

# Fuzzing MLIR by Synthesizing Custom Mutations

Ben Limpanukorn

University of California, Los Angeles  
blimpan@cs.ucla.edu

Jiyuan Wang

University of California, Los Angeles  
wangjiyuan@cs.ucla.edu

Hong Jin Kang

University of California, Los Angeles  
hjkang@cs.ucla.edu

Eric Zitong Zhou

University of California, Los Angeles  
zitongzhou@cs.ucla.edu

Miryung Kim

University of California, Los Angeles  
miryung@cs.ucla.edu

**Abstract**—A growing trend in compiler design is to enable modular extensions to intermediate representations (IRs). Multi-Level Intermediate Representation (MLIR) is a new effort to enable faster compiler development by providing an extensible framework for downstream developers to define custom IRs with MLIR dialects. Sets of MLIR dialects define new IRs that are tailored for specific domains. The diversity and rapid evolution of these IRs make it impractical to pre-define custom test generator logic for every available dialect.

We design a new approach called SYNTHFUZZ that automatically infers and applies custom mutations from existing tests. The key essence of SYNTHFUZZ is that inferred custom mutations are parameterized and context-dependent such that they can be concretized differently depending on the target context. By doing this, we obviate the need to manually write custom mutations for newly introduced MLIR dialects. Further, SYNTHFUZZ increases the chance of finding effective edit locations and reduces the chance of inserting invalid edit content by performing  $k$ -ancestor-prefix and  $l$ -sibling-postfix matching.

We compare SYNTHFUZZ to three baselines: Grammarinator—a grammar-based fuzzer without custom mutators, MLIRSmith—a custom test generator for MLIR, and NeuRI—a custom test generator with support for parameterized generation. We conduct this comprehensive comparison on 4 different MLIR projects where each project defines a new set of MLIR dialects that would take months of effort to manually write custom input generation and mutation logic. Our evaluation shows that SYNTHFUZZ on average improves input diversity by  $1.51\times$ , which increases branch coverage by  $1.16\times$ . Further, we show that our context dependent custom mutation increases the proportion of valid tests by up to  $1.11\times$ , indicating that SYNTHFUZZ correctly concretizes its parameterized mutations with respect to the target context. Parameterization of the mutations reduces the fraction of tests violating general MLIR constraints by  $0.57\times$ , increasing the time spent fuzzing dialect-specific code.

**Index Terms**—MLIR, grammar-based fuzzing, code patterns, program transformation

## I. INTRODUCTION

A common compiler design consists of three components—a frontend, an optimizer, and a backend [1] where the frontend translates source code to an intermediate representation (IR), the optimizer performs optimizations on the IR, and the backend translates the IR into instructions specific to the target architecture. The LLVM project lowered the barrier to entry by providing a common LLVM intermediate representation with different architecture backends.

There is a growing trend to advance modular compiler design through extensible infrastructure for intermediate representations (IR). Multi-Level Intermediate Representation (MLIR) is one such effort. It aims to create a unified infrastructure for manipulating an extensible IR format, enabling reusable compiler components. Unlike LLVM, which defines a single common intermediate representation (IR), MLIR enables developers to extend the underlying IR through the concept of MLIR dialects. Each MLIR dialect essentially defines a new IR consisting of a unique set of operations, types, and attributes with domain-specific semantics. For example, the IR for machine learning is modeled as computation graphs, while the IR for LLVM is modeled as a sequence of program instructions.

Take the Circuit IR Compilers and Tools (CIRCT) [2] project as an example. It leverages the MLIR framework to build a compiler for heterogeneous compilation by defining 26 new dialects with 145 new operations. For example, CIRCT uses a generic hardware abstraction by defining the `hw` and `comb` custom dialects with operations to represent abstract hardware modules and combinational logic. Listing 1 shows a snippet of MLIR representing a hardware module containing a custom MLIR operation called `comb.add` which represents combinational addition.

The fast evolution of the underlying IR presents challenges for developing custom test generators. Google reported in 2020 that over 60 dialects have been defined internally in MLIR [3]. In the four years since the MLIR project’s initial public release, 29 downstream projects like CIRCT have each contributed one or more custom dialects [2], [4]. General-purpose fuzzers such as AFL++ [5] fail to effectively generate or mutate MLIR due to its highly structured form. For instance, syntactically correct MLIR must have proper nesting of operations, the correct number of operands and outputs for each operation, valid type annotations, and valid attribute names and values. Grammar-based fuzzers, such as Grammarinator, use a context-free grammar to constrain input generation, but such grammars cannot encode context-specific constraints [6]. Custom generator-based fuzzers in the vein of CSmith [7], NNSmith [8], and MLIRSmith [9] manually encode constraints in terms of imperative code. Such constraints require significant effort to write and struggle to

```

1 "hw.module"() ({ // Hardware module definition
2 ^bb0(%a1: i2): 2-bit integer input
3 // Defines a new constant with value -2, bitwidth 2:
4 %c1 = "hw.constant"() {value = -2 : i2} : () -> i2
5 // Adds the constants %0 and %1, outputting %2:
6 %o1 = "comb.add"(%a1, %c1) <{twoState}> : (i2, i2) -> i2
7 "hw.output"(%o1) : (i2) -> () // Output of module is %2:
8 }) // extra boilerplate omitted

```

Listing 1. This MLIR code snippet uses a new hardware dialect `comb` for combinational logic in the CIRCT project.

adapt to the fast continual introduction of new dialects.

We observe that the fast-evolving compiler infrastructure of IRs requires a test input generator that can encode or learn constraints automatically. To this end, we propose a new approach called SYNTHFUZZ, drawing inspiration from techniques for automated patch synthesis [10], [11], [12]. These techniques synthesize code edits from examples, learning the code contexts in which the transformations are appropriate and concretizing the code edits to the matching code contexts.

Like generator-based fuzzers, SYNTHFUZZ is capable of preserving context-sensitive constraints such as the cardinality of operation arguments and return values, the def-use relationships of values, and the consistency of type annotations. The key novelty is that SYNTHFUZZ can do so without the significant manual effort required to write custom generators by hand. SYNTHFUZZ accomplishes this by synthesizing parameterized mutations from seed test cases.

Colored pairs in Listing 1 represent the def-use relationships and type consistency that needs to be satisfied. A parameterized mutation derived from Listing 1 would encode the knowledge that the operation `comb.add` is nested within the `hw.module` denoted as ancestor  $k_1$ , is preceded by one `hw.constant` operation denoted as  $l_1$ , and followed by one `hw.output` operation denoted as  $r_1$ . This knowledge enables SYNTHFUZZ to select an appropriate context to apply the mutation by matching the  $k$ -ancestors and  $l$ -siblings (and  $r$ -siblings) of the `comb.add` operation. The parameterized mutation also encodes the knowledge that `comb.add` takes two arguments `%a1` and `%c1`, and returns one value `%o1` all of which have the same type `i2`. SYNTHFUZZ parameterizes these arguments and types and then re-concretizes them based on the target context to which the mutation is applied.

We compare the effectiveness of SYNTHFUZZ against Grammarinator, MLIRSmith, and NeuRI. Grammarinator was chosen as a representative grammar-based fuzzer. MLIRSmith was chosen as a representative generator-based fuzzer. NeuRI was chosen as a custom test generator with limited parameterization—i.e. it parameterizes the tensor shapes and operation numerical attributes. We evaluate SYNTHFUZZ on four MLIR-based compiler projects: LLVM, ONNX-MLIR, Triton, and CIRCT. These are chosen as representative MLIR projects that define 42, 2, 4, and 26 custom dialects respectively. For all dialects except the 13 targeted by MLIRSmith and the 1 `onnx` dialect that can be targetted by NeuRI, no custom test generators exist. Writing test generators for these custom dialects is time-consuming. As an example, MLIRSmith’s implementation totals 11,434 lines of code with

447 lines of code per dialect on average [13].

We assess SYNTHFUZZ’s fault detection potential by measuring code coverage and input diversity. We measure input diversity in terms of MLIR dialect pair coverage. Dialect pair coverage [9] is defined as the number of unique pairs of operations/dialects that have a data dependency or control dependency. Averaged across over four MLIR compiler projects, SYNTHFUZZ outperforms Grammarinator, MLIRSmith and NeuRI in terms of branch coverage by  $1.22\times$ ,  $29.78\times$  and  $17.47\times$  respectively. In terms of dialect pair coverage, SYNTHFUZZ outperforms Grammarinator, MLIRSmith and NeuRI on average by  $1.75\times$ ,  $4.60\times$ , and  $5.56\times$ . Compared to MLIRSmith and NeuRI, SYNTHFUZZ is capable of covering 60 new custom dialects defined by the four MLIR projects. SYNTHFUZZ discovers a previously undiscovered bug in CIRCT.

In summary, this paper makes the following contributions:

- 1) We design a novel compiler fuzzing technique that obviates the need for defining custom mutations apriori, which is impractical when the target IR is highly extensible and constantly evolving.
- 2) Our method automatically synthesizes and applies multi-edit, dependence-aware, custom mutations on the fly. The key enabler is the construction of parameterized mutations from test examples, and the concretization of the mutations after establishing the context through ancestor path or prefix(postfix) matching.
- 3) We show that our method achieves  $1.16\times$  greater code coverage and  $1.51\times$  greater dialect coverage within the same time budget compared to existing baseline fuzzers.

The remainder of this paper is organized as follows. Section II introduces MLIR and a motivating example. Section III presents the design and implementation of SYNTHFUZZ. Section IV provides the design of our experiments and their results. Section V discusses possible threats to validity. Section VI presents related work. Finally, we draw the conclusions of our work in Section VII.

## II. BACKGROUND

### A. MLIR: Multi-Level Intermediate Representation

Multi-Level Intermediate Representation (MLIR) is a modular compiler framework that differs from traditional approaches by enabling developers to extend the intermediate representation. Rather than defining a single monolithic IR with a fixed set of types and instructions like LLVM’s IR, MLIR is extensible by design. Compiler developers may define new MLIR dialects consisting of custom operations and types tailored to the domain, language, or architecture the compiler targets. MLIR dialects can be progressively lowered, forming a modular compilation pipeline, in contrast with traditional compiler infrastructure that offers limited extensibility. However, this presents a challenge for test generation, since custom operations introduce semantic constraints that are operation-specific, such as the def-use relationships and type consistency illustrated by colored pairs in Listing 1.

```

1 "hw.module"() { {
2 ^bb0(%arg0: i2):
3   %c1 = "hw.constant"() {value = -2 : i2} : () -> i2
4   %o1 = "comb.add"(%arg0, %c1) : (i2, i2) -> i2
5   "hw.output"(%o1) : (i2) -> ()
6 }}

```

Listing 2. A donor program  $P_d$  from which a mutation for inserting the `comb.add` operation is synthesized from.

```

1 "hw.module"() { {
2 ^bb0(%arg0: i4, %arg1: !hw.array<2xi2>):
3   %0 = "hw.bitcast"(%arg1) : (!hw.array<2xi2>) -> i4
4   %i1 = "comb.sub"(%arg0, %0) : (i4, i4) -> i4
5   "hw.output"(%i1) : (i4) -> ()
6 }}

```

Listing 3. A recipient program  $P_r$  to which the mutation for inserting the `comb.add` operation from Listing 2 should be applied to.

```

1 "hw.module"() { {
2 ^bb0(%arg0: i4, %arg1: !hw.array<2xi2>):
3   %0 = "hw.bitcast"(%arg1) : (!hw.array<2xi2>) -> i4
4   %i1 = "comb.sub"(%arg0, %0) : (i4, i4) -> i4
5   %o1 = "comb.add"(%arg0, %c1) : (i2, i2) -> i2
6   "hw.output"(%i1) : (i4) -> ()
7 }}

```

Listing 4. A test case created by Grammarinator[6]’s recombine operation. It deletes line4 and adds line 5. This test case is invalid as it violates the **def-use** relation and the **type** consistency.

```

1 "hw.module"() { {
2 ^bb0(%arg0: i4, %arg1: !hw.array<2xi2>):
3   %0 = "hw.bitcast"(%arg1) : (!hw.array<2xi2>) -> i4
4   %i1 = "comb.sub"(%arg0, %0) : (i4, i4) -> i4
5   %o1 = "comb.add"(%arg0, %0) : (i4, i4) -> i4
6   "hw.output"(%i1) : (i4) -> ()
7 }}

```

Listing 5. A test case created by SYNTHFUZZ’s context-dependent, parameterized mutation. This mutation replaces an operation `comb.sub` with `comb.add`. The test case is valid as SYNTHFUZZ matched a corresponding context before concretizing its mutation to the target context.

Take the Circuit IR Compilers and Tools (CIRCT) [2] as an example. CIRCT is a unified framework built on MLIR that enables optimized hardware design across different backends catering to the needs of heterogeneous compilation. It defines 26 new dialects with 145 new operations, including the `comb` and `hw` dialects that define low-level hardware operations. An example of the `comb.add` operation is shown on line 4 of Listing 3. The full name of this operation is `comb.add` where `comb` is a dialect name and `add` is the operation name. As shown in the snippet, the operation takes two operands `%arg0` and `%c1` and returns a single value `%o1`. Its type signature indicates that the operation takes an input operands of type `i2` (2-bit integers) and produce an output of type `i2`.

## B. Motivating Example

Existing mutation strategies such as recombining test fragments frequently fail to generate test cases capable of exercising deeper compiler logic. This failure is caused by the large proportion of *invalid* test cases generated, which violate early checks made by the compiler. For example, a) the definition of identifiers needs to exist before they are used (*def-use*), b) the types of variables need to remain consistent through the test case (*type consistency*), and c) the number and type of arguments that match what is required by an operation

(*signature consistency*). To address the limitation of existing fuzzers, we present an approach that synthesizes parameterized mutations, aiming to implicitly capture these constraints.

Consider the following seed test cases: a “donor” program  $P_d$  in Listing 2 that contains the `comb.add` operation to be inserted in the “recipient” program  $P_r$  in Listing 3. We demonstrate how grammar-based generation and recombination are unlikely to produce a valid program shown in Listing 5.

A grammar-based mutator following a general MLIR grammar is unlikely to produce Listing 5 because the grammar does not include operation semantics defined by MLIR dialects. For any given operation, the generic MLIR grammar only specifies that the syntax of an operation must have a name (e.g. `comb.sub`), 0 or more return values, arguments and attributes, and a type signature. Therefore, a grammar-based fuzzer generating a `comb.add` operation would be unaware of the signature of the operation as defined by the `comb` dialect. Specifically, without a grammar for the dialect, the fuzzer would be unaware of the associations between variables and their types, e.g. `%arg0`, `%0` has the type `i4`, and operation-specific signature such as `comb.add` having exactly two input values and one return value.

Another common grammar-based mutation strategy is to recombine the *fragments* of existing tests with other test cases. To illustrate, `comb.sub` at line 4 of the recipient program  $P_r$  in Listing 3 is replaced with `comb.add` from line 4 of the donor program  $P_d$  in Listing 2, producing the mutated test in Listing 4. However, after the replacement, the values and types of the `comb.add` operation are inconsistent with its new surrounding context. The resulting test will be rejected early by the compiler as it violates the *def-use* constraint, since the value `%c1` was not defined before it was used. Inferring such semantic constraints (often Turing-complete) is challenging as discussed in previous works [14] [15] [16].

Listing 5 highlights the changes required to adapt the code using `comb.add` from program  $P_d$  to the context of program  $P_r$ . To satisfy the *def-use* constraint, the values referenced as arguments to an operation (e.g. `%0` on line 5) must be previously defined (e.g. `%0` on line 3). To satisfy *type consistency*, the initially assigned types, such as `i4`, for `%0` on line 3 must remain consistent in its subsequent references, such as its use as an argument and the resulting return type on line 5.

To generate test cases that satisfy these constraints, one can write custom generators or refine new grammars to encode these constraints. However, this requires hand-coding the constraints of each operation defined in a dialect, and the cost is exacerbated for rapidly evolving projects—those that use MLIR—because the specialized grammar or custom generators will need to be updated as new dialect operations are added, modified, or removed.

In this paper, we propose to *automatically synthesize custom mutators*. Existing test cases of MLIR dialects demonstrate how the dialect-specific operations should be invoked. Our key insight is that *these test cases implicitly encode the various constraints of MLIR dialects, e.g., def-use, operations’ type*

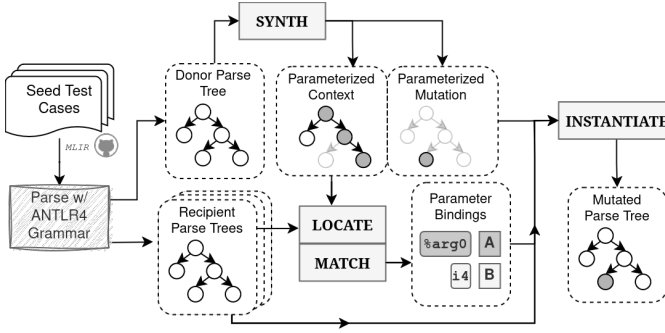


Fig. 1. A flowchart of SYNTHFUZZ's fuzzing loop

*signature*, and therefore, matching the code context would lead to a higher chance of successfully generating valid test cases. For example, the donor test case in Listing 2 and the recipient test case in Listing 3 exhibit a structural similarity. SYNTHFUZZ is able to identify such similarity and synthesize a parameterized mutation that captures the context of the operation `comb.add` in the donor test case. Applying the mutation, SYNTHFUZZ produces a valid test case as shown in Listing 5, which respects the dialect-specific constraints, and hence is able to exercise deeper logic in the MLIR compiler.

### III. APPROACH

Figure 1 describes SYNTHFUZZ's fuzzing loop. Each iteration synthesizes a custom mutation from a donor test case and transplants it onto a recipient test case. In the SYNTH step (Section III-A), SYNTHFUZZ selects a donor test case and a recipient test case. From the donor test case, SYNTHFUZZ infers a *parameterized mutation* with a *parameterized context*. In the LOCATE step (Section III-B), as the mutations are context-dependent, SYNTHFUZZ matches the parameterized context against the recipient test case to identify a suitable location for the mutation. The MATCH step (Section III-C) then creates a variable binding of parameters to concrete program fragments from the matching context. Finally, INSTANTIATE (Section III-D) concretizes the mutation and transplants it into the recipient test case.

#### A. SYNTH: Synthesizing a parameterized mutation.

Parameterized mutations are synthesized from seed test case which we will refer to as *donor test cases*. Given a donor test case like Listing 6, SYNTHFUZZ parses the test case using a generic MLIR grammar to produce a *donor parse tree*. From the donor parse tree, SYNTHFUZZ constructs custom mutator comprising of an *parameterized mutation* and a *parameterized context* as shown in Figure 2.

The parameterized mutation is a parameterized partial parse-tree that contains the content to be inserted by SYNTHFUZZ when mutating a test case. The parameterized context is similarly a partial parse tree that captures the conditions in which parameterized mutation may be instantiated.

SYNTH uniformly randomly selects a sub-tree in the donor's parse tree as the parameterized mutation to be transplanted. The parameterized context and parameterized muta-

```

1 "hw.module"() { {
2   ^bb0(%arg0: i2):
3     %c1 = "hw.constant"() {value = -2 : i2} : () -> i2
4     %o1 = "comb.add"(%arg0, %c1) : (i2, i2) -> i2
5     "hw.output"(%o1) : (i2) -> ()
6 })

```

Listing 6. In this donor test case  $P_d$ , the boxed area represents the source of a grafted, parameterized mutation. The rest unboxed area represents the potential source of a corresponding, parameterized context.

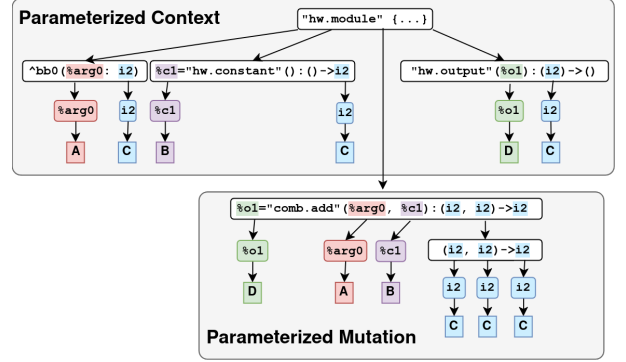


Fig. 2. This diagram illustrates how SYNTH decomposes the donor test case  $P_d$  shown in Listing 6 into a parameterized context and parameterized mutation. For example, concrete symbols such as `%arg0`, `%c1`, `i2`, and `%o1` are now parameterized as placeholders such as `A`, `B`, `C`, and `D` respectively.

tion are extracted by bisecting the donor parse tree along at the selected sub-tree's root node as shown in Figure 2. The parameterized context encodes information such as the operations before and after the parameterized mutation, their ordering, the nesting of operations, and the potential locations of parameters. For example, as shown in Figure 2, the parameterized context encodes the information that `comb.add` is preceded by a block label `bb0` and the operation `hw.constant`, and succeeded by a `hw.output` operation. It also encodes the nesting information that `bb0`, `hw.constant`, `comb.add`, and `hw.output` are nested within `hw.module`.

Since the donor's mutation may contain identifiers and types that are not defined in recipient programs, a naive transplantation will likely produce an invalid input that violates def-use and type consistency constraints. SYNTH uses a heuristic that common sub-strings in the input can be indicators of context-dependent constraints such as def-use and type consistency. SYNTH parameterizes the context and mutation by introducing a parameter for each common sub-string between the context and the mutation. In Figure 2 SYNTH creates 4 parameters: `A`, `B`, `C`, and `D` with an initial binding of `%arg0`, `%c1`, `i2`, and `%o1` as illustrated with matching colors in Figure 2. Later, each parameter can be concretized with a suitable sub-string from the context of the recipient.

#### B. LOCATE: Selecting mutation location depending on the target context

SYNTHFUZZ looks for locations where the parameterized context matches the recipient input to increase the likelihood of satisfying the constraints. Three factors are considered:



the number of matching ancestor nodes  $k$ , left-sibling nodes  $l$ , and right-sibling nodes  $r$ . Matching  $k$  ancestors with the parameterized context ensures that the mutation is made in a similar level of nesting. This prevents situations such as improperly nested functions, or operations being inserted outside of functions. Matching  $l$  left-siblings and  $r$  right-siblings with the parameterized context ensures that if an mutation pattern is of an operation, then the operation will likely be inserted in a location where the required number of operand values (left) is available, and the results of the parameterized mutation will be used accordingly (right).

**Algorithm 1** LOCATE: Finding a valid mutate location by matching  $k$ -ancestors and  $l(r)$  siblings.

**Input:**

- $context \leftarrow$  the parametrized context
- $recipient \leftarrow$  the recipient parse tree

**Output:**

- $mutateLocation \leftarrow$  a parse-tree node in the recipient test case that represents a valid mutate location

```

1: for  $candidate \leftarrow walk(recipient)$  do
2:    $isMatch \leftarrow true$ 
3:   ▷ The following loop jointly assigns getNext and  $m$ 
4:   for  $(getNext, m) \in \{ (getParent, k),$ 
5:      $(getLeft, l),$ 
6:      $(getRight, r) \}$ 
7:   do
8:      $pNode \leftarrow context$ 
9:      $cNode \leftarrow candidate$ 
10:    for  $i \in [0, m)$  do
11:      if  $pName(pNode) \neq pName(cNode)$  then
12:         $isMatch \leftarrow false$ 
13:        break
14:       $pNode \leftarrow getNext(pNode)$ 
15:       $cNode \leftarrow getNext(cNode)$ 
16:  if  $isMatch$  then
17:    yield candidate

```

Algorithm 1 describes how the mutate location is selected. The values  $k$ ,  $l$ , and  $r$  are global hyper-parameters that are set by the user. In our evaluation, we set  $k, l, r = 4$ .

On line 1, the walk function returns each node of the recipient parse tree in breadth-first order. Each node considered a candidate mutate location.

On line 4 of the algorithm, getNext and  $m$  are jointly assigned so that when  $getNext \leftarrow getParent$  then  $m \leftarrow k$  and when  $getNext \leftarrow getLeft$  then  $m \leftarrow l$ . The  $getParent(n)$ ,  $getLeft(n)$ , and  $getRight(n)$  functions on line 3 take a parse tree node  $n$  and return the parent node of  $n$ , the node that precedes  $n$  at the same depth, or the node that succeeds  $n$  at the same depth respectively. In Listing 7 calling  $getParent$ ,  $getLeft$ , and  $getRight$  on Location A returns the enclosing

block node (representing lines 2-7), the prior operation node (representing line 2), and the next operation node (representing line 5) respectively.

On line 11 of the algorithm, the  $pName(n)$  function returns the name of the production rule that corresponds with the parse-tree node  $n$ . For example, calling  $pName$  on the `hw.bitcast` operation of Listing 7 returns the rule name "operation".

Listings 7 and Figure 3 show an example of the  $k$ -ancestor,  $l$ -sibling, and  $r$ -sibling matching process. LOCATE compares the parameterized context to each candidate location in the recipient test case. Location A is a valid location since it has an operation as a left and right sibling (lines 3 and 5 respectively), and is nested within a block (line 2) and within a operation (line 1). Location B is invalid because it is at the end of the block and thus has no right-siblings. Location C is invalid because it has no right-siblings and its ancestor node is an operation, whereas the parameterized context requires the first ancestor to be a block.

Apart from replacing existing parse-tree nodes, SYNTHFUZZ is also able to transplant content by inserting them in locations corresponding to quantifiers in production rules of the grammar. Such quantifiers indicate that a varying length collection of terms can be generated. SYNTHFUZZ locates parse-tree nodes that correspond to production rule and inserts a new node corresponding to the quantified term. SYNTHFUZZ can then apply the same  $k$ -ancestor,  $l(r)$ -sibling matching logic described earlier to decide if the newly inserted node is a suitable mutation location. This grants SYNTHFUZZ greater flexibility, increasing the diversity of inputs generated.

```

1 "hw.module"() { {
2   ^bb0(%arg0: i4, %arg1: !hw.array<2xi2>):
3   %0 = "hw.bitcast"(%arg1) : (!hw.array<2xi2>) -> i4
4   Location A
5   "hw.output"(%1) : (i4) -> ()
6   Location B
7 }
8 Location C
9 )

```

Listing 7. A set of potential insertion locations are marked as A, B, and C in the recipient test case.

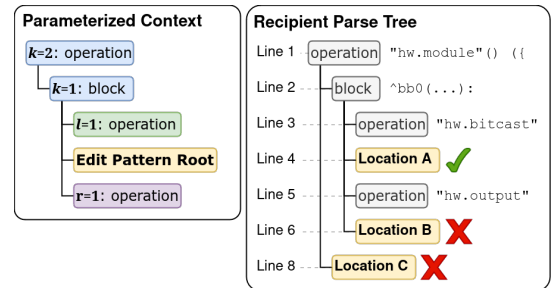


Fig. 3. Illustration of  $k$ -ancestor and  $l(r)$ -sibling context matching. Location B is invalid due to not matching the postfix context with  $r = 1$ . Location C is invalid due to not matching the  $k$ -ancestor path context as the parent node is an operation, not a block with  $k = 2$ .

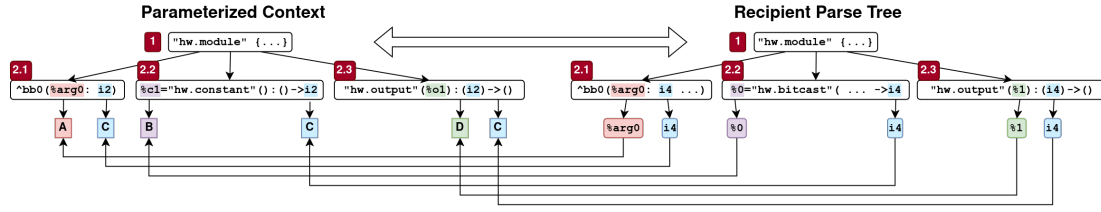


Fig. 4. An illustration of the MATCH step. When the recipient test case in Listing 8 is matched with the parameterized context shown in this figure, the parameters A, B, C, and D are bound to the concrete values %arg0, %0, %i4, and %1 respectively.

### C. MATCH: Matching and extracting parameters

Once an mutate location is selected in the LOCATE step, MATCH performs a joint breadth-first traversal of the parameterized context tree and the parse-tree of the recipient test case to compute parameter assignments. The traversal starts at the mutate location of the recipient tree and the root of the parameterized mutation within the parameterized context. An annotated example of this traversal is illustrated in Figure 4. A node pair is considered a match if the parse tree node name is the same between the parameterized context and recipient parse tree. This allows parse tree nodes of different concrete operations to match. For example, node pair 2.2 in Figure 4 is considered a match even though the operation names (`hw.constant` and `hw.bitcast`), types (`i2` and `i4`), and values (`%c1` and `%0`) differ.

When a parameter node is encountered during the traversal of the parameterized context, SYNTHFUZZ assigns it to the corresponding sub-tree in the traversal of the parse tree of the recipient input. In Figure 4, the parameters A, B, C, and D from the parameterized context are assigned to the matching nodes for %arg0, %0, %i4, and %1 in the recipient parse tree. When there are duplicate assignments as in parameter C, MATCH will uniformly randomly select one of the assignments to use.

### D. INSTANTIATE: Concretize the mutation

```

1 "hw.module"() { {
2   ^bb0(%arg0: i4, %0 = "hw.bitcast"(%arg1) : (!hw.array<2xi2>)
   -> i4
3   %1 = "comb.add"(%arg0, %0) : (i4, i4) -> i4
4   "hw.output"(%1) : (i4) -> ()
5 })

```

Listing 8. The recipient test case where the parameterized mutation is inserted and concretized. The parameterized mutation is boxed on line 4. By instantiating the pattern in this new context, the following substitutions are made: A ← %arg0, B ← %0, C ← %i4, and D ← %1.

In the INSTANTIATE step, SYNTHFUZZ adapts the parameterized mutation to the recipient test case by substituting in the parameter assignments extracted during the MATCH step.

Listing 8 shows how the parameterized mutation is instantiated. Here, parameters A, B, C, and D corresponding the return value, two operands, and the types are assigned the values %arg0, %0, %i4, and %1 respectively.

SYNTHFUZZ also checks that the mutated input conforms to generic MLIR constraints before passing them to the compiler.

For example, a mutation may lead to test cases that redefine the same variable twice or use an undefined variable. Such test cases are invalid in any MLIR program, regardless of dialect.

## IV. EVALUATION

In our study, we examine the following research questions:

- RQ1:** How effective is SYNTHFUZZ in terms of increasing code coverage?
- RQ2:** What is the diversity of test cases generated by SYNTHFUZZ in terms of dialect pair coverage?
- RQ3:** Does context-based positioning of mutation locations improve the likelihood of a valid mutation?
- RQ4:** Does parameterization of the mutation content improve the likelihood of a valid mutation?

### A. Experiment Design

TABLE I  
BENCHMARK PROGRAMS

Subject Program	Description	# of Seed Test Cases	# of Dialects
mlir-opt (P1)	Includes the core and contributed MLIR dialects part of the LLVM project.	1,692	42
onnx-mlir-opt (P2)	An MLIR-based ONNX compiler.	1,885	2
triton-opt (P3)	An MLIR-based compiler for the Triton language.	29	4
circt-opt (P4)	An MLIR-based compiler for electronic design automation (EDA).	377	26

It is costly to hand-write custom test generators for MLIR-opt and CIRCT projects due to the large number of MLIR dialects they define. MLIRSmith supports only 13 dialects currently.

In our evaluation of SYNTHFUZZ, we select four active projects that use the MLIR compiler infrastructure, shown in Table I. Each project provides a utility executable named `<project>-opt` which is used to independently invoke and test one or more compiler passes. To fuzz each project, we invoke the `<project>-opt` executable with a pipeline of  $P$  randomly selected compiler passes on each test case generated by a fuzzer. In practice, we set  $P = 5$  since most unit test cases written by developers only invoke one to three compiler passes at a time.

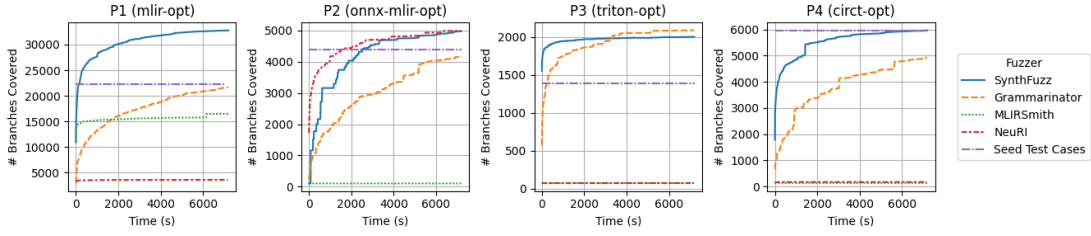


Fig. 5. Branch coverage for each subject program. SYNTHFUZZ outperforms a grammar-based fuzzer by up to  $1.51\times$  and improves coverage by up to  $1.47\times$  compared to existing seed tests.

For ONNX-MLIR, LLVM, and CIRCT, we select compiler passes based on the dialects present, identified by the operator names in the test case. For example, if a test case contains the `arith.maxsi` operation, then our test driver will select a compiler pass that operates on the `arith` dialect such as the `arith-to-llvm` pass which lowers certain `arith` operations to the `llvm` dialect. As Triton’s compiler passes do not follow this naming scheme and this heuristic cannot be used, we select compiler passes randomly from all available passes.

1) *Locating Seed Test Cases*: We build a corpus of seed test cases for each subject program by locating and splitting “.mlir” files in each subject program’s respective project repositories for a total of 1,692, 1,885, 26, and 377 seed test cases respectively. Then we convert each test case into its generic MLIR syntax form to remove syntactic sugar and enable it to be parsed using a single generic MLIR grammar for all subject programs.

2) *Evaluation Measures*: To evaluate and compare the fault detection potential of SYNTHFUZZ, we use the following measures:

- **Branch coverage** is computed as the number of covered branches as reported by LLVM’s SanitizerCoverage code coverage instrumentation.
- **Dialect coverage** is a measure of the input diversity used in the evaluation of prior work, MLIRSmith [9]. This notion can be further divided into control dependent dialect coverage and data dependent dialect coverage. The idea is to use multiple dialects in tandem, and to ensure that distinct dialects are used together in a meaningful manner by being connected through data or control dependencies.
  - **Control dialect coverage** is computed by counting the number of unique dialect pairs whose operations are linked by a control dependency. For example, Listing 9 contains two control dependent dialect pairs, (`comb`, `sv`) and (`hw`, `sv`), since `hw.constant` (line 2) and `comb.icmp` (line 3) are nested within `sv.if` (line 1) forming a control dependence.
  - **Data dialect coverage** is computed by counting the number of unique dialect pairs whose operations are linked by a data dependency. For example, Listing 9 contains one data dependent dialect pair, (`scf`, `hw`) since `comb.icmp` (line 3) takes the output `%18` of

`hw.constant` (line 2), thus forming a data dependence.

```
1 "sv.if"(%18 = "hw.constant"() {value = 10 : i32} : () ->
   i32
2), { // ...
```

Listing 9. In this example, `hw.constant` (line 3) and `comb.icmp` (line 4) have a control dependence on `sv.if` (line 2), forming two dependent dialect pairs (`comb`, `sv`) and (`hw`, `sv`). `comb.icmp` (line 4) also has a data dependence on `hw.constant` (line 3) forming a data dependent dialect pair (`scf`, `hw`).

3) *Baselines*: We evaluate SYNTHFUZZ against Grammarinator [6], MLIRSmith [9], and NeuRI [17]. Grammarinator represents a baseline grammar-based fuzzer, while MLIRSmith and NeuRI represent the state of the art custom generators for MLIR and deep learning models.

Grammarinator converts ANTLR4 grammar definitions into generation models that can be used to generate, mutate, and recombine test cases following the grammar. In our experiments, we provide both SYNTHFUZZ and Grammarinator with the same seed corpus for a fair comparison.

MLIRSmith is a custom generator-based fuzzer that targets MLIR’s core dialects. MLIRSmith was chosen for comparison as it specializes in generating valid random MLIR programs, but requires users to implement custom generators for each new dialect. As of March 2024, MLIRSmith supports 13 out of 42 available core dialects [13]. NeuRI is a DL model-level fuzzer that generates computation graphs based on DL-model specific, API-level constraints. NeuRI supports parameterization only in the form of tensor shape constraints and shape propagation rules for machine learning models. While NeuRI does not directly target MLIR, the `onnx-mlir` tool can be used to lower the models generated by NeuRI to the core MLIR dialects for P1 (`mlir-opt`) or to the `onnx` dialect for P2 (`onnx-mlir-opt`).

We also compare SYNTHFUZZ against running existing test cases (i.e., the seeds for SYNTHFUZZ and Grammarinator). By measuring the branch and dialect coverage of the seed test cases alone, we separate the contributions of the increased coverage afforded by the fuzzers’ mutation capability from the innate capability of the seed corpus they draw from.

4) *Experimental Environment*: All experiments are performed on an AMD Ryzen 2950X 16-Core Processor with 32 GB of RAM running on Ubuntu 22.04.

TABLE II  
DIALECT COVERAGE FOR EACH SUBJECT PROGRAM.

Subject	P1 (mlir-opt)			P2 (onnx-mlir-opt)			P3 (triton-opt)			P4 (circt-opt)		
	dialects	control	data	dialects	control	data	dialects	control	data	dialects	control	data
SynthFuzz	27	<b>129</b>	<b>100</b>	<b>9</b>	<b>23</b>	<b>24</b>	<b>7</b>	<b>15</b>	<b>4</b>	20	<b>86</b>	<b>40</b>
Seed tests as is	<b>28</b>	68	56	<b>9</b>	17	11	<b>7</b>	12	<b>4</b>	<b>23</b>	57	<b>40</b>
Grammarinator	27	63	49	8	15	8	<b>7</b>	12	<b>4</b>	17	43	27
MLIRSmith	13	62	65	4	6	3	3	3	1	6	15	10
NeuRI	7	17	12	6	9	10	3	3	2	6	14	10

Seed test cases refers to each subject’s respective test suite. SYNTHFUZZ achieves greater dialect coverage compared to baseline fuzzers.

## B. RQ1: Branch Coverage

Figure 5 shows the branch coverage of SYNTHFUZZ and the baseline fuzzers on the four subject programs. Averaging across all subject programs, SYNTHFUZZ outperforms Grammarinator, MLIRSmith, and NeuRI by  $1.22\times$ ,  $29.78\times$ ,  $17.47\times$  respectively.

On P1 (mlir-opt) SYNTHFUZZ outperforms Grammarinator, MLIRSmith, and NeuRI by  $1.51\times$ ,  $1.99\times$ , and  $9.06\times$  respectively. On P4 (mlir-opt) SYNTHFUZZ outperforms Grammarinator, MLIRSmith, and NeuRI by  $1.21\times$ ,  $43.25\times$ , and  $34.50\times$  respectively.

On P2 (onnx-mlir-opt), SYNTHFUZZ is similar to NeuRI, with less than 1% difference in coverage. This demonstrates that SYNTHFUZZ can match the performance of a domain-specific fuzzer, NeuRI, by bootstrapping parameterized mutations from existing test cases without hand-coding any custom generator logic. NeuRI implements custom test generator logic for ONNX models (computation graphs for DL models).

On P3 (triton-opt), Grammarinator outperforms SYNTHFUZZ in terms of branch coverage by  $1.04\times$ . P3 only provides 36 test cases in its repository that SYNTHFUZZ and Grammarinator could use as seeds. Since SYNTHFUZZ relies upon seed test cases to synthesize its custom mutations, the low number of seeds constrained SYNTHFUZZ’s ability to generate diverse test cases.

**An example bug found.** SYNTHFUZZ discovered a new bug in CIRCT (Issue #6799), which has been confirmed and fixed by the developers. This bug occurs when the `--convert-llhd-to-llvm` pass is invoked on an `llhd.proc` operation with a block without a terminator operation, such as `llhd.wait` or `llhd.halt`. The CIRCT compiler incorrectly assumes that the `llhd.proc` operation always contains a terminator, and crashes due to the violation of this dialect-specific requirement. The CIRCT verifier should have detected this absence of a terminator and rejected the `llhd.proc` operation.

```
module {llhd.proc @empty() -> () { }}
```

Listing 10. CIRCT crashes on this minimized input due to not checking for a terminator within `llhd.proc`. Reported on Mar 7, 2024, the bug was fixed immediately on Mar 8.

SYNTHFUZZ demonstrates up to  $1.51\times$  improvement on branch coverage over most baseline fuzzers without requiring any hand-coding of custom generator logic.

## C. RQ2: Dialect Diversity

Table II summarizes the performance of SYNTHFUZZ as compared to Grammarinator, MLIRSmith and NeuRI in terms of dialect control/data pair coverage (described in Section IV-A2). SynthFuzz outperforms Grammarinator, MLIRSmith and NeuRI by  $1.70\times$  and  $4.16\times$  and  $5.32\times$  in terms of control dependent dialect pairs and  $1.88\times$ ,  $4.38\times$ ,  $4.18\times$  in terms of data dependent dialect pairs.

To validate whether SYNTHFUZZ discovers new dialect pairs not covered by the seed corpus, we also measure the dialect pair coverage of the seed corpus as a baseline. For each subject and fuzzer combination, we report the number of unique dialects and the number of unique control and data dialect pairs. Across all subjects, SYNTHFUZZ covers 99 new control-dependent and 57 new data-dependent dialect pairs that did not already exist in the seed corpus.

```
1 "func.func"() <{function_type = (i1, i32, i32) -> i32,
   sym_name = "main"}> ({
2   ^bb0(%arg0: i1, %arg1: i32, %arg2: i32):
3   %0 = "arith.addi"(%arg1, %arg2) <{overflowFlags =
   #arith.overflow<none>}> : (i32, i32) -> i32
4   %1 = "arith.shli"(%0, %arg1) <{overflowFlags =
   #arith.overflow<none>}> : (i32, i32) -> i32
5   %6 = "comb.icmp"(%1, %1) <{predicate = 1 : i64}> : (i32,
   i32) -> i1
6   %2 = "arith.subi"(%1, %0) <{overflowFlags =
   #arith.overflow<none>}> : (i32, i32) -> i32
7   %3 = "arith.select"(%arg0, %3, %0) : (i1, i32, i32) -> i32
8   "func.return"(%4) : (i32) -> ()
9 }) : () -> ()
```

Listing 11. A test case generated by SYNTHFUZZ that was not generated by any baselines. SYNTHFUZZ inserts the underlined line 5 and substitutes in the operand %1 and type i32, satisfying the required def-use and type consistency constraints. This test case achieves a new dialect pair coverage, (comb, arith) by introducing the `comb.icmp` operator.

Listing 11 shows an example test case that can be generated by SYNTHFUZZ, but cannot be generated by the baselines. The `comp` dialect is not supported by MLIRSmith. MLIRSmith currently implements test generator logic for 13 dialects only, as it takes 447 lines of code on average to support each dialect. Grammarinator’s naive recombination fails to satisfy def-use and type consistency constraints by inserting line 5 as `%6 = "comb.icmp"(%5, %4) <{predicate = 0 : i64}> : (i2, i2) -> i1`, which references an un-



defined variable %5 and uses an incorrect type i2 for variable %4. This is because Grammarinator’s recombination is context-unaware and is not concretized to fit the target insertion context.

While Grammarinator takes the same set of seed tests as SYNTHFUZZ, it achieves lower dialect diversity than SYNTHFUZZ or the seed tests alone, because Grammarinator alternates between generation, mutation, and recombination modes. Approximately, one third of its time is spent on the pure generation mode that does not use seed test cases.

SYNTHFUZZ achieves greater dialect diversity, compared to other baseline fuzzers: average  $1.50\times$  improvement in terms of control-dependent dialect pairs and average  $1.43\times$  in data-dependent dialect pairs.

#### D. RQ3: Context-based Positioning of Mutation

SYNTHFUZZ selects an appropriate insertion location to inject a parameterized mutation by matching the mutation’s parameterized context against the target context of the recipient test case. To test the individual effect of the context matching requirement for a parameterized mutation, we vary  $k$  from 0, 2, and 4 when matching a  $k$ -ancestor path. Similarly, we vary  $l$  from 0, 2 and 4, when matching a  $l$ -sibling prefix  $l$ , and we vary  $r$  from 0, 2, and 4 when matching a  $r$ -sibling postfix. Each trial consists of 10,000 test cases generated by SYNTHFUZZ for P1 (mlir-opt).

As shown in Table III, setting each parameter  $k$ ,  $l$ , and  $r$  to 4 improves the number of valid test cases by  $1.11\times$ ,  $1.07\times$ , and  $1.03\times$  respectively. This indicates that using more context information increases the chance of finding an appropriate location for injecting a grafted mutation, thus increasing the portion of valid test cases. A test case is considered *valid*, if feeding the generated input to the target program returns zero, indicating a success.

Setting  $k = 4$  decreases dialect pair coverage by 10%. This may indicate that a very restrictive requirement for matching context by increasing  $k$  can negatively affect the input diversity of generated tests. Additional experiments with  $k$  greater than 4 had minimal effect on the number of valid tests, as most seed tests have an operation nesting depth less than 4.

Increasing ancestor-path, prefix, and postfix requirements for context positioning improves the proportion of valid test cases.

#### E. RQ4: Effect of Parameterization

SYNTHFUZZ has capability to parameterize and concretize a variable name, an argument’s type, and an operation’s attribute (e.g. "hw.constant"() {value = -2 : i2} has a value attribute -2 with the type i2) to fit the target context where a grafted mutation is inserted into. The goal of parameterization is to preserve context-sensitive constraints such as the def-use constraint (i.e., a value must be defined before use) and the type-constraint (i.e., the type annotation of a value must be consistent throughout its scope).

TABLE III  
AVERAGE BRANCH COVERAGE, DIALECT PAIR COVERAGE, AND VALID TEST CASES.

Parameter	Branch Cov.	Dialect Pair Cov.	Valid Test Cases
$k = 0$	24,764	<b>100</b>	677
$k = 2$	24,765	<b>100</b>	698
$k = 4$	<b>24,987</b>	90	<b>749</b>
$l = 0$	<b>25,055</b>	94	684
$l = 2$	24,749	97	710
$l = 4$	24,713	<b>98</b>	<b>729</b>
$r = 0$	24,657	97	706
$r = 2$	24,901	95	687
$r = 4$	<b>24,958</b>	<b>98</b>	<b>731</b>

Increasing  $k$ ,  $l$ , and  $r$  to 4 increases the number of valid test cases by  $1.11\times$ ,  $1.07\times$ , and  $1.03\times$

TABLE IV  
VALIDITY OF TEST CASES GENERATED BY SYNTHFUZZ.

Violation Type		W/ Param.	W/O Param.
Invalid	Dialect Specific	Count	4,259
		Percent	38.1%
	General MLIR	Count	1,777
		Percent	15.9%
	Invalid Options	Count	4,356
		Percent	39.0%
Valid	Valid	Count	772
		Percent	6.9%

Parameterization reduces General MLIR violations from 30.2% to 15.9%.

We create a downgraded version of SYNTHFUZZ by turning off its parameterization and concretization capability denoted as W/O Param in Table IV. We generate 10,000 test cases with each version and categorize the test cases based on the error message returned by the target program.

Parameterization increases the proportion of valid test cases by 0.01% only. However, when we further inspect the underlying reasons for invalid test cases, we find that SYNTHFUZZ increases the chance of adhering to the general MLIR constraints.

With parameterization, 772 tests are valid with the return value zero indicating success, when the tests are fed to the target program. We then categorize the remaining 9228 invalid test cases into three categories based on the type of violation reported by the target program.

- **Dialect Specific:** 4,259 test cases generated with parameterization are rejected by the target program with a dialect-specific error message such as: `tosa.logical_or op result #0 must be tensor of 1-bit signless integer values`, `tosa.floor op requires a single operand`, etc.
- **General MLIR:** 1,777 test cases generated with parameterization are rejected by the target program with a general MLIR error message such as: an undefined symbol, use of undeclared SSA, redefinition of SSA value, etc.

- **Invalid Options:** 4,356 test cases generated with parameterization are rejected with an error message, “no such option exists.” This occurs due to the test driver’s random pass selection which may pair an option with a pass that does not accept said option.

With parameterization enabled, SYNTHFUZZ generates 1,343 fewer test cases that violate general MLIR constraints out of 10,000 tests. The proportion of Invalid Options category is approximately the same with and without parameterization. However, the proportion of General MLIR invalidity increases from 15.9% to 30.2% when disabling parameterization. This is due to the fact that without parameterization, the content of parameterized mutation is not concretized to fit the recipient context. Thus it is more likely to violate general MLIR constraints such as def-use and type consistency.

Listing 11 shows an example of a test case generated by SYNTHFUZZ which nests the `comb.icmp` operation within a `func.func` operation. SYNTHFUZZ parameterizes the input arguments and their types, thus passing the general MLIR constraints such as def-use and type consistency.

SYNTHFUZZ reduces the proportion of general MLIR constraint violating tests from 30.2% to 15.9% by parameterizing the injected mutation’s content.

## V. THREATS TO VALIDITY

1) *Limited Fuzzing Time:* In our experiments on code and dialect coverage, we limit the fuzzing budget to 4 hours for each fuzzer. While unlikely, continuing the fuzzing campaign for longer may reveal different trends than our evaluation showed.

2) *Choice of Subject Programs:* The results of our evaluation is influenced by our choice of subject programs. To minimize bias, we selected four MLIR projects to represent a wide variety domains among 40 possible public MLIR projects. P1 (the LLVM/MLIR project) was chosen as it contains the original core MLIR dialects that MLIRSmith defines its custom generators for. P2 is a deep-learning compiler for ONNX models that NeuRI can directly fuzz. P3 is a compiler for the Triton language. P4 (CIRCT) is a novel application of MLIR to the domain of hardware design and synthesis.

## VI. RELATED WORK

**Grammar-based fuzzing.** Grammar-based fuzzers, e.g. Grammarinator [6], Nautilus [14], LangFuzz [18], are guided by a grammar. PolyGlott [19] transforms the high-level languages into a general IR with the BNF grammar given by the users and uses constraint mutators to preserve the grammar. Our experiments found that 97% of Grammarinator’s generated inputs fail to satisfy semantic constraints, e.g., type consistency, or more complex relationships over shapes and types. SYNTHFUZZ, instead, automatically encodes these semantic constraints from examples.

**Custom generator- and mutation-based fuzzers.** Generator-based fuzzers [20], [21] require generators, written by human

developers in an imperative language, to produce valid inputs. CSmith [7] generates random C programs for fuzzing C compilers. Related to our target subject of MLIR, MLIRSmith [9] generates random MLIR programs. Unlike grammar-based fuzzers, they can encode constraints that are difficult to express in grammar. Despite the additional human effort required, MLIRSmith underperforms SYNTHFUZZ as it is unable to satisfy the constraints of invoking each operator of a new dialect. SYNTHFUZZ does not require the definition of domain-specific custom mutators to successfully invoke operators from novel dialects.

Several studies have proposed fuzzers targeted at specific domains. Apart from fuzzing MLIR dialects, BigFuzz [22] was proposed for Apache Spark programs, Qdiff [23] for quantum programs, HeteroFuzz [24] for heterogeneous applications. GrayC [25] designed custom mutators for fuzzing C programs. NNSmith [8] designed a generator for computation graphs to fuzz deep learning compilers. To effectively generate inputs, these fuzzers employ custom mutators. These mutators are hand-crafted and manually implemented for each domain. The amount of work needed to develop a fuzzer for each domain highlights the need for approaches capable of automatically synthesizing custom mutators.

The approaches above do not allow for mutations that are parameterized. The closest work to SYNTHFUZZ is NeuRI [17], which infers constraints over tensor shapes to generate deep learning (DL) models for fuzzing DL compilers. However, because it is specific to DL compilers and limited in the types of constraints inferred, NeuRI did not achieve a high coverage in our experiments.

**Learning constraints.** Other techniques check that inputs satisfy constraints during fuzzing. Dewey et al. [26] uses Constraint Logic Programming to specify constraints test generation. ISLA [16] allows semantic constraints to be applied during grammar-based fuzzing. These techniques require significant human effort, requiring human-written constraints by hand [26] or templates [16].

**Learning code patterns and transformations.** Code pattern inference techniques have been adopted in studies for code search [27], mining rules for detecting bugs [28], and code quality [29]. Other techniques synthesize patches, or the transformation of one program to a subsequent version of the program. These techniques [10], [30], [12], [31], [11], [32], [33] learn to transform programs from examples. They identify parameterized patches given examples of the transformation, using the observation that code elements reoccurring across multiple patches are usually essential. SYNTHFUZZ draws inspiration from these techniques by learning parameterized mutations from examples. While these techniques aim to mutate a buggy program into a single, correct program, SYNTHFUZZ aims to generate a diversity of mutants for fuzzing.

## VII. CONCLUSION

We present SYNTHFUZZ, a novel approach to compiler fuzzing that address the challenges in testing extensible compilers. SYNTHFUZZ is able to synthesize custom mutations

for new MLIR dialects. In contrast, domain-specific fuzzers require months of development, which is impractical given the rapid evolution of target languages and representations. SYNTHFUZZ synthesizes custom parameterized, context-dependent mutations from the test cases of each dialect compiler, exploiting the observation that the dialect-specific constraints are implicitly encoded in these tests. SYNTHFUZZ can cover 60 dialects that cannot be covered by existing fuzzers, and outperforms existing grammar-based and domain-specific fuzzers in terms of branch coverage by  $1.16\times$  and dialect coverage by  $1.51\times$ .

A replication package has been made available at <https://figshare.com/s/b96ea4a64f6c6a0ece12>.

## REFERENCES

- [1] A. F. Brown and G. Wilson, “The architecture of open source applications,” 2011. [Online]. Available: <https://api.semanticscholar.org/CorpusID:14285235>
- [2] “CIRCT: Circuit IR Compilers and Tools,” <https://circt.llvm.org/>, 2023, accessed: 2023-03-11.
- [3] M. Amini, R. Riddle, and Google, “MLIR: Multi-level intermediate representation,” [https://llvm.org/devmtg/2020-09/slides/MLIR\\_Tutorial.pdf](https://llvm.org/devmtg/2020-09/slides/MLIR_Tutorial.pdf), 2020, [Accessed 2024-03-15].
- [4] <https://mlir.llvm.org/users/>, 2022.
- [5] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, Aug. 2020.
- [6] R. Hodován, A. Kiss, and T. Gyimóthy, “Grammarinator: A grammar-based open source fuzzer,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 45–48. [Online]. Available: <https://doi.org/10.1145/3278186.3278193>
- [7] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” *SIGPLAN Not.*, vol. 46, no. 6, p. 283–294, jun 2011. [Online]. Available: <https://doi.org/10.1145/1993316.1993532>
- [8] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, “Nnsmith: Generating diverse and valid test cases for deep learning compilers,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 530–543. [Online]. Available: <https://doi.org/10.1145/3575693.3575707>
- [9] H. Wang, J. Chen, C. Xie, S. Liu, Z. Wang, Q. Shen, and Y. Zhao, “Mlirsmith: Random program generation for fuzzing mlir compiler infrastructure,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 1555–1566. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00120>
- [10] N. Meng, M. Kim, and K. S. McKinley, “LASE: locating and applying systematic edits by learning from examples,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 502–511.
- [11] R. Rolim, G. Soares, L. D’Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, “Learning syntactic program transformations from examples,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 404–415.
- [12] L. Serrano, V.-A. Nguyen, F. Thung, L. Jiang, D. Lo, J. Lawall, and G. Muller, “SPINFER: Inferring semantic patches for the linux kernel,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 235–248.
- [13] Colloportus0, “GitHub - colloportus0/mlirsmith — github.com,” <https://github.com/Colloportus0/MLIRSmith>, [Accessed 20-03-2024].
- [14] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, “NAUTILUS: Fishing for deep bugs with grammars,” in *NDSS*, 2019.
- [15] J. Wang, B. Chen, L. Wei, and Y. Liu, “Superion: Grammar-aware greybox fuzzing,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 724–735.
- [16] D. Steinhöfel and A. Zeller, “Input invariants,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 583–594.
- [17] J. Liu, J. Peng, Y. Wang, and L. Zhang, “Neuri: Diversifying dnn generation via inductive rule inference,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 657–669. [Online]. Available: <https://doi.org/10.1145/3611643.3616337>
- [18] C. Holler, K. Herzig, and A. Zeller, “Fuzzing with code fragments,” in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 445–458.
- [19] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, “One engine to fuzz ‘em all: Generic language processor testing with semantic validation,” in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 642–658.
- [20] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, “Semantic fuzzing with zest,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.
- [21] R. Padhye, C. Lemieux, and K. Sen, “Jqf: Coverage-guided property-based testing in java,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 398–401.
- [22] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, “BigFuzz: Efficient fuzz testing for data analytics using framework abstraction,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 722–733.
- [23] J. Wang, Q. Zhang, G. H. Xu, and M. Kim, “QDiff: Differential testing of quantum software stacks,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 692–704.
- [24] Q. Zhang, J. Wang, and M. Kim, “HeteroFuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 242–254.
- [25] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, “Grayc: Greybox fuzzing of compilers and analysers for c,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1219–1231. [Online]. Available: <https://doi.org/10.1145/3597926.3598130>
- [26] K. Dewey, J. Roesch, and B. Hardekopf, “Language fuzzing using constraint logic programming,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 725–730.
- [27] A. Sivaraman, T. Zhang, G. Van den Broeck, and M. Kim, “Active inductive logic programming for code search,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 292–303.
- [28] H. J. Kang and D. Lo, “Active learning of discriminative subgraph patterns for API misuse detection,” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2761–2783, 2021.
- [29] P. Garg and S. H. Sengamedu, “Synthesizing code quality rules from examples,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA2, pp. 1757–1787, 2022.
- [30] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, “Semantic patch inference,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 382–385.
- [31] M. Lamothe, W. Shang, and T.-H. P. Chen, “A3: Assisting android api migrations using code examples,” *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 417–431, 2020.
- [32] S. A. Haryono, F. Thung, D. Lo, L. Jiang, J. Lawall, H. J. Kang, L. Serrano, and G. Muller, “Androevolve: automated android api update with data flow analysis and variable denormalization,” *Empirical Software Engineering*, vol. 27, no. 3, p. 73, 2022.
- [33] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, “Inferring program transformations from singular examples via big code,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 255–266.