Fuzzing Deep Learning Compilers with HirGen

Haoyang Ma
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
China
haoyang.ma@connect.ust.hk

Qingchao Shen
College of Intelligence and Computing, Tianjin University
China
qingchao@tju.edu.cn

Yongqiang Tian
yongqiang.tian@uwaterloo.ca
Cheriton School of Computer Science, University of Waterloo
Canada
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
China

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

Shing-Chi Cheung*
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
China
scc@cse.ust.hk

ABSTRACT

Deep Learning (DL) compilers are widely adopted to optimize advanced DL models for efficient deployment on diverse hardware. Their quality has profound effect on the quality of compiled DL models. A recent bug study shows that the optimization of high-level *intermediate representation* (IR) is the most error-prone compilation stage. Bugs in this stage account for 44.92% of the whole collected ones. However, existing testing techniques do not consider high-level optimization related features (e.g. high-level IR), and are therefore weak in exposing bugs at this stage. To bridge this gap, we propose HirGen, an automated testing technique that effectively expose coding mistakes in the optimization of high-level IR. The design of HirGen includes 1) three coverage criteria to generate diverse and valid computational graphs; 2) the use of high-level IR's language features to generate diverse IRs; 3) three test oracles of which two are inspired by metamorphic testing and differential testing. HirGen has successfully detected 21 bugs that occur at TVM, with 17 bugs confirmed and 12 fixed. Further, we construct four baselines using state-of-the-art DL compiler fuzzers that can cover the high-level optimization stage. Our experiment results show that HirGen can detect 10 crashes and inconsistencies that cannot be detected by the baselines in 48 hours. We also evaluate the usefulness of our proposed coverage criteria and test oracles.

CCS CONCEPTS

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

KEYWORDS

Deep Learning Compiler, Software Testing

1 INTRODUCTION

Deep learning (DL) compilers, such as TVM [1], Glow [2], XLA [3] and nGraph [4], have shown the effectiveness in optimizing advanced DL models for efficient model deployment at diverse devices [5]. They take as input a DL model, extract its computational graph, and re-represent the DL model using intermediate representations (IRs) [5]. DL compilers consist of multiple compilation stages, which include high-level and low-level optimizations. DL compilers arrange these two optimizations in order, with high-level optimization followed by low-level optimization. The optimizations aim to compile deep learning models into binary executables that can run efficiently on target hardware devices.

Like conventional compilers[6, 7], DL compilers are prone to bugs. These bugs can cause undesired compiler behaviors, such as crash, unexpected wrong behavior and poor performance [8]. These undesired behaviors could result in catastrophic effects on the correctness and reliability of mission-critical DL applications (*e.g.*, autonomous driving cars [9] and aircraft collision avoidance systems [10]).

Techniques have been recently proposed to detect bugs in DL compilers, including TZER [11], TVMFuzz [12], MT-DLComp [13] and NNSmith [14]. Despite preliminary reported success in bug detection for TVM, they are inefficient in revealing bugs that occur in high-level optimization, which account for 44.92% of the bugs found in DL compilers [8]. TZER and TVMFuzz [11, 12] are proposed to detect low-level optimization bugs in a DL compiler with generated low-level IRs. Since these two techniques test DL compilers by mutating low-level IR, and low-level IR cannot be used by high-level optimization, they theoretically cannot detect bugs in the high-level optimization stage. MT-DLComp [13] tests a DL compiler by constructing mutated DL models. Since its mutation strategies only insert operators that yield zero, the kinds of operators and the available places to insert these operators are limited. Therefore, it cannot generate models of diverse computational graphs to cover corner high-level optimization cases. In test oracle

要经过两个优化：低级和高级

提出在生成低级IR的DL编译器中检测低级优化bug

插入运算符受限，难以触及高级优化部分

*Shing-Chi Cheung is the corresponding author.

design, MT-DLComp does not take advantage of the language features of high-level IR and high-level optimizations. As a result, it cannot effectively detect bugs in high-level optimizations (We will prove it in section 5). NNSmith [14] mainly focuses on revealing the hidden defects in DL compilers, such as arithmetic problem, fragile type system, poor support for specific data layouts, by generating computational graphs and inputs. Since its generation process and test oracle design do not consider language features of high-level IR in DL model generation, and do not consider high-level optimizations in test oracle design, it cannot efficiently detect high-level optimization-related bugs. In Section 5, we will show NNSmith is orthogonal to HirGen in terms of bug detection ability.

To bridge the gap, we propose the first DL compiler fuzzing technique that focuses on high-level optimization: HirGen. HirGen is designed to satisfy the following four objectives: 1) the satisfaction of integrity constraints, such as type match and tensor shape match, that govern high-level IR to avoid an early crash before invoking optimization, 2) the exploration of the diversity of computational graphs, 3) utilization of high-level IR language features for the construction of diverse high-level IRs, 4) the capability of detecting multiple types of optimization bugs. To achieve the first objective, HirGen performs type checking and shape checking in each insertion of the operator node by leveraging the information of each existing node, including its type, shape, and connections. After insertion, HirGen also updates the information of the new node for future use. To meet the second objective, HirGen is coverage-guided in input space to explore diverse operator nodes, operator edges, and the combination of operator type and data type. To meet the third objective, HirGen can construct diverse high-level IRs from a single computational graph to achieve full use of IR's language features. To meet the fourth objective, HirGen incorporates three test oracles, two of which are designed purposely for DL compilers. An example of test oracles is that a model should not make a different prediction after optimization. Besides functional correctness, HirGen can also test the robustness of DL compilers. Specifically, HirGen provides an option of generating invalid computational graphs that violate type constraints and shape constraints [15]. The option aims to test whether DL compilers can catch such invalid computational graphs and throw the expected exceptions. In this way, HirGen can also detect bugs caused by incorrect exception handling.

Following prior works on DL compiler testing, we evaluate the performance of HirGen on TVM, which is the most popular DL compiler. For baseline selection, we choose 1) TVMfuzz (with lowercase f), a preliminary proof-of-concept application from a bug study [8]. The tool is chosen because it is the only testing technique that focuses on detecting bugs arising from high-level optimizations in DL compilers; 2) MT-DLComp [13], a metamorphic testing framework that can cover the high-level optimization stage; 3) LEMON [16], a fuzzing technique for DL libraries (e.g., Tensorflow [17], PyTorch [18]) testing; and 4) NNSmith [14], a generation-based DL compiler fuzzer. We repeated the comparison experiment for ten times to mitigate the influence of randomness. Our experimental results show that 1) HirGen can detect around ten distinct crashes that are not detectable by other techniques in a two-day execution; 2) TVMfuzz, MT-DLComp, and LEMON are all inefficient in detecting bugs, they found about three crashes in total; 3) NNSmith is also efficient in bug detecting, detecting 9 distinct crashes. But except for one crash, all the crashes that they found are orthogonal to the crashes found by HirGen. We will elaborate on the experimental results in section 5.1.2. In addition to this comparison experiment, we examine the usefulness of the coverage-guided strategy in generating diverse computational graphs by an ablation study in section 5.5.

In summary, we make three major contributions.

- This work introduces a new focus on testing the most bug-prone stage, high-level optimization, of DL compilers. We propose a computational graph generation algorithm and three test oracles to detect bugs of diverse root causes in the implementation of high-level optimization.
- We have implemented HirGen, a fuzzing technique targeting at TVM. HirGen is implemented in 3K lines of C++ code. It has detected 21 bugs, of which 17 have been confirmed, 12 have been fixed, and 14 were previously unknown. Among the 17 confirmed bugs, 14 are highly related to high-level optimizations, and three are about low-level optimization and deployable code generation. Furthermore, we have conducted an experimental study to compare HirGen with TVMfuzz, MT-DLComp, LEMON and NNSmith, the state-of-the-art testing techniques that can cover the high-level optimization stage. We have also discussed the utility of each component of HirGen.
- We release HirGen, the details of detected bugs and experiment data at https://github.com/anonymousWork000/HirGen.

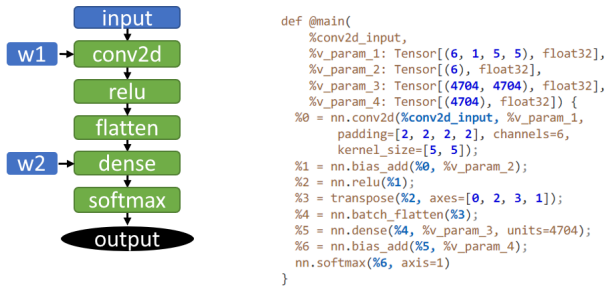
2 BACKGROUND

2.1 DL Compilers

DL compiler takes as input DL models. These models can be constructed with the help of DL frameworks, such as Tensorflow[17] and PyTorch[18]. After interpreting the input model's computational graph (will be detailed in the next subsection), the DL compiler converts it into high-level IR. Each node in the computational graph is represented by one or several IR expressions. For instance, a *conv2d* (two-dimensional convolution) node is represented by *nn.conv2d* in the high-level IR of TVM. DL compilers then optimize the computational graph at the high-level IR. For instance, a static subgraph independent of inputs can be optimized through constant folding at IR. After optimization, high-level IR is translated into low-level IR for further optimization. In this step, a high-level IR expression (e.g., `nn.conv2d`) is expanded into a nested loop of low-level computation instructions. Subsequently, low-level optimizations are performed to improve efficiency. For instance, loop tiling can be performed at low-level optimization to accelerate the computation of *conv2d* on a specified hardware device. Finally, low-level IR is translated into deployable code for diverse hardware using traditional compilers and platforms.

2.2 Computational Graph and High-level IR

A computational graph is a directed graph that expresses the data flow in computation. High-level IR, also known as graph-level IR, is an intermediate representation to express the computational graph. In DL community, high-level IR is widely used for describing computational graph by DL compilers (e.g., TVM) and also frameworks (e.g., ONNX). Figure 1a illustrates the computational



(a) Computational Graph **(b) High-level IR**

Figure 1: Computational Graph and the Corresponding High-level IR

graph of a 2-dimensional convolutional neural network, where variable/constant nodes and operator nodes are colored in blue and green, respectively. The end of a graph is denoted by a black ellipse. Arrows in this graph represent data flows. Specifically, variable nodes and constant nodes are the starting point of a data flow, passing their data to the next nodes, while operator nodes work as a relay to extend the data flow, passing the results they calculate. Figure 1b illustrates an example of TVM's high-level IR, Relay IR, of the computational graph in Figure 1a. This IR is not unique but is rewritable in multiple other ways in the same semantics by utilizing IR language features. For instance, Relay offers first-class functions to separate the main function into multiple functions with complex call chains. And ONNX also plans to support this feature in the future[1].

3 APPROACH

This section presents the design and underlying methodology of HirGen whose workflow is given in Figure 2. HirGen maintains a

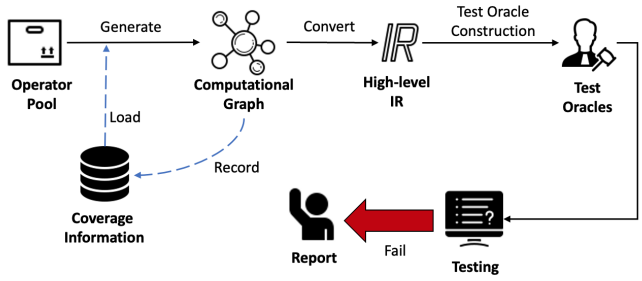


Figure 2: Workflow of HirGen

pool of 58 operators that can be expressed by high-level IRs in popular high-level frameworks (e.g. Relay [19], ONNX [20]). HirGen first loads existing coverage information, generates a computational graph based on it, and updates coverage information from the newest graph. Then, HirGen leverages an high-level framework such as Relay or ONNX, to convert the graph into a high-level IR and feeds it into the DL compiler. To capture the defects in the target DL compiler more efficiently, HirGen constructs two test oracles from the spirits of metamorphic testing and differential testing and also includes crash, a commonly-used test oracle. Any test case that violates the oracles is regarded as a witness to a bug of the compiler and will be reported to developers. The remainder of this section is divided into three parts. In Section 3.1, we will elaborate on the details of our computational graph generation algorithm. Section 3.2 will present how to utilize the language features of high-level IR and convert computational graph into high-level IR. In Section 3.3, we will introduce the design of our created test oracles.

[1]https://github.com/onnx/onnx/blob/main/docs/Operators.md

operator通常指的是操作符，也就是神经网络中的基本运算单元

3.1 Computational Graph Generation

3.1.1 Overview. We consider the generation of a computational graph as a process of continuously inserting various operator nodes into the initially empty graph $CG = \{\}$ until the number of operator nodes equals to the required number. Generally speaking, HirGen selects one operator from the operator pool, loads the operator into CG as a node nd, and constructs a connection between nd and other existing nodes. In these steps, HirGen maintains node information, including data type, tensor shape, and connection information, for each node. To improve the diversity of the graph, HirGen also involves three coverage criteria and tries to improve the coverage in each insertion of node.

With these prerequisites, HirGen provides two generation modes. To generate valid computational graphs, HirGen utilizes the above-mentioned node information for strict type checking and shape checking. In this way, each insertion is valid and breaks no constraint. We call this mode *strict generation*. On the other hand, strictly following the constraints may miss the opportunity to test the exception handling ability of DL compilers when constraints are violated. Therefore, HirGen also provides *disruptive generation* to deliberately break type constraints and shape constraints. We will first elaborate strict generation and then disruptive generation.

3.1.2 Node Information. Inserting a node into CG requires node information of all existing nodes to perform type-checking and shape-checking. Node information describes the typical features of node, and such information is essential to promise the correctness of each insertion. For instance, in the insertion of an operator named add that sums two nodes, HirGen first checks all available nodes (including operator nodes, variable nodes and constant nodes) and selects two nodes n_a and n_b from available nodes, such that n_a and n_b have the compatible tensor shapes and data types, and their data types are acceptable by the operator add. Each type of node has its own node information, as detailed in Table 1.

Node Type	Node Information
variable	*dataType*, *tensorShape*
constant	*dataType*, *tensorShape*, *value*
operator	*dataType*, *parentNodes*, *tensorShape* = INFERENCE(*parentNodes*)

Table 1: Node Information

Specifically, HirGen considers the following three types of nodes, namely, *variable*, *constant* and *operator*.

(1) *Variable* node. It involves data type *dataType* and tensor shape *tensorShape* describing the details of the tensor wrapped in this node. *dataType* corresponds to the data type of all elements in this tensor, such as *int*64 and *float*32. *tensorShape* is a vector of the scale of all dimensions in the tensor.

(2) *Constant* node. Besides the $dataType$ and $tensorShape$, *constant* node includes the value of tensor $value$ as a part of its information.
(3) *Operator* node. Operators require parameter(s) and thus they are all connected with other nodes in the graph. To document this connection information for each operator node, Besides $dataType$, HirGen records its parent node(s) $parentNodes$ to which this node connects and records its tensor shape inferred from parent node(s).

3.1.3 Coverage Guidance. To explore diverse data types, shapes, operators in computational graph generation, we design three coverage criteria.

(1) **Operator-datatype Coverage**. Let op_i be the i_{th} operator in the operator pool. Let $dtype_j$ be the j_{th} data type in the collection of data types. Let $Cov(op_i, dtype_j)$ be 1 when op_i has once been inserted into the graph as a node with data type $dtype_j$. Otherwise, it is 0.
(2) **Operator-shape Coverage**. Let op_i be the i_{th} operator in the operator pool. Let $shape$ be the shape of the output tensor of this operator node after being inserted into the graph. Let $Cov(op_i, shape)$ be 1 if op_i has once been inserted into the graph as a node with tensor shape $shape$, and 0 otherwise.
(3) **Operator-edge Coverage**. Let op_i and op_j be the i_{th} and j_{th} operator in the operator pool. Let $Cov(op_i, op_j)$ be 1 if there exists one edge from op_i to op_j, and 0 otherwise.

The design of the first two coverage criteria are motivated by the fact that type problem and shape problem are the two major root causes of DL compiler bugs [8]. The design of the third one tries to complicate the data flow of the computational graph since the third coverage encourages HirGen to interleave different operators in a computational graph. Specifically, with operator-datatype coverage, HirGen is encouraged to 1) involve different operators in the graph and 2) utilize diverse data types since the data type problem is a big concern for DL compilers [8]. With operator-shape coverage, HirGen is encouraged to try various calculations with diverse tensor shapes and thus increase the probability of encountering calculation problem, such as the poor implementation of some operators in special shapes or different calculation results on different platforms. With operator-edge coverage, HirGen is guided to connect the new operator node to the existing operator nodes instead of variable nodes and constant nodes. In this way, the generated computational graph contains more complex and deep data flow instead of parallel connection of several simple data flows. In implementation, we can easily extend operator-edge coverage to operator-path coverage with 3 or more operator nodes included. In this way, we can explore a more diverse and complicated computational graph, but at the cost of greater time costs. With these three coverage criteria, HirGen is prevented from producing previously generated subgraphs in computational graph generation. Since high-level optimizations often involve identifying, annotating, re-constructing, and shrinking optimizable subgraphs, duplicate subgraphs can only find duplicate bugs. For instance, *operator fusion* is a high-level optimization that can fuse operators in a high-level expression into a larger operator. Fusing different operators could encounter different situations and cover different codes. With coverage criteria, a group of operators will unlikely appear in a high-level expression in the previous order. Therefore, these three coverage criteria facilitate HirGen to find new bugs more efficiently. Bug_2 in Table 2 was detected by HirGen in three days but was never detected by HirGen_r (HirGen without coverage guidance) in our 2-week experiment.

3.1.4 Constraint-Awareness Graph Generation. Algorithm 1 presents two procedures used by HirGen to generate computational graphs that strictly follow the type and shape constraints of IR. GENERATION is the main procedure. It takes as input the required number of operators $rOpNum$ to be contained in the output computational graph. PREINSERT is an auxiliary procedure that enforces type-checking and shape-checking in generation. GENERATION procedure includes the following two main parts.

Algorithm 1 Computational Graph Generation

```
1:  procedure Generation(rOpNum)
2:      CG ← {}
3:      opnum ← 0
4:      opPool ← {add, subtract, multiply, divide, ...}
5:      dataTypeSet ← {int64, int32, int16, int8, uint64, uint32,
6:                                  uint16, uint8, float64, float32, bool}
7:      repeat
8:          opNode ← select(opPool)
9:          dataType ← select(dataTypeSet)
10:         connection, shape, CG ← preInsert(opNode, dataType, CG)
11:         coverage ← CalculateCoverage(opNode, dataType, connection, shape)
12:         if newCoverage(coverage) then
13:             opnum ← opnum + 1
14:             UpdateCoverage(coverage)
15:             opNode.info ← (connection, shape, dataType)
16:             CG ← CG ⋃ {node}
17:         end if
18:     until opnum = rOpNum
19:     return CG
20: end procedure
21: procedure preInsert(opNode, dataType, CG)
22:     availableNodes ← typeCheck(CG, dataType)
23:     nodeGroup1, nodeGroup2, ... ← shapeCheck(availableNodes)
24:     paramNodes ← select (nodeGroup1, nodeGroup2, ...)
25:     if nodeNotEnough(paramNodes) then
26:         node1, node2, ... ← create(dataType, paramNodes)
27:         CG ← CG ⋃ {node1, node2, ...}
28:         paramNodes ← paramNodes ⋃ {node1, node2, ...}
29:     end if
30:     for node in paramNodes do
31:         connection ← (opNode, node)
32:     end for
33:     shape ← Inference(connection)
34:     return connection, shape, CG
35: end procedure
```

Initialization. HirGen performs initialization from Line 2 to Line 6. Specifically, computational graph CG is initialized as empty and the number of operators $opnum$ in CG is set to 0. Operator Pool and data type set are both initialized for future use.

Generation Loop. In each iteration (Lines 7-18), HirGen generates an operator node, updates its node information and finally inserts it into CG if new coverage is explored. Specifically, HirGen first randomly selects an operator and data type (Line 8, 9). Then it seeks for connection from CG and infers the tensor shape of the operator node $opNode$ built from the newly selected operator (Line 10). Subsequently, HirGen calculates coverage and performs update and insertion if new coverage is explored (Line 11-17). The exploration of new coverage is detected by any increment of the three coverages defined in section 3.1.3. During update, HirGen adds $opnum$ by 1, updates coverage and node information of $opNode$. The generation loop stops when $opnum$ equals $rOpNum$ and HirGen returns CG eventually.

Procedure preInsert shows the details of building connection between *opNode* and existing nodes of *CG* and shape inference. In type-checking, HirGen absorbs type constraints in the target DL compiler and use them to avoid type mismatch (e.g., float32 and int64 for *add* operator). In shape-checking, HirGen includes shape rules in the target DL compiler, such as broadcasting rule, which specifies that if the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with the ones on its leading side. Shape-checking avoids shape mismatch. With type-checking (Line 22) and shape-checking (Line 23), HirGen sorts out several node groups from *CG*. All nodes of each node groups are mutually shape-compatible and nodes from these groups are all type-compatible with *opNode*. Then HirGen selects a node group and dumps all its nodes into *paramNodes* (Line 24). The number of required parameter nodes is fixed for each kind of operator. In implementation, HirGen has a certain probability of connecting one node to an operator node for multiple times, such as connecting one variable node to add node for twice, meaning add the node to itself. But to complicate data flow, HirGen discourages this behavior in favor of connecting the required number of different nodes. Therefore, if the number of parameter nodes in *paramNodes* is insufficient for *opNode*, HirGen has large probability in creating variable nodes or constant nodes that are shape-compatible with all parameter nodes and type-compatible with *opNode*, inserts them into *CG* and updates *paramNode* (Line 25-29). Finally, HirGen creates connection information (Line 30-32), infers the tensor shape of *opNode* and returns them both plus the possibly updated *CG*.

3.1.5 Disruptive Generation Algorithm. Disruptive generation is similar to constraint-awareness generation. It also needs coverage to memorize what type constraints and shape constraints have been broken. In addition, node information is also required. Since it contains data type and tensor shape of each node, which is necessary for breaking constraints. Specifically, during disruptive generation, HirGen purposely 1) connects operator node to other node(s) with the data type(s) it can not accept in TVM (e.g., `add` operator with `bool` data type), and 2) connects nodes that are type-incompatible or shape-incompatible (e.g., add two nodes of which the shapes are `[3,4]` and `[2,3]` respectively).

3.2 High-level IR Generation

High-level IR generation is simple with the help of existing high-level frameworks, such as Relay and ONNX. Taking Relay as example, it provides ample APIs for receiving node information of various types of nodes and diverse operator nodes. For instance, `relay.var` takes as inputs its name, data type and tensor shape, and `relay.add` takes as input only its connection information. These APIs contain strict type constraints and shape constraints, and it is easy to crash early before optimization if the computational graph contains an error.

Besides the plain conversion by loading each node into its corresponding high-level expressions and assembling them into a high-level IR, we can also utilize the primitive features of these high-level frameworks. Take Relay for example, to improve expressivity, it allows using a function to wrap a subgraph and call the function in other ones. ONNX also plans to support this feature by supporting Function API. To better utilize these features, we also consider extracting a subgraph from the generated computational graph and wrap it with a high-level function. In this way, we can better test how DL compilers tackle the situation where functions are included.

Algorithm 2 Conversion from Computational Graph to High-level IR

```
1:  procedure Conversion(CG)
2:      Functions ← {}
3:      Expressions ← {}
4:      for node in CG do
5:          Expressions ← load(node.info, Functions, Expressions)
6:          if roll() == func then
7:              inputNodes, outputNodes ← analyze(Expressions)
8:              function ← composeFunction (inputNodes, outputNodes)
9:              Functions ← Functions ⋃ function
10:             Expressions ← {}
11:         end if
12:     end for
13:     return Expressions ⋃ Functions
14: end procedure
15: procedure load(node.info, Functions, Expressions)
16:     expression ← ConstructExpression (node.info)
17:     if parentInFunction(node.info) then
18:         functions ← find(node.info)
19:         callExprs ← createCallExpression(functions)
20:         Expressions ← Expressions ⋃ {callExprs}
21:     end if
22:     Expressions ← Expressions ⋃ {expression}
23:     return Expressions
24: end procedure
```

The overall algorithm of converting computational graph into high-level IR is shown in Algorithm 2. Conversion procedure takes as input computational graph *CG* and outputs its corresponding high-level IR. During initialization, HirGen creates two empty sets, named *Functions* and *Expressions* respectively (Line 2, 3). They represent the collection of functions and high-level expressions, respectively. In the for loop (Line 4-12), HirGen traverses all nodes in *CG*, loads each node into high-level expression and update *Expressions* (Line 5). It randomly selects a set of high-level expressions and wraps them with a function (Line 6-11). To compose a function, HirGen first analyzes the input nodes and output nodes of the underlying subgraph of the expressions (Line 7). It composes a function using these nodes (Line 8. Finally, HirGen updates *Functions* and *Expressions* (Line 9-10). Conversion procedure returns the union of *Expressions* and *Functions* as High-level IR. load procedure presents the detail of loading a node into high-level expression. During loading, HirGen takes care of connection information by inquiring whether the node connects to other nodes wrapped in function(s) (Line 17), if it is the case, then a call expression is created (Line 19). This procedure returns *Expressions* after update.

3.3 Test Oracles

Test oracles determine if a test passes or fails. In this paper, we consider three test oracles to detect different types of failures. Any failed test case determined by these oracles will be reported.

3.3.1 $Oracle_1$: Crash. Crash is widely used in test oracle construction to decide whether the testing fails [11]. Besides, according to the statistics in a compiler bug study [8], the number of bugs with crash symptom occupy 59.37% of all collected 603 bugs. This huge proportion shows a urgent need to take crash seriously. As for crash bugs detected when type-checking and shape-checking

are turned off, we only report the bug if the crash is a segmentation fault because other crashes with detailed bug trace is largely due to explicit violation of constraints in the computational graph. As for other crash bugs, we report them all since the generated computational graph under checking strictly follows all constraints in TVM and the crash is largely due to the poor implementation of TVM.

3.3.2 $Oracle_2$: Result Inconsistency among original high-level IR, optimized high-level IR and mutated high-level IR. Intuitively, high-level optimization is only related to performance boost such as calculation acceleration and memory cost saving, but can not change results. In addition to involving high-level optimization, we also design a mutation strategy named function rewrite to generate mutated high-level IRs who has the same output as the original high-level IR given the same input. This mutation strategy is inspired by Relay's support for functional programming features. By function rewriting, we can better utilize Relay's expressions and better test TVM with richer high-level IRs. Specifically, this mutation strategy can rewrite function expressions in high-level IR in the following ways.

- Turn a global function f into the local closure of another newly created global function g. g has the same parameters as f and its returned value is a call to f with these parameters. After this tuning, this mutation also substitute all calls to f with calls to g.
- Wrap a function f with an empty function g which returns f and also change all calls to f to calls to the call to g.
- Call a function f and return the call in another function g, then substitute all calls to f with calls to g.

The mutated high-level IR only differs from the original high-level IR in the function call chain. Therefore, it is sound to expect the same calculation results among these three high-level IRs given the same input. This metamorphic relation inspires us to form this oracle. In addition to different calculation results, if the original high-level IR passes compilation and runtime but the optimized one or mutated one fails in one of these two processes, we also count it as result inconsistency.

3.3.3 $Oracle_3$: Result Inconsistency across hardware devices. To maintain the same predictive capability of a DL model on different supported hardware devices, TVM should promise to output the same results on diverse hardware given the same input to a DL model. And similar to $Oracle_2$, inconsistent execution status (e.g., crash on CPU but execute well on GPU) is also counted as result inconsistency. Following this common sense, we build $Oracle_3$ with the spirit of different testing. Given any high-level IR, after compiling it with multiple provided compilation approaches, feeding an input and executing it on CPU and GPU, it is reasonable to expect the same calculation results.

4 EXPERIMENT SETUP

4.1 Research Questions

In this study, we aim to address the following research questions:

RQ1 How effective is HirGen in detecting bugs of TVM?
RQ2 Are all the test oracles effective in detecting bugs?
RQ3 Are bugs found by HirGen highly related to high-level optimization?
RQ4 Is disruptive generation useful in finding exception handling bugs?
RQ5 Can coverage-guided generation benefit the diversity of graph?

4.2 HirGen Implementation

We implement HirGen in C++ with around 3K lines of code. Our implementation involves 58 operators[2] to generate computational graph, 25 high-level optimizations for catch optimization bugs and four compilation methods to conduct testing.

4.2.1 Operators. In total, HirGen includes 58 operators supported by TVM, including 23 binary operators and 35 unary operators. And it is easy to extend HirGen with other operators.

4.2.2 Optimization and Compilation Methods. We select in total 25 high-level optimizations supported by TVM[3] The main reason for choosing these high-level optimizations in TVM is our generated computational graph can trigger them. Besides collecting these high-level optimizations, we also utilize different compilation methods provided by TVM. Different compilation methods deal with different scenarios and include different optimization sequences. Overall, we include `relay.build()`, `relay.build_module.create_executor('debug')`, `relay.build_module.create_executor('graph')` and `relay.build_module.create_executor('vm')` in HirGen. Besides these high-level optimizations, it is easy to extend HirGen with other optimizations.

4.3 Bug Report

For each bug we have found, we report it in one of the three channels: 1) upload the bug-triggered script and experiment environment on TVM Community[4]; 2) report the bug on Github Issue[5] with a reproducible script, experimental environment, and most importantly, our analysis on the reason for triggering it; 3) create a pull request with the elaboration of this bug and our code patch. We choose our reporting channels primarily based on our expertise of the problem. For the least familiar bug, we submit it on TVM Community in the form of the question to get rid of misdiagnosis. Then, we wait for an official fix or some comments from developers on this problem. For the most familiar one, we directly fix it, and we succeed in creating two pull requests and fixing two bugs. For other situations, we choose the second way and leave some comments on how to fix the bug.

4.4 Baseline Selection

4.4.1 TVMfuzz. TVMfuzz is a preliminary proof-of-concept application for fuzzing TVM[8]. It can learn TVM API call chains from unit test scripts, then re-order and mutate them. By learning from high-level IR and optimization related unit test scripts, TVMfuzz can cover this stage.

4.4.2 MT-DLComp. MT-DLComp is an automated testing framework for DL compilers[13]. It mutates existing DL models to generate equivalent models and test DL compilers by three oracles.

[2]https://github.com/anonymousWork000/HirGen/blob/master/README.md
[3]https://github.com/anonymousWork000/HirGen/blob/experiment/optimizations
[4]https://discuss.tvm.apache.org/
[5]https://github.com/apache/tvm/issues

Though this technique is not specially created for detecting bugs in high-level optimization, it can cover this bug-prone stage. Therefore, we also include it as a baseline.

4.4.3 LEMON. LEMON is a testing technique for deep learning frameworks[21]. It generates Keras [22] models by mutating existing models. By setting different backends of Keras, LEMON detects prediction difference incurred by these backends. Though LEMON is not for testing DL compiler, we can retrofit it to barely achieve the goal. In short, we remain the mutation part to generate new models and test DL compilers by two test oracles: 1) crash, and 2) above-threshold prediction difference between original Keras models and compiled Keras models.

4.4.4 NNSmith. NNSmith is a generation-based fuzzer for DL compilers [14]. During generation, it generates diverse computational graphs, converts them into DL models using different DL frameworks, and uses gradient-guided search to generate inputs. During testing, it conducts differential testing among several DL compilers. In the testing process, NNSmith captures all prediction difference and crashes.

4.5 Metrics

We mainly target *bug counting* for evaluation. To evaluate HirGen, we count bugs based on independent fixes and developers' confirmation in section 5.1.1. In section 5.1.2 and 5.5, we studied five baselines in total and obtain a number of crashes/inconsistencies. Since many of them are duplicate, reporting them to TVM community for bug confirmation can be time-consuming and possibly receives no reply according to our interaction with the developers. So we use the proximity of bug counting in the experiment of these two sections. In particular, manually-deduplicated bugs in section 5.1.1 have totally different stack traces, and thus comparing the number of crashes/inconsistencies with distinct stack traces is reasonable. The proximity of bug counting is also used in the evaluation of other works [23, 24].

4.6 Timeout Setting

There are two comparison experiments involving timeout. The first is comparing HirGen with the four baselines. The second is comparing HirGen with HirGen$_r$. We executed each of the involved techniques separately for two days, and each execution was conducted ten times to mitigate the influence of randomness. Since all the studied techniques do not find distinct crashes/inconsistencies after 26 hours, it indicates that our 2-day timeout is reasonable to a large extent.

4.7 Experiment Environment

We conducted experiments on a server with Intel(R) Xeon(R) CPU, NVIDIA GeForce GTX1080Ti GPU, and 128 RAM, coordinated with 64-bit Ubuntu 16.04 OS.

5 EVALUATION

5.1 RQ1: Bug Detection Capability of HirGen

5.1.1 Summary of Detected Bugs by HirGen. Running three months under strict mode and one week under disruptive mode, HirGen has found 21 bugs, of which 17 have been confirmed. Among these confirmed bugs, 12 have been fixed, 10 were previously unknown bugs and 5 fixed bugs were previously unknown. Table 2 presents the details about all the confirmed bugs discovered by HirGen, including their symptoms, root causes, the test oracles detecting them, the fixing status, whether they are previously unknown, by which generation mode were they detected, were they also found by other techniques (blanks mean no other techniques detected the bugs in experiments) and whether they are high-level optimization bugs. Symptom includes crash and inconsistency. The former means that TVM terminates unexpectedly while the latter means that different results or statuses are caught in testing. We also manually investigate the root cause of each bug adopting the taxonomy of a recent bug study [8]. Specifically, We carefully compare these bugs with the collected historical bugs and assign each of them with a root cause.

The root causes of these bugs are divided into the following classes:

- Type Problem. This category of bugs is triggered by data type related problems, including incorrect type inference, incomplete implementation of an operator on one data type, etc.
- Incorrect Exception Handling. This category of bugs occurs when TVM lacks rich and readable warning messages or even has no handling of some extreme situations. This kind of bugs are related to the robustness of TVM.
- Incorrect Numerical Computation. This root cause involves incorrect numerical computations, values, or usages.
- Internal API Incompatibility. This category of bugs is triggered because TVM can not handle the combination of some APIs correctly. For instance, unexpected refuse of one combination of several high-level optimizations is counted as this kind of bug.
- Memory Allocation Problem. This root cause refers to the poor or incorrect memory allocation.

Check mark in *Previously Unknown* column in Table 2 means that the corresponding bug was unknown before we reported it. Since we tested TVM v0.9 (commit id: 124813f) at the beginning of our experiment and TVM was evolving fast, we found some cases early in the experiment that crashed on the version we tested but worked fine on the latest version. These bugs have been actually fixed before being reported and thus marked as previously known bugs.

5.1.2 Comparison with State-of-the-art Techniques. On average, HirGen detected 11.8 distinct crashes/inconsistencies. The variance of the number of them in the 10 repeated experiments is 0.36. We also conducted a manual inspection on these crashes/inconsistencies by two experienced researchers. We observed that the average number of crashes/inconsistencies related to high-level optimization is 8.8, and the variance of the number is 0.36. TVMfuzz detected 3.7 distinct crashes on average, of which 1.4 crashes are related to high-level optimizations. By Mann-Whitney U Test [25], p-value of the difference between HirGen and TVMfuzz is $0.00018 < 0.01$, which implies the result that HirGen outperforms TVMfuzz in DL compiler bug detection has statistical significance. MT-DLComp and LEMON detected nothing. As for NNSmith, it detected 10 distinct crashes/inconsistencies on average. Among these crashes/inconsistencies, data layout problems and data type problems are predominant, altogether accounting for 52.2% of all

Table 2: Confirmed Bugs found by HirGen

Bug ID	Symptom	Root Cause	Test Oracle	Status	Previous Unknown	Generation Mode	Found By	High-level Optimization
1	Crash	Incorrect Numerical Computation	$Oracle_1$	Fixed	✓	strict	NNSmith	
2	Inconsistency	Incorrect Exception Handling	$Oracle_2$	Confirmed	✓	strict		✓
3	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed	✓	strict		✓
4	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed		strict		
5	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed	✓	strict		
6	Inconsistency	Incorrect Exception Handling	$Oracle_2$	Fixed	✓	strict		✓
7	Inconsistency	Type Problem	$Oracle_2$	Fixed		strict		✓
8	Inconsistency	Type Problem	$Oracle_2$	Fixed		strict		✓
9	Inconsistency	Internal API Incompatibility	$Oracle_2$	Fixed	✓	strict		✓
10	Inconsistency	Incorrect Exception Handling	$Oracle_2$	Fixed		strict		✓
11	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed		disruptive		✓
12	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed		disruptive		✓
13	Crash	Incorrect Exception Handling	$Oracle_1$	Fixed		disruptive		✓
14	Crash	Memory Allocation Problem	$Oracle_1$	Confirmed	✓	strict		✓
15	Inconsistency	Incorrect Numerical Computation	$Oracle_3$	Confirmed	✓	strict		
16	Inconsistency	Incorrect Numerical Computation	$Oracle_3$	Confirmed	✓	strict		
17	Inconsistency	Incorrect Numerical Computation	$Oracle_3$	Confirmed	✓	strict		

crashes/inconsistencies. They are captured with bug messages such as *"WCHN layout is not supported"* or *"TVM cannot support type matching between int32 and int64"*. Among other crashes/inconsistencies, on average 3.5 crashes/inconsistencies are related to high-level optimization, and the variance is 1.45, showing that NNSmith is unstable in detecting high-level optimization bugs. The p-value of the high-level optimization crashes detection difference between HirGen and NNSmith is also $0.00018 < 0.01$, implying the result that HirGen outperforms NNSmith in high-level optimization bug detection has statistical significance. During the manual inspection, we only found one overlapping crash detected by both HirGen and NNSmith, showing that these two techniques have almost complementary bug detection ability. We will discuss the reason in section 7.2.

5.2 RQ2: Effectiveness of Test Oracles

To demonstrate the effectiveness of our test oracles, we conduct a case study of several representative confirmed bugs detected by each test oracle.

$Oracle_1$: Crash. $Oracle_1$ caught the most bugs among all test oracles. In total, it finds eight bugs of three root causes, including *Incorrect Numerical Computation*, *Incorrect Exception Handling*, *Memory Allocation Problem*.

Incorrect Numerical Computation. Take Bug_1 as an example. In the computational graph to trigger this bug, a `divide` operator first calculates the result of dividing a constant by a variable and then passes the calculation result R to `floor_mod` as dividend. All involved variable nodes and constant nodes are of data type *uint* and this type finally flows into `floor_mod`. However, TVM precalculates the possible value range of R and detects it could probably be 0. Therefore, TVM incorrectly throws an exception and terminates even before we give values to $var1$ and $var2$, This bug only happens when the data type is *uint* and is caused by incorrect value range estimation. After developers confirmed this bug and fixed `const_int_bound` analyzer, this numerical computation related bug was fixed.

Incorrect Exception Handling. Bug_{11}, Bug_{12} and Bug_{13} are three bugs of Incorrect Exception Handling. They are detected under disruptive generation. To trigger these bugs, HirGen must generate computational graphs containing obvious breaks of constraints. Take Bug_{11} for example, its corresponding computational graph includes a constant node of type $int16$, a `tan` operator node and the connection between these two nodes. The constant node passes its $int16$ data to the operator node. In this tiny subgraph, HirGen purposely breaks the constraint that `tan` only accepts $float$ data type defined in TVM and receives a segmentation fault during compilation. This is because TVM does not have exception handling for this operator and its unacceptable data types.

Memory Allocation Problem. Bug_{14} is the only bug of root cause, named Memory Allocation Problem. Specifically, when HirGen leverages `relay.shape_of` to infer to tensor shape of variable node with static tensor shape $(1, 2)$, an unexpected crash happens with warning message `Cannot allocate memory symbolic tensor shape [?, ?]`.

$Oracle_2$: Result Inconsistency among original high-level IR, optimized high-level IR and mutated high-level IR. $Oracle_2$ caught a total of six confirmed bugs, and five of them have been fixed. These bugs are caused by three different root causes, including *Incorrect Exception Handling*, *Type Problem* and *Internal API Incompatibility*.

Incorrect Exception Handling. Take Bug_{10} as an example HirGen catches this bug because it finds a high-level IR passes compilation while its optimized version fails. Specifically, HirGen places `FirstOrderGradient` before `FuseOps` in a optimization sequence and detects that TVM cannot successfully handle this optimization sequence. This is because exception handling is too strict. Concretely, TVM performs a traversal on the high-level IR after `FirstOrderGradient` for conducting `FuseOps`. When visiting a constant node, TVM finds this node is not scalar because

`FirstOrderGradient` has rewritten this attribute. Therefore, TVM throws an exception and the compilation terminates. However, this check about scalar attribute is too strict and does not consider data type. A fix for this bug completes this exception handling and makes the optimized version successfully passes compilation. Besides, Bug_6 is also a representative, detected by our effort in utilizing IR language features. HirGen takes advantage of first-ciziten functions in Relay IR and tries to return a function in another function. Since TVM v0.9 cannot well support the lowering of this high-level language feature into low-level counterpart, segmentation fault is thrown. The effort in utilizing high-level IR's language features also help us find Bug_5, Bug_7 and Bug_8.

Type Problem. Take Bug_8 for instance. This bug is found with function rewrite mutation. Specifically, after changing a global function f into the local closure of another empty global function g and return f in g, TVM can not infer the type of g. This is because after successfully inferring the type of f, this type information is lost when TVM begins to infer the type of g.

Internal API Incompatibility. Take Bug_9 for example. This bug is detected because `relay.build_module.create_executor('vm')` fails, but compilation in other ways run smoothly. Specifically, after HirGen transforms high-level IR into a A Norm Form. Compilation with virtual machine cannot figure out the bound relation between x_{91} and a global function. However, other compilation ways do not encounter this problem.

$Oracle_3$: Result Inconsistency across hardware devices. $Oracle_3$ caught a total of three confirmed bugs, but none of them has been fixed. This is because difference among computation results on CPU and GPU is caused by platform specific differences. More specifically, LLVM and CUDA has different implementations on the same operator, while TVM lacks full specification about this operator or lacks complete warning message of using this operator. Developers responded with a confirmation of this deficiency but they consider it unnecessary to remedy it without it violating the effectiveness of TVM seriously.

Take Bug_{15} as an example. HirGen creates a simple computational graph containing a `right_shift` operator node. This operator node takes as input two other variable nodes. Subsequently, HirGen first generates the corresponding high-level IR, then compiles the IR with `relay.build` to generate runtime model and finally creates the input and run runtime model on CPU and GPU to get two computation results. When the second variable is larger than the first one, results are inconsistent. This is because this situation incurs a poison value in LLVM and the use of it in an operator is undefined. Though this confirmed bug does not come from poor implementation of TVM but from problems in external compiler, it is still confusing to users when their DL model triggers this inconsistency. The refinement of the exception handling system could be a compromise approach for this ill situation.

5.3 RQ3: Relation between our found bugs and high-level Optimization.

As a DL compiler fuzzer focusing on high-level optimization, HirGen is capable of detecting bugs in high-level optimization or bugs highly related to this stage. In this subsection, we manually study the code patch of each fixed bug detected by HirGen and analyze their relationship with high-level optimization and how the detection of them improve this stage.

Bug_2, Bug_8, Bug_9, Bug_{10} are bugs detected in high-level optimization. Bug-triggered pattern for these four bugs are similar: after high-level optimizations, HirGen detects a violation of $Oracle_2$. These bugs show inability to optimize the structure several high-level optimizations should have optimized, and incompatibility among several optimizations. For instance, Bug_8 shows that after performing `InferType` on one function, the solved types cannot be passed to the next function and thus triggers a type problem. Bug_{10} shows `FuseOp` can not be well performed after performing `FirstOrderGradient`. Fix of these bugs directly improves performance of the optimization and facilitates the possibility of multiple optimization combinations.

Besides, HirGen finds eight bugs with crash symptom and all of them trigger crash during compilation. Among them, Bug_3, Bug_{14} are directly related to high-level optimization. To improve efficiency, TVM calls `OptimizeImpl` during compilation and invokes 11 high-level optimization implicitly. These optimizations work by one or several passes on the high-level IR, which performs rewrite at any optimizable expression. In each pass, all expressions in high-level IR are visited and assertions embedded in TVM check each expression. Bugs in this process may prevent high-level optimizations from being well executed, or even result in a crash to stop the optimization. Fixes for these bugs are indirect fixes for required IR passes needed by high-level optimizations. Besides, Bug_{11}, Bug_{12}, and Bug_{13} are bugs in high-level IR construction. Since construction happens before optimization, these bugs also prevent high-level optimizations.

Although our approach is proposed for high-level optimization, the test cases generated by our approach can also execute the low-level optimization and deployable code generation. Thus, it has the side effect of testing the other stages. And the results also confirm it. Bug_{15}, Bug_{16} and Bug_{17} are all related to low-level part and backend of TVM. They are detected due to inconsistent calculation results on different backends (LLVM and CUDA) given the same inputs. These bugs show the need to couple TVM with these backends better. Bug_1 and Bug_5 are arithmetic problems at low-level. HirGen can detect them because the generated computational graphs contain error-triggered computational logic.

5.4 RQ4: Contribution of Disruptive Generation

During experiments, we have generated 170 computational graphs with different bug-triggering combinations of operator, data type and tensor shape. All these graphs can incur crash of TVM with only "segmentation fault" information, showing the deficiency of exception handling ability. In the latest TVM version, all these bugs have been fixed. All these obvious breaks of constraints trigger crash with detailed bug information now. By comparing the bug information of the latest TVM, we found there are three bugs in total found by these 170 graphs.

5.5 RQ5: Contribution of Coverage-guided generation to The Diversity of Graph

To generate diverse computational graphs with various data types, tensor shapes and operators, we design three coverage criteria. To answer this RQ, We implement a simplified version of HirGen, saying HirGen$_r$. HirGen$_r$ is identical to HirGen except that HirGen$_r$ is not guided to generate computational graphs but selects operators, shapes, and types under the rule that all selections are valid, and connects the new operator to the existing operator(s) randomly under the rule that the connection is valid. We conducted two experiments for comparison between these two techniques.

The first experiment is about the bug detection ability of HirGen and HirGen$_r$ under strict generation mode. In each round of the experiment, we executed these two techniques for two days independently. To mitigate the influence of randomness, our experiment includes ten rounds. On average, HirGen found 11.8 distinct crashes/inconsistencies. Among them, 8.8 crashes/inconsistencies are related to high-level optimization. The variance of the number of distinct crashes/inconsistencies and that of the number of distinct crashes/inconsistencies related to high-level optimizations are both 0.36. As for HirGen$_r$, it found 8.9 distinct crashes/inconsistencies. 6.7 crashes/inconsistencies are related to high-level optimizations. The variance of the number of distinct crashes/inconsistencies is 3.16 and the variance of the number of distinct crashes/inconsistencies related to high-level optimizations is 1.12. By Mann-Whitney U Test, the p-value of HirGen outperforming HirGen$_r$ in detecting distinct crashes/inconsistencies related to high-level optimizations is $0.0028 < 0.01$, and the $p - value$ of HirGen outperforming HirGen$_r$ in detecting distinct crashes/inconsistencies is $0.00512 < 0.01$, implying the result that HirGen outperforms HirGen$_r$ in detecting high-level optimization bugs has statistical significance. Besides, HirGen$_r$ has a bigger variance, showing that it's unstable in bug detection.

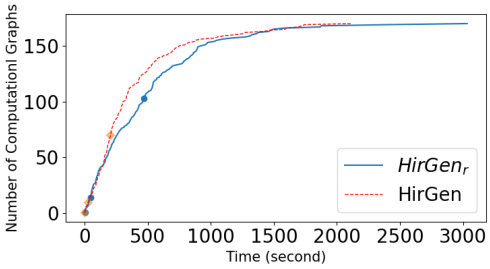


Figure 3: HirGen *vs.* HirGen$_r$ under Disruptive Generation

The second experiment is similar to the first one. In this experiment, we compare the bug detection ability of HirGen and HirGen$_r$ under the disruptive generation mode. Since disruptive generation promises that each insertion contains a violation of constraints and must trigger failure, there is no need to generate multiple-operator graph. So we let them generate one-node graphs. Figure 3 presents the experiment results. HirGen and HirGen$_r$ both generated 170 bug-triggered computational graphs, each of which contains unique tuples of (*operator*, *tensor shape*, *data type*). In this figure, HirGen shows more exploratory nature in the diversity of graphs and thus detects bugs faster. Further, two techniques both found 3 bugs using these 170 graphs. And the timestamps of bug detection are also marked in this figure, showing that HirGen found bugs faster than HirGen$_r$.

6 DISCUSSION

6.1 Threats to Validity

The threat to *internal* validity mainly lies in the implementation of HirGen. To reduce this threat, two authors of this paper have carefully checked and tested the functionality of all components of HirGen.

The threat to *external* validity mainly lies in the DL compiler we chose in our study. Until now, TVM is one of the most popular and active open-source DL compilers, with 8K stars on Github. Though HirGen now mainly supports converting its generated computational graph into high-level IR of TVM with Relay. The technical approach of it is also useful for testing other DL compilers with the help of ONNX[20]. ONNX is an open format to represent diverse DL models defined by various DL frameworks and is currently supported by popular DL compilers. Similar to Relay, we can use ONNX's APIs to easily convert a computational graph into a high-level IR of ONNX. This IR is transformable to high-level IRs of existing DL compilers. And more support for ONNX to test more DL compilers is also our future work.

The threat to *construct* validity mainly lies in randomness and settings. In computational graph generation, though with coverage guidance, the selection of operator and connection also involves randomness. To alleviate the negative impact of randomness, we 1) repeated all randomness-involved experiments 10 times and utilize average, variance and Mann-Whitney U Test to promise the results are statistically significant. The threshold setting for comparing different prediction results in $Oracle_2$ and $Oracle_3$ (mainly $Oracle_3$) is still an open problem in DL compiler testing. One existing work [13] has shown that different floating-point precision settings in different platforms may lead to false positives in bug detection. Therefore, in comparing prediction results, threshold setting is vital in reducing false positives. Since no systematic study on how to set threshold exists, our settings are mainly based on experience and expertise in testing TVM. Since our test oracles did not frequently detect prediction differences, we set the threshold to a tiny floating number, 10^{-3}, to not miss any minor difference, and thus not miss any new bug. In experiment, HirGen did not find false positives due to floating-point roundoff.

6.2 Limitation and Benefit of $Oracle_3$

In our experiment, $Oracle_3$ is less effective than others. It only detected three confirmed but not fixed bugs, implying that 1) it's not as efficient as other oracles in bug detection, and 2) fix of these bugs is of low priority from the perspective of developers. The reason for this phenomenon is that the floating-point precision settings differ across platforms [13]. And it is hard for developers to tell whether the differences are caused by bugs or inconsistent precision settings. Therefore, they are reluctant in checking code or refining the exception-handling module to warn users of the large differences in calculation results among platforms when they use some special operators. Despite this limitation, $Oracle_3$ could still help find confusing scenarios and provide experience for users

who do not have enough knowledge of special error-triggering operators in low-level platforms. Take Bug_{15} as an example. If users of TVM include `right_shift` in computational graph construction, then LLVM may skew the results. Since TVM offers no warning, this phenomenon is confusing. We posted such findings[6] online and obtained such a response from a developer: *There is not a full specification for right_shift intrin in TVM*. Therefore, we counted it as a TVM bug of incomplete documentation and exception handling module.

7 RELATED WORK

7.1 Generation-based Fuzzing

Generation-based fuzzing is a class of common fuzzing techniques [23, 26–29]. Different from mutation-based fuzzing that mutates existing seed inputs to create new test inputs, generation-based fuzzing constructs new inputs from scratch according to some pre-defined grammar models. Since generation-based fuzzing does not rely on the seed inputs, it may cover more diverse input space that is not covered by the seed inputs and trigger more code logic of the program under test[30, 31].

Generation-based fuzzing has been widely used in many domains, such as C compilers [23] and so on [27–29, 32]. However, these techniques cannot be directed adopted to test DL compilers due to its characteristics. To our best knowledge, TVMFuzz[12] is the first generation-based technique to fuzzing low-level IR and low-level optimization of TVM. However, this open-source fuzzer can not cover high-level optimization and thus is incapable of detecting bugs in this most bug-prone stage. Different from TVMFuzz, HirGen is able to generate complicated and valid computational graphs independently from scratch to cover high-level optimization stage. This work is also achievable in different approaches by other generation-based fuzzers, such as GraphFuzzer [24] for inference engine and NNSmith for DL compilers [14]. To distinguish from these techniques, HirGen also utilizes IR language features and generates diverse high-level IRs. In this way, HirGen can effectively detect bugs related to high-level optimization.

7.2 DL Compiler Testing

With the development of DL compiler, the importance of DL compiler testing has been noticed by more and more researchers. To our best knowledge, existing DL compiler testing techniques can be divided into two categories according to their testing focus. The first category aims at testing the whole workflow of DL compilers. MT-DLComp and NNSmith are of this category. MT-DLComp [13] can perform semantics-preserving mutation on seed DL models to generate new models with theoretically the same prediction capability. During testing, any prediction difference among mutated models and the seed model or any compilation failure will be captured. NNSmith [14] is a state-of-the-art DL compiler fuzzer that can generate computational graphs and their inputs/weights from scratch and use them to test DL compilers. Although NNSmith and HirGen both generate computational graphs, HirGen are fundamentally different from NNSmith at least in terms of their testing purposes, their usage of the generated computational graphs in DL compiler testing, and the types of bugs detected. As for testing purposes, HirGen focuses on the most error-prone stage [8], i.e., the high-level optimization stage, while NNSmith has no testing preference for any compilation stage and focuses on validating the prediction correctness of the compiled models. Though HirGen can also cover low-level and codegen components, most of its technical details, including mutation strategies, use of high-level optimizations, and construction of $Oracle_2$, are all designed for the only stage. With different testing purposes, they utilize computational graphs differently. HirGen further generates multiple semantics-equivalent high-level IRs from the computational graphs and uses high-level optimizations to optimize them. NNSmith does not perform these steps. The utilization of high-level IRs and optimizations helps HirGen find much more high-level optimization bugs than NNSmith. NNSmith further finds a set of inputs and weights for the computational graphs such that the compiled DL models produce numerically valid outputs given such inputs and weights. Then NNSmith can validate prediction results and catch any prediction error. Though HirGen could also catch several prediction errors, it's not as efficient as NNSmith. Because of these differences, HirGen and NNSmith has nearly orthogonal bug detection ability, as shown in section 5.1.2.

Different from MT-DLComp and NNSmith, several other techniques [8, 11, 12] focus on the testing of single stage but not the whole workflow. Besides, they perform white-box testing to utilize knowledge gained from codebase to achieve more efficient and effective testing results. For instance, TZER[11] collects low-level IR passes and mutates them to focus on bug detection in low-level optimizations, while the above-mentioned TVMfuzz focuses on high-level optimization with generation-based approaches. Similar to these techniques, HirGen also focuses on single stage: high-level optimization. This stage is the most vulnerable to bugs and has not been systematically studied by previous literature. HirGen is therefore proposed to fill this gap.

7.3 Metamorphic Testing for Compiler

Metamorphic testing (MT) [33] is a popular approach to address the test oracle problem. Researchers proposed different metamorphic relations (MR) to construct test oracles for different systems under test based on the characteristics of the systems.

In compiler testing, the follow-up test inputs in MR are mostly programs equivalent to their seed programs [34]. Various semantics-preserving mutations have been proposed to generate equivalent programs for compiler testing [13, 35–38]. MT-DLComp [13] conducts metamorphic testing on DL compilers via two semantics-preserving mutations on computational graphs. Specifically, it inserts always-yield-zero nodes into computational graphs to generate new graphs without skewing the calculation. GLFuzz [38] tests OpenGL by six semantics-preserving mutators, including dead code injection, dead jump injection, live code injection, expression mutation, vectorization and control flow wrapping. Unlike MT-DLComp, $Oracle_2$ of HirGen is for high-level IRs instead of computational graphs, and semantics-preserving mutations in HirGen is also for high-level IRs. Moreover, the mutations of HirGen focus on modifying the function call chain, which is orthogonal to all six mutations in GLFuzz. In semantics-preserving mutation strategy

[6] https://discuss.tvm.apache.org/t/operator-right-shift-obtains-different-results-in-different-devices/11939

design, EMI [35], which stands for equivalence modulo inputs, is a methodology for constructing MRs [34]. The key insight is that given a seed program, a set of inputs can induce a collection of programs giving the same outputs as the seed one on these inputs. EMI has inspired several compiler testing techniques [35–37, 39]. The semantics-preserving mutation in EMI-based techniques requires profiling program executions before mutation. In addition, EMI-based mutants are constructed to be semantics-equivalent only under specific inputs. In contrast, HirGen requires no profiling, and the mutants generated are semantically equivalent to their seed high-level IR under all inputs.

8 CONCLUSION

High-level optimization is the most bug-prone stage in the workflow of DL compilers. However, there is no systematic study on testing this stage. To fill this gap, we offer HirGen, a generation-based fuzzer with effective computational graph generation approach and three test oracles. Different from existing works, HirGen can explore more complicated and valid high-level IR and thus detect deeper bugs. Besides, three test oracles in HirGen also improve its capability of detecting bugs of various root causes. Our effort has been recognized by TVM community and improved the robustness and functional correctness of high-level optimization.

REFERENCES

[1] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, ser. OSDI'18. USENIX Association, 2018.

[2] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," 2018.

[3] Google, "Xla: Optimizing compiler for machine learning," 2016. [Online]. Available: https://www.tensorflow.org/xla

[4] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. Kumar, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb, "Intel ngraph: An intermediate representation, compiler, and executor for deep learning," 2018.

[5] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," vol. 32, 2021.

[6] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 294–305.

[7] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in gcc and llvm," *Journal of Systems and Software*, vol. 174, p. 110884, 2021.

[8] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, 2021.

[9] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

[10] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 2016.

[11] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *arXiv preprint arXiv:2202.09947*, 2022.

[12] D. Pankratz, "Tvmfuzz," 2020. [Online]. Available: https://github.com/dpankratz/TVMFuzz

[13] D. Xiao, Z. LIU, Y. Yuan, Q. Pang, and S. Wang, "Metamorphic testing of deep learning compilers," *Proc. ACM Meas. Anal. Comput. Syst.*, 2022.

[14] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Finding deep-learning compilation bugs with nnsmith," 2022.

[15] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "DocTer: documentation-guided fuzzing for testing deep learning API functions," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2022.

[16] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 788–799.

[17] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Curran Associates Inc., 2019.

[19] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, "Relay: A new ir for machine learning frameworks," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018, 2018.

[20] "Onnx," 2022. [Online]. Available: https://onnx.ai/

[21] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 788–799.

[22] "Keras," 2022. [Online]. Available: https://keras.io/

[23] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[24] W. Luo, D. Chai, X. Run, J. Wang, C. Fang, and Z. Chen, *Graph-Based Fuzz Testing for Deep Learning Inference Engines*. IEEE Press, 2021.

[25] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing."

[26] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, 2022.

[27] H. Abdelnur, O. Festor, and R. State, "Kif: a stateful sip fuzzer," 2007.

[28] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using clp," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015.

[29] K. Dewey, J. Roesch, and Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. Association for Computing Machinery, 2014.

[30] C. Miller, Z. N. Peterson *et al.*, "Analysis of mutation and generation-based fuzzing," *Independent Security Evaluators, Tech. Rep*, vol. 4, 2007.

[31] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[32] P. Amini and A. Portnoy, "Sulley fuzzing framework," 2010.

[33] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," 2020.

[34] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," 2020.

[35] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," ser. PLDI '14, 2014.

[36] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," ser. OOPSLA 2015, 2015.

[37] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," ser. OOPSLA 2016.

[38] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," no. OOPSLA, 2017.

[39] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," ser. PLDI '15, 2015.