FLUX: Finding Bugs with LLVM IR Based Unit Test Crossovers

Eric Liu
University of Toronto
Canada
ec.liu@mail.utoronto.ca

Shengjie Xu
University of Toronto
Canada
shengjie.xu@mail.utoronto.ca

David Lie
University of Toronto
Canada
lie@eecg.toronto.edu

***Abstract*—Optimizing compilers are as ubiquitous as they are crucial to software development. However, bugs in compilers are not uncommon. Among the most serious are bugs in compiler optimizations, which can cause unexpected behavior in compiled binaries. Existing approaches for detecting such bugs have focused on end-to-end compiler fuzzing, which limits their ability for targeted exploration of a compiler's optimizations.**

This paper proposes FLUX (*F*inding bugs with *L*LVM IR based *U*nit test cross(*X*)overs), a fuzzer that is designed to generate test cases that stress compiler optimizations. Previous compiler fuzzers are overly constrained by having to construct well-formed inputs. FLUX sidesteps this constraint by using human-written unit test suites as a starting point, and then selecting random combinations of them to generate new tests. We hypothesize that tests generated this way will be able to explore new execution paths through compiler optimizations and find new bugs. Our evaluation of FLUX on LLVM indicates that it is able to increase path coverage over the baseline LLVM unit test suite and explores more edge coverage than previous work. Further, we demonstrate FLUX's ability to generate miscompiled and crash-producing IR on LLVM's optimizations. After a month of fuzzing, FLUX found 28 unique bugs in LLVM's active development branch. We have reported 11 of these bugs which led to 6 of them being patched by LLVM developers. 22 of these are crashes that are triggered by well-formed input programs, and 6 of these are miscompilation bugs that silently produced incorrect code.

***Index Terms*—compiler testing, fuzzing, miscompilation, compiler defect, crossover**

I. Introduction

Compilers are an indispensable component of any modern development pipeline. Popular optimizing compilers like GCC and LLVM have grown to contain tens of millions of lines of code [1]. As with any code base that is highly complex, bugs in compilers are difficult to avoid. However, compilers are often blindly trusted by end users and are expected to preserve source program semantics. Unfortunately, these assumptions are not always correct, despite the best efforts of compiler developers.

Compiler bugs can range from inconvenient compile time crashes to potentially dangerous miscompilations that can cause unexpected behaviour in the resultant binary. LLVM for instance reports hundreds of bugs each month [2], [3]. Undetected bugs in LLVM have real-world consequences. For instance, the LLVM infrastructure has been integrated into consumer-facing technologies, such as the Android NDK and PS4 SDK [4]. Beyond quality of service degradation and the occasional unexpected behaviour, bugs in LLVM can also be exploitable [5]. Hence, ensuring that a compiler like LLVM remains free of crashes and miscompilations is a difficult but crucial goal.

To find compiler bugs, previous work has employed fuzzing to automatically generate bug-triggering test cases. Unlike general-purpose software fuzzers [6]–[8], compiler fuzzers must respect the syntactic and semantic constraints of the source language. Any fuzzer that does not respect these conventions will quickly become stuck in the compiler's lexer, parser, and type checker.

Popular compiler fuzzers have tried a variety of methods to generate valid tests. Some generate tests cases with the aid of a grammar [9], [10], heuristics [11], [12], deep learning [13], [14], and by mutating existing tests to form new valid inputs [15]–[18].

Although successful in finding bugs, previous work is not able to fully explore the entire space of input programs to their target compilers [10]. This means that after sufficient time, compiler fuzzers saturate their entire output space, which is a biased subset of potential inputs. This bias leads to under-explored components, such as the compiler's middle-end optimizations. Due to a focus on end-to-end fuzzing, existing work has relied on heuristics and intuition to stress optimizations. For instance, YARPGen [10], restricts portions of its generation scheme to only use arithmetic, logical, and bitwise operations. Their intuition is that dense clusters of these operations will better trigger certain peephole optimizations. Unfortunately, heuristics like this may also suffer from bias or are narrow in scope, only focusing on specific optimizations. As opposed to source-level fuzzing, we suggest IR-level test generation as an effective way to target optimizations.

Instead of generating test cases from the bottom up, our work makes a key observation: compilers like LLVM contain a large suite of high-quality unit tests that are *known* to stress 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 high code coverage [19]. This work aims to leverage these feature-rich unit tests in order to generate new test cases.

However, despite the high line-coverage that the unit tests provide, new optimization bugs continue to be found. In fact, previous studies have shown that coverage metrics like line coverage are weakly correlated with actual bugs [20]. Hence, a single execution pass through each code segment is insufficient, we must also explore the same code in varying orders and with varying numbers of iterations. In essence, we believe that increasing *path coverage* through the compiler will lead to more found bugs.

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*. Though the space of crossover mutations is, in theory, infinite, we begin by proposing two simple crossover mutations. The first is a function-level mutation that composes unit tests in a manner that is akin to *inlining*. The second is a module-level mutation that we denote as a *sequencing* crossover.

We implement FLUX on top of libFuzzer [8]. FLUX targets LLVM and generates LLVM IR test cases to stress LLVM's middle-end optimizations. To evaluate the effectiveness of FLUX's crossovers, we measure FLUX's ability to find new paths and new edges through LLVM's optimizer. We estimate path coverage with libFuzzer's logarithmic edge count bucketing. Our evaluation of FLUX yields promising results. We show that FLUX's crossover mutations are able to find more new path coverage through LLVM's optimizer than CSmith, an industry-standard compiler fuzzer [9] and more new edge coverage than YARPGen [10], a state-of-the-art fuzzer.

Furthermore, FLUX is able to find a significant number of compiler bugs. To detect miscompilations, we rely on Alive2 [21] as a bug oracle and we instrument LLVM with AddressSanitizer to detect memory errors. Over the course of a month, FLUX triggers 28 unique bugs in LLVM optimizations, 22 of which are various types of crashes in the optimization code and 6 are miscompilation bugs. 6 of our reported bugs have already been acknowledged and patched by LLVM developers.

To summarize, this paper makes the following contributions:

- We propose *crossovers*, which combine high line-coverage test cases in order to generate new path coverage in LLVM.
- Two novel crossover mutations, the *inlining* and the *sequncing* mutations, which are capable of exploring new paths through existing unit test code.
- A prototype implementation 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 28 bugs during our evaluation.

We begin in Section II with background on LLVM and its unit test suite, which FLUX uses as inputs. Next, we detail FLUX's design in Section III, and implementation details in Section IV. We evaluate FLUX's ability to increase edge and path coverage, as well ability to find LLVM bugs in Section V and discuss generalisability and threats to validity in Section VI. Finally we conclude in Section VII and provide details on anonymous data availability in Section VIII.

II. Background

A. LLVM

Much of LLVM's appeal lies in the modularity of its components, which is facilitated by LLVM's intermediate representation (IR). The middle-end optimization pipeline `opt` optimizes the input IR through a series of transformation 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.

TABLE I
Transformation unit test suit coverage

Component	Total LoC	Missed LoC	Coverage (%)
InstCombine	28820	3264	89.61
IPO	35731	20140	43.63
Scalar	53534	30692	42.67
Utils	39420	16688	57.67
Vectorize	25804	6272	75.69
Total	**183309**	**76786**	**58.11**

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 targeting compiler optimizations, there are around 8000 test files labeled as "Transformation" tests which contain over 60000 LLVM IR functions. To give a sense of the thoroughness of the unit test suite, Table I summarizes the line coverage achieved by passing the unit test suite through `opt` at the `-O3` optimization level with `x86_64` as the target architecture. The table lists the main optimization directories in LLVM. Note that many optimizations rely on target architecture information, which results in missed coverage in our `opt` run. Further, many transformations, such as a number of interprocedural optimizations and scalar optimizations, are not enabled on the default optimization levels. Despite this, the unit tests still obtain 58.11% line coverage with a single `opt` configuration. Most impressively, the unit tests are able to achieve near 90% on LLVM's InstCombine source code.

Despite the high unit test coverage, previous work has found that, across all compiler components, optimizations rank among the top ten buggiest components in LLVM and GCC[1] [3]. These statistics are not just historical artifacts. Recently, researchers have had success in finding optimization bugs by simply selecting appropriate optimization orders and settings to differentially test LLVM [22], [23]. Moreover, even with 90% coverage from the unit test suite, other work has found InstCombine to be the buggiest optimization in LLVM [2]. Even with its shortcomings code coverage remains a popular guidance metric among compiler fuzzers [24].

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

B. Related Compiler Fuzzers

We discuss related generation-based fuzzers (II-B1) which create test programs from scratch, and mutation-based fuzzers (II-B2) which mutate existing test cases to form new test programs. We note that the fuzzers described in this section generate source languages and most also target the LLVM compiler.

1) Generative Compiler Fuzzers: Likely the most widely used compiler fuzzer, Csmith [9] generates 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 [25] across different compilers. Csmith inspired numerous follow-up works that either build upon [24] or utilize it [22], [23]. For instance, one testing approach called swarm testing [26] utilizes Csmith by omitting grammar features during test generation in order to "swarm" a smaller subset of the compiler.

Despite its success, Csmith has exhausted its output space in recent years. As a result, YARPGen a C/C++ generator was released [10]. 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. For instance, one policy might favour arithmetic expressions to target sub-expression elimination optimizations. In a similar vein, other fuzzers have focused solely on generating programs that target arithmetic optimizations [11], [12]. These targeting strategies are heuristic-based and rely largely on the intuition of their authors.

Finally, we briefly mention compiler fuzzers that leverage neural networks to perform program generation. Previous approaches have utilized recurrent neural networks to automatically generate C [14] and OpenCL programs [13].

2) Mutation-based Compiler Fuzzers: We first discuss mutational fuzzers that use *semantics preserving mutation* [24] to modify an existing test to generate different but behaviourally equivalent programs. Because of their semantic equivalence, these programs enable differential testing on a single compiler. Orion [15] generates semantically equivalent programs by randomly pruning non-executed parts of a seed program. Athena, additionally inserts code into unexecuted regions [16]. Beyond mutating dead code, Hermes, introduced semantics preserving mutation in live code [17]. They accomplish this by inserting random code that is side-effect free into paths that the program actually executes. Similar to the generative approaches described earlier, these fuzzers focus on end-to-end testing and hence value semantic validity more than targeting optimizations.

As for mutations that do not preserve program semantics, skeletal program enumeration (SPE) [18] mutates program skeletons. A program's skeleton is defined as a program that has each of its variable definitions and uses represented as placeholders. SPE then enumerates every possible assignment of program variables into these placeholders. The authors then propose a way to minimize the set of enumerated programs to avoid generating functions that are equivalent. SPE does not prevent undefined behavior in its output and relies on manual inspection to detect miscompilations, yet finds a vast number of bugs. Another mutation-based fuzzer, LangFuzz [27], 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 invalid behaviour in the past. In contrast to the rest of the fuzzers in this section, which generate C or C++, LangFuzz targets the JavaScript interpreter.

C. Translation Validation

Orthogonal to compiler fuzzing, translation validation approaches compiler bug-finding from a formal verification angle. One previous work, Alive2 [21], [28], verifies optimizations by mapping the correctness of the 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.

Many of the fuzzers we discussed in II-B that rely on differential testing are careful to avoid introducing undefined behaviour into their generated tests. Compiler optimizations have free rein to make arbitrary transformations on the code if it contains undefined behaviour [29]–[31]. 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.

III. FLUX's Design

FLUX's high-level goal is to detect compiler bugs and miscompilations in LLVM's middle-end optimizations. To motivate the design decisions we made in working towards this goal, we note the following about previous compiler fuzzers:

Most previous work (II-B) spends considerable effort respecting source program semantics and ensuring their generated tests remain undefined behaviour free. Although these design choices have proved successful for end-to-end compiler testing, it limits a fuzzer's ability to specifically target middle-end optimizations. We identify two limitations of existing approaches:

- **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, while also generating well-formed test cases to pass the frontend checks of the compiler. For instance, generating tests for a specific source language will not allow a complete exploration of LLVM IR features; LLVM IR is a general representation that numerous source languages can compile to. Further, the source code is at the mercy of the compiler frontend's lowering process.
- **L2:** Compiler fuzzers that do target optimizations, rely on heuristics and intuition to trigger optimization code.

This is not ideal for scaling and automating the testing of the hundreds of optimizations that LLVM uses. The prerequisite code structures for many optimizations are very specific and frequently changing, making existing intuition-based guidance insufficient.

To address these limitations, we design FLUX's mutations to leverage LLVM's high code coverage unit tests to generate programs that are able to explore new paths through the compiler's optimizations. First, as the unit tests are implemented in LLVM IR, we conveniently avoid the compiler's frontend and since we observe a direct mapping from unit test to LLVM optimization, we need not rely on randomness and heuristics to target optimizations (**L1**). Moreover, operating at the LLVM IR level allows for source-language agnostic testing of LLVM's optimizations. Second, LLVM unit tests are fairly complete (II-A) and contain a wealth of LLVM developer knowledge. Beyond coverage, these tests contain insights into edge cases and programs that previously triggered errors. 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 (**L2**). Instead, we need to ensure that the mutated unit tests retain enough of their original structure such that they still stress the same optimization code, but explore new paths through the same coverage. We classify FLUX as a non-semantics preserving mutation-based fuzzer, akin to skeletal program enumeration [18] and LangFuzz [27] albeit the nature of our FLUX's mutations is quite different.

A. Crossover Design Goals

To generate new test cases that inherit the optimization-hitting properties of LLVM's unit tests, we consider ways to combine unit tests with a *crossover* mutation. We envision our crossover as a method of building input programs that use feature-rich unit tests as building blocks. Our intuition is that combining tests across disparate code structures and target optimizations will yield interesting exploration. Further, many unit tests contain IR that previously triggered bugs. A suitable crossover mutation will ideally implant these previously problematic tests into unexplored contexts. To realize these goals, we note two things when designing our crossover mutation.

1) Structure preservation: The unit tests are foundational components that encode developer knowledge on what prerequisite code structure is needed to trigger and stress optimizations. Hence, the crossover should try to preserve the structures of its inputs, when generating new tests. This means we should avoid any mutation that alters the test code in an unprincipled way, lest we lose the targeting benefits of using the high-coverage unit tests. A crossover mutation that arbitrarily clobbers unit test code will struggle to explore new paths through the same optimizations.

2) New path exploration: The crossover should 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 new paths. For instance, consider a crossover that takes as input two LLVM IR functions and simply returns a module that contains the two functions. Undoubtedly, the input coverages are preserved since we have not modified any code structure. However, the extra exploration that this provides is minimal.

We propose two crossovers that strike a balance between structure preservation and new path exploration. One which mutates inputs at the function level (III-B) and one that operates at the module level (III-C).

B. 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 with 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, instead of inlining an existing function call, 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.

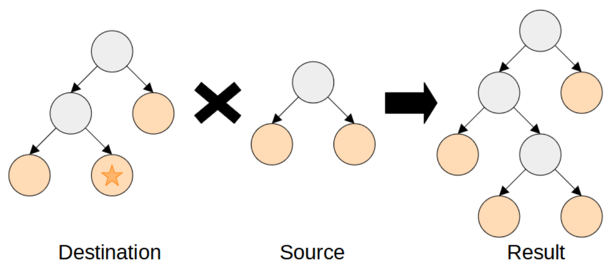

Fig. 1. Inlining mutation on two ASTs. The star denotes the insertion location.

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

To reduce the chance that dead code elimination trivially deletes our inserted code, we link the dataflow of the destination function with the source function's body—a form of dataflow stitching. We accomplish this by finding type-compatible variables in the destination function to pass into the source function's arguments. We then transplant the return value of the call site into an operand of the destination function instruction that immediately follows our inserted call site. We denote this destination operand as the *insertion location*. From an 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, as shown in Figure 1).

Algorithm 1 describes the inlining process in detail. Line 2 collects all potential insertion locations in the destination function. Collecting these insertion locations 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 iterate over each candidate insertion location to

Algorithm 1: Function inlining mutation

```
1  Function InlineMutation($D$, $S$):
       Input: destFunc: $D$, srcFunc: $S$
       Output: Crossover Result
2      $Loc \leftarrow$ getAllInsertionLocations($D$);
3      for $\ell \in Loc$ do
4          $C \leftarrow$ getAllArgCandidates($D, \ell$);
5          if $S.ArgTypes \subseteq C.ArgTypes$ then
6              $S_{InArgs} \leftarrow \{\}$;
7              for $a \in S.Args$ do
8                  $\alpha \leftarrow$selectRandArg($C, a$.type);
9                  $S_{InArgs}$.add($\alpha$);
10             $\beta \leftarrow D$.insertCallAt($S, S_{InArgs}, \ell$);
11             $\ell$.setOperand($\beta$);
12             $\beta$.inlineFunction();
13             return $D$;
```

determine its feasibility. A feasible insertion location must allow a sufficient number of type-compatible variables to use as arguments to the source function. Hence, we collect all potential argument candidates on line 4. To abide by dominance constraints, we can only consider variables that are defined in blocks that dominate the insertion location. We achieve this with 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 5, we check if we have enough typed values that the source function arguments require.

Now that the algorithm has established that the insertion location is valid, on line 8, it randomly selects arguments to use. The algorithm then inserts a call instruction to the source function, links the return value dataflow, and finally inlines 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.

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

C. *Sequencing Mutation*

Next, we introduce a crossover mutation that operates at the module level by introducing dependencies between functions. To inject dependencies 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.

Simply calling the component functions in sequence may be optimized away by dead code elimination. To reduce the chance of this, we inject interprocedural dataflow into the caller function. To do so, we consider three sequencing strategies in the caller function that inject dataflow:

1) Return Value Chaining Strategy: We pass the return value of the first function call into one of the arguments of the second function call. The caller function then returns the second function call's value. The prerequisite for this mutation is that the return value type of one function matches one of the argument types of the other function.

2) Pointer Argument Strategy: We pass the same pointer value (i.e. an LLVM IR `alloca` instruction) to both functions. The input functions are eligible for this strategy if a pointer argument of the same type exists in the parameters of both functions.

3) Binary Operation Strategy: If both input functions have return values that are compatible with a certain binary operation(s), we insert dataflow by randomly generating one of these operations, and sending both values into the operation as operands. We take the result of the binary operation as the return value of the caller function. For instance, a binary `add` instruction could be generated if both functions return an `i32` integer.

For each of the above sequencing strategies, we patch any 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.

D. *Fuzzer Structure*

To utilize the inlining and sequencing mutations, we integrate our two crossovers with a fuzzing engine. Figure 2 gives a high-level overview of FLUX's structure and main fuzzing loop. The FLUX fuzzer first initializes the fuzzing corpus to contain all LLVM transformation unit tests.

The workflow of each fuzzer iteration in the main loop is denoted by each of the numbered arrows. First, two unit test modules are selected from the fuzzing corpus. These unit tests, which we denote as the "M1" and "M2" in the figure, are passed into FLUX's crossovers. The engine then randomly selects one of the inline or sequencing mutations to generate a crossover result which is then fed to a coverage-instrumented LLVM optimization pipeline. If any optimizations crash or trigger a memory error, then we exit the fuzzing loop and generate a crash report. Otherwise, step 3 returns control to the fuzzer engine's behavior monitor. If the coverage bitmaps indicate that the crossover result triggered new coverage, then the test case is saved to the corpus in step 4. Finally, test cases in the fuzzing corpus are also fed to Alive2 to detect miscompilations.

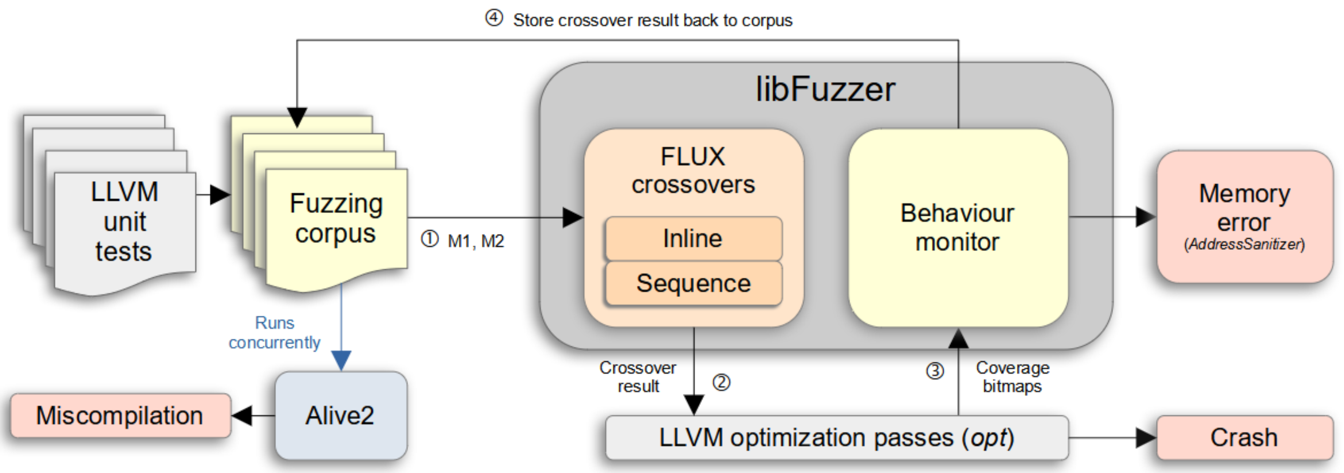

Fig. 2. Overview of the FLUX fuzzer. The orange components are contributions of this paper. The numbered edges denote the main fuzzing loop.

IV. Implementation

To implement the structure described in Section III-D, we build our FLUX mutations on top of libFuzzer's engine [8]. libFuzzer is a coverage-guided fuzzer that is linked with the program or library under test (i.e. `opt`). We implement FLUX's crossovers on LLVM's main active development branch, which at the time of writing this paper, is on LLVM 16.0. To evaluate our crossovers, we extend an existing naive IR fuzzer in LLVM called `llvm-opt-fuzzer`. We disable all of the existing fuzzer's mutations and drop in the implementation of our crossover, which is comprised of around 2000 lines of C++ code.

To detect memory errors, we instrument LLVM's optimization pipeline with AddressSanitizer [32]. Further, we compile LLVM in debug mode to find crashes that are caused by assertion failures and LLVM errors during our fuzzing runs. The rest of this section describes implementation-specific challenges and considerations we faced when developing our FLUX prototype.

A. LLVM IR Constraints

It should be noted that the crossover algorithms that we introduced in Section III, do not produce correct code for all possible LLVM IR inputs. LLVM's well-formedness constraints were challenging to conform to and contain many esoteric requirements. Luckily, LLVM provides an internal verifier that checks the well-formedness of any input LLVM IR. We rely on the verifier to filter out any invalid IR that is generated by our mutations so that we do not pollute the crossover corpus with invalid IR. For the vast majority of cases, our crossover produces correct code. However, for efficiency reasons, we did make small tweaks to our algorithm to handle edge cases. For example, we tweaked our algorithm to exclude certain intrinsic instructions from being used as insertion locations.

B. Alive2 Considerations

We use Alive2's standalone validation tool, `alive-tv`[2] to verify our generated LLVM IR files. Unfortunately, as previously mentioned in Section II-C, Alive2 does not support any interprocedural optimizations. Hence, the transformations that `alive-tv` can verify are limited to intraprocedural passes. Specifically, `alive-tv` verifies an optimization pipeline that is similar to `opt`'s `-O2` optimization level, but without interprocedural optimizations. This means that FLUX as a whole, cannot automatically detect miscompilations in interprocedural optimizations. However, it can still catch crashes and LLVM errors in these passes.

Another Alive2 constraint that we consider in our implementation is its memory and complexity limits. Since Alive2 relies on an SMT solver, it limits the size and complexity of its input programs. For instance, Alive2 will throw an out-of-memory error after consuming 500 MB of data. Hence, any test case that fails to meet Alive2's resource restrictions will cause wasted cycles. To amend this, we use a Python script that orders each file in the crossover corpus by increasing file size. The script then continually feeds the smallest tests to `alive-tv` for verification. Further, due to the time overhead of the SMT solver, we also choose to run the script in a separate thread that continually monitors the fuzzing corpus (Figure 2)

A final implementation consideration concerning Alive2 is 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. When performing fuzzing runs with our concurrent Alive2 script, we want to minimize the amount of unsupported behaviour in tests generated by FLUX. Hence, in these runs, we choose to reduce the LLVM unit test suite that is loaded into FLUX to only contain functions that are supported by Alive2. Note that FLUX is able to run without Alive2 with

[2]https://github.com/AliveToolkit/alive2

the entire unit test suite, and indeed we will demonstrate that we are still able to detect compiler crashes without it.

V. Evaluation

To validate FLUX's design (III) choices, we ask the following research questions:

RQ1 Are FLUX's *inlining* and *sequencing* mutations able to find new path coverage through LLVM's optimizations? How does FLUX compare to existing fuzzers? (V-B)

RQ2 Is FLUX able to find new crashes in LLVM's optimizations? If so, what do these crashes look like? (V-C)

RQ3 Is FLUX's Alive2 integration able to find new miscompilation bugs in LLVM's optimizations? (V-D)

A. Fuzzing Setup

For our path coverage evaluation, we perform our experiments on a 40-core Intel Xeon Gold 6336Y machine with 64 GB of memory running Ubuntu 20.04. For our longer bug-finding fuzzing campaigns, we use a machine with 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 for LLVM's optimizations, we instrument `opt` with SanitizerCoverage [33]. 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).

B. Path Coverage Statistics

To answer evaluation question, **RQ1**. 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.

Due to the exponential nature of storing every permutation of paths, true path coverage is difficult to measure efficiently. As a result, most fuzzers implement a proxy for path coverage—to be precise, industry-standard fuzzers, like AFL++ [34], [35], simulate path coverage by maintaining coarse branch-taken hit counts. SanitizerCoverage, takes a similar approach using `inline-8bit-counters`, which is a byte map that records the hit counts of each unique `(branch_src, branch_dst)` pair in the instrumented code. SanitizerCoverage performs logarithmic bucketing on its edge counts, which places ranges of hit counts in one of eight buckets. Hence, to estimate path coverage, we use SanitizerCoverage's edge counters to instrument `opt`, and we remove all other instrumentation.

We stress that this path coverage estimate is an underapproximation of actual path coverage. It is not able to differentiate between executions that traverse the same set of edges, the same number of times, but in a different order. However, executions that discover a new number of traversals through the same code will be captured to an extent. For instance, the edge counters we use will be able to capture new iterations of a loop's execution if the number of iterations lands in a new bucket.

We conduct a path coverage ablation study by targeting the `-O3` optimization pipeline[3] with each of FLUX's crossover components on the entire LLVM unit test suite and compare the results with Csmith and YARPGen. Since YARPGen can generate both C and C++, we test both languages separately to account for any distributional differences between C and C++. We prepare the unit test suite by collecting around 50,000 LLVM IR functions as seed inputs. 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 of FLUX's configurations, CSmith, YARPGen-c, and YARPGen-cpp until 10,000 new test cases are generated. We compile the C and C++ test cases generated by CSmith and Yarpgen into LLVM-IR to test `opt`. We disable all optimizations with the `-O0` flag for this lowering. Table II shows our results after averaging over five separate runs.

TABLE II
Path coverage increase over the baseline unit test corpus. * denotes $P < 10^{-8}$ when compared with Csmith

Fuzzer	Path Cov. Features	New Features	% Increase
Inline + Sequence	1235204.8*	63183.8	+5.39
Inline	1230825.8*	58804.8	+5.02
Sequence	1231369.6*	59348.6	+5.06
Csmith [9]	1198194.8	26173.8	+2.23
YARPGen-c [10]	**1245884.8***	**73863.8**	**+6.30**
YARPGen-cpp [10]	1238450.0*	66429.0	+5.67
Baseline	1172021.0		

Table II lists the number of path coverage "features" that are achieved by the baseline seed corpus combined with 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. Since edge counts are logarithmically bucketed, if the execution count of an edge maps to a bucket value that has not been previously explored, a new feature ID will be generated from this coverage data. Whenever a new feature ID is encountered, libFuzzer stores it in a set of all unique features. Every fuzzer iteration compares the feature IDs of the newly generated test program with all previously found unique feature IDs. Hence, the "Path Cov. Features" that we report in the second and third columns of Table II are the counts of all unique feature IDs of the fuzzing corpus.

Compared to the baseline, all fuzzers were able to improve upon the path coverage of the entire unit test suite. Further, each of FLUX's and YARPGen's configurations showed statistically significant improvement over CSmith. Focusing just on FLUX's inlining and sequencing mutations, we observe a complementary effect from using both mutations in conjunction, compared to separately. This indicates that the two mutations explore distinct paths through LLVM's optimizations.

[3]We choose `-O3` as it is the most aggressive standard optimization pipeline, containing the most transformation passes.

However, among all fuzzing runs, we find that YARPGen-c and YARPGen-cpp achieve the highest gain in path coverage. We note that our path coverage proxy may unfairly weight generators that explore the same edges a large number of times. YARPGen's generation policies randomly and deliberately skew the distributions of its generation parameters such that certain optimizations are targeted more frequently. For instance, one of YARPGen's policies targets arithmetic peephole optimizations by randomly selecting regions of code to only contain arithmetic instructions. This targeted generation can lead to an increased frequency of executing the same optimizations multiple times in the same test case. These repetitive sub-paths through the optimizer are captured as new coverage by our logarithmic edge count metric, which may explain why YARPGen performs so well in this experiment when compared to CSmith and FLUX, which are less well-tuned to target specific optimizations.

Given that FLUX focuses on finding interesting ways to combine existing unit tests and largely preserves the existing structure of its component tests, a test generated by FLUX is less likely to execute the same optimization code with a new number of hits unless by chance FLUX selects two identical unit tests to perform its crossover. Despite this, even though YARPGen outperforms FLUX in path coverage measurements, FLUX's inlining + sequencing is able to remain competitive.

We note that although YARPGen is able to find more paths, these new paths may not be interesting. For example, consider a code snippet that triggers a peephole optimization. If we generate a test case that contains multiple copies of this snippet, the peephole optimization will run multiple times, but may not result in any interesting bug-finding exploration.

Because of the difficulty in determining whether repetitive executions of the same sub-paths are actually interesting, we perform a second experiment that measures unique edge coverage. To show that our crossovers succeed in discovering interesting paths, we measure the unique edges discovered by our fuzzer-generated tests over that of the baseline unit test suite. We pass the same 10,000 test case corpora into a build of `opt` that is instrumented for edge coverage. Our results are listed in Table III

TABLE III
EDGE COVERAGE INCREASE OVER THE BASELINE UNIT TEST CORPUS. * DENOTES $P < 10^{-8}$ WHEN COMPARED WITH CSMITH AND YARPGEN

Fuzzer	Total Edges	New Edges	% increase
Inline + Sequence	211479.4*	3175.4	+1.52
Inline	**211757.8***	**3453.8**	**+1.66**
Sequence	211262.2*	2958.2	+1.42
Csmith [9]	209453.8	1149.8	+0.55
YARPGen-c [10]	209452.4	1148.4	+0.55
YARPGen-cpp [10]	209363.8	1059.8	+0.51
Baseline	208304.0		

In contrast to the path coverage results, all of FLUX's crossover configurations show significant improvement in finding new edges over the baseline unit tests, Csmith and YARPGen. The disparity between YARPGen's high path coverage improvement and low edge coverage improvement indicates that YARPGen's path coverage gains are a result of repetitive executions of the same edges. This is not to diminish the efficacy of YARPGen's approach, as repeated execution of the same edges may also lead to bug-findings. However, it does show that FLUX is able to find edges that CSmith and YARPGen were not able to find.

Interestingly, these results also indicate that the inline mutation is more adept at exploring new edges than the combination of inline and sequence. This may mean that the rate at which inline explores new edges is greater than that of the sequence crossover. This could be explained by the larger amount of intra-procedural optimization code on the `-O3` pipeline, of which the inline crossover is more suited to explore.

We note that the raw % increase is not particularly informative as there is optimization code that is not enabled under `-O3` and in general, there is no way to simultaneously enable all optimizers in LLVM. Rather, we note that FLUX is able to increase the additional path coverage of Csmith over the Baseline by more than 2$\times$ and the additional edge coverage of CSmith and YARPGen by 3$\times$, demonstrating the effectiveness of our approach.

It should be noted that, to generate 10,000 tests, Csmith's runs took around 5.3 hours and YARPGen's runs only took 5 minutes, while FLUX's mutations ranged from 16 to 33 hours. This is largely due to implementation artifacts. For instance, crashes or found bugs cause fuzzer re-initialization, which requires the entire 50K test suite coverage to be reloaded. Further, since the inlining mutation fails if the two input tests are not type-compatible, many libFuzzer iterations were skipped because the randomly selected inputs were incompatible. We leave the development of a more efficient prototype to future work.

We further acknowledge that the coverage comparisons to CSmith and YARPGen do not paint a complete picture, as we do not evaluate the efficacy of FLUX's bug detection versus CSmith and YARPGen's differential testing approach. Further, we mention that our approach to evaluating YARPGen doesn't take into account its fast generation speed. However, we note that our edge coverage findings in Table III closely match the results reported by YARPGen authors [10].

C. LLVM Crashes

To evaluate our second research question, **RQ2**, we run FLUX for a month 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 22 unique crashes, 19 of which are in the `-O3` pipeline.

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

- *Assertion Failures and Unreachables*[4]. Assertion failures and unreachables in LLVM immediately crash the compiler. We perform our fuzzing runs on a release build of `opt`, with assertions enabled.

[4]Unreachables in LLVM denote points in the code that should not be executed and are similar to Assertion Failures.

TABLE IV
CRASHES FOUND IN THE `-O3` OPTIMIZATION PIPELINE

Optimization	**File**	**Crash Type**
Aggressive-InstCombine	AggressiveInstCombine/ AggressiveInstCombine.cpp	LLVM Error
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	ADT/APInt.h	Assertion
	InstCombine/ InstructionCombining.cpp	Unreachable
IPSCCP	IR/Value.cpp	Assertion
	Utils/SCCPSolver.cpp	Assertion
	IR/Constants.cpp	Assertion
Inliner	CodeGen/BasicTTIImpl.h	Assertion
	Analysis/ConstantFolding.cpp	Unreachable
AlignmentFrom-Assumptions	Support/Alignment.h	Assertion
GVN	Scalar/GVN.cpp	LLVM Error
		Memory Error
LICM	Scalar/LICM.cpp	LLVM Error
SimpleLoop-Unswitch	Scalar/ SimpleLoopUnswitch.cpp	Assertion
SROA	Scalar/SROA.cpp	LLVM Error
	Total Crashes	**19**

- *LLVM Error*. 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*. Crashes caused by use-after-free, out-of-bounds memory accesses, and null pointer dereferences.

Table IV lists the crashes we found with the optimization and file that triggered the crash and the crash type. We note that we are able to find a crashing input test in a wide range LLVM's optimization components, despite only targeting the general `-O3` optimization pipeline. This result contributes some evidence towards the generality of FLUX's approach. One of the LLVM crashes has been confirmed and patched so far.

TABLE V
CRASHES FOUND WITH ONLY THE `-INSTCOMBINE` FLAG

File	**Crash Type**
IR/Instructions.cpp	Assertion
InstCombine/InstCombineLoadStoreAlloca.cpp	LLVM Error
InstCombine/InstCombineMulDivRem.cpp	LLVM Error

To test FLUX under a different optimization configuration, we perform a narrowed fuzzing campaign that only targets LLVM's instruction combine optimizations. Further, we prepare a seed corpus comprised entirely of LLVM unit tests that target the `instcombine` pass. Recall that the unit tests that serve as a starting point for FLUX already achieve 90% line coverage, yet FLUX can still discover new bugs using those tests. We run FLUX with these configurations for 2 days and are able to find 3 unique crashes (Table V). This result provides encouraging feedback for FLUX's ability to find bugs in contexts other than the `-O3` pipeline.

```
1  define i64 @f(ptr %arg, i8 %b) {
2    %g1 = getelementptr i8, ptr %arg, i64 1
3    %ld0 = load i8, ptr %arg, align 1
4    %ld1 = load i8, ptr %g1, align 1
5    %z0 = zext i8 %ld0 to i64
6    %z1 = zext i8 %ld1 to i64
7    %z6 = zext i8 %b to i64
8    %s0 = shl i64 %z0, %z6
9    %s1 = shl i64 %z1, 8
10   %o7 = or i64 %s0, %s1
11   ret i64 %o7
12 }
13 =>
14 define i64 @f(ptr %arg, i8 %b) {
15   %ld0 = load i16, ptr %arg, align 1
16   %1 = zext i16 %ld0 to i64
17   %2 = shl i64 %1, %z6
18   %z6 = zext i8 %b to i64
19   ret i64 %2
20 }
```

Fig. 3. An LLVM Error caused by the AggressiveInstCombine pass

AggressiveInstCombine LLVM error case study: To illustrate what an LLVM error found by FLUX looks like, we examine a crash in the aggressive instruction combine pass that has been verified and patched by LLVM developers. The bug-triggering test case is provided in figure 3. The bug was discovered when fuzzing the `-O3` pipeline.

The above bug-triggering code on lines 1 to 12 was originally much larger when it was found by FLUX however, it has been reduced by LLVM developers after this bug was confirmed and fixed. The program was generated by FLUX with a number of crossovers on unit tests that target the SLPVectorizer transformation[5]. Lines 14 to 20 show the optimized code after being passed through the aggressive instruction combine pass.

Notice that there are 2 sequences of `load`, zero extension (`zext`), and shift left (`shl`) in the pre-optimized code. Because pointers `%arg` and `%g1` point to consecutive addresses, these two sequences of instructions can be widened to a single sequence of width `i16`. The aggressive instruction combine pass detects this and performs multiple instruction fold optimizations on the two load sequences in the test program.

However, due to the data dependencies injected by FLUX's crossovers, the intertwined results of each load sequence causes the optimization to perform an error-producing transform. Notice that in the unoptimized code, a third `zext` instruction on line 7 is used as an operand in the `shl` instruction on line 8. The widened sequence of instructions in the optimized code on lines 15-18 neglects the order of this dependency. This results in the use of `%z6` on line 17, before the variable has been defined on line 18! Hence, after the optimization completes, the LLVM IR verifier detects this use-before-definition error and crashes.

[5]SLPVectorizer/X86/bad-reduction.ll

This demonstrates that FLUX's mutations are able to take unit tests that target one optimization and use them to explore new paths in other passes of the optimizer.

D. Miscompilation Bugs

To evaluate **RQ3**, we investigate whether FLUX is able to generate input programs that are miscompiled and detectable with Alive2. At the time of writing this paper, we were able to debug and verify 6 miscompilation bugs. We have reported these 6 miscompilations to LLVM developers, of which 5 have already been patched. We present two case studies to illustrate.

```
1  define i32 @bar(i32 %ih) {
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(i32 %h) {
13 %0:
14     %sd1 = udiv i32 %h, 2
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 %sd1, i32 %r.i
19     ret i32 %r
20 }
```

Fig. 4. A miscompilation caused by the CorrelatedValuePropagation pass

CVP miscompilation case study: Figure 4 lists the error causing input on lines 1 to 10 and the misoptimized output on 12 to 20. The input test was generated by inlining a function in an InstCombine unit test into itself[6]. The input unit tests check a simple peephole replacement however, our inlining mutation exposes an optimization opportunity for the CorrelatedValuePropagation (CVP) pass. CVP determines it is safe to replace `sdiv` with `udiv` on line 3.

Unfortunately, this optimization produces incorrect code[7]. Note that `udiv`, unsigned division, fills the most significant bits of its result with zeros which does not match the functionality of `sdiv`, the signed division instruction.

Consider passing an undefined value in the inputs of both the pre and post-optimized functions. An undefined value of type T can take any value in the set of defined values for T and can differ across executions. Consider a case where `%sd = 0x00000000` and `%sd1 = 0x40000000` (i.e. a large negative number). This leads to different `icmp` values on lines 4 and 15 respectively in each function, hence one `%t` is assigned 0 and the other a value of 1. Consequently, the `select` instructions on lines 7 and 18, having been fed different booleans, will return different values. The pre-optimized code returns the value of `%r.i` which traces its provenance to the constant 1, whereas the optimized code returns the value of `%sd1` which is derived from the input argument `%h`.

[6]InstCombine/preserve-sminmax.ll
[7]https://github.com/llvm/llvm-project/issues/62200

```
1  define <2 x i32> @f(<2 x i1> %x, <2 x i32> %y) {
2  %0:
3    %sext.i1 = sext <2 x i1> %x to <2 x i32>
4    %r.i2 = xor <2 x i32> { 42, 4294967289 }, %sext.i1
5    %r.i = xor <2 x i32> { 42, 4294967289 }, %r.i2
6    %r = urem <2 x i32> %y, %r.i
7    ret <2 x i32> %r
8  }
9  =>
10 define <2 x i32> @f(<2 x i1> %x, <2 x i32> %y) {
11 %0:
12   %1 = icmp eq <2 x i32> %y, { 4294967295, 4294967295 ↩
         }
13   %r = select <2 x i1> %1, <2 x i32> { 0, 0 }, <2 x ↩
         i32> %y
14   ret <2 x i32> %r
15 }
```

Fig. 5. A miscompilation caused by the InstCombine pass

InstCombine miscompilation case study: Figure 5 lists another miscompilation bug that was caught with a FLUX-generated test case. Lines 1 to 8 list the pre-optimized input program and lines 10 to 15 show the optimized output. The test case was generated with functions that were extracted from an InstSimplify unit test[8] and an InstCombine unit test[9]. The two unit tests check for peephole replacements on vectors of booleans. Both tests check a replacement involving extended vectors (`sext`), as the dividend of an unsigned remainder instruction (`urem`) and as an operand in an exclusive or operation (`xor`), for InstSimplify and InstCombine respectively.

The combination of the two test cases triggers optimizations in the InstCombine pass that replaces the sequence of `sext`, `xor`, and `urem` instructions with an `icmp` instruction followed by a `select`. This replacement does not produce the same behavior under all possible inputs[10].

Consider passing input vectors of `%x = < 1, 1 >` and `%y = < poison, undef >` into the two functions. A poison value is a way to represent undefined behavior (UB) in LLVM IR. Poison values propagate through instructions and raises UB if the poison reaches a side-effect producing instruction.

Notice that the optimized function has two uses of the input argument `%y`. Recall that an undefined value can resolve to different values during runtime. We will demonstrate that passing in an undefined value as an element of `%y` can cause divergent results during runtime. If we trace the flow of these two arguments through the pre-optimized code, we observe that the `sext` instruction results in `%sext.i1 = < -1, -1 >` and consequently `%r.i = < -1, -1 >` due to the cancelling of the two `xor` instructions. Next, assuming that the `undef` value resolves to -1 (i.e. 4294967295 unsigned). The return value of this function will be `%r = < poison, 0 >`. However, if we trace the same input through the optimized

[8]InstSimplify/rem.ll
[9]InstCombine/binop-cast.ll
[10]https://github.com/llvm/llvm-project/issues/62401

code, and assume that undef resolves to any value that is not -1 on this first invocation, we see that `%1 = < poison, 0 >`. If the second use of `undef` on line 13 resolves to -1, then we have a return value of `%r = < poison, -1 >`. Hence the two functions can produce different outputs.

VI. Discussion

A. Generalisability

We acknowledge that much of FLUX's design relies intimately on LLVM specifics. The current FLUX prototype can only test the LLVM compiler. Despite this, we believe that using a high-coverage unit test suite with our sequencing and inlining mutations can be effective in other contexts. For instance, this may also work with GCC's intermediate representations. We leave this exploration to future work.

B. Threats to validity

There are several threats to the validity of this study. First is our choice of proxy measurement for path coverage. Although edge counts and edge coverage are commonly used, it is difficult to assess how their use as an approximation for path coverage affects our conclusions. We believe ultimately, the increases in these values rather than just code coverage, contribute to FLUX's ability to find new bugs. A final note is that not all of the bugs found by FLUX have been confirmed and fixed by LLVM developers yet. However, we are currently working towards this.

VII. Conclusion

This paper contributes a targeted bug-finding method that explores LLVM'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 able to trigger optimization paths is a challenge, one that previous work has mitigated with heuristics. FLUX 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 contribute two novel crossovers, the sequencing and inlining mutations. We demonstrate the efficacy of FLUX by conducting fuzzing runs on LLVM's middle-end optimization pipeline. We find that both of FLUX's crossovers explore more new path coverage than existing work. Further, when used in extended fuzzing runs, FLUX is able to find 22 crashes in LLVM. The variety in the types and compiler optimizations of these crashes support the general applicability of FLUX's fuzzing approach. Finally, FLUX is also able to find 6 compiler miscompilations by using existing translation validation tools to validate our generated tests.

VIII. Data Availability

We have made the code for our FLUX prototype available at https://github.com/ericliuu/flux. Our repository does not include the input unit tests used in our experiments. Those are taken directly from the LLVM repository at https://github.com/llvm/llvm-project/tree/main/llvm/test/Transforms, at commit 0303eafcb34647f7d5a4015aad266b5766f5dc5e. Because we have only tested on the x86 backend, we exclude any unit tests that target other architectures. Further, we trivially filter out incompatible unit tests by running each test through `opt` with the `-O3` optimization level and excluding test cases that crash. We store each unit test in a flat directory that is passed in as seed input into libFuzzer. In total, this results in nearly 50,000 LLVM IR functions in our corpus.

IX. Acknowledgements

We would like to thank our anonymous reviewers for their insightful feedback. We also thank the LLVM community for their help in confirming and fixing our bug findings. Financial support for this work was provided in part by an Ontario Graduate Scholarship, NSERC Discovery Grant RGPIN-2018-05931 and a Tier 1 Canada Research Chair.

References

[1] LLVM enjoyed record growth in 2021, many exciting compiler advancements. Accessed: Mar. 9, 2023. [Online]. Available: https://www.phoronix.com/news/LLVM-Record-Growth-2021

[2] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in GCC and LLVM," vol. 174, 2021, pp. 1–13.

[3] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in GCC and LLVM," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 294–305.

[4] T. Scheller. Where is LLVM being used today? Accessed: Mar. 13, 2023. [Online]. Available: https://llvm.org/devmtg/2016-01/slides/fosdem-2016-llvm.pdf

[5] S. Bauer, P. Cuoq, and J. Regehr, "Deniable backdoors using compiler bugs," *International Journal of PoC or GTFO*, vol. 0x08, pp. 7–9, 2015.

[6] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[7] M. Zalewski, "American Fuzzy Lop," accessed: Mar. 13, 2023. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[8] K. Serebryany, "Continuous fuzzing with libFuzzer and AddressSanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*, 2016, pp. 157–157.

[9] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[10] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with YARPGen," *Proceedings of the ACM on Programming Languages*, vol. 4, pp. 196:1–196:25, 2020.

[11] E. Nagai, A. Hashimoto, and N. Ishiura, "Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions," *IPSJ Transactions on System LSI Design Methodology*, vol. 7, no. 0, pp. 91–100, 2014.

[12] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda, "Random testing of c compilers targeting arithmetic optimization," in *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies*, 2012, pp. 48–53.

[13] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 95–105.

[14] X. Liu, X. Li, R. Prajapati, and D. Wu, "DeepFuzz: Automatic generation of syntax valid c programs for fuzz testing," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 1, pp. 1044–1051, 2019.

[15] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[16] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 386–399, 2015.

[17] C. Sun, V. Le, and Z. 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*, 2016, pp. 849–863.

[18] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 347–361.

[19] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[20] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron, "Mythical unit test coverage," *IEEE Software*, vol. 35, no. 3, pp. 73–79, 2018.

[21] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: bounded translation validation for LLVM," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 65–79.

[22] H. Jiang, Z. Zhou, Z. Ren, J. Zhang, and X. Li, "CTOS: Compiler testing for optimization sequences of LLVM," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2339–2358, 2022.

[23] J. Chen and C. Suo, "Boosting compiler testing via compiler optimization exploration," *ACM Transactions on Software Engineering and Methodology*, p. 72:1–72:33, 2022.

[24] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys*, vol. 53, no. 1, pp. 4:1–4:36, 2020.

[25] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, 1998.

[26] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 78–88.

[27] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012, pp. 445–458.

[28] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr, "Provably correct peephole optimizations with alive," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 22–32.

[29] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined behavior: what happened to my code?" in *Proceedings of the Asia-Pacific Workshop on Systems*, 2012, pp. 1–7.

[30] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 260–275.

[31] T. Kelly, W. Gu, and V. Maksimovski, "Schrödinger's code: Undefined behavior in theory and practice," *Queue*, vol. 19, no. 2, 2021.

[32] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," *2012 USENIX Annual Technical Conference*, p. 309–318, 2012.

[33] SanitizerCoverage — clang 17.0.0git documentation. Accessed: Mar. 15, 2023. [Online]. Available: https://clang.llvm.org/docs/SanitizerCoverage.html

[34] The AFL++ fuzzing framework. Accessed: Mar. 25, 2023. [Online]. Available: https://aflplus.plus/

[35] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," *14th USENIX Workshop on Offensive Technologies*, 2020.