CFG of Figure 2 was chosen. The directions sequence [0] would force this path to be taken.

From a (CFG, path) pair, (G, p), a concrete low-level program and program input are synthesised, where the program input is the sequence of directions associated with p. The program is designed to (a) consume the directions and follow the corresponding control flow path, and (b) output the sequence of nodes that is traversed during execution. Therefore, on execution, the program should output the path p.

The synthesised low-level program is then decompiled and recompiled, and executed on the input, and a check is performed to confirm that the program output does indeed indicate that the required path was followed. If not, a misdecompilation has occurred and can be investigated.[3] The comparison of the expected and actual path forms the test oracle, which is self-contained within every test case.

Returning to the example of Figure 2, Figure 3 shows C-like pseudocode illustrating how the graph can be fleshed into a program. The actual output is recorded at each node in the `actual_output` array. The `directions` array is read from every time that a conditional jump is encountered. The target language of the flesher depends on the toolchain under test.

When configured to target Java, our FuzzFlesh tool generates a textual Java bytecode program for this example, which is converted to a `class` file using the Jasmin assembler [47]. FuzzFlesh uses CFR to decompile the program, uses `javac` to recompile it, and executes it on the input [0], which should force the path $[A, C]$ to be followed. The actual output indicated that the path $[A, B, C]$ had been followed: the bytecode program had been misdecompiled. We have reported this issue to the CFR developers.

An attractive feature of CFG-based program generation is that it is easy to retarget for various low-level languages. For example, the graph- and path-generation parts of our FuzzFlesh implementation are entirely language-independent. The only language-specific work that is required is the straightforward logic to flesh a (CFG, path) pair into a low-level program. The tool is thereby able to test a range of decompilers for various languages.

In practice, FuzzFlesh generates larger and more complex CFGs than the one shown in Figure 2. Once a potential bug is identified, FuzzFlesh reduces the program to a manageable

[3] Is is theoretically possible that the problem could actually be a compiler bug introduced during the recompilation process, but this has not occurred in practice in our experience.

size by performing a series of operations on the CFG data structure and then re-fleshing it into a (smaller) program.

We now provide further detail on how we have realised each component of CFG-based program generation in FuzzFlesh. Section 3.1 and Section 3.2 set out the CFG and path generation approaches. Section 3.3 describes how these CFGs and paths are fleshed into programs. Section 3.4 explains the test oracle used to identify potential misdecompilations. Finally, Section 3.5 describes the CFG-based program reducer.

3.1 Graph generation

The aim of the graph generator is to create a CFG that can be used as the basis for a program. It generates graphs using a set of user-configurable input parameters including a random seed. Each graph is randomly generated from scratch; there is no initial corpus of graphs. The output is a directed graph data structure that contains nodes and directed edges. The generated graphs must be a valid basis for programs, which is achieved by ensuring that they meet a set of conditions. First, there must exist a single entry point to the graph; second, there must exist at least one path from the entry node to an *exit node*: a node with no successors. All target languages (x86-64, Java bytecode, and .NET assembly) allow irreducible control flow (see Section 2.1), which means we do not impose further restrictions on the shape of the graphs, although other target languages that do not support arbitrary control flow (such as WebAssembly) would require additional checks. This issue is discussed further in Section 4.2. Disconnected graphs are valid programs, providing there is only one entry point in the graph. The disconnected region can be considered 'dead' code that the decompiler may eliminate.

The graph generator in FuzzFlesh grows a graph of a parameterised target size using a breadth-first algorithm adapted from Krapivsky and Redner [35]. Starting with the root, the generation algorithm adds a random number of successors nodes. It continues to add a random number of successor nodes to each existing node until the target graph size is reached. 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, we add edges to a randomly chosen number of nodes in the graph. A proportion of these additional edges jump backwards to a direct ancestor of the current node, which introduce cycles into the CFG so that important program constructs such as loops are represented. Finally, additional exit nodes are optionally added to a randomly selected sample of nodes in the graph. This reduces the likelihood that a path will become stuck in a part of the graph that has no naturally-occurring exit nodes due to the back edge annotations.

We also implemented an alternative graph-generation algorithm adapted from Erdős and Rényi [19]. This graph generator creates a set of nodes according to a graph size parameter and randomly connects them via directed edges. A single node is randomly selected as the root node. In addition, exit nodes are randomly added. In practice we found that this method generates graphs with many disconnected components and short paths, and we found it to be ineffective at triggering bugs in practice, so do not discuss it further.

FuzzFlesh has been designed in a modular fashion so that other graph generation approaches would be straightforward to integrate. For example, one could mine existing programs for interesting control flow graphs and using these as the basis for fuzzing; this is an area for future work.

This stage of the toolchain is entirely language-independent.

3.2 Path generation

The FuzzFlesh path generator performs a random traversal of each generated CFG. It records the sequence of nodes visited on the path, and the sequence of directions that will force this path to be followed. 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. FuzzFlesh does not attempt to cover all paths through the CFG. Instead, a user-configurable number of paths through each CFG are selected at random.

This stage of the toolchain is also language-independent: paths are a data structure containing the path and the directions array.

3.3 Program fleshing

At this stage, FuzzFlesh becomes language-specific. The inputs to the flesher component are a graph data structure and a path. The output is a program in a particular low-level language, ready to be decompiled. By construction, the output program will write the IDs of each node visited at runtime to an actual output array. The contents of this output array can be compared with the expected output from the path to provide the test oracle.

Program fleshing involves traversing the CFG and fleshing each node based on its characteristics. As described in Section 3.1, the CFG is constructed as an abstract graph (i.e. not in a low-level language format) and fleshing provides a direct mapping between CFG nodes and appropriate low-level language operations. 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 children that each node has and fleshes it with an appropriate basic control instruction such as `goto`, `if/else`, or `switch`. Exit nodes with no children contain a return instruction. We do not include more complex language features in order to ensure that the resulting program is simple enough to be decompiled and subsequently recompiled.

During the fleshing stage, the directions array corresponding to the expected path can be configured to be statically unknown or known. This provides respectively less or more information about the program to the decompiler under test.

If the directions are statically unknown, then the directions array is 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 decompile-time. This creates a program in which every node in the graph is potentially dynamically reachable at run-time, given suitable inputs.

Alternatively, the directions array can be hard-coded into the function during the fleshing step. In this case, the directions are known to the decompiler.

These approaches may result in different de-compilation heuristics. The overall impact on the decompiler is unclear: providing more information to the decompiler by hard-coding the directions could make the program easier to decompile. However, as discussed in Section 4.1, in some cases we generate programs in a high-level language such as C and use a compiler to generate the low-level program (rather than directly generating the low-level program). In this setting, statically-known directions can be used to perform optimisations at compile-time. The resulting optimised binary may have a more complicated and less easily recoverable

structure. Alternatively, compiler optimisations may have greatly simplified the program, which could result in more straightforward decompilation. To maximise diversity during testing, we generate the low-level programs using a range of compiler optimisation levels.

3.4 Test oracle evaluation

Each program generated by FuzzFlesh contains a built-in self-checking test oracle in the form of the expected output. A test harness performs the testing process and evaluates the test oracle to determine whether the test has failed. The testing process for decompilers is as follows:

1. **Decompile:** We use the decompiler under test to decompile the program to a target source code. At this stage, the decompiler may crash or fail and throw an exception if it encounters code that it is unable to process.
2. **Recompile:** If the decompilation was able to produce a source program, we recompile the resulting program.
3. **Execute:** Finally, we execute the de- and recompiled program. If the actual output recorded by the program at runtime differs from the expected output, then a misdecompilation bug is flagged for investigation.

3.5 Test case reduction

Once a possible bug is found, FuzzFlesh reduces the bug-triggering test case to a manageable size in order to pinpoint the source of the error. We have designed a custom automated test case reducer to make the simplification and triage of bug-triggering test cases easier. Most of the reduction is done at the language-independent graph and path level, which means that it can be reused across multiple languages. The reducer performs the following types of operation on a (CFG, path) pair:

- Merge nodes: Two nodes n_1 and n_2 are merged together into a single node n' so that the total number of nodes in the graph is reduced by one. Any inwards edge to n_1 or n_2 becomes an inward edge to n', and similarly for any outwards edge. If either of the nodes was on the path, then the path and associated directions are also updated. Node merge operations must ensure that at least one exit node is retained so that the program does not become non-terminating.
- Remove edges: Edges are removed from the graph. If an edge is on the path, then the path and directions are also updated.

After each operation, the (CFG, path) pair are re-fleshed into a program. If compiling, decompiling, and recompiling still trigger the same bug, then the program is saved and reduced further. Otherwise, the updated program is discarded and a different reduction operation is attempted.

4 Decompilers under test

We test three Java decompilers: CFR [5], FernFlower [30], and JADX [32], the popular binary decompiler Ghidra [50], and the C# decompiler ILSpy [29]. We selected these decompilers because they are relatively well established and currently maintained, and they have been reviewed in previous studies [41, 43, 71]. Although it is considered in prior work we did not include the decompiler RetDec because it describes itself as in 'limited maintenance mode' and only has one commit in the past six months [3]. We also do not consider the

angr decompiler [57] because we found that extensive engineering effort is required to allow outputs from angr to be recompiled.

Table 1 summarises the key characteristics of these decompilers, which we now discuss in more detail.

Table 1 Decompilers under test

Name	Input	Target	CFG recovery
CFR	Java bytecode	Java	Pattern-matching
FernFlower	Java bytecode	Java	Tarjan's algorithm
JADX	Java bytecode	Java	Pattern-matching
Ghidra	Binary	C	Tarjan's algorithm
ILSpy	.NET assembly	C#	Pattern-matching

4.1 Binary decompilers

Ghidra [50] is an open-source reverse engineering tool that allows users to perform a range of analyses on compiled code. We specifically test the decompiler configuration that decompiles x86-64 machine code to C. Rather than directly generating machine code, FuzzFlesh instead fleshes graphs into C programs, which are then compiled to machine code using gcc with a randomly chosen optimisation level. Ghidra disassembles and decompiles these programs to C code. It uses Tarjan's algorithm for identifying strongly connected graph components [58] for control flow recovery. The output programs can typically be re-compiled and executed. Some minimal adjustments are occasionally required. For example, Ghidra output often includes a function call to `__stack_chk_fail()` to indicate that there may be an issue with the decompilation process, but it does not define this function and so the program fails to recompile. This adjustment is easily automatable: we simply insert a line into the decompiled program: `void __stack_chk_fail(){ return; }`, which allows the program to recompile and execute without errors or warnings. Based on our experience, Ghidra tends to insert such calls frequently even within programs that it otherwise appears to have decompiled correctly, which indicates a degree of over-caution. In the case that the `__stack_chk_fail()` call relates to a genuine problem with the decompiled code, our test oracle should identify the decompilation as incorrect since the program output will be altered.

4.2 Java decompilers

FernFlower [30], CFR [5], and JADX [32] are open-source decompilers that decompile Java bytecode class files to Java source code. CFR and JADX use pattern-matching control flow recovery algorithms that are supplemented by heuristics (for example in the handling of switch statements). FernFlower uses some pattern-matching for code generation but, similarly to Ghidra, also makes use of Tarjan's algorithm (see Section 4.1). It was the first Java decompiler that specifically aimed to be able to deobfuscate code, and it includes modules specifically for restructuring control flow that is not representable in Java.

FuzzFlesh generates test programs in textual Java bytecode, which are converted to binary Java class files using Jasmin [47]. We generate Java bytecode rather than Java because it allows more expressive control flow. Specifically, Java does not allow irreducible control

flow [23, Section 2.2.6] (see Section 2.1). However, Java bytecode *does* allow irreducible control flow; the bytecode instruction set [51] includes an unrestricted `goto` command that can be used to transfer control to anywhere in the program. This difference in restrictions allows compilers that generate bytecode to perform optimisations that introduce irreducible control flow.

It is possible that some Java bytecode programs do not have a corresponding representation in Java due to differences in restrictions between the languages. In this case, it is expected that a bytecode decompiler will gracefully refuse to decompile the program. Nevertheless, since decompilers can be used for security purposes to inspect potentially malicious class files, it is desirable for bytecode decompilers to be as capable as possible in decompiling atypical bytecode. At a minimum, the decompiler should provide an informative error message stating which part of the bytecode it was unable to process.

4.3 C# decompilers

ILSpy [29] is an open source .NET decompiler that decompiles programs expressed in the .NET intermediate representation, Common Intermediate Language (CIL) to C#. Unlike the Java decompilers, it does not have to convert irreducible control flow to a reducible graph because C# supports irreducible control flow via a `goto` statement. FuzzFlesh has two modes for targeting CIL: one in which CIL code is generated directly, and another where C# programs are generated, and then compiled into CIL using the .NET C# compiler csc.

C# programs are typically compiled to CIL, which is then JIT-compiled using the .NET runtime on Windows and Linux or the mono runtime on Mac OS. The intermediate CIL programs are not usually optimised ahead of runtime, and there are no options to do so in .NET or mono. ILSpy decompiles these programs and returns a corresponding C# program.

ILSpy does not apply algorithms to remove or reduce `goto` statements. This means that the decompiler output is often very long and consists mainly of `goto`s rather than other control flow structures.

5 Evaluation: Bugs found

In this section we answer research questions **RQ1** and **RQ2** set out in Section 1 and recapped below. In Section 6 we further evaluate FuzzFlesh in comparison to other decompiler testing tools to answer the remaining research questions.

RQ1 How effective is FuzzFlesh at identifying decompiler bugs? (Section 5.1)

RQ2 How does the efficacy of FuzzFlesh vary across languages and decompiler toolchains? (Section 5.2)

5.1 RQ1: Effectiveness of FuzzFlesh for bug-finding

We tested five decompilers, which are described in detail in Section 4 and summarised in Table 1. FuzzFlesh identified 12 bugs in total, comprising six misdecompilation bugs and six decompiler failures. Details of these bugs are provided in Table 2. The bugs were found across a total of four decompilers, which target two different low- to high-level language pairs. At the time of writing, eight bugs have been confirmed and six fixed by developers. We label each bug as ‘M’ to denote a misdecompilation, or ‘R’ to denote a refusal to decompile that we deemed to be likely to be in-scope for the decompiler in question, in line with our definitions of bug types in Section 2.

Table 2 List of decompilation bugs found (GitHub issues are hyperlinked to the bug ID) Type M = misdecompilation, Type R = refusal to decompile

ID	Decompiler	Type	Description	Fixed
G1	Ghidra	M	Missing jump table entry	✓
G2	Ghidra	M	Incorrect counter increment	✓
G3	Ghidra	M	Counter array incorrectly sized	
G4	Ghidra	M	Locals instead of array	
G5	Ghidra	R	Incorrect switch decompilation	✓
C1	CFR	M	Missing switch case	
C2	CFR	R	Nested switch	✓
C3	CFR	R	Irreducible control flow cannot be restructured	✓
F1	FernFlower	M	Incorrect infinite loop	
F2	FernFlower	R	Irreducible control flow cannot be restructured	
J1	JADX	R	Code restructure failure	✓
J2	JADX	R	Unreachable block	

Most bugs are related to control flow restructuring. The bugs can be grouped into themes: cycle bugs, switch bugs, and Java-specific irreducible control flow bugs.

5.1.1 Cycles

At a low level, all program cycles are implemented via branch instructions. The control flow recovery algorithm must translate these to high-level looping constructs, potentially resorting to the use of `goto` statements if these are supported by the target high-level language. As discussed in Section 2, some decompilers use advanced recovery algorithms that aim to minimise or completely avoid the use of `goto`, but can be hard to implement correctly. FuzzFlesh found several bugs related to incorrect decompilation of cycles.

As an example, Figure 4 shows a CFG and path that generated by FuzzFlesh that triggered a bug in FernFlower after being fleshed into Java bytecode; this is issue F1 in Table 2. The path is shown by solid edges, while other off-path edges are shown by dotted lines. Figure 5 shows the CFG and path corresponding to the Java program that was emitted by FernFlower. The decompiler output contains an infinite loop: FernFlower attempted to form the code into a `while` structure but did not correctly determine how the termination should occur.

Other cycle-related bugs included misdecompilations of loop-control variables by Ghidra.

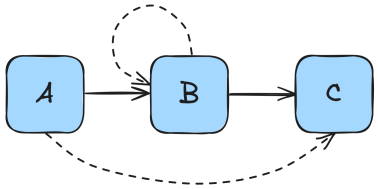

Figure 4 CFG and path of original program

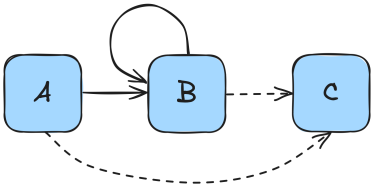

Figure 5 CFG and path of program decompiled by FernFlower containing an infinite loop

5.1.2 Switch statements

Three bugs were related to the decompilation of switch statements. Figures 6a and 6b show an example of a switch misdecompilation by Ghidra. The original program contains a jump table with five distinct instruction addresses (lines 4–8 of Figure 6a). However, the decompiled code only contains four switch cases (lines 3–6 of Figure 6b): there should be another case for the value 0. The problem was caused by an issue in the algorithm that reconstructs loops, which did not correctly handle the existence of a self-edge. This has been fixed by the Ghidra developers in response to our bug report [36].

```
1 switchD_001000b9::switchdataD_0010018c
2
3 0010018c d4 fe ff ff     uint     FFFFFED4h
4 00100190 98 ff ff ff     uint     FFFFFF98h
5 00100194 db ff ff ff     uint     FFFFFFDBh
6 00100198 51 ff ff ff     uint     FFFFFE51h
7 0010019c 76 ff ff ff     uint     FFFFFE76h
8
```

(a) Original jump table

```
1 void test(int* p_1, int* p_2) {
2 switch(p_2[local_2]){
3     case 1: goto block_D; ...
4     case 2: goto block_E; ...
5     case 3: goto block_I; ...
6     case 4: goto block_F; ...
7 }
8
```

(b) Decompiled switch statement

Figure 6 Illustration of a bug in Ghidra related to switch statements (issue G1 of Table 2). The jump table contains five entries, but only four of these are captured by switch cases in the decompiled code.

5.1.3 Java-specific irreducible control flow

As discussed in Section 4.2, Java bytecode allows irreducible control flow while Java does not. This means that decompilers must structure any irreducible control flow to produce valid Java. Irreducible control flow can be converted to reducible control flow using several techniques [8]. We found a number of bugs relating to this conversion process. These bugs are particularly interesting because they are unlikely to be found by program generation approaches that do not specifically aim to create unusual control flow structures.

An example of an irreducible CFG produced by FuzzFlesh is shown in Figure 7. This formed the basis for a program that caused a decompiler failure in FernFlower, which we reported in issue F2. Feedback in the discussion thread for the issue indicates that FernFlower should be able to handle simple cases of irreducible control flow such as this one. 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.

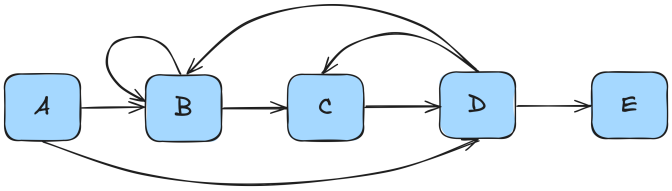

Figure 7 CFG that triggered a decompiler failure bug in FernFlower (issue F2 of Table 2)

5.2 RQ2: Efficacy of FuzzFlesh across languages and decompiler toolchains

FuzzFlesh found bugs in binary and Java decompilers but was not successful at finding bugs in C# decompilers. This is likely to be because CIL is fairly high level (unlike machine code), C# allows unrestricted control flow (unlike Java), and ILSpy does not appear to attempt to minimise the number of `goto` statements in the output code.

FuzzFlesh is not able to test decompilers whose output requires extensive engineering effort to recompile, for example angr [57].

6 Evaluation: Comparison to other approaches

In this section we answer research questions **RQ3** and **RQ4** set out in Section 1 and recapped below by evaluating the efficacy of FuzzFlesh and its ability to exercise the control flow recovery components of decompilers in comparison to other decompiler testing approaches.

RQ3 How thoroughly is FuzzFlesh able to test the control flow recovery components of decompilers in comparison to other decompiler testing approaches? (Section 6.2)

RQ4 To what extent can FuzzFlesh identify bugs that are different from those identified by other decompiler testing approaches? (Section 6.3)

6.1 Overview of decompiler testing approaches

We compare FuzzFlesh to two other tools[4] that aim to test decompilers:

- **DecFuzzer** [41], a binary decompiler fuzzer that targets open-source decompilers Ghidra and RetDec and commercial decompilers IDA-Pro and JEB3. It uses Csmith to generate an initial corpus of programs for de- and recompilation. Only a subset of the Csmith grammar is used to improve the likelihood of recompilability. Mutations are applied to any programs that successfully recompile, designed to yield programs that are equivalent modulo inputs [37]. Comparing results between original and mutated programs then provides a metamorphic test oracle [11].
- **JD-Tester** [43], a Java decompiler fuzzer that targets three Java decompilers: CFR, FernFlower, and JADX. JD-Tester uses two existing Java program generators: JavaFuzzer and Hephaestus. Each program is compiled to a class file, decompiled by the decompiler under test, and recompiled. The output of the original and recompiled program are compared to find decompilation bugs.

Table 3 Overview of decompiler testing tools

Tool	Generator	Decompilation scenarios			Control flow bugs
		bin to C	class to Java	CIL to C#	
FuzzFlesh	FuzzFlesh	✓	✓	✓	✓
DecFuzzer	Csmith	✓			
JD-Tester	JavaFuzzer / Hephaestus		✓		✓

[4] A recent paper evaluates binary decompilers using Dsmith [7], a fuzzer based on Csmith. The artifact was released in September 2024; due to time constraints we have not included it in our evaluation.

Table 3 summarises the characteristics of each testing approach, and highlights two key features of FuzzFlesh. First, FuzzFlesh covers multiple (low-level, high-level) language pairs, while the other tools only cover a single language pair. We are not aware of any decompiler testing tool that is able to target multiple languages. This is because of the simple nature of program generation in FuzzFlesh. For example, FuzzFlesh devotes only 86 lines of code to Java-specific program generation, while JavaFuzzer consists of 3,412 lines and Csmith contains around 40,000 lines of code.[5] This feature means that it is easy to add an additional language pair to FuzzFlesh. Most of the program generation logic is language-independent, which means it does not need to be repeated for each new language.

Second, Fuzzflesh targets control flow recovery. DecFuzzer and JD-Tester programs do contain complex control flow, however DecFuzzer is not able to find any control-flow related bugs. A detailed comparison of the nature of bugs found by each tool is given in Section 6.3.

6.2 RQ3: Comparison of thoroughness of control flow recovery testing

Code coverage measures the proportion of the decompiler source code that is executed ('covered') during testing. This provides a high-level indication of the extent to which a testing tool is exercising the decompiler. However, coverage is limited as an evaluation metric because (a) it does not provide any information on the relative importance of different lines, and (b) it is possible to execute code that contains a bug without actually triggering the bug. We therefore also include a comparison of actual bugs in Section 6.3.

Table 4 summarises the aggregate line coverage of each decompiler by FuzzFlesh, JD-Tester and DecFuzzer. The coverage is calculated by running each tool in fuzzing mode 8 hours on a 16-core 128GB RAM machine running Ubuntu 22.04. The "path known" and "path unknown" configurations of FuzzFlesh refer to whether the directions array that guides the path through the CFG are hard-coded into the program or passed at runtime respectively (see Section 3.3). The "JavaFuzzer" and "Hephaestus" configurations of JD-tester refer to the underlying Java program generator used (see Section 6.1). We report bytecode instruction coverage for Java and line coverage for C throughout; branch coverage follows a similar pattern but is lower for all tools (not reported here but available in the accompanying artifact).

We summarise our main findings, before giving more specific detail on the experimental setup and results for binary decompilers and Java decompilers in Sections 6.2.1 and 6.2.2, respectively.

Overall, coverage of each decompiler is highest for the language-specific testing tools JD-Tester and DecFuzzer. This is expected because they are far more syntactically expressive than the FuzzFlesh programs. Nevertheless, despite its limited syntax, FuzzFlesh achieves reasonable overall coverage of up to 36% of Ghidra, 31% of CFR, 42% of FernFlower, and 37% of JADX. The different FuzzFlesh generation modes generally achieve similar levels of overall coverage, as do the different program generators used by JD-Tester.

As part of these experiments we also computed coverage data at checkpoints of 2 and 4 hours of each 8 hour fuzzing run, and found that coverage had already plateaued after 2 hours for all tools.

[5] Lines of code were measured using the cloc tool.

Table 4 Instruction coverage comparison for an 8 hour fuzzing run (%)

Decompiler	Testing tool				
	FuzzFlesh path known	FuzzFlesh path unknown	DecFuzzer	JD-Tester JavaFuzzer	JD-tester Hephaestus
Ghidra	36	31	39	-	-
CFR	31	30	-	53	42
FernFlower	42	41	-	55	48
JADX	37	36	-	50	44

6.2.1 Binary decompiler coverage

Ghidra is implemented in a combination of Java and C++. We restrict our coverage comparison to the decompiler component, which is implemented entirely in C++. We use gcov and gcovr to analyse coverage.

DecFuzzer generates programs using Csmith. These are compiled to binaries, decompiled using Ghidra, and recompiled. Successfully recompiled programs are used as seeds for equivalence modulo inputs testing. DecFuzzer aims to improve the likelihood of successful recompilation by restricting Csmith to produce only one function (besides `main()`), limiting the complexity of expressions, and excluding arrays, structs, unions, pointers and various other language features. The restricted Csmith command in full is:

```
csmith -no-arrays -no-structs -no-unions -no-safe-math -no-pointers
-no-longlong -max-funcs 1 -max-expr-complexity 5
```

As discussed further in Section 6.3.2, despite these restrictions DecFuzzer's use of Csmith does not lead to programs that can be successfully recompiled after decompilation by recent versions of Ghidra.

In addition to overall coverage, we compare the unique coverage of each testing tool. Table 5 shows that FuzzFlesh is able to cover around 12,700 lines within Ghidra that DecFuzzer programs are not able to cover. In particular, FuzzFlesh covers parts of the control flow restructuring algorithms (within `block.cc`) that DecFuzzer does not cover.

Table 5 Unique line coverage comparison for the binary decompiler Ghidra

Decompiler	Lines covered by				Total lines
	FuzzFlesh only	DecFuzzer only	Both	Neither	
Ghidra	12,700	63,00	12,800	39,800	71,600

6.2.2 Java decompiler coverage

We use the Jacoco code coverage tool to measure coverage of the bytecode-to-Java decompilers, which are implemented in Java. Fuzzing mode involves program generation, compilation, decompilation, recompilation, and test oracle evaluation. Neither of these tools requires an initial seed corpus for test generation. We then gather the programs that were generated by

each testing tool and decompile them using versions of CFR, FernFlower, and JADX that are instrumented with Jacoco. The motivation for separating the coverage measurement from the previous step is that coverage instrumentation can slow program execution, which could lead to underestimating the coverage achieved in a given time budget.

In addition to overall coverage, we also analyse the unique coverage of each tool. Although JD-Tester achieves higher overall coverage, FuzzFlesh is able to cover some parts of each decompiler's code that are not covered by JD-tester. Table 6 shows the number of instructions that are covered by only FuzzFlesh or only JD-Tester. The instructions covered by FuzzFlesh that are not covered by JD-Tester mainly relate to control-flow recovery. For example, FuzzFlesh covers instructions in the following classes that are not covered by JD-Tester (non-exhaustive list):

- CFR: SwitchReplacer, LoopIdentifier, JumpsIntoDoRewriter, and JoinBlocks
- FernFlower: IrreducibleCFGDeobfuscator, ControlFlowGraph, SwitchStatement, JumpInstruction, and DomHelper (which calculates node dominance),
- JADX: BlockSplitter, BlockProcessor, FixMultiEntryLoops, SwitchInsn, and LoopRegionVisitor

Table 6 Unique bytecode instruction coverage comparison for Java decompilers

Decompiler	Instructions covered by				Total instructions
	FuzzFlesh only	JD-Tester only	Both	Neither	
CFR	800	35,300	51,000	79,000	166,100
FernFlower	4,700	15,900	34,900	38,300	93,800
JADX	1,300	17,100	43,400	60,400	122,200

6.3 RQ4: The kinds of bugs found by FuzzFlesh vs. other approaches

While code coverage provides a useful overview of the potential capabilities of decompiler testing tools, it is only a proxy for the ideal fuzzer performance measurement: the potential number of 'useful' bugs that can be found [33]. In this section we evaluate the bug-finding ability of FuzzFlesh in comparison to DecFuzzer and JD-Tester.

6.3.1 Bugs in all decompilers

FuzzFlesh is the only decompiler testing tool that we are aware of that is able to find bugs in decompilers that target different language-pairs. In this respect it has a higher bug-finding potential than other tools, since it can be easily extended to other languages in the future, while DecFuzzer and JD-Tester both rely on language-specific program generators.

6.3.2 Bugs in binary decompilers

Problems with recompilability within DecFuzzer mean that we cannot perform a head-to-head comparison of bug-finding ability. No DecFuzzer programs are recompilable using the latest Ghidra plugin, which means that they are not able to identify misdecompilations. DecFuzzer runs on Radare2, which is a reverse-engineering framework that uses a Ghidra

plugin for decompilation. However, multiple compiler errors occur when trying to recompile the decompiled programs within DecFuzzer's workflow. The errors include trivial issues such as 'unknown type name uint' but also more complex issues such as the incorrect number of arguments to functions. Modifying DecFuzzer to support recompilability would likely require a significant amount of engineering work to address the latest Ghidra output format. This illustrates the difficulty of ensuring recompilability. We contacted the authors of DecFuzzer, and they shared with us the commit used for Radare2 and their best estimate of the versions of the Ghidra plugin, r2ghidra, and other tools that were used, however some of the original artifacts from the paper (such as the contemporary binaries) are no longer available. It has therefore not been possible to reconstruct a working version of the tool to perform a direct bug-finding comparison due to incompatibilities between the historic Ghidra versions and current binaries. The simple syntax used by FuzzFlesh means that we are less likely to encounter this type of recompilation issues in the future, because we expect that future versions of all tools will continue to support basic syntax.

Considering the original bugs reported to have been found by DecFuzzer [41] and the actual bugs found by FuzzFlesh: FuzzFlesh mostly identifies control flow related bugs, while DecFuzzer mainly identified bugs relating to type recovery and optimisations, and was not able to find any bugs within the control flow recovery module of Ghidra or any other binary decompiler. FuzzFlesh found a total of 6 bugs in Ghidra, at least three of which are confirmed by developers to be directly related to control flow recovery. While the sample size is small, this indicates that FuzzFlesh is likely to be capable of finding control flow related bugs that DecFuzzer would not identify.

6.3.3 Bugs in Java decompilers

JD-Tester is able to find control flow related bugs. This is because (a) the programs generated by JD-Tester include control flow structures, and (b) decompiling bytecode to Java is easier than decompiling machine code (for example because instructions and data are separate). This means that more programs are recompilable than in the case of binary decompiler testing, which enables the detection of control flow misdecompilations. However, some of the bugs identified by FuzzFlesh are unlikely to be discoverable by JD-Tester due to the complex control flow employed. To compare the tools directly, we evaluate whether JD-Tester is able to find the control flow related bugs that FuzzFlesh identified in Java decompilers. We restrict the comparison to bugs that have been fixed.

To do this, we rely on the notion of 'correcting commit' used in an empirical study of compiler testing techniques [9]: the commit in which a bug is fixed (so that the bug manifests before the commit but not after). This is essential because otherwise, especially in the case of misdecompilation bugs, it is not possible to directly identify which underlying bug is triggered by a bug-inducing test program. Our evaluation works as follows:

- We identify correcting commits associated with two CFR bugs and one JADX bug.
- We run JD-Tester on a preceding commit (i.e. one for which we know that the bug is present) for 2 hours and identify any failing test cases. These are potential bug-triggering programs that could identify the bug.
- We run the failing test cases on the correcting commit. If any test now passes, then we infer that its original failure was caused by the bug. Such a test identifies the bug.

We found that none of the JD-Tester programs that failed on the preceding commit passed on the correcting commit, indicating that none of the potential bug-triggering test programs produced by JD-Tester were caused by these specific bugs. Although the time

budget is limited, in contrast FuzzFlesh was able to identify these bugs almost immediately. These results suggest that JD-Tester is not able to identify some bugs that are found by FuzzFlesh. This evaluation is based on a very small sample size of the two correcting commits (recall that we focus here only on bugs that have been fixed), however the coverage analysis provides supporting evidence that JD-Tester may not be able to trigger bugs in some parts of the decompiler code. While neither of the FernFlower bugs have been fixed, one relates to irreducible control flow restructuring. This would certainly not be detectable by JD-Tester because its programs do not contain irreducible control flow. Similarly, the Jadx bug would not be detectable by JD-Tester because it related to irreducible control flow restructuring.

To summarise, in prior work JD-Tester has been shown to perform well at finding a range of bugs in Java decompilers, including some control flow bugs [41]. However, FuzzFlesh was able to identify some previously-unknown and potentially significant bugs that JD-Tester appears unable to find.

7 Related Work

Decompiler testing. Historically, the main evaluation metrics used to assess decompilers were focused on readability, for example code compactness and complexity (measured by cyclic complexity and the number of `goto`s in a decompiled program). In recent years there has been an increasing focus on correctness within decompiler research [41].

As discussed in Section 6, as far as we are aware FuzzFlesh is the first random program generator that is specifically targeted at decompilers. JD-Tester [43] uses a differential testing approach with the pre-existing program generator JavaFuzzer [66]. It finds 62 bugs in three Java decompilers: CFR, FernFlower, and JADX. A vast majority of the bugs reported in this study are decompilation failures and crashes, most of which have not been fixed by developers. This highlights the importance of focusing on misdecompilations that are more likely to present a serious problem to users.

DecFuzzer [41] evaluates leading commercial and open-source decompilers using the existing program generator Csmith [68] to generate a corpus of programs. It uses equivalence modulo inputs testing [37] to compare the performance of four binary decompilers (Ghidra [50], RetDec [3], IDA-Pro [27] and Jeb3 [53]). They find 13 bugs, however they report that this required 'extensive manual effort' from expert security analysts.

We recently became aware of D-Helix [71], which proposes a general framework for decompiler testing that is intended to assist developers with decompilation tasks, and identifies 25 bugs across binary decompilers Ghidra and angr. D-Helix detects semantic discrepancies between the binary input and the decompiled output by performing symbolic execution on both the original binary and a recompiled decompiled output. It also includes a debugging tool that helps to pinpoint the source of the discrepancy. In future work it would be interesting to compare FuzzFlesh experimentally with D-Helix.

Compiler testing. We have used techniques from the wider compiler fuzzing domain. In recent years there has been a surge of interest in randomised compiler testing techniques, for languages such as C [68, 42, 37], Java / Java bytecode [12, 69, 62], various GPU programming languages [38, 14, 15], Rust [56, 61], Verilog [26], as well as techniques for testing machine learning compilers [39, 40, 44], and techniques that use large language models for randomised compiler testing [67].

Our approach is directly inspired work on testing compilers for SPIR-V, a low level language for GPU programming [34], in which skeleton control flow graphs generated by the

Alloy tool [31] are fleshed into executable SPIR-V programs. Our approach is somewhat related to a compiler testing method called *skeletal program enumeration* where the control flow structure of an existing program is used as the basis for generating a large set of programs that populate this structure with different uses of program variables [70]. Other program generation approaches include creating full programs from scratch [68, 42] and mutating existing full programs [37, 15, 49, 38]. A comprehensive review of current state-of-the-art compiler testing approaches is given by Chen et al. [10]. Recently, Large Language Models (LLMs) have been used to generate programs in a range of languages based on user prompts [63]. However, these programs are only designed to detect crash bugs due to the difficulty of ensuring that LLM-generated programs are free from undefined behaviour (UB). We are not aware of any other program generator targeting miscompilations or misdecompilations that is designed to be easily extensible to other languages.

Strategies for constructing test oracles include differential testing [46, 38, 68, 42], where the output of similar compilation systems are compared, and metamorphic testing [11, 37, 49, 38, 15], which specifies how changes in the input source code should affect the output code produced by the compiler.

8 Conclusion and future work

CFG-based program generation is a novel decompiler testing approach that we have used to identify 12 bugs across four decompilers covering two languages. Our implementation, FuzzFlesh, demonstrates the feasibility of sharing a language-independent random CFG and path generator across several different languages with a small amount of incremental programming time.

Areas for further work include alternative CFG and path generation algorithms. For example, mining a corpus of graph structures from real-world programs and mutating them. This would ensure that realistic control flow structures are included in test cases. In addition, more sophisticated path generation algorithms could be created to ensure that each generated CFG is well-explored. For example, repeating path cycles to ensure that loop-like behaviour is included in the test programs.

CFG-based program generation also has the potential to be applied to find bugs in general-purpose compilers. The self-checking nature of the test cases mean that only a single compiler is required, and the ease of adding new languages mean that the approach could be used for newly established compiler toolchains. It is most likely to be useful for testing the control flow optimisation passes of compilers. However, our current implementation is unlikely to produce programs that will detect bugs in mainstream compilers without further work. Decompiler control flow recovery is challenging because it involves imposing structure onto spaghetti code: the more bizarre the control flow, the more challenging decompilation is. Compiling complex control flow has different challenges: it is permissible for a compiler to produce spaghetti code as output, and if the control flow of the source program is too complex then it is possible that the compiler will not attempt to optimise it. Therefore, further work in this area could explore graph generation approaches that may be more likely to find bugs in compilers. For example, graphs could be generated in a way that will be sufficiently complex to thoroughly exercise control flow optimisation compiler passes, but not so complex that compilers will not attempt to perform optimisation.

References

1 Jaime Acosta and Daniel Krych. Hands-on cybersecurity studies: Uncovering and decoding malware communications-malware analysis with Ghidra. Technical Report ARL-TR-9129, Computational and Information Sciences Directorate, DEVCOM Army Research Laboratory, 2020. Accessed: 2025-04-25. URL: https://apps.dtic.mil/sti/trecms/pdf/AD1119396.pdf.

2 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

3 Avast. RetDec: A retargetable machine-code decompiler based on LLVM, 2022. Accessed: 2025-04-25. URL: https://github.com/avast/retdec.

4 Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy SAILR! there is no need to DREAM of C: A Compiler-Aware structuring algorithm for binary decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 361–378, Philadelphia, PA, August 2024. USENIX Association. Accessed: 2025-04-25. URL: https://www.usenix.org/system/files/usenixsecurity24-basque.pdf.

5 Lee Benfield. CFR - another Java decompiler, 2025. Accessed: 2025-04-25. URL: https://www.benf.org/other/cfr/.

6 David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, Washington, D.C., August 2013. USENIX Association. Accessed: 2025-04-25. URL: https://www.usenix.org/system/files/conference/usenixsecurity13/sec13-paper_schwartz.pdf.

7 Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 491–502, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3650212.3652144.

8 Larry Carter, Jeanne Ferrante, and Clark D. Thomborson. Folklore confirmed: reducible flow graphs are exponentially larger. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 106–114. ACM, 2003. doi:10.1145/604131.604141.

9 Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 180–190. ACM, 2016. doi:10.1145/2884781.2884878.

10 Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1):4:1–4:36, 2021. doi:10.1145/3363562.

11 T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998. Accessed: 2025-04-25. URL: https://www.cse.ust.hk/faculty/scc/publ/CS98-01-metamorphictesting.pdf.

12 Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 85–99. ACM, 2016. doi:10.1145/2908080.2908095.

13 Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw. Pract. Exp.*, 25(7):811–829, 1995. doi:10.1002/SPE.4380250706.

14 Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA):93:1–93:29, 2017. doi:10.1145/3133917.

15 Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1017–1032. ACM, 2021. doi:10.1145/3453483.3454092.

16 Luke Dramko, Jeremy Lacomis, Edward J. Schwartz, Bogdan Vasilescu, and Claire Le Goues. A taxonomy of C decompiler fidelity issues. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. Accessed: 2025-04-25. URL: https://www.usenix.org/system/files/usenixsecurity24-dramko.pdf.

17 Lukás Durfina, Jakub Kroustek, and Petr Zemek. PsybOt malware: A step-by-step decompilation case study. In Ralf Lämmel, Rocco Oliveto, and Romain Robbes, editors, *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 449–456. IEEE Computer Society, 2013. doi:10.1109/WCRE.2013.6671321.

18 Mike Van Emmerik and Trent Waddington. Using a decompiler for real-world source recovery. In *11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004*, pages 27–36. IEEE Computer Society, 2004. doi:10.1109/WCRE.2004.42.

19 P Erdös and A Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

20 Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Decompilation of Java bytecode to Prolog by partial evaluation. *Inf. Softw. Technol.*, 51(10):1409–1427, 2009. doi:10.1016/J.INFSOF.2009.04.010.

21 Amber Gorzynski. FuzzFlesh, 2024. URL: https://github.com/ambergorzynski/control_flow_fleshing.

22 Amber Gorzynski. FuzzFlesh artifact, 2024. URL: https://github.com/ambergorzynski/control_flow_fleshing/tree/ecoop-2025.

23 James Gosling and Henry McGilton. The Java Language Environment, 1996. Accessed: 2025-04-25. URL: https://www.oracle.com/java/technologies/simple-familiar.html.

24 Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1176–1186. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00120.

25 Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled C code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 637–651, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3320269.3384766.

26 Yann Herklotz and John Wickerson. Finding and understanding bugs in FPGA synthesis tools. In Stephen Neuendorffer and Lesley Shannon, editors, *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, pages 277–287. ACM, 2020. doi:10.1145/3373087.3375310.

27 Hex-Rays. IDA Pro, 2025. Accessed: 2025-04-25. URL: https://hex-rays.com/ida-pro/.

28 Joshua Homan. Deobfuscating Python bytecode, 2016. Accessed: 2025-04-25. URL: https://cloud.google.com/blog/topics/threat-intelligence/deobfuscating-python.

29 ICSharpCode. ILSpy, 2025. Accessed: 2025-04-25. URL: https://github.com/icsharpcode/ILSpy.

30 IntelliJ. Fernflower, 2025. Accessed: 2025-04-25. URL: https://github.com/JetBrains/intellij-community/blob/master/plugins/java-decompiler/engine/src/org/jetbrains/java/decompiler.

31 Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019. doi:10.1145/3338843.

32 JADX Authors. JADX, 2025. Accessed: 2025-04-25. URL: https://github.com/skylot/jadx.

33 George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2123–2138. ACM, 2018. doi:10.1145/3243734.3243804.

34 Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. Taking back control in an intermediate representation for GPU computing. *Proc. ACM Program. Lang.*, 7(POPL):1740–1769, 2023. doi:10.1145/3571253.

35 P. L. Krapivsky and S. Redner. Organization of growing random networks. *Physical Review E*, 63(6), 5 2001. doi:10.1103/physreve.63.066123.

36 Ryan Kurtz. Git commit fixing a bug found by fuzzflesh in Ghidra, 2024. Accessed: 2025-04-25. URL: https://github.com/NationalSecurityAgency/ghidra/commit/6ede2b498f75e29c13fe64d3ce7549e90e17aa2b.

37 Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014. doi:10.1145/2594291.2594334.

38 Christopher Lidbury and Alastair F. Donaldson. Dynamic race detection for C++11. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 443–457. ACM, 2017. doi:10.1145/3009837.3009857.

39 Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. NNSmith: Generating diverse and valid test cases for deep learning compilers. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 530–543. ACM, 2023. doi:10.1145/3575693.3575707.

40 Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–26, 2022. doi:10.1145/3527317.

41 Zhibo Liu and Shuai Wang. How far we have come: Testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 475–487, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3395363.3397370.

42 Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with yarpgen. *Proc. ACM Program. Lang.*, 4(OOPSLA):196:1–196:25, 2020. doi:10.1145/3428264.

43 Yifei Lu, Weidong Hou, Minxue Pan, Xuandong Li, and Zhendong Su. Understanding and finding Java decompiler bugs. *Proc. ACM Program. Lang.*, 8(OOPSLA1), 4 2024. doi:10.1145/3649860.

44 Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. Fuzzing deep learning compilers with HirGen. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 248–260. ACM, 2023. doi:10.1145/3597926.3598053.

45 Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery - A case study. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*, pages 602–615. ACM, 2022. doi:10.1145/3488932.3497764.

46 William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10(1):100–107, 1998. Accessed: 2025-04-25. URL: http://www.dtjcd.vmsresource.org.uk/pdfs/dtj_v10-01_1998.pdf.

47 Jonathan Meyner and Daniel Reynaud. Jasmin, 2004. Accessed: 2025-04-25. URL: https://jasmin.sourceforge.net/.

48 Omid Mirzaei, Roman Vasilenko, Engin Kirda, Long Lu, and Amin Kharraz. SCRUTINIZER: detecting code reuse in malware via decompilation and machine learning. In Leyla Bilge, Lorenzo Cavallaro, Giancarlo Pellegrino, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 18th International Conference, DIMVA 2021, Virtual Event, July 14-16, 2021, Proceedings*, volume 12756 of *Lecture Notes in Computer Science*, pages 130–150. Springer, 2021. doi:10.1007/978-3-030-80825-9_7.

49 Kazuhiro Nakamura and Nagisa Ishiura. Random testing of C compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2016, Jeju, South Korea, October 25-28, 2016*, pages 676–679. IEEE, 2016. doi:10.1109/APCCAS.2016.7804063.

50 National Security Agency. Ghidra - reverse software engineering framework, 2025. Accessed: 2025-04-25. URL: https://ghidra-sre.org/.

51 Oracle. The Java virtual machine instruction set, 2025. Accessed: 2025-04-25. URL: https://docs.oracle.com/javase/specs/jvms/se24/html/jvms-6.html.

52 Jihee Park, Sungho Lee, Jaemin Hong, and Sukyoung Ryu. Static analysis of JNI programs via binary decompilation. *IEEE Trans. Software Eng.*, 49(5):3089–3105, 2023. doi:10.1109/TSE.2023.3241639.

53 PNF Software. JEB3, 2013. Accessed: 2025-04-25. URL: https://www.pnfsoftware.com/blog/category/jeb3/.

54 Bernardo Quintero. From assistant to analyst: The power of Gemini 1.5 Pro for malware analysis, 2024. Accessed: 2025-04-25. URL: https://cloud.google.com/blog/topics/threat-intelligence/gemini-for-malware-analysis.

55 Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. Automatically mitigating vulnerabilities in x86 binary programs via partially recompilable decompilation. *CoRR*, abs/2202.12336, 2022. arXiv:2202.12336.

56 Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. RustSmith: Random differential compiler testing for rust. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1483–1486. ACM, 2023. doi:10.1145/3597926.3604919.

57 Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016. doi:10.1109/SP.2016.17.

58 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

59 Daniel Votipka, Seth M. Rabin, Kristopher K. Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. An observational investigation of reverse engineers' processes. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1875–1892. USENIX Association, 2020. Accessed: 2025-04-25. URL: https://www.usenix.org/system/files/sec20-votipka-observational.pdf.

60 Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. Software protection on the go: a large-scale empirical study on mobile app obfuscation. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 26–36. ACM, 2018. doi:10.1145/3180155.3180169.

61 Qian Wang and Ralf Jung. Rustlantis: Randomized differential testing of the Rust compiler. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1955–1981, 2024. doi:10.1145/3689780.

62 Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. JITfuzz: Coverage-guided fuzzing for JVM just-in-time compilers. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 56–68. IEEE, 2023. doi:10.1109/ICSE48619.2023.00017.

63 Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal fuzzing with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 126:1–126:13. ACM, 2024. doi:10.1145/3597503.3639121.

64 Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 158–177. IEEE Computer Society, 2016. doi:10.1109/SP.2016.18.

65 Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. Accessed: 2025-04-25. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/11_4_2.pdf.

66 Andrey Yakovlev, Mohammad R. Haghighat, and Dmitry Khukhro. AzulSystems/JavaFuzzer, 2018. Accessed: 2025-04-25. URL: https://github.com/AzulSystems/JavaFuzzer.

67 Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. Whitefox: White-box compiler fuzzing empowered by large language models. *Proc. ACM Program. Lang.*, 8(OOPSLA2):709–735, 2024. doi:10.1145/3689736.

68 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1993498.1993532.

69 Zhiqiang Zang, Nathan Wiatrek, Milos Gligoric, and August Shi. Compiler testing using template java programs. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 23:1–23:13. ACM, 2022. doi:10.1145/3551349.3556958.

70 Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 347–361. ACM, 2017. doi:10.1145/3062341.3062379.

71 Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian. D-Helix: A generic decompiler testing framework using symbolic differentiation. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. Accessed: 2025-04-25. URL: https://www.usenix.org/system/files/usenixsecurity24-zou.pdf.