Fuzzing MLIR Compiler Infrastructure via Operation Dependency Analysis

Chenyao Suo
College of Intelligence and Computing, Tianjin University
Tianjin, China
chenyaosuo@tju.edu.cn

Junjie Chen[*]
College of Intelligence and Computing, Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Shuang Liu
School of Information, Renmin University of China
Beijing, China
shuang.liu@ruc.edu.cn

Jiajun Jiang
College of Intelligence and Computing, Tianjin University
Tianjin, China
jiangjiajun@tju.edu.cn

Yingquan Zhao
College of Intelligence and Computing, Tianjin University
Tianjin, China
zhaoyingquan@tju.edu.cn

Jianrong Wang
College of Intelligence and Computing, Tianjin University
Tianjin, China
wjr@tju.edu.cn

Abstract

MLIR (Multi-Level Intermediate Representation) compiler infrastructure has gained widespread popularity in recent years. It introduces dialects to accommodate various levels of abstraction within the representation. Due to its fundamental role in compiler construction, it is critical to ensure its correctness. Recently, a grammar-based fuzzing technique (i.e., MLIRSmith) has been proposed for it and achieves notable effectiveness. However, MLIRSmith generates test programs in a random manner, which restricts the exploration of the input space, thereby limiting the overall fuzzing effectiveness. In this work, we propose a novel fuzzing technique, called MLIRod. As complicated or uncommon data/control dependencies among various operations are often helpful to trigger MLIR bugs, it constructs the operation dependency graph for an MLIR program and defines the associated operation dependency coverage to guide the fuzzing process. To drive the fuzzing process towards increasing operation dependency coverage, MLIRod then designs a set of dependency-targeted mutation rules. By applying MLIRod to the latest revisions of the MLIR compiler infrastructure, it detected 63 previously unknown bugs, among which 38/48 bugs have been fixed/confirmed by developers.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Compiler Fuzzing, MLIR Compiler Infrastructure, Test Program Generation

[*]Junjie Chen is the corresponding author.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680360

ACM Reference Format:
Chenyao Suo, Junjie Chen, Shuang Liu, Jiajun Jiang, Yingquan Zhao, and Jianrong Wang. 2024. Fuzzing MLIR Compiler Infrastructure via Operation Dependency Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680360

1 Introduction

MLIR is a novel compiler infrastructure for facilitating building domain-specific compilers [24]. It not only provides a comprehensive set of shared resources benefiting multiple domains simultaneously, but also introduces dialects (defining a number of operations) to support multi-level IRs and facilitate their transformations. Its rapid adoption and recognition in both academia and industry have promoted extensive research works [33, 38, 48] and empowered a spectrum of compilers targeting diverse domains, such as the FORTRAN compiler Flang [1] and the DL compiler IREE [3]. Given the fundamental role of the MLIR compiler infrastructure, ensuring its correctness is of paramount importance. Specifically, numerous domain-specific compilers are built upon this infrastructure, and thus any bugs in it could potentially result in unexpected behaviors across all these empowered compilers. That is, the bugs in the MLIR compiler infrastructure can have a broader impact than the bugs in an individual compiler, and particularly the perniciousness of the latter has been emphasized by lots of existing studies [8, 17, 27, 37, 42, 50]. Hence, the practical significance of fuzzing the MLIR compiler infrastructure becomes self-evident.

MLIR exhibits unique characteristics, notably employing dialects to handle multi-level IRs within the infrastructure and featuring its proprietary data structure and semantics [24]. These distinctive features make the existing compiler fuzzing techniques inapplicable to the MLIR compiler infrastructure. Specifically, these techniques are typically designed to generate high-level source programs as test inputs for a specific compiler rather than the general compiler infrastructure, and also, they lack alignment in terms of data structure and semantics with MLIR. Although the generated high-level source programs can be transformed into MLIR programs by corresponding frontends, such an indirect method limits test diversity, which has been demonstrated ineffective in fuzzing the MLIR compiler infrastructure [39]. Recently, the first fuzzing technique specific to the MLIR compiler infrastructure (called MLIRSmith) has been developed, which randomly constructs MLIR programs according to its grammars [39]. Indeed, it helped detect a number of bugs, but such a random manner for generating test programs limits the effectiveness in exploring the input space largely, thereby limiting its test effectiveness.

In this work, we propose a novel fuzzing technique for the MLIR compiler infrastructure, called **MLIRod** (**MLIR** fuzzing guided by **o**peration **d**ependencies), to improve the test effectiveness. Specifically, MLIRod utilizes the dependencies between operations to guide the fuzzing process, instead of the random strategy. This is because complicated or uncommon data/control dependencies among various operations are often helpful to trigger MLIR bugs, which is also confirmed by our preliminary manual investigation on historical MLIR bugs. With this intuition, MLIRod first introduces an operation dependency graph (ODG) and its associated operation dependency coverage (OD coverage) for an MLIR program, and employs this new type of coverage as the guidance of fuzzing. To drive the fuzzing process towards increasing OD coverage, MLIRod then elaborately designs a set of mutation rules to help construct such high-quality MLIR programs, thereby enhancing the test effectiveness. In particular, OD coverage is measured in a black-box manner, further improving the practicability of MLIRod.

We applied MLIRod to fuzz the latest version of the MLIR compiler infrastructure for 50 days. In total, MLIRod detected 63 previously unknown bugs, among which 38/48 bugs have already been fixed/confirmed by MLIR developers. To further investigate the effectiveness of MLIRod, we compared MLIRod with the state-of-the-art MLIR fuzzing technique, i.e. MLIRSmith [39]. During 24-hour fuzzing (with five repeated experiments), MLIRod detected 31 bugs, while MLIRSmith detected only 14 bugs. The former improves the latter by 121.43% in terms of the number of detected bugs. The results demonstrate the significant superiority of MLIRod over MLIRSmith in fuzzing the MLIR compiler infrastructure. We also confirmed the contribution of each main component in MLIRod through extensive ablation experiments.

In summary, our work makes the following contributions:

- We design a new type of coverage (operation dependency coverage) to measure MLIR program diversity in terms of data and control dependencies among operations from operation dependency graphs.
- We propose a novel fuzzing technique for the MLIR compiler infrastructure (called MLIRod) by guiding the fuzzing process with operation dependency coverage and a set of dependency-targeted mutation rules.
- We conducted an extensive study to demonstrate the effectiveness of MLIRod, outperforming the state-of-the-art MLIR fuzzing technique. In particular, MLIRod detected 63 previously unknown bugs, among which 38/48 bugs have been fixed/confirmed by developers.

2 Background and Motivation

2.1 MLIR Compiler Infrastructure

The MLIR compiler infrastructure is a general framework that supports building diverse domain-specific compilers without creating a new Intermediate Representation (IR) with a single abstraction for each domain. It achieves this by introducing *dialects* to support multi-level IR, and providing systematic *passes* that cater to a broad spectrum of transformation and optimization functionalities, benefiting multiple domains simultaneously.

Dialects are employed to depict distinct levels of abstraction. Introducing or modifying dialects enables convenient addition or refinement of abstraction levels, making the incorporation of new domains or hardware targets straightforward. Each dialect defines a set of operations tailored to a specific domain. For example, the `TOSA` dialect defines a set of whole-tensor operations commonly employed by deep neural networks for different processors [5]. An *operation* represents a fundamental unit of computation in MLIR. For example, the `reverse` operation defined in the `TOSA` dialect aims to return a tensor with identical type and values as the input, wherein the data is reversed along the specified axis.

Dialects enable MLIR to support the representation of operations across various levels of abstraction. This collective representation of operations in different dialects forms an *MLIR program*, which is regarded as a test input for the MLIR compiler infrastructure. An illustrative example of an MLIR program is shown in Figure 1(c). In this example, the `func` dialect provides the `func` operation to define a function, the `arith` dialect provides the `constant` operation to create constant values, the `memref` dialect provides the `alloc` operation for memory allocation, and so on. An operation can receive some operands and produce results. For example, the result `%f0` produced by the `arith.constant` operation at Line 9 is used as an operand of the `affine.store` operation at Line 10. Such a collective representation facilitates the optimization of various operations at their respective appropriate levels, contributing to enhanced code generation efficiency and maximizing optimization capabilities across different hardware targets.

Passes execute a range of transformations or optimizations on MLIR programs, focusing particularly on operations among diverse dialects. It takes an MLIR program as input and outputs the transformed or optimized MLIR program. Some passes provide common transformations or optimizations across multiple dialects, such as common subexpression elimination, while some passes are tailored for specific dialects, such as the "-tosa-layerwise-constant-fold" pass for the `TOSA` dialect, which enables folding of full-layer operations on constant tensors. In particular, a number of passes can perform conversions between dialects or operations. For example, the "-tosa-to-tensor" pass transforms the operations in the `TOSA` dialect to the corresponding operations in the `tensor` dialect.

2.2 A Motivating Example

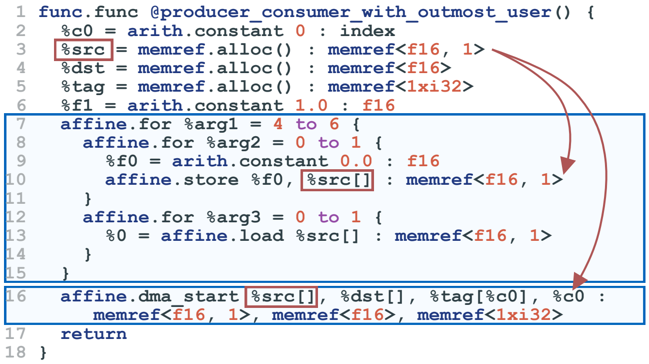
Figure 1(a) shows the Bug#76281 detected by MLIRod, which triggers a null pointer de-reference at Line 1. The reason is that the `findAncestorOpInRegion` function returns `null` that is not properly captured. Figure 1(b) shows the patch fixing the bug, where a proper check for the return result of `findAncestorOpInRegion` is added (Lines 2-3). The code in Figure 1(a) is in `isEscapingMemrefaw` function, which is invoked by the `LoopFusion::runOnOperation` function corresponding to the "-affine-loop-fusion" pass. This code snippet begins by utilizing `block->getParent()` to obtain the region (a list of blocks) that defines `block`. Subsequently, it invokes

```
1  if ( block -> getParent()
        -> findAncestorOpInRegion(*user)
        -> getBlock() != block )
2    return false;
```

(a) Buggy code of Bug#76281

```
1  auto ancestorOp = block -> getParent()
          -> findAncestorOpInRegion(*user);
2  if ( !ancestorOp )
3    return true;
4  if ( ancestorOp -> getBlock() != block )
5    return false;
```

(b) Patch for fixing Bug#76281



```
1  func.func @producer_consumer_with_outmost_user() {
2    %c0 = arith.constant 0 : index
3    %src = memref.alloc() : memref<f16, 1>
4    %dst = memref.alloc() : memref<f16>
5    %tag = memref.alloc() : memref<1xi32>
6    %f1 = arith.constant 1.0 : f16
7    affine.for %arg1 = 4 to 6 {
8      affine.for %arg2 = 0 to 1 {
9        %f0 = arith.constant 0.0 : f16
10       affine.store %f0, %src[] : memref<f16, 1>
11     }
12     affine.for %arg3 = 0 to 1 {
13       %0 = affine.load %src[] : memref<f16, 1>
14     }
15   }
16   affine.dma_start %src[], %dst[], %tag[%c0], %c0 :
        memref<f16, 1>, memref<f16>, memref<1xi32>
17   return
18 }
```

(c) The MLIR program triggering Bug#76281

Figure 1: The motivation example with Bug#76281

the findAncestorOpInRegion function to determine if the definition of *user is within the region. If *user is defined within the region, the function returns *user or its ancestor (i.e., the operation defining the block that contains *user); Otherwise, a null pointer is returned. Finally, it performs a comparison between two blocks.

The MLIR program presented in Figure 1(c) triggered this bug under the "-affine-loop-fusion" pass. Specifically, the presence of two fusionable affine.for operations within the same block (Lines 7-15) enables the triggering of the LoopFusion::runOnOperation function corresponding to the "-affine-loop-fusion" pass. In this program, the affine.store operation (Line 10) within the affine.for operation (Lines 8-11) writes the memory location (i.e., %src[]) and the affine.dma_start operation (Line 16) outside the aforementioned block (Lines 7-15) reads the same memory location. This makes the LoopFusion::runOnOperation function invoke the isEscapingMemrefaw function, executing Line 1 in Figure 1(a). In this case, the block at Line 1 in Figure 1(a) refers to the block (Lines 7-15) containing the affine.store operation (Line 10) in Figure 1(c). Hence, block->getParent() returns the region containing the only block (Lines 7-15) in Figure 1(c). The *user in Figure 1(a) refers to the affine.dma_start operation (Line 16) in Figure 1(c), which reads the same memory location as the affine.store operation (Line 10). After calling the findAncestorOpInRegion function on the affine.dma_start operation, null is returned since Line 16 is not in the region returned by block->getParent(). This ultimately triggers a null pointer de-reference bug.

From this example, we can observe that the triggering of Bug#76281 requires to satisfy complicated scoping and data referencing constraints, which are difficult to achieve by randomly generating MLIR programs (like MLIRSmith [39]). We have explicitly marked the bug-triggering constraints in Figure 1(c), where the arrows show the def-use chains of variables and the boxes show the scope of code (which can be captured by control dependencies). Note that Bug#76281 was not introduced recently, but has persisted for an extensive period (since the revision fe9d0a committed in 2022-12-14). This also implies that the state-of-the-art MLIRSmith has failed to detect it, since it had been applied to fuzzing all the revisions from 2022-10-14 to 2023-03-13 according to its paper/artifact [39], although all the operations involved in the bug-triggering MLIR program have been supported by MLIRSmith. In other words, generating those operations is not difficult for MLIRSmith, but it is actually not the case that this bug can be triggered as long as the test program contains these operations. This confirms the limitation of MLIRSmith, further **motivating our technique that carefully considers the dependencies among operations**. In particular, there are a large number of dialects and operations in the MLIR compiler infrastructure, which forms the huge input space especially when considering various dependencies among operations. Hence, this **motivates us to design an effective strategy in our fuzzing technique to guide the exploration of the huge space**.

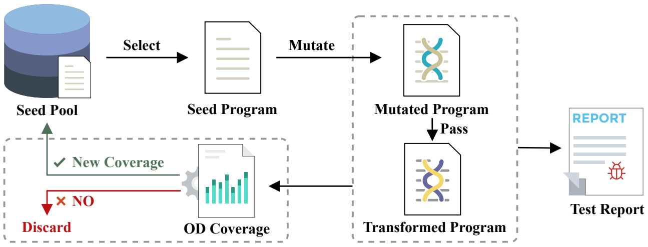


Figure 2: Overview of MLIRod

3 Approach

We propose a novel fuzzing technique for the MLIR compiler infrastructure, called **MLIRod**, to improve the test effectiveness. MLIRod proposes to exploit the dependencies between operations for guiding the fuzzing process, in order to explore the input space more efficiently. Specifically, MLIRod constructs an operation dependency graph (ODG) for each MLIR program and then measures the corresponding operation dependency coverage (OD coverage) based on the graph. The fuzzing process is thus driven towards increasing OD coverage, instead of the random manner employed by the state-of-the-art MLIRSmith [39]. To produce the MLIR programs facilitating increasing OD coverage, MLIRod elaborately designs a set of mutation rules, which focus on establishing new dependencies or modifying the existing ones between operations.

Figure 2 shows the overview of MLIRod. In the following, we first introduce ODG in Section 3.1 and its associated OD coverage in Section 3.2, then present the mutation rules in MLIRod for constructing diverse MLIR programs in Section 3.3, and finally describe the overall fuzzing process of MLIRod guided by the defined ODG and OD coverage as well as the mutation rules in Section 3.4.

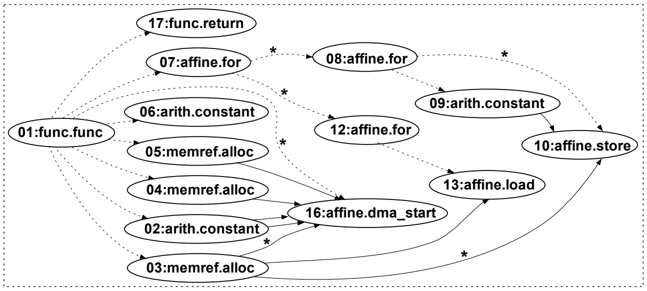


Figure 3: ODG of the MLIR program shown in Figure 1(c)

3.1 Operation Dependence Graph

As illustrated in our motivation example, the dependencies between operations are relevant to the detection of bugs in the MLIR compiler infrastructure. Furthermore, certain optimizations in the MLIR compiler infrastructure can only be activated with the occurrence of some dependencies between operations in an MLIR program. For example, the "-buffer-loop-hoisting" optimization, which aims to move allocation operation out of loop nests, requires the control dependency between an allocation operation and a loop operation (such as `scf.for`). Therefore, MLIRod takes operation dependencies as the core to improve the fuzzing process.

Inspired by the concept of program dependency graph (PDG) [11], we define operation dependency graph (ODG) in MLIRod. Similarly, we consider both data dependencies and control dependencies between operations in ODG. For ease of presentation, we call them *operation data dependencies* and *operation control dependencies* in this paper. As introduced in Section 2.1, an operation consists of its name, operands, and results. If we just treat an operation with a specific name as the basic unit for analyzing data dependencies and control dependencies, it is relatively coarse-grained since some bugs can be triggered under certain types of operands for an operation. If we treat an operation with specific values of operands and results as the basic unit, it is relatively fine-grained and thus leads to the enormous space for ODG due to the huge value space for operands and results. To balance effectiveness and efficiency, we consider the types of operands and results for an operation to form the basic unit for subsequent analysis, which is called *operation instance* for ease of presentation in this paper.

In the following, we formally define operation instance, operation data dependency, operation control dependency, and operation dependency graph (ODG).

Definition 1 (Operation Instance). *The operation instance (oi) can be defined as a tuple: (Oname, ODtypes, Rtypes), where Oname is an operation name, ODtypes is a list of operand types, and Rtypes is a list of result types.*

Definition 2 (Operation Data Dependency). *Let oi_i and oi_j be two operation instances in an MLIR program, oi_j is data dependent on oi_i if oi_i defines a value v and oi_j accesses the value v. We use a directed edge $e^{data}_{i,j}$ from oi_i to oi_j to represent that oi_j is data dependent on oi_i.*

Definition 3 (Operation Control Dependency). *Let oi_i and oi_j be two operation instances in an MLIR program, oi_j is control dependent on oi_i if the execution of oi_j is dominated by the outcome of the execution involving oi_i. We use a directed edge $e^{control}_{i,j}$ from oi_i to oi_j to represent that oi_j is control dependent on oi_i.*

Definition 4 (Operation Dependency Graph (ODG)). *ODG is defined as a 2-tuple (N, E), where N is a set of nodes and E is a set of edges. Each node is an operation instance and each edge represents either operation data dependency or operation control dependency.*

Figure 3 shows an example ODG corresponding to the MLIR program depicted in Figure 1(c). In Figure 3, the dashed arrows represent the operation control dependency and the solid arrows represent the operation data dependency. The edges labeled with "*" are the operation dependencies relevant to the triggering of this bug presented in Section 2.2. Here, operation control dependency is more about the block nesting relationship and variable accessibility. Specifically, if an operation instance oi_i has a code block containing an operation instance oi_j, then the edge $e^{control}_{i,j}$ exists.

3.2 Operation Dependency Coverage

Traditional white-box or grey-box code coverage is typically expensive, especially for larger-scale compiler infrastructure. To enable efficient and effective guiding of the fuzzing process, MLIRod defines a black-box coverage criterion from ODG, i.e., operation dependency coverage (OD coverage), and utilizes the OD coverage to guide the test generation process.

Definition 5 (d-step Reachability). *Given an ODG G=(N, E) and a node $oi_i \in N$, for each node $oi_j \in N$, we say oi_j is d-step reachable from oi_i iff there exists a node sequence <$oi_0, oi_1, \cdots, oi_l$>, where $l \leq d \wedge oi_0 = oi_j \wedge oi_l = oi_i$ and $\forall k \in [0, l-1], e^{data}_{k,k+1} \in E \vee e^{control}_{k,k+1} \in E$. We use ($oi_i \overset{d}{\rightsquigarrow} oi_j$) to represent this relation.*

Definition 6 (Operation Dependency Pattern). *Given an ODG $G = (N, E)$, the OD pattern within d-step reachability regarding a given node $oi_i \in N$ is the largest subgraph $G' = (N', E')$ of G, where $N' \subset N \wedge E' \subset E$ and $\forall oi_j \in N', oi_j = oi_i \vee (oi_i \overset{d}{\rightsquigarrow} oi_j)$.*

According to this definition, the OD pattern with 0-step reachability regarding the node 13 in Figure 3 is $G = (N, E)$, where $N = \{oi_{13}\}$ and $E = \emptyset$. Similarly, the OD pattern with 1-step reachability regarding the node 13 is $G' = (N', E')$, where $N' = \{oi_{03}, oi_{12}, oi_{13}\}$ and $E' = \{e^{control}_{12,13}, e^{data}_{03,13}\}$. Note that the corresponding nodes and edges are the concrete operation instances in actual implementation. For example, the node oi_{13} denotes the operation instance for `affine.load` at Line 13 (in Figure 1(c)), while the edge $e^{control}_{12,13}$ denotes that the operation instance for `affine.load` at Line 13 is control-dependent on the operation instance for `affine.for` at Line 12. In this way, we define that two OD patterns are equal to each other if they contain the same operation instances with the same dependency relations, i.e., the same subgraph from an ODG. Then, we define the concept of operation dependency coverage.

Definition 7 (Operation Dependency Coverage). *The operation dependency coverage (OD coverage) of a given test suite T of MLIR programs is defined in formula (1),*

$$OD_{cov} = \frac{|\bigcup_{t \in T} P_d^{G_t}|}{|\bigcup_{t' \in \mathcal{A}} P_d^{G_{t'}}|} \tag{1}$$

where G_t is the ODG for the MLIR program t, and $P_d^{G_t}$ is a set of all possible OD patterns over G_t regarding d-step reachability, and $\mathcal{A}$ is the set of all possible MLIR programs. That is, OD coverage is measured on the entire input space rather than a single MLIR program.

It is impossible to enumerate all tests in $\mathcal{A}$, and thus it is hard to estimate the denominator of OD_{cov}. One potential way to conduct the estimation is to randomly generate a large number of MLIR programs and calculate the number of unique OD patterns, until the number of OD patterns does not increase, or increases very slowly. However, this is rather expensive [43]. In our work, we only use the OD coverage to guide the exploration of input space, rather than use it for test adequacy evaluation. Hence, we can directly use the enumerator, i.e., the concrete number of covered OD patterns in the executed MLIR programs, to achieve the same purpose. In the following, we mean the concrete number of covered OD patterns in the executed MLIR programs when we refer to OD coverage.

Note that in MLIRod, we provide a method of defining OD coverage from ODG based on OD pattern. However, this may be not the only method and also not the optimal one. For example, we could also extract paths from ODG to define OD coverage. Here, we aim to provide a cost-effective method to enable our MLIRod idea and then take the exploration of more (potentially better) methods for further improving its effectiveness as our future work.

3.3 Mutation Rules

In MLIRod, we elaborately design a set of mutation rules to change the existing MLIR programs towards increasing OD coverage. These mutation rules aim to establish new dependencies or modify the existing dependencies between operations. From the perspective of ODG, they can operate both nodes and edges in an ODG to achieve this goal. Specifically, we have designed four types of mutation rules, including node insertion, node deletion, data dependency modification, and control dependency modification. In the following, we will introduce each mutation rule in detail. Due to the space limit, we prepared a set of examples to facilitate understanding these mutation rules at our project homepage [4].

Node Insertion (R1) denotes inserting a new random node into the ODG of a given MLIR program and then constructing the dependency edges between it and the existing nodes according to the operands required by the new node. This mutation rule can introduce new nodes to make the ODG more complicated and thus have a larger possibility to detect hard-to-trigger bugs.

Formally, assuming that the newly inserted node (i.e., an operation instance) is represented as $oi_i = (\_, ODtypes_i, \_)$[1]. Then, for each $t_k \in ODtypes_i$, MLIRod tries to find an *accessible* operation instance $oi_e = (\_, \_, Rtypes_e)$ whose result types contain the type t_k, i.e., $t_k \in Rtypes_e$. In this way, a new data dependency edge $e_k = e_{e,i}^{data}$ can be constructed. We use $E_d = \{e_1, e_2, ..., e_n\}$ (n is equal to the size of $ODtypes_i$) to represent the set of data dependency edges to be constructed according to the required operands in oi_i. Here, we use "*accessible*" to denote that the referenced value is defined before use. If no such existing operation instance oi_e exists for t_k, MLIRod then generates a new node oi_e with a random value of the corresponding type for this operand (which can be implemented by invoking the APIs provided by MLIRSmith).

To construct more complicated MLIR programs for fuzzing the MLIR compiler infrastructure with higher OD coverage, MLIRod prefers to insert the new node to some location where it can be controlled by a certain existing operation instance oi_c. In this way, a new control dependency edge $e_{c,i}^{control}$ can be introduced. As a result, given that the ODGs before and after applying this mutation rule as $G = (N, E)$ and $G' = (N', E')$ respectively, we have $N' = N \cup \{oi_i, oi_e\} \wedge E' = E \cup E_d \cup \{e_{c,i}^{control}\}$.

Node Deletion (R2) denotes randomly deleting an existing node from the OGD of a given MLIR program and then updating the data and control dependencies broken by the deletion. Specifically, new data dependencies have to be constructed for those nodes that are data dependent on the deleted one in order to make the mutated program valid. This mutation rule can help cover more diverse OD patterns by changing nodes in the ODG and the corresponding dependency edges, thereby increasing OD coverage.

Formally, assuming that the deleted node is represented as oi_d and the original ODG is $G = (N, E)$. The deletion of oi_d may affect some other node oi_e that is either dependent on or depended by oi_d, and the edges affected by this deletion are denoted as $e_{e,d}^{data}$, $e_{e,d}^{control}$, $e_{d,e}^{data}$, and $e_{d,e}^{control}$. Here, $e_{e,d}^{data}$ and $e_{e,d}^{control}$, denoting that the deleted node oi_d is data or control dependent on some existing node oi_e, can be directly deleted. However, for $e_{d,e}^{data}$ and $e_{d,e}^{control}$, additional modifications are required to ensure the validity of the mutated program. Specifically, $e_{d,e}^{data}$ represents that some existing node oi_e uses the result of oi_d, and thus is data dependent on oi_d. For this case, similar to the rule of node insertion, this mutation rule tries to find another *accessible* node oi_r with the same result type as oi_d, and then updates the edge $e_{d,e}^{data}$ to $e_{r,e}^{data}$. In other words, the operand of oi_e is updated to be another value. For $e_{d,e}^{control}$, representing that some existing node oi_e is control dependent on oi_d, this mutation rule deletes it from E and its associated node oi_e from N accordingly. Note that this process is recursive since the deletion of the node oi_e can be viewed as another round of node deletion, where another updating procedure has to be conducted.

Data Dependency Modification (R3) denotes replacing all the nodes that are data depended by a randomly selected node in the ODG of a given MLIR program with other *accessible* and type-compatible nodes. In other words, all the operand values of the selected node are replaced with other type-compatible ones. This mutation rule can help establish new data dependencies in the MLIR program by reorganizing some data dependencies associated with this node, which facilitates the detection of more diverse bugs.

Formally, assuming that the node $oi_{dm} = (\_, ODtypes, \_)$ is the selected node, whose operands (or data dependencies) are required to be updated in the given ODG $G = (N, E)$. For each operand type $t_k \in ODtypes$, if there is an associated edge $e_{j,dm}^{data} \in E$ (i.e., oi_{dm} is data dependent on oi_j), MLIRod tries to find another *accessible* node $oi_r = (\_, \_, Rtypes)$, where $t_k \in Rtypes \wedge oi_r \neq oi_j$, for updating the data dependency from $e_{j,dm}^{data}$ to $e_{r,dm}^{data}$. More explicitly, the k^{th} operand value of oi_{dm} is updated to use the result of oi_r rather than oi_j. If there is no alternative oi_r found for replacing oi_j, it keeps unchanged. Finally, if no data dependency edge is updated in E,

[1] For ease of presentation, we use "_" to denote any valid instances, symbols, or values if they do not affect the understanding in the paper.

a new node oi_{dm} will be selected to perform the same mutation process until it succeeds or reaches a given constraint (e.g., a time budget or times of mutation).

Control Dependency Modification (R4) denotes updating the control dependency of a randomly selected node in the ODG of a given MLIR program, while keeping all the data dependencies unchanged. Specifically, this mutation rule aims to move the operation instance from an inner conditional block to the outer one, and thus achieves the modification of the control dependency, which can complement the other mutation rules to a large extent.

Formally, assuming that the selected node is oi_{cm} for control dependency modification and the associated ODG is $G = (N, E)$. If there exist nodes $oi_j \in N$ and $oi_r \in N$ that satisfy $e^{control}_{j,cm} \in E \wedge e^{control}_{r,j} \in E$, then this mutation rule is to change $e^{control}_{j,cm}$ to $e^{control}_{r,cm}$. In other words, the control dependent node of oi_{cm} is updated from oi_j to oi_r. That is, the operation instance oi_{cm} is moved from an inner conditional block to the outer one. However, this change may break the data dependencies of the node oi_{cm} since the movement may cause the reference to some existing node to be invalid. To keep the data dependencies unchanged and ensure the validity of the mutated MLIR program, this rule also moves the dependent nodes of oi_{cm} accordingly. That is, for each node $oi_e \in N$, if $e^{data}_{e,cm} \in E \wedge e^{control}_{j,e} \in E$ (i.e., oi_{cm} is data dependent on oi_e and they are in the same conditional block), it then updates $e^{control}_{j,e}$ to $e^{control}_{r,e}$. That is, the node oi_e is moved from the inner conditional block to the outer one together. In this way, the data dependency $e^{data}_{e,cm}$ can keep unchanged as long as their relative order is preserved. Similarly, the movement of the node oi_e may further affect other nodes. We recursively handle this issue by following the same process presented above.

3.4 Overall Fuzzing Process

The general workflow of MLIRod for fuzzing the MLIR compiler infrastructure, as shown in Figure 2, consists of seed pool initialization, mutation-based MLIR program generation, MLIR-pass-based fuzzing, and OD-coverage-based seed pool maintenance.

Seed Pool Initialization: MLIRod initializes the seed pool by utilizing the state-of-the-art MLIR program generator (i.e., MLIRSmith [39]) to generate N MLIR programs. Note that MLIRod is not specific to MLIRSmith, and in theory, the seed pool can be initialized by any MLIR program generators (however, MLIRSmith is the only one until now) or open source MILR programs. Then, MLIRod calculates the OD coverage of these seed programs.

Mutation-based MLIR Program Generation: In each fuzzing iteration, MLIRod randomly selects a seed program from the seed pool and randomly selects a mutation rule to mutate it. In this way, a new MLIR program can be generated for fuzzing the MLIR compiler infrastructure.

MLIR-Pass-based Fuzzing: MLIRod collects the entire set of MLIR passes from documentation [7]. For a generated MLIR program, MLIRod randomly selects k passes with replacement from the entire set of MLIR passes and randomly determines the order of these selected passes to form an MLIR pass sequence. Then, MLIRod applies each pass in the sequence to the MLIR program. If a crash occurs during this process, we regard that this MLIR program detects a bug. Note that since we just considered crash as the test oracle, we did not specially investigate lowering paths, which can be regarded as our future work. Each specified pass is designed to produce a newly transformed or optimized MLIR program, potentially introducing new dialects and operations. To broaden the scope of dialects and operations under fuzzing, MLIRod gathers all the newly transformed or optimized MLIR programs for OD-coverage-based seed pool maintenance.

OD-Coverage-based Seed Pool Maintenance: If a generated MLIR program via mutation achieves new OD coverage, MLIRod puts the generated MLIR program into the seed pool, inspired by the general coverage-based fuzzing practice (coverage-increasing test cases tend to help generate more effective tests) [6, 12, 44]. Furthermore, MLIRod checks whether the transformed program by each MLIR pass improves the OD coverage. If the improvement is identified, MLIRod also puts it into the seed pool. In this way, the fuzzing process can be effectively driven towards increasing OD coverage, and thus the fuzzing effectiveness can be improved.

4 Evaluation

To evaluate MLIRod, we conducted an extensive study to answer the following research questions (RQs):

- **RQ1**: How does MLIRod perform in detecting previously unknown bugs in the MLIR compiler infrastructure?
- **RQ2**: How does MLIRod perform in bug detection compared with the state-of-the-art MLIRSmith?
- **RQ3**: How does each main component in MLIRod contribute to the overall effectiveness?
- **RQ4**: How does the step of reachability d in measuring OD coverage affect the effectiveness of MLIRod?

4.1 Experimental Setup

To answer RQ1, we applied MLIRod to fuzz the latest revisions of the MLIR compiler infrastructure (from revision fe5370d to revision 6e90f1) for 50 days. It aims to detect previously unknown bugs. To answer RQs 2-4, we selected the revision eb6014 (the latest one when we started the experiments for RQs 2-4), totaling 429.5K lines of code, as the subject for 24-hour fuzzing to perform fair comparisons. To reduce the influence of randomness, we repeated the experiments in RQs 2-4 for five times, and aggregated the results same as the existing study [39]. To balance the testing effectiveness and efficiency, we set the step of reachability in measuring OD coverage (d) to 2, the number of passes for constructing an MLIR pass sequence (k) to 10, and the number of initial seed programs (N) to 50 in MLIRod by default. In RQ4, we will investigate the influence of an important parameter (i.e., d) on the effectiveness of MLIRod. All our experiments were conducted on a machine with Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and 128G Memory, Ubuntu 20.04.6 LTS.

4.1.1 *Compared Techniques.*

To answer RQ2, we compared MLIRod with the state-of-the-art technique for fuzzing the MLIR compiler infrastructure (i.e., **MLIRSmith** [39]). We adopted the released implementation of MLIRSmith and used the recommended parameter settings in the work of MLIRSmith for fair comparison [39].

MLIRod is orthogonal to MLIRSmith to some extent, since the former can produce more MLIR programs through mutation based on the programs generated by the latter. Nevertheless, it is still important to compare their fuzzing effectiveness. To more clearly highlight the contribution of our mutation-based MLIRod, we used MLIRSmith to generate only 50 programs as the initial seed programs in MLIRod. During the 24-hour fuzzing, MLIRSmith generates 12,855 programs, which is significantly more than the 50 initial seed programs employed by MLIRod, while MLIRod generates 9,291 programs through mutations (whose sizes are ranging from 94 to 18,776 lines of code). That is, MLIRod under this setting does not incorporate too much from MLIRSmith. Hence, such an experiment can help compare the mutation-based MLIRod and the grammar-based MLIRSmith clearly.

Note that the existing study [39] has demonstrated MLIRSmith outperforms the indirect method of transforming high-level source programs generated by NNSmith [31] (the state-of-the-art test generator for fuzzing deep learning compilers by generating ONNX computation graphs) into MLIR programs through available frontends. Hence, we chose the better one (MLIRSmith) as our compared technique in our study. For sufficient comparison, we also conducted an experiment to compare MLIRod with the method using NNSmith, but put the detailed results on our project homepage [4] due to the space limit. In summary, during 24-hour fuzzing, MLIRod (31) indeed detected much more bugs than NNSmith (10).

In RQ3, we investigated the contributions of each mutation rule, the mechanism of enriching dialects and operations with MLIR passes, and our OD coverage guidance in MLIRod. Accordingly, we constructed seven variants of MLIRod for comparison. **MLIRod**$^{R1}_{w/o}$, **MLIRod**$^{R2}_{w/o}$, **MLIRod**$^{R3}_{w/o}$, and **MLIRod**$^{R4}_{w/o}$ remove each mutation rule (R1, R2, R3, or R4) from MLIRod, respectively. **MLIRod**$^{pass}_{w/o}$ removes the mechanism of putting pass-produced programs for seed pool maintenance from MLIRod. **MLIRod**$_{rand}$ removes the OD coverage guidance from MLIRod and randomly puts each generated MLIR program into the seed pool. **MLIRod**$_{edge}$ replaces the OD coverage with the widely-studied edge coverage [12] as the guidance in MLIRod. We collected edge coverage following AFL++ [12].

To answer RQ4, we studied several settings for the step of reachability d in measuring OD coverage in MLIRod, i.e., 0, 1, 2, 3. The 0-step OD coverage means that MLIRod only collects individual operation instances in the MLIR program for measuring coverage.

4.1.2 Metrics. We adopted two metrics to measure the effectiveness of each technique: 1) the number of detected bugs and 2) the number of covered lines in the subject. Both of them have been widely used in the existing work on fuzzing [25, 47, 49]. During the fuzzing process, a number of crashes may be triggered, but many of them may be caused by the same root causes. Hence, it is important to de-duplicate them to accurately measure the number of detected bugs. We de-duplicated them according to crash messages same as the existing work [39]. Then, we submitted unique crashes to MLIR compiler infrastructure developers, and counted the number of detected bugs based on their feedback. Note that developers may directly fix bugs without updating issue reports. As our bugs were detected on the latest revisions at the time of our fuzzing and reporting, we also checked whether each bug without developers' response still existed on subsequently-commited revisions following the existing work [10]. If it is a crash bug and did not exist on subsequently-commited revisions, we regard it as the case of developer fixing without updating reports. Regarding line coverage, we collected it using the widely-used gcov [2].

4.2 RQ1: New Bugs Detected by MLIRod

In total, MLIRod detected 63 previously unknown bugs during 50-day fuzzing, among which 48/38 bugs have been confirmed/fixed by developers and 15 bugs are still awaiting feedback. Table 1 shows the details of these detected bugs, including the bug ID, the root cause for each fixed bug (identified by developers), the type of the MLIR pass where each bug occurs, and the bug status. The bugs detected by MLIRod exhibit diversity, spanning across a broad spectrum of MLIR passes and root causes. Subsequently, we conducted a comprehensive analysis of these bugs from these two aspects.

Bug-occurring Pass Analysis: As introduced in the existing work [39], in general, there are four types of passes in the MLIR compiler infrastructure:

- Conversion passes perform transformations between dialects to lower the abstraction level. 14 out of 63 bugs are identified within conversion passes.
- Bufferization passes transform the operations that exhibit tensor semantics into the operations with memref semantics. 4 bugs are associated with bufferization passes.
- General transformation passes are universally applicable to all dialects and are designed to perform common optimizations/transformations. 20 bugs manifest within general transformation passes.
- Domain-specific passes perform domain-specific optimizations/transformations within each specific dialect. 25 bugs occur in domain-specific passes.

Root Cause Analysis: The detected bugs by MLIRod covered all the five root causes introduced in the existing work [39], i.e., Incomplete Verifier (IV), Incorrect Pattern (IP), Incorrect Rewrite Logic (IRL), Unregistered Dialect (UD), and Incorrect Assertion (IA). Note that the root causes were identified by the developers of the MLIR compiler infrastructure and thus only the 38 *fixed* bugs have been labeled with the corresponding root causes.

7 bugs are caused by *Incomplete Verifier*. Specifically, each pass is equipped with a verifier to assess the compatibility between the pass and specific operations. This root cause is the absence or incompleteness of a necessary verifier for a pass. This deficiency results in the pass operating on incompatible operations, ultimately leading to a crash. That is, compilers should normally reject invalid programs rather than directly crash. For example, Bug#70418 (Figure 4) was triggered when the "-convert-func-to-spirv" pass was applied to the program with the `affine.vector_load` operation, which uses an invalid value for the `%dim` operand. Such an invalid value can be produced only if the `memref.dim` operation takes an out-of-bound index as its operand (i.e., the value of `%c6` is larger than the shape of `%alloc_4`). The root cause lies in missing a verifier between this pass and the `affine.vector_load` operation to capture such an exception in advance, resulting in a crash. Also, the data dependency between `memref.dim` and `affine.vector_load` makes a large contribution to detecting this bug, demonstrating the importance of considering data dependencies between operations in MLIRod. Regarding MLIRSmith, it adopts the random strategy

Table 1: Previously unknown bugs detected by MLIRod

Bug Id	Root Cause[1]	Pass Category	Status	Bug Id	Root Cause	Pass Category	Status
64385*	IP	Conversion	fixed	71281	UD	Domain Specific(async)	fixed
64408	IP	Domain Specific(linalg)	fixed	73190	IRL	General Transformation	fixed
64409	IP	Conversion	fixed	73191	—	Bufferization	confirmed
64622	IRL	Domain Specific(linalg)	fixed	73285	—	General Transformation	submitted
64638	—	Bufferization	submitted	73288*	IRL	Domain Specific(linalg)	fixed
64639	—	Domain Specific(GPU)	confirmed	73289	IP	Conversion	fixed
64674	IP	General Transformation	fixed	73381	—	Domain Specific(arith)	submitted
67760	IV	Domain Specific(TOSA)	fixed	73382	IA	General Transformation	fixed
67761	IV	Conversion	fixed	73383	IP	General Transformation	fixed
67763	IP	Domain Specific(TOSA)	fixed	73532	—	General Transformation	submitted
67977	IRL	Bufferization	fixed	73534	—	Conversion	submitted
68187	IV	Domain Specific(TOSA)	fixed	73540	—	Domain Specific(affine)	submitted
68481	IV	Conversion	fixed	73547	—	General Transformation	confirmed
68483	IP	Domain Specific(memref)	fixed	74227*	IRL	Domain Specific(sparse_tensor)	fixed
68486	IV	Conversion	fixed	74234	—	Domain Specific(arith)	confirmed
68948	IRL	Bufferization	fixed	74236	—	General Transformation	confirmed
68950*	IP	Domain Specific(transform)	fixed	74237	IP	Domain Specific(TOSA)	fixed
70180	IP	Domain Specific(memref)	fixed	74301	—	General Transformation	submitted
70183	—	General Transformation	submitted	74306	IP	General Transformation	fixed
70278	—	Conversion	confirmed	74308	—	Conversion	submitted
70415	IV	Domain Specific(TOSA)	fixed	74313*	IRL	General Transformation	fixed
70418	IV	Conversion	fixed	74453	IRL	Domain Specific(llvm)	fixed
70439	—	Domain Specific(affine)	submitted	74461	—	General Transformation	confirmed
70633	IP	General Transformation	fixed	74466	—	Conversion	submitted
70884	IP	General Transformation	fixed	74937	—	Domain Specific(arith)	confirmed
70887	IP	General Transformation	fixed	74940	—	Conversion	submitted
70902	IP	General Transformation	fixed	75758	IP	Domain Specific(scf)	fixed
70913	—	General Transformation	submitted	75770	—	Conversion	confirmed
71036	IP	Domain Specific(vector)	fixed	76281	IRL	Domain Specific(affine)	fixed
71147	—	Domain Specific(affine)	submitted	76309	—	Domain Specific(affine)	submitted
71150	IP	Conversion	fixed	77420	—	General Transformation	confirmed
71153	IP	General Transformation	fixed				

[1] Full Name of Root Cause: IV (Incomplete Verifier), IP (Incorrect Pattern), IRL (Incorrect Rewrite Logic), UD (Unregistered Dialect), IA (Incorrect Assertion)

* These bugs belong to the cases of developer fixing without updating reports.

```
%dim = memref.dim %alloc_4, %c6 :
       memref<4xi64> // out-of-bound
%70  = affine.vector_load %alloca_100[%dim] :
       memref<100xi64>, vector<31xi64>
```

Figure 4: Program snippet for triggering Bug#70418 (IV)

```
memref.dealloc %arg0 : memref<?xf32>
%0 = bufferization.clone %arg0 : memref<?xf32> to
     memref<32xf32>
```

Figure 5: Program snippet for triggering Bug#74306 (IP)

for program generation, and thus it is hard for it to construct such a data dependency and thus detect this bug.

20 bugs are caused by *Incorrect Pattern*. Specifically, a set of patterns is employed by each pass to identify the operations that this pass intends to transform or optimize. If certain patterns are incorrect, this pass may inadvertently transform or optimize unexpected operations, resulting in a crash. For example, Bug#74306 (Figure 5) was caused since the "-canonicalize" pass improperly optimizes the `bufferization.clone` operation. This optimization relies on the assumption that the `memref.dealloc` operation follows the `bufferization.clone` operation. However, there is an `memref.dealloc` operation preceding the `bufferization.clone` operation in this program, causing that this optimization processes the variable (`%arg0`) that has been released by this `memref.dealloc` operation and thus crashes. This bug has been fixed by modifying the pattern to avoid activating this optimization under such cases. The data dependency between `bufferization.clone` and `memref.dealloc` on `%arg0` contributes to the triggering of this bug.

```
llvm.func @func1() {
  scf.parallel (%arg0) = (%c0) to (%c22) step (%c1) {
    ...
  }
}
```

Figure 6: Program snippet for triggering Bug#71281 (UD)

9 bugs are caused by *Incorrect Rewrite Logic*. Specifically, passes rewrite matched operations into new forms. However, if the logic governing this rewriting is flawed, the pass may generate incorrect operations. We have introduced such a bug in Section 2.2.

One bug is caused by *Unregistered Dialect*. Specifically, to facilitate the transformation from an operation in one dialect to an operation in another dialect, the latter dialect must be registered within the pass. Without this registration, the transformation process will crash. For example, Bug#71281 (Figure 6) was caused due to missing to register the `func` dialect for the "-async-parallel-for" pass. Specifically, when the "-async-parallel-for" pass is activated by the `scf.parallel` operation, it requires the information of the `func` dialect. However, this dialect is not loaded due to missing registration, ultimately leading to a crash. This bug cannot be triggered by MLIRSmith, since all programs generated by it use the `func` dialect. This bug-triggering program is produced by MLIRod on the seed program complemented by the "-convert-func-to-llvm" pass, which lowers all the operations in the `func` dialect to the

```
scf.for %arg4 = %c0 to %c22 step %c1 {
  %dim = memref.dim %alloc, %c1 :
          memref<?xi1> // undefined behavior
  ...
}
```

Figure 7: Program snippet for triggering Bug#73382 (IA)

corresponding operations in the `llvm` dialect. It leads to the disappearance of the `func` dialect in the seed program as well as this bug-triggering program. This demonstrates the necessity of the mechanism of putting pass-produced programs for seed pool maintenance in MLIRod.

One bug is caused by *Incorrect Assertion*. Specifically, the MLIR compiler infrastructure includes numerous assertions designed to verify internal states. Incorrect assertions will make the transformation/optimization process crash, even if internal states are correct. For example, Bug#73382 (Figure 7) was caused due to an incorrect assertion in the "-loop-invariant-code-motion" pass. Specifically, this pass works since there is an `scf.for` operation in this program. However, the `memref.dim` operation within the `scf.for` operation has an out-of-bound dimension index, which is an undefined behavior. The assertion in this pass incorrectly processes this undefined behavior, leading to a crash. The control dependency between the `scf.for` operation and the `memref.dim` operation with an out-of-bound dimension index contributes to the triggering of this bug.

4.3 RQ2: MLIRod v.s. MLIRSmith

Figure 8(a) shows the number of bugs detected by each technique as the fuzzing process progresses for 24 hours. As aforementioned, we repeated the experiments for five times and aggregated the results to reduce the influence of randomness involved in fuzzing. Overall, MLIRod detected 31 bugs, while MLIRSmith just detected 14 bugs during the same fuzzing time. The improvement of MLIRod over MLIRSmith is 121.43%. From this figure, MLIRSmith reached saturation in bug detection quickly, while MLIRod can detect bugs continuously within the fuzzing time. The results demonstrate the superiority of MLIRod over MLIRSmith in bug detection. We also analyzed the overlap of the bugs detected by MLIRod and MLIRSmith. 21 bugs detected by MLIRod cannot be detected by MLIRSmith during the given fuzzing time, while only 4 bugs detected by the latter cannot be detected by the former, further confirming the effectiveness of MLIRod. The possible reason behind the 4 missed bugs by MLIRod is that, the random fuzzing strategy employed in MLIRSmith could make it explore a portion of input space that is still unexplored by MLIRod within the given fuzzing time.

We further investigated why MLIRod outperforms MLIRSmith significantly. Specifically, during the given fuzzing time, MLIRod covered 116,641 lines of code for the MLIR compiler infrastructure, while MLIRSmith covered 113,971 lines. Among the lines covered by MLIRod, 13,373 lines cannot be covered by MLIRSmith. The maximum branch depth achieved by MLIRod is 7, same as that by MLIRSmith, but MLIRod covered 715 more branches than MLIRSmith. Besides, the MLIR programs produced by MLIRod involved 430 operations from 20 dialects (including the 13 dialects supported by MLIRSmith and the 7 dialects produced through lowering/optimizations), while those by MLIRSmith just involved 256 operations from 13 dialects during the given fuzzing time. Note that all dialects and operations generated by MLIRod and MLIRSmith can be found at our homepage [4]. That is, MLIRod explored larger input space due to its ability of guiding the fuzzing process with OD coverage and efficiently supporting more dialects and operations with MLIR passes. Hence, the fuzzing effectiveness of MLIRod can be significantly enhanced.

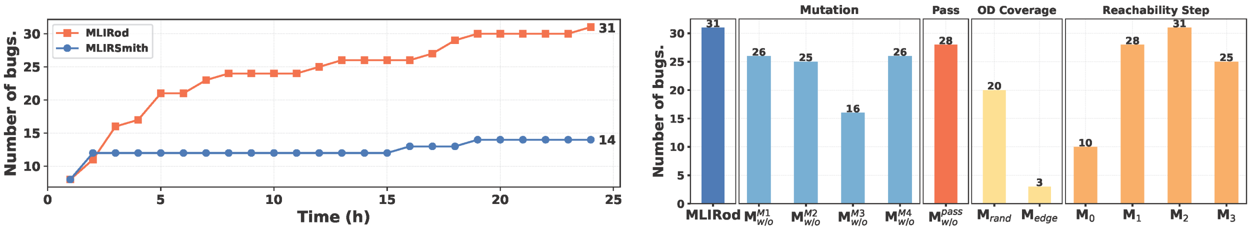
4.4 RQ3: Contribution of Each Main Component in MLIRod

To evaluate the contribution of each component, we applied the seven variants (introduced in Section 4.1.1) to fuzzing the MLIR compiler infrastructure, respectively. Figure 8(b) shows the number of bugs detected by each variant during the given fuzzing time (24 hours with five repeated experiments).

By comparing MLIRod with MLIRod$_{w/o}^{R1}$, MLIRod$_{w/o}^{R2}$, MLIRod$_{w/o}^{R3}$, and MLIRod$_{w/o}^{R4}$, MLIRod detected more bugs than all the four variants, demonstrating the contribution of each mutation rule. Among the four variants, MLIRod$_{w/o}^{R3}$ detected the smallest number of bugs (i.e., 16), indicating the most significant contribution of data dependency modification. The possible reasons are that (1) the mutation rule of data dependency modification has larger mutation space than that of control dependency modification, since the former may involve the combinations of various operations while the latter just involves the combinations with the control-related operations (e.g., `scf.if` and `scf.for`); (2) data dependency modification is more efficient than node insertion and node deletion to generate a mutated MLIR program, which can generate more programs for fuzzing during the same time.

By comparing with MLIRod$_{w/o}^{pass}$, we found that the former (i.e., 31) outperforms the latter (i.e., 28) in bug detection during the given fuzzing time. The results demonstrate the contribution of the mechanism of putting pass-produced programs for seed pool maintenance. Through further analysis, the MLIR programs generated by MLIRod involved 430 operations from 20 dialects while those by MLIRod$_{w/o}^{pass}$ involved 256 operations from 13 dialects. That is, MLIRod explored larger input space than MLIRod$_{w/o}^{pass}$, explaining the superiority of MLIRod over MLIRod$_{w/o}^{pass}$. In fact, MLIRod$_{w/o}^{pass}$ has the same space of dialects and operations as MLIRSmith, but the former (i.e., 28) still detected more bugs than the latter (i.e., 14) during the given fuzzing time. These results demonstrate the effectiveness of the remaining components in MLIRod.

By comparing MLIRod with MLIRod$_{rand}$, we found that the former detected 31 bugs while the latter detected 20 bugs during the given fuzzing time. Also, the former achieved 429.6K OD coverage while the latter achieved 240.7K OD coverage. The results demonstrate the contribution of guiding the fuzzing process with OD coverage in MLIRod. Moreover, we compared our OD coverage guidance with the widely-used edge coverage guidance in the area of fuzzing by constructing the corresponding variant (i.e., MLIRod$_{edge}$). From Figure 8(b), we found that MLIRod$_{edge}$ just detected 3 bugs during the same fuzzing time, which even performs worse than MLIRod$_{rand}$. After investigation, we found that edge coverage is hard to increase during the fuzzing process for the MLIR compiler infrastructure, causing that very few generated programs were put into the seed pool for further mutations. The results indicate that



(a) Number of detected bugs by MLIRod and MLIRSmith

(b) Number of detected bugs by MLIRod and its variants

Figure 8: Number of detected bugs by MLIRod and comparison approaches (M in variant names refers to MLIRod)

OD coverage seems more proper than the widely-used edge coverage in our scenario of fuzzing the MLIR compiler infrastructure.

4.5 Influence of the Step of Reachability in Measuring OD Coverage

We finally investigated the influence of one important parameter in MLIRod (i.e., the step of reachability *d* in measuring OD coverage). As presented in Section 4.1.1, we studied four different settings (*d* = 0, 1, 2, 3, respectively). For ease of presentation, we call them MLIRod_0, MLIRod_1, MLIRod_2 (i.e., MLIRod), MLIRod_3, respectively.

Figure 8(b) presents the number of detected bugs by MLIRod under different settings of *d* during the given fuzzing time. We found that they detected 10, 28, 31, and 25 bugs, respectively. Among them, MLIRod_0 performs the worst significantly, since it solely considers the diversity of operation instances (leading to saturated coverage quickly) but ignores the dependencies between them. Under the remaining three settings, MLIRod can consistently outperform MLIRSmith. The results demonstrate the necessity of considering the dependencies between operation instances in MLIRod for fuzzing the MLIR compiler infrastructure. Besides, we found that MLIRod_2 (i.e., MLIRod) performs better than both MLIRod_1 and MLIRod_3, showing that $d = 2$ seems to achieve a balance between effectiveness and efficiency. Specifically, MLIRod_2 collects more sufficient OD information than MLIRod_1, and meanwhile it spends less time on coverage collection than MLIRod_3, leading to generating more programs for fuzzing during the given time. Hence, we recommend $d = 2$ as the default setting in MLIRod for practical use.

4.6 Threats to Validity

The threat to *internal* validity mainly lies in the implementation of MLIRod. To reduce this kind of threat, two authors have carefully checked all source code and written unit tests for guaranteeing the correctness. Regarding the compared technique (i.e., MLIRSmith), we directly adopted the released implementation and the recommended settings. The threat to *external* validity mainly lies in the used subject. Here, we adopted the latest revision (at the time of starting our experiments) for comparisons between MLIRod and both MLIRSmith and the variants of MLIRod, even though there are many versions for the MLIR compiler infrastructure. This is because it is helpful to detect previously unknown bugs by fuzzing the latest revision, which tends to be more significant following the existing work [39]. To further reduce this kind of threat, we also performed continuous fuzzing with MLIRod on more revisions for longer time (i.e., 50 days) as presented in Section 4.1, and our results indeed demonstrate that MLIRod can continuously detect new bugs on these studied revisions. The threat to *construct* validity mainly lies in parameter settings in MLIRod. To reduce this kind of threat, we have presented the specific settings of all the parameters in MLIRod for future replication. Moreover, we empirically investigated the influence of one important parameter (i.e., *d*) in MLIRod in RQ4 and left the investigation on other parameters as our future work due to the significant fuzzing cost on such experiments. Particularly, we took 50 randomly-generated programs by MLIRSmith without special selection as MLIRod's seeds in our study. As a mutation-based fuzzer, MLIRod is supposed to follow the general conclusion that seeds could affect the effectiveness of mutation-based fuzzers [18]. We will evaluate MLIRod with different sets of seeds. Currently, our work has provided a set of effective seeds for practical use with MLIRod.

5 Discussion

Novelty in Using Program Dependencies for Fuzzing. Program dependency has been used in fuzzing [14, 22, 32], vulnerability detecting [29], code naturalness [46], and software debugging [21]. These existing fuzzing works utilizing data/control dependencies focused on library-API fuzzing. Their test cases are API sequences, which can be clearly mapped to library's functionalities. Therefore, during test case generation, they mainly focused on the sequences of invoked APIs and parameter values within APIs, which considers data dependencies to some degree. However, control dependencies are built for the target library rather than test cases, guiding to achieve higher branch coverage.

Different from them, we focus on fuzzing MLIR compilers, which has two new challenges. First, test cases for MLIR compilers are programs written according to MLIR's language constraints, making it difficult to map MLIR programs directly to MLIR compiler functionalities. For example, testing the dead-code-elimination optimization requires analyzing program structure and def-use relationships between variables, and determining whether certain branch conditions are unsatisfiable in MLIR programs. This mapping involves complex data and control dependencies. To solve it, MLIRod considers both types of dependencies and their various combinations in generating MLIR programs by designing OD patterns and corresponding mutation rules. Note that while existing graph-level mutations generally involve node and edge modifications, MLIRod's mutations are specifically designed to cover more OD patterns. Second, ensuring the validity of MLIR programs during mutation is challenging due to MLIR's language constraints. We thus designed corresponding mechanisms in mutations to ensure program validity. Particularly, we used MLIR-program dependencies to guide mutation, which prior library-API fuzzing works did not do.

Significance of MLIRod. While MLIRod is implemented to fuzz the MLIR compiler infrastructure, its significance is not limited to the single system. This is because many compilers (such as Flang [1] and IREE [3]) are built on top of it, fuzzing the MLIR compiler infrastructure can contribute to ensuring the quality and robustness of all compilers leveraging this infrastructure. That is, fuzzing the MLIR compiler infrastructure can have a broader impact than fuzzing a single compiler system.

Moreover, the idea of MLIRod could be generalized to the compilers for other programming languages by adjusting dependency patterns and mutation rules according to compilers' and languages' characteristics. It can be also generalized to other kinds of systems sharing similar characteristics in the form of test inputs to MLIR programs (especially operations in them). For instance, the test input for operating systems involves a set of system calls. Like operations, each system call includes its name, operand types, and result types, and these system calls can also involve complicated dependencies, even though system calls have different constraints and semantics from MLIR operations. Hence, transferring the idea of operation dependency graph/coverage and the associated mutation rules in MLIRod to exploit the dependencies between system calls may help improve the fuzzing effectiveness for operating systems.

Future Work. Both MLIRod and MLIRSmith take *crash* as the test oracle for fuzzing the MLIR compiler infrastructure. In fact, comparing the execution results of an MLIR program under different pass sequences is also a natural test oracle. However, they do not incorporate it since the MLIR programs generated by them may have undefined behaviors [26], leading to potential false positives under this execution-output-based test oracle. That is, the current MLIRod cannot detect the bugs that make MLIR compiler produce wrong code without crash. In the future, we can improve MLIRod by incorporating some mechanisms to identify and avoid undefined behaviors during program generation.

6 Related Work

Fuzzing has been widely studied to guarantee the quality of various software systems, such as compilers [9, 20, 30, 36, 45, 47], operating systems [16, 23, 34], and browser engines [8, 17, 27, 42]. Our work is related to compiler fuzzing.

The most related work to ours is MLIRSmith [39], which is the first technique to fuzz the MLIR compiler infrastructure. It belongs to *grammar-based fuzzing* and there are also some grammar-based fuzzing techniques for other types of compilers, e.g., Csmith [47] for C compilers, CLSmith [30] for OpenCL compilers, NNSmith [31] for deep learning compilers, and Skyfire [40] for JS compilers. Different from them, MLIRod is a mutation-based fuzzing technique guided by OD coverage, and it designs dependency-targeted mutation rules to efficiently explore the input space.

There are some *mutation-based compiler fuzzing* techniques [8, 13, 15, 17, 27, 35, 49]. For example, Le et al. [25] proposed semantic-preserving mutation rules (e.g., mutating dead code) for compiler fuzzing. Holler et al. [19] proposed LangFuzz to generate JS programs via program synthesis. Schumi et al. [35] designed semantic coverage based on language specification for guiding fuzzing, but we did not compare to it because it is based on the K Framework, which does not support MLIR. Wang et al. [42] proposed FuzzJIT to generate JS programs by mutating seed programs with JIT-related program elements and structures. These techniques were typically designed to generate high-level source programs as tests for specific compilers rather than the general compiler infrastructure. They are inapplicable to the MLIR compiler infrastructure due to lacking alignment in terms of data structure and semantics with MLIR. Even though MLIRod is a mutation-based technique, different from the existing ones, it designs a set of mutation rules associated to ODG for MLIR programs by carefully considering the data dependencies and control dependencies between MLIR operations.

In addition, MLIRod can be categorized as *coverage-guided fuzzing* by designing ODG and OD coverage corresponding to MLIR program characteristics. In the area of fuzzing, most of the existing coverage-guided fuzzing techniques take edge coverage as the guidance [6, 15, 28, 41, 44]. Among them, AFL (American Fuzzy Lop) [6], which generates test inputs by applying byte-level and token-level mutations to increase edge coverage, is the most representative one. Based on AFL, Wang et al. [41] proposed Superion to support the generation of structured test inputs. Wu et al. [44] proposed JIT-Fuzz, which employs edge coverage to guide program mutation for fuzzing JIT compilers. Our evaluation also compared OD coverage and edge coverage for guiding the fuzzing process on the MLIR compiler infrastructure. Our results demonstrate the superiority of MLIRod with OD coverage in fuzzing MLIR infrastructure. In particular, OD coverage is collected in a black-box manner, making MLIRod more practical (compared to the gray-box edge-coverage-guided fuzzing techniques).

7 Conclusion

We propose a mutation-based MLIR compiler infrastructure fuzzing technique, MLIRod, for better MLIR compiler infrastructure fuzzing. To generate diverse MLIR programs, MLIRod designs the ODG coverage to systematically take data and control dependence into consideration to evaluate the diversity of generated MLIR programs and recognize valuable MLIR programs. MLIRod also designs several mutation rules based on ODG to improve ODG coverage. MLIRod has detected 63 previously unknown bugs during 50 days fuzzing, 38/48 bugs in which have been fixed/confirmed by developers.

8 Data Availability

We released our tool MLIRod (totaling 11.5K lines of C++ code) and experimental data at our project homepage for experimental replication and practical use [4].

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant Nos. 62322208, 62232001, 62202324, and CCF Young Elite Scientists Sponsorship Program (by CAST). We thank the anonymous reviewers for their constructive suggestions to help improve the quality of this paper.

References

[1] 2023. Flang. https://github.com/llvm/llvm-project/tree/main/flang.

[2] 2023. gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[3] 2023. IREE. https://openxla.github.io/iree/.

[4] 2023. MLIRod. https://github.com/tju-chenyaosuo/MLIRod.

[5] 2023. TOSA Dialect. https://mlir.llvm.org/docs/Dialects/TOSA/.

[6] 2024. M. Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl.

[7] 2024. MLIR Pass Documentation. https://mlir.llvm.org/docs/Passes.

[8] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 351–364. https://doi.org/10.1145/3548606.3560624

[9] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36. https://doi.org/10.1145/3363562

[10] Junjie Chen and Chenyao Suo. 2022. Boosting compiler testing via compiler optimization exploration. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–33. https://doi.org/10.1145/3508362

[11] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. https://doi.org/10.1007/3-540-12925-1_33

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[13] Tianchang Gao, Junjie Chen, Yingquan Zhao, Yuqun Zhang, and Lingming Zhang. 2023. Vectorizing Program Ingredients for Better JVM Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 526–537. https://doi.org/10.1145/3597926.3598075

[14] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs. In *Proceedings of the 44th International Conference on Software Engineering*. 1070–1081. https://doi.org/10.1145/3510003.3510228

[15] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://doi.org/10.14722/ndss.2023.24290

[16] Yu Hao, Hang Zhang, Guoren Li, Xingyu Du, Zhiyun Qian, and Ardalan Amiri Sani. 2022. Demystifying the Dependency Challenge in Kernel Fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 659–671. https://doi.org/10.1145/3510003.3510126

[17] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. 2021. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2229–2242. https://doi.org/10.1145/3460120.3484823

[18] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed selection for successful fuzzing. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 230–243. https://doi.org/10.1145/3460319.3464795

[19] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458.

[20] He Jiang, Zhide Zhou, Zhilei Ren, Jingxuan Zhang, and Xiaochen Li. 2021. CTOS: Compiler testing for optimization sequences of LLVM. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2339–2358. https://doi.org/10.1109/TSE.2021.3058671

[21] Jiajun Jiang, Yumeng Wang, Junjie Chen, Delin Lv, and Mengjiao Liu. 2023. Variable-Based Fault Localization via Enhanced Decision Tree. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–32. https://doi.org/10.1145/3624741

[22] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust library fuzzing via API dependency graph traversal. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 581–592. https://doi.org/10.1109/ASE51524.2021.9678813

[23] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://doi.org/10.14722/ndss.2020.24018

[24] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[25] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226. https://doi.org/10.1145/2666356.2594334

[26] Bastien Lecoeur, Hasan Mohsin, and Alastair F Donaldson. 2023. Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1801–1825. https://doi.org/10.1145/3591294

[27] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2613–2630.

[28] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 475–485. https://doi.org/10.1145/3238147.3238176

[29] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258. https://doi.org/10.1109/TDSC.2021.3051525

[30] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76. https://doi.org/10.1145/2737924.2737986

[31] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543. https://doi.org/10.1145/3575693.3575707

[32] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with data dependency information. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 286–302. https://doi.org/10.1109/EuroSP53844.2022.00026

[33] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 45–59. https://doi.org/10.1109/PACT52795.2021.00011

[34] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 729–743.

[35] Richard Schumi and Jun Sun. 2021. SpecTest: Specification-Based Compiler Testing. In *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12649)*, Esther Guerra and Mariëlle Stoelinga (Eds.). Springer, 269–291. https://doi.org/10.1007/978-3-030-71500-7_14

[36] Mayank Sharma, Pingshi Yu, and Alastair F Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1483–1486. https://doi.org/10.1145/3597926.3604919

[37] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

[38] Nicolas Vasilache, Oleksandr Zinenko, Aart JC Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, et al. 2023. Structured Operations: Modular Design of Code Generators for Tensor Compilers. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 141–156. https://doi.org/10.1007/978-3-031-31445-2_10

[39] Haoyu Wang, Junjie Chen, Chuyue Xie, Shuang Liu, Zan Wang, Qingchao Shen, and Yingquan Zhao. 2023. MLIRSmith: Random Program Generation for Fuzzing MLIR Compiler Infrastructure. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1109/ASE56229.2023.00120

[40] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 579–594. https://doi.org/10.1109/SP.2017.23

[41] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[42] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 1865–1882.

[43] Zan Wang, Yingquan Zhao, Shuang Liu, Jun Sun, Xiang Chen, and Huarui Lin. 2019. Map-coverage: A novel coverage criterion for testing thread-safe classes. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 722–734. https://doi.org/10.1109/ASE.2019.00073

[44] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 56–68. https://doi.org/10.1109/ICSE48619.2023.00017

[45] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. *Proc. IEEE/ACM ICSE* (2024). https://doi.org/10.1145/3597503.3639121

[46] Chen Yang, Junjie Chen, Jiajun Jiang, and Yuliang Huang. 2024. Dependency-aware code naturalness. *Proceedings of the ACM on Programming Languages* OOPSLA2 (2024).

[47] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. https://doi.org/10.1145/1993316.1993532

[48] Hongbin Zhang, Mingjie Xing, Yanjun Wu, and Chen Zhao. 2023. Compiler Technologies in Deep Learning Co-Design: A Survey. *Intelligent Computing* (2023). https://doi.org/10.34133/icomputing.0040

[49] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1133–1144. https://doi.org/10.1145/3510003.3510059

[50] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021), 110884. https://doi.org/10.1016/j.jss.2020.110884

Received 2024-04-12; accepted 2024-07-03