FLUX: Finding Bugs with LLVM IR Based Unit Test Crossover

by

Eric Liu

A thesis submitted in conformity with the requirements
for the degree of Master of Science

Department of Computer Science
University of Toronto

FLUX: Finding Bugs with LLVM IR Based Unit Test Crossover

Eric Liu
Master of Science

Department of Computer Science
University of Toronto
2023

Abstract

Optimizing compilers are an indispensable component of the development of any modern software. Unfortunately, as with any sufficiently complex code, bugs in compilers are common. Among the most serious, are bugs in compiler optimizations, which can cause unexpected behavior in the compiled binary. This thesis proposes FLUX, a fuzzer that is designed to generate test cases that explore new execution paths through compiler optimizations. We hypothesize that exploring new paths will lead to newly found bugs. We design FLUX such that it explores new paths by combining high-coverage test cases to form new tests. FLUX implements this test combination with two novel crossover mutations. Our evaluation on the LLVM compiler indicates that FLUX is able to expand upon the path coverage of the LLVM unit test suite and discover new bugs in LLVM optimization passes. In total, FLUX discovers 25 unique bugs in LLVM's active development branch.

Acknowledgements

I am truly grateful for the support, guidance, and encouragement provided by my supervisor, Professor David Lie. Thank you for being a shepherd of reason throughout the course of my degree. Your inquisitive, level-headed, and patient approach to research was a major source of inspiration for me.

I am thankful for my past and present labmates John, Wei, Yuqin, Tony, and Xiangyu. The discussions we had were insightful and I've greatly enjoyed getting to know each of you these last two years. I would also like to thank my friend and fellow graduate student Towaki, for being a frequent sounding board for my many bad ideas.

Thank you to my parents for being there during the rough parts. I could not have completed my degree (while maintaining my sanity) without your support.

Finally, I would like to thank the University of Toronto and the Government of Ontario for providing the financial support that facilitated this whole project.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Compilers are an indispensable component of any modern development pipeline, responsible for translating human-readable source code into machine-executable instructions. The most widely distributed and used compilers in today's software landscape regularly apply a wide array of optimizations during the lowering process. Due to the constant forward march of performance improvement and incremental language and architecture support, popular optimizing compilers like GCC and Clang/LLVM have grown to contain lines of code on the order of tens of millions [15]. As with any code base that is highly complex, bugs in compilers are difficult to avoid. Despite this, compilers are often blindly trusted by end users and are expected to preserve source program semantics. Unfortunately, these assumptions are not always safe, despite the best efforts of compiler developers.

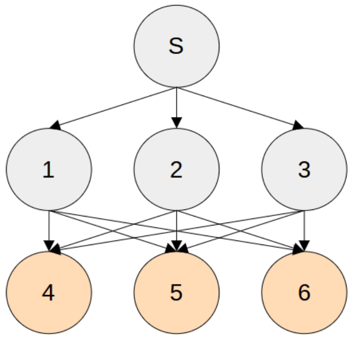
Compiler bugs can range from mildly inconvenient compilation time crashes to potentially dangerous miscompilations that can cause unexpected behaviour in the resultant binary. LLVM for instance reports hundreds of crash and misoptimization bugs each month [34, 30]. Undetected bugs in LLVM have real-world consequences. The LLVM infrastructure has been integrated into consumer-facing technologies, such as the Android NDK and PS4 SDK [26]. Further, due to LLVM's modular infrastructure and mid-end optimizations, it is able to support many general-purpose and GPU language frontends and a bevy of backends. This in turn leads to a user base of millions. Beyond quality of service degradation and occasional unexpected behaviour in the binary, bugs in LLVM can also be exploitable. In one case, researchers have demonstrated that a compiler miscompilation can enable an adversary to insert a deniable backdoor into the Unix `sudo` command [3]. Hence, ensuring that a compiler like LLVM remains bug-free is a difficult but crucial effort.

In order to find compiler bugs, previous work has employed fuzzing to automatically generate bug-triggering test cases. General-purpose software fuzzers traditionally involve generating malformed and random inputs to pass to a program under test [19, 1, 28]. The program is monitored to detect exceptions such as crashes or memory errors. Although effective for programs that accept arbitrary input, it is very unlikely that unrestrained random fuzzing is able to fully stress a compiler, much less find a bug in it. Due to syntactic and semantic constraints, any fuzzer that does not respect source language conventions will quickly become stuck in the compiler's lexer, parser, and type checker. These constraints make it difficult for traditional fuzzing metrics like coverage guidance to be effective.

To overcome these stringent requirements, existing work has focused foremost on automatically generating test cases that the compiler will actually accept while putting the issue of full compiler exploration secondary. Popular compiler fuzzers have tried a variety of methods to generate valid test cases. Some examples being with the aid of a grammar [32, 14], heuristics [23, 24], and with deep learning [6, 13]. In contrast to generating input programs from scratch, other compiler fuzzers mutate existing test cases to form new inputs [11, 12, 29, 33]. These approaches generally mutate a valid input program such that the mutated test has its validity preserved.

Although successful in finding bugs, previous work is not able to fully explore the entire space of input programs to their target compilers [14]. In fact, the range of programs that most fuzzers are able to generate is bounded and biased. This means that after sufficient time, compiler fuzzers saturate their entire output space. In other words, since the test case generation schemes of existing fuzzers only model a subset of all language features, after exploiting the full distribution of potential outputs, these fuzzers are no longer able to exercise new paths in the compiler. One such unsaturated region is a compiler's optimization middle-end. Existing work has largely relied on heuristics and intuition in order to stress compiler optimizations. For instance, YARPGen [14], restricts some portions of its generation scheme to only use arithmetic, logical, and bitwise operations. Their intuition is that dense clusters of these three operations will more effectively trigger certain peephole optimizations, like common subexpression elimination. Unfortunately, these types of heuristics either suffer the same bias as their parent fuzzer or are narrow in scope, only focusing on specific optimizations. Targeting less explored regions of the compiler is a non-trivial task.

Instead of generating test cases from the bottom up or inferring models of the source code in order to determine suitable tests, our work makes a key observation: compilers like LLVM already have a vast suite of unit tests that are *known* to test most components of the compiler. Further, in the case of LLVM, each optimization in the middle-end has a corresponding set of unit tests that boast a very high code coverage[1]. This work aims to leverage these feature-rich unit tests in order to generate new test cases.



Line Coverage	Path Coverage
{S,1,4}	{S,1,4}
{S,2,5}	{S,1,5}
{S,3,6}	{S,1,6}
	{S,2,4}
	{S,2,5}
	{S,2,6}
	{S,3,4}
	{S,3,5}
	{S,3,6}

Figure 1.1: Example control-flow graph (CFG) comparing line and path coverage. The table to the right of the CFG demonstrates the difference between full line coverage and full path coverage

We note that despite the high unit test line coverage, new optimization bugs continue to be found. In fact, previous studies have shown that basic coverage metrics like function and line coverage (i.e. whether each function or line has been executed), are weakly correlated with actual bugs [2]. This

[1]A good example are the instruction combine optimizations in LLVM. With the standard optimization pipeline, the unit tests are already able to reach around 90% line coverage. We discuss more details on the LLVM unit tests in §2.1.

means that it is not enough to stress compiler optimization with a single execution pass through each code segment, we must also explore the same optimization code in varying orders and with varying input programs. In essence, as opposed to line or function coverage, we believe that increasing *path coverage* (e.g. Fig. 1.1) through the compiler will lead to more found bugs.

Specifically, we conjecture that generating test cases by combining these high-coverage unit tests will allow us to stress execution paths through the compiler optimizer that previous work could not. We denote these test case combination techniques as unit test *crossovers*. We propose two new crossover mutations. To target intraprocedural optimizations, we introduce a mutation that composes unit tests in a manner that is akin to *inlining*. As for interprocedural optimizations, we introduce a mutation that we denote as a *sequencing* crossover.

We implement FLUX (**F**inding Bugs with **L**LVM IR Based **U**nit Test Cross(**X**)over) on top of libFuzzer [28]. The FLUX fuzzer targets the LLVM compiler and generates LLVM IR test cases to stress its middle end optimizations. We compile LLVM with AddressSanitizer [27] to detect memory errors during fuzzing and rely on Alive2 to detect compiler miscompilations [17]. Our evaluation of FLUX yields promising results. We show that FLUX's crossover mutations are able to find new path coverage through LLVM's optimzer. Furthermore, when used in longer fuzzing runs, FLUX is able to find a significant number of compiler bugs. Over the course of three weeks, FLUX reports 25 unique bugs in LLVM optimizations, 24 of which are various types of crashes in the optimization code and 1 being a miscompilation bug that is detected by Alive2.

1.1 Contributions

To summarize, this thesis makes the following contributions:

- The notion of combining high line coverage test cases in order to explore new path coverage.
- Two novel crossover mutations, the *inlining* and the *sequncing* mutations, which are capable of exploring new paths through existing unit test code.
- A prototype implemention of FLUX, a fuzzer that implements the inlining and sequencing mutations on top of libFuzzer.
- An evaluation of FLUX's ability to explore new path coverage and find new bugs in LLVM's optimization passes. In total we discover **25** bugs during our evaluation.

1.2 Thesis structure

The structure of the rest of this thesis is as follows. We first introduce relevant background on LLVM in chapter 2. Then we discuss related bug studies and compiler bug-finding techniques in chapter 3. In chapter 4 we motivate and then elucidate FLUX's design. This includes a detailed explanation of FLUX's crossover mutations. Chapter 5 dives into specifics of how FLUX was implemented, and chapter 6 evaluates the bug-finding effectiveness of FLUX. We mention limitations of the FLUX fuzzer in chapter 7 and finally conclude in chapter 8.

Chapter 2

Background

This chapter will introduce relevant background information on LLVM, which serves as the foundation for the rest of the thesis. We briefly introduce LLVM's infrastructure in §2.1, then discuss the LLVM IR in §2.2 and conclude with a description of LLVM's unit tests in §2.3

2.1 LLVM

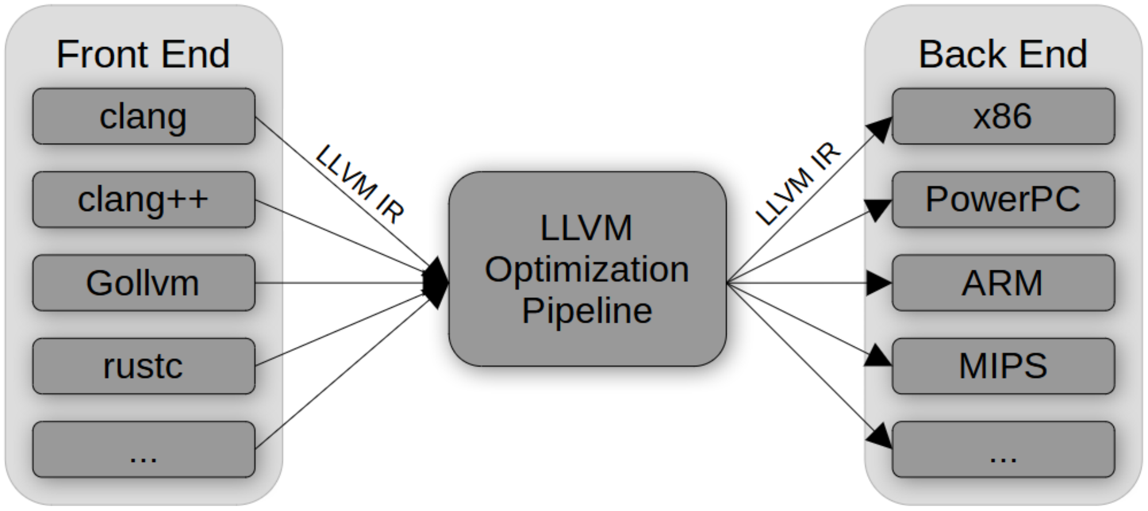


Figure 2.1: LLVM's infrastructure has a modular frontend, middle-end, and backend

LLVM is a compiler infrastructure that supports an ecosystem of tools, frontends, backends, and middle-end optimizations. Much of LLVM's appeal lies in the modularity of its components, which is facilitated by LLVM's intermediate representation. As demonstrated in figure 2.1, any source language frontend can take advantage of LLVM's middle-end optimizations as long the source language is compilable to LLVM's intermediate representation (IR). The middle-end optimization pipeline `opt` optimizes the input IR through a series of transformation passes. LLVM aims to make each pass fairly modular by ensuring that each output leaves the LLVM IR in a valid state. This modularity allows an end-user to reorder many of the transformations at will. The output of `opt` is optimized LLVM IR, which is then fed into the backend. The LLVM backend, in turn, is capable of cross-compiling the LLVM IR to a variety of target architectures.

2.2 LLVM IR

LLVM IR is the intermediate representation that is used by LLVM's optimization passes. Although "lower level" than source code, LLVM IR still contains a significant amount of semantic information that facilitates the many optimizations in the middle-end. For instance, LLVM IR is strongly typed and supports most types that a C programmer would be familiar with[1]. A group of straight line instructions form a *basic block,* and the IR models control flow with branch instructions that jump from the end of one basic block to the beginning of another. Each basic block is located inside a function, of which the control flow begins at an entry basic block. Another feature of the LLVM IR is that it must adhere to static single assignment (SSA) form. This means that each variable is assigned exactly once. SSA form is useful because it significantly simplifies control and data flow analysis. For values that could originate from more than one basic block, LLVM IR handles this elegantly with the use of the PHI instruction. The PHI instruction selects the value that corresponds with the current basic block's predecessor block.

Component	**Total LOC**	**Missed LOC**	**Coverage (%)**
AggressiveInstCombine	898	58	93.54
InstCombine	27922	2936	89.48
IPO	35731	20140	43.63
Scalar	53534	30692	42.67
Vectorize	25804	6272	75.69
Utils	39420	16688	57.67
Total	**183309**	**76786**	**58.11**

Table 2.1: Coverage achieved by the transformation unit test suite. Displayed above is the line coverage of the main optimization categories of LLVM. IPO denotes Interprocedural Optimizations. Scalar includes transformations like DCE and LICM. Utils include canonicalization optimizations like Mem2Reg and LCSSA. The coverage was collected with SanitizerCoverage [25] by running each unit test through the default `-O3` optimization pipeline with x86 as the target architecture

2.3 LLVM unit test suite

The LLVM unit tests consist of a large collection of LLVM IR files that each test a singular or a set of compiler features for correctness. For the purposes of targeting compiler optimizations, there are around 8000 test files labeled as "Transformation" tests. These tests contain over 60000 LLVM IR functions. To give a sense of the thoroughness of the unit test suite, table 2.1 summarizes the line coverage achieved by running every permissible test case through `opt` with the `-O3` optimization level[2] and with `x86_64` as the target architecture. Note that many optimizations behave differently given different target architectures, which translates to missed coverage in our single `opt` run. Further, many transformations are not enabled in any default optimization level. Namely, many

[1] i.e. integers, floats, pointers, structs, vectors, etc.
[2] `-O3` is the most aggressive default optimization pipeline

interprocedural optimizations and scalar optimizations are not enabled on the default pipeline[3]. Despite this, the unit tests still obtain 58.11% line coverage. Most impressively, the unit tests are able to achieve near 90% on LLVM's InstCombine source code.

Nevertheless, as we will demonstrate in §3.1, LLVM's optimizations continue to be buggy. Further, §3.1 provides evidence that, despite being the most widely covered set of optimizations, InstCombine has historically been the buggiest optimization component. Due to this discrepancy, the FLUX fuzzer instead targets *path coverage*. As we alluded to in the introduction (1), we believe that considering more than one execution path through the same optimization code will lead to more robust bug-finding.

[3]To give an intuition on the number transformation not included in the default pipeline, in IPO alone the list of excluded transformations is as follows: `cross-dso-cfi`, `extract-blocks`, `function-import`, `globalsplit`, `hotcoldsplit`, `internalize`, `iroutliner`, `mergefunc`, `module-inline`, `partial-inliner`, `loop-extract`, `lowertypetests`, `pseudo-probe`, `sample-profile`, `scc-oz-module-inliner`, `strip`, `strip-dead-debug-info`, `strip-dead-prototypes`, `synthetic-counts-propagation`, `wholeprogramdevirt`

Chapter 3

Related Work

This chapter summarizes related work. In §3.1, we build upon the previous chapter by describing some existing studies on LLVM bugs and link this to the unit test statistics in §2.3. We then discuss related methods of finding these compiler bugs. Namely, we go into detail on previous compiler fuzzers in §3.2 and finally introduce previous translation validation approaches to bug finding in §3.3. We relate these existing bug-finding methods to FLUX and describe how the success and shortcomings of previous work have influenced FLUX's approach to test generation.

3.1 Bugs in LLVM

To further motivate our focus on optimizations bugs, we relate a number of compiler bug studies. Previous work has found that, across all compiler components, optimizations rank among the top ten buggiest for LLVM and GCC[1] [30]. These statistics are not just historical artifacts. Recently, researchers have had success in finding optimization bugs, not by generating, but by simply selecting appropriate optimization orders and optimization settings to pass test input to [10, 4]. These approaches work by using an existing fuzzer, most often *Csmith* [32] to generate inputs that are used to differentially test the outputs of different optimization configurations.

With optimizations bugs being so prevalent, it raises the question of unit test efficacy. We established that the LLVM unit tests achieve high coverage of the optimization source code in §2.3. How many bugs could truly remain if unit testing was rigorous enough to reach 100% coverage? Code coverage is an old adequacy metric [35], and continues to be the de facto driver in popular fuzzers [5]. Despite this, previous research has shown evidence that code coverage is very weakly correlated with real bugs, if at all [2]. This reality is also reflected in LLVM. As shown in table 2.1, the unit tests very easily achieved high coverage in all variants of instruction combine optimizations. Nevertheless, LLVM's suite of instruction combine transformations continues to be error-prone. A recent study of optimization bugs from Zhou et al. finds instruction combine to be the buggiest optimization in LLVM [34].

Clearly, having tests that execute every line of a compiler's source code does not entail bug-free code. Dijkstra famously stated: "Program testing can be used to show the presence of bugs, but never to show their absence" [7]. Despite Dijkstra's pessimism, we can still incrementally improve.

[1]The paper defines LLVM and GCC as having 96 and 52 components respectively

The shortcomings of unit test coverage inform us that a single pass over each section of the source code is insufficient. However, this means that bugs can be found when we execute the same code with different combinations of execution orders and with different values. It is this insight that motivates FLUX's approach to test generation. We are interested in creating new programs that are able to stress more complex orders of compiler source code. We refer to this metric as path coverage. To find new paths we utilize LLVM's high coverage unit test suite as building blocks to feed into our test generator.

3.2 Compiler Fuzzers

In general, fuzzing has been a successful approach to finding compiler bugs. A paper from Marcozzi et al. finds that fuzzer-found bugs have at least as much impact on the resultant binary as user-reported bugs [20]. We discuss two classes of compiler fuzzers and compare their approaches with FLUX's: generation-based fuzzers which create test programs from scratch (§3.2.1) and mutation-based fuzzers which mutate existing test cases to form new test programs (§3.2.2). We mention here that all fuzzers described in this section generate source languages (i.e. C, C++, JavaScript) and every fuzzer except LangFuzz [9] chooses LLVM/clang as one of its fuzzing targets. As we will describe in the following subsections, existing work tends to spend considerable effort in respecting source program semantics. This focus on source program generation limits a fuzzer's ability to target middle-end optimizations. The reasons for this are twofold. One, generating tests for a specific source language will not allow complete exploration of LLVM IR features; LLVM IR is a general representation that a wide variety of source languages can compile to. And two, as we will demonstrate below, since source code is at the mercy of the compiler frontend's lowering process, source-based fuzzers must rely on heuristics to target specific optimizations. The IR that is actually sent to LLVM's optimizer, may not resemble the source code, making it harder to predict if certain optimizations will be triggered. In contrast, FLUX generates test cases at the LLVM IR level, allowing it to bypass the compiler frontend and interface directly with optimizations.

3.2.1 Generative Compiler Fuzzers

Likely the most widely used compiler fuzzer, *Csmith* [32] finds success by generating C programs via random sampling from a pre-defined grammar. The grammar, a subset of C, ensures that generated programs are able to pass through the compiler's frontend. Each *Csmith* program prints a checksum of the program's randomly generated non-pointer global variables. This checksum allows *Csmith* to be used for differential testing [21]. In contrast, FLUX does not use a checksum, but instead relies on Alive2 [17], a translation validation tool, which is described in §3.3.

Csmith inspired numerous follow-up works which either build upon [5] or utilize it [10, 4]. One testing approach called swarm testing [8] utilizes *Csmith* by omitting grammar features during test generation in order to "swarm" a smaller subset of the compiler. This idea of limiting the search space to perform more thorough fuzzing is highly compatible with FLUX. We discuss this more in chapter 6.

Despite its success, *Csmith* has exhausted the vast majority of its output space in recent years. As a result, *YARPGen* a C/C++ grammar-based generator was released [14]. *YARPGen* notes that existing compiler fuzzers suffer from distributional bias and will eventually lose effectiveness.

To solve this, *YARPGen* develops *generation policies* that skew the random sampling distribution towards certain distributions. For instance, one policy might favour arithmetic expressions to target sub-expression elimination optimizations. In a similar vein, the work of Nagai et al. focuses solely on generating programs that target arithmetic optimizations [23, 24]. Unfortunately, the *YARPGen* generation policies and the work of Nagai et al. rely on heuristics in order to target compiler optimizations. To illustrate, the writers of *YARPGen* state:

> "These policies are drawn from our knowledge of how optimizing compilers work, and where bugs in them are likely to be found; it is not clear to us that this kind of insight can be automated in any meaningful fashion."

In contrast, the FLUX fuzzer combines code that is known to trigger optimizations already. The trade-off is that we sacrifice the level of randomness provided by generative methods in favour of mutating code that has a high likelihood of hitting optimizations, without the use of any sweeping heuristics.

Finally, we briefly mention compiler fuzzers that leverage neural networks to perform program generation. Both Liu et al. and Cummins et al. utilize recurrent neural networks in order to automatically generate C and OpenCL programs, respectively [13, 6]. Although interesting, we consider ML approaches orthogonal to our work since it is difficult to achieve mechanistic understanding for neural network applications, and hence difficult to compare to principled approaches.

3.2.2 Mutation-based Compiler Fuzzers

Besides generative fuzzers, other approaches have found success in mutating existing test cases in order to generate new compiler inputs. One class of mutational fuzzers relies on *semantics preserving mutation* in order to generate many different but behaviourally equivalent programs. Because of their semantic equivalence, these programs enable differential testing on a single compiler, whereas *Csmith* relied on differential testing between different compilers or different compiler optimization schemes. *Orion* was the first fuzzer to introduce this idea [11]. It generates semantically equivalent programs by randomly pruning non-executed parts of a seed program. The same authors then released *Athena,* which additionally inserts code into unexecuted regions [12]. Further, *Athena* guided their mutation to create programs with a large distance from the seed program. This distance is based on the properties of the program's CFG. Beyond mutating dead code, *Hermes,* which was again created by the same authors[2], introduced semantics preserving mutation in live code. They accomplish this by inserting random code that is side-effect free into paths that the program actually executes. In contrast, FLUX does not attempt semantic equivalence in its generated programs. Instead, it utilizes translation validation to detect miscompilations.

As for mutations that do not preserve program semantics, one interesting approach is *skeletal program enumeration (SPE)* [33]. A program skeleton is defined as a program that has each of its variable definitions and uses represented as "holes". *SPE* then enumerates every possible assignment of program variables into these holes. The authors then propose a way to minimize the set of enumerated programs to avoid generating functions that are equivalent. Another mutation-based fuzzer, *LangFuzz* [9], mutates a seed program by replacing non-terminals with random grammar walks and terminal *code fragments.* These code fragments are extracted from programs known to have caused

[2]In case you could not tell from the naming scheme

invalid behaviour in the past. The key insight that this fuzzer evaluates is: can a recombination of previously problematic input lead to a higher chance of causing new problems? Although *LangFuzz* targets the JavaScript interpreter, we consider its mutation strategy to be conceptually the most similar to FLUX. Instead of inserting code fragments into a seed program, we insert entire unit tests into other unit tests, and instead of replacing non-terminals, our inline mutation replaces terminal nodes. More details on this are in chapter 4.

3.3 Translation Validation

Orthogonal to compiler fuzzing, translation validation approaches compiler bug-finding from a formal verification angle. We discuss two relevant LLVM-IR based approaches to formal verification. In order to verify peephole optimization in LLVM, Lopes et al. developed *Alive* [18], a domain-specific language (DSL) based on LLVM-IR that automatically proves whether a transformation is correct. Alive takes two inputs, the pre and post-transformed code that is written in Alive's DSL. Then the correctness criteria and "definedness" for the transformation are automatically encoded into SMT queries. Alive then transfers these queries to an SMT solver like Z3 [22]. Hence, the correctness of the transformation is automatically proven.

To extend the applicability of Alive's design, Lopes et al. introduced Alive2 [17]. Rather than operate on a custom DSL, Alive2 directly operates on LLVM IR. As with Alive, Alive2 verifies a single optimization pass by mapping the correctness of a transformation to an equivalent set of SMT queries. Due to its compatibility with LLVM IR, Alive2 is able to directly verify optimizations in LLVM's `opt` pipeline. Importantly, Alive2 is also able to handle code that contains undefined behaviour. For example, code that contains uninitialized variables, division by zero, or any sort of overflow is considered undefined behaviour. Many of the fuzzers discussed in §3.2 that rely on differential testing (i.e. *Csmith, YARPGen, Orion, Athena, Hermes*) are careful to avoid introducing undefined behaviour into their generated tests. The reason for this is compiler optimizations have free rein to make arbitrary transformations on the code if it contains undefined behaviour. This means that we cannot expect the output of a program containing undefined behaviour to be consistent, which is a necessary assumption when performing differential tests.

Because FLUX uses Alive2 to detect compiler miscompilations, we do not emphasize undefined behaviour prevention in our design of FLUX's mutations. Unfortunately, this approach is not without its own tradeoffs. Due to the memory limits of Z3 and the complexity of modeling constraints, Alive2 cannot support arbitrarily large test cases. We discuss the effect of this program size bound on our implementation in chapter 5. Further, Alive2 can only verify intraprocedural optimizations, meaning that Alive2 cannot verify any IPO transformations. We accept this trade-off in favour of a less restricted output space of test programs.

Chapter 4

Design

In this chapter, we introduce the specifics of FLUX's design. We begin by describing the high-level overview of our fuzzer in §4.1. This section sets up our design goals and discusses the challenges in designing mutations in LLVM IR. We then describe in detail, FLUX's two crossovers: the inlining crossover and the sequencing crossover in §4.2.

4.1 High-Level Approach

To contextualize our approach to designing FLUX, we re-iterate the limitations of existing approaches to compiler fuzzing, which we discussed in §3.2:

- **L1:** The output space of existing compiler fuzzers is typically a skewed subset of the target compiler's input space. This arises from the difficulty in incorporating all grammar elements into an automatic test generator.
- **L2:** The focus on generating well-formed test cases to pass the frontend checks of the compiler makes it difficult to fully stress the compiler middle and backend.
- **L3:** Compiler fuzzers that do target optimizations, rely on heuristics and intuition to trigger optimization code.

Due to these existing fuzzer limitations, we believe there is a gap in the current compiler bug-finding space. The gap is this: optimizations in the middle end of the compiler (e.g. LLVM's `opt`) are currently under-explored. We further collected evidence from previous bug studies in §3.1 that shows that compiler optimizations were, and if previous data is any indicator, likely still are one of the buggiest components of the compiler.

We further make the observation in §2.3 that LLVM's unit tests are able to make at least a single pass on a large majority of optimization features in LLVM. Unfortunately, as we established previously, line coverage does not entail bug-free code, which motivates our path coverage approach. This sets up the problem statements that FLUX attempts to solve:

> *Can we leverage these high line coverage unit tests to generate new tests that are able to explore new paths through the compiler's optimizations? Are these new tests able to explore bug-triggering paths in LLVM's optimization code?*

Answering these problems would amend the gaps left by previous compiler fuzzers. Namely, since the LLVM unit tests are fairly complete, new tests that are able to replicate and extend the unit test's coverage will be able to stress a larger range of compiler optimizations than existing fuzzers (**L1**). Further, by operating on LLVM IR, we conveniently avoid the compiler's frontend and the strict requirements that go hand-in-hand with it (**L2**). Finally, since we observe a direct mapping from unit test to LLVM optimization, we need not rely on randomness and heuristics to target certain optimizations (**L3**). Instead, we have a different problem. Our fuzzer attempts to stress the compiler with different combinations and orderings of the coverage that the unit tests reach.

4.1.1 Design Goals

In order to generate new test cases that inherit the optimization hitting properties of LLVM's unit tests, we consider a way to combine two tests together. We call this combination of two unit tests a *crossover* mutation. To design a successful crossover mutation, we note two things.

Coverage preservation. The first goal is that the crossover should try to preserve the coverage of its component tests. This means we should avoid any mutation that significantly alters coverage on the target optimization code. Crossover mutations in general purpose fuzzers [1] traditionally splice two test inputs together. The resultant test retains the first half of one test, while the second half of the resultant test belongs to the other. Besides the code validity concerns that would accompany this kind of mutation, it also has a high likelihood of forfeiting the coverage of the individual inputs.

Listing 4.1: Flip and mask pre-optimization

```
define i32 @flip_and_mask(i32 %x) {
  %shl = shl i32 %x, 31
  %shr = ashr i32 %shl, 31
  %inc = add i32 %shr, 1
  ret i32 %inc
}
```

Listing 4.2: Flip and mask post-optimization

```
define i32 @flip_and_mask(i32 %x) {
  %1 = and i32 %x, 1
  %inc = xor i32 %1, 1
  ret i32 %inc

}
```

To illustrate this point, consider listings 4.1 and 4.2. Listing 4.1 is from LLVM's directory of InstCombine unit tests, and listing 4.2 is the expected result after passing the test through `opt` with the `-passes="instcombine"` flag. Notice that the `shl` instruction shifts the input variable left by 31 bits, then `ashr` (arithmetic shift right) shifts the result back to the right by 31 bits. This leaves one potentially non-zero bit in the least significant position of the result. The code in listing 4.2 optimizes this by replacing the shifts with a simple `and` mask. Further, since `ashr` fills the most significant bits of the result with the signed bit of `%shl`, the variable `%shr` is either 0 or -1. This means the addition and consequently, the return value of the function is either 0 or 1. Listing 4.2 replaces the `add` with a faster `xor`.

Clearly, the prerequisite code structure required to trigger these peephole optimizations is very specific. A crossover mutation that arbitrarily drops unit test code will struggle to explore new paths through the same optimizations. In the first place, the advantage of using unit tests is that they are already tuned to specific optimizations. A crossover that destroys the code structure that triggers an optimization will devolve into a random mutator, leading us to rely more on luck to target new path coverage.

New path exploration. Beyond preservation, the second design goal is to explore new paths through the coverage already achieved by the unit tests. A crossover mutation that is overly conservative in preserving the coverage of its inputs may not explore any interesting paths through the compiler. For instance, consider a crossover mutation that takes as input two LLVM IR functions that target specific intraprocedural optimizations. The mutation then simply returns a module that contains the two functions[1]. Undoubtedly, the input coverages are largely preserved since we have not modified any intraprocedural structure. However, the extra exploration that this provides is minimal, if it provides any at all. LLVM's `opt` separates transformations passes by the scope they operate on. For instance, ModulePasses operate on modules, FunctionPasses operate on functions, etc. Returning to our simple crossover example, although we may see some extra code executed in a module pass (e.g. IPO), we will likely not observe any new exploration in intraprocedural code as `opt` will invoke two separate function pass executions on the two disconnected functions.

4.1.2 Well-formedness Constraints

Before describing our crossover mutations, we briefly mention the constraints and challenges imposed by the LLVM IR. Although operating on LLVM IR circumvents the generation of a source language like C, malformed IR will still be quickly filtered by LLVM's IR parser and verifier. Hence, any crossover mutation should be careful to adhere to the LLVM IR's "well-formedness" constraints [16].

As stated in §2.2, LLVM IR is a strongly typed language. This means that each IR instruction as well as each of its operands is explicitly assigned a type. Each instruction has rules on the types of values it can operate on. For instance, the binary `add` operation only takes integers or vectors. Any program mutation that aims to generate or modify instructions must adhere to these type rules.

Well-formed LLVM IR must also have consistent dominance relations. That is, each variable definition must dominate all of its uses. This means that each value can only reference values that are either defined in the same basic block, but occur before the reference, or are defined in predecessor blocks that dominate the reference's basic block. Mutations that do not respect dominance relations will quickly invalidate the entire input module.

Other considerations include LLVM's SSA form (§2.2), which disallows variable re-definition, and the many other esoteric constraints that we discovered during our test runs of FLUX. For instance, PHI nodes must be grouped together at the beginning of each basic block, the `landingpad` instruction must be the first non-PHI instruction in its basic block, etc.

4.2 Crossover Mutations

Our goal is to design crossover mutations that attempt to satisfy both the *coverage preservation* of the LLVM unit tests, and introduce *new path exploration* through the existing coverage. As demonstrated by the examples presented in §4.1.1, this requires striking a balance between preservation and exploration. Approaches that forgo the original unit test structure are likely to explore different paths, however, these different paths won't be as well targeted to LLVM's optimizations. Conversely, overly conservative mutations won't be able to explore new paths at all. To find a middle ground, we consider intraprocedural and interprocedural optimizations separately.

[1]That is, places the two functions in the same file, without any interaction between functions

4.2.1 Inlining Mutation

To target intraprocedural optimizations, we introduce the *inlining* mutation. This mutation combines two LLVM IR functions and generates a new function that combines the code of its inputs. This crossover is congruent to the inline expansion optimization, which replaces a function call site with the body of the called function. Our inlining mutation combines two test cases by selecting a function from each test and inlining the entire body of one function into the other. The difference between our crossover and the inline expansion optimization is that we randomly select an insertion location in one input function to insert a call site to the other input function, before completely inlining the call site.

We denote the LLVM IR function that acts as the destination of the inlined code, the *destination function*. We denote the function whose body we are inserting (i.e. the called function) as the *source function*.

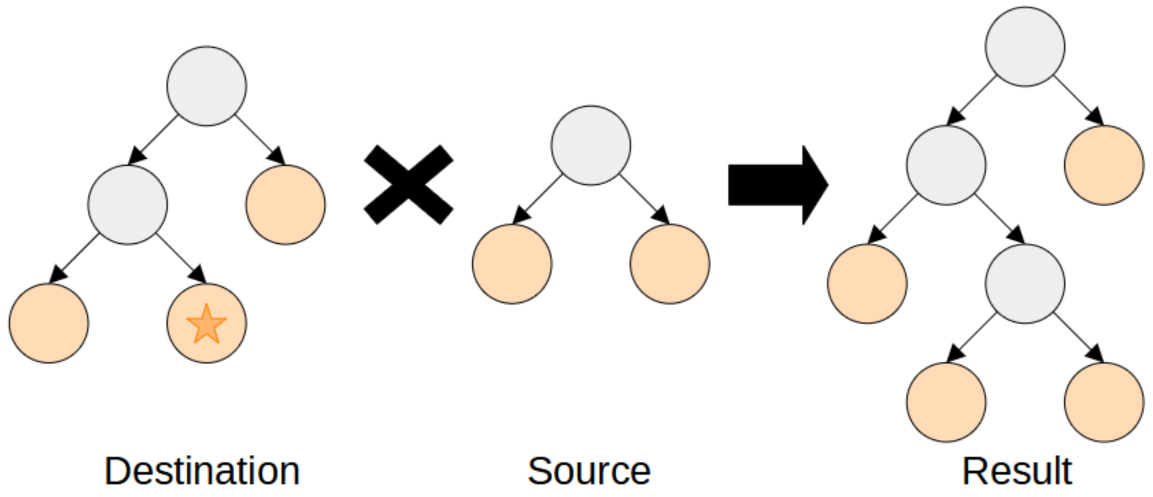


Figure 4.1: Example of the inlining crossover on two IR functions which are represented by abstract syntax trees. Light grey nodes are non-terminal and orange nodes are terminal. The star denotes the insertion location for the source function.

To induce new exploration in the compiler, we link the dataflow of the destination function with the source function's function body. We accomplish this by passing variables of the destination function into the source function's arguments. The return value of the source function is then transplanted into an operand of one of the destination function's instructions. We denote this destination operand as the *insertion location*. This form of dataflow stitching is desirable, as it reduces the chance that dead code elimination optimizations immediately delete our inserted code.

Thus, after the crossover concludes, the crossover result has the source function's entire dataflow embedded in the body of the destination. From an abstract syntax tree (AST) interpretation of the IR, this crossover is akin to the replacement of a non-terminal in the destination function, with the entire source function AST. Figure 4.1 demonstrates what the inlining mutation output will look like. Note that this notion of AST expansion is similar to the code fragment mutation method used by *LangFuzz* [9] which we described in §3.2.2. The difference with the inlining mutation is that we are replacing terminal nodes with entire AST trees, while *LangFuzz* fills non-terminal nodes with terminal code fragments extracted from buggy code.

We consider the inline mutation a good compromise between coverage preservation and new code exploration. On an intraprocedural level, this crossover maintains the original code of both the source and destination functions by inlining an entire function body. Further, this crossover introduces new dataflow dependencies at the boundaries between the source function's entry and exit block, as well as a new control flow structure.

Inlining Algorithm

Now that we have described the inline mutation at a high level, we provide a more detailed look into the inlining algorithm.

Algorithm 1: Function inlining mutation

```
1  Function InlineMutation($F$, $F'$):
       Input: Destination Function: $F$, Source Function: $F'$
       Output: Crossover Result: $F''$
2      $F'' \leftarrow NULL$;
3      $F' \leftarrow$ resolveGlobalSymbols($F, F'$);
4      $L \leftarrow$ getAllInsertionLocations($F$);
5      $L \leftarrow$ randomShuffle($L$);
6      /* Try all insertion locations in the destination function */;
7      for $i \in L$ do
8          $ArgCands \leftarrow$ getAllArgumentCandidates($F, i$);
9          if $F'$.ArgTypes $\subseteq$ $ArgCands$.ArgTypes then
10             /* Select a random argument of matching type for each source arg */;
11             $SrcArgs \leftarrow \{\}$;
12             for $a \in F'$.Args do
13                 $matchingTypeArg \leftarrow$selectRandArg($ArgCands, a$.type);
14                 $SrcArgs$.add($matchingTypeArg$);
15             $CI \leftarrow F$.insertCallInstBefore($F'$, $SrcArgs$, $i$);
16             $i$.setOperand($CI$);
17             $CI$.inlineFunction();
18             $F'' \leftarrow F$;
19             break;
20     return $F''$
```

Algorithm 1 describes the main inlining process. The algorithm pays special attention to satisfying the well-formedness constraints described in §4.1.2. To prepare the two input LLVM IR functions for inlining, the algorithm first resolves any conflicting global symbols on line 2. This involves renaming any duplicate global function, variable, and alias names. Line 3 collects all potential insertion locations in the destination function and stores them in the set L. Collecting these insertion locations simply involves traversing all instructions in the destination function and iterating over each operand of each instruction. Every operand that matches the return type of the source function is a viable insertion location. Next, we randomly iterate over each potential insertion location in L to determine its feasibility. As stated previously, we want to pass existing variables in the destination function as arguments into the source function. Hence, we collect all potential argument candidates on line 7. Note that we are inserting our call instruction, and consequently our inlined function, right before the insertion location. Therefore, to abide by dominance constraints, we can only use variables that are defined in blocks that dominate the insertion location as arguments. We achieve this in the `getAllArgumentCandidates` function by a reverse iteration through all predecessor basic blocks in the dominator tree, storing every variable in each of these predecessor blocks as a candidate. On line 8, we check if we have enough matching values of the same type as the source function arguments.

Now that the algorithm has established that the insertion location is valid, on line 11, it randomly selects a destination function variable for each of its arguments. The algorithm then inserts a call instruction to the source function, stores the return value of the call into the insertion operand, and finally calls another function to inline the entire source function. If the algorithm exhausts every insertion location without finding enough candidate arguments, then the mutation fails.

In practice, to minimize the number of failures, we run algorithm 1 on every combination of function pairs that are contained in each unit test file. As we will show in §6.2, this has a synergistic effect with the sequencing mutation (§4.2.2).

Inlining Examples

To give a concrete example of the inlining algorithm, consider listings 4.3 and 4.4. We take 4.3 as the destination function and 4.4 as the source function. Note that these two LLVM IR functions are taken directly from the unit test suite.

Listing 4.3: `LoopSimplifyCFG/scev.ll`

```
define void @t_run_test() {
entry:
  br label %loop.ph
loop.ph:
  br label %loop.header
loop.header:
  %iv = phi i32 [0, %loop.ph], [%inc, %loop.body]
  br label %loop.body1
loop.body1:
  br label %loop.body
loop.body:
  %inc = add i32 %iv, 1
  %cmp = icmp ult i32 %inc, 10
  br i1 %cmp, label %loop.header, label %exit
exit:
  br label %loop.body2
loop.body2:
  %iv2 = phi i32 [0, %exit], [%inc2, %loop.body2]
  %inc2 = add i32 %iv2, 1
  %cmp2 = icmp ult i32 %inc2, 10
  br i1 %cmp2, label %loop.body2, label %exit2
exit2:
  ret void
}
```

Listing 4.4: `GVN/edge.ll`

```
define i32 @f2(i32 %x) {
bb0:
  %cmp = icmp ne i32 %x, 0
  br i1 %cmp, label %bb1, label %bb2
bb1:
  br label %bb2
bb2:
  %cond = phi i32 [ %x, %bb0 ], [ 0, %bb1 ]
  %foo = add i32 %cond, %x
  ret i32 %foo
}
```

Following algorithm 1, we first randomly select an insertion location. For instance, let us choose the second operand to the `icmp` instruction on line 13, the constant integer `10`. Next, we iterate backward from line 13 to collect all argument candidates. From this iteration, we find that variables `%inc` and `%iv` on lines 12 and 7 respectively are valid candidates since they match the type of the source function `@f2`. Let's randomly choose `%inc` our argument to pass to `@f2`. Finally, we insert a call instruction to `@f2` with `%inc` as the argument and replace the second operand of line 13's `icmp` instruction with the return value. This function call is then inlined. The result of this process is shown in listing 4.5. The source function code that is inlined into the destination function is highlighted in red.

Listing 4.5: Example of an inlining mutation result from combining destination function `GVN/edge.ll` and source function `LoopSimplifyCFG/scev.ll`

```
 1 define void @t_run_test() {
 2 entry:
 3   br label %loop.ph
 4 loop.ph:                                          ; preds = %entry
 5   br label %loop.header
 6 loop.header:                                      ; preds = %f2.exit, %loop.ph
 7   %iv = phi i32 [ 0, %loop.ph ], [ %inc, %f2.exit ]
 8   br label %loop.body1
 9 loop.body1:                                       ; preds = %loop.header
10   br label %loop.body
11 loop.body:                                        ; preds = %loop.body1
12   %inc = add i32 %iv, 1
13   %cmp.i = icmp ne i32 %inc, 0
14   br i1 %cmp.i, label %bb1.i, label %f2.exit
15 bb1.i:                                            ; preds = %loop.body
16   br label %f2.exit
17 f2.exit:                                          ; preds = %loop.body, %bb1.i
18   %cond.i = phi i32 [ %inc, %loop.body ], [ 0, %bb1.i ]
19   %foo.i = add i32 %cond.i, %inc
20   %cmp = icmp ult i32 %inc, %foo.i
21   br i1 %cmp, label %loop.header, label %exit
22 exit:                                             ; preds = %f2.exit
23   br label %loop.body2
24 loop.body2:                                       ; preds = %loop.body2, %exit
25   %iv2 = phi i32 [ 0, %exit ], [ %inc2, %loop.body2 ]
26   %inc2 = add i32 %iv2, 1
27   %cmp2 = icmp ult i32 %inc2, 10
28   br i1 %cmp2, label %loop.body2, label %exit2
29
30 exit2:                                            ; preds = %loop.body2
31   ret void
```

Note that the crossover result has split the basic block `loop.body` and inserted the entire source function body, including its CFG structure, into this basic block. Further, the entry block of the source function is merged with the beginning of the `loop.body` block. The exit block of the source function is renamed and merged with the end of `loop.body` block. Further, note that the instances of the source function argument `%x` are replaced with variable `%inc`. We also mention that duplicate variable names like the two instances of the `%cmp` name are resolved during the inlining.

4.2.2 Sequencing Mutation

Since we have covered a crossover mutation that targets intraprocedural optimizations, next we introduce a crossover that explores interprocedural optimization code. Interprocedural optimizations generally analyze entire programs. In the context of our unit tests, this means a collection of functions that exist within the same module and test file. Examples of interprocedural optimization include global dead code elimination, program partitioning to improve instruction locality, and function inlining. Hence, we aim to create a crossover mutation that increases the complexity of a module by introducing dependencies between functions.

To inject dependencies between functions while preserving the original test coverage, we propose a crossover mutation that generates a new function that calls the component units tests in sequence. We denote this crossover as a *sequencing mutation*. As opposed to the inlining mutation which takes two functions as input, and returns a function, our sequencing mutation instead returns an LLVM IR module. This module, at minimum, will contain the two input functions, and a third function that serves as the *caller* for the other two functions.

Of course, simply calling the component functions in sequence will be trivially optimized away by dead code elimination passes. To prevent this, we inject interprocedural dataflow into the caller function. One example of dataflow injection is passing the return value of one function into the arguments of the other. However, due to type constraints, this type of dataflow injection is not possible between functions with conflicting types. In order to support a larger set of crossover mutants, we try a number of dataflow generating strategies:

- *Return Value Chaining Strategy.* As we just described, this strategy involves passing the return value of the first function call into one of the arguments of the second function call. The caller function then returns the return value of the second function call. The prerequisite for this mutation is that the return value type of one function matches the type of one of the arguments of the other function.

- *Pointer Argument Strategy.* Another option for introducing dataflow is passing the same pointer value to both functions. To check whether the input functions are eligible for this strategy, we check whether a pointer argument of the same type exists in the parameters of both functions.

- *Binary Operation Strategy.* Finally, we consider the case where both input functions have a return value. If both return values are the same type, we insert dataflow by sending both values into a randomly selected binary operation. We take the result of the binary operation as the return value of the caller function.

For each of the above sequencing strategies, we patch up any remaining unfilled arguments with randomly generated constants. In cases that do not satisfy the requirements for any of the above three crossovers, we default to a caller function that does not contain any dataflow between the two input functions.

Sequencing Algorithm

Having described the different dataflow injection strategies, we now give an overview of the entire sequencing mutation flow in the algorithm below.

Algorithm 2: Function sequencing mutation

```
1  Function SequencingMutation($F$, $F'$):
       Input: Function: $F$, Function: $F'$
       Output: Result Module: $M$
2      $M \leftarrow$ createEmptyModule();
3      $F' \leftarrow$ resolveGlobalSymbols($F, F'$);
4      $M \leftarrow M$. linkFunctionsInto($F, F'$);
5      $Strategies \leftarrow \{$ RetValStrat(), PtrArgStrat(), BinOpStrat() $\}$;
6      $Strategies \leftarrow$ randomShuffle($Strategies$);
7      /* Try all dataflow injection strategies in a random order*/;
8      for $s \in Strategies$ do
9          if $s.satisfiesRequirements(F, F')$ then
10             $s$.generateCallerFunc($M, F, F'$);
11             return $M$;
12     generateDefaultCallerFunc($M, F, F'$);
13     return $M$
```

Algorithm 2 begins by creating the empty module, M which serves as the return value for this crossover. Similar to the inlining mutation, we first rename the global variables in F' to prevent any symbol collisions. Then the mutation links both functions into the empty module. This involves copying over each function from its parent module, as well as any global values that each function depends upon. After this, the algorithm collects the sequencing strategies described earlier into an array, which we randomly iterate over on line 7. We randomize the order of the strategies to avoid unneeded bias in our dataflow injection scheme. If the two input functions satisfy the requirements that correspond with the current dataflow injection strategy, then we generate the strategy-specific caller function on line 10. Finally, as we stated before, if all strategies fail, we default to a caller function that does not contain a dataflow injection.

Sequencing Examples

To demonstrate how the sequencing mutation would look in LLVM IR, consider the listings 4.6, 4.7, and 4.8. Each listing omits the function bodies of the input functions since we only require the information in the function signature in order to apply the algorithm. Lines 1 and 2 of each of the listings hold the input function signatures.

Listing 4.6: Example return value chaining strategy for the sequencing mutation which uses functions from `GVN/edge.ll`

```
1  define double @fcmp_oeq_not_zero(double %x, double %y) {...}   ; Input function 1
2
3  define double @fcmp_une_not_zero(double %x, double %y) {...}   ; input function 2
4
5  define double @function_sequence_caller() {                    ; Mutation result
6  entryBB:
7    %0 = call double @fcmp_oeq_not_zero(double 0.000000e+00, double 4.940660e-324)
8    %1 = call double @fcmp_une_not_zero(double %0, double %0)
9    ret double %1
10 }
```

Listing 4.6 shows an example of the return value chaining strategy. Notice that the variables highlighted in red demonstrate that the arguments to `fcmp_une_not_zero` are the same as the value returned by `fcmp_oeq_not_zero`. Further, the arguments to `fcmp_oeq_not_zero` are randomly generated constants.

Listing 4.7: Example pointer argument strategy for the sequencing mutation which uses functions from `InstSimplify/ptr_diff.ll`

```
define i64 @ptrdiff2(ptr %ptr) {...}                          ; Input function 1

define i64 @ptrdiff1(ptr %ptr) {...}                          ; Input function 2

define i64 @function_sequence_caller() {                      ; Mutation result
entryBB:
  %A = alloca ptr, align 8
  %0 = call i64 @ptrdiff1(ptr %A)
  %1 = call i64 @ptrdiff2(ptr %A)
  ret i64 %1
}
```

Listing 4.7 illustrates the pointer argument strategy. Since both functions take a pointer argument, we insert an `alloca` instruction in order to generate a pointer to stack memory. As illustrated by the red test, we then pass this pointer into both functions.

Listing 4.8: Example binary operation strategy for the sequencing mutation which uses functions from `Float2Int/basic.ll`

```
define i16 @neg_remainder(i8 %a) {...}                        ; Input function 1

define i16 @simple_fneg(i8 %a) {...}                          ; Input function 2

define i16 @function_sequence_caller() {                      ; Mutation result
entryBB:
  %0 = call i16 @neg_remainder(i8 16)
  %1 = call i16 @simple_fneg(i8 0)
  %B = sub i16 %0, %1
  ret i16 %B
}
```

Finally, listing 4.8 presents an example of the binary operation strategy. Since both input functions in this listing return an `i16` integer, we can randomly generate a binary operation that takes both return values as operands. As the red text shows, the sequencing mutation selected a `sub` instruction to pass our input function return values into. The result of `sub` is then returned.

Chapter 5

Implementation

In this chapter, we describe FLUX's implementation. We begin by describing the overall fuzzer structure in §5.1. This section will focus on how the mutations described in chapter 4 fit into the wider fuzzer architecture. In §5.2, we dive into the implementation specifics of the FLUX prototype's crossovers. Included in this section are some examples of implementation-specific tweaks that we needed to apply to conform to LLVM IR's constraints. Finally, §5.3 describes the details of how we integrate Alive2 [17] in FLUX.

5.1 Fuzzer Structure

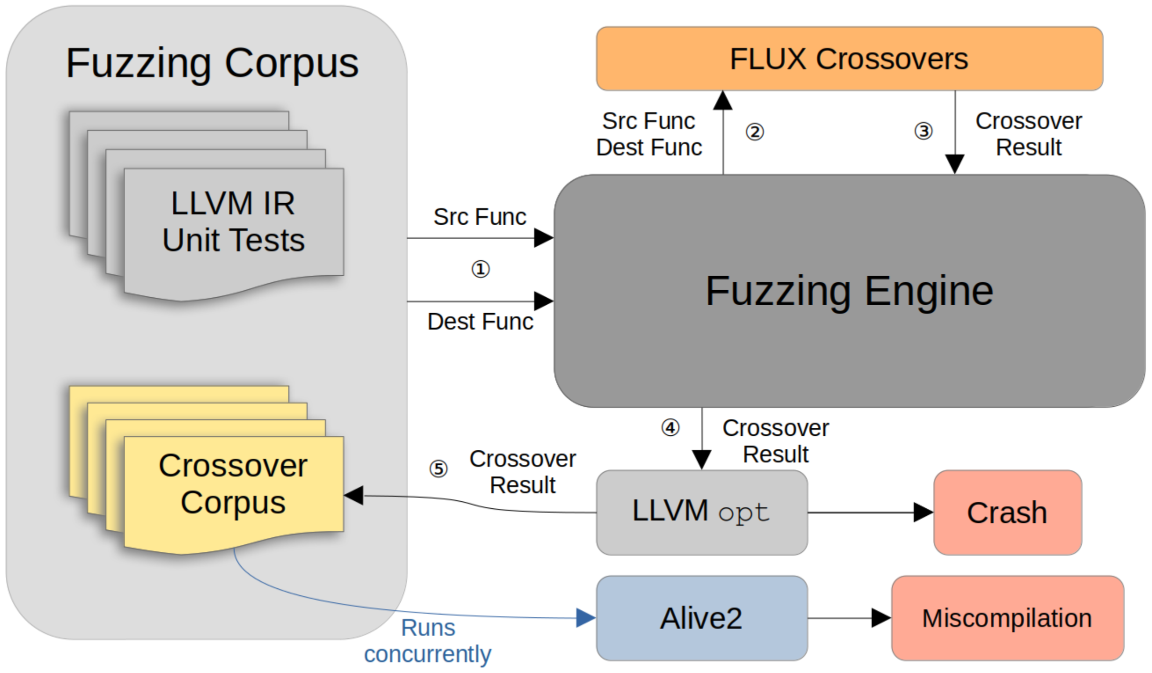


Figure 5.1: FLUX's high-level structure and work flow

Figure 5.1 gives a high-level overview of FLUX's structure. The fuzzer structure consists of the following components:

- *The seed corpus.* The seed corpus is the initial set of LLVM IR unit test files that FLUX uses to generate new tests.
- *The crossover corpus.* The crossover corpus stores the results of all crossover mutations.
- *FLUX crossovers.* FLUX's crossovers are implementations of the *inlining* and *sequencing* mutation described in §4.2.
- *Alive2.* The Alive2 component takes the crossover corpus as input and validates each input by running it through Alive2's `alive-tv` tool (§5.3)
- *The fuzzing engine.* The fuzzing engine is the framework responsible for selecting inputs, applying mutations on them to create new inputs, testing the new inputs for crashes, and finally storing the mutation results.
- *LLVM opt.* In order to test whether new tests generate crashes in LLVM's optimization passes, we pass each test through LLVM's optimizer, `opt`

The workflow of each fuzzer iteration is denoted by each of the numbered arrows in figure 5.1. The fuzzer initializes by reading in each LLVM IR unit test in the seed corpus. Then, fuzzer iterations commence by first randomly selecting two unit tests from the fuzzing corpus. These unit tests, which we denote as the "Src Func" and "Dest Func" in the figure, are passed into FLUX's crossover functions in step 2 and are returned to the fuzzing engine in step 3. Step 4 tests whether the crossover result triggers any crashes in the specified optimization passes, and step 5 stores this crossover result if no crash was found.

On the other hand, if a crash was found, then we perform a simple triaging of the crash's stack dump. Unfortunately, by design, any LLVM assertion failures are not meant to be gracefully handled. This means that we cannot prevent the compiler, and likewise the entire fuzzing loop, from crashing when FLUX is able to trigger an assertion crash. As we will describe in the next section (§5.2) since the fuzzing engine we use is linked with opt, we cannot gracefully handle all compiler crashes within the confines of the fuzzing engine. Hence, we use a script that re-executes the FLUX fuzzer when it triggers a crash. The script also hashes the crash's stack dump and compares it against previously found crashes. It then categorizes each found crash as new, or a duplicate of an existing crash.

5.2 FLUX's Crossovers

To implement the structure described in §5.1, we build our FLUX mutations on top of libFuzzer's fuzzing engine [28]. libFuzzer is a coverage-guided fuzzer that is linked with the program or library under test (i.e. `opt`). To use libFuzzer, a number of API functions must be implemented. For instance, the `LLVMFuzzerTestOneInput` function acts as an interface to the fuzz target, which libFuzzer can use to test a single input. Further, libFuzzer exposes API functions for custom mutations, namely `LLVMFuzzerCustomMutator` and `LLVMFuzzerCustomCrossOver`. We note that LLVM's tool set includes a basic optimization pass fuzzer that provides a simple LLVM IR-based mutator. This tool, called `llvm-opt-fuzzer`, implements `LLVMFuzzerTestOneInput` and `LLVMFuzzerCustomMutator`.

Specifically, the implementation for the `LLVMFuzzerTestOneInput` function constructs an optimization pipeline that calls the same code used by `opt`. The input IR file that is passed to `LLVMFuzzerTestOneInput` is first verified for correctness with LLVM's IR verifier, then fed through the pipeline. Optimized IR is returned from the pipeline and is again sent through the verifier to ensure that the optimization did not produce invalid code. Further, `llvm-opt-fuzzer` supplies an existing random IR mutator in `LLVMFuzzerCustomMutator` that performs random instruction deletions, insertions, and modifications.

FLUX leverages the existing infrastructure provided by `llvm-opt-fuzzer` and adds an implementation for libFuzzer's `LLVMFuzzerCustomCrossOver` API function. We implement FLUX's crossovers on LLVM's main active development branch, which at the time of writing this thesis, is on LLVM 16.0. We introduce a little less than 2000 lines of C++ code to LLVM to implement FLUX.

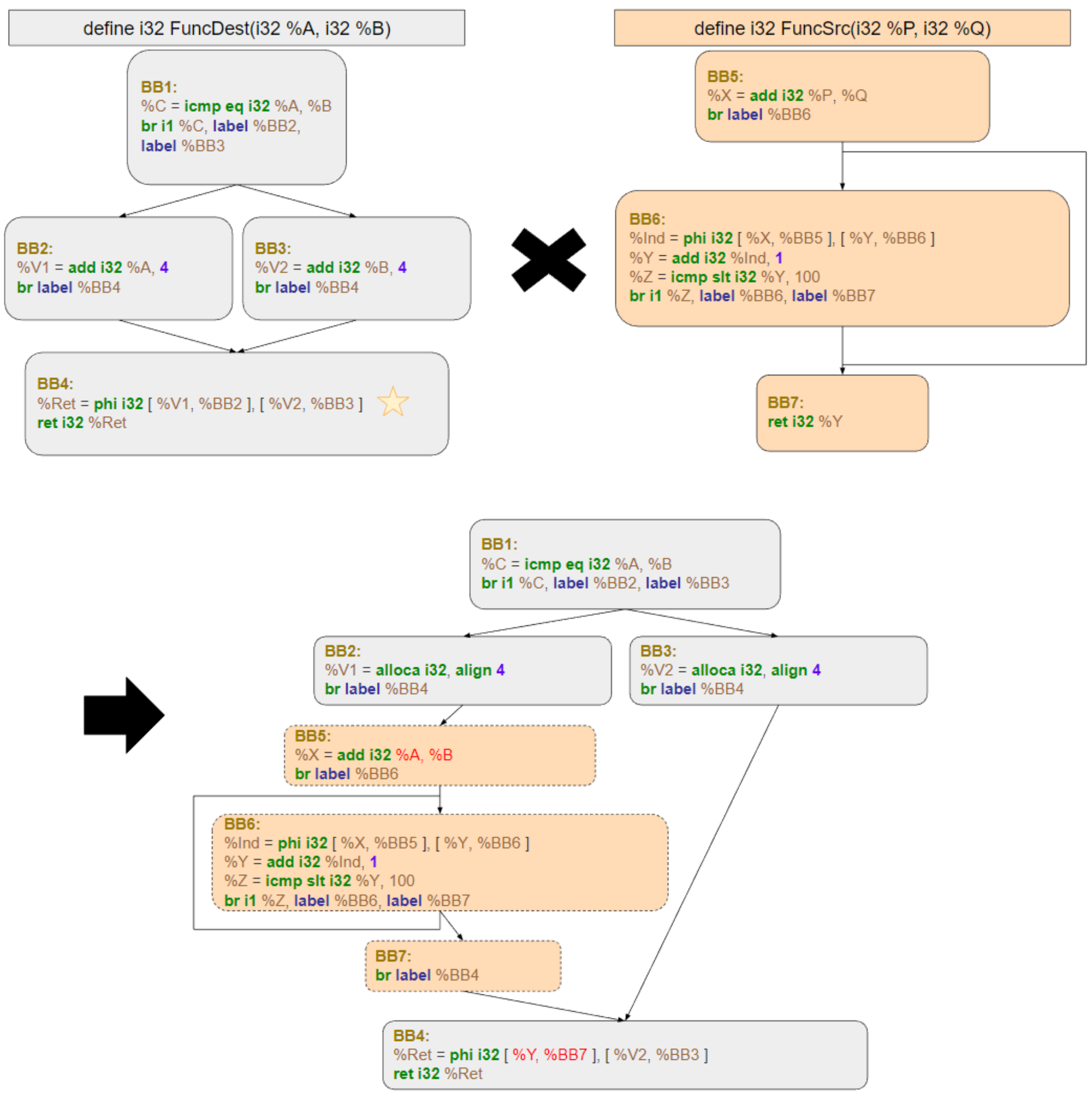
5.2.1 LLVM IR Constraints

In this section, we describe some challenges we faced in conforming to LLVM IR's well-formedness constraints §4.1.2, and the solutions we devised to satisfy these constraints. First, it should be noted that the general crossover algorithms that we introduced in §4.2, do not produce correct code for all possible IR inputs. Therefore, during our fuzzing experiments, we encountered numerous unexpected IR requirements, which we patched before conducting our evaluation. Among these requirements, we "fix" our code in one of two ways. **(1)** We completely disallow certain IR properties that are either very niche or are difficult to support. **(2)** We support certain IR constraints by integrating them into our crossover mutations.

(1) For the "completely disallow" case, we first describe modifications that we had to make for our inlining mutation (§4.2.1). There are instructions that we are not allowed to use as the *insertion location*. For example, we cannot choose a landing pad instruction as our insertion location, since the landing pad instruction must be the first non-PHI instruction in its basic block. This is simply a hard-coded rule in LLVM IR's language reference [16]. As for the sequencing mutation, we do not support functions that take certain argument types. This is because we cannot generate random constants for all types. Examples are the `x86_mmx` and `x86_amx` types. The language reference states that these types do not have constants associated with them.

(2) The only language constraint that we change our implementation to support is insertions into PHI nodes[1] for our inlining mutation. In contrast to the list of error-causing insertion locations that we disallowed in **(1)**, we consider the PHI node too prevalent to ignore. As described in 2.2, LLVM conforms to an SSA structure. The single static assignment form makes code analysis easier but makes it difficult to define value-flows that could originate from more than one predecessor basic block. The PHI instruction makes this simple by selecting the value that corresponds with the current basic block's predecessor block dynamically. Hence, PHI instructions are extremely common. Unfortunately, as stated in §4.1.2, one well-formedness requirement is that PHI nodes must be grouped at the beginning of each basic block. If algorithm 1 chooses a PHI instruction operand as its insertion location, then the crossover result will have the start of the exit block of the source function preceding the PHI instruction. Hence, the IR validator will immediately filter this crossover result.

[1] I used "node" interchangeably with "instruction", with respect to PHI



```
define i32 FuncDest(i32 %A, i32 %B)

BB1:
%C = icmp eq i32 %A, %B
br i1 %C, label %BB2,
label %BB3

BB2:
%V1 = add i32 %A, 4
br label %BB4

BB3:
%V2 = add i32 %B, 4
br label %BB4

BB4:
%Ret = phi i32 [ %V1, %BB2 ], [ %V2, %BB3 ]
ret i32 %Ret
```

```
define i32 FuncSrc(i32 %P, i32 %Q)

BB5:
%X = add i32 %P, %Q
br label %BB6

BB6:
%Ind = phi i32 [ %X, %BB5 ], [ %Y, %BB6 ]
%Y = add i32 %Ind, 1
%Z = icmp slt i32 %Y, 100
br i1 %Z, label %BB6, label %BB7

BB7:
ret i32 %Y
```

```
BB1:
%C = icmp eq i32 %A, %B
br i1 %C, label %BB2, label %BB3

BB2:
%V1 = alloca i32, align 4
br label %BB4

BB3:
%V2 = alloca i32, align 4
br label %BB4

BB5:
%X = add i32 %A, %B
br label %BB6

BB6:
%Ind = phi i32 [ %X, %BB5 ], [ %Y, %BB6 ]
%Y = add i32 %Ind, 1
%Z = icmp slt i32 %Y, 100
br i1 %Z, label %BB6, label %BB7

BB7:
br label %BB4

BB4:
%Ret = phi i32 [ %Y, %BB7 ], [ %V2, %BB3 ]
ret i32 %Ret
```

Figure 5.2: Example of inlining into a PHI instruction. The star denotes the insertion instruction. The red text in the crossover result highlights the variable replacements performed by the inlining.

To support PHI operand insertion locations, we add new functionality to `insertCallInstBefore` on line 14 of algorithm 1. When the insertion location is a PHI node, `insertCallInstBefore` will insert a new basic block immediately preceding the PHI node. This new basic block "splits" the edge that corresponds to the PHI operand insertion location. Conceptually, this edge split simply injects a single basic block between the previous CFG edge of the destination function. The call instruction and consequent inlined code will exist in this new basic block. To illustrate this edge splitting, we provide a CFG view of an inlining crossover in figure 5.2.

The example above inlines the source function, coloured in orange, into the PHI instruction, highlighted with the star. Specifically, the inline algorithm chooses `%V1`, the first operand of the PHI instruction. Thus, after performing the previously described steps, `insertCallInstBefore` splits the edge that connects basic block `BB2` to `BB4`. After inlining the source function, we see that the

CFG of the source function is completely separate from that of the destination function. Note that this crossover result satisfies well-formedness constraints.

Beyond the patches we described in this section, we acknowledge that our FLUX prototype may still produce incorrect IR. Given that the number of possible crossover outcomes for even a single LLVM unit test crossover is to the scale of 2.5 billion[2], we are certain that there exist errors that we overlooked. However, as we will demonstrate in our evaluation (6), our prototype is able to demonstrate increased path coverage and find new bugs.

5.3 Alive2 Details

Besides the main libFuzzer driven fuzzing loop, we also describe our implementation decisions for integrating Alive2. The Alive2 repository[3] provides a number of ways to integrate their translation validation techniques into LLVM. Among them are drop-in replacements for `clang` and `clang++`, plugins for `opt` which verify the before and after of each LLVM transformation pass, and a standalone validation tool, `alive-tv`. Due to our goal of directly stressing LLVM's optimizations, we decide against using Alive2's `clang` replacements. Further, Alive2, on average, verifies LLVM IR files significantly slower than FLUX's crossovers can generate tests. Due to this, we decide against the direct `opt` plugins, which would slow our fuzzing throughput down. This, in turn, would reduce the speed in discovering memory errors and LLVM crashes that are detectable with LLVM's verifier. Instead, we choose to run `alive-tv` in a separate process that operates on the same crossover corpus that is used by a running libFuzzer instance.

The `alive-tv` takes in a single LLVM IR file, passes the file through a number of LLVM optimizations, then verifies its correctness. As previously mentioned in 3.3, Alive2 does not support any verification of interprocedural optimizations. Hence, the transformations that `alive-tv` can apply to the input IR only includes intraprocedural passes. Specifically, `alive-tv` applies an optimization pipeline that is similar to `opt`'s `-O2` optimization level, but without interprocedural optimizations.

Another Alive2 constraint that we consider in our implementation are the memory and complexity limits of Alive2. Since Alive2 relies on an SMT solver, it encodes a number of restrictions on the size and complexity of its input programs. For instance, Alive2 will throw an out-of-memory error after consuming 500 MB of data. Due to the nature of the crossover mutations, newly generated test inputs can be very large. Hence, any test case that fails to meet Alive2's resource restrictions will cause wasted cycles. In order to amend this, we use a python script that orders each file in the crossover corpus by increasing file size. The script then feeds the smallest tests to `alive-tv` for verification.

A final implementation consideration concerning Alive2 are its unsupported instructions. Alive2 does not support every one of LLVM's vast set of language features. For instance, any IR that contains a pointer-to-integer cast will immediately be rejected by Alive2. In order to minimize the amount of unsupported behaviour in tests generated by FLUX, we choose to reduce the LLVM unit test suite to only contain functions that are supported by Alive2, when performing fuzzing runs that run concurrently with our Alive2 script. Note that FLUX is able to run without Alive2 with the entire unit test suite, and indeed we demonstrate in chapter 6, that we are still able to detect

[2]i.e. 50K to the power of 2
[3]https://github.com/AliveToolkit/alive2

compiler crashes. We make a further note that when we perform our fuzzing runs for the evaluation section, the set of usable `x86_64` unit tests functions decreases from around 48K to around 33K if we filter out every function that contains unsupported constructs. We are not concerned with this decrease in the input space, as even with 33,000 IR functions, even performing a single crossover on each combination of pairs results in around 550 million possible outputs.

Chapter 6

Evaluation

To recap, the main hypotheses that drove FLUX's design (§4.1.1) are:

- Despite the robust line coverage of LLVM unit tests, `opt`'s optimizations remain buggy. We hypothesize that exploring new path coverage through the same optimizations will lead to new bugs.
- We postulate that the design of FLUX's crossover mutations are able to induce these new paths, and likewise discover bugs.

To verify these claims, we guide our evaluation by answering the following questions:

Q1 Are FLUX's *inlining* and *sequencing* mutations able to find new path coverage through LLVM's optimizations? (§6.2)

Q2 Is FLUX able to find new crashes in `opt`? If so, what do these crashes look like? (§6.3)

Q3 Is FLUX's Alive2 integration able to find new miscompilation bugs in LLVM's optimizations? (§6.4)

6.1 Fuzzing Setup

We run our fuzzing campaigns on an Intel Core i7-12700K processor with 32 GB of memory, running Ubuntu 22.04. We target the active development branch of the LLVM repository, which corresponds to release 16.0. In order to collect path coverage information, we instrument `opt` with Sanitizer-Coverage [25]. To detect memory errors, we instrument `opt` with AddressSanitizer [27]. For each fuzzing run, we set the target architecture to `x86_64` to allow for cost estimation optimizations to run (e.g. function inlining considers architecture-specific costs).

6.2 Path Coverage Statistics

To answer evaluation question, **Q1**. We first conduct a path coverage experiment to verify whether our *inlining* and *sequencing* mutations are able to explore new paths through the optimization code.

First, we mention that true path coverage is very difficult to maintain efficiently. Due to the exponential nature of storing every permutation of paths[1], most fuzzers implement a proxy of this coverage metric. Namely, industry-standard fuzzers like AFL++ [31], simulate path coverage by maintaining coarse branch-taken hit counts. This is implemented with a byte map that records each unique `(branch_src, branch_dst)` pair in the instrumented code. Similarly, SanitizerCoverage, implements this with its `inline-8bit-counters` edge count instrumentation.

Specifically, libFuzzer maintains these byte mappings via a global array of modules (i.e. files). Each module map maintains an array of regions and each region is represented by a flexible byte array. When we instrument the compiler with SanitizerCoverage's `inline-8bit-counters` functionality, an increment function is inserted beside each unique edge in the optimizer. When libFuzzer initializes, it maps a byte in each of its region maps to every unique edge in the compiler source code. Thus after executing the instrumented optimizations, these inline counters are captured by the libFuzzer internals.

Hence, to estimate path coverage, we use SanitizerCoverage's edge counters to instrument `opt`[2], and we remove all other instrumentation (i.e. we disable coverage for block, function, arithmetic operation trackers, stores, loads, and stack depth).

6.2.1 Ablation Study

We conduct our path coverage measurement by targeting the `-O3` optimization pipeline with FLUX. For our first experiment, we evaluate each of FLUX's crossover components on the entire LLVM unit test suite. We prepare the unit test suite by collecting around 50,000 LLVM IR functions as seed input. We separately compare the results of fuzzing runs that use both the inlining and crossover mutations and runs that use one or the other. We run each mutation configuration five times for 70 minutes each run[3]. Table 6.1 shows our results after averaging over the five runs.

Mutations	**Path Cov. Features**	**New Coverage**
Inline + Sequence	1089776.8	**13691.8**
Sequence	1087988.6	11903.6
Inline	1076408.2	323.2
Baseline	1076085.0	

Table 6.1: Path coverage increase on `opt`'s `-O3` optimization pipeline after a 70-minute run on the entire LLVM unit test suite. Results are averaged over 5 runs

The above table lists the number of path coverage "features" that are achieved by the baseline seed corpus and the resultant corpus of each mutation scheme. To expand on the meaning of "feature", libFuzzer flattens each element of its coverage maps into a single integer when comparing test case coverages. This flattening process involves generating a unique feature ID for each of its byte map values. libFuzzer iterates over each byte element of each region of each module and assigns a unique integer value to each 8-bit bundle. As we described earlier, with our instrumentation, each of these bytes corresponds with a unique edge in the optimization code. For instance, this means that a program that triggers 1 hit on a single edge will produce a different feature ID than a program

[1]Figure 1.1 demonstrates an example of this.

[2]We use flags `-fsanitize-coverage=edge,inline-8bit-counters,no-prune`

[3]We choose 70 minutes because out of our 20 fuzzing runs, that is the time that corresponds with the earliest crash.

that triggers 2 hits on the same edge. Whenever a new feature ID is encountered, libFuzzer stores it in a set of all unique features. Every new fuzzer iteration compares the feature IDs of the newly generated test program with that of all previously found unique feature IDs.

Hence, the "Path Cov. Features" that we report in the second and third columns of table 6.1 are the counts of all unique feature IDs of the fuzzing corpus. Compared to the baseline, all mutation strategies are able to improve upon the path coverage of the entire unit test suite. However, among all fuzzing runs, we find that using both of FLUX's crossovers leads to the highest gain in path coverage.

Surprisingly, FLUX's inline mutation, on its own, performs significantly worse than the other mutations, achieving an average of only 323.2 new path coverage features. This is due to the vast amount of mutation failures that occur when running the inlining mutation by itself. As described in §4.2.1, the inlining mutation requires type compatibility between its source and destination functions, and for the vast majority of possible function pairs formed by the entire unit test corpus, the type compatibility constraints cannot be satisfied. Luckily, as we can observe from the synergy between the inlining and sequencing mutations from the first row of table 6.1, the sum of FLUX's crossovers is more than its parts. Each unit test is a single file, and each sequencing mutation increases the function count of a single file by at least 2. Due to the increase in the number of function candidates with each sequencing mutation, the likelihood of the inlining mutation succeeding increases rapidly with time. A takeaway is that the current FLUX prototype benefits from longer fuzzing runs.

To add support to this claim and to better evaluate the inlining mutation, instead of running our mutations over 50,000 functions, we simulate a longer run by taking a selected sample of 71 tests that are more likely to satisfy the inlining constraints. To ensure that this small sample of tests contains an interesting distribution of code, we group each test case by the optimization that they target, then select two functions from each optimization. The two functions that we select from each optimization produce the highest basic block coverage out of the functions in each grouping. This results in 268 unit tests, which we further reduce in order to improve the chances of inlining success. To do this, we keep only the functions that return an integer value, leaving 71 tests. We run the same ablation experiment as earlier with the new seed corpus. We choose a fuzzing time of 30 minutes to match the smaller seed corpus and average over five runs. The results are listed in table 6.2

Mutations	**Path Cov. Features**	**New Coverage**	**Increase (%)**
Inline + Sequence	**690349.2**	**154940.2**	**28.9**
Inline	675820.6	140411.6	26.2
Sequence	639036.0	103627.0	19.4
Baseline	535409.0		

Table 6.2: Path coverage increase on `opt`'s `-O3` optimization pipeline after a 30-minute run on a selected seed corpus of 71 tests. Results are averaged over 5 runs

As the table demonstrates, the inlining mutation performs significantly better when applied to a seed corpus that has a lower probability of mutation failure. As opposed to table 6.1, the inlining mutation in table 6.2 is able to explore more path coverage than the sequencing mutation.

In order to visualize the coverage gain statistics in table 6.1, we plot the growth in path coverage for each fuzzing iteration in figure 6.1. The graph demonstrates that using both FLUX crossovers

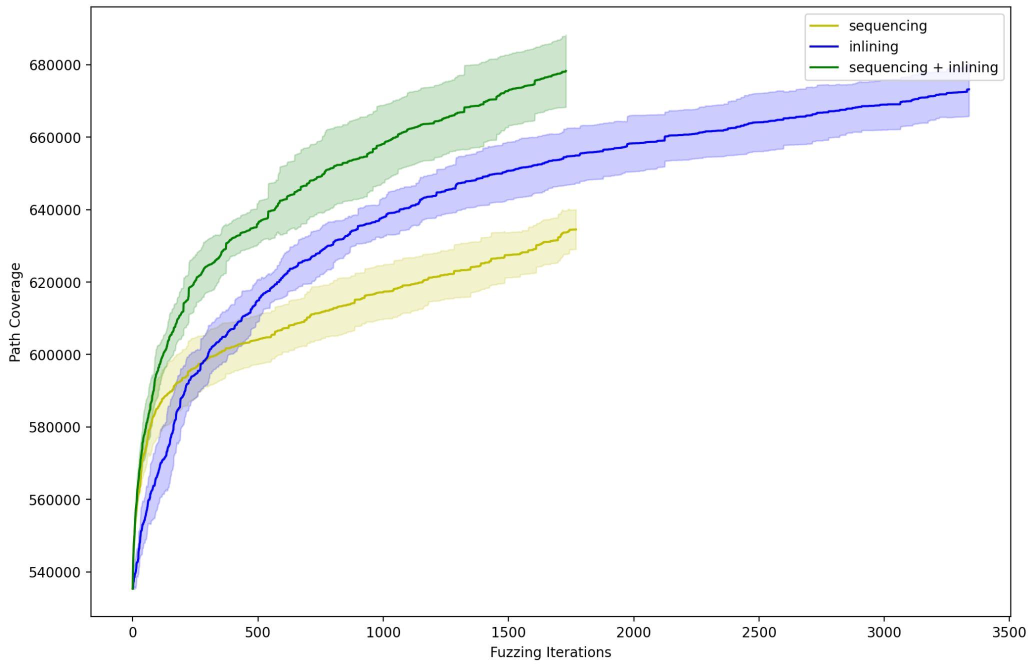


Figure 6.1: Plot of the path coverage obtained by test cases generated with FLUX. The graph compares the FLUX crossovers individually, then in tandem. Each line is averaged over 5 separate 30-minute fuzzing runs. The shaded region around each line represents the standard deviation.

yields the fastest path coverage growth rate. Note that each fuzzing run was capped at 30 minutes, hence there are fuzzing iteration speed discrepancies between mutation configurations. Notably, the inlining mutation is able to execute nearly twice as fast as the other two. The reason is due to the number of test case pairs that are immediately rejected by mutation's type-checking constraints.

In summary, our ablation study has provided evidence that our crossovers indeed are capable of targeting new path coverage. Further, the high mutation failure rate of the inlining mutation suggests that better crossover candidate selection could improve exploration speeds. We discuss this further in chapter 7.

6.3 LLVM Crashes

To evaluate our second research question, **Q2**, we let FLUX loose for three weeks on LLVM's optimizations. We mainly target `opt`'s `-O3` optimization pipeline. We use the entire unit test suite as a seed corpus for our fuzzing runs. In total, we find 24 unique crashes, 21 of which are in the `-O3` pipeline.

Crash Type	**Crashes Found**
LLVM Error	11
Assertion	10
Unreachable	1
Memory Error	2

Table 6.3: Types of crashes found in `opt`

We classify the LLVM crashes bugs into one of two categories:

- *Assertion Failures and Unreachables.* Assertion failures in LLVM immediately crash the compiler if LLVM was built on debug mode. They can be deactivated in release builds. However, this often leads to memory errors or unexpected functionality. We perform our fuzzing runs on a release build of `opt`, hence every crash labeled as "assertion" also crashes on release builds. LLVM's unreachable macro acts the same as an assertion if built in debug mode, but it can not be deactivated, even in release mode.

- *LLVM Error.* An LLVM error occurs when compiler internals can not process an intermediate state of the input IR. In the context of the crashes we discovered, LLVM errors are thrown when valid IR is passed into the optimizer but an intermediate optimization pass transforms it into an invalid state.

- *Memory Error.* Memory errors are detected with AddressSanitizer. For instance, this includes crashes caused by use-after-free, out-of-bounds memory accesses, and null pointer dereferences in the compiler source code.

	Component	**Crashes Found**
Interprocedural	IPO	7
Intraprocedural	AggressiveInstCombine	1
	InstCombine	5
	Scalar	6
	Coroutines	5

Table 6.4: Crashes found in each optimization component

Table 6.3 lists the number of crashes found by their type. We note that FLUX finds an equal distribution of assertion failures and LLVM errors.

If we group the crashes by LLVM optimization component, we also observe a wide range of covered optimizations. Table 6.4 lists the crashes we found by their component. We note that we are able to find a crashing input in nearly all of LLVM's optimization components, despite only targeting the general `-O3` optimization pipeline. This result contributes some evidence towards the generality of FLUX's approach and suggests that we are able to effectively leverage the code coverage of the LLVM unit tests. Hence, we consider the first part of question **Q2** to be a yes.

Table 6.5 gives a more detailed breakdown of every crashing bug that FLUX discovered. The table summarizes the optimization pipeline that was used to find each crash, the optimization code and pass that is responsible for the crash, and the file in which the responsible code resides. We note that besides the `-O3` optimization pipeline, we are also able to find bugs by just targeting the `-instcombine` pass. we discuss this in detail in the next section (§6.3.1).

Target Pipeline	**Component**	**Optimization**	**File**	**Crash Type**
`-O3`	Aggressive-InstCombine	Aggressive-InstCombine	AggressiveInstCombine/AggressiveInstCombine.cpp	LLVM Error
	Coroutines	CoroEarly	Support/Casting.h	Assertion
			IR/Constants.cpp	Assertion
			Coroutines/CoroEarly.cpp	LLVM Error
		CoroSplit	Coroutines/CoroFrame.cpp	Assertion
			Coroutines/CoroSplit.cpp	Memory Error
	InstCombine	InstCombine	ADT/APInt.h	Assertion
			InstCombine/InstCombineVectorOps.cpp	LLVM Error
	IPO	DeadArgument-Elimination	IPO/DeadArgument-Elimination.cpp	LLVM Error
		IPSCCP	IR/Value.cpp	Assertion
			Utils/SCCPSolver.cpp	Assertion
			IR/Constants.cpp	Assertion
		Inliner	CodeGen/BasicTTIImpl.h	Assertion
			Analysis/ConstantFolding.cpp	Unreachable
			IPO/Inliner.cpp	LLVM Error
	Scalar	AlignmentFrom-Assumptions	Support/Alignment.h	Assertion
		GVN	Scalar/GVN.cpp	LLVM Error
				LLVM Error
				Memory Error
		LICM	Scalar/LICM.cpp	LLVM Error
		SROA	Scalar/SROA.cpp	LLVM Error
`-instcombine`	InstCombine	InstCombine	IR/Instructions.cpp	Assertion
			InstCombine/InstCombine-LoadStoreAlloca.cpp	LLVM Error
			InstCombine/Inst-CombineMulDivRem.cpp	LLVM Error
			Total Crashes	**24**

Table 6.5: All crashes found in `opt`. The columns correspond with the optimization pipeline used to find the crash, the LLVM optimization component, the optimization pass, the file, and finally the crash type, respectively. IPSCCP denotes “Interprocedural Sparse Conditional Constant Propagation”. GVN denotes “Global Value Numbering”. LICM denotes “Loop Invariant Code Motion”. SROA denotes ”Scalar Replacement of Aggregates”

6.3.1 Swarm Testing

As we mentioned in §3.2.1, we believe that FLUX's design is well suited towards *swarm testing* [8]. To recap, the original swarm testing paper reduces the output space of *Csmith* by omitting certain grammar features during test generation. The rationale is that a more finely scoped search space exploration will allow deeper exploration in specific compiler components, and indeed the authors are able to find bugs.

As for how this applies to FLUX, a similar narrowing can be applied to FLUX's fuzzing campaigns by simply reducing the scope of the input unit test seed corpus. Further, as we showed during our ablation study in §6.2.1, reducing the number of test cases in FLUX's seed corpus can potentially speed up the rate at which new paths are explored.

To validate this idea, we perform a number of fuzzing runs that only target instruction combine optimizations. Further, we prepare a seed corpus comprised entirely of LLVM unit tests that target the `instcombine` pass. We run FLUX with these configurations for 2 days and are able to find 3 unique bugs. We list the details of these bugs with the rest of our crashes in table 6.5.

This result provides encouraging feedback for a swarm testing approach to using FLUX. We leave further testing of different optimizations to future work.

6.3.2 Crash Case Studies

To answer the second part of question **Q2** and to better understand the mechanisms behind these crashing test cases, we performed a detailed examination of two of our found bugs.

Assertion failure in InstCombine Pass

The first bug we examine is an assertion error in the instruction combine pass. The bug-triggering test case is provided in listing 6.1. The listing causes a crash on the `-instcombine` pipeline.

Listing 6.1: Test case which caused an assertion error in the `instcombine` pass. The location of the failing assertion is in `IR/Instructions.cpp`

```
1  define <4 x double> @invalid_extractelement.SM0.SM0(<2 x double> %a, <4 x double> ↩
       %b, ptr %p) {
2    %t3 = extractelement <2 x double> %a, i32 0
3    %t4 = insertelement <4 x double> %b, double %t3, i32 2
4    %e = extractelement <4 x double> %t4, i32 1
5    store double %e, ptr %p, align 8
6    %e1 = extractelement <2 x double> %a, i32 4
7    %r = insertelement <4 x double> %t4, double %e1, i64 0
8    %t3.i = extractelement <2 x double> %a, i32 0
9    %t3.i1 = extractelement <2 x double> %a, i32 0
10   %t4.i2 = insertelement <4 x double> %b, double %t3.i1, i32 2
11   %e.i3 = extractelement <4 x double> %t4.i2, i32 1
12   store double %e.i3, ptr %p, align 8
13   %t3.i.i = extractelement <2 x double> %a, i32 0
14   %t4.i.i = insertelement <4 x double> %t4.i2, double %t3.i.i, i32 2
15   %e.i.i = extractelement <4 x double> %t4.i.i, i32 1
16   store double %e.i.i, ptr %p, align 8
17   %t3.i2 = extractelement <2 x double> %a, i32 0
18   %t4.i3 = insertelement <4 x double> %r, double %t3.i2, i32 2
19   %e.i4 = extractelement <4 x double> %t4.i3, i32 1
20   store double %e.i4, ptr %p, align 8
21   %t3.i.i5 = extractelement <2 x double> %a, i32 0
22   %t4.i.i6 = insertelement <4 x double> %t4.i3, double %t3.i.i5, i32 2
23   %e.i.i7 = extractelement <4 x double> %t4.i.i6, i32 1
24   store double %e.i.i7, ptr %p, align 8
25   %t4.i = insertelement <4 x double> %t4.i.i6, double %t3.i, i32 2
26   %e.i = extractelement <4 x double> %t4.i, i32 1
27   store double %e.i, ptr %p, align 8
28   ret <4 x double> %t4.i
29 }
```

This test case was produced by a number of inlining mutations on unit tests that target instruction combine optimizations. When InstCombine optimizes the above code, it attempts to replace each chain of `insertelement-extractelement` instructions with the `shufflevector` instruction. It successfully replaces the `insertelement-extractelement` pairs on lines 3-4, 7-8, 10-11, and 14-15 before failing the replacement on lines 18-19. Due to the complicated chain of dataflow, the calculation of the `shufflevector`'s mask operand produces an erroneous result. This malformed mask is then fed into the `shufflevector` constructor, which triggers an assertion failure that checks the validity of the operands in `IR/Instructions.cpp`. Specifically, the assertion error is as follows:

```
Assertion 'isValidOperands(V1, V2, Mask) && "Invalid shuffle vector
instruction operands!"' failed.
```

This error indicates that one of the operands is invalid. Upon investigating the execution of this unit test, we discovered that the first value stored in the `Mask` parameter is greater than the size that is supported by the input vector arguments.

LLVM Error in Inliner Pass

To examine an LLVM Error, we describe a crash discovered in the inliner pass. The test case that causes this crash is listed in 6.2.

Listing 6.2: Test case which caused an LLVM error in the inliner pass. The location of the failing assertion is in `IPO/Inliner.cpp`

```
 1 define i64 @sext.SM(i8 %x, i8 %y, i8 %z) {...}
 2 define <8 x i32> @strided_load_4x2(ptr %in, i64 %stride) {
 3 entry:
 4   %load = call <8 x i32> @llvm.matrix.column.major.load.v8i32.i64(ptr %in, i64 ↩
        %stride, i1 false, i32 4, i32 2)
 5   ret <8 x i32> %load
 6 }
 7
 8 declare <8 x i32> @llvm.matrix.column.major.load.v8i32.i64(ptr nocapture, i64, i1 ↩
      immarg, i32 immarg, i32 immarg)
 9
10 define <8 x i32> @function_sequence_caller() {
11 entryBB:
12   %0 = call i64 @sext.SM(i8 127, i8 127, i8 0)
13   %1 = call <8 x i32> @strided_load_4x2(ptr undef, i64 %0)
14   ret <8 x i32> %1
15 }
```

As evident by the structure of the code, this test case was generated by the sequencing mutation. Namely, the `function_sequence_caller` function was created to call the functions `sext.SM` and `strided_load_4x2` in sequence. We omit the body of `sext.SM` for brevity, as it does not affect the crashing optimization.

Notice that `strided_load_4x2` calls the LLVM intrinsic function `llvm.matrix.column.major` and does little else but return its result. The inliner notices this and attempts to replace the call to `strided_load_4x2` on line 14 directly with a call to `llvm.matrix.column.major`. When the inliner performs the replacement, the value that is passed to the second argument of `strided_load_4x2`, `i64 %0`, is consequently passed to the second argument of `llvm.matrix.column.major`, the stride, `i64 %stride`. This immediately breaks the module as the stride parameter must be greater than 0.

6.4 Miscompilation Bugs

Finally, to evaluate **Q3**, we investigate the effectiveness of FLUX's Alive2 integration. Beyond the crashes we discussed in §6.3, the goal of using a translation validation tool was to detect miscompilations. We define a miscompilation as an optimization bug that produces valid LLVM IR that does not match the behavior of the input LLVM IR. These types of bugs can be the most insidious as they are very difficult to detect and can have unexpected outcomes.

6.4.1 Fuzzing With Alive2

As we described previously (§5.3), in order to integrate Alive2 into our pipeline we had to cater to a number of restrictions imposed by Alive2. We briefly restate the constraints that are relevant to the evaluation:

- Since Alive2 does not support all LLVM IR language features, all unit tests that contain unsupported constructs are pruned from the seed corpus we use for fuzzing in this section.
- Alive2 and the `alive-tv` tool that we use to verify LLVM IR does not support interprocedural optimizations.
- As the `alive-tv` tool's verification pipeline is very similar to the `-O2` pipeline, we decide to also generate tests that target `-O2`.

Hence, following the above constraints, we perform a fuzzing run with the FLUX fuzzer that targets `opt`'s `-O2`. We use a reduced set of 33K unit test functions that have been pruned to Alive2's liking. Although Alive2 does not support interprocedural optimizations, we choose to keep our sequencing mutation in FLUX's crossovers. We had two reasons for this. The first is the synergistic effect that the sequencing mutation had on the inlining mutation (§6.2). The second reason is that Alive2 will still validate an LLVM IR file that contains multiple functions, it just validates each function separately. We perform a two-day fuzzing run and check whether any miscompilations were detected. In total, we were able to find one true miscompilation bug among a large number of false positives. We describe the miscompilation bug we found in §6.4.2 and then describe the challenges we faced when fuzzing with Alive2 in §6.4.3.

6.4.2 Miscompilation Case Study

In this section, we examine the one miscompilation bug that we were able to scavenge. The LLVM IR function that causes the miscompilation and the erroneous optimized code is presented in listing 6.3 on lines 1-10 and 12-20 respectively, note that this listing is directly pulled from the `alive-tv` output. The erroneous optimization that generates the miscompilation is the InstCombine pass. To the right of the miscompiled code, listing 6.4 communicates the counter-example output that is generated by `alive-tv`.

Listing 6.3: An example miscompilation bug detected by Alive2

```
 1 define i32 @bar.SM(i32 %h) {
 2 %0:
 3   %sd = sdiv i32 %h, 2
 4   %t = icmp sgt i32 %sd, 1
 5   %sd.i = sdiv i32 %sd, 2
 6   %t.i = icmp sgt i32 %sd.i, 1
 7   %r.i = select i1 %t.i, i32 %sd.i, ↩
       i32 1
 8   %r = select i1 %t, i32 %sd, i32 %r.i
 9   ret i32 %r
10 }
11 =>
12 define i32 @bar.SM(i32 %h) nofree ↩
     willreturn memory(none) {
13 %0:
14   %sd12 = lshr i32 %h, 1
15   %t = icmp sgt i32 %h, 3
16   %sd.i = sdiv i32 %h, 4
17   %r.i = smax i32 %sd.i, 1
18   %r = select i1 %t, i32 %sd12, i32 %r↩
       .i
19   ret i32 %r
20 }
```

Listing 6.4: An example explanation for the miscompilation in listing 6.4

```
 1 Example:
 2 i32 %h = undef
 3
 4
 5 Source:
 6 i32 %sd = #x00000000 (0)
 7 i1 %t = #x0 (0)
 8 i32 %sd.i = #x00000000 (0)
 9 i1 %t.i = #x0 (0)
10 i32 %r.i = #x00000001 (1)
11 i32 %r = #x00000001 (1)
12
13
14 Target:
15 i32 %sd12 = #x40000000 (1073741824)
16 i1 %t = #x1 (1)
17 i32 %sd.i = #x00000000 (0)
18 i32 %r.i = #x00000001 (1)
19 i32 %r = #x40000000 (1073741824)
20
21
22 Source value: #x00000001 (1)
23 Target value: #x40000000 (1073741824)
```

Notice that a number of instructions in the optimized code on lines 12-20 are direct peephole replacements of instructions in the unoptimized code on lines 1-10. More importantly, the signed division instruction, `sdiv`, on line 3 is replaced by a logical shift right instruction, `lshr`, on line 14. Note that this replacement is already problematic as the logical right shift fills the most significant bits of its result with zeros. This will destroy the signed bit of a negative number, whereas the signed division in the unoptimized code will note. Because of this misoptimization, divergent behaviour can occur between the functions. Listing 6.4 gives an example execution chain of the source and target (i.e. unoptimized and optimized functions respectively) that produces different outputs. By walking through the counterexample of `i32 %h = undef`, the listing demonstrates this divergence.

An undefined value of type T can take any value in the set of defined values for T. Hence, an arbitrary assignment of values to `i32 %h = undef` can cause the `sdiv` and `lshr` results to take on drastically different values. For instance. the example lists results `i32 %sd = 0x00000000` and `i32 %sd12 = x40000000`. As listing 6.4 demonstrates, this leads to different results of the two `icmp` instructions in each function. One `%t` is assigned 0 and the other a value of 1. Because of this, the `select` instructions on lines 7 and 18, having been fed different booleans, will return different values.

6.4.3 Challenges of Fuzzing With Alive2

Even though we are able to find one miscompilation with Alive, the fuzzing process did not go as smoothly as the fuzzing runs that targeted LLVM crashes. The problem we encountered with Alive2 was not that FLUX found no bugs. The problem is that Alive2 reported far too many! The number of miscompiled LLVM IR files for a two-day fuzzing run was on the order of thousands. And due to the non-standard nature of Alive2's output, we could not apply the same triaging methods that we used to de-duplicate LLVM crashes.

Upon manual investigation, we discovered that Alive2 reports a huge number of false positive miscompilations. However, this is at no fault of Alive2. We find that any optimizations that rely on interprocedural information will have a high likelihood of triggering a miscompilation. Consider listing 6.5 below. This listing shows the output of the `alive-tv` tool on a test case generated by the sequencing mutation. The crossover output demonstrates that the sequencing mutation failed to inject any dataflow and defaulted to calling each function serially. The tool prints the input IR before and after optimization which corresponds with lines 1-6 and 8-11 respectively.

Listing 6.5: Example of an Alive2 false positive on a test case generated by the sequencing mutation

```
 1 define void @function_sequence_caller() {
 2 %entryBB:
 3   %0 = call <2 x i8> @t4_vec(<2 x i8> { 0, 0 })
 4   call void @f.SM()
 5   ret void
 6 }
 7 =>
 8 define void @function_sequence_caller() nofree willreturn memory(none) {
 9 %entryBB:
10   ret void
11 }
12 Transformation doesn't verify!
13 ERROR: Source and target don't have the same return domain
```

The `alive-tv` tool also prints a counterexample that proves that the optimization does not hold for all cases. For listing 6.5, it states that if the function call `t4_vec` never returns, then the return domains of the pre and post-optimized code will be different. That is, the post-optimized code will always return.

This would be a miscompilation if `t4_vec` were to never return however, it does return and Alive2 cannot detect this because it has no knowledge of anything outside of a function-level scope. Therefore, any unit tests that contain function calls may lead to false positives. Hence, our current sequencing mutation is grossly incompatible with Alive2. Nevertheless, this is not an unfixable problem and we leave a more suitable fuzzing configuration for future work.

Despite the vast amount of false positives, we still suspect that the inlining mutation by itself can lead to positive bug-finding results with Alive2. As evidenced by the one miscompilation we discovered in the previous section, we believe an inlining-only approach to fuzzing can yield more bugs. We note that it is probable that more true miscompilations exist in our crossover corpus however, the number of miscompilation was too vast to explore within our given time constraints. Hence, due to a lack of manpower and willpower, we leave this effort to future work. To answer question **Q3**, the response is a weak yes.

Chapter 7

Limitations and Future Work

Although we have shown that FLUX is able to effectively explore new path coverage and find a wide variety of new bugs, FLUX still contains many design and implementation limitations. These limitations are briefly discussed in scattered parts throughout the paper. We collect these limitations here and use them to motivate direction for future work.

Generalisability. We acknowledge that much of FLUX's design relies intimately on LLVM specifics. Namely, LLVM IR provides a natural interface into LLVM's optimizer. Compared to the source level fuzzer we discussed in §3.2.1 which can target any compiler with a compatible frontend, the current FLUX prototype can only test the LLVM compiler. Despite this, we believe that the usage of a high coverage unit test suite in combination with our sequencing and inlining mutations can be applied to testing other compilers. For instance, this method may also work with some of GCC's intermediate representations. We leave this exploration to future work.

Reducing mutation failures. As demonstrated by §6.2.1, the FLUX's prototype inlining mutation suffers from a high rate of failure. This leads to many wasted iterations during fuzzing. We explained that randomly selecting pairs from the entire unit test corpus naturally leads to a high chance of type incompatibility for our crossover. We envision two directions that can mitigate this limitation.

- The first is a significant reduction in the scope of the unit test seed corpus. Namely, reducing the seed corpus to only contain tests more targetted tests[1] or reducing the seed corpus to only contain tests that have a higher likelihood of inlining success [2]. Further, instead of manual reduction as we performed in this thesis, future work can potentially build this into the fuzzer seed selection driver itself.

- A second approach to reducing the number of failures is modifying the inlining mutation such that the crossover is always performed, even with type incompatibility. One option is simply inlining the source function arbitrarily without any dataflow linkage. However, we suspect that this type of crossover will only succeed in stressing dead code elimination passes.

[1]Much like the swarm testing we conducting in §6.3.1
[2]Similar to the curated unit test set that we used in §6.2.1

Better Alive2 integration. To summarize our fuzzing woes from §6.4.3. We found that our current fuzzing design (Fig. 5.1) is incompatible with the validation capabilities of Alive2. Specifically, our sequencing mutation led Alive2 to report thousands of false positive miscompilations. Despite this, we intuit that a fuzzing pipeline that leverages only intraprocedural optimization unit tests along with our inlining mutation will be able to detect more miscompilation errors.

Chapter 8

Conclusion

Given the vast complexity of optimizing compilers, it remains a continual effort to find new bugs in these widely-used tools. This thesis contributes a targeted bug-finding method that explores the compiler's middle-end optimizations. Our approach posits that exploring new path coverage through the compiler's optimization code will lead to the discovery of new bugs. However, generating code that is even able to trigger optimization paths is a challenge, one that previous work has mitigated with heuristics. Our thesis makes the observation that compilers such as LLVM contain suites of unit tests that are already capable of stressing most compiler optimization components.

To leverage these high-coverage unit tests, we propose FLUX, a fuzzer that is designed to generate test cases that explore new execution paths through the optimizations that the unit tests target. We contribute two novel crossovers, the sequencing and inlining mutations, which target interprocedural and intraprocedural optimizations respectively. We demonstrate the efficacy of FLUX by conducting fuzzing runs on LLVM's middle-end optimization pipeline `opt`. We find that both of FLUX's crossover mutations are capable of exploring new path coverage and that the combination of the two mutations produces a synergistic effect. Further, when used in extended fuzzing runs, FLUX is able to find 24 crashes in LLVM, 21 in the `-O3` optimization pipeline, and 3 in the `-instcombine` pass. The variety in the types of crashes and in the optimization passes in which the crashes occur, reveal the general applicability of FLUX's fuzzing approach. Finally, FLUX is also able to find compiler miscompilations by using existing translation validation tools to validate the fuzzer's output.

Bibliography

[1] *american fuzzy lop*. URL: https://lcamtuf.coredump.cx/afl/ (visited on 03/13/2023).

[2] Vard Antinyan et al. "Mythical Unit Test Coverage". In: *IEEE Software* 35.3 (May 2018). Conference Name: IEEE Software, pp. 73–79. ISSN: 1937-4194. DOI: 10.1109/MS.2017.3281318.

[3] Scott Bauer, Pascal Cuoq, and John Regehr. "Deniable Backdoors Using Compiler Bugs". In: *Int. J. PoC——GTFO* 0x08 (2015), pp. 7–9.

[4] Junjie Chen and Chenyao Suo. "Boosting Compiler Testing via Compiler Optimization Exploration". In: *ACM Transactions on Software Engineering and Methodology* (Mar. 5, 2022), p. 3508362. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3508362.

[5] Junjie Chen et al. "A Survey of Compiler Testing". In: *ACM Computing Surveys* 53.1 (Feb. 6, 2020), 4:1–4:36. ISSN: 0360-0300. DOI: 10.1145/3363562.

[6] Chris Cummins et al. "Compiler fuzzing through deep learning". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, July 12, 2018, pp. 95–105. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213848.

[7] Edsger W Dijkstra. "I. Notes on Structured Programming". In: *Academic Press* (1972).

[8] Alex Groce et al. "Swarm testing". In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, July 15, 2012, pp. 78–88. ISBN: 978-1-4503-1454-1. DOI: 10.1145/2338965.2336763.

[9] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments". In: 21st USENIX Security Symposium (USENIX Security 12). 2012, pp. 445–458.

[10] He Jiang et al. "CTOS: Compiler Testing for Optimization Sequences of LLVM". In: *IEEE Transactions on Software Engineering* 48.7 (July 2022). Conference Name: IEEE Transactions on Software Engineering, pp. 2339–2358. ISSN: 1939-3520. DOI: 10.1109/TSE.2021.3058671.

[11] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler validation via equivalence modulo inputs". In: *ACM SIGPLAN Notices* 49.6 (June 9, 2014), pp. 216–226. ISSN: 0362-1340. DOI: 10.1145/2666356.2594334.

[12] Vu Le, Chengnian Sun, and Zhendong Su. "Finding deep compiler bugs via guided stochastic program mutation". In: *ACM SIGPLAN Notices* 50.10 (Oct. 23, 2015), pp. 386–399. ISSN: 0362-1340. DOI: 10.1145/2858965.2814319.

[13] Xiao Liu et al. "DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.1 (July 17, 2019). Number: 01, pp. 1044–1051. ISSN: 2374-3468. DOI: 10.1609/aaai.v33i01.33011044.

[14] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. "Random testing for C and C++ compilers with YARPGen". In: *Proceedings of the ACM on Programming Languages* 4 (OOPSLA Nov. 13, 2020), 196:1–196:25. DOI: 10.1145/3428264.

[15] *LLVM Enjoyed Record Growth In 2021, Many Exciting Compiler Advancements.* URL: https://www.phoronix.com/news/LLVM-Record-Growth-2021 (visited on 03/10/2023).

[16] *LLVM Language Reference Manual.* URL: https://llvm.org/docs/LangRef.html (visited on 03/19/2023).

[17] Nuno P. Lopes et al. "Alive2: bounded translation validation for LLVM". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Virtual Canada: ACM, June 19, 2021, pp. 65–79. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454030.

[18] Nuno P. Lopes et al. "Provably correct peephole optimizations with alive". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '15. New York, NY, USA: Association for Computing Machinery, June 3, 2015, pp. 22–32. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737965.

[19] Valentin J. M. Manes et al. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *arXiv:1812.00140 [cs]* (Apr. 7, 2019). arXiv: 1812.00140.

[20] Michaël Marcozzi et al. "Compiler fuzzing: how much does it matter?" In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA Oct. 10, 2019), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3360581.

[21] William M McKeeman. "Differential Testing for Software". In: *Digital Technical Journal* 10.1 (1998).

[22] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[23] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. "Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions". In: *IPSJ Transactions on System LSI Design Methodology* 7.0 (2014), pp. 91–100. ISSN: 1882-6687. DOI: 10.2197/ipsjtsldm.7.91.

[24] Eriko Nagai et al. "Random Testing of C Compilers Targeting Arithmetic Optimization". In: (SASIMI 2012 2012), pp. 48–53.

[25] *SanitizerCoverage — Clang 17.0.0git documentation.* URL: https://clang.llvm.org/docs/SanitizerCoverage.html (visited on 03/15/2023).

[26] Tilmann Scheller. *Where is LLVM being used today?* URL: https://llvm.org/devmtg/2016-01/slides/fosdem-2016-llvm.pdf.

[27] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *2012 USENIX Annual Technical Conference* (USENIX ATC’12 2012), p. 10.

[28] Kosta Serebryany. “Continuous Fuzzing with libFuzzer and AddressSanitizer”. In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016 IEEE Cybersecurity Development (SecDev). Nov. 2016, pp. 157–157. DOI: 10.1109/SecDev.2016.043.

[29] Chengnian Sun, Vu Le, and Zhendong Su. “Finding compiler bugs via live code mutation”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. New York, NY, USA: Association for Computing Machinery, Oct. 19, 2016, pp. 849–863. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984038.

[30] Chengnian Sun et al. “Toward understanding compiler bugs in GCC and LLVM”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, July 18, 2016, pp. 294–305. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931074.

[31] *The AFL++ fuzzing framework*. AFLplusplus. URL: https://aflplus.plus/ (visited on 03/25/2023).

[32] Xuejun Yang et al. “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. New York, NY, USA: Association for Computing Machinery, June 4, 2011, pp. 283–294. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993532.

[33] Qirun Zhang, Chengnian Sun, and Zhendong Su. “Skeletal program enumeration for rigorous compiler testing”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’17: ACM SIGPLAN Conference on Programming Language Design and Implementation. Barcelona Spain: ACM, June 14, 2017, pp. 347–361. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062379.

[34] Zhide Zhou et al. “An empirical study of optimization bugs in GCC and LLVM”. In: *Journal of Systems and Software* 174 (Apr. 2021), p. 110884. ISSN: 01641212. DOI: 10.1016/j.jss.2020.110884.

[35] Hong Zhu, Patrick A. V. Hall, and John H. R. May. “Software unit test coverage and adequacy”. In: *ACM Computing Surveys* 29.4 (Dec. 1, 1997), pp. 366–427. ISSN: 0360-0300. DOI: 10.1145/267580.267590.