IRFuzzer: Specialized Fuzzing for LLVM Backend Code Generation

Yuyang Rong*† PeterRong96@gmail.com
Zhanghan Yu† hnryu@ucdavis.edu
Zhenkai Weng† zweng@ucdavis.edu
Stephen Neuendorffer* stephen.neuendorffer@amd.com
Hao Chen† chen@ucdavis.edu

*Advanced Micro Devices, Inc.
†University of California, Davis

***Abstract*—Modern compilers, such as LLVM, are complex. Due to their complexity, manual testing is unlikely to suffice, yet formal verification is difficult to scale. End-to-end fuzzing can be used, but it has difficulties in discovering LLVM backend problems for two reasons. First, frontend preprocessing and middle optimization shield the backend from seeing diverse inputs. Second, branch coverage cannot provide effective feedback as LLVM backend contains much reusable code.**

In this paper, we implement IRFuzzer to investigate the need of specialized fuzzing of the LLVM compiler backend. We focus on two approaches to improve the fuzzer: guaranteed input validity using constrained mutations to improve input diversity and new metrics to improve feedback quality. The mutator in IRFuzzer can generate a wide range of LLVM IR inputs, including structured control flow, vector types, and function definitions. The system instruments coding patterns in the compiler to monitor the execution status of instruction selection. The instrumentation not only provides new coverage feedback on the matcher table but also guides the mutator on architecture-specific intrinsics.

We ran IRFuzzer on 29 mature LLVM backend targets. IRFuzzer discovered 78 new, confirmed bugs in LLVM upstream, none of which existing fuzzers could discover. This demonstrates that IRFuzzer is far more effective than existing fuzzers. Upon receiving our bug report, the developers have fixed 57 bugs and back-ported five fixes to LLVM 15, which shows that specialized fuzzing provides actionable insights to LLVM developers.

***Index Terms*—fuzzing, LLVM, software analysis**

I. Introduction

Modern compilers, such as LLVM [1], are complex software. For example, LLVM consists of over seven million lines of C/C++ code contributed by more than 2500 developers. Given the size of this codebase and its importance in the computing ecosystem, an effective, scalable verification method is critical. Despite extensive testing, latent bugs remain and their impact on users can be quite significant given the widespread distribution and long lifetime of compilers.

To reduce latent bugs, various techniques have been used to automate the verification of compilers, such as partial model checking [2], fuzzing [3–5], and differential testing [6, 7]. Although end-to-end formal verification of compilers has been applied [8, 9], these techniques have not yet scaled to practical compilers such as LLVM, which supports a wide range of architectures, programming languages, and use models.

In the specific case of LLVM, another factor making verification difficult is that the interface between compiler optimization and machine code generation is widely used but not completely specified. As a result, it can be difficult for backend developers to understand whether they have completely implemented the wide range of possible inputs. In addition, backends often differ greatly in their relative code maturity, including some targets that are relatively mature and other targets for new devices that are in active development.

We find that the state-of-the-art fuzzers failed to find new bugs of a compiler backend for various reasons. General-purpose fuzzing techniques, such as AFL++ [10], often do not consider input validity and struggle to explore control paths in the compiler backend since most binary strings are invalid compiler inputs. In order to test the compiler backend more effectively, we aim to generate LLVM Intermediate Representation (LLVM IR) that complies with the language reference. LLVM includes `llvm-opt-fuzzer` and `llvm-isel-fuzzer`, which generate valid IR for middle end and backend fuzzing, respectively [11]. Both of them are based on the library FuzzMutate [12] for valid IR mutation. However, FuzzMutate can't construct complex control flows and only generates a few instructions with scalar types. On the other hand, end-to-end fuzzing tools, such as CSmith [4] and GrayC [13], test the whole pipeline of the compiler, but they cannot explore control paths in the compiler backend efficiently. CSmith does not take any feedback from the compiler, which contributes to its ineffectiveness. A more fundamental reason is that front-end parser and middle-end optimizations may limit the set of features seen by the compiler backend. High level languages, such as C, may not exercise all backend features in LLVM. Therefore, even if GrayC used branch coverage feedback from libFuzzer [14], it missed many backend bugs introduced before LLVM 12, which were found by us. As a result, when a new language, such as Rust, is introduced, new backend bugs may still arise [15].

Generating valid IR is challenging with three major difficulties. In order to generate a complex control flow graph (CFG), we have to maintain all data dependencies to avoid use-before-definition situations. A valid CFG can be easily invalidated by a jump, as shown in Figure 2. This challenge does not exist in C generation as long as one does not generate `goto` statements. Besides, modelling the instructions missing in FuzzMutate isn't trivial. We must make sure that the types of the operands in each IR instruction match, but enumerating the large numbers of natively supported vector

types is infeasible. Finally, it is difficult to model intrinsic functions for all architectures, as intrinsics are often poorly documented and vary from architecture to architecture.

We also observe that AFL++'s feedback mechanism performed poorly when testing the backend. It uses branch coverage as feedback, which runs into severe branch collision problems when fuzzing large code bases such as LLVM. Naively increasing the branch counting table size introduces huge overhead [16]. A more fundamental reason is that much code generation logic in the LLVM backend is implemented using table-driven state machines. A matcher table encapsulates all possible states as a constant byte array, meaning that branch counting can't observe this logic during fuzzing. The fuzzer needs a better feedback on whether the seed is interesting or not. If the seed is not interesting, the feedback should also inform the mutator what type of input is desired.

To address these issues, we design a specialized fuzzer, IRFuzzer, for fuzzing the LLVM compiler backend. Figure 1 shows the overall structure of IRFuzzer. We first design a mutator that generates valid IR (Section III-A). We maintain the domination relation in a CFG during mutation by inserting subgraphs (*s*CFG) into the existing CFG as shown in Figure 2c. We also use a descriptive language to list the requirements of each instruction type. This approach ensures that inputs to the compiler backend are *always* valid, increasing the efficiency of fuzzing. Our work expands FuzzMutate to include special handling by compiler backends, such as multiple basic blocks with complex control flow, function calls, intrinsic functions, and vector types. Using IRFuzzer, we are able to generate a wider range of instructions and explore control paths in the compiler backends more efficiently.

Then, we introduce a new coverage metric (Section III-B) by instrumenting the table-driven state machines in LLVM, enabling the design space to be more efficiently explored. New entries covered in the matcher table indicate that new features are executed. Working together with branch coverage, they provide better feedback on whether a seed is interesting. Furthermore, the matcher table contains all the information about the instructions and intrinsics in one architecture. As a result, we use the matcher table to determine which instructions and intrinsics haven't been fuzzed. We design a feedback loop from the matcher table coverage to our mutator. IRFuzzer periodically sends to the mutator a coverage report containing the states that haven't been executed to guide mutations. This allows IRFuzzer to test on different backends with no prior knowledge of the architecture.

We evaluated IRFuzzer on 29 mature backend architectures in LLVM (Section V). Our results show that IRFuzzer is more effective than the state-of-the-art fuzzers AFL++ and GrayC. IRFuzzer generated inputs code with better branch coverage and matcher table coverage on all LLVM backends. Leveraging these techniques, we were able to find and report 78 new bugs in LLVM, of which all have been confirmed, 57 have been fixed, and five have been back ported to LLVM 15. This demonstrates the high impact on improving the correctness of LLVM backend targets.

This paper uses LLVM to demonstrate the importance of having a specialized fuzzer for the compiler backend. Since modern compilers have similar intermediate representations, we expect that our approach can apply to other compilers without requiring heavy engineering efforts. We made the following contributions in this paper:

- We designed and implemented IRFuzzer. To the best of our knowledge, IRFuzzer is the first backend fuzzer that uses matcher table coverage feedback to guide mutation.
- We compared IRFuzzer with other state-of-the-art fuzzers on LLVM upstream and found it to be the most effective on matcher table coverage.
- We carefully analyzed and categorized the bugs we found during our testing. In total, we discovered 78 confirmed new bugs in LLVM, of which 57 have been fixed and five have been back ported to LLVM 15.

II. Background

A. LLVM

LLVM [1] is a mature compiler framework consisting of many components that can be targeted to different architectures. At its core lies the LLVM Intermediate Representation (LLVM IR), which serves as a target-independent abstraction separating the concerns of high-level programming languages from the low-level details of particular architectures. LLVM can be roughly partitioned into three layers. The *frontend*, such as `clang`, translates programming languages to LLVM IR, including lexer, parser, AST transformation, etc. The *middle-end*, called `opt`, processes LLVM IR and performs code analysis and many common target-independent optimizations. The *backend*, called `llc`, converts LLVM IR to a target-specific machine code representation and eventually assembly code for the target architecture. The LLVM backend supports multiple target architectures through a plug-in abstraction, and the code to support a target architecture typically involves the implementation of API functions to describe common aspects along with target specific code to implement more unusual concepts.

The LLVM IR describes a static single-assignment (SSA) form [17], with a fixed set of instructions. Instructions are strongly typed, and the type of each value must match between its definition and all uses. A wide range of types are supported, including integers with arbitrary bitwidth, floating point values, pointers, vectors, and other aggregate types. As with most high-level languages, LLVM IR allows the definition of functions, and the control flow between functions is implemented using the `call` instruction. Architecture specific intrinsics have no corresponding IR instructions, but are represented as function calls at IR level.

Control flow within a function in LLVM IR is represented using basic blocks and branch instructions. Special PHI instructions allow instructions in a basic block to refer to values defined in other basic blocks. Therefore, PHI instructions must respect control flow constraints and may only refer to values defined in predecessor blocks. This *domination constraint* [18]

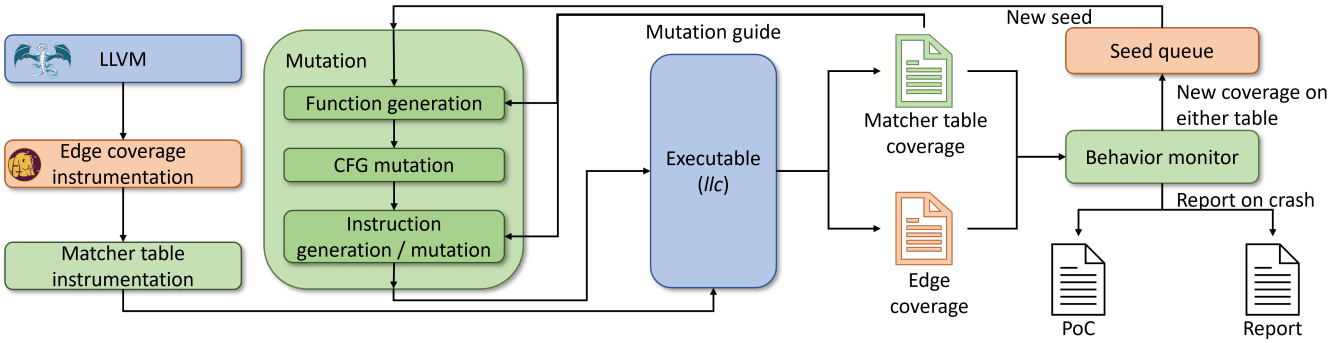
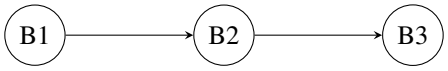
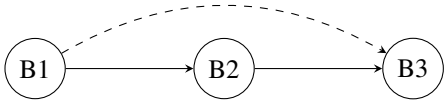


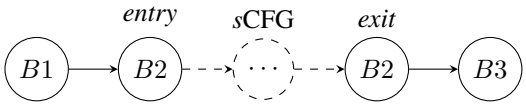
Fig. 1: Overview of IRFuzzer. Green shaded components are the contributions of this paper, orange shaded components are AFL++, and blue shaded components are from LLVM. We created an LLVM IR mutator that guarantees the correctness of the generated input (Section III-A). We introduced a new coverage metric to track the backend code generation while guiding the mutation module (Section III-B).



(a) Original CFG



(b) Incorrect mutation: randomly adding an edge



(c) Correct mutation: splitting node and adding *s*CFG

Fig. 2: Examples of incorrect and correct CFG mutations. Figure 2a is the original linear CFG. Figure 2b naively adds an edge from B1 to B3. After that, B2 no longer dominates B3, so every value defined in B2 and used in B3 may cause use-before-definition error. Figure 2c breaks B2 into an entry and an exit node and inserts an sCFG between them. This guarantees that B2 still dominates B3.

means that techniques used in high-level language generation cannot be easily adapted to LLVM IR.

The process of *instruction selection* in the LLVM backend replaces target-independent LLVM IR instructions with target-specific machine code instructions. LLVM provides two different frameworks to implement instruction selection that may be leveraged by the target backend plug-in. SelectionDAG [19] is the more mature instruction selection framework and is leveraged by all targets. In SelectionDAG, the code in each basic block is converted into a directed acyclic graph (DAG) representing the data dependency between instructions, and instruction selection is performed on the DAG. Since SelectionDAG processes each basic block independently, it can miss opportunities for optimization across basic blocks. GlobalIsel [20] is a newer framework that is only leveraged by some targets. GlobalIsel preserves the basic block structure within a function during instruction selection, enabling more optimization opportunities.

B. Coverage guided fuzzing

American Fuzzy Lop (AFL) [21], an open source fuzzing framework, implements coverage-guided fuzzing. It instruments the program under test (PUT) with the ability to track control-flow coverage. When an input increases code coverage, AFL stores it in a *seed cache* and mutates it to generate new inputs. This strategy allows AFL to explore different control-flow paths of the PUT efficiently.

Many variations of coverage guided fuzzing have been developed, with the goal of finding bugs more efficiently by exploring a wider range of program behaviors with future executions of the PUT [22]. There are studies on the impact of different feedback algorithms [23–25]. Different methods are proposed to prioritize seeds to improve the performance of fuzzing. [26–28]. Some fuzzers target specific bugs [29–31] and libraries [32, 33]. More advanced mutation strategies also show better fuzzing performance compared with random mutation [34–39]. Many improvements have been implemented in AFL++ [10], making it a good framework for further development. Fuzzing not only finds bugs but also helps program understanding [40].

LLVM also introduces its own coverage guided fuzzing framework libFuzzer [14], coupled with FuzzMutate [12], it can be used to fuzz LLVM backend. However, FuzzMutate only generates a limited type of code and is not under active development. Still, the framework provides us with helpful insights into how should we mutate LLVM IR.

III. Design

We design IRFuzzer with two new components. Figure 1 shows the new components of IRFuzzer. During the mutation stage, we first generate a function if there isn't one (Section III-A1). Then we change the control flow graph (CFG) to create more control flows (Section III-A2). Finally, we generate new IR instructions and mutate them (Section III-A3). Af-

```
1  define i64 @f(i32 zeroext %I, <4 x i32> %V) noinline  {
2  Entry:    %ret_p = alloc i64, 1
3            %ret = load i64, ptr %ret_p
4            ret i64 %ret
5  }
```

Listing (1) A piece of LLVM IR program generated by function generation (Section III-A1). The function returns a 64 bit integer, so we allocate a stack memory and load from it to return. We will fill the memory in later mutations.

```
1  define i64 @f(i32 zeroext %I, <4 x i32> %V) noinline {
2  EntrySrc:     %ret_p = alloc i64, 1
3                switch i32 %I, label %sCFG_Default [
4                  i32 1, label %sCFG_1
5                ]
6  sCFG_Default: br label %EntrySink
7  sCFG_1:       br label %EntrySink
8  EntrySink:    %ret = load i64, ptr %ret_p
9                ret i64 %ret
10 }
```

Listing (2) IR program mutated from Listing 1. Line 3 to 7 are introduced by *s*CFG insertion (Section III-A2). We insert *s*CFG by splitting the `Entry` block into two and generating a `switch` instruction.

```
1  define i64 @f(i32 zeroext %I, <4 x i32> %V) noinline {
2  EntrySrc:    %ret_p = alloc i64, 1
3               switch i32 %I, label %sCFG_Default [
4                 i32 1, label %sCFG_1
5               ]
6  sCFG_Default:%I64 = zext i32 %I, i64
7               br label %EntrySink
8  sCFG_1:      %I1 = add i32 %I, 1
9               %J64 = call @f(i32 %I1, <4 x i32> %V)
10              br label %EntrySink
11 EntrySink:   %PHI = phi i64 [%J64, %sCFG_1],
12                             [%I64, %sCFG_Default]
13              store i64 %PHI, %ret_p
14              %ret = load i64, ptr %ret_p
15              ret i64 %ret
16 }
```

Listing (3) IR program mutated from Listing 2. Instruction insertion(Section III-A3) generates line 6, 9, and 12. The placeholder memory is also used by `%PHI` to avoid undefined behavior (Line 13).

Fig. 3: An example of how IRFuzzer mutates a module using different strategies.

ter the mutation stage, we create a new method to measure the coverage of the program (Section III-B). Although IRFuzzer borrows from FuzzMutate and other tools, all the components described in this section are novel unless otherwise noted.

A. LLVM IR mutation

To generate a wide variety of input while avoiding invalid inputs, we adopt a mutation-based strategy. This strategy starts with small valid seed inputs and modifies the seed inputs in ways that should also generate valid inputs. By randomly selecting between a number of small, well-defined mutations, we expect to eventually generate a broad class of valid inputs while avoiding invalid inputs. Figure 3 shows an example of our mutator in practice. We first generate an empty function if none is present (Listing 1). Then, we mutate the control flow by *s*CFG insertion (Listing 2). Finally, we modify or insert instructions in basic blocks (Listing 3).

1) Function generation: The LLVM backend has many target-specific code related to function calls. As a result, it is important to generate a wide range of function definitions and function calls with different arguments and return types.

IRFuzzer implements a mutation strategy capable of generating new function definitions with arbitrary arguments and return types. One important constraint is that the return type of the function signature matches the type of each `return` instruction in the function definition. To ensure this, IRFuzzer synthesizes a load instruction of an appropriate type as the operand for a `return` instruction. Although the value returned from the load may be uninitialized, later mutations may store values to the memory, validating the return value.

IRFuzzer also implements a mutation strategy to generate new `call` instructions that refer to specific function declarations. The mutator is free to select from any declared functions and generate compatible arguments and return values for the call, as with any other primitive instruction. Intrinsic functions are target specific operations that correspond to complicated machine instructions, so generating them will increase the code we can test. Yet they are treated as functions at middle-end. In particular, this mutation strategy will also select intrinsic functions to call.

Function attributes can impact backend behavior. These attributes are often set by the compiler frontend and middle end to optimize the code. For example, `noinline` can change how a function is treated during code generation. LLVM categorizes attributes into function attributes, argument attributes, and return value attributes. To demonstrate IRFuzzer's potential, we include 15 attributes in IRFuzzer. Most attributes do not affect the validity of the program. However, we need to model the contradicting ones to ensure that they do not appear at the same time, such as `OptForFuzzing` and `OptimizeForSize`.

2) CFG mutation: Generating diverse CFGs is necessary to test LLVM backends. Many machine code optimizations restructure control flow. In addition, certain compiler optimizations may select specific jump instructions, but this optimization can only be performed after instruction selection when the code size and alignment are known. For instance, a common compiler optimization is to first select jump instructions in a "short" form with a limited offset range and then only replace the short form with a "long" jump instruction if a larger offset is required.

A challenge in mutating CFGs is how to preserve the dominator constraints. Naively inserting and removing arbitrary edges in a CFG may violate dominator constraints between basic blocks, which may cause use-before-definition. For example, starting from the CFG in Figure 2a, after we add an edge, $B2$ no longer dominates $B3$ in Figure 2b. If $B3$ uses any value defined in $B2$, the program will cause a use-before-definition error if it follows the new edge.

To overcome this problem, IRFuzzer inserts *sub-control flow graphs* (*s*CFG), as shown in Figure 2c, to maintain dominator constraints.

Definition 1. *A sub-control graph (sCFG) is a CFG with a single entry block and a single exit block. The exit block must have a single outgoing edge, and all the other blocks must either branch to another block in this sCFG or* `return`*.*

Definition 2. *A block S dominates a block T if S precedes T on all the reachable paths ending in T.*

Theorem 1. *Let block S dominate block T. Let B be a block, and C be an sCFG. Then, after we split B into B_{entry} and B_{exit} and insert C between them, S still dominates T.*

Proof. Before we insert C, since S dominates T, S precedes T on every reachable path ending in T. Let $p = (\cdots, S, \cdots, T)$ be such a path. After we insert C,

- If no block in C is inserted between S and T on p, then p does not violate the property that S dominates T.
- If some blocks in C are inserted between S and T on p, then p becomes one or more new paths $p' = (\cdots, S, \cdots, B_{\text{entry}}, C_{\text{entry}}, C*, C_{\text{exit}}, B_{\text{exit}}, T)$ where $C*$ represents a sequence of blocks in C. If no block in $C*$ contains a `return`, then S still precedes T on p'. On the other hand, if any block in $C*$ contains a return, then T is not reachable on p'. In both these cases, p' does not violate the property that S dominates T.

□

Theorem 1 guarantees that splitting a block and inserting an sCFG between them preserves dominator constraints. Based on this property, IRFuzzer mutates CFG in the following steps.

1) IRFuzzer selects a block, and a non-terminating instruction inside it as the split point.
2) IRFuzzer splits the block into an *entry* block, containing all the instructions before the split point, and an *exit* block, containing all the remaining instructions. Then, it randomly generates a `branch` or `switch` instruction as the entry block's new terminating instruction.
3) IRFuzzer creates empty blocks as the destinations of the `branch` or `switch` instruction created in the previously step. For each empty block, IRFuzzer randomly selects `branch`, `switch`, or `return` as its terminator.
4) If IRFuzzer selects `branch` or `switch`, it either routes the control flow to the *exit* block or creates a self loop.

3) Instruction modeling and generation: A key aspect of the LLVM backend is to convert the wide range of LLVM IR types to the (usually small) set of types natively implemented by each target architecture. Therefore, to exercise all the features of code generation, it is necessary to generate IR instructions with as many data types as possible. Much of the compiler backend handles IR instructions with vector types, but FuzzMutate modelled only scalar types.

To overcome FuzzMutate's limitations, we rewrite its modelling, as shown in Table I. We not only include vectors as allowed types but also model vector operations and casting operations. These definitions are reflected in the code as declarations expressing both restrictions on the types of operands and constraints between the types of different operands. For example, the `anyIntOrVecInt` constraint restricts the valid types for a particular operand to be any integer type or vector of integer type. This allows us to model vector operations, such as `extractelement`, `insertelement`, and `shufflevector`, which were unsupported by FuzzMutate.

In addition, `store` and `load` memory operations are structured differently enough from other operations that modeling them declaratively is unnecessary. Some other instructions have constraints which are too complex , so we resort to custom generators. For instance, instructions representing PHI nodes must be created with a number of operands equal to the number of predecessor blocks and must occur at the start of their basic block. Similarly, *call* instructions are handled manually, since we must select a function declaration and find values that exactly match the operand types of the declaration.

When generating a new instruction, we first randomly select an opcode and use the declarations to randomly select values that exist in the code with a compatible type. To ensure that values are defined before they are used, the mutator searches for values defined in the following locations: global variables, function arguments, values in dominators, and values defined by previous instructions in the same basic block. If no value with a compatible type exists, then the mutator can either generate a poison or generate a load from a pointer if one exists.

When generating instructions, the mutator may allocate new stack memories as placeholders. To avoid undefined behaviors, the mutator will again attempt to replace loads from these placeholders with other values of a compatible type. If no such value exists, then the mutator will store a value into the placeholder location.

We model no intrinsic functions, as they vary from architecture to architecture, potentially consuming much time with little outcome. Instead, we rely on the feedback from matcher table coverage (Section III-B2), which shows the intrinsics that haven't been generated yet. The mutator will then randomly generate *call* instructions to those intrinsics.

4) Instruction shuffling: Changing instruction orders inside a basic block will change how the backend schedules instructions. When shuffling instructions, we must carefully handle instruction orders; otherwise, a use-before-definition may arise. We use topological sort to ensure that for each define-use relation, define precedes use after instruction shuffling.

B. Matcher table feedback

1) Matcher table instrumentation: LLVM uses *patterns* to describe rewrite rules applied during instruction selection. Some simple patterns replace a single LLVM IR instruction with a single machine instruction. More complex patterns may replace multiple LLVM IR instructions or generate multiple machine instructions. Patterns may also apply in specific situations by including complex predicates. For example, a pattern may only apply when a particular operand is a constant, or a certain hardware feature is enabled.

Most patterns are described declaratively in an LLVM-specific language called TableGen [41]. To optimize the ap-

TABLE I: Extended instruction modeling for IR instructions. Note that FuzzMutate only implements binary and bitwise operations with no vector support.

Operation type	Opcode		Argument descriptions		
Unary operation	fneg	:	anyFloatPointOrVectorFloatPoint		
Binary operations	add, sub, mul, (s|u)(div|rem)	:	anyIntOrVecInt	sameAsFirst	
	fadd, fsub, fmul, fdiv, frem	:	anyFPOrVecFP	sameAsFirst	
Bitwise operations	shl, lshr, ashr, and, or, xor	:	anyIntOrVecInt	sameAsFirst	
Vector operations	extractelement	:	anyVector	anyInt	
	insertelement	:	anyVector	matchScalarOfFirst	anyInt
	shufflevector	:	anyVector	matchLengthOfFirst	VecOfConstI32
Aggregate operations	extractvalue	:	anyAggregateOrArray	anyConstInt	
	insertvalue	:	anyAggregateOrArray	matchScalarOfFirst	anyConstInt
Memory operation	getelementptr	:	anySized	pointerOfFirst	anyInt
Casting operations	trunc	:	anyNonBoolIntOrVecInt	anyIntOrVecIntWithLowerPrecision	
	zext, sext	:	anyIntOrVecInt	anyIntOrVecIntWithHigherPrecision	
	fptrunc	:	anyNonHalfFPOrVecFP	andFPOrVecFPWHigherPrecision	
	fptoui, fptosi	:	anyFPOrVecFP	matchLengthOfFirstWithInt	
	uitofp, sitofp	:	anyIntOrVecInt	matchLengthOfFirstWithFP	
	ptrtoint	:	anyPtrOrVecPtr	matchLengthOfFirstWithInt	
	ptrtoint	:	anyIntOrVecInt	matchLengthOfFirstWithPtr	
	bitcast	:	anyType	anyTypeWithSameBitWidth	
Other operations	icmp	:	anyIntOrVecInt	sameAsFirst	
	fcmp	:	anyFPOrVecFP	sameAsFirst	
	select	:	anyBoolOrVecBool	matchLengthOfFirst	sameAsSecond

plication of patterns, TableGen represents patterns in a state-machine and implements it as a large byte array known as the *matcher table*. During compilation, the state machine determines the best pattern to apply to each IR instruction. Listing 4 is a C++ code snippet for evaluating the matcher table in SelectionDAG. `SDNode` is a data structure that represents an IR instruction. The while loop iteratively reads a command from the matcher table based on the current state, represented by the `idx` variable, evaluates the command, and selects the next state to be evaluated. For example, `Opc_CheckOpcode` will check if the opcode of a given `SDNode` representing an instruction in the SelectionDAG graph matches a particular opcode. The `Result` will be used in future iterations, depending on the next entry in the matcher table. The compiler continues to evaluate the matcher table until it selects a single pattern or reaches a state where no pattern applies.

Since the program in Listing 4 evaluates all the patterns using the same set of conditional branches in the switch statement, its control flow coverage does not reflect what patterns have been exercised. To overcome this, we track the usage of the matcher table directly.

Similar to how AFL tracks branch coverage, we allocate a table, *matcher table coverage table*, for tracking the coverage of the matcher table. Each entry in this table corresponds to an entry in the matcher table and records if the latter has been accessed. The instrumented compiler dumps matcher table coverage after every execution. If either the branch coverage table and or the matcher table coverage table shows new coverage, then the fuzzer considers the input as new.

Tracking matcher table coverage incurs memory overhead, which may reduce fuzzing throughput [23]. The second and

```
1  void SelectCodeCommon(SDNode *N, char *MatcherTable) {
2   bool Result = true;
3   unsigned Opc;
4   while (true) {
5    if (!Result) break;
6    switch (MatcherTable[Idx++]){
7      case OPC_CheckOpcode: {
8       uint16_t Opc = MatcherTable[Idx++];
9       Opc |= (unsigned short) MatcherTable[Idx++] << 8;
10      Result = (Opc == N->getOpcode());
11     }
12     ...
13     case OPC_MoveChild0: {
14       unsigned ChildNo = Opc - OPC_MoveChild0;
15       if (ChildNo >= N.getNumOperands())
16         break;  // Match fails if out of range child #.
17       N = N.getOperand(ChildNo);
18       NodeStack.push_back(N);
19       continue;
20     }
21   }
22  }
23 }
24 void AArch64SelectionDAG::SelectCode(SDNode *N){
25  #define TARGET_VAL(X) X & 255, unsigned(X) >> 8
26  static const unsigned char MatcherTable[] = {
27  ...
28  /*25929*/OPC_CheckOpcode,TARGET_VAL(ISD::ADD),
29  /*25932*/OPC_MoveChild0,
30  /*25933*/OPC_CheckOpcode,TARGET_VAL(AArch64ISD::UMULL),
31  /*25936*/OPC_MoveChild0,
32  ...
33  };
34  SelectCodeCommon(N, MatcherTable,sizeof(MatcherTable));
35 }
```

Listing 4: SelectionDAG in LLVM that consumes a matcher table to do instruction selection. We also show AArch64's matcher table from index 25929 to 25936. Switch case `OPC_MoveChild0` can be executed with different `Opc`, rendering branch coverage ineffective to track the behavior of this code. Therefore, we also track individual entries of the matcher table.

TABLE II: The number of entries in the matcher tables used by SelectionDAG in mature architectures (LLVM commit `860e439f`). To track the coverage of the matcher table, we use one bit to track each entry in the matcher table.

Arch	# of entries	Arch	# of entries
AArch64	489 789	PowerPC	190 304
AMDGPU	493 556	RISC-V	2 191 899
ARM	201 172	SystemZ	53 271
Hexagon	178 277	VE	71 577
Mips	54 044	WASM	25 991
NVPTX	186 134	X86	680 916

fourth column of Table II show the size of the matcher table in different mature architectures. The matcher tables for mainstream architectures, such as X86 and AArch64, have several hundred thousand entries, whereas RISC-V has about two million entries. Since the entries in the matcher table represent different features, to determine which features have been covered, we can individually track whether each entry has been accessed because the order of access is irrelavent. To reduce memory footprint, we use one byte to track eight entries in the matcher table. For example, the largest matcher table, of the RISC-V architecture, has 2 191 899 entries. Tracking its coverage takes $\lceil 2191899/8 \rceil$ bytes, or 274 kB.

2) IR mutation feedback: The mutator needs to know which patterns in the matcher table have been executed so that it can generate more diverse inputs. However, when LLVM prepares the matcher table, it hides which pattern each entry in the matcher table represents.

To recover this information, we generate a look-up table to map each matcher table entry to its corresponding machine instruction pattern. Compiler developers program different patterns into TableGen, and the compiler translates those patterns into the matcher table. We modify TableGen to reverse that process to create the look-up table.

Prior to fuzzing, we create this look-up table for each architecture. During fuzzing, we use the matcher table coverage table and the look-up table to determine which patterns haven't been generated. Finally, we send this report to the mutator to encourage it to generate those patterns, which is done every ten minutes to avoid excessive runtime overhead.

IV. Implementation

Our implementation is based on prior work FuzzMutate[12] and AFL++ [10]. Compared with FuzzMutate, we added the following new mutation strategies, which have been incorporated into the upstream LLVM's repository:

- A new function template generator with the ability to modify function attributes.
- A new control flow graph mutation strategy, *s*CFG insertion strategy, which modifies the control flow while preserving domination relations.
- Extended modelling of IR instructions, including PHI nodes, memory operations, vector operations, and support for non-scalar types.

Compared with AFL++, we measure the coverage of the matcher table, which not only helps determine if a new input is interesting but also guides mutation. This feedback allows our mutator to generate architecture specific intrinsics without any prior knowledge of the architecture.

V. Evaluation

We evaluated IRFuzzer by fuzzing LLVM with different settings and tools to answer the following research questions:

- **RQ1**: How does IRFuzzer compare with state-of-the-art backend fuzzers?
- **RQ2**: How does IRFuzzer compare with end-to-end fuzzers like CSmith and GrayC?
- **RQ3**: Do mutator and matcher table feedback individually contribute to IRFuzzer?
- **RQ4**: Can IRFuzzer find new bugs in LLVM?

The upstream LLVM repository (commit `860e439f`) currently supports 21 architectures. We only tested on mature architectures that had a matcher table size larger than 25 000, as shown in Table II. In addition, each architecture may provide different features that can be enabled on different hardware. For simplicity, we selected the backends of some popular microchips, which had a predefined set of features. These backends were widely used from user product to server applications, justifying the variety of our choice. All the architectures that we tested were under active development. As a result, we selected 29 target CPUs[1] across 12 architectures.

We used two baseline fuzzers: (1) AFL++ with no modification, and (2) AFL++ whose mutation module was replaced with FuzzMutate, referred to as FuzzMutate thereafter. All fuzzers used AFL++'s default scheduling. For fairness, we collected the seeds generated by each fuzzer and measured their branch coverage and matcher table coverage. AFL++ reported branch coverage using classical instrumentation and a default 64 kB table.

We prepared two versions of IRFuzzer:

- IRFuzzer has all the mechanisms described in Section III.
- IRFuzzer$_{\text{bare}}$ excludes the feedback mechanism described in Section III-B. Its performance reveals the contribution of our mutator when compared with FuzzMutate, and of the feedback mechanism when compared with IRFuzzer.

Each fuzzer process ran exclusively on a single processor core on an x86_64 server. Each fuzzing process ran for one day to allow adequate exploration [42]. We repeated each experiment five times to average the results to reduce random effects. To demonstrate IRFuzzer's ability to mutate IR modules and to provide a fair comparison with AFL++, we initialized each fuzzer process with 92 seeds. We randomly selected the seeds from LLVM's unit tests. Each seed was smaller than 256 bytes to increase the throughput. We anonymously published the seeds in the artifact [43].

[1] "Target CPU" was used in LLVM to label a backend corresponding to a microchip. It can also refer to GPU, DSP or virtual targets like WebAssembly.

A. Baseline comparison

We compared our mutation strategy with two baseline implementations: AFL++ and the upstream LLVM implementation of FuzzMutate. AFL++ lacks an LLVM IR-aware mutator, whereas FuzzMutate has a limited LLVM IR-aware mutator.

Table III shows the branch and matcher table coverages, which we calculated by dividing the number of non-empty entries in the coverage table by the size of the table. The *seeds* columns show the coverage brought by the initial seeds. On each target CPU, Target CPU, IRFuzzer and IRFuzzer$_{\text{bare}}$ achieved more coverage than the baseline fuzzers, and the difference is statistically significant ($p < 0.05$).

IRFuzzer achieved the highest branch coverage on all the target CPUs. It may seem counterintuitive that AFL++ has higher branch coverage than FuzzMutate on most target CPUs. Our investigation revealed that AFL++'s high branch coverage mostly comes from error handling code since it can hardly generate valid input. This is further demonstrated by AFL++'s low matcher table coverage, which indicates that most executions did not reach the instruction selection stage before the compiler terminated.

It is insufficient to compare only branch coverage [16]. More significantly, IRFuzzer achieved the best matcher table coverage on all CPUs, indicating significantly better coverage of instruction selection patterns.

Comparison between FuzzMutate and AFL++ also cast insights on which fuzzer is better to fuzz the backend compiler. FuzzMutate can generate valid input to reach deeply nested code more easily, as demonstrated by its higher matcher table coverage in Table III compared with AFL++. On the other hand, AFL++'s high branch coverage and low matcher table coverage show that most inputs didn't reach the instruction selection stage before the compiler terminated. Therefore, AFL++ is useful mainly for testing error handling and the frontend.

In summary, IRFuzzer achieved higher branch coverage and matcher table coverage on all target CPUs compared with AFL++ and FuzzMutate. To answer **RQ1**, IRFuzzer is better in coverage when fuzzing LLVM code generation compared with state-of-the-art fuzzers.

B. Comparison with end-to-end fuzzers

To better understand the benefits of targeted fuzzing over end-to-end fuzzing, we evaluated CSmith [4] and GrayC [13]. Unlike IRFuzzer, end-to-end fuzzers generate C code, which must be processed by the compiler frontend and middle-end before reaching the backend. As a result, they exercise the entire compilation pipeline, rather than focusing on just the backend. Note that although CSmith generates random, syntactically correct C code, it does not implement any instrumentation and lacks feedback to guide the generation process. While GrayC relies on branch coverage feedback, it does not have feedback that is customized for the backends of the compilers. Besides, to test end-to-end fuzzers, we had to cross compile C to different architectures, which was difficult and time-comsuming. Therefore, we tested on three most widely used architectures using generic backend.

CSmith generates C files with no initial seed. To make the comparison fair, we also ran IRFuzzer with **no** initial seed, since IRFuzzer is capable of generating LLVM IR from scratch. GrayC relies on deprecated APIs in LLVM 12 and cannot instrument the latest LLVM. So we download the artifact provided by GrayC [44]. The artifact consists of 715 147 C programs across ten trials. We ran CSmith for 24 hours and repeated it eight times, generating a total of 506 971 C programs.

We cross-compiled these C programs to different architectures. After compilation, we measured the resulting branch and matcher table coverage in the compiler backend, using the same instrumentation as IRFuzzer. We only tested on `O2` and `O3`, as `O0` and `O1` are often subsets of `O2`. The results are shown in Table IV.

IRFuzzer achieved the highest matcher table and branch coverage on all the architectures and all the optimizations. Even with branch coverage feedback, GrayC was unable to generate C inputs with more matcher table coverage, which demonstrating the need for specialized backend fuzzing. We looked into the code generated by end-to-end fuzzers and found that their low matcher table coverage was mainly because they could not handle vector data types. Vector instructions are generated only when the front-end and middle-end decide that a vector instruction will speed up a particular piece of code, which is uncommon in random C programs generated by end-to-end fuzzers. In comparison, since IRFuzzer operates directly on IR instructions, it can generate vector operations easily.

To answers **RQ2**: IRFuzzer achieved higher matcher table coverage than state-of-the-art end-to-end fuzzers. This shows that compiler backend testing should not solely rely on end-to-end fuzzing, and that specialized fuzzing can improve matcher table coverage significantly.

C. Individual contributions

To evaluate how each component of IRFuzzer helps, we stripped all the feedbacks in IRFuzzer to get IRFuzzer$_{\text{bare}}$.

Table III shows that IRFuzzer$_{\text{bare}}$ *always* reached higher branch coverage and matcher table coverage than FuzzMutate, indicating that our mutator was able to generate more diverse inputs. Although FuzzMutate is also a structured mutator, it lacks many advanced features that we designed in Section III-A. The sifive-x280 CPU best demonstrates this improvement, where IRFuzzer$_{\text{bare}}$ covered 3.42% of the matcher table while FuzzMutate covered only 0.31%.

The last two columns of Table III show that IRFuzzer is able to cover more matcher table in 28 out of 29 target CPUs compared with IRFuzzer$_{\text{bare}}$. This demonstrates that our matcher table feedback can help the mutator during fuzzing trials. This effect can be best observed on NVPTX, where IRFuzzer$_{\text{bare}}$ covered only 6.3% of the matcher table while IRFuzzer covered 26.9%.

TABLE III: Branch table coverage and matcher table coverage on 29 target CPUs across 12 targets in SelectionDAG. Statistics are the arithmetic mean over five trials. Bold entries are the best among baseline fuzzers. FM means AFL++ coupled with FuzzMutate, IRF means IRFuzzer, IRF_{bare} means IRFuzzer without matcher table feedback.

Arch	Target CPU	Branch coverage					Matcher table coverage				
		Seeds	AFL++	FM	IRF_{bare}	IRF	Seeds	AFL++	FM	IRF_{bare}	IRF
AArch64	apple-a16	59.8%	87.1%	82.9%	95.2%	**96.9%**	0.7%	1.6%	2.6%	7.5%	**8.9%**
	apple-m2	59.8%	86.9%	83.3%	94.9%	**97.0%**	0.7%	1.6%	2.6%	7.6%	**9.2%**
	cortex-a715	60.0%	87.7%	83.2%	94.9%	**96.9%**	0.7%	1.7%	2.6%	7.4%	**10.9%**
	cortex-r82	60.1%	87.0%	82.9%	95.2%	**96.7%**	0.7%	1.6%	2.6%	7.3%	**8.8%**
	cortex-x3	60.0%	93.3%	85.2%	96.6%	**96.8%**	0.7%	7.1%	2.7%	7.9%	**10.5%**
	exynos-m5	60.3%	87.4%	83.2%	**96.5%**	96.2%	0.7%	1.7%	2.6%	7.9%	**8.5%**
	tsv110	60.0%	87.3%	82.9%	**95.9%**	95.7%	0.7%	1.6%	2.6%	7.7%	**8.2%**
AMDGPU	gfx1036	70.8%	90.0%	89.1%	96.2%	**97.0%**	0.9%	2.1%	2.7%	4.3%	**5.1%**
	gfx1100	71.2%	89.7%	89.9%	96.6%	**96.8%**	1.0%	2.1%	2.9%	4.4%	**4.9%**
ARM	generic	55.5%	87.9%	82.5%	88.6%	**91.6%**	1.7%	4.3%	4.3%	4.3%	**5.4%**
Hexagon	hexagonv71t	64.8%	88.0%	86.0%	93.2%	**94.8%**	1.7%	6.6%	17.0%	21.6%	**33.2%**
	hexagonv73	64.9%	89.5%	85.7%	93.0%	**94.7%**	1.7%	7.3%	17.4%	20.7%	**32.5%**
Mips	mips64r6	52.5%	81.0%	72.7%	**87.0%**	84.8%	3.8%	10.0%	15.3%	**18.4%**	18.3%
NVPTX	sm_90	46.6%	77.5%	77.5%	90.6%	**91.3%**	1.7%	3.1%	4.7%	6.3%	**26.9%**
PowerPC	pwr9	60.3%	87.3%	86.9%	95.6%	**95.9%**	1.2%	3.6%	7.1%	19.0%	**23.6%**
RISC-V	rocket-rv64	53.7%	83.0%	76.6%	87.1%	**88.3%**	0.12%	0.20%	0.22%	0.23%	**0.23%**
	sifive-u74	54.5%	83.1%	75.9%	**88.3%**	88.2%	0.14%	0.24%	0.29%	0.31%	**0.32%**
	sifive-x280	55.0%	84.1%	75.7%	90.7%	**92.0%**	0.14%	0.27%	0.31%	3.42%	**3.70%**
SystemZ	z15	55.3%	84.0%	81.5%	93.7%	93.8%	5.2%	13.7%	27.1%	43.9%	**50.6%**
	z16	55.3%	83.7%	81.8%	93.3%	**93.7%**	5.2%	14.1%	26.5%	43.7%	**50.2%**
VE	generic	49.0%	80.4%	70.2%	**89.6%**	89.0%	3.5%	8.1%	11.4%	13.0%	**14.1%**
WASM	bleeding-edge	46.8%	84.7%	70.5%	88.8%	**90.0%**	4.1%	36.9%	10.9%	40.2%	**41.5%**
	generic	46.6%	80.2%	69.7%	87.4%	**88.4%**	4.1%	11.8%	10.6%	12.0%	**12.4%**
X86	alderlake	61.2%	88.0%	84.6%	96.3%	**97.2%**	0.7%	1.8%	3.1%	7.1%	**9.3%**
	emeraldrapids	60.5%	93.4%	84.4%	96.2%	**97.5%**	0.6%	12.5%	3.2%	14.8%	**18.9%**
	raptorlake	61.2%	93.5%	85.8%	96.8%	**97.2%**	0.7%	6.2%	3.3%	7.4%	**9.4%**
	sapphirerapids	60.5%	88.4%	85.4%	96.7%	**97.4%**	0.6%	1.8%	3.3%	15.3%	**19.1%**
	znver3	61.8%	86.6%	84.0%	96.5%	**97.4%**	0.7%	1.6%	3.0%	7.3%	**9.3%**
	znver4	61.0%	87.6%	84.0%	96.3%	**97.5%**	0.7%	1.8%	3.2%	14.4%	**17.7%**

TABLE IV: Average branch table coverage and matcher table coverage of CSmith (CS), GrayC, and IRFuzzer (IRF). `O2` and `O3` are different optimization levels. Bold entries are the winners.

	Arch	Branch table coverage			Matcher table coverage		
		CS	GrayC	IRF	CS	GrayC	IRF
`O2`	AArch64	94.8%	96.1%	**96.7%**	5.2%	6.9%	**8.9%**
	ARM	90.7%	92.3%	**92.5%**	4.5%	4.5%	**5.4%**
	X86	94.8%	96.1%	**96.9%**	3.5%	4.2%	**5.9%**
`O3`	AArch64	95.3%	96.2%	**96.9%**	5.4%	6.9%	**8.9%**
	ARM	91.1%	92.5%	**92.5%**	4.5%	4.5%	**5.4%**
	X86	94.9%	96.2%	**96.8%**	3.5%	4.2%	**5.9%**

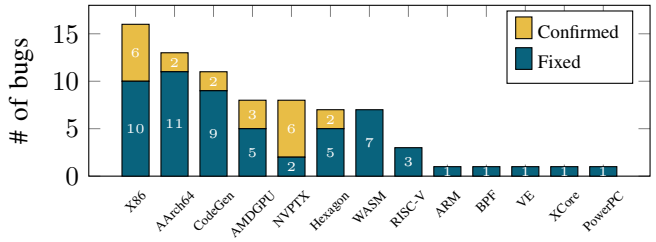
In rare cases IRFuzzer has lower branch coverage than $\text{IRFuzzer}_{\text{bare}}$. This is because the feedback mechanism incurs a tradeoff. Calculating matcher table coverage and sending it to the mutator reduce the throughput, which lowers branch coverage. On the other hand, this feedback is valuable for generating more diverse inputs, which contributes to higher matcher table coverage. Among all the 29 target CPUs, IRFuzzer had lower branch coverage than $\text{IRFuzzer}_{\text{bare}}$ on only 5 target CPUs, so we believe that the tradeoff is acceptable and justified. Besides, both IRFuzzer and $\text{IRFuzzer}_{\text{bare}}$ outperformed baseline fuzzers on all the target CPUs. We can answer **RQ3** confidently that both the mutator and feedback mechanism contributed to improved matcher table coverage.

D. Bug categories and analysis

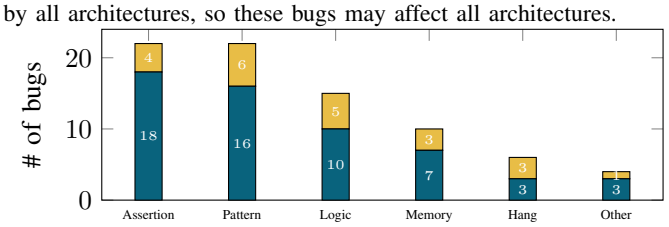
We collected all the crashes found in Section V-A and Section V-B. We also fuzzed other architectures with no features to extend our scope. Since GlobalIsel also uses matcher table design, we can apply IRFuzzer on it with little modification. This demonstrates that our approach can be generalized to other frameworks with little effort.

In the process, we found hundreds of crashes in the LLVM compiler. Even though these crashes all have unique stack traces, they do not necessarily indicate different bugs because some crashes have different paths but the same root cause. Therefore, we manually analyzed all of them and reported the ones that we believe are bugs. In this section, we only report the bugs that have been *confirmed*. In total, IRFuzzer found 78 confirmed bugs. We manually verified that these bugs are found only by IRFuzzer and published the details anonymously [43].

These bugs are distributed in different places in the LLVM codebase. Figure 4a shows the distribution of these bugs across



(a) Bugs categorized by **locations**. CodeGen refers to the code shared by all architectures, so these bugs may affect all architectures.



(b) Bugs categorized by **causes**. Most of the severe bugs are compiler hangs, memory errors, and assertion failures.

Fig. 4: Distributions of bugs found by IRFuzzer. IRFuzzer has found **78** new bugs, out of which **57** have been fixed.

```
1 bool IRTranslator::translateExtractElement(
2 const User &U, MachineIRBuilder &MIRBuilder) {
3   Register Idx;
4   const LLT Ty = LLT::scalar(PreferredVecIdxWidth);
5   Idx = MIRBuilder.buildSExtOrTrunc(Ty, Idx).getReg(0);
6 }
```

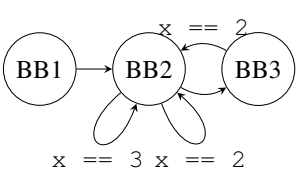
Listing 5: A snippet of code in LLVM where the index of a vector is treated as a signed value.

LLVM. CodeGen is the library shared between architectures, meaning that a bug in CodeGen may affect all architectures.

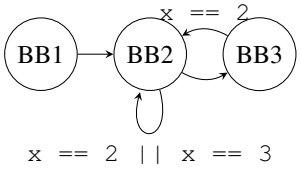
1) Bugs found by baseline fuzzers: Our evaluation of the baseline fuzzer — AFL++, FuzzMutate, CSmith, and GrayC — shows that none of them found any backend bugs. All the 78 confirmed bugs were found exclusively by IRFuzzer. AFL++ found many crashes in the `llc` lexer and module verifier. However, all of them were caused by a malformed input and are not considered bugs. FuzzMutate did not find any crashes because its mutator is very limited and only covers common use cases of the compiler backend.

2) Distribution of bugs: We categorize these bugs into six categories: hang, memory errors, assertion failures, logic errors, missing patterns, and other bugs. Hang, memory errors and assertion failures are the most severe because they stall compilation. A missing pattern bug occurs when a certain machine instruction is permitted by the hardware specification but no matching instruction selection pattern exists. Logic errors and missing patterns do not stall compilation but may generate ineffective or even wrong machine instructions. Figure 4b shows the number of bugs in each category. Assertion bugs are the most common. They arise from the developers' false assumption that some properties hold during compilation, which our fuzzer disapproved.

We demonstrate two bugs found by IRFuzzer. Listing 5 shows a bug in IR Translator. When translating the IR instruction `extractelement`, the bug extends the index as



(a) Original CFG



(b) Optimized CFG

Fig. 5: A piece of code generated by IRFuzzer, simplified to CFG only. `TurnSwitchRangeIntoICmp` transforms Figure 5a into Figure 5b, and `FoldValueComparisonIntoPredecessors` will undo the transformation, causing an infinite loop.

a signed integer, e.g., translating `char 255` into `-1`. This bug generates incorrect machine instructions and affects the LLVM backend for seven architectures. Introduced in LLVM nine years ago, the bug was never noticed for several reasons. First, it is less common for compiler frontends to generate vector operations, as we have seen in Section V-B, and is even rarer to use an index that is large enough to wrap around to a negative integer. However, IRFuzzer can generate such a test case easily because it directly mutates on IR instruction. More importantly, the documentation was ambiguous with respect to the desirable behavior. The documentation states "The index may be a variable of any integer type" [45] without giving details on how it should be interpreted. Therefore, when this bug was introduced, it complied with the incomplete documentation at the time. This exemplifies how complex software interfaces can be incompletely specified, which further justifies our specialized fuzzing. In this case, we fixed the bug and updated the documentation to reflect the intended interpretation of the index as an unsigned integer.

IRFuzzer also found compile hangs. Figure 5 shows a simplified CFG corresponding to the code generated by IRFuzzer. This CFG will cause a compiler hang due to the interaction between two optimization passes. BB2 in Figure 5a consists of a `switch` statement with two self loop edges. The `TurnSwitchRangeIntoICmp` optimization attempts to rewrite the condition as a branch predicate because `x == 2 || x == 3` can be optimized using bit operations, rewriting Figure 5a into Figure 5b. However, the `FoldValueComparisonIntoPredecessors` optimization converts this code back into a `switch` statement to reduce the number of comparison operations, turning the CFG back to Figure 5a. As a result, a fixed point is never reached, creating an infinite loop. This bug is hard to trigger since the bug can only be triggered when the `switch` in Figure 5b has exactly two destinations (BB2 and BB3), and the switch conditions are consecutive, enabling the `TurnSwitchRangeIntoICmp` optimization. This combination is unlikely to be created during manual testing, and can only happen through the interaction of two largely unrelated pieces of code. Yet, we are able to discover this catastrophic combination through our CFG mutation strategy in a time frame amenable to run fuzzing on every nightly build with little human intervention.

We are working closely with the LLVM community to fix the bugs discovered by IRFuzzer. 57 bugs were fixed, five of which were back ported to LLVM 15 as security patches. The developers confirmed that all the bugs that we reported and they fixed had been introduced prior to LLVM 15. Despite heaving testing, they remained in LLVM 15 until IRFuzzer discovered them. This demonstrates that specialized fuzzing for compiler backend is necessary, and it provides actionable insight to developers.

3) Accuracy: IRFuzzer guarantees to generate valid LLVM IR. Since IR is the input to backends, a robust backend should take any valid IR without crashing, so any crash indicates either a bug or an incomplete feature in the backend. IRFuzzer found 78 bugs, all of which we confirmed and reported to the developers. Of these bugs, 57 have been fixed, and five bugs were back-ported as security patches. This shows that the developers agree that these are true bugs regardless of whether C programs corresponding to the IR exist.

We can answer **RQ4** now. In total, IRFuzzer found 78 new bugs. All have been confirmed, 57 have been fixed, and five have been back ported to LLVM 15 as security patches. All these bugs were found only by IRFuzzer and not by any of the baseline fuzzers. These bugs contain six compiler hangs, ten memory errors, and 22 assertion failures. We also found 15 logic errors and 22 missing patterns in the matcher table.

VI. Related work

Prior work has focused on compiler testing [46–48]. One popular approach is to generate inputs for compilers to compile. Purdom[49] generates program based on context free grammar. Superion[50] and Nautilus[51] also relies on context free grammar for fuzzing. However, context free grammar based methods cannot generate semantically meaningful programs. These efforts are effective in testing frontend parsers, but cannot reach the backends effectively.

While many fuzzers are testing the frontend of the compiler using grammar based method [52], some work also tests the correctness of middle-end [2, 53–55]. To the best of our knowledge, IRFuzzer is the first one to verify the compiler backend using an architecture independent approach.

Some work does end-to-end tests using high-level programming languages. CSmith [4], YARPGen [5], and Grayc [13] generate C and C++ programs. AI has also been used for program generate for the purpose of compiler testing [56–58]. However, end-to-end testing implies that there is a need to create a generator for every language, like JavaScript [59], Rust [60], and Java [61–63]. POLYGLOT[64] introduced a language-free IR and mutator based on it. Most fuzzers have no feedback from the compiler. Even though Grayc [13] introduced branch coverage feedback, it was unable to trigger backend bugs due to language limitations and compiler optimizations. Instead of directly generating a program, Equivalence Modulo Inputs [6, 65] mutates an existing C program to preserve its semantics. Therefore, the program before and after mutation should have the same behavior. Combining CSmith and EMI, Lidbury et al. mutate program to test OpenCL compiler [66]. However, the language limits these work, since the generator cannot help when the language frontend cannot exercise a feature in the compiler.

Formal verification is another valuable part of compiler verification [67]. Verasco [8] is a formally verified C analyzer. CompCert [68] is a compiler for a subset of C that is formally verified. There is work that verifies other languages, like Rust [69] and Lustre [9]. However, formal verification cannot scale to large compilers like LLVM, therefore it has a limited impact in the community.

There is also work that considered generating a valid intermediate representation for testing purposes. FuzzMutate directly generates LLVM IR [12]. However, FuzzMutate has no feedback unless combined with fuzzers like AFL++ [10] or libFuzzer [14]. Some work focus on testing of a specific compiler [70, 71]. Tzer focuses on IR mutation in the context of a tensor compiler [71]. However, Tzer relies on LLVM's Coverage Sanitizer that only tracks code coverage. Similar to IRFuzzer's approach, ClassMing directly mutates on Java byte code [72]. Neither Tzer nor ClassMing designed a feedback approach, except for branch coverage. However, as we demonstrate in Section V-C, a customized feedback metric can greatly help the fuzzer to reach deeper into the code base. With the development of large language models (LLM), it has been used more and more in fuzzing and code generation [73–76]. However, LLM doesn't guarantee the correctness of input like IRFuzzer does.

VII. Conclusion

We described IRFuzzer, a fuzzer specializing in fuzzing LLVM instruction selection. To generate semantically and syntactically correct inputs, we identified the challenges in IR generation that did not exist in high-level language generation. We created a mutator that maintained semantic correctness by splitting blocks and inserting a *s*CFG in between. Then, we ensured that the IR instructions that we inserted were syntactically correct using a descriptive language to model all IR instructions. Therefore, the IR program that IRFuzzer generated could always be compiled by the backend. We proposed a new metric to track the coverage of the matcher table and decoded the coverage table to guide mutation.

Our evaluation shows that IRFuzzer outperformed existing backend and end-to-end state-of-the-art fuzzers. IRFuzzer achieved higher matcher table coverage on all the LLVM backend architectures. IRFuzzer is also efficient enough to become part of the development process.

IRFuzzer identified 78 new, confirmed bugs in upstream LLVM code. Upon receiving our bug report, the developers have fixed 57 bugs and back-ported five fixes to LLVM 15. This demonstrates that IRFuzzer is effective in finding bugs in LLVM backend and provides useful, actionable insights to LLVM developers. Our experience shows that there are fertile opportunities for specialized fuzzing despite popular end-to-end compiler testing.

Acknowlegement

This work is partially supported by UC Noyce Initiative.

References

[1] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". In: *CGO*. 2004.

[2] Nuno P. Lopes et al. "Alive2: Bounded Translation Validation for LLVM". In: *PLDI*. 2021.

[3] Andrea Fioraldi et al. "LibAFL: A Framework to Build Modular and Reusable Fuzzers". In: *Conference on Computer and Communications Security (CCS)*. 2022.

[4] Xuejun Yang et al. "Finding and Understanding Bugs in C Compilers". In: *PLDI*. 2011.

[5] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. "Random Testing for C and C++ Compilers with YARPGen". In: OOPSLA (2020).

[6] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler Validation via Equivalence modulo Inputs". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI. Edinburgh, United Kingdom, 2014.

[7] Qirun Zhang, Chengnian Sun, and Zhendong Su. "Skeletal Program Enumeration for Rigorous Compiler Testing". In: *PLDI*. 2017.

[8] Jacques-Henri Jourdan et al. "A Formally-Verified C Static Analyzer". In: *POPL*. 2015.

[9] Timothy Bourke et al. "A formally verified compiler for Lustre". In: *PLDI*. 2017.

[10] Andrea Fioraldi et al. "AFL++ : Combining Incremental Steps of Fuzzing Research". In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. 2020.

[11] *Fuzzing LLVM libraries and tools*. https://llvm.org/docs/FuzzingLLVM.html. [Online; accessed 15-Mar-2024].

[12] Justin Bogner. *Adventures in Fuzzing Instruction Selection*. https://llvm.org/devmtg/2017-03/assets/slides/adventures_in_fuzzing_instruction_selection.pdf. [Online; accessed 15-Mar-2024]. Mar. 2017.

[13] Karine Even-Mendoza et al. "GrayC: Greybox Fuzzing of Compilers and Analysers for C". In: ISSTA. 2023.

[14] Kosta Serebryany. In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016.

[15] Seo Sanghyeon. *Rust triggers LLVM ARM backend bug*. https://github.com/rust-lang/rust/issues/9117. [Online; accessed 15-Mar-2024]. 2013.

[16] Shuitao Gan et al. "CollAFL: Path Sensitive Fuzzing". In: *Security and Privacy*. 2018.

[17] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *POPL*. 1988.

[18] Reese T Prosser. "Applications of boolean matrices to the analysis of flow diagrams". In: *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. 1959.

[19] *The LLVM Target-Independent Code Generator*. https://llvm.org/docs/CodeGenerator.html. [Online; accessed 15-Mar-2024].

[20] *GlobalIsel*. https://llvm.org/docs/GlobalISel/index.html. [Online; accessed 15-Mar-2024].

[21] *American fuzzy lop*. URL: http://lcamtuf.coredump.cx/afl/.

[22] Xiaogang Zhu et al. "Fuzzing: A Survey for Roadmap". In: *ACM Comput. Surv.* (2022).

[23] Jinghan Wang et al. "Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 2019.

[24] Jinghan Wang, Chengyu Song, and Heng Yin. "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing". In: (2021).

[25] Cornelius Aschermann et al. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *Security and Privacy*. 2020.

[26] M. Böhme, V. Pham, and A. Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *IEEE Transactions on Software Engineering* 45.5 (2019), pp. 489–506. DOI: 10.1109/TSE.2017.2785841.

[27] Chenyang Lyu et al. "MOPT: Optimized Mutation Scheduling for Fuzzers". In: *Security and Privacy*. 2019.

[28] Dongdong She, Abhishek Shah, and Suman Jana. "Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis". In: *Security and Privacy*. 2022.

[29] Sebastian Österlund et al. "ParmeSan: Sanitizer-guided Greybox Fuzzing". In: *USENIX Security*. 2020.

[30] Yuyang Rong, Peng Chen, and Hao Chen. "Integrity: Finding Integer Errors by Targeted Fuzzing". In: *Security and Privacy in Communication Networks (SecureComm)*. Springer. 2020.

[31] Dae R. Jeong et al. "Razzer: Finding Kernel Race Bugs through Fuzzing". In: *Security and Privacy*. 2019.

[32] Peng Chen et al. "HOPPER: Interpretative Fuzzing for Libraries". In: *ACM Conference on Computer and Communications Security (CCS)*. Copenhagen, Denmark, 2023.

[33] Yunlong Lyu et al. "Prompt Fuzzing for Fuzz Driver Generation". In: *ACM Conference on Computer and Communications Security (CCS)*. Salt Lake City, UT, USA, 2024.

[34] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *NDSS*. 2019.

[35] Dongdong She et al. "NEUZZ: Efficient fuzzing with neural program smoothing". In: *Security and Privacy*. IEEE. 2019.

[36] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *Security and Privacy*. 2018.

[37] Peng Chen, Jianzhong Liu, and Hao Chen. "Matryoshka: Fuzzing Deeply Nested Branches". In: *ACM Conference on Computer and Communications Security (CCS)*. 2019.

[38] Mingyuan Wu et al. "One Fuzzing Strategy to Rule Them All". In: *ICSE*. 2022.

[39] Yuyang Rong et al. “Valkyrie: Improving Fuzzing Performance Through Deterministic Techniques”. In: *International Conference on Software Quality, Reliability, and Security (QRS)*. 2022.

[40] Jianyu Zhao et al. “Understanding Programs by Exploiting (Fuzzing) Test Cases”. In: *Findings of the Association for Computational Linguistics (ACL)*. 2023.

[41] *TableGen Overview*. https://llvm.org/docs/TableGen/. [Online; accessed 15-Mar-2024].

[42] George Klees et al. “Evaluating Fuzz Testing”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada, 2018, pp. 2123–2138.

[43] Anonymous. *IRFuzzer artifacts*. Sept. 2023. DOI: 10.5281/zenodo.13139630.

[44] Karine Even-Mendoza et al. *Artifact of GrayC*. Version GrayC-ISSTA-2023-V1.0. July 2023. DOI: 10.5281/zenodo.7978251.

[45] Peter Rong. *Using ZExt for extractelement indices*. https://reviews.llvm.org/D132978. [Online; accessed 15-Mar-2024]. 2022.

[46] Junjie Chen et al. “A Survey of Compiler Testing”. In: *ACM Comput. Surv.* (2020).

[47] Haoyang Ma. *A Survey of Modern Compiler Fuzzing*. 2023. arXiv: 2306.06884 [cs.SE].

[48] Michaël Marcozzi et al. “Compiler Fuzzing: How Much Does It Matter?” In: OOPSLA (2019).

[49] Paul Purdom. “A sentence generator for testing parsers”. In: *BIT Numerical Mathematics* 12.3 (1972), pp. 366–375.

[50] Junjie Wang et al. “Superion: Grammar-Aware Greybox Fuzzing”. In: *ICSE*. 2019, pp. 724–735.

[51] Cornelius Aschermann et al. “NAUTILUS: Fishing for Deep Bugs with Grammars.” In: *NDSS*. 2019.

[52] Andreas Zeller et al. *The fuzzing book*. 2019.

[53] William Mansky and Elsa Gunter. “A framework for formal verification of compiler optimizations”. In: *Interactive Theorem Proving: First International Conference (ITP)*. 2010.

[54] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. “Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages”. In: PLDI (2023).

[55] Nuno P. Lopes et al. “Practical Verification of Peephole Optimizations with Alive”. In: *Communications of the ACM* 61.2 (2018), pp. 84–91.

[56] Xiao Liu et al. “DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing”. In: *AAAI* (2019).

[57] Yinlin Deng et al. “Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models”. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023.

[58] Chunqiu Steven Xia et al. *Universal Fuzzing via Large Language Models*. 2023. arXiv: 2308.04748 [cs.SE].

[59] Soyeon Park et al. “Fuzzing JavaScript Engines with Aspect-preserving Mutation”. In: *Security and Privacy*. 2020.

[60] Kyle Dewey, Jared Roesch, and Ben Hardekopf. “Fuzzing the Rust Typechecker Using CLP (T)”. In: *ASE*. 2015.

[61] Yuting Chen et al. “Coverage-Directed Differential Testing of JVM Implementations”. In: *PLDI*. 2016.

[62] Emin Gün Sirer and Brian N. Bershad. “Using Production Grammars in Software Testing”. In: *Proceedings of the 2nd Conference on Domain-Specific Languages*. DSL. 2000.

[63] Mingyuan Wu et al. “JITfuzz: Coverage-Guided Fuzzing for JVM Just-in-Time Compilers”. In: *ICSE*. 2023.

[64] Yongheng Chen et al. “One Engine to Fuzz ’em All: Generic Language Processor Testing with Semantic Validation”. In: *Security and Privacy*. 2021.

[65] Vu Le, Chengnian Sun, and Zhendong Su. “Finding Deep Compiler Bugs via Guided Stochastic Program Mutation”. In: OOPSLA. 2015.

[66] Christopher Lidbury et al. “Many-Core Compiler Fuzzing”. In: *PLDI*. 2015.

[67] Maulik A. Dave. “Compiler Verification: A Bibliography”. In: *SIGSOFT Softw. Eng. Notes* 28.6 (2003).

[68] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Commun. ACM* 52.7 (2009).

[69] Vytautas Astrauskas et al. “Leveraging Rust Types for Modular Specification and Verification”. In: 3.OOPSLA (2019).

[70] Haoyang Ma et al. “Fuzzing Deep Learning Compilers with HirGen”. In: ISSTA. 2023.

[71] Jiawei Liu et al. “Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation”. In: OOPSLA (2022).

[72] Yuting Chen, Ting Su, and Zhendong Su. “Deep Differential Testing of JVM Implementations”. In: *ICSE*. 2019.

[73] Hongxiang Zhang et al. *LLAMAFUZZ: Large Language Model Enhanced Greybox Fuzzing*. 2024. arXiv: 2406.07714 [cs.CR].

[74] Chunqiu Steven Xia et al. “Fuzz4All: Universal Fuzzing with Large Language Models”. In: *ICSE*. 2024.

[75] Chenyuan Yang et al. “WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models”. In: OOPSLA (2024).

[76] Yifeng He et al. “UniTSyn: A Large-Scale Dataset Capable of Enhancing the Prowess of Large Language Models for Program Testing”. In: ISSTA. 2024.