



# DESIL: Detecting Silent Bugs in MLIR Compiler Infrastructure

CHENYAO SUO, Tianjin University, China

JIANRONG WANG, Tianjin University, China

YONGJIA WANG, Tianjin University, China

JIAJUN JIANG, Tianjin University, China

QINGCHAO SHEN, Tianjin University, China

JUNJIE CHEN\*, Tianjin University, China

MLIR (Multi-Level Intermediate Representation) compiler infrastructure provides an efficient framework for introducing a new abstraction level for programming languages and domain-specific languages. It has attracted widespread attention in recent years and has been applied in various domains, such as deep learning compiler construction. Recently, several MLIR compiler fuzzing techniques, such as MLIRSmith and MLIRod, have been proposed. However, none of them can detect silent bugs, i.e., bugs that incorrectly optimize code silently. The difficulty in detecting silent bugs arises from two main aspects: (1) **UB-Free Program Generation**: Generates programs that are free from undefined behaviors to suit the non-UB assumptions required by compiler optimizations. (2) **Lowering Support**: Converts the given MLIR program into an executable form with a suitable lowering path that reduces redundant lowering passes and improves the efficiency of fuzzing. To address the above issues, we propose DESIL. DESIL enables silent bug detection by defining a set of UB-elimination rules based on the MLIR documentation and applying them to input programs. To convert dialects in the MLIR program into executable form, DESIL designs a lowering path optimization strategy to convert the dialects in the given MLIR program into executable form. Furthermore, DESIL incorporates the differential testing for silent bug detection. It introduces an operation-aware optimization recommendation strategy into the compilation process to generate diverse executable files. We applied DESIL to the latest revisions of the MLIR compiler infrastructure. It detected 23 silent bugs and 19 crash bugs, of which 17/16 have been confirmed or fixed.

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

Additional Key Words and Phrases: Compiler Fuzzing, MLIR Compiler Infrastructure, Undefined Behavior

## ACM Reference Format:

Chenyao Suo, Jianrong Wang, Yongjia Wang, Jiajun Jiang, Qingchao Shen, and Junjie Chen. 2025. DESIL: Detecting Silent Bugs in MLIR Compiler Infrastructure. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 383 (October 2025), 27 pages. <https://doi.org/10.1145/3763161>

## 1 Introduction

The MLIR (Multi-Level Intermediate Representation) compiler infrastructure is a powerful and extensible framework designed to facilitate compiler construction across diverse domains, including machine learning, high-performance computing, and hardware accelerators [12]. By providing a structured representation at multiple abstraction levels, MLIR enables efficient transformations,

\*Junjie Chen is the corresponding author.

Authors' Contact Information: **Chenyao Suo**, Tianjin University, Tianjin, China, [chenyaosuo@tju.edu.cn](mailto:chenyaosuo@tju.edu.cn); **Jianrong Wang**, Tianjin University, Tianjin, China, [wjr@tju.edu.cn](mailto:wjr@tju.edu.cn); **Yongjia Wang**, Tianjin University, Tianjin, China, [yongjiawang@tju.edu.cn](mailto:yongjiawang@tju.edu.cn); **Jiajun Jiang**, Tianjin University, Tianjin, China, [jiangjiajun@tju.edu.cn](mailto:jiangjiajun@tju.edu.cn); **Qingchao Shen**, Tianjin University, Tianjin, China, [qingchao@tju.edu.cn](mailto:qingchao@tju.edu.cn); **Junjie Chen**, Tianjin University, Tianjin, China, [junjiechen@tju.edu.cn](mailto:junjiechen@tju.edu.cn).



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/10-ART383

<https://doi.org/10.1145/3763161>

optimizations, and target-specific code generation, making it a cornerstone of modern compiler design. However, given its growing adoption in critical applications such as deep learning and hardware synthesis, ensuring the correctness of MLIR is paramount. Particularly, bugs in MLIR can propagate through the compilation pipeline, leading to incorrect program execution, degraded performance, or even security vulnerabilities [19, 21]. Therefore, rigorous testing techniques are essential, ensuring that MLIR remains a robust and trustworthy infrastructure for compiler development and optimization.

Due to the unique characteristics of MLIR (such as its use of dialects to manage multi-level IRs and its proprietary data structures and semantics), traditional compiler testing techniques are largely inapplicable. Therefore, in recent years, some testing techniques tailored to the MLIR compiler infrastructure have been proposed [19, 21]. For example, MLIRSmith [21] generates MLIR programs based on its grammar for the testing purpose. MLIRod [19] mutates existing MLIR programs for testing, guided by the diversity of operation dependencies within MLIR programs. However, these techniques are limited to detecting crash bugs, failing to capture silent bugs (also known as wrong code bugs [24]), which generate incorrect executable code without triggering crashes. This limitation arises due to the challenging issue of undefined behavior (UB) — a scenario where program execution lacks a well-defined outcome due to violations of language specifications, leading to unpredictable execution results [24]. Note that silent compiler bugs pose a severe risk, as they can go unnoticed during compilation and cause erroneous behaviors at runtime, potentially leading to critical failures in real-world applications.

In the literature, eliminating UB has been recognized as an important yet challenging task [13–15, 24]. This challenge arises from the diverse root causes of UB — such as memory safety violations, uninitialized variables, integer overflows, and type mismatches — which can emerge at any stage of compilation and propagate silently through optimizations. The unique characteristics of MLIR further exacerbate this problem. Specifically, MLIR supports multiple dialects, each with its own operations, attributes, and verification rules, significantly expanding the scope of potential UB. Moreover, MLIR introduces dialect-specific UB root causes, such as shape inconsistency in *memref* and *linalg* dialects, which require specialized runtime checks and analysis for effective detection and elimination. Unlike traditional programming languages, MLIR lacks dedicated UB detection tools, making even well-known UB issues more difficult to identify and mitigate.

Assuming UB-free MLIR programs can be obtained, using them to detect silent bugs still faces the compilation challenge — the process of transforming an MLIR program into an executable form (i.e., solely represented by the *llvm* or *spirv* dialect). This challenge arises because MLIR programs often require multiple lowering stages across different dialects (especially their operations) before reaching a fully executable representation. Specifically, an MLIR program may contain operations from various dialects, each necessitating specific lowering passes to transition into an executable representation. For ease of presentation, we call a sequence of lowering passes to transform a dialect operation to the specified executable dialect an *operation-specific lowering path*, and a sequence of lowering passes to transform an MLIR program to the executable form a *lowering path*. Furthermore, new dialects can be introduced dynamically during the lowering process, leading to an expansive and evolving space of possible lowering paths. While an exhaustive enumeration of all possible lowering sequences could theoretically ensure successful compilation, it would impose a significant efficiency bottleneck — a crucial factor in compiler testing [8]. Therefore, determining an appropriate lowering path is essential to balance compilation feasibility and testing efficiency, enabling more effective detection of silent bugs.

To bridge the gap in detecting silent MLIR bugs, we propose **DESIL** (**DE**tecting **SIL**ent bugs), a novel technique that jointly generates UB-free programs and determines an optimal lowering path for each program to facilitate effective bug detection. Specifically, to tackle the first challenge,

DESIL designs a set of MLIR program transformation rules to eliminate UB in UB-prone operations. For example, to address the UB-prone operation of copying a memref variable to another when both have dynamic shapes, DESIL introduces a transformation rule that replaces the source memref value with one that deterministically matches the destination variable's shape. Compared to existing tools like Csmith [24] that address UBs specific to C, we design targeted UB-elimination rules for MLIR's diverse dialects, ensuring well-defined tests without losing semantic integrity. Unlike Csmith, which integrates UB avoidance into the code generation process, DESIL tackles the UB issue through post-processing of already-generated MLIR programs, making it orthogonal to all existing MLIR testing techniques. This allows DESIL to be seamlessly combined with them, enhancing the detection of silent bugs and demonstrating its practicality. To tackle the second challenge, DESIL determines an optimal lowering path that prevents redundant or circular application of lowering passes, ensuring efficient compilation to an executable representation. Specifically, DESIL first builds a mapping between lowering passes and dialect operations based on MLIR documentation, recording an operation-specific lowering path for each operation. Then, given an MLIR program, DESIL determines its optimal lowering path by performing *topological sorting* on the lowering passes derived from the operation-specific lowering paths of the program's operations. Note that our lowering-path optimization is essential for sequencing passes and generating executable tests, given the intricate interactions among MLIR dialects and its complex lowering pipelines, which is an innovation beyond the single-language scope of existing works (such as Csmith [24]).

With these UB-elimination rules and the lowering-path optimization algorithm, DESIL can effectively and efficiently compile a UB-free MLIR program into an executable form. However, it is hard to directly determine whether the executable program is as expected. Therefore, to make DESIL self-contained, we incorporate the differential testing mechanism into DESIL. Specifically, DESIL introduces operation-aware optimization recommendation, which specifies optimization passes according to the operations in the given MLIR program, and obtains a set of executable programs produced by different optimization passes for differential testing. Any inconsistent result produced by their executions are regarded as a silent bug detected by DESIL.

To evaluate the effectiveness of DESIL, we applied DESIL to test the latest versions (from adbf21 to b6d5fa) of the MLIR compiler infrastructure over approximately four months. Specifically, we integrated DESIL with MLIRSmith and MLIRRod to process their generated MLIR programs, naming them DESIL<sub>smith</sub> and DESIL<sub>od</sub>, respectively. In total, DESIL detected 42 previously unknown bugs, including 23 silent bugs and 19 crash bugs, of which 26 have been fixed and 33 confirmed by developers. We further compared DESIL with two enhanced state-of-the-art techniques, MLIRSmith<sub>enhanced</sub> and MLIRRod<sub>enhanced</sub> (since neither of them can transition MLIR programs into executable forms to detect silent bugs), through five repeated 12-hour fuzzing sessions. The results show that DESIL<sub>smith</sub> and DESIL<sub>od</sub> detected 29 and 38 bugs, respectively, outperforming MLIRSmith<sub>enhanced</sub> (20) and MLIRRod<sub>enhanced</sub> (25), while also significantly reducing false positives in silent bug detection. The latter techniques suffered from extremely high false positive rates (97.33% and 96.96%) due to the UB issue. Additionally, our ablation study confirmed the essential contributions and practicality of DESIL's lowering path optimization and operation-aware optimization recommendation strategies. For evaluating lowering path optimization strategy, we replaced this strategy with a random lowering pass selection strategy and designed two variants: DESIL<sub>smith</sub><sup>w/o lower</sup> and DESIL<sub>od</sub><sup>w/o lower</sup>. Through five repeated 12-hour fuzzing sessions, neither DESIL<sub>smith</sub><sup>w/o lower</sup> nor DESIL<sub>od</sub><sup>w/o lower</sup> successfully lowered any MLIR program within 50 lowering passes. In contrast, DESIL<sub>smith</sub> and DESIL<sub>od</sub> required only 21 lowering passes on average to lower an MLIR program. These results demonstrate the effectiveness of the lowering path optimization strategy. For evaluating operation-aware optimization recommendation strategy, we replaced it with a random optimization selection strategy and designed

two variants:  $\text{DESIL}_{smith}^{w/o\ opt}$  and  $\text{DESIL}_{od}^{w/o\ opt}$ . We conducted five repeated 12-hour fuzzing sessions. The results show that  $\text{DESIL}_{smith}$  and  $\text{DESIL}_{od}$  detected 29 and 38 bugs, respectively, significantly outperforming  $\text{DESIL}_{smith}^{w/o\ opt}$  (21) and  $\text{DESIL}_{od}^{w/o\ opt}$  (31). These findings demonstrate the effectiveness of the operation-aware optimization recommendation strategy.

In this paper, we make the following main contributions:

- We propose DESIL, the first testing technique designed to detect silent bugs in the MLIR compiler infrastructure.
- We design a set of MLIR program transformation rules to eliminate UB-prone operations in any given MLIR program, enabling the feasible generation of UB-free MLIR programs for effective silent bug detection.
- We introduce a lowering-path optimization strategy by performing topological sorting on the lowering passes associated with the operations in a given MLIR program, identified through MLIR documentation analysis. This strategy ensures efficient compilation into an executable form, optimizing the lowering process by eliminating redundant lowering.
- We evaluate DESIL on the latest versions of the MLIR compiler infrastructure, uncovering 42 previously unknown bugs, of which 26/33 have been fixed/confirmed by developers. Notably, we have publicly released our experimental data and implementation at our project homepage [2] and our artifact [20].

## 2 Background and Motivation

### 2.1 Terminology

MLIR (Multi-Level Intermediate Representation) is a versatile and extensible intermediate representation designed to support multiple levels of abstraction and facilitate the development of various domain-specific compilers. To enable efficient compilation and optimization across different hardware and software targets, MLIR introduces the concept of **dialects**, which are modular and extensible units that define custom operations, types, and attributes for specific domains or abstraction levels. An **operation** in MLIR is a fundamental unit of computation or transformation, representing a specific task or behavior within a dialect. It takes as input a list of operands and attributes, performs a defined action, and produces one or more results as output.

For example, Figure 1(a) illustrates an example of an MLIR program. The program performs the following operations: (1) Defines a constant integer value %1 via `arith.constant` operation (Line 1). (2) Reads a vector %v from a `memref` value %m with a beginning position %idx9 via `affine.vector_load` operation (Line 2). (3) Extracts an element from the vector value via `vector.extract` operation (Line 3). (4) Performs arithmetic calculations via `arith` operations (Lines 4-6). Specifically, the `arith.addi` operation takes as input two `i32` type operands (%0 and %1), and produces a result of an `i32` type value (%2) in line 5 in Figure 1(a). Particularly, the attributes of an operation in MLIR actively participate in the computation process. For example, the attribute `value=1:i32` in the `arith.constant` operation defines the literal value of the constant %1 (Line 1 in Figure 1(a)).

An MLIR program typically comprises operations from multiple dialects. To compile it into an executable representation, these operations must be transformed into those within **target-specific dialects** (e.g., the `llvm` and `spirv` dialects, referred to as **executable dialects** for clarity in this paper). This transformation enables program execution, which is essential for detecting silent bugs. To achieve the transformation, MLIR provides a collection of lowering passes. Specifically, a **lowering pass** is a transformation that converts operations from one dialect to another, typically moving from higher-level abstractions to lower-level representations. For example, the “-convert-arith-to-llvm” pass converts an `arith` operation into an `llvm` operation shown in Figure 1(b). Additionally,

```

1  %l = "arith.constant"() <{value = 1 : i32}> : () -> i32
2  %v = affine.vector_load %m[%idx9] : memref<14xi32>, vector<6xi32>
3  %p = vector.extract %v[5] : i32 from vector<6xi32>
4  %0 = arith.minui %arg, SIGNED_INT_MAX-1 : i32
5  %2 = arith.addi %0, %l : i32
6  %3 = arith.divsi %2, 4: i32

```

(a) An example of MLIR program

```

4  %0 = llvm.intr.umin (%arg, SIGNED_INT_MAX-1) : (i32, i32) -> i32
5  %2 = llvm.add %0, %l : i32
6  %3 = llvm.sdiv %2, 4: i32

```

(b) Lowered MLIR program (for lines 4-6 in Figure 1(a)) by applying “-convert-arith-to-llvm”

```

4  %0 = arith.minui %arg, SIGNED_INT_MAX-1 : i32      Ranges from 0 to 2147483646
5  %2 = arith.addi %0, %l : i32                      Ranges from 1 to 2147483647
6  %3 = arith.divui %2, 4: i32

```

(c) Optimized MLIR program (for lines 4-6 in Figure 1(a)) by applying “-arith-unsigned-when-equivalent”

```

1  %HasEnoughSpace = index.cmp uge (Volume(%m), %BoundedFlattenIdx+Volume(vector<6xi32>))
2  ① %m1 = scf.if %HasEnoughSpace -> (memref<?xi32>) {
3      %new-m = memref.cast %m : memref<14xi32> to memref<?xi32>    Check the volume of %m
4      scf.yield % %new-m : memref<?xi32>
5  } else {
6      %new-m = memref.alloc(%EnoughVolume) : memref<?xi32>        New memref with enough
7      ② linalg.fill ins(%RandVal : i32) outs(%new-m : memref<14xi32>) volume generation
8      scf.yield %new-m : memref<?xi32>
9  }
10  ③ %v = affine.vector_load %m1[%idx9 mod m1.dim(0)] : memref<?xi32>, vector<6xi32>
11  %p = vector.extract %v[5] : i32 from vector<6xi32>    Replace old operand with safe version

```

(d) MLIR program after utilizing undefined behavior elimination (for lines 2-3 in Figure 1(a))

```

1  %v = vector.load %m1[%idx9 mod m1.dim(0)] : memref<?xi32>, vector<6xi32>
2  %p = vector.extract %v[5] : i32 from vector<6xi32>

```

(e) Lowered MLIR program (for lines 10-11 in Figure 1(b)) by applying “-lower-affine”

```

1  %v = affine.vector_load %m1[%idx9 mod m1.dim(0)] : memref<?xi32>, vector<6xi32>
2  %p = llvm.extraclement %v[5] : vector<6xi32>

```

(f) Lowered MLIR program (for lines 10-11 in Figure 1(b)) by applying “-convert-vector-to-llvm”

Fig. 1. Motivating example.

MLIR provides a diverse set of **optimization passes**, each designed to enhance an MLIR program by improving performance, reducing resource consumption, or simplifying its structure while preserving its semantics. These passes operate at various levels of abstraction and can be applied before or after lowering passes to refine the program and optimize execution efficiency. For example, the “-arith-unsigned-when-equivalent” pass optimizes the program by replacing signed *arith* operations to equivalent unsigned *arith* operations. For instance, in Figure 1(a), the signed division operation *arith.divsi* in line 6 is replaced by the unsigned division operation *arith.divui*, as both operands are positive, as shown in Figure 1(c). Note that in our work, specifying lowering passes for a given MLIR program aims to compile it into an executable representation, ensuring



successful execution. In contrast, specifying optimization passes facilitates **cross-optimization differential testing** by generating multiple executable versions of the same MLIR program under different optimization strategies, helping to expose silent bugs.

Compiler bugs are generally categorized into two main types: crash bugs and silent bugs (also known as wrong code bugs) [24]. Currently, no existing testing techniques are capable of detecting silent bugs in the MLIR compiler infrastructure, primarily due to the challenge posed by undefined behavior. **Undefined behavior (UB)** refers to program constructs that result in unpredictable execution outcomes due to violations of a language's semantics or underlying constraints [24]. Unlike traditional programming languages, MLIR is an extensible compiler infrastructure with diverse dialects, each enforcing specific rules on operations, memory management, and data flow. UB can arise from various sources, such as uninitialized or out-of-bounds memory accesses, invalid type conversions, or violations of dialect-specific constraints (e.g., shape mismatches in *tensor* operations). For example, the `affine.vector_load` operation in line 2 of Figure 1(a) demonstrates undefined behavior. In this case, the remaining space in `%m` starting from index `%idx9` (value == 9) is 5, which is insufficient to accommodate the required vector size of 6. These issues present a major obstacle to silent bug detection in MLIR, as they can lead to non-deterministic behavior, masking actual compiler bugs or causing false positives during differential testing. Therefore, addressing UB is crucial to ensuring the reliability of MLIR-based compilation workflows and enabling effective silent bug detection.

## 2.2 A Motivating Example

Figure 1(a) illustrates an MLIR program with undefined behavior. Specifically, In Line 2, the operation `affine.vector_load` attempts to read a value of type `vector<6xi32>` from a memory reference value (`memref<14xi32>`) `%m`, starting at position `%idx9` (value == 9). Undefined behaviors may occur when the `affine.vector_load` operation encounters either of the following two conditions: (1) Invalid index for the `%idx9`: If `%idx9` exceeds the bounds of `%m`, undefined behavior will occur. (2) Insufficient remaining space for the `%m`: If the remaining space in `%m`, starting from `%idx9`, is insufficient to accommodate the vector being loaded (`%v`), undefined behavior will also occur. In this case, the operation requires space for 6 elements, but only 5 elements remain from `%idx9` in `%m`. This out-of-bounds access leads to undefined behavior, causing the loaded vector `%v` to contain unreliable values. These unreliable values propagate through subsequent operations (e.g., `vector.extract` in Line 3), ultimately affecting the execution result (assuming `%p` is printed). Such undefined behavior makes all existing testing techniques hard to detect silent bugs due to unreliable execution results stemming from genuine optimization errors or undefined behaviors. To detect silent bugs, it is essential to remove all undefined behavior in the MLIR programs. Hence, DESIL is proposed. It solved this question by eliminating all undefined behavior and the updated program is shown in Figure 1(d), where modified code sections are highlighted in red. Specifically, DESIL begins by inserting runtime checkers (marked by ① in Figure 1(d)) that verify the volume of problematic `memref %m`. These checks calculate both the `memref`'s volume `Volume(%m)` and required vector space `Volume(vector<6xi32>)`, then compare them with the flattened index after index-bounding (`%BoundedFlattenIdx`) to validate sufficient capacity. When insufficient space is detected, the system generates a safe operand (marked by ② in Figure 1(d)) by allocating and initializing a properly-sized `%new-m`; otherwise, it preserves the original `%m` (Lines 3–4). Finally (marked ③), all unsafe operands are replaced with their verified versions. DESIL achieves this by replacing problematic operands `%m` with `%m1`, and applying index bounding through modulo arithmetic (`%idx9 mod m1.dim(0)`) to ensure memory safety while maintaining program semantics. Through the above steps, DESIL eliminates the undefined behavior, producing a UB-free MLIR program. This updated program is now suitable for differential testing.

After obtaining the UB-free program, another challenge is to lower the MLIR program into an executable form by converting all dialects in the program into their executable forms. Since numerous operations may coexist within an MLIR program, and new dialects or operations can be generated during the lowering process, it is crucial for DESIL to select an appropriate lowering path tailored to the given MLIR program. For instance, consider the operations `affine.vector_load` and `vector.extract` in Figure 1(a): while `vector.extract` can be directly lowered to `llvm` using “-convert-vector-to-llvm” (shown as Figure 1(e)), `affine.vector_load` first requires conversion via “-lower-affine” followed by “-convert-vector-to-llvm”. That is, applying passes in the wrong order creates inefficiencies. Specifically, prematurely using “-convert-vector-to-llvm” leaves `affine.vector_load` unresolved (shown as Figure 1(f)), forcing redundant pass reapplications. The optimal approach first converts `affine.vector_load` to `vector.load` form using “-lower-affine”, then handles all `vector` (i.e., `vector.load` and `vector.extract`) operations in a single “-convert-vector-to-llvm” pass. To find a suitable lowering path for given MLIR program, DESIL further optimizes the lowering process by introducing a lowering-path optimization algorithm to convert all dialect in the given MLIR program into an executable form, which finally supports the differential testing.

### 3 Approach

In this section, we introduce the methodology of our approach, named DESIL. It is the first technique that is specially designed for detecting silent bugs in the MLIR compiler infrastructure as far as we are aware. As introduced in Section 1, DESIL incorporates two major innovative components, i.e., **Undefined Behavior Elimination** (Section 3.1) and **Lowering Path Optimization** (Section 3.2), to address the challenges of undefined behaviors and inefficiency of dialect lowering for generating executable MLIR programs.

Figure 2 presents the overall workflow of our approach. Specifically, DESIL comprises a set of undefined behavior elimination rules, which effectively eliminate undefined behaviors for diverse UB-prone MLIR operations under certain conditions. Subsequently, to achieve lowering path optimization, we have defined a set of *operation-specific lowering paths* in DESIL for effectively transforming each dialect operation into the executable form through analyzing the corresponding documentation. By following this, DESIL performs the lowering process via dynamically performing *topological sorting* over all involved passes required by the dialect operations in the current MLIR program, thereby exploring the optimal *lowering path* for efficient transformation. Particularly, to evaluate whether the MLIR program is correctly compiled by the MLIR compiler, DESIL leverages **differential testing** mechanism to detect inconsistent checksum values of the same MLIR program across different optimization passes. We will introduce the detailed process in Section 3.3. Finally, we will outline the complete bug detection process in Section 3.4.

#### 3.1 Undefined Behavior Elimination

Undefined behaviors (UBs) in MLIR programs for testing the MLIR compiler infrastructure can lead to unpredictable execution results, making it difficult to accurately detect silent bugs since it is hard to determine whether the unexpected execution results of the MLIR programs are induced by their inherent UBs or the silent bugs in the compiler. Therefore, eliminating the undefined behaviors is essential. However, ensuring that the generated MLIR programs are UB-free is challenging due to the diverse root causes of such behaviors, especially those that are specific to MLIR programs. For instance, MLIR programs may easily cause UBs that are due to the shape (or dimensions) inconsistency while involving array-like values (e.g., tensor), requiring effective UB elimination methods tailored to handle such cases. To address this challenge, we conducted a comprehensive analysis of those operations that are supported and frequently used by

Table 1. Undefined behavior elimination rules for different operations under certain conditions.

| Operation Type  | Example  | Conditions   | UB Elimination Rules  |
|---|--|--|---|
| <b>UBs from Shape Inconsistency<sup>†</sup></b>             |  |  |   |
| affine.yield.   | affine.for<br>iter_args(%arg0=%m)<br>affine.yield %m1                            | The operand (%m1) shape differs from argument (%m) shape of parent operation (affine.for).   | Replace %m1 with a new value that has the same shape as %m.   |
| linalg.broadcast.   | linalg.broadcast<br>ins(%0)  outs(%1)<br>dimensions=[1]                          | The shapes of the two operands (%0 and %1) are not same, except for the dimensions in dimensions.  | Replace %0 with a new tensor that has the same shape as %1, except for the dimensions in dimensions.  |
| linalg.generic.   | linalg.generic {<br>iterator_types =<br>["p", "r"] } ins(%1,<br>%2, %3) outs(%4) | <b>C1:</b> The shapes of ins operands (%1, %2, %3) differ. <b>C2:</b> The dimension sizes of ins operands (%1, %2, %3) specified as p in iterator_types differ from the outs operand (%4).   | For <b>C1</b> , ensure shapes of %1, %2, and %3 are same. For <b>C2</b> , replace %1, %2, and %3 with new tensors that have same shape as %4 except for dimensions marked as r.   |
| linalg.matmul.  | linalg.matmul<br>ins(%1, %2)<br>outs(%3)   | <b>C1:</b> The 2nd dimension of 1st ins operand (%1) differs from the 1st dimension of 2nd ins operand (%2). <b>C2:</b> The 1st dimension of 1st ins operand (%1) differs from the 1st dimension of outs operand (%3). <b>C3:</b> The 2nd dimension of 2nd ins operand (%2) differs from the 2nd dimension of outs operand (%3). | Replace %1 and %2 with new tensors. For <b>C1</b> , ensure the 2nd dimension of %1 equals to 1st dimension of %2. For <b>C2</b> , ensure the 1st dimension of %1 equals to 1st dimension of %3. For <b>C3</b> , ensure the 2nd dimension of %2 equals to 2nd dimension of %3. |
| linalg.transpose.   | linalg.transpose<br>ins (%1) outs (%2)<br>permutation=[1, 0]                     | The dimension of the ins operand (%1) differs from the corresponding dimension of the outs operand (%2) as specified in permutation.   | Replace %1 with a new tensor that has the same dimension as %2 specified in permutation.  |
| Cast Operations (2).  | %m1 = memref.cast<br>%m0 : memref<?xi32><br>to memref<10xi32>                    | The static dimension of result (%m1) differs from corresponding runtime dynamic dimension size of operand (%m0).   | Replace %m0 with new memref with dynamic dimension equals to corresponding static dimension of %m1.   |
| Operations with same shape operands (3).                    | linalg.copy ins(%0)<br>outs(%1)  | The shapes of two operands (%0 and %1) differ.   | Replace %0 with new tensor that has the same shape as %1.   |
| <b>UBs from Index Out-of-Bounds</b>                         |  |  |   |
| Scalar Value Load and Store Operation (8)                   | affine.store %0,<br>%m[%idx1, %idx2]   | Any index (%idx1, %idx2) exceeds the dimensions of the array-like operand (%m).  | Confine the index values to the dimensions of %m using index.remu.  |
| Dim Operations (2)  | tensor.dim %t, %0  | The index (%0) exceeds the rank of array-like value (%t).  | Confine %0 within the rank of %t.   |
| Array-Like Value Store and Load Operations (2) <sup>†</sup> | affine.vector_store<br>%0,%c[%idx0,%idx1]  | <b>C1:</b> Any index (%idx0, %idx1) exceeds the dimensions of the data container (%c). <b>C2:</b> The remaining space in container (%c) starting from position (%idx0, %idx1) is insufficient to load/store the object (%0).   | For <b>C1</b> , confine the index to the dimensions of %c using index.remu operation. For <b>C2</b> , replace the container (%c) with a new one with sufficient space.  |
| <b>UBs from Invalid Memory References</b>                   |  |  |   |
| memref.assume_alignment. <sup>†</sup>                       | memref.assume_<br>alignment %m, 4  | The alignment attribute (4) differs from the original alignment information of the operand (%m).   | Replace alignment attribute with alignment information of %m, or default value if %m has no information.  |
| memref.realloc.   | %m1 = memref.<br>realloc %m0   | The original memref value (%m0) is used after this operation.  | Replace the use of %m0 after this operation with a new memref value.  |
| Memory Allocation Operations.                               | %0 = memref.alloca<br>() : memref<1xi32>   | Directly use the value in the result memref value without initialization.  | Initialize the content of memref value with linalg.fill.  |
| <b>UBs from Scalar Calculations</b>                         |  |  |   |
| Shift Operations (6).                                       | index.shrui %1, %2   | The 2nd operand (%2) exceeds the bit width of the 1st (%1).  | Replace %2 with a random constant value within the bit width of %1.   |
| Signed Integer Division Operations (6)                      | index.divs %1, %2  | <b>C1:</b> The 2nd operand (%2) is zero. <b>C2:</b> The 1st operand (%1) is INT_MIN (specific to its bitwidth), and the 2nd operand (%2) is -1.  | For <b>C1</b> , make %2 unequal to 0. For <b>C2</b> , make %2 unequal to -1.  |
| Unsigned Integer Division (4) and Remainder Operations (4)  | index.divu %1, %2  | The 2nd operand (%2) is zero.  | Replace %2 with nonzero random signed or unsigned integer value (according to the type of %2).  |

<sup>†</sup> The conditions and elimination rules of these undefined behaviors are MLIR-specific.



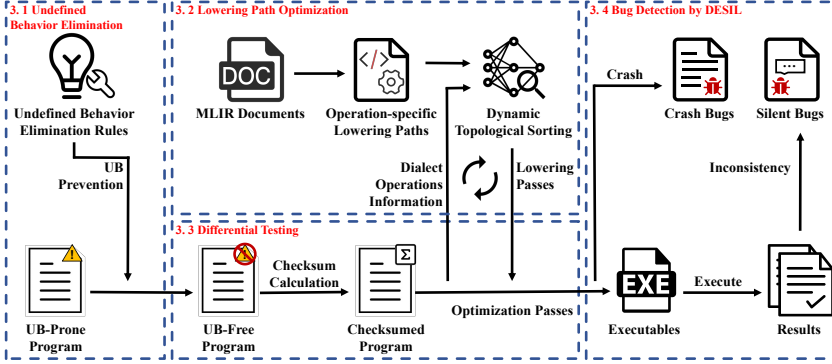


Fig. 2. Overview of DESIL

existing MLIR fuzz techniques and summarized the potential UBs they may induce by carefully examining their usage documentation. We refer to operations that may cause undefined behaviors as *UB-prone* operations. Specifically, DESIL supports 127 operations across 10 dialects covered by the state-of-the-art MLIR test generator, MLIRSmith. They are core dialects for MLIR-based compilation, as identified in prior work [19, 21]. Based on the conditions that trigger potential UBs for each UB-prone operation, we manually defined a set of undefined behavior elimination rules to modify programs and ensure that UBs cannot be triggered. Based on the UB identification process mentioned above (i.e., performing a systematic review of official documentation for each supported operation in core dialects), these UB-elimination rules target common UBs in practice.

The details of the undefined behavior elimination rules are presented in Table 1. In this table, we list the types of UB-prone operations, followed by an example to clearly present the conditions for triggering potential UBs, and then we summarized the undefined behavior elimination rules to eliminate the trigger of the UBs. In particular, one operation type may involve multiple UB-prone operations, which share similar root causes and undefined behavior elimination rules. The number in the brackets shown in the first column indicates the number of involved operations belonging to the specific operation type. For clarity, we only present one representative example in the table to aid the understanding and illustration. The complete operations and their associated undefined behavior elimination rules can be found at our project’s homepage [2].

Consequently, given an MLIR program for testing MLIR compilers, DESIL first identifies all UB-prone operations within it. For each identified operation, DESIL applies the corresponding undefined behavior elimination rule to generate the correct program. It is important to note that since the lowering process (as discussed in Section 3.2) should maintain the semantics of the MLIR program, i.e., UB-free programs should not encounter any UBs after the transformation. As a consequence, the fix process is a one-off task for each MLIR program. By sufficiently fixing all potential UBs in the initial MLIR programs, our approach ensures comprehensive mitigation and ensures that UBs cannot be triggered in the target executable programs. In the following, we will provide a proof-of-concept introduction of these undefined behavior elimination rules. The detailed implementations for fixing each UB can be found in our open-source repository.

**3.1.1 Undefined Behaviors from Shape Inconsistency.** This type of UBs is typically due to the calculation related to vector-like values, such as matrices or tensors that are usually involved in deep learning programs. These UBs are usually triggered because the shapes of two tensors (or dimensions of matrices) do not match each other. Actually, this type of UBs is typically specific to MLIR programs due to their frequent use of array-like values. In contrast, traditional programming

languages typically decompose such operations into loops and scalar value computations, and thus are free from this type of UBs. For example, two tensor values `[1, 2]` and `[1, 2, 3]` cannot perform multiplication since their shapes (or dimensions) are unmatched. Specifically, we summarized three situations where the shape inconsistency may cause potential UBs: (1) shape inconsistency between arguments and return values, such as the operation of `linalg.matmul ins(%1,%2) outs(%3)` requiring the dimensions of arguments (i.e., `%1` and `%2`) and the return value (i.e., `%3`) to match each other; (2) Shape inconsistency in a specified dimension, such as the argument dimension in `linalg.transpose` is inconsistent with the output dimension specified by `permutation=[1, 0]`; (3) Shape inconsistency between source and target operand, such as `linalg.copy` should not change the value dimension during copying.

To address these inconsistencies, our approach will insert shape-related runtime checking code as the checker of memory allocation explained above, and generate suitable operands for replacement if any inconsistency was found. It is important to note that, in general, DESIL avoids modifying the shape of the return value of operations, as it tends to affect all follow-up uses of the result value, and thus increases the risk of the modified MLIR program being rejected by the MLIR front-end due to checks related to shape. As a consequence, DESIL will always update the shapes of the others except for the return value. Different from the above situations, the `tensor.empty` operation is commonly used by some MLIR fuzz testing techniques (e.g., MLIRSmith) for generating MLIR programs. However, as explained in the corresponding documentation, this operation may cause unpredictable results since its values are unpredictable. To avoid UBs induced by it, DESIL replaces all appearance of `tensor.empty` with either `tensor.from_elements` or `tensor.splat` for initializing new tensors.

**3.1.2 Undefined Behaviors from Index Out-of-Bounds.** This type of UBs is prevalent across various programming languages, occurring when an MLIR program attempts to access a memory location exceeding the bound of a valid range. This UB is critical as it always results in unreliable execution results and execution crashes, and thus should be eliminated. Like many other programming languages, this kind of UBs in MLIR programs usually happen in two scenarios: (1) accessing an array-like value with the specified index (e.g., `tensor.dim`), (2) and storing an object to a data container without sufficient available memory space (e.g., `affine.store`). For the first scenario, our straightforward idea for fixing is to check whether the given index exceeds the range of the array-like value, and then replace the index with a valid value within the range. For the second scenario, the undefined behavior elimination rule is to allocate another memory to ensure the available space is sufficient for storing the object. It is important to note that this scenario is specific to MLIR programs due to its high-level abstraction of data types [1].

In particular, checking whether the remaining memory is sufficient is not statically doable since the memory will be dynamically allocated and consumed during the running of the MLIR program. Therefore, to ensure the fix is valid and effective, sometimes we are expected to insert new code logic for dynamically checking the triggering conditions of certain UBs and eliminating them on demand. For example, as shown in Table 1, the operation (`affine.vector_store %o,%c[%idx0,%idx1]`) is designed to store a vector object (i.e., `%o`) into the data container `%c` starting from position `[%idx0,%idx1]`. In this case, to ensure the store operation is correctly performed, our approach will insert multiple lines of code for dynamically checking the size of `%o` and the available memory of `%c`, and allocate additional memory if needed. In this way, memory is guaranteed to meet the requirement during the execution of the MLIR program, and thus the undefined behavior can be avoided.

**3.1.3 Undefined Behaviors from Invalid Memory References.** This type of UBs is primarily caused by the invalid references to memory, and the associated operations are commonly from the *memref* dialect. Similarly, these UBs may cause crashes during running the MLIR program or produce unpredictable results. In summary, the root causes of these UBs are twofold: (1) conflict between actual memory alignment and specified alignment assertions by using the operation `memref.assume_alignment`; (2) reference to invalid memory, such as accessing uninitialized or reallocated memory by `memref.realloc`. Regarding the first root cause, DESIL will update the specified alignment attribute in the assertion operation and make it align with the actual value. Regarding the second root cause, DESIL incorporates a define-use chain analysis [6, 11] for identifying the invalid memory access, and then updates the invalid references to a valid one or initializes the referenced memory directly. In particular, to avoid out-of-memory crashes caused by the memory allocation operations (e.g., `memref.alloc` and `memref.alloca`) during continuously allocating memory space, DESIL restrains the dimensions of an array-like operand not exceeding  $4 \times 32$ .

**3.1.4 Undefined Behaviors from Scalar Calculations.** This type of UBs is primarily caused by the operations related to scalar value calculations. In particular, it mainly includes three root causes – shift overflow (e.g., `index.shrui`), signed integer overflow (e.g., `index.divs`), and division by zero (e.g., `arith.ceildivsi`). Effectively eliminating this kind of UBs is crucial since they always cause crashes or unpredictable execution results while executing the compiled MLIR program, disabling the precise detection of silent bugs. To address these UBs, we have defined a viable undefined behavior elimination rule for each kind of root cause (as presented in Table 1). For example, in the division operations (e.g., `arith.ceildivsi`), undefined behavior will arise due to division by zero or signed division overflow (e.g., dividing the minimum signed integer value by -1). To fix this, we first check whether the divisor operand in this operation is zero or not through either static or dynamic analysis, and then replace zeros with a randomly generated integer value unequal to zero. Similarly, in shift operations (e.g., `index.shl` and `arith.shli`), if the second operand (i.e., the bits of shifting) exceeds the bitwidth of the initial value (i.e., the first operand), an unpredictable value will be returned. In this case, the undefined behavior elimination rule is to confine the value of the second operand within the bitwidth of the first operand. In this way, the UBs can be effectively avoided. Different from existing methods (e.g., Csmith), typically adopting predefined safe wrapper functions, for avoiding UBs from scalar calculations, DESIL directly seeds the UB checking and elimination logic into the initial MLIR program. As a consequence, our approach can effectively reduce the code size compared to existing methods by solely generating relevant elimination logic for used bitwidths, which significantly improves the lowering efficiency (will be introduced in Section 3.2) by avoiding much irrelevant code involved by predefined wrapper functions.

**3.1.5 UB-Irrelevant Fix for Normal Compilation.** Besides preventing undefined behaviors presented above, there is another unique case – 0-dimension objects – that arises from MLIR’s rich semantics supporting 0-dimensional constructs some array-like data types such as tensors and memrefs. This may cause the dialect lowering process (i.e., compilation) failed since some low-level dialects (e.g., vector) do not support the dimension of objects to be zero. As a consequence, DESIL further incorporates an additional undefined behavior elimination rule for such cases. Specifically, DESIL replaces 0-dimension objects with non-0-dimension objects to ensure the MLIR program can be successfully transformed into the executable ones.

**UB Elimination Algorithm:** Based on the undefined behavior elimination rules introduced above, given an MLIR test program, DESIL tries to fix all potential UBs in it by following the process presented in Algorithm 1. In general, the algorithm takes an MLIR program that may contain UB-prone operations as the input, and outputs a new MLIR program that is expected to be UB-free by applying the necessary undefined behavior elimination rules explained above.

**Algorithm 1:** Undefined Behavior Elimination**Input** : *Program*: an MLIR program that may contain UB-prone operations**Output** : *Program*: a UB-free MLIR program after applying undefined behavior elimination rules

---

```

1 Function FixUB(Program):
    /* Collect all UB-prone operations from the top level operation of Program. */
2    UBProneOps = CollectUBProneOps(Program.TopLevelOp)
3    foreach op in UBProneOps do
        /* Eliminate potential UBs in each operation by applying the associated undefined behavior
        elimination rule. */
4        program = UBEliminationForOp(program, op)
5    end
    /* Conduct UB-Irrelevant fix for normal compilation. */
6    program = UBIrrelevantFix(program)
7 return Program

8 Function UBEliminationForOp(program, op):
    /* Insert runtime checkers to the MLIR program for detecting and avoiding potential UBs. */
9    program = InsertRuntimeCheck(program, op)
    /* Generate the safe version of operands that eliminate the UB based on the runtime checker. */
10   program, safeOp = PrepareSafeOperands(program, op)
    /* Replace the old UB-prone operands with corresponding UB-free operands. */
11   program = ReplaceOldOperandWithUBFree(program, safeOp)
12 return Program

13 Function CollectUBProneOps(operation):
14   UBProneOps = [];
    /* Recursively collect the UB-prone operations if the operation has Block. */
15   if operation.hasBlock() then
16       for o in operation.getBody() do
17           | UBProneOps = UBProneOps  $\cup$  CollectUBProneOps(o)
18       end
19   end
    /* Collect UB-prone operations that may cause UBs according to their operation type. */
20   if IsUBProneOpType(operation) then
21       | UBProneOps.add(operation)
22   end
23 return UBProneOps

```

---

Specifically, given the input MLIR *Program*, DESIL first collects all UB-prone operations in it (Line 2). In particular, an MLIR program comprises a set of operations, which are typically structured as recursively nested code regions like traditional programming languages (e.g., While statements usually include other statements in a Java or C++ program). That is, an operation may have one or multiple nested regions (known as Blocks) [5], each of which will also consist of a list of operations. By following this structure, DESIL recursively traverses the MLIR program in a top-down fashion (*Program.TopLevelOp* indicates the outmost operation in the program) for collecting all UB-prone operations within the program (Lines 15-22). Then, for each UB-prone operation (Line 3), DESIL tries to eliminate the potential UBs by applying the associated undefined behavior elimination rules (Line 4) by calling the function of *UBEliminationForOp*() (Lines 8-12). Specifically, DESIL inserts necessary runtime checkers into the MLIR program for checking whether any conditions associated to the UB-prone operation are satisfied during the MLIR program execution (Line 9), and then generates safe operands based on the checking results and the associated undefined

behavior elimination rules (Line 10) for replacing the original operands used in the operation (Line 11). For example, to fix the potential UBs from index-out-of-bounds errors caused by operation `affine.vector_store %o,%c[%idx0,%idx1]`, DESIL inserts several lines of code (i.e., runtime checker) to the MLIR program for check whether the available memory in `%c` is sufficient to store the object `%o`. If it is insufficient, DESIL inserts new operand (e.g., `%d`) with sufficient memory in the MLIR program. Finally, DESIL replaces the original operand `%c` with `%d`. The overall process concludes with the undefined behavior-irrelevant fixes for normal compilation, as introduced in Section 3.1.5. In this way, the modified MLIR program will include those runtime checkers that can effectively avoid the trigger of potential UBs during the execution of the MLIR program.

### 3.2 Lowering Path Optimization

Given an MLIR program, after eliminating the undefined behaviors in it presented above, the next process is to transform the program into executable form. As aforementioned, this transformation process is indeed challenging because MLIR programs usually contain operations from diverse dialects at different levels, and each dialect operation may require a specific sequence of lowering passes – *operation-specific lowering path* – before reaching a fully executable form. Furthermore, new dialect operations will be dynamically produced during the lowering process, aggravating the difficulty of this process. However, exhaustively enumerating all possible sequences of lowering passes is impractical since it would impose a significant efficiency bottleneck, and thus affect the effectiveness of silent bug detection.

To address this challenge, DESIL incorporates an innovative lowering-path optimization algorithm that dynamically determines the optimal lowering pass based on the dialect operations included in the MLIR program. Specifically, DESIL first builds a mapping between lowering passes and dialect operations based on the MLIR documentation, recording an *operation-specific lowering path* for each operation. Then, given an MLIR program, DESIL determines its optimal lowering path by performing *topological sorting* on the lowering passes derived from the operation-specific lowering paths of the program's operations. In this way, DESIL can efficiently transform a given MLIR program into the executable form by avoiding the circular application of the same lowering passes. In summary, the MLIR program compilation (or lowering path optimization) process in DESIL consists of two key stages which are presented as follows:

- (1) **Operation-Specific Lowering Path Construction:** To ensure that every operation in the MLIR program can be successfully transformed into the executable form, DESIL builds the mapping between each dialect operation and a sequence of lowering passes, which can transform the associated operation into the executable form. Specifically, we call such a sequence of lowering passes *operation-specific lowering path*. Formally, it is defined as a tuple of  $\langle o, P, R \rangle$ , where  $P = \{p_1, p_2, \dots, p_n\}$  is a set of lowering passes that are needed to transform the operation  $o$  into the executable form, and  $R = \{p_i \succ p_j \mid p_i, p_j \in P\}$  defines the partial order between two lowering passes, specifying the pass  $p_i$  in  $P$  should be executed before  $p_j$  for transforming  $o$ . Figure 3 presents an example of the lowering process for the dialect operation `affine.for`, which will be transformed into the executable `llvm.cond_br`. Consequently, the *operation-specific lowering path* associated with the operation `affine.for` is  $\langle \text{affine.for}, \{p_1, p_2, p_3\}, \{p_1 \succ p_2, p_2 \succ p_3\} \rangle$ . Specifically, the output of this stage is the *operation-specific lowering path* for each operation. In particular, to ensure the reliability of mapping results, we manually analyzed the documentation of operations and lowering passes. Moreover, we verified all the operation-specific lowering paths by constructing associated MLIR programs to ensure they actually work.

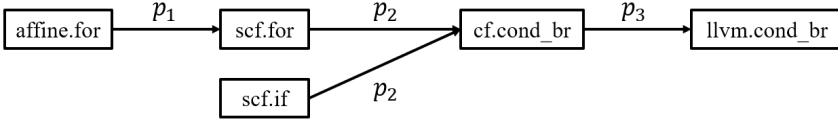


Fig. 3. The lowering process for the dialect operation `affine.for` by applying passes  $\{p_1, p_2, p_3\}$ .

- (2) **Lowering Pass Topological Sorting:** Given that MLIR programs may include a variety of operations and each of them is associated with an *operation-specific lowering path*. To achieve an efficient lowering process (i.e., compilation) and avoid circular application of the same lowering pass, DESIL exploits the optimal pass execution by leveraging *topological sorting* over all the passes associated to all the operations in the current MLIR program. Formally, assuming  $O$  represents all the operations in the program, and  $P$  represents all the passes involved. Then,  $\forall p_1, p_2 \in P$ ,  $p_1 \succ p_2$  holds *iff* it holds for any  $o \in O$ . According to this, DESIL always chooses the pass  $p_i \in P$  as the first one for execution *iff*  $\nexists p_j \in P, p_j \succ p_i$ . It is important to note that such a pass  $p_i$  always exists since each dialect operation is guaranteed to be transformed into the executable form by the corresponding *operation-specific lowering path*, indicating no circular dependency exists for all the passes. However, if more than one pass meets the condition, DESIL randomly chooses one of them. Figure 3 presents such an example, where the lowering pass  $p_1$  will be the first lowering pass for execution.

In summary, the first stage (i.e., Operation-Specific Lowering Path Construction) is a one-off task. Once the operation-specific lowering paths are constructed, they can be directly reused during the compilation process for diverse MLIR programs. In contrast, the second stage will be dynamically performed by DESIL – choosing a lowering pass to execute for transforming the MLIR program into another form – until all the operations in the program are transformed into the executable form. In particular, the lowering-path optimization of DESIL requires only knowledge of which passes transform operations into executable form. It remains effective even though internal changes occur in passes or dialects, as long as the overall transformation relationships between passes and dialects stay intact. This ensures robustness to compiler evolution while preserving high-level lowering capabilities.

### 3.3 Differential Testing

After transforming the given MLIR program into an executable form, the next step is to run the program and examine whether it was correctly compiled by the MLIR compiler. However, like all fuzz testing techniques, it is infeasible to automatically obtain the oracles of the test execution without the specification of the test program [22]. To address this issue, DESIL employs differential testing to check whether the MLIR program is correctly compiled by the compiler. To achieve that, DESIL comprises a compilation operation-aware optimization recommendation strategy along with the lowering pass topological sorting (introduced in Section 3.2). The objective of this strategy is to produce different test execution results by applying diverse optimizations during compiling the same MLIR program, where potential silent bugs in the MLIR compiler would be detected. Specifically, given an MLIR program, DESIL collects all the operations involved in it and then selects *optnum\_each* (which is evaluated in Section 4.5) optimization passes according to the collected operations. Then, DESIL checks the execution results of these different executions (DESIL by default generates *diffnum* executable programs for each MLIR program). Different execution results among them indicate incorrect compilations, and represent the detection of silent bugs in compilers.



However, unlike traditional test programs for high-level programming languages (e.g., Java and C++), which usually associate with a relatively complete functionality, the MLIR programs may not perform a meaningful function as they are usually generated in a random fashion by assembling low-level dialect operations. As a consequence, the final output of the MLIR programs may not well reflect their whole execution behaviors. This issue potentially reduces the detection capability of silent bugs in MLIR compilers since the mis-compiled operations may not affect the final execution results. To improve the capability of the MLIR program for detecting silent bugs, DESIL further incorporates an “test oracle” generation process inspired by Csmith [24] – A well-known fuzz testing method for C/C++ compilers. Specifically, DESIL calculates the checksum of the MLIR program by summing up all the accessible integer values (values in array-like objects are also included) at the end of the MLIR program’s main function. In particular, DESIL does not consider floating-point values since the precision issue during calculation may cause false positives in bug detection. Finally, the checksum will play as an estimation of the test “oracle”. Since any integer value error will propagate to the final checksum, it should have a strong power to uncover incorrect execution results, thereby improving the capability of detecting silent bugs in MLIR compilers.

### 3.4 Bug Detection by DESIL

Given an MLIR program, DESIL first transforms it to the executable forms by adopting different sequences of optimizations. Then, it runs the test programs and compares the checksum correspondingly, and reports potential silent bugs if the checksum values are inconsistent. More specifically, the overall silent bug detection process of DESIL consists of four stages, which are presented as follows.

- (1) **Test Program Generation:** In order to generate diverse MLIR programs for detecting silent bugs in MLIR compilers, DESIL utilizes an MLIR program generator. It is important to note that DESIL is a post-processing method for MLIR program generators, and thus can be combined with any off-the-shelf generators as a plugin.
- (2) **Undefined Behavior Elimination:** For each candidate MLIR program, DESIL fixes the undefined behaviors in it by leveraging the UB elimination algorithm presented in Section 3.1. As aforementioned, this process is essential to ensure the capability of precisely detecting silent bugs since UBs are easy to produce false positives.
- (3) **Lowering Path Optimization:** After eliminating potential UBs in the MLIR programs, DESIL leverages the lowering path optimization component (introduced in Section 3.2) to transform the programs into executable forms.
- (4) **Differential Testing for Bug Detection:** Given different executable programs compiled from the same MLIR program, DESIL generates the calculations of the checksum for each one. For differential testing, DESIL leverages the operation-aware optimization recommendation component for generating different versions of the MLIR programs. Finally, DESIL executes the programs and detects potential silent bugs by checking the consistency of their associated checksum values. In particular, DESIL also has the ability to detect crash bugs in MLIR compilers during the compilation process if any crashes are encountered.

## 4 Evaluation

To evaluate DESIL, we designed the following research questions (RQs) in the study:

- RQ1: How effective is DESIL in detecting previously unknown MLIR bugs?
- RQ2: How does DESIL perform compared to the state-of-the-art MLIR testing techniques?
- RQ3: How does each component contribute to the overall effectiveness of DESIL?
- RQ4: What is the influence of different configurations on the effectiveness of DESIL?

## 4.1 Experimental Setup

To answer RQ1, we applied DESIL to fuzz the latest versions of the MLIR compiler infrastructure, aiming to uncover previously unknown bugs. Over a four-month fuzzing period, we consistently updated the infrastructure to the latest version, covering revisions from adbf21 to b6d5fa. To answer RQs 2-4, we selected the latest version of the MLIR compiler infrastructure at the time of performing these experiments (i.e., revision c6d6da). We ran each studied technique for 12 hours on this version. To reduce the influence of randomness involved in testing, we repeated each technique for five times (except the variant techniques investigated in RQ4) and reported the aggregated results. Due to the large number of studied variant techniques in RQ4 and the fuzzing cost for each technique, we repeated them for three times to balance the conclusion robustness and evaluation cost, and then reported the aggregated results.

By default, we set the number of optimization passes per lowering step (*optnum\_each*) to 1 and the number of compilations for differential testing (*diffnum*) to 2 in DESIL for seeking cost-effectiveness. The influence of different settings for them will be investigated in RQ4. 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 Studied Techniques.** Due to the pluggable design of DESIL presented before, DESIL can be combined with any existing MLIR program generation tools. That is, for any given MLIR program, DESIL can be applied to transform it into a UB-free one and then compile it to the executable program for testing. In the study, to ensure the generalizability of DESIL, we used two state-of-the-art MLIR program generation tools (i.e., MLIRSmith [21] and MLIRod [19]) to prepare initial MLIR programs for DESIL, respectively. For ease of presentation, we call the two instantiations **DESIL<sub>smith</sub>** and **DESIL<sub>od</sub>**.

Since the MLIR programs generated by MLIRSmith and MLIRod may contain undefined behaviors, neither includes a lowering component to transition these programs into executable forms. As a result, their original versions cannot detect silent bugs. To enable a more comprehensive comparison, we integrated lowering path optimization and differential testing components from DESIL into MLIRSmith and MLIRod, equipping them with the capability to detect silent MLIR bugs. To distinguish these enhanced versions from their originals, we refer to them as **MLIRSmith<sub>enhanced</sub>** and **MLIRod<sub>enhanced</sub>**, respectively. Specifically, these variants retain their original program generation mechanisms but follow DESIL's compilation process to produce executable programs for differential testing. However, since MLIRSmith<sub>enhanced</sub> and MLIRod<sub>enhanced</sub> do not eliminate UB, they may produce a high number of false positives in silent bug detection. Comparing them against DESIL allows us to evaluate RQ2 and also demonstrates the contribution of the UB elimination component in DESIL.

Besides the above-mentioned UB elimination component, there are another two main components in DESIL - the lowering path optimization and differential testing components. Their contributions will be investigated in RQ3. To investigate the contribution of the lowering path optimization component, we constructed the corresponding variants **DESIL<sub>smith</sub><sup>w/o lower</sup>** and **DESIL<sub>od</sub><sup>w/o lower</sup>** by removing this component from DESIL. Specifically, these variants randomly select a sequence of lowering passes to construct a lowering path for each MLIR program after UB elimination. To prevent the lowering process from hanging due to the application of an excessive number of passes, we set the lowering path length to 50, which is significantly larger than the average length of the determined lowering paths in DESIL during our study.

Regarding the differential testing component, DESIL modifies the application of optimizations to generate a set of executable programs for differential comparison. To enhance the effectiveness of differential testing, it employs a recommendation mechanism that selects optimization passes

Table 2. Details of previously unknown bugs detected by DESIL

| Bug Id | Type   | Root Cause <sup>1</sup> | Component               | Status    | Bug Id              | Type   | Root Cause | Component               | Status    |
|--------|--------|-------------------------|-------------------------|-----------|---------------------|--------|------------|-------------------------|-----------|
| 80960  | silent | MAES                    | Documentation           | fixed     | 114652              | silent | IRL        | Domain Specific(arith)  | duplicate |
| 81228  | silent | EAL                     | Domain Specific(arith)  | fixed     | 114654              | silent | IRL        | Domain Specific(arith)  | fixed     |
| 82158  | silent | IRL                     | Domain Specific(arith)  | fixed     | 114656              | silent | –          | Domain Specific(linalg) | confirmed |
| 82168  | silent | IRL                     | Domain Specific(arith)  | fixed     | 114657              | silent | –          | Domain Specific(linalg) | fixed     |
| 82622  | silent | –                       | Domain Specific(math)   | submitted | 115293              | silent | IRL        | Domain Specific(arith)  | fixed     |
| 83530  | silent | –                       | Domain Specific(arith)  | fixed     | 115294 <sup>2</sup> | silent | –          | Domain Specific(vector) | confirmed |
| 92057  | crash  | IP                      | General                 | fixed     | 115294 <sup>2</sup> | crash  | –          | Conversion              | confirmed |
| 94423  | crash  | IP                      | Domain Specific(arith)  | fixed     | 116664              | silent | –          | Domain Specific(scf)    | submitted |
| 94431  | silent | MAES                    | Domain Specific(arith)  | fixed     | 118224              | crash  | –          | Domain Specific(affine) | fixed     |
| 95246  | crash  | IRL                     | Conversion              | fixed     | 118225              | crash  | –          | Domain Specific(affine) | submitted |
| 102576 | crash  | IP                      | Conversion              | fixed     | 126195              | silent | –          | Domain Specific(arith)  | confirmed |
| 102577 | crash  | –                       | Conversion              | submitted | 126196              | crash  | IRL        | General                 | fixed     |
| 111241 | crash  | –                       | Conversion              | fixed     | 126197              | crash  | IP         | Domain Specific(vector) | fixed     |
| 111242 | crash  | IV                      | Conversion              | fixed     | 126213              | crash  | –          | General                 | confirmed |
| 111243 | crash  | IP                      | Domain Specific(linalg) | fixed     | 126371              | crash  | –          | Domain Specific(vector) | confirmed |
| 111244 | crash  | –                       | Domain Specific(vector) | submitted | 128273              | silent | –          | Domain Specific(affine) | submitted |
| 112878 | silent | –                       | Domain Specific(linalg) | fixed     | 128275              | crash  | IP         | Domain Specific(math)   | fixed     |
| 112881 | silent | MT                      | Domain Specific(linalg) | fixed     | 128277              | crash  | IP         | Domain Specific(affine) | fixed     |
| 113687 | silent | –                       | Domain Specific(affine) | submitted | 129414              | silent | IP         | General                 | fixed     |
| 113689 | silent | –                       | General                 | submitted | 129415              | silent | –          | Conversion              | submitted |
| 113690 | silent | –                       | Domain Specific(linalg) | fixed     | 129416              | crash  | IP         | General                 | fixed     |

<sup>1</sup> Full Names of Root Causes: IV (Incomplete Verifier), IP (Incorrect Pattern), MAES (Misalignment with Executable Semantic), EAL (Erroneous Analysis Logic), MT (Missing Trait), IRL (Incorrect Rewrite Logic).

<sup>2</sup> These two bugs were reported together.

based on the operations present in the MLIR program, rather than choosing them randomly. This mechanism increases the likelihood of optimizations affecting the program’s behavior, improving the chances of exposing silent bugs. Therefore, in RQ3, we also investigated the contribution of this optimization recommendation mechanism by constructing the corresponding variants **DESIL<sup>w/o opt smith</sup>** and **DESIL<sup>w/o opt od</sup>**, which remove this mechanism from DESIL<sub>smith</sub> and DESIL<sub>od</sub> respectively. Specifically, these variants randomly select optimization passes to generate a set of executable programs for differential testing, rather than leveraging operation-aware recommendations.

To answer RQ4, we configured the number of optimization passes per lowering step (*optnum\_each*) to {1, 3, 5, 7, 9} and the number of compilations for differential testing (*diffnum*) to {2, 4, 6, 8, 10}, respectively. Notably, when examining the influence of one hyperparameter, we maintain the default setting for the other to ensure an isolated analysis.

**4.1.2 Metrics.** Following the existing work on MLIR testing [19, 21], we used **the number of detected bugs** to measure the effectiveness of each studied technique. For this metric, de-duplication is a necessary step [9, 10, 23]. In the study, for crash bugs, we de-duplicated them based on crash messages following the existing work [18, 19]. For silent bugs, we de-duplicated them by analyzing their bug-triggering operations and passes, which are obtained based on delta debugging on both programs and passes [10, 23]. Then, we reported each bug to the developers for further confirmation. Based on their feedback, the used de-duplication mechanisms are indeed accurate to a large extent.

## 4.2 RQ1: Previously Unknown Bugs Detected by DESIL

Table 2 provides the details of the previously unknown bugs detected by DESIL, including the bug ID, type of bug, root cause, buggy component, and bug status. In total, DESIL detected 42 bugs, comprising 23 silent bugs and 19 crash bugs. Among these, 17 silent bugs and 16 crash bugs have been confirmed or fixed by developers. However, existing techniques, such as MLIRSmith and MLIRod, cannot theoretically detect silent bugs due to the potential UB in their generated test

```

...
1  %22 = arith.shrui %17, %true : i1
...
2  vector.print %22 : i1

```

(a) Program snippet for triggering Bug#80960 (silent)

```

...
1  %32 = arith.cmpi sge, %7, %31 : i16
2  vector.print %32 : i1

```

(b) Program snippet for triggering Bug#81228 (silent)

```

...
1  %mapped = linalg.map ins(%t1, %t1 : tensor<9xi32>, tensor<9xi32>) outs(%t2 : tensor<9xi32>)
2  (%in1: i32, %in2: i32) {
3    %2 = arith.constant 10 : i32
4    memref.store %2, %alloc[%c1] : memref<24xi32>
5    linalg.yield %2 : i32
6  }
7  %1 = memref.load %alloc[%c1] : memref<24xi32>
8  vector.print %1 : i32

```

(c) Program snippet for triggering Bug#112881 (silent)

Fig. 4. previously unknown bug examples detected by DESIL.

programs and the absence of a lowering mechanism to compile these programs into executable forms. The results underscore that DESIL is effective in exposing silent bugs, which is orthogonal to all existing testing techniques for the MLIR compiler infrastructure.

**4.2.1 Buggy Components Analysis.** From Table 2, we observed that the bugs detected by DESIL span various components within the MLIR compiler infrastructure. These bugs are detailed as follows:

**Documentation** defines the semantics of operations. Documentation changes can alter the semantics of MLIR operations' behavior. Bugs in this category cannot be detected by existing approaches (e.g., MLIRSmith and MLIRod) since crashes can not reveal semantic issues in MLIR programs since they are not executed. One bug detected by DESIL belongs to this category.

**Domain-Specific Passes** are designed to apply specialized optimizations to MLIR programs, targeting specific types of operations or dialects that are relevant to a particular domain. There are 28 bugs detected in these domain-specific passes by DESIL, covering 6 dialects. Specifically, there are 10 bugs in arith-dialect specific passes, 2 bugs in math-dialect specific passes, 6 bugs in linalg-dialect specific passes, 4 bugs in vector-dialect specific passes, 5 bugs in affine-dialect specific passes and 1 bug in scf-dialect specific passes.

**General passes** operate on MLIR programs at a more generic or broad level, typically affecting multiple dialects or operations. These passes are designed to be domain-agnostic, providing optimizations that are universally applicable rather than tailored to a specific use case. 6 bugs detected by DESIL in this category.

**Conversion Passes** transform higher-level dialects into lower-level dialects. When conversion passes contain bugs, they either cannot produce executable IRs or generate erroneous ones. 7 bugs detected by DESIL belong to this category.

These bugs are distributed across 4 categories with 9 different components of the MLIR compiler infrastructure, demonstrating the effectiveness of DESIL in bug detection.

**4.2.2 Root Cause Analysis.** Based on developers' feedback, we conducted a detailed analysis of the root causes of bugs detected by DESIL. In particular, out of the 27 bugs that have been fixed by developers, we have successfully identified the root causes for 21 bugs, while the remaining six were fixed silently and therefore excluded from this analysis. More concretely, through a careful analysis of these bugs, the root causes mainly involve six categories: Incorrect Rewrite Logic (IRL), Incomplete Verifier (IV), Incorrect Pattern (IP), Misalignment with Executable Semantic (MAES), Erroneous Analysis Logic (EAL), and Missing Trait (MT). We have also presented the analysis results in Table 2 (i.e., the third and eighth columns). Specifically, the first three categories have been introduced in existing studies [19, 21] – **Incorrect Rewrite Logic (IRL)** refers to flawed transformation logic within MLIR passes that rewrite matched operations into new forms. Seven bugs fall into this category. **Incomplete Verifier (IV)** refers to the absence, incompleteness, or inadequacy of a verifier associated with an MLIR pass. One bug belongs to this category. **Incorrect Pattern (IP)** refers to faulty patterns used by MLIR passes to match operations for transformation or optimization. Nine bugs have been attributed to this root cause. Additionally, our analysis also identified three new root causes from these DESIL-detected bugs: Misalignment with Executable Semantic (MAES), Erroneous Analysis Logic (EAL), and Missing Trait (MT).

**Misalignment with Executable Semantics (MAES)** refers to a discrepancy between the semantics described in MLIR's documentation or implementation and the actual semantics of the corresponding executable IR (i.e., LLVM IR). Such misalignment leads to incorrect conversions from MLIR to LLVM IR, causing any MLIR program containing the affected operations to produce incorrect outputs, thereby significantly undermining the reliability of the MLIR compiler infrastructure. Two bugs fall under this category.

For instance, issue#80960 (shown in Figure 4(a)) is a MAES bug caused by a semantic mismatch between the MLIR operation `arith.shrui` and its LLVM IR counterpart `lshr`. While LLVM IR explicitly treats shift amounts equal to the bit width in `lshr` as undefined behavior, the original `arith.shrui` specification defined the behavior for this edge case. This mismatch causes MLIR programs using `arith.shrui` to yield unexpected results, casting doubt on the correctness of its semantics in MLIR. The issue was resolved by aligning the semantics of `arith.shrui` with LLVM IR, explicitly defining this case as undefined behavior. Notably, this latent bug had persisted since the introduction of `arith.shrui` and went undetected by existing approaches because it caused behavioral inconsistencies rather than crashes.

**Erroneous Analysis Logic (EAL)** refers to flawed analysis logic within MLIR that misinterprets or provides inaccurate information (such as data or control flow details) to subsequent rewrite passes. Typically, an analysis serves multiple passes and other analyses; for instance, the `DataFlowSolver` analysis supplies data flow information used by integer range optimization, sparse conditional constant propagation, and dead code elimination. Errors in this analysis propagate incorrect information to all dependent passes, leading to erroneous MLIR program outputs and undermining trust in the MLIR compiler. One bug detected by DESIL falls under this category.

Figure 4(b) shows an MLIR program that triggers a bug in this category. The MLIR compiler generated different IRs for the given program across multiple runs under the same optimization due to the buggy data flow analysis in MLIR. Specifically, the “-int-range-optimizations” pass utilizes a `DataFlowSolver` after performing fold optimizations. However, when the fold optimization deletes an original operation and creates a new one at the same memory location, the solver fails to detect the change and returns the old operation's state. This bug can significantly impair compiler reliability as said by developers: “its impact on other passes (e.g., Sparse Conditional Constant Propagation and dead code analysis) makes debugging challenges”. Developers have fixed it by adding a listener to track deleted operations, preventing the solver from returning outdated states.

Table 3. Comparison between DESIL and lifted existing techniques in silent bug detection

| Techniques                    | #Inconsistencies | #FP Inconsistencies | FP Rate | #TP Inconsistencies | #Silent Bugs |
|-------------------------------|------------------|---------------------|---------|---------------------|--------------|
| MLIRSmith <sub>enhanced</sub> | 4,914            | 4,783               | 97.33%  | 131                 | 14           |
| MLIRod <sub>enhanced</sub>    | 4,542            | 4,404               | 96.96%  | 138                 | 15           |
| DESIL <sub>smith</sub>        | 519              | 0                   | 0%      | 519                 | 25           |
| DESIL <sub>od</sub>           | 470              | 0                   | 0%      | 470                 | 31           |

**Missing trait (MT)** refers to the absence of a required trait in the definition of an MLIR operation, attribute, or type. In MLIR, traits act as implicit markers that convey semantic properties (such as memory side effects) and guide compiler optimizations like dead code elimination. When a necessary trait is missing, optimizations relying on that trait may behave incorrectly, leading to faulty program transformations. This type of bug can have broad consequences, as the lack of a trait disables all optimizations or checks that depend on it, reducing the reliability of the compiler. One bug detected by DESIL falls into this category.

For example, issue#112881 is an MT bug caused by the omission of the `RecursiveMemoryEffects` trait in `linalg` operations (shown in Figure 4(c)). When the MLIR program is optimized using the “-linalg-named-op-conversion” pass, it produces incorrect results. Specifically, a `memref.store` operation inside a `linalg.map` stores the value %2 into `%alloc[%c1]`, which is later printed using `vector.print`. The expected output is 10 (the value of %2), but the actual printed value is 1 (the value before the `memref.store` operation) due to this bug. The root cause is the missing of `RecursiveMemoryEffects` trait on `linalg.map`, which prevents the compiler from recognizing the memory effects of `memref.store` within the operation. Consequently, since the return value of `linalg.map` is unused, the compiler wrongly assumes it has no side effects and eliminates it. This incorrect elimination of operations with memory side effects leads to unreliable program outputs. To fix this, developers added the `RecursiveMemoryEffects` trait to the relevant `linalg` operations, ensuring internal memory effects are properly detected and preserved during optimization.

### 4.3 RQ2: Compared to (Lifted) Existing MLIR Testing Techniques

As presented in Section 4.1.1, we lifted both MLIRSmith and MLIRod as MLIRSmith<sub>enhanced</sub> and MLIRod<sub>enhanced</sub>, enabling the comparison between DESIL and the existing MLIR testing techniques in silent bug detection. Table 3 presents the comparison results among these studied techniques in silent MLIR bug detection during the given testing time. In this table, columns 2-6 represent the number of inconsistencies detected via differential testing, the number of false positives among these inconsistencies, the ratio of false positives (dividing the number of false positives by the total number of inconsistencies), the number of true inconsistencies caused by silent bugs, the number of silent bugs after de-duplicating inconsistencies, respectively. For each detected inconsistency by MLIRSmith<sub>enhanced</sub> or MLIRod<sub>enhanced</sub>, we applied DESIL to check whether the corresponding MLIR program has UB and then eliminate it. If the inconsistency still exists by running the UB-free program, it is regarded as a true inconsistency; Otherwise, it is a false positive. Indeed, through our manual analysis on these true inconsistencies detected by existing techniques and the inconsistencies detected by DESIL, all of them are real bugs.

From Table 3, DESIL<sub>smith</sub> (25) and DESIL<sub>od</sub> (31) detected more silent bugs than MLIRSmith<sub>enhanced</sub> (14) or MLIRod<sub>enhanced</sub> (15), respectively, demonstrating the effectiveness of DESIL in detecting silent MLIR bugs. Although MLIRSmith<sub>enhanced</sub> and MLIRod<sub>enhanced</sub> were able to detect some silent bugs, they suffered from extremely high false positive rates (97.33% and 96.96%, respectively). Moreover, DESIL played a crucial role in distinguishing the silent bugs detected by MLIRSmith<sub>enhanced</sub> and



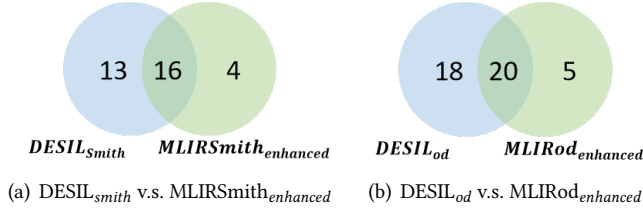


Fig. 5. The number of unique bugs detected by DESIL and lifted existing techniques in bug detection.

$MLIRod_{enhanced}$  from a large number of false positive inconsistencies. This indicates that even with enhancements,  $MLIRSmith$  and  $MLIRod$  remain impractical for reliably detecting silent bugs.

Besides silent bugs, all these studied techniques are able to detect crash bugs. Hence, we further compared them in the overall bug detection capability. In total,  $DESIL_{smith}$  and  $DESIL_{od}$  detected 29 and 38 MLIR bugs respectively, while  $MLIRSmith_{enhanced}$  and  $MLIRod_{enhanced}$  detected 20 and 25 bugs respectively, demonstrating the superiority of DESIL over baselines in terms of overall bug detection effectiveness. To investigate whether DESIL significantly outperforms baselines in statistics, we then conducted Mann-Whitney U-tests [7] between DESIL and each baseline in terms of the number of detected bugs across multiple runs at the significant level of 0.05. The p-value for  $DESIL_{smith}$  v.s.  $MLIRSmith_{enhanced}$  is 0.0057 (smaller than 0.05), and that for  $DESIL_{od}$  v.s.  $MLIRod_{enhanced}$  is 0.0055 (smaller than 0.05). The results demonstrate that the improvements of DESIL over both baselines are statistically significant.

Figure 5 shows the overlap analysis results for  $DESIL_{smith}$  v.s.  $MLIRSmith_{enhanced}$  and  $DESIL_{od}$  v.s.  $MLIRod_{enhanced}$ . From this figure, by comparing  $DESIL_{smith}$  with  $MLIRSmith_{enhanced}$ , the former detected 13 unique bugs (including 12 silent bugs and one crash bug) while the latter detected only 4 unique bugs (including one silent bug and 3 crash bugs). Similarly, by comparing  $DESIL_{od}$  with  $MLIRod_{enhanced}$ , the former detected 18 unique bugs (including 16 silent bugs and 2 crash bugs) while the latter detected only 5 unique bugs (including 0 silent bug and 5 crash bugs). The results further confirm the effectiveness of DESIL. Through further observation, we found that  $DESIL_{smith}$  (4) and  $DESIL_{od}$  (7) detected slightly less crash bugs than  $MLIRSmith_{enhanced}$  (6) and  $MLIRod_{enhanced}$  (10), respectively. This is as expected, since DESIL requires extra time cost for UB elimination, and thus ran less MLIR programs for testing. Specifically, during the same testing period, the number of executed programs for  $DESIL_{smith}$  and  $DESIL_{od}$  is 13,269 and 13,541 respectively, while that for  $MLIRSmith_{enhanced}$  and  $MLIRod_{enhanced}$  is 21,681 and 21,420 respectively. Nevertheless, the strong capability of DESIL in detecting silent bugs far outweighs its slight drawback in crash bug detection, which is due to the additional time cost incurred by UB elimination.

#### 4.4 RQ3: Ablation Study

We first investigated the contribution of the lowering path optimization mechanism in DESIL by running  $DESIL_{smith}^{w/o \text{ lower}}$  and  $DESIL_{od}^{w/o \text{ lower}}$ . Over five repeated 12-hour fuzzing sessions, both  $DESIL_{smith}^{w/o \text{ lower}}$  and  $DESIL_{od}^{w/o \text{ lower}}$  failed to compile any MLIR program into an executable form, even when applying 50 lowering passes – more than twice the number typically required by DESIL (i.e., 21 on average). That is, randomly selecting a sequence of lowering passes hardly succeeds in converting all operations from diverse dialects into an executable dialect within a reasonable number of steps. This underscores the critical role of our lowering path optimization strategy in ensuring the feasibility of DESIL for silent bug detection.

We then investigated the contribution of the operation-aware optimization recommendation strategy in DESIL by comparing with  $DESIL_{smith}^{w/o \text{ opt}}$  and  $DESIL_{od}^{w/o \text{ opt}}$ . During the given testing period,  $DESIL_{smith}$  and  $DESIL_{od}$  detected 29 and 38 bugs while  $DESIL_{smith}^{w/o \text{ opt}}$  and  $DESIL_{od}^{w/o \text{ opt}}$  detected 21

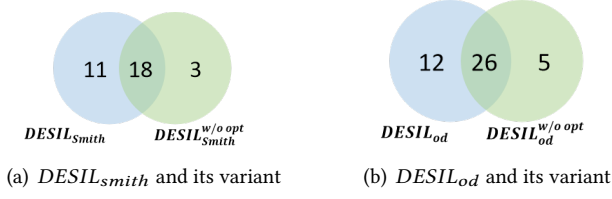


Fig. 6. The number of unique bugs with and without optimization recommendation component.

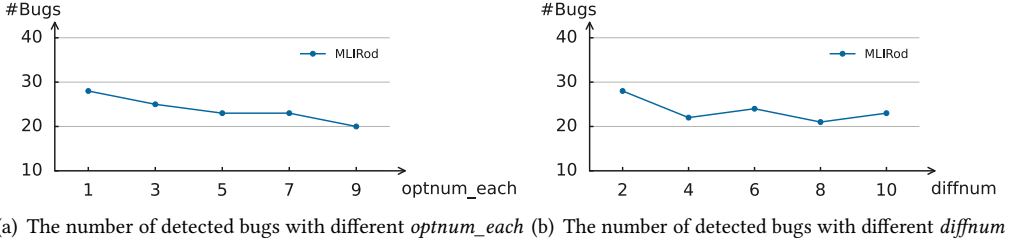


Fig. 7. The number of bugs detected by DESIL under different configurations.

and 31 bugs respectively. Figure 6 further shows the overlap analysis results among the studied techniques. As shown in the figure,  $DESIL_{smith}$  detected 11 unique bugs compared to  $DESIL_{smith}^{w/o opt}$ , and  $DESIL_{od}$  detected 12 unique bugs compared to  $DESIL_{od}^{w/o opt}$ . In contrast,  $DESIL_{smith}^{w/o opt}$  and  $DESIL_{od}^{w/o opt}$  detected only 3 and 5 unique bugs, respectively. These results highlight the superiority of the operation-aware optimization recommendation strategy in DESIL over the random strategy for specifying optimization passes.

#### 4.5 RQ4: Influence of Different Configurations

Figure 7 shows the effectiveness of DESIL under different configurations of  $optnum\_each$  (the number of optimization passes per lowering step) and  $diffnum$  (the number of compilations for differential testing), respectively. The y-axis represents the total number of bugs detected by  $DESIL_{smith}$  and  $DESIL_{od}$ . In general, as the values of  $optnum\_each$  or  $diffnum$  increase, the effectiveness of DESIL decreases to some extent (especially for the former). This phenomenon arises from the trade-off between bug detection capability and time efficiency. While larger values enable broader exploration of the optimization space, potentially increasing the likelihood of uncovering bugs, the associated time overhead ultimately reduces these benefits. This aligns with the conclusion of an existing study [8], which highlights testing efficiency as one of the most critical factors in compiler testing. Similarly, the decreasing trend for  $diffnum$  is less pronounced, as the overhead it incurs under these settings is lower than that of  $optnum\_each$ . Based on this insight, we set  $optnum\_each$  to 1 and  $diffnum$  to 2 as the default configurations of DESIL for practical use.

### 5 Discussion

**Significance of DESIL.** While DESIL is designed to fuzz the MLIR compiler infrastructure, its impact extends beyond a single system. Many compilers, such as Flang [3] and IREE [4], are built on top of MLIR, meaning that improving the reliability of the MLIR compiler infrastructure enhances the quality and robustness of all compilers that depend on it. In other words, fuzzing the MLIR compiler infrastructure has a far-reaching effect, benefiting multiple compiler systems rather than just one. Moreover, DESIL is independent of MLIR test program generation techniques and can

be integrated with any of them. Specifically, for any given MLIR program, DESIL can transform it into a UB-free version and then compile it into an executable form for silent bug detection. Our evaluation has demonstrated the effectiveness of DESIL when combined with two state-of-the-art MLIR program generation tools, i.e., MLIRSmith and MLIRod. Therefore, we are confident that DESIL can also enhance future, more advanced MLIR program generation techniques, further extending its impact and applicability.

**Limitations of DESIL.** The first limitation is that the dialects, operations, and execution platforms supported by DESIL are incomplete. Currently, DESIL supports most of the widely used dialects and operations in MLIR fuzzing, specifically the dialects and operations supported by MLIRSmith. However, the dialects supported by MLIRSmith are primarily middle-level dialects, meaning that certain higher-level dialects, such as the TOSA dialect, are not yet supported. Fortunately, DESIL has already defined a comprehensive set of undefined behavior elimination rules, which can be reused to handle undefined behaviors in the TOSA dialect. As a result, supporting new dialects in DESIL generally requires only a minimal number of additional elimination rules. To expand the range of supported execution platforms, new executable dialects should be supported, which can be accomplished by adjusting operation-specific lowering paths within DESIL. No additional mechanisms are required to enable the support of new executable dialects.

Second, DESIL has false negatives due to two main factors. One factor is the side effect of UB-elimination rules. Some rules may be overly conservative, restricting the diversity of generated MLIR programs and potentially reducing DESIL's bug detection capability (resulting in missed bugs). This limitation can be mitigated by refining the UB-elimination rules with greater attention to preserving program diversity. Another factor lies in the limitation of the checksum mechanism. Currently, DESIL includes only integer variables in the checksum, excluding floating-point variables. This may cause bugs related to floating-point computations to go undetected. To address this, we plan to extend DESIL with existing solutions for handling floating-point imprecision [13, 14]. Moreover, subtle semantic discrepancies may not be captured by numerical checksum comparisons alone. In such cases, integrating formal verification tools like Alive2 [16] may help strengthen bug detection further. Nevertheless, DESIL's ability to expose previously undetectable silent bugs outweighs the limitation of false negatives.

The third limitation arises from MLIR evolution. Dialects and operations in MLIR may be changed or newly introduced over time, indicating that the corresponding UB-elimination rules may need to be updated or designed. Maintaining these rules poses a challenge for DESIL. To relieve this, we plan to leverage the strong semantic understanding of LLMs to analyze code and documentation, facilitating (semi-)automatic evolution of existing rules and synthesis of new ones.

**Generalizability of DESIL.** The core principles of DESIL (UB elimination, lowering-path optimization, and differential testing) are broadly applicable to other IRs. Some UB-elimination rules (e.g., signed integer overflow elimination) are common and easily transferable. As most IRs have simpler lowering pipelines than MLIR's multi-dialect design, the lowering-path optimization could be simplified accordingly. These indicate the generalizability of DESIL to a large extent. Nevertheless, full adaptation still requires handling domain-specific UBs and transformations.

**Overhead of DESIL.** Integrating DESIL into bug detection incurs an average overhead of 5.045 seconds per MLIR program. Specifically, programs not processed by DESIL take 8.123 seconds on average, while those processed by DESIL require 13.168 seconds. This increase is largely due to longer compilation times caused by UB-elimination rules, which expand MLIR program size. Specifically, compiling unprocessed programs takes 7.630 seconds on average, compared to 12.637 seconds for processed ones (an added cost of 5.007 seconds). Other components of DESIL contribute negligible overhead, adding less than 0.1 seconds per program. While the UB-elimination rules

slightly reduce fuzzing efficiency, they enable DESIL to detect silent bugs in the MLIR compiler infrastructure that would otherwise go unnoticed, making this trade-off worthwhile.

**Threats to Validity.** The threat to *internal* validity primarily concerns the implementation of DESIL. To mitigate this, two authors meticulously reviewed the source code and designed unit tests to ensure the correctness of DESIL. In addition, we further validated DESIL by applying the “-generate-runtime-verification” pass, which is equipped in the MLIR compiler for verifying the correctness of operations, to 10,000 test programs generated by DESIL. None of the test programs triggered the check failure in the “-generate-runtime-verification” pass, demonstrating the stability of DESIL. For the existing techniques (i.e., MLIRSmith and MLIRod), we directly adopted their publicly released implementations and followed the recommended settings.

The threat to *external* validity arises from the choice of the subject under test. To address this, we selected the latest versions of the MLIR compiler infrastructure as the subject and conducted continuous fuzzing to thoroughly evaluate the effectiveness of DESIL in detecting previously unknown bugs. This is consistent with prior work [19, 21], targeting the MLIR infrastructure itself rather than individual compilers built on it. The reason is that MLIR is foundational, and thus bugs in its infrastructure could affect all dependent compilers. In the future, we will adapt DESIL to test specific MLIR-based compilers to expand the applicability.

The threat to *construct* validity arises from the randomness in the experiments and the hyper-parameter settings in DESIL. To mitigate the impact of randomness, we repeated each experiment five times (three times for parameter-setting experiments due to their extensive time cost). To address concerns regarding hyper-parameter settings, we evaluated DESIL under various configurations, as detailed in Section 4.5.

## 6 Related Work

In recent years, several testing techniques have been proposed for the MLIR compiler infrastructure [19, 21]. For example, Wang et al. introduced MLIRSmith [21], the first MLIR program generator designed for testing the MLIR compiler infrastructure. MLIRSmith generates MLIR programs by first constructing program templates based on MLIR’s grammar and then filling these templates according to semantic rules to ensure the generation of semantically valid MLIR programs. Suo et al. proposed MLIRod [19], which defines operation dependency coverage as the testing guidance and employs four types of dependency-specific mutations to generate new MLIR programs, enhancing the effectiveness of testing the MLIR compiler infrastructure. They have demonstrated significant effectiveness in detecting crash bugs by generating semantically valid MLIR programs.

As explained in Section 1, all existing techniques suffer from the UB issue, which prevents them from detecting silent bugs. Due to this issue, none of these techniques incorporates a lowering process to compile MLIR programs into executable programs for execution, making it impossible for them to detect silent bugs. In contrast, our work introduces DESIL, the first technique designed to detect silent MLIR bugs by addressing the UB issue through a set of UB-elimination rules and designing a lowering path optimization strategy for checking program execution. Thus, DESIL is orthogonal to all existing MLIR testing techniques, offering a significant improvement in quality assurance for the MLIR compiler infrastructure.

A recent parallel effort, Ratte [25], also targets silent bug detection in the MLIR compiler infrastructure. Below, we provide a detailed comparison between DESIL and Ratte: (1) Ratte employs a reference interpreter to provide runtime semantics for guiding test generation from scratch, requiring manual implementation of both the generator and interpreter for each operation. In contrast, DESIL eliminates UBs in existing MLIR programs using UB-elimination rules, which only need to be defined as necessary (i.e., without a strict one-to-one mapping between operations and

rules). (2) Ratte detects bugs by comparing interpreter results with those from a fixed lowering and optimization pipeline, while DESIL explores diverse lowering paths via a lowering-path optimization strategy, thereby covering a broader range of MLIR passes. (3) Regarding effectiveness, as reported in the respective papers, DESIL detected 17 confirmed or fixed silent bugs, whereas Ratte identified 5. Further, we evaluated Ratte by running it for 12 hours per mode on the same MLIR version used in our study, but did not observe any inconsistencies, confirming DESIL's effectiveness.

Additionally, there are a lot of testing techniques for traditional compilers in the literature [13, 14, 17, 24]. Some of these techniques are capable of detecting silent bugs in traditional compilers by ensuring UB-free test programs. For example, Csmith [24], a grammar-based C program generator, leverages predefined rules (such as safe arithmetic wrappers) and built-in dynamic checks to prevent undefined behaviors during test program generation. YARPGen [14], another C/C++ program generator, ensures expression safety by defining a set of safe expressions that prevent undefined behaviors at the generation phase. Lecoœur et al. introduced reconditioning [13], a technique that eliminates undefined behaviors in OpenGL Shading Language (GLSL) and WebGPU Shading Language (WGSL) through post-processing with program transformations.

Reconditioning is the most relevant technique to DESIL, as both address UB through post-processing code transformations rather than during program generation. However, DESIL differs in several key aspects. First, DESIL targets a fundamentally different domain, focusing on intermediate representations (IRs) rather than high-level languages. Reconditioning operates on languages with restricted memory allocation (e.g., GLSL, which disallows variable-length arrays), while DESIL handles IRs that support flexible memory allocation (e.g., dynamic shapes) and complex operation semantics (e.g., `linalg.matmul` for matrix multiplication). As a result, DESIL must address unique categories of UB and requires more sophisticated transformation rules. Second, DESIL introduces an additional challenge absent in reconditioning: lowering path optimization. MLIR compilation involves multiple dialects, requiring careful selection of lowering passes to ensure successful translation to an executable representation. DESIL tackles this problem with a structured lowering-path optimization strategy, making it fundamentally distinct from reconditioning.

## 7 Conclusion

In this paper, we presented DESIL, a novel technique designed to bridge the gap in detecting silent bugs in the MLIR compiler infrastructure. DESIL addresses two key challenges in MLIR bug detection: (1) eliminating undefined behavior (UB) in UB-prone operations through a set of undefined behavior elimination rules, and (2) determining an optimal lowering path to prevent redundant or circular application of lowering passes, ensuring efficient compilation to an executable representation. By incorporating a differential testing oracle, DESIL further enhances its ability to detect silent bugs by comparing the results of executable programs affected by different optimization passes. Our evaluation demonstrates DESIL's effectiveness in detecting silent MLIR bugs, identifying 42 previously unknown bugs (23 silent and 19 crash bugs) over a four-month testing period, with 26 fixed and 33 confirmed by the developers.

## Data-Availability Statement

We released the source code of DESIL (implemented in 4.6K lines of C++ code), along with all experimental data in our artifact [20].

## Acknowledgments

This work was supported by the National Key Research and Development Program of China (Grant No. 2024YFB4506300), and the National Natural Science Foundation of China (Grant Nos. 62322208, 12411530122, 62232001, 62202324).



## References

- [1] 2025. Affine Documentation. <https://mlir.llvm.org/docs/Dialects/Affine>.
- [2] 2025. DESIL repository. <https://github.com/tju-chenyaosuo/DESIL>.
- [3] 2025. Flang. <https://github.com/llvm/llvm-project/tree/main/flang>.
- [4] 2025. IREE. <https://github.com/iree-org/iree>.
- [5] 2025. MLIR language reference. <https://mlir.llvm.org/docs/LangRef>.
- [6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. <https://www.worldcat.org/oclc/12285707>
- [7] Andrea Arcuri and Lionel C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 1–10. doi:10.1145/1985793.1985795
- [8] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 180–190. doi:10.1145/2884781.2884878
- [9] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2021. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2021), 4:1–4:36. doi:10.1145/3363562
- [10] Alastair F. Donaldson, Paul Thomson, Vasyli Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1017–1032. doi:10.1145/3453483.3454092
- [11] Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient Computation of Interprocedural Definition-Use Chains. *ACM Trans. Program. Lang. Syst.* 16, 2 (1994), 175–204. doi:10.1145/174662.174663
- [12] 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. doi:10.1109/CGO51591.2021.9370308
- [13] Bastien Lecoq, Hasan Mohsin, and Alastair F. Donaldson. 2023. Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1801–1825. doi:10.1145/3591294
- [14] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. doi:10.1145/3428264
- [15] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1826–1847. doi:10.1145/3591295
- [16] Nuno P. Lopes, Jueyoungh Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 65–79. doi:10.1145/3453483.3454030
- [17] 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, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1483–1486. doi:10.1145/3597926.3604919
- [18] Qingchao Shen, Yongqiang Tian, Haoyang Ma, Junjie Chen, Lili Huang, Ruifeng Fu, Shing-Chi Cheung, and Zan Wang. 2025. A Tale of Two DL Cities: When Library Tests Meet Compiler. (2025), 2201–2212. doi:10.1109/ICSE55347.2025.00025
- [19] 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 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 1287–1299. doi:10.1145/3650212.3680360
- [20] Chenyao Suo, Jianrong Wang, Yongjia Wang, Jiajun Jiang, Qingchao Shen, and Junjie Chen. 2025. DESIL-artifact. Zenodo. doi:10.5281/zenodo.15727517
- [21] 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 *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1555–1566. doi:10.1109/ASE56229.2023.00120
- [22] Tao Xie. 2006. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4067)*, Dave Thomas (Ed.). Springer, 380–403. doi:10.1007/11785477\_23



- [23] Chen Yang, Junjie Chen, Xingyu Fan, Jiajun Jiang, and Jun Sun. 2023. Silent Compiler Bug De-duplication via Three-Dimensional Analysis. 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, 677–689. doi:10.1145/3597926.3598087
- [24] 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, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. doi:10.1145/1993498.1993532
- [25] Pingshi Yu, Nicolas Wu, and Alastair F. Donaldson. 2025. Ratte: Fuzzing for Miscompilations in Multi-Level Compilers Using Composable Semantics. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach (Eds.). ACM, 966–981. doi:10.1145/3676641.3716270

Received 2025-03-26; accepted 2025-08-12