# Directed Testing in MLIR: Unleashing Its Potential by Overcoming the Limitations of Random Fuzzing

WEIYUAN TONG, Northwest University, China
ZIXU WANG, Northwest University, China
ZHANYONG TANG*, Northwest University, China
JIANBIN FANG, National University of Defense Technology, China
YUQUN ZHANG, Southern University of Science and Technology, China
GUIXIN YE, Northwest University, China

MLIR is a new way of creating compiler infrastructures that can be easily reused and extended. Current MLIR fuzzing methods focus primarily on test case generation or mutation using randomly selected passes. However, they often overlook the hierarchical structure of MLIR, resulting in inefficiencies in bug detection, especially for issues triggered by downstream dialects. Random testing lacks a focused approach to exploring the code space, resulting in wasted resources on normal components and overlooking bug-prone areas. To address these limitations, we introduce MLIRTracer, a top-down fuzzing approach that targets the highest level of MLIR programs (i.e., tosa IR) with a directed testing strategy. Our method systematically traverses the hierarchical code space of MLIR, from tosa IR to the lower levels, while prioritizing tests of bug-prone areas through directed exploration. MLIRTracer has successfully detected 73 bugs, with 61 already resolved by MLIR developers.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Fuzzing, Compiler Testing, Multi-Layer Intermediate Representation

**ACM Reference Format:**
Weiyuan Tong, Zixu Wang, Zhanyong Tang, Jianbin Fang, Yuqun Zhang, and Guixin Ye. 2025. Directed Testing in MLIR: Unleashing Its Potential by Overcoming the Limitations of Random Fuzzing. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE102 (July 2025), 23 pages. https://doi.org/10.1145/3729372

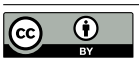
## 1 Introduction

MLIR [38] (Multi-Layer Intermediate Representation) is an emerging compiler framework that provides unprecedented extensibility and generality, holding great promise in facilitating the rapid implementation of domain-specific compilers. With its modular and reusable design, MLIR enables support for a wide variety of programming languages/models and target hardware using the same intermediate representation (IR) infrastructure capable of handling various levels of abstraction.

The emergence of MLIR steers the development of compilers toward greater modularity, efficiency, and customizability, benefiting various fields. It has since inspired numerous downstream projects [20] and become a popular solution for developing deep-learning compilers [7, 13, 18, 34].

*Zhanyong Tang is the corresponding author.

Authors' Contact Information: Weiyuan Tong, Northwest University, Xi'an, China, wytong@stumail.nwu.edu.cn; Zixu Wang, Northwest University, Xi'an, China, wangzixu1@stumail.nwu.edu.cn; Zhanyong Tang, Northwest University, Xi'an, China, zytang@nwu.edu.cn; Jianbin Fang, National University of Defense Technology, Changsha, China, j.fang@nudt.edu.cn; Yuqun Zhang, Southern University of Science and Technology, Shenzhen, China, zhangyq@sustech.edu.cn; Guixin Ye, Northwest University, Xi'an, China, gxye@nwu.edu.cn.




ACM 2994-970X/2025/7-ARTFSE102
https://doi.org/10.1145/3729372

Additionally, MLIR's versatility is reflected in its growing adoption for addressing domain-specific problems, such as circuit compilation [4], data processing [19], quantum compilation [3], high-performance computing [15, 44], and formal verification [2].

A key property of MLIR is its ability to represent programs at multiple levels of abstraction (i.e., dialects [6]), which enables the compiler to optimize at various levels. This capability enhances flexibility, allowing developers to customize the compilation pipeline across these dialects to meet their specific requirements. However, it also introduces a higher risk of faults. The compilation and optimization stack within the MLIR framework often involves complex interactions between various transformation passes operating at different levels of abstraction. This makes MLIR susceptible to compatibility issues, leading to unexpected and possibly harmful misbehavior. In particular, given its foundational role, ensuring the correctness and reliability of the MLIR infrastructure is crucial, as any bug within MLIR could impact domain-specific compilers built on top of it. Hence, it is essential to investigate effective methods for testing the MLIR infrastructure.

MLIR's modularization and hierarchical code space present significant challenges to finding bugs. In contrast, previous deep learning (DL) compilers were relatively easy to test due to their fixed levels of abstraction and lowering processes (e.g., TVM [25], which lowers from the Relay IR to the TIR and then to the LLVM IR). Unlike TVM, MLIR provides dozens of dialects and supports the creation of a hybrid IR that combines operations from different dialects. Recent efforts (i.e., MLIRSmith [53] and MLIRod [50]) on MLIR testing focus on generating such hybrid MLIR programs to detect bugs. Specifically, MLIRSmith directly constructs MLIR programs with 12 dialects based on the template, while MLIRod further mutates such MLIR programs to serve as test input. However, designing hybrid IRs manually, based on the semantics of multiple dialects, is highly impractical due to the diversity and rapid evolution of dialects. Additionally, grammar-based generation methods are limited in test case structure [45] and fail to cover all possible hybrid IR patterns adequately. Despite these approaches, bugs still exist in MLIR infrastructure, as we have found in this work. The primary reason is that both approaches rely on applying random sequences of transformation passes to MLIR programs, which neglect the exploration of MLIR's hierarchical structure [11] and fail to fully exercise the code.

A promising way is to employ a top-down testing strategy that starts with high-level IR and lowers it to the lowest level along various dialects through MLIR's existing incremental lowering capabilities. This process naturally yields a range of hybrid IRs covering multiple dialects, helping to uncover bugs in dialect-specific transformations. One primary challenge is *how to generate test cases that satisfy the top-down testing requirements of MLIR?* At present, the abstraction of the machine learning graph (i.e., tosa[1] dialect) is the highest level, which can be progressively lowered to other lower dialects using various conversions provided by MLIR. Generating such IR is not a difficult task, as it is solely based on the tosa operations and considers grammar targeting for only one dialect. However, to ensure extensive test coverage, creating graph-like data structures requires advanced algorithmic support. If a graph is too simple, it fails to conduct comprehensive fuzzing due to a lack of diversity [31, 40].

The second challenge in MLIR fuzzing is *how to efficiently uncover bugs within MLIR's hierarchical structure space?* Existing MLIR fuzzing approaches based on random testing are inefficient because many transformations are simply neglected by the compiler [35, 42], as confirmed in our study (Section 5.4). In MLIR, dialect-specific passes are designed to operate only on specific types or operations, making it crucial to consider the dependencies between the IR and the passes [12, 14]. Additionally, phase-ordering of lowering passes [36, 51] is crucial, as the output of the preceding IR conversion serves as the context for the subsequent conversions, directly influencing the execution

[1]Tensor Operator Set Architecture. https://www.mlplatform.org/tosa

path. Therefore, the lowering passes must be arranged in an order so that the fuzzer can effectively explore MLIR's hierarchical structure space from top to down. Moreover, exposing bugs more quickly remains challenging without proper guidance. Given that MLIR supports flexible compilation paths [23, 24, 36, 51], testing MLIR is equivalent to testing innumerable customized compilers, each with a different compilation stack. Exhaustively enumerating all compilation paths for fuzzing is impractical. Therefore, a more efficient and effective strategy should direct fuzzing efforts toward modules that are more likely to contain bugs.

This paper presents MLIRTracer, a top-down fuzzing approach for MLIR that starts with high-level MLIR programs (i.e., tosa IR) to explore MLIR's hierarchical structure, while employing a directed strategy to enhance fuzzing efficiency. MLIRTracer first generates a diverse set of tosa IRs using a tosa IR generator. The compilation stack for tosa IR spans multiple MLIR dialects, facilitating the application of a wide range of dialect-specific components. The core insight behind top-down testing lies in MLIR's design philosophy of progressive lowering [38]. MLIRTracer progressively lowers tosa IR in multiple small steps, adhering to the dependency-based lowering rules of MLIR. This ensures that the lowering process follows the order imposed by the MLIR framework. Directedness is achieved by lowering tosa IR toward dialects that are more prone to bugs, thus prioritizing the testing of components specific to these dialects. This targeted approach helps identify bugs earlier in the conversion and transformation phases by testing areas more likely to expose potential issues. In short, we propose a top-down directed strategy tailored for MLIR testing, guiding the fuzzer to focus on the vulnerable areas within the hierarchical code space, thereby improving the overall efficiency of MLIR fuzzing. We also develop two mutation methods, *mixing IR mutation* and *similarity operator replacement*, which aim to produce interesting patterns of hybrid IR and improve the coverage of operations, resulting in a more extensive test space.

In summary, our contributions are as follows:

- **Idea.** We introduce MLIRTracer, to the best of our knowledge, the first MLIR fuzzing framework that systematically explores MLIR's hierarchical code space following its design philosophy, while achieving directedness to enhance fuzzing efficiency.
- **Technique.** MLIRTracer primarily comprises a test case generator designed to create high-level MLIR programs, along with a top-down directed testing strategy. This approach facilitates top-down fuzzing across multiple abstraction levels and guides the fuzzer to target areas prone to bugs.
- **Evaluation.** We conduct an extensive study to demonstrate the effectiveness of MLIRTracer. Our results show that MLIRTracer detects 2.6× more bugs and achieves 1.4× code coverage than the state-of-the-art MLIR fuzzing techniques. Additionally, MLIRTracer successfully detects 73 previously unknown bugs, 61 of which have already been fixed by the MLIR developers.

## 2 Background and Motivation

### 2.1 MLIR Preliminaries

MLIR is a reusable and extensible compiler infrastructure and a subproject of the LLVM project [8], designed for building domain-specific compilers and bridging compilers for different programming languages. Like LLVM, MLIR uses traditional three-address *SSA* (Static Single Assignment) to encode its multi-level IR. The MLIR framework consists of two key modular components: dialect and pass system. Dialects are used to define and represent specific operations within IR, i.e., MLIR program. *Passes* are employed to implement various transformations or optimizations on these IRs.

**Dialects.** Each level of abstraction can be represented as a dialect, which is a mechanism to engage with and extend MLIR ecosystem. MLIR provides a range of built-in dialects to represent different

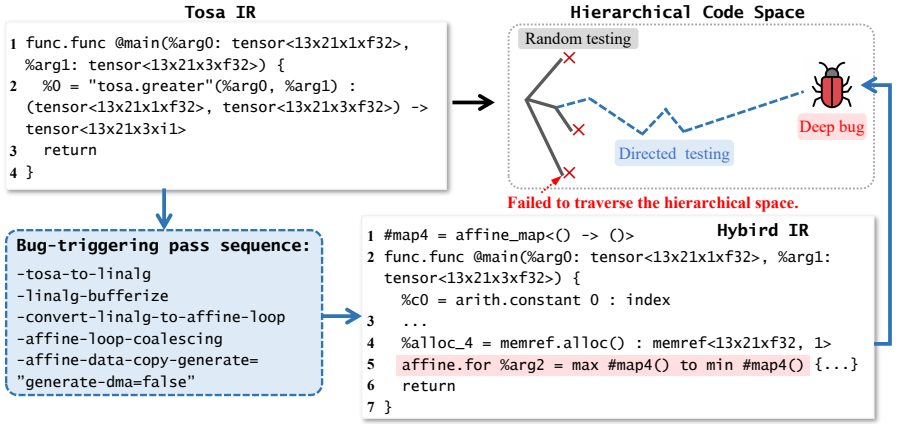



Fig. 1. **Motivation Example.** Existing methods, represented by the black arrow, perform random fuzzing without any knowledge of the MLIR compiler infrastructure internals. This results in inadequate guidance for exploring MLIR's hierarchical code space. In contrast, the MLIRTracer testing process, represented by the blue arrow, provides a more targeted and effective approach to overcome these limitations.

levels of abstraction, ranging from dataflow graphs to target-specific instructions and even hardware circuitry. Each dialect consists of a set of operations and types, and is given a unique namespace. For example, the affine dialect contains operations that represent polyhedral structures such as *for* and *map*. Compatibility issues may arise when converting between dialects, as operations in one dialect may not have direct counterparts in another. Additionally, certain types in one dialect may not be supported in another due to differences in the set of supported types.

**Operations.** Operation is the basic unit of semantics. Each operation is identified by a unique string (e.g. *tosa.conv2d*, *affine.parallel*, *vector.load*, etc.). An operation has a list of *SSA* operands, may have attributes that store static information, and has zero or more results. Operations are defined using a TableGen-based [16] specification for an operation descriptor. For example, the *tosa.matmul* operation is defined to take two 3D tensors as inputs and return a 3D tensor of the same type as the output. The creation of the operation must comply with such constraints imposed by MLIR.

**Passes.** MLIR supports a wide range of passes that operate on the IR to enable progressive conversion and optimization from higher to lower levels of representation. These include conversion passes, which facilitate seamless transitions between dialects, and transformation passes, which optimize operations within specific dialects or across the entire IR. MLIR enables the flexible composition of these passes into compilation pipelines for code generation. In a pass pipeline, each pass iteratively transforms the IR, producing a new representation at each step.

**Hybrid representation.** MLIR allows for separate dialects to co-exist together forming a hybrid program representation. Such hybrid IRs typically emerge during the intermediate stages of the compilation pipeline, where high-level dialects are progressively lowered into lower-level representations. For example, a program may begin in a high-level dialect, such as *tensor* or *affine*. Through a series of dialect conversions, certain operations may be lowered to a low-level dialect like *llvm* or *spirv*, while others remain at a higher level.

**MLIR fuzzing.** The introduction of new key features in MLIR poses challenges for fuzzing. In particular, the hybrid representation makes it difficult to achieve comprehensive and valid test case generation. Additionally, the flexibility and complexity of the multi-level compilation process create further difficulties, primarily due to the lack of guidance.

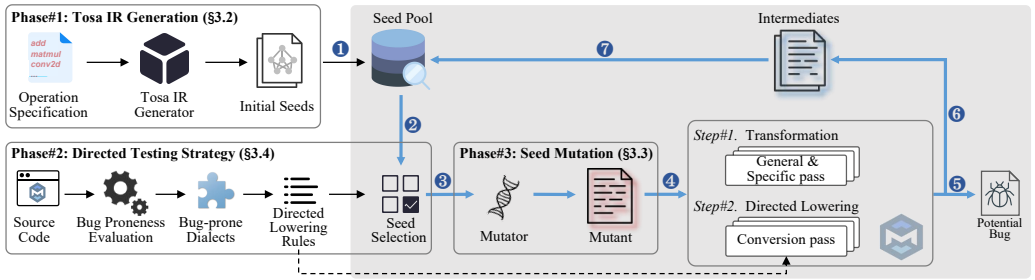



Fig. 2. Overview of MLIRTracer.

## 2.2 Motivation

Figure 1 presents a motivating example of a bug in MLIR detected by MLIRTracer. The input tosa IR (shown in the top-left corner) exposes a bug under a specific pass sequence. In particular, this tosa IR itself is a simple IR with only one operation and contains no bug. However, it can be lowered to the hybrid IR[2] (shown in the bottom-right corner of the figure), which directly triggers the bug. The cause of this bug is the inaccurate calculation of lower and upper bounds in the "-affine-data-copy-generate" pass, which results in empty loop bound maps (line 1 in the hybrid IR). Hence, the MLIR compiler crashes when the "-lower-affine" pass works on an empty loop bound (line 5 in the hybrid IR). It is worth noting that, in practical scenarios, lowering IR into the hybrid IR shown in Figure 1 is common for enabling loop optimizations based on affine representations, thereby facilitating efficient code generation.

This example illustrates the challenge of detecting such bugs, as it can be difficult to connect the high-level intent of the code with the low-level issues that may arise during compilation. When using fuzzing techniques to test the MLIR infrastructure, the main challenge is how to effectively explore MLIR's hierarchical code space to uncover complex bugs. Current methods typically perform random fuzzing that expends too much energy revolving around the initial MLIR program and is unable to find bugs triggered by its downstream dialects. Furthermore, random testing lacks direction in exploring the code space. This can result in the fuzzer not being able to dive deep into critical areas of code that are more likely to contain bugs. This motivates us to design an effective strategy in our fuzzing technique to 1) systematically explore MLIR's hierarchical code space from top to down, and 2) implement the directedness, enabling the fuzzer to prioritize and focus on the critical code areas that are most prone to bugs.

# 3 MLIRTracer Design

## 3.1 Overview

Figure 2 presents an overview of MLIRTracer, which generates high-level MLIR programs (i.e., tosa IR), and applies a top-down directed testing strategy starting from the tosa IR, to prioritize testing bug-prone areas. MLIRTracer constructs a tosa IR generator (Sec. 3.2) to generate the tosa IRs for initializing the seed pool. During the progressive lowering of tosa IR down to the lowest level, all dialects lower than tosa are covered. MLIRTracer introduces a directed testing strategy (Sec. 3.3), enabling lowering tosa IR towards bug-prone dialects (called directed lowering). These bug-prone dialects are identified by evaluating the bug proneness of MLIR source code using an offline machine learning model. Furthermore, specialized lowering rules are built that take into account the dependency between dialect and conversion pass. This enables targeted seed

[2]For brevity, we omit constant declarations and memory allocation IRs. The full example can be found in our repository [21].

selection for bug-prone dialects while ensuring normal directed lowering. For each selected seed, MLIRTracer schedules mutator (Sec. 3.4) and then performs a two-step test on the mutant: lowering it following the directed lowering rule and performing general or dialect-specific transformations. During this process, if the compiler hangs or crashes, it indicates a potential bug that will then be reported to the developers; the resulting *intermediates*, i.e., transformed IRs and lowered IR, will be stored as optional seeds in the seed pool.

### 3.2 Tosa IR Generation

Considering the tosa dialect as the highest-level dialect, we take tosa IR as the initial seed and leverage MLIR's existing conversion infrastructure to progressively lower it. This enables the tests to cover dialects that are at a lower level of abstraction than tosa dialect. To this end, we implement a generator for tosa dialect, generating initial seeds to support top-down testing for MLIR. This approach eliminates the need to develop individual test generators for each dialect, which would be time-consuming and labor-intensive given the existence of over 40 dialects.

Tosa IR contains a sequence of tosa operations that operate on high-dimensional tensors. Since MLIR's verifiers automatically perform strict static checks during compile time, the operation's inputs, attributes, and results must adhere to the restrictions imposed by MLIR [5]. To ensure the operation can be created correctly, we extract the operation specification. In particular, we create "invalid" operations by breaking operation constraints to test the robustness of MLIR. Moreover, we propose an incremental graph generation strategy based on model structures that are widely used in the real world, which is inspired by the prior generators in DL libraries or compiler testing [31, 40]. These commonly used model structures (i.e., the chain structure with skips and multi-branch structure [28, 48]) provide flexibility in accommodating various graph topologies and generate a more realistic graph-level representation. Additionally, incremental graph generation, i.e., creating random tosa operations one by one, allows for a greater diversity of tosa IRs than the method of constructing a structure and then expanding the skeleton.

For tosa IR generation, MLIRTracer randomly selects an operator every time to create a corresponding operation and incrementally introduces it into the graph following commonly used model structure topologies. The details are provided below:

**Extracting Operation Specification.** We extract the specifications for each tosa operation, i.e., a tuple of constraints on operands, attributes, and results, such as their shapes and types. This is done by parsing the TableGen file of tosa operations [17], which defines the operation set for the tosa dialect. Furthermore, the tensor type and shape of the operation's result need to be explicitly specified. Therefore, we implement a result inference function for each operator that derives the tensor format of the result based on its inputs and attributes.

**Creating Operation.** By utilizing the operation specification, each operation is created strictly following the pre-defined type and shape constraints, thereby avoiding failures during tosa IR generation. Additionally, to test MLIR's exception-handling capabilities, MLIRTracer deliberately selects dimensions, types, or attribute values that fall outside the operation specification to create operations (e.g., providing a 2D tensor to an operator that only accepts 4D tensors).

**Constructing the Tosa IR.** We formulate a tosa IR as a Directed Acyclic Graph (DAG) and describe its construction process using graph theory, which is in line with prior work [31, 43]. Specifically, the nodes in the tosa IR represent a collection of tosa operations (i.e., an instance of the tosa operator), and the edges are the connections between these operations, representing the data dependency. In constructing a tosa IR, MLIRTracer incrementally inserts random operators through three different methods: creating a new branch, tail insertion, and random insertion. Among these ways, the first two are used to widen and deepen the graph, resulting in either a single chain or a more

**Algorithm 1:** Tosa IR Generation

```
   Input: OpPool: set of tosa operators
          ODS: operation definition specification
          OpNum: number of nodes in a TG
1  TG ← {}                                                          ▷ tosa graph
2  T ← {}                                                           ▷ set of tail nodes in TG
3  while TG.size < OpNum do
4      op ← randomSelect(OpPool)
5      CompatibleNode ← typeMatch(TG, ODS[op].input)
6      if CompatibleNode does not exist then
7          node ← createNewBranch(TG, op)                           ▷ Creating new branch
8          updateTail(node)
9      else
10         T' ← CompatibleNode ∪ T
11         if T' is not empty then
12             v ← randomSelect(T')
13             node ←NodeInsert(op, v, CompatibleNode)              ▷ Tail insertion
14             updateTail(node)
15         else
16             v ← randomSelect(CompatibleNode)
17             node ← NodeInsert(op, v, CompatibleNode)             ▷ Random insertion
18         end
19     end
20 end
21 Function NodeInsert(op, v, C):
22     addConnection(op, v)
23     if ODS[op].input_num > 1 then
24         v' ← getCompatibleInputs(v, C)
25         addConnection(op, v')
26     end
```

complex multi-branch architecture to simulate typical DL models in the real world. The last way increases the complexity of the graph to provide additional testing opportunities.

Algorithm 1 presents our tosa graph generation algorithm. The generator maintains a pool of all tosa operators. When constructing a tosa IR, the generator randomly selects an operator *op* from the operator pool (line 4) and inserts it into the graph until the number of nodes reaches the preset value of *OpNum*. To insert *op* compatibly, the generator first identifies a compatible node (i.e., operation) in the graph whose result satisfies the input constraints of *op*. To do this, the generator uses a simple type match to pick out all feasible nodes *CompatibleNode* (line 5). For example, when inserting the *tosa.matmul* operator, all nodes whose result is a rank-3 tensor are considered as candidates.

If no *CompatibleNode* is found, it indicates that no node can serve as the producer for *op* due to the violation of *op*'s input constraint. In such cases, *op* is inserted without being connected to any existing node, which lets it be an initial node of a new branch (lines 7-8). If *CompatibleNode* includes a tail node, one of the tail nodes is randomly selected as the producer for *op*; subsequently, the newly inserted node is updated as the tail node in the current branch (lines 12-14). If no tail node is available, the generator inserts *op* by selecting a randomly compatible node as its input (lines 16-17). Note that both tail insertion and random insertion rely on an existing node in the graph. For *op* that only requires a single input (e.g., *tosa.cast(a)*), one input edge from node *v* suffices. However, for *op* with multiple operands (e.g., *tosa.matmul(a,b)*), additional input edges are required to satisfy constraints on the number of inputs. To do this, we randomly choose a compatible node (or a placeholder referring to the passed parameter of the *FuncOp*) as *op*'s additional input (lines 24-25).

Table 1. Features used to represent functions

| | Feature | Definition |
|---|---|---|
| Code static features | CountLine | The total lines of code |
| | CountStmt | The number of statements |
| | CommentRatio | Lines of comments / total lines of code |
| | Cyclomatic | The cyclomatic complexity of a function |
| | NestDepth | The nested block depth |
| Project evolution feature | Numfixes | The cumulated number of bug fixed |
| | NumChanged | The cumulated number of change |
| | NumDevelopers | The number of developers |

Table 2. Bug-prone dialects

| ID | Dialect | Abstraction | ID | Dialect | Abstraction |
|---|---|---|---|---|---|
| 1 | affine | polyhedral structures | 7 | memref | memref creation and manipulation |
| 2 | tosa | tensor operator set architecture | 8 | bufferization | converting tensor to memref |
| 3 | async | asynchronous programming | 9 | vector | vector creation and manipulate |
| 4 | scf | structured control flow | 10 | spirv | SPIR-V IR |
| 5 | linalg | linear algebra operation | 11 | tensor | tensor creation and manipulation |
| 6 | gpu | GPU kernel invocations | | | |

### 3.3 Top-down Directed Testing

Our goal is to perform a top-down fuzzing approach to improve random testing on MLIR. To make this process more efficient, we propose a directed testing strategy that enhances the directedness of reaching bug-prone dialects. In particular, we employ bug prediction technology for bug proneness evaluation to identify bug-prone dialects. Then, we guide the fuzzer to focus tests on these bug-prone dialects and related components.

*3.3.1 Bug-prone Dialects Identification.* Our methodology involves assessing the bug proneness of MLIR source code using bug prediction technology [29, 33, 46], followed by determining the bug proneness at the dialect level.

**Bug Proneness Evaluation.** We utilize a machine learning model trained on historical defect data for bug-proneness evaluation. Specifically, our approach begins with automated crawling and retrieval of code commits via GitHub APIs over the past six months. We extract modified code segments, i.e., functions or methods, and label them as buggy or non-buggy based on keywords in the commit messages related to bug fixes, as done in previous studies [30, 32, 37]. Note that the extracted code segments are exclusively derived from the official MLIR project repository [10] and do not include any other LLVM subprojects. We represent these code segments using two types of features (see Table 1): static code features and project evolution features, which together form a feature vector. Static code features capture the complexity of the MLIR implementation, while the inclusion of project evolution features is motivated by two observations: frequent bug fixes often introduce new bugs [39], and the fragmented contributions from a large number of developers (over 350 in MLIR project) can negatively impact software quality [22, 47].

For the bug proneness evaluation task, we built a machine-learning model using a random forest algorithm. The model is trained offline using 15,490 labeled training samples. The trained model is subsequently applied to predict the bug proneness of each function in the new version, producing a probability ranging from 0 to 1, which indicates the likelihood of containing a bug. Finally, we obtain the bug-proneness score for all functions.

**Identifying Bug-prone Dialects.** MLIR employs a pass-driven architecture to optimize and transform code. Each pass is implemented by multiple functions, where subtle errors in a function can cause the pass that invokes it to fail. Moreover, most transformations are dialect-specific, operating on operations within a particular dialect. As a result, we further assess dialect proneness to identify bug-prone dialects.

```
{
"dialect": "tosa",
"dependency":[
  {"OPS":["tosa.const", "tosa.applyscale"],
   "PASS":["-tosa-to-arith"]},
  {"OPS":["tosa.conv2d","tosa.matmul", ...],
   "PASS":["tosa-to-linalg-named"]},
  {"OPS":["tosa.add","tosa.sub","tosa.mul", ...],
   "PASS":["tosa-to-linalg"]}
  ]
}
```

Fig. 3. An excerpt of lowering rule.

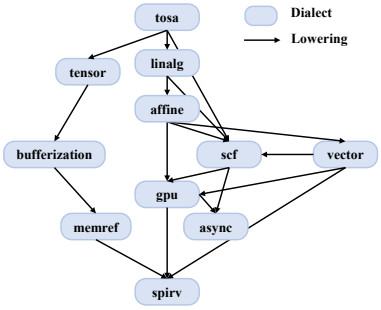



Fig. 4. Illustration of directed lowering.

We first analyze the function call chain and trace it to identify the passes that can trigger a given function. Since obtaining a complete set of function callers through dynamic analysis is challenging, we instead perform static call chain analysis. In this step, we parse all files in MLIR projects using ANTLR 4 [1] and construct an abstract syntax tree (AST) containing all function nodes. We then use a graph traversal algorithm to trace the call sequence of target nodes (i.e., bug-prone functions) in the AST. The traversal starts at the target node and moves up to its parent node (i.e., the caller) until the root node (i.e., the pass that can trigger it) is found.

Next, we calculate the dialect's bug-proneness score based on scores on function-level scores using the following formula:

$$P_d = \sum_{j=0}^{N-1} \left( \sum_{i=0}^{M_j-1} f_i \right) \text{if } f_i > \sigma \tag{1}$$

Given a dialect $d$, we assume that there are $N$ passes specific to it, with each pass consisting of $M_j$ functions. The threshold $\sigma$ is used to identify which functions are considered bug-prone. The scores of these bug-prone functions contribute to the bug-proneness score of their associated passes, and ultimately to the bug-prone dialect. Dialects with a bug-proneness score greater than 0 are identified as bug-prone, as shown in Table 2. In this paper, we set the $\sigma$ to 0.7 (discussed in Section 6). Note that $\sigma$ is a configurable threshold based on available resources. A higher value of $\sigma$ prioritizes a narrow set of high-risk targets, thereby maximizing bug detection efficiency within a limited time. In contrast, reducing the threshold allows for the inclusion of components, even with moderate risks.

*3.3.2 Top-down Directed Strategy.* From tosa IR down to the lowest level, progressive lowering is performed along multiple abstraction levels, allowing various dialects to be covered. MLIR provides a set of available passes used for dialect conversion (i.e., lowering). The input IR and conversion pass accepted by the MLIR compiler must have dependencies between them; otherwise, the passes will be ignored by the MLIR compiler. This is because the conversion pass is driven by a set of specific operations within a particular dialect. However, such dependencies are ignored by the existing works, which apply randomly selected passes to the input IR.

**Dependency-guided Lowering.** To ensure the orderly lowering of tosa IR, we extract these dependencies, which can guide the selection of relevant conversion passes to execute dialect conversion, rather than applying random passes. We observe that the conversion pass is implemented by composing multiple match and rewrite patterns, each of which defines how to convert one operation or sequence of operations into another. Therefore, by automatically parsing these patterns in the source code to identify which operations they are responsible for, we can extract lowering rules

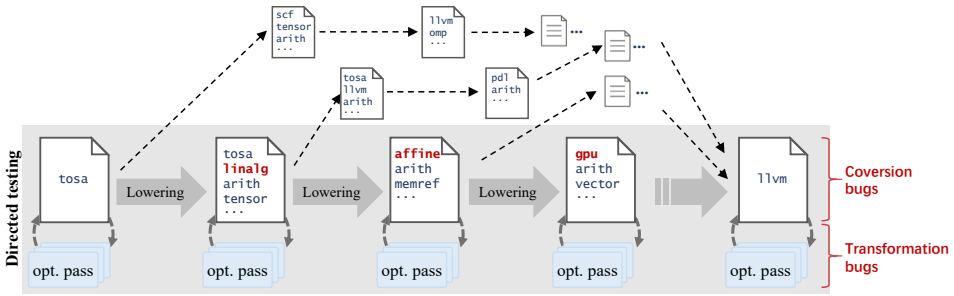



Fig. 5. The foundation of MLIRTracer is based on a top-down dependency strategy that systematically explores the hierarchical code space. Building on this, a directed strategy enhances test directionality, enabling rapid coverage of bug-prone dialects and related components.

with dependencies, called general lowering rules (GLR). Figure 3 shows part of the dependency lowering rules for operations in the tosa dialect in JSON format. Each dependency rule consists of a pass and a set of operations for which that pass is responsible. For example, if *tosa.const* or *tosa.applyscale* appear in a given IR, they can be lowered to arith dialect using *tosa-to-arith* pass, while other operations remain.

**Directed Lowering.** Based on these above general lowering rules, it can ensure "top-down" testing of MLIR, but it still involves a random exploration for all dialects due to a lack of directionality in reaching bug-prone dialects. In particular, a dialect has several different conversion passes that convert it to various downstream dialects. For a given IR, there are numerous compilation paths, making it difficult to enumerate all for fuzzing. Hence, we construct a set of directed lowering rules (DLR), which enable lowering tosa IR to cover all bug-prone dialects. We construct directed lowering rules for all bug-prone dialects (see Table 2). We first pick up dependency rules that are used to produce bug-prone dialects by examining whether the target of the conversion pass is a bug-prone dialect. In total, we extracted 59 general lowering rules, with 17 used to construct the directed lowering rules. These rules are then arranged in descending order of the dialect's abstraction level. Figure 4 shows the directed lowering along 11 bug-prone dialects. Tosa IR will be lowered towards these bug-prone dialects, enabling MLIRTracer to test specific transformations on them. In summary, with the directed lowering rules, MLIRTracer can focus on the code areas that are most likely to expose bugs, offering higher efficiency compared to random exploration.

*3.3.3 Fuzzing Process.* Figure 5 illustrates the basic idea of MLIRTracer. To address the multi-layered structure of the MLIR framework, we propose a top-down directed strategy. This approach systematically explores MLIR's code space by guiding the test path to comprehensively cover the entire transformation process, from tosa IR to the lowest levels. With the directed strategy (highlighted in gray boxes), MLIRTracer efficiently covers bug-prone dialects to test high-risk components, enabling earlier detection of conversion and transformation bugs.

We formally describe MLIRTracer's fuzzing process in Algorithm 2. Within a time budget $T$, MLIRTracer iteratively performs directed testing (lines 2-5) and top-down random testing (lines 6-7). The directed testing strategy guides the fuzzer to focus on these high-risk areas, while random testing explores edge cases, ensuring comprehensive testing. The seed pool is initialized by creating a diverse set of tosa IRs (see Sec 3.2). During each iteration of directed testing, MLIRTracer first performs operation-aware seed selection by choosing a set of seeds containing the specific operations *r.ops* from the seed pool (line 4). For each selected seed $s$, MLIRTracer schedules mutation (line 11) to facilitate the diversity of the MLIR program (see Sec 3.4). The mutated seed $s'$ is then

**Algorithm 2:** The Workflow of MLIRTracer

```
   Input: S_0: the seed pool
          DLR: directed lowering rules
          GLR: general lowering rules
          T: time budget
1  while within time budget T do
2      foreach r ∈ DLR do
3          S ← selectSeeds(S_0, r.ops)
4          Execute(S,r)
5      end
6      S ← randomSeeds(S_0)
7      dependentTest(S, GLR)
8  end
9  Function Execute(S,r):
10     foreach s ∈ S do
11         s' ← Mutator(s)
12         IR_l ← directedLowering(s', r)
13         IR_t ← Transformation(s')
14         if ∃err then
15             bugReport()
16         else
17             updateSeedPool(IR_l, IR_t)
18         end
19     end
```

processed through the conversion pass in rule $r$ (line 12), producing lowered $IR_l$. Concurrently, MLIRTracer applies a randomized combination of general and dialect-specific transformation passes to $s'$ (line 13). If either process causes the compiler to crash, it indicates a potential bug, which will then be reported to the developers (line 15). The newly generated $IR_l$ and $IR_t$ will be stored as optional seeds in the seed pool (line 17). After a round of directed testing, MLIRTracer performs an additional round of testing based on the dependency lowering strategy (lines 6-7), where it randomly selects seeds, lowers them using only general lowering rules without bug-proneness guidance, and then applies random transformations.

### 3.4 Seed Mutation

Existing mutation strategies [42, 43, 55, 57] are often designed for certain specific IRs and do not benefit multi-level IRs in MLIR together. We argue that mutation strategies should consider MLIR's unique hierarchical design. During the lowering tosa IR in step ❶ of Figure 6, We find that certain operations may fail to achieve coverage, as they cannot be produced from upstream dialects due to missing corresponding conversion patterns in MLIR's built-in lowering facility. Since transformations in MLIR are implemented as a matching rewrite at the operational granularity, the comprehensiveness of operations is crucial for MLIR testing. Furthermore, the diverse combinations of operations in hybrid representations can be further enhanced by mutation to yield interesting hybrid patterns. Hence, we propose two mutation strategies for hybrid IRs in step ❷ of Figure 6: *similar operation replacement*, designed to establish sufficient testing opportunities for uncovered operations, and *mixing IR mutation*, which aims to explore more hybrid patterns.

**Similar Operation Replacement (SOR).** This mutation is implemented by replacing an existing API with a new one with similar functions or the same types of return values. For example, in Figure 6, the *linalg.pooling_nhwc_max* is replaced by a similar operation *linalg.pooling_nhwc_min* after mutation. Such mutations allow for the incorporation of operations that were initially overlooked into the test cases. Additionally, there are many similar operations from different dialects that can be interchangeable to yield interesting IR patterns. For example, replacing *affine.load* with *memeref.load* in an affine loop body–such an IR pattern cannot be naturally generated during the

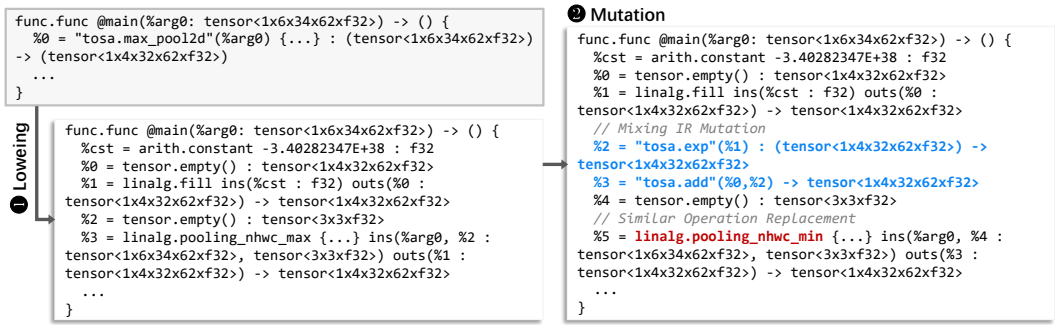



Fig. 6. **Example of mutation on hybrid IR, which is derived from lowering tosa IR.**

lowering process. In our work, all similar operations are automatically collected using a script that parses operation definition specification documents.

**Mixing IR Mutation (MIM).** Mixing mutation aims to enhance the complexity of the lower-level IR by introducing the features provided by the higher-level abstractions. In this paper, mixed mutation is implemented by inserting operations of higher-level dialects into the lower-level IR. Then, a series of operations are inserted into IR by reusing the aforementioned tosa IR generator, while maintaining valid connectivity with the original IR. As shown in Figure 6, MLIRTracer inserts two new tosa operations (i.e., *tosa.exp* and *tosa.add*) into the original IR, altering the original dataflow and enhancing the dialect diversity of the hybrid representation.

In SOR, each similar operation pair is validated for effectiveness through pre-executed replacement, ensuring the compatibility of the replacement mutations during fuzzing. In MIM, we develop an analyzer to identify feasible insertion points, and the operations to be inserted are created based on operation specifications, which minimizes the likelihood of producing invalid IR. Furthermore, each mutated IR undergoes validation by the MLIR's built-in verifier before progressing to the next stage. The verifier rigorously checks the IR, rejecting any that do not meet MLIR's requirements, ensuring that MLIRTracer can explore edge cases while maintaining syntactic validity.

## 4 Experimental Setup

### 4.1 Research Questions

We evaluate the effectiveness of MLIRTracer by addressing the following research questions:

- **RQ1:** Can MLIRTracer detect previously unknown bugs? (Sec. 5.1)
- **RQ2:** How does the effectiveness of MLIRTracer compare with state-of-the-art fuzzing techniques in testing the MLIR infrastructure? (Sec. 5.2)
- **RQ3:** Are all components of MLIRTracer contributing positive improvements to its final effectiveness? (Sec. 5.3)
- **RQ4:** How effective is MLIRTracer's directed testing compared to random testing? (Sec. 5.4)

### 4.2 Implementation

We implement MLIRTracer with the following main components:

**Generator.** Our test case generator is implemented as an MLIR pass, following Algorithm 1 to construct the tosa graph. MLIRTracer initializes the seed pool with the generated tosa graph- its superiority has been confirmed in our study. Note that the seed pool can be initialized using any MLIR program.

**IR analyzer.** We implement an IR analyzer that extracts all dialects and operations from the input MLIR program. With the IR analyzer, we conduct our directed testing strategy, which includes

selecting bug-prone dialects, applying specific lowering and transformation passes, and performing mutation on specific operations.

**Mutator.** We apply two mutation methods to increase seed space, with each method being randomly applied during the mutation process.

**Directed testing.** We implement the top-down testing strategy based on Algorithm 2. The directed testing focuses on bug-prone dialects, progressively lowering the Tosa IR along these dialects, and testing specific optimizations and transformations on them. MLIRTracer also performs extensive testing without specifying the lowering direction, covering a broader range of dialects.

**Executor.** Once MLIRTracer selects a seed file and specifies the corresponding pass sequence, they are sent to a sub-process for running transformation and conversion on the MLIR program. If a crash occurs during this process, it indicates that the specific pass sequence in the MLIR program triggers a bug. Additionally, once executed successfully, the newly transformed or lowered IRs will be added to the seed pool for future runs.

**Bug Reporter.** MLIRTracer automatically filters duplicate bugs by checking error messages to identify unique bugs. Ultimately, we report minimized test cases and steps to reproduce the bugs to the MLIR compiler infrastructure developers.

### 4.3 Compared Work

To answer RQ2, we compared MLIRTracer with the existing fuzzers specifically targeting MLIR compiler infrastructure, as well as high-level source program generators that can be adapted for indirect fuzzing MLIR.

- **MLIRSmith** [53]: This is the first fuzzing technique specifically targeting MLIR. It follows a pure generation-based approach, which randomly constructs MLIR programs according to their grammar.
- **MLIRod** [50]: This is the state-of-the-art fuzzing technique for fuzzing MLIR. The random generation approach of MLIRSmith restricts the exploration of the input space. To resolve this limitation, MLIRod designs a set of dependency-targeted mutation rules to drive the fuzzing process toward increasing operation dependency coverage.
- **NNSmith** [40]: This is one of the state-of-the-art test generators for fuzzing deep learning compilers by generating ONNX computation graphs. Its generated graph-level model can be transformed into a format acceptable to MLIR through the available front-end and then utilized as seeds for fuzzing MLIR. In this work, we run a MLIRTracer variant of NNSmith's model seeds (transformed into MLIR programs).

### 4.4 Evaluation Metrics

We use the following metrics for evaluation:

**Number of Detected Bugs.** We count the actual bugs detected by MLIRTracer based on fixes and developers' confirmation. We also present the unique bugs revealed by dialect-specific passes. This can help identify components that may require additional attention.

**Dialect and Operation Coverage.** Given the large number of dialects, we measure the covered dialect to examine the scope of the test across the entire MLIR infrastructure. Following existing work on testing DL library or compiler [27, 56], we also use operation coverage (i.e., API coverage) as an important metric for evaluating testing adequacy.

**Code Coverage.** For comparison, we measure branch coverage over MLIR's source code, which is collected using the llvm-cov [9] tool.

**Valid Testing Rate.** To compare our directed testing strategy with random testing, we also present the percentage of valid tests compared to the total number of tests conducted. Valid testing refers to

Table 3. Summary of Detected Bugs

| Bud ID | Status | Root Cause | Pass Category | Bud ID | Status | Root Cause | Pass Category |
|---|---|---|---|---|---|---|---|
| #58643 | *Fixed | - | Transformation (affine) | #58649 | Fixed | ICL | Conversion (affine) |
| #58658 | *Fixed | - | Transformation (affine) | #58664 | *Fixed | - | Transformation (affine) |
| #59929 | Reported | - | Conversion (affine) | #59932 | Fixed | IA | Transformation (scf) |
| #60579 | *Fixed | - | Transformation (affine) | #60922 | *Fixed | UD | Conversion (async) |
| #61085 | *Fixed | ICL | Conversion (linalg) | #61088 | *Fixed | - | Transformation (Bufferization) |
| #61107 | Fixed | IV | Conversion (scf) | #61154 | *Fixed | ICL | Conversion (memref) |
| #61155 | *Fixed | ICL | Conversion (memref) | #61167 | Fixed | ICL | Transformation (affine) |
| #61282 | Reported | IMA | Transformation (affine) | #61288 | Reported | - | Transformation (affine) |
| #61292 | *Fixed | IMA | Transformation (affine) | #61304 | Fixed | ICL | Conversion (tosa) |
| #61306 | *Fixed | - | Conversion (vector) | #61308 | Fixed | ICL | Transformation (affine) |
| #61309 | Fixed | ICL | Conversion (affine) | #61310 | Fixed | ICL | Transformation (affine) |
| #61311 | Fixed | IA | Transformation (Bufferization) | #61342 | Reported | IA | Conversion (scf) |
| #61343 | *Fixed | IA | Conversion (vector) | #61344 | Fixed | IV | Conversion (tosa) |
| #61345 | Fixed | ICL | Transformation (memref) | #61367 | Fixed | ICL | Transformation (memref) |
| #61371 | Fixed | ICL | Transformation (affine) | #61372 | fixed | ICL | Conversion (vector) |
| #61375 | Reported | - | Transformation (Bufferization) | #61376 | *Fixed | UD | Conversion (vector) |
| #61377 | *Fixed | - | Transformation (spirv) | #61378 | Fixed | IV | Conversion (tosa) |
| #61383 | Reported | - | Conversion (tosa) | #61385 | *Fixed | ICL | Conversion (tosa) |
| #61526 | Reported | IMA | General Transformation | #61527 | *Fixed | - | Transformation (func) |
| #61528 | Fixed | IA | Transformation (affine) | #61529 | *Fixed | ICL | Transformation (affine) |
| #61530 | Fixed | ICL | Transformation (linalg) | #61534 | Fixed | ICL | Transformation (affine) |
| #61578 | Fixed | IA | General Transformation | #61707 | Fixed | IA | Transformation (sparse) |
| #61709 | *Fixed | - | Transformation (linalg) | #61710 | Fixed | IMA | Transformation (spirv) |
| #61715 | Confirmed | - | Others | #61716 | Reported | - | Others |
| #61717 | Fixed | IA | Conversion (func) | #61734 | *Fixed | - | Transformation (memref) |
| #61735 | Confirmed | ICL | Others | #61793 | *Fixed | ICL | Transformation (sparse_tensor) |
| #61795 | *Fixed | - | Transformation (arith) | #61844 | *Fixed | - | Others |
| #61845 | Fixed | ICL | Transformation (linalg) | #61858 | Fixed | ICL | Others |
| #61863 | Confirmed | UD | Conversion (tosa) | #61867 | Reported | UD | Others |
| #61870 | Reported | - | Others | #61871 | Fixed | IA | Others |
| #61872 | Fixed | IA | Conversion (func) | #61842 | Fixed | ICL | Conversion (affine) |
| #61832 | Fixed | IA | Others | #62315 | Fixed | IA | Transformation (linalg) |
| #62317 | Fixed | UD | Others | #62318 | Fixed | ICL | Transformation (spirv) |
| #62319 | *Fixed | ICL | Conversion (linalg) | #62323 | Reported | - | Transformation (affine) |
| #62352 | *Fixed | - | Conversion (affine) | #62367 | Fixed | IA | Conversion (linalg) |
| #62368 | Fixed | IMA | Transformation (spirv) | #62369 | Fixed | IA | Transformation (Bufferization) |
| #62375 | Reported | - | Conversion (vector) | | | | |

* **Fixed** = The developer silently fixed a bug after we reported it.

cases where the input IR can be successfully transformed, triggering the compiler's actual processing logic rather than being disregarded. This metric is essential for measuring the effectiveness of the testing process.

### 4.5 Experimental Configurations

To answer RQ1, we applied MLIRTracer to fuzz MLIR compiler infrastructure (from revision e6c23f4 to revision da0730b) for 4 months to detect previously unknown bugs. To answer RQs 2-4, we selected the revision eb6014 (aligned with the released implementations of MLIRSmith and MLIRod) to perform fair comparisons. We adopted the released implementation of the compared works and used the recommended parameter settings. MLIRod initializes the seed pool using the state-of-the-art MLIR program generator (i.e., MLIRSmith). The DL models generated by NNsmith are transformed into a format acceptable to MLIR by the front-end tool onnx-mlir [13]. For MLIRTracer and its variants in the comparison experiment, the initial seed pool size was set to 500. In terms of tosa IR generation, we set the maximum number of operators (i.e., OpNum) to 30. For directed fuzzing, the seed size was set to 50 for each round of seed selection. During the experiments, directed fuzzing was enabled by default, except for the directed fuzzing evaluation experiment. To ensure sufficient testing, each experiment was conducted with a default time budget of 24 hours. Additionally, we repeated each experiment five times to reduce the influence of randomness. All experiments were conducted on a machine with 62 cores (Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz), 376GB RAM, and Ubuntu 20.04.6 LTS.

```
#map = affine_map<(d0)[s0] -> (2, -d0 + s0)>
  func.func @main(%arg0: memref<i64>) {
    %c1 = arith.constant 1 : index
    %c2 = arith.constant 2 : index
    %c4 = arith.constant 4 : index
    scf.for %arg1 = %c1 to %c4 step %c2 {
      %0 = affine.min #map(%arg1)[%c4]
    }
    return
  }
```

(a) Bug #61832: Incorrect Assertion.

```
#map = affine_map<() -> (0)>
  func.func @main(%arg0: tensor<f32>) -> vector<1xf32> {
    %0 = llvm.mlir.constant (0.000000e+00 : f32) : f32
    %1 = vector.transfer_read %arg0[], %0
         {permutation_map = #map} : tensor<f32>, vector
         <1xf32>
    return %1 : vector<1xf32>
  }
```

(b) Bug #61376: Unregistered Dialects.

```
func.func @main(%arg0: tensor<?xf32>) -> tensor<?xi32> {
  %0 = "tosa.argmax"(%arg0) {axis = 1 : i64} : (tensor<?xf32>) -> tensor<?xi32>
  return %0 : tensor<?xi32>
}
```

(c) Bug #61344: Incomplete Verifier.

```
spirv.module Logical GLSL450 requires #spirv.vce<v1.0, [Shader], []> {
  spirv.func @main(%arg0: !spirv.ptr<!spirv.struct<(!spirv.array<1 x f32, stride=4> [0])>, StorageBuffer>) "None" {
    %0 = spirv.Constant 0 : i32
    %1 = spirv.AccessChain %arg0[%0, %0] : !spirv.ptr<!spirv.struct<(!spirv.array<1 x f32, stride=4> [0])>,
         StorageBuffer>, i32, i32
    spirv.Return
  }
}
```

(d) Bug #62368: Invalid Memory Access.

Fig. 7. Example bugs detected by MLIRTracer.

## 5 Experimental Results

### 5.1 Previously Unknown Bugs Detected by MLIRTracer

**Bug Statistics.** To date, we have reported 73 crash bugs detected by MLIRTracer, where 61 bugs have been fixed or confirmed by developers. Of these 61 bugs, 22 are *conversion bugs*, which occur during the dialect conversion, and 32 are *transformation bugs*, which crash on invoke transformation pass. The remaining 7 bugs do not fall into either of these two categories and are labeled as 'others'. The detailed results are shown in Table 3. Notably, most of the transformation bugs were specific to dialect, with only 2 being related to general transformation.

**Case Study.** According to the developers' discussion and corresponding patches, we further summarized the root causes of these bugs, categorized as including Incorrect Assertion (IA), Unregistered Dialects (UD), Incomplete Verifier (IV), Invalid Memory Access (IMA), and Incorrect Code Logic (ICL). Moreover, for each root cause, we selected one bug as the illustrative example.

*Incorrect Assertion.* There are numerous assertions in MLIR compiler infrastructure designed to verify internal states. However, incorrect assertions can lead to crashes during the transformation or optimization process. For example, Bug#61832 (Figure 7a) was caused due to an incorrect assertion in the "-test-loop-unrolling" pass. In this pass, there is an assertion *upperBoundUnrolledCst* <= *ubCst* which ensures that after the loop is unrolled, the upper bound computed *upperBoundUnrolledCst* does not exceed the upper bound *ubCst* of the original loop. This assertion is incorrect in some cases, such as when the step size is greater than 1, to keep the number of iterations in the unrolled loop the same as in the original loop, the *upperBoundUnrolledCst* may be greater than the *ubCst*. This bug has been fixed by removing this incorrect assertion in loop unrolled.

Table 4. Comparison with Existing Work

| Tools | # Bug | # Dialects | # Operations |
|---|---|---|---|
| MLIRSmith | 16 | 12 | 172 |
| MLIRod | 19 | 20 | 350 |
| MLIRTracer | 51 | 28 | 404 |
| MLIRTracer (NNSmith seeds) | 41 | 23 | 278 |

*Unregistered Dialects.* To convert an operation from one dialect to another, the target dialect must be registered in the pass. Without this registration, the transformation process will crash. For example, Bug#61376 (Figure 7b) was triggered by the failure to register the tensor dialect for the "-convert-vector-to-scf" pass. During the lowering of the vector dialect, *tensor.extract* operation would be created, but since it was not registered in this *MLIRContext*, this led to a crash.

*Incomplete Verifier.* MLIR infrastructure uses a verifier to assess the validity of the operation. This root cause is the absence or incompleteness of a necessary verifier, leading to a crash when the pass activates on incompatible operations. Bug#61344 (Figure 7c) exposes an incomplete verifier issue caused by "tosa-to-linalg" pass working on the wrong axis. This test case is "invalid", generated by MLIRTracer to deliberately violate the attribute constraint defined in the operation specification. The input tensor is one-dimensional and requires reduction along axis 0, not axis 1. The issue was resolved by adding a verifier that ensures the operation is valid by checking if the axis falls within the rank of the input tensor.

*Invalid Memory Access.* Invalid memory access typically occurs due to out-of-bound access or NULL pointer dereference. Bug #62368 (Figure 7d) triggers this bug when using "-spirv-unify-aliased-resource" work on function arguments. This bug has been fixed by adding a check to ensure that the operation is not a nullptr.

*Incorrect Code Logic.* Code logic refers to the implementation of an algorithm (e.g., an optimization of loop tiling). Bugs in code logic often arise due to the complexity of compiler design. We have introduced such a bug (#58649) in Section 2.2.

## 5.2 Comparison with Existing Work

Table 4 shows the number of detected bugs and the number of covered dialects (and operations), respectively. The reported results are the average values from five experiments. From this table, MLIRTracer detected more bugs than other compared techniques during the 24-hour fuzzing process, suggesting better bug-exposing capability. MLIRTracer detected 3.19× and 2.68× more bugs than MLIRSmith and MLIRod. Moreover, MLIRTracer outperforms both techniques in terms of covered dialects and operations. Specifically, MLIRTracer covered 1.4× and 2.33× more dialects and 1.15× and 2.35× more operations than MLIRod and MLIRSmith, respectively. Further, we can see that with tosa graph generator, MLIRTracer discovered 1.24× more bugs, 1.22× more dialects, and 1.45× more operations than MLIRTracer with NNsmith seeds. Notably, the MLIRTracer with NNSmith seeds beats the best state-of-the-art technique (i.e., MLIRod) by exposing 2.16× more bugs.

Further, we investigated the reason for the superiority of MLIRTracer by measuring the covered branches in MLIR compiler infrastructure. Figure 8 presents the branch coverage trends for both MLIRTracer and the compared existing work within the default 24-hour budget. From this figure, the existing techniques tend to saturate very quickly, while MLIRTracer is able to keep coverage growth within the fuzzing time. To show the unique coverage for each studied technique, In total, MLIRTracer can at best achieve 43,273 branch coverage, which is 1.4× higher than MLIRod and MLIRSmith, and 1.3× higher than MLIRTracer with NNSmith seeds. In particular, 6,384 branches covered by MLIRTracer could not be covered by the compared techniques, demonstrating the effectiveness of MLIRTracer.

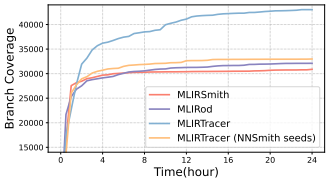
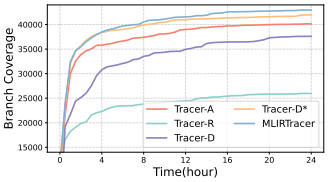
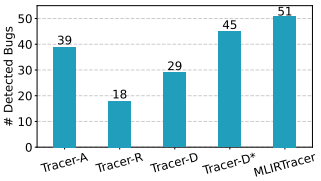



Fig. 8. The tendency of branch coverage over time on different fuzzers.




(a) Branch coverage trend




(b) Number of detected bugs

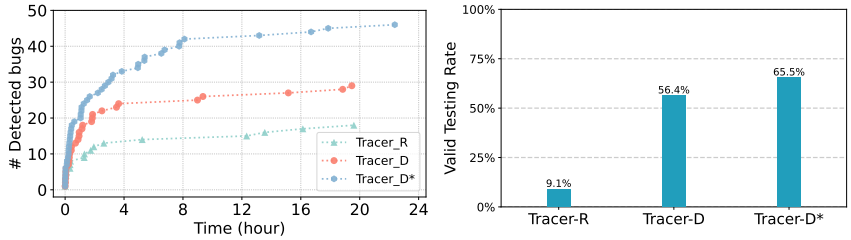
Fig. 9. Ablation study of MLIRTracer's components.

Overall, utilizing the MLIR programs transformed from high-level source programs generated by NNSmith is not effective enough to fuzz MLIR. These programs are constructed based on the characteristics of the target high-level programming languages rather than MLIR. As a result, the diversity of such MLIR programs is limited, leading to fewer dialects and operation coverage, code coverage, and fewer detected bugs compared to MLIRTracer. Furthermore, MLIRTracer outperforms the existing techniques MLIRSmith and MLIRod across all metrics. This is mainly because they focus on test case generation or mutation, which lacks the necessary guidance for exploring a large code base of MLIR and does not consider the dependency of IR and passes. In contrast, MLIRTracer considers the mutual effect of IR and pass, directs the fuzzer to bug-prone modules, and thus significantly improves the fuzzing effectiveness.

### 5.3 Ablation Study of MLIRTracer

Recall that MLIRTracer consists of three components: 1) a tosa graph generator that produced the highest-level MLIR programs (Sec. 3.2), 2) directed fuzzing strategy that guides exploration towards areas likely to expose bug (Sec. 3.3), and 3) mutators that mutate hybrid multi-level IR (Sec. 3.4). We evaluate the impact of tosa IR diversity for MLIR testing by variant Tracer-A. Tracer-A uses an API-level tosa IR generator that generates IR with only one tosa operator as initial seeds. We evaluate the effectiveness of the proposed directed fuzzing scheme using three MLIRTracer variants, Tracer-R, Tracer-D, and Tracer-D*. Tracer-R randomly selects a seed from the seed pool and randomly selects passes in MLIR compiler infrastructure to transform or optimize it, as done in MLIRSmith and MLIRod. In Tracer-D, the dependency between IR and pass is considered, but seed selection and directed lowering are disabled. Tracer-D* implements the whole directed testing strategy. In all of the above variants, the mutation strategy is disabled to avoid the test bias caused by the randomness of the test case mutation. We compared MLIRTracer with all five variants with a test time budget of 24 hours.

The experimental results for all variants are also shown in Figure 9, which presents the coverage trends and the number of bugs detected by each variant. Tracer-A can expose 39 bugs in MLIR, indicating the effectiveness of our top-down testing. By augmenting the diversity of tosa IR, Tracer-D* improves Tracer-A by exposing 6 more bugs. In terms of code coverage, Tracer-D* also outperforms Tracer-A. The main reason is that only generating one-operator tosa IR fails to test transformations working on the multiple-operator pattern. Comparing Tracer-D with Tracer-R, we observe that Tracer-D achieves 1.4× code coverage and detects 1.6× bugs than Tracer-R. Likewise, comparing Tracer-D* with Tracer-D, we can see that with our directed testing strategy, Tracer-D* discovered 1.55× bugs, suggesting a better bug-exposing capability. In terms of code coverage, the improvement of Tracer-D* over Tracer-D is 1.12×. Even if the bug-proneness of dialects becomes unavailable (as discussed in Section 6), MLIRTracer's top-down testing strategy, based on dependency-guided lowering, remains effective. Although its bug detection performance is inferior to that of directed testing, it still outperforms random testing. As a result, this confirms that both our dependency-guided strategy and bug-proneness guidance are effective, and together they




(a) Number of detected bugs (b) Valid testing rate.

Fig. 10. Directed Testing vs. Random Testing.

can fuzz the MLIR's hierarchical structure space in a top-down way, ensuring directed exploration of bug-prone areas. Lastly, the default MLIRTracer outperforms Tracer-D* in both bug detection and code coverage, as mutating hybrid IR helps generate more interesting IR patterns and improve test coverage.

### 5.4 Directed Testing vs. Random Testing

The directed testing strategy is the main contribution of MLIRTracer, and thus we further investigate why this strategy outperforms the random testing method for MLIR fuzzing. Figure 10a shows the growth of bugs across 24 hours. Tracer-D* outperforms the other two variants in terms of both bug count and detection efficiency. With directed fuzzing, Tracer-D* can quickly expose bugs early in the testing process. Specifically, in the first 4-hour window, Tracer-D* and Tracer-D detected 33 and 24 bugs, respectively, while Tracer-R, using random testing, exposed only 13 bugs. Although random testing can also detect these bugs, it takes more time.

Figure 10b further illustrates the proportion of valid tests (i.e., tests in which the compiler passes are actually executed) throughout the fuzzing process. The valid testing rate of Tracer-R is only 9.1%, indicating that random testing is very inefficient for fuzzing MLIR. Random testing lacks knowledge of compiler internals, leading to most passes being simply neglected by the MLIR compiler. In contrast, Tracer-D implements dependency-based testing, achieving a higher valid testing rate of 56.4%, with a 47.3% improvement over random testing. The superiority of dependency-based testing lies in its significantly improved efficiency, as it performs dialect conversion with the guidance of lowering rules. Compared Tracer-D* with Tracer-D, we observe that Tracer-D* achieves 9.1% higher valid testing rates. The reason behind this could be that directed testing specifies the direction of IR lowering, while arbitrary lowering for a hybrid IR could lead to more conversion failures due to possible implicit constraints in the order of dialect conversion.

Further, we extracted the shortest bug-triggering sequences and conducted a statistical analysis on the bugs detected by MLIRTracer. On average, the length of the bug-triggering paths (i.e., the number of enabled passes) is approximately 11, while the length of the lowering paths (considering only the lowering passes) is around 5. The longest bug-triggering path length is 24. This highlights the importance of systematically exploring the code space, as bugs in MLIR are distributed in components of different dialects, and bugs in lower-level dialects often require specific, longer sequences to be triggered. MLIRTracer's dependency-guided strategy facilitates top-down exploration of the code space, while the directed strategy further focuses on vulnerable areas to improve the efficiency of bug discovery.

## 6 Threats to validity

The threats to *internal validity* mainly lie in the potential faults in the implementation of MLIRTracer. To reduce this kind of threat, we carefully checked all our code via code review and designing test cases.

The threats to *external validity* mainly lie in the evolution of MLIR. The MLIR framework is rapidly evolving, with an increasing diversity of domain-specific and custom IR in MLIR ecosystems. This issue can be mitigated due to the MLIRTracer is easy to extend. To maintain MLIRTracer's effectiveness on new MLIR versions, it is necessary to update the tosa operation specifications (if tosa operators get changed), construct lowering rules for emerging or custom dialects, and introduce new transformation passes. In addition, MLIRTracer relies on bug proneness within dialect implementations to guide testing. This necessitates regularly collecting recent git commits and retraining the model, although the training process can largely be automated. If git history becomes unavailable as MLIR evolves and matures, static code features from source code (see Table 1 could still serve as a metric for indicating bug proneness.

The threats to *construct validity* mainly lie in threshold settings in MLIRTracer. The threshold $\sigma$, as discussed in Section 3.3, controls the number of bug-prone dialects. A threshold that is too high results in identifying too few dialects due to excessive function filtering. On the other hand, a low threshold may lead to low confidence scores, which can cause resources to be wasted on testing non-critical components. In this paper, we set $\sigma$ to 0.7, achieving a balance between reliability in bug-proneness identification and the number of dialects identified as bug-prone.

## 7 Related work

**MLIR Fuzzing.** Recently, two MLIR fuzzing techniques have been proposed, i.e., MLIRSmith [53] and MLIRod [50]. These techniques focus on generating hybrid MLIR programs to detect bugs. MLIRSmith directly constructs MLIR programs containing 12 dialects according to the template, while MLIRod further mutates these MLIR programs as test inputs. However, these methods are inefficient because they perform random testing without leveraging any knowledge of MLIR compiler internals.

**Generation-based Fuzzing.** There has been much research on generation-based fuzzing for DL libraries or compilers [26, 49, 54, 56, 58]. FreeFuzz [56] performs API-level fuzzing by mining API inputs from open source. Similarly, DocTer [58] extracts specific input constraints to generate valid API input. However, these API-based fuzzers cannot detect bugs triggered by a specific sequence of APIs, leading to an inefficient bug-revealing ability. Subsequent studies have proposed by generating diverse computation graphs to improve test effectiveness [31, 40, 41, 55]. LEMON [55] presents a mutation-based approach. Muffin [31] generates a multi-operator model layer by layer for testing DL libraries. NNSmith [40] generates graphs and operator attributes using an SMT solver to meet connection constraints between operators. Nerui [41] improves model generation by inferring operator rules automatically. These approaches focus on finding bugs in traditional DL libraries or DL compilers, but they are not tailored for MLIR testing.

**Directed Fuzzing.** Recent coverage-guided fuzzers have been proposed in the field of deep learning [42, 59–61]. The state-of-the-art Tzer [42] presents a coverage-guided fuzzing framework to detect potential bugs in tensor compilers. However, this approach is not suitable for MLIR testing because it cannot capture complex code paths and provide effective guidance for covering deep code paths. Our directed fuzzing approach differs from previous work and is motivated by new features of the MLIR compiler framework, which provide targeted guidance for fuzzing MLIRs more effectively.

## 8 Discussion

**Scalability.** Since MLIR serves as an upstream extension of LLVM, MLIRTracer can be extended to support LLVM testing by lowering the generated MLIR programs into LLVM IR. Although our top-down testing strategy cannot be directly applied to LLVM due to its single-level structure, the key idea of directed testing strategy—prioritizing testing components with higher bug-proneness—can

still be effectively adapted and utilized for LLVM testing. Moreover, while MLIRTracer approach may not be directly applicable to some existing compilers at present, multi-level IR architectures are emerging as a new trend in compilers, e.g., CIRCT [4] and Tensor Comprehensions [52], indicating that MLIRTracer will have broader generalizability in the future.

**Significance of MLIRTracer.** Many compilers are built on top of MLIR compiler infrastructure, and fuzzing it can contribute to ensuring the quality and robustness of all MLIR-based compilers. Now, MLIR's growing use in fields like machine learning and high-performance computing, demands robust tools for testing. MLIRTracer highlights the importance of knowledge of MLIR infrastructure internals and the critical need for guidance from top to down in exploring the hierarchical space of MLIR. By overcoming the limitations of random fuzzing methodology, MLIRTracer represents a significant advancement in the fuzzing of MLIR compiler infrastructures. We believe that MLIRTracer will inspire future work in this promising direction.

**Future Work.** Following MLIR's design philosophy of progressive lowering, MLIRTracer currently focuses on exploring the hierarchical, modular structure of MLIR through multiple steps, rather than employing an end-to-end code generation path. In addition, under the guidance of dependency lower rules, while MLIRTracer achieves 65.5% valid testing rate, it was still not enough to build the entire end-to-end code generation path. Our future work is to excavate possible implicit constraints between passes and expand MLIRTracer to achieve end-to-end inference, which will reveal inconsistencies and detect more crashes. This enhancement will involve the development of algorithms and techniques to identify and resolve discrepancies, irregularities, or conflicts in the code, culminating in a more comprehensive and robust testing framework.

## 9 Conclusion

This paper presents MLIRTracer, a top-down directed testing methodology that effectively explores MLIR's hierarchical code space. It begins at the highest abstraction level, progressively lowers through various levels of abstraction, and tests the transformation functionalities at each level. MLIRTracer builds a test case generator based on the highest dialect to create diverse and valid tosa IRs. We consider dialect conversion dependencies to progressively lower Tosa IRs in the correct order. To handle the large test space of MLIR, we incorporate directed testing, which guides the fuzzer toward specific bug-prone areas to improve testing efficiency. In addition, we propose mutation strategies that mutate the hybrid IR to enhance test coverage. By now, we have successfully detected 73 newly found bugs, with 61 fixed. This demonstrates the effectiveness of MLIRTracer in improving the quality and reliability of MLIR-based systems.

## 10 Data Availability

Our data and tool are publicly available at [21].

## Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions. The work is partly funded by the National Key Research and Development Program of China (2023YFB3001804), the National Natural Science Foundation of China (NSFC) under grant agreements 62372373 and 62472351, the Shaanxi Province Key Research and Development Project (2024CY2-GJHX-60), the China Postdoctoral Science Foundation Fellowship (2022M712575), and the Graduate Innovation Project of Northwest University (CX2024199). Additionally, this work is supported by the Shaanxi Key Laboratory of Passive Internet of Things and Neural Computing and the Xi'an Key Laboratory of Advanced Computing and Software Security.

## References


[1] [n. d.]. ANTLR v4. https://github.com/antlr/antlr4. Accessed on September 2024.
[2] [n. d.]. BTOR2MLIR: A Format and Toolchain for Hardware Verification. https://github.com/jetafese/btor2mlir. Accessed on September 2024.
[3] [n. d.]. Catalyst. https://github.com/PennyLaneAI/catalyst. Accessed on September 2024.
[4] [n. d.]. CIRCT:Circuit IR Compilers and Tools. https://github.com/llvm/circt. Accessed on September 2024.
[5] [n. d.]. Developer Guide. https://mlir.llvm.org/getting_started/DeveloperGuide/. Accessed on September 2024.
[6] [n. d.]. Dialects. https://mlir.llvm.org/docs/Dialects/. Accessed on September 2024.
[7] [n. d.]. IREE:Intermediate Representation Execution Environment. https://openxla.github.io/iree/. Accessed on September 2024.
[8] [n. d.]. The LLVM Compiler Infrastructure. https://github.com/llvm/llvm-project. Accessed on September 2024.
[9] [n. d.]. llvm-cov. https://www.llvm.org/docs/CommandGuide/llvm-cov.html. Accessed on September 2024.
[10] [n. d.]. MLIR Code Repository. https://github.com/llvm/llvm-project/tree/main/mlir. Accessed on September 2024.
[11] [n. d.]. MLIR Language Reference. https://mlir.llvm.org/docs/LangRef/. Accessed on September 2024.
[12] [n. d.]. MLIR passes. https://mlir.llvm.org/docs/Passes/. Accessed on September 2024.
[13] [n. d.]. ONNX-MLIR. https://onnx.ai/onnx-mlir/. Accessed on September 2024.
[14] [n. d.]. Pass Infrastructure. https://mlir.llvm.org/docs/PassManagement/. Accessed on September 2024.
[15] [n. d.]. Polyblocks. https://www.polymagelabs.com/technology/#polyblocks. Accessed on September 2024.
[16] [n. d.]. TableGen Overview. https://llvm.org/docs/TableGen/. Accessed on September 2024.
[17] [n. d.]. TOSA specfication. https://github.com/llvm/llvm-project/blob/main/mlir/include/mlir/Dialect/Tosa/IR/TosaOps.td. Accessed on September 2024.
[18] [n. d.]. Triton. https://github.com/triton-lang/triton. Accessed on September 2024.
[19] [n. d.]. ULingoDB - Revolutionizing Data Processing with Compiler Technology. https://www.lingo-db.com/. Accessed on September 2024.
[20] [n. d.]. Users of MLIR. https://mlir.llvm.org/users/. Accessed on September 2024.
[21] 2025. MLIRTracer. Online. https://github.com/compiler-testing/MLIRTracer
[22] Mohamed Abdelrahman Aljemabi and Zhongjie Wang. 2018. Empirical Study on the Evolution of Developer Social Networks. *IEEE Access* 6 (2018), 51049–51060. https://doi.org/10.1109/ACCESS.2018.2868427
[23] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* 19, 4, Article 50 (Sept. 2022), 25 pages. https://doi.org/10.1145/3544559
[24] Uday Bondhugula. 2020. High performance code generation in mlir: An early case study with gemm. *arXiv preprint arXiv:2003.00532* (2020).
[25] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen
[26] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. 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*. 423–435. https://doi.org/10.1145/3597926.3598067
[27] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing Deep-Learning Libraries via Automated Relational API Inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 44–56. https://doi.org/10.1145/3540250.3549085
[28] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *The Journal of Machine Learning Research* 20, 1 (2019), 1997–2017. http://jmlr.org/papers/v20/18-598.html
[29] Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, Anderson Uchôa, Ana Carla Bibiano, Alessandro Garcia, João Lucas Correia, Filipe Santos, Gabriel Nunes, Caio Barbosa, et al. 2018. The buggy side of code refactoring: Understanding the relationship between refactorings and bugs. In *Proceedings of the 40th international conference on software engineering: companion proceeedings*. 406–407. https://doi.org/10.1145/3183440.3195030
[30] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. 171–180. https://doi.org/10.1145/2372251.2372285
[31] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*. 1418–1430. https://doi.org/10.1145/3510003.3510092

[32] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*. 200–210. https://doi.org/10.1109/ICSE.2012.6227193

[33] Peng He, Bing Li, Yutao Ma, Lulu He, et al. 2013. Using software dependency to bug prediction. *Mathematical Problems in Engineering* 2013 (2013). https://doi.org/10.1155/2013/869356

[34] Pengchao Hu, Man Lu, Lei Wang, and Guoyue Jiang. 2022. TPU-MLIR: A Compiler For TPU Using MLIR. *arXiv preprint arXiv:2210.15016* (2022).

[35] 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

[36] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. 2022. MLIR-based code generation for GPU tensor cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 117–128. https://doi.org/10.1145/3497776.3517770

[37] Sunghun Kim, E James Whitehead, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on software engineering* 34, 2 (2008), 181–196. https://doi.org/10.1109/TSE.2007.70773

[38] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[39] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A Large-Scale Empirical Study on Vulnerability Distribution within Projects and the Lessons Learned. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 1547–1559. https://doi.org/10.1145/3377811.3380923

[40] 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

[41] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. 2023. Neuri: Diversifying dnn generation via inductive rule inference. (2023), 657–669. https://doi.org/10.1145/3611643.3616337

[42] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–26. https://doi.org/10.1145/3527317

[43] Weisi Luo, Dong Chai, Xiaoyue Ruan, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-based fuzz testing for deep learning inference engines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 288–299. https://doi.org/10.1109/ICSE43902.2021.00037

[44] William S Moses, Ivan R Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. 2023. High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs. (2023), 119–134. https://doi.org/10.1145/3572848.3577475

[45] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. 2020. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1224–1228. https://doi.org/10.1145/3324884.3418930

[46] Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. 2017. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering* 45, 2 (2017), 194–218. https://doi.org/10.6084/m9.figshare.4542709.v6

[47] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can Developer-Module Networks Predict Failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 2–12. https://doi.org/10.1145/1453101.1453105

[48] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. 2021. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–34. https://doi.org/10.1145/3447582

[49] Jingyi Shi, Yang Xiao, Yuekang Li, Yeting Li, Dongsong Yu, Chendong Yu, Hui Su, Yufeng Chen, and Wei Huo. 2023. Acetest: Automated constraint extraction for testing deep learning operators. (2023), 690–702. https://doi.org/10.1145/3597926.3598088

[50] 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*. 1287–1299. https://doi.org/10.1145/3650212.3680360

[51] Nicolas Vasilache, Oleksandr Zinenko, Aart JC Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, et al. 2022. Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. *arXiv preprint arXiv:2202.03293* (2022).

[52] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[53] 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 *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1555–1566. https://doi.org/10.1109/ASE56229.2023.00120

[54] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. 2022. EAGLE: creating equivalent graphs to test deep learning libraries. In *Proceedings of the 44th International Conference on Software Engineering*. 798–810. https://doi.org/10.1145/3510003.3510165

[55] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 788–799. https://doi.org/10.1145/3368089.3409761

[56] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 995–1007. https://doi.org/10.1145/3510003.3510041

[57] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–28. https://doi.org/10.1145/3489048.3522655

[58] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–188. https://doi.org/10.1145/3533767.3534220

[59] Xiaofei Xie, Hongxu Chen, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. Coverage-guided fuzzing for feedforward neural networks. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1162–1165. https://doi.org/10.1109/ASE.2019.00127

[60] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157. https://doi.org/10.1145/3293882.3330579

[61] Pengcheng Zhang, Bin Ren, Hai Dong, and Qiyin Dai. 2021. Cagfuzz: coverage-guided adversarial generative fuzzing testing for image-based deep learning systems. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4630–4646. https://doi.org/10.1109/TSE.2021.3124006

Received 2024-09-04; accepted 2025-04-01