Abstract—Deep Learning (DL) compilers are widely adopted to optimize advanced DL models for efficient deployment at diverse hardware. However, the optimization of high-level intermediate representation (IR) is found to be error-prone. Testing the implementation of high-level optimization is challenging. First, high-level IRs are subject to integrity constraints. IRs that violate these constraints are rejected by DL compilers before entering the optimization stage. Such IRs are not useful to reveal optimization bugs. Since high-level optimization modules are so implemented to process possible computational graph structures, bug detection for these modules can be facilitated by feeding DL compilers with diverse computational graphs, which can be readily converted to high-level IRs. However, generating diverse computational graphs whose converted high-level IRs never violate the integrity constraints required by high-level optimization is non-trivial. Second, defining test oracles for optimization bugs in DL compilers is challenging.

To address the challenges, we propose HirFuzz, a fuzzing technique for bug detection at the optimization of high-level IR in DL compilers. We propose coverages based on data types, tensor shapes and computation operators to guide diverse computational graph generation, which is governed by the conformance of IR’s integrity constraints. We note that a primary implementation requirement of IR optimization is to avoid crash and preserve the computation results after optimization. We, therefore, design effective test oracles to detect crash and inconsistent computation. Leveraging the diverse computational graph generation and test oracles, HirFuzz has successfully detected 21 new high-level optimization bugs that occur at TVM, with 17 bugs confirmed and 12 fixed. Further, we construct three baselines using the state-of-the-art DL compiler fuzzers that can cover the high-level optimization stage. Our experiment results show that HirFuzz outperforms these baselines in bug detection ability. Besides, our ablation study and case study validate the usefulness of the proposed coverage criteria and test oracles.

Index Terms—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 first and low-level optimization second. 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] and MT-DLComp [13]. Despite preliminary reported success in bug detection for TVM, they are weak in revealing the bugs that occur in high-level optimization, which accounts 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 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 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).

To bridge the gap, we propose the first DL compiler fuzzing technique that focuses on high-level optimization: HIRFUZZ. HIRFUZZ is designed to satisfy the following three objectives: 1) the satisfaction of integrity constraints, such as type match and tensor shape match, that govern high-level IR to avoid early crash before invoking optimization, 2) the exploration of diversity of computational graph, and 3) the capability of detecting multiple types of optimization bugs. To achieve the first objective, HIRFUZZ performs type checking and shape checking in each operator node insertion by leveraging the information, including type, shape and connection information, of the existing nodes. After insertion, HIRFUZZ also updates the information of the new node for future use. To meet the second objective, HIRFUZZ incorporates a coverage-guided strategy to explore diverse operator nodes, operator edges and the combination of operator type and data type. To meet the third objective, HIRFUZZ 1) explores functions call chains to test function-related high-level optimization, and 2) 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 different prediction after optimization. Besides functional correctness, HIRFUZZ can also test the robustness of DL compilers. Specifically, HIRFUZZ provides an option of generating invalid computational graphs that violate type constraints and shape constraints [14]. The option aims to test whether DL compilers can catch such invalid computational graphs and throw the expected exceptions. In this way, HIRFUZZ can also detect bugs caused by incorrect exception handling.

Following existing works on DL compiler testing, we evaluate the performance of HIRFUZZ on TVM, which is the most popular DL compiler. In baseline selection, we choose 1) TVMfuzz (with lower-case f), preliminary proof-of-concept application from a bug study [8]. The tool is chosen because it is the only testing technique focusing on detecting bugs arising from high-level optimizations in DL compilers; 2) MT-DLComp [13], a metamorphic testing framework that can cover high-level optimization stage; and 3) LEMON [15], a fuzzing technique for DL library (e.g., Tensorflow [16] and PyTorch [17]) testing. 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, the conv2d (two-dimensional convolution) node in Figure 1 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 (nn.conv2d in Figure 1) 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 Background
2.1 DL Compilers

Figure 1 gives an overview of DL compilers. DL compiler takes as input DL models. These models can be constructed with the help of DL frameworks, such as Tensorflow [16] and PyTorch [17]. 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, the conv2d (two-dimensional convolution) node in Figure 1 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 (nn.conv2d in Figure 1) 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 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 2a illustrates the computational 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. Some DL compilers such as TVM provide APIs to convert a computational graph into a high-level IR. For instance, relay.nn.conv2d is the API to represent the conv2d node with the corresponding high-level IR expression nn.conv2d. Figure 2b illustrates
TVM’s high-level IR, Relay IR, of the computational graph in Figure 2a. These APIs simplify the generation of high-level IR. HirFuzz also takes advantage of this convenience and generates high-level IR by generating computational graph and utilizing these APIs.

### 2.3 Type Constraints and Shape Constraints

Similar to specifications of common programming languages [18], [19], constraints in high-level IR is useful to distinguish valid IR from the invalid one. Constraints include type constraints and shape constraints. Any break of these constraints will cause an early crash before high-level optimizations. Since high-level IR is used to describe computational graph, any constraint breaking in the graph will propagate to high-level IR and thus trigger an early crash. Common constraints include 1) each operator has acceptable data types on its operands 2) type-compatibility and shape-compatibility across the multiple parameters of each operator. In TVM, type system is so rigorous that type-compatibility is implemented as type-equality. As for shape-compatibility, take BroadcastRel as an example, TVM allows broadcast in basic operators to make shape inference more flexible. This broadcast is similar to that in Numpy [20], allowing small tensors to be operated with big tensors. For instance, TVM supports addition between a scalar (small tensor) with an array (big tensor). Therefore, HirFuzz needs to consider type and shape constraints in computational graph generation so that they are not violated by the underlying high-level IRs of the generated graphs.

### 3 Approach

This section presents the design and underlying methodology of HirFuzz whose workflow is given in Figure 3. Hir-
3.1 Computational Graph Generation

3.1.1 Overview.

We consider the generation of 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 number required. Generally speaking, HIRFUZZ selects one operator from the operator pool, loads the operator into \( CG \) as a node \( nd \), and constructs connection between \( nd \) and other existing nodes. In these steps, HIRFUZZ maintain node information, including data type, tensor shape and connection information, for each node. To improve the diversity of the graph, HIRFUZZ also involves three coverage criteria and try to improve the computational graph generation approach coveraged-guided generation.

With these prerequisites, HIRFUZZ provides two generation modes. To generate valid computational graphs, HIRFUZZ 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, HIRFUZZ 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 \( \text{add} \) that sums two nodes, HIRFUZZ 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 \( \text{add} \). 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, parentNode, tensorShape = INFERENCeparentNodes |

TABLE 1: Node Information

Specifically, HIRFUZZ considers three types of nodes, namely, \textit{variable}, \textit{constant} and \textit{operator}. The detailed introduction of them is as follows.

1) \textit{Variable} node. It involves data type \textit{dataType} and tensor shape \textit{tensorShape} describing the details of the tensor wrapped in this node. \textit{dataType} corresponds to the data type of all elements in this tensor, such as \textit{int64} and \textit{float32}. \textit{tensorShape} is a vector of the scale of all dimensions in the tensor.

2) \textit{Constant} node. Besides the \textit{dataType} and \textit{tensorShape}, \textit{constant} node includes the value of tensor \textit{value} as a part of its information.

3) \textit{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 \textit{dataType}, HIRFUZZ records its parent node(s) \textit{parentNode} to which this node connects and records its tensor shape inferred from parent node(s).

3.1.3 Coverage Guidance.

To let HIRFUZZ intelligently generate diverse computational graph with in-depth thoughts about selection of data type, operator, etc, we design three coverage criteria.

1) \textbf{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 \( \text{Cov}(op_i, dtype_j) \) 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) \textbf{Operator-shape Coverage}. Let \( op_i \) be the \( i_{th} \) operator in the operator pool. Let \( \text{shape} \) be the shape of the output tensor of this operator node after being inserted into the graph. Let \( \text{Cov}(op_i, \text{shape}) \) be 1 if \( op_i \) has once been inserted into the graph as a node with tensor shape \( \text{shape} \), and 0 otherwise.

3) \textbf{Operator-edge Coverage}. Let \( op_i \) and \( op_j \) be the \( i_{th} \) and \( j_{th} \) operator in the operator pool. Let \( \text{Cov}(op_i, op_j) \) 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 \cite{8}. The design of the third one tries to complicate the data flow of the computational graph since the third coverage encourages HIRFUZZ to interleave different operators in a computational graph. Specifically, with operator-datatype coverage, HIRFUZZ is encouraged to 1) involve different operators into the graph and 2) utilize diverse data types since data type problem is a big concern for DL compilers \cite{8}. With operator-shape coverage, HIRFUZZ is encouraged to try various calculation with diverse tensor shapes and thus increase the probability of encountering calculation problem, such as poor implementation of some operator in special shape or different calculation results on different platform. With operator-edge coverage, HIRFUZZ 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. HIRFUZZ is encouraged to explore the diversity of computational graph by increasing these three coverages. Therefore, we name our computational graph generation approach coverage-guided generation.

3.1.4 Strict Generation

Algorithm 1 presents how HIRFUZZ strictly generates computational graph with type-checking and shape-checking by two procedures. GENERATION is the main procedure. This procedure takes as input the required number of operators \( rOpNum \) contained in computational graph. It outputs a computational graph of which the number of operators equals to \( rOpNum \). PREINSERT is the auxiliary procedure.
It details how HirFuzz embeds type-checking and shape-checking in generation. GENERATION procedure includes the following two main parts.

Algorithm 1 Computational Graph Generation

1. procedure GENERATION(opNum)
2.   CG ← \{
3.     opnum ← 0
4.     opPool ← \{add, subtract, multiply, divide, ...\}
5.     dataTypeSet ← \{int64, int32, int16, int8, uint64, uint32, uint16, uint8, float64, float32, bool\}
6.     repeat
7.       opNode ← select(opPool)
8.       dataType ← select(dataTypeSet)
9.       connection, shape, CG ← PREINSERT(opNode, dataType, CG)
10.      coverage ← CALCULATECOVERAGE(opNode, dataType, connection, shape)
11.     if NOT COVERAGE(coverage) then
12.       opnum ← opnum + 1
13.     end if
14.     if NODE NOT ENOUGH(paramNodes) then
15.       nodeGroup1, nodeGroup2, ... ← SHAPECHECK(availableNodes)
16.       paramNodes ← SELECT(nodeGroup1, nodeGroup2, ...)
17.       CG ← CG∪\{node\}
18.     end if
19.     while opnum = rOpNum
20.     return CG
21. end procedure
22. procedure PREINSERT(opNode, dataType, CG)
23.   availableNodes ← TYPECHECK(CG, dataType)
24.   nodeGroup1, nodeGroup2, ... ← SHAPECHECK(availableNodes)
25.   paramNodes ← SELECT(nodeGroup1, nodeGroup2, ...)
26.   if NODE NOT ENOUGH(paramNodes) then
27.     node1, node2, ... ← CREATE(dataType, paramNodes)
28.     CG ← CG∪\{node1, node2, ...\}
29.   end if
30.   if 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. HirFuzz 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), HirFuzz generates an operator node, updates its node information and finally inserts it into CG if no new coverage is explored. Specifically, HirFuzz 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, HirFuzz calculates coverage and performs update and insertion if no 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, HirFuzz adds opnum by 1, updates coverage and node information of opNode. The generation loop stops when opnum equals rOpNum and HirFuzz returns CG eventually.

Procedure PREINSERT shows the details of building connection between opNode and existing nodes of CG and shape inference. With type-checking (Line 22) and shape-checking (Line 23), HirFuzz sorts out several node groups from CG. All nodes of each node group are mutually shape-compatible and nodes from these groups are all type-compatible with opNode. Then HirFuzz 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, HirFuzz 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, HirFuzz discourages this behavior in favor of connecting the required number of different nodes. Therefore, if the number of parameter nodes in paramNodes are insufficient for opNode, HirFuzz 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, HirFuzz creates connection information (Line 30-32), infers tensor shape of opNode and returns them both plus the possibly updated CG.

3.1.5 Disruptive Generation Algorithm.

Disruptive generation is similar to strict 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, HirFuzz purposefully 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 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 API1. 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.

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, HirFuzz creates two empty sets, named Functions

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

and Expressions respectively (Line 2, 3). They represent the collection of functions and high-level expressions, respectively. In the for loop (Line 4-12), HIRFUZZ traverses all nodes in CG, loads each node into high-level expression and update Expressions (Line 5). Then it randomly selects a set of high-level expressions and wraps them with a function (Line 6-11). To compose a function, HIRFUZZ first analyzes the input nodes and output nodes of the underlying subgraph of the expressions (Line 7). Then it composes a function using these nodes (Line 8. Finally, HIRFUZZ 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, HIRFUZZ 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 return Expressions after update.

3.3 Test Oracles

Test oracle is an important mechanism to determine if a test passes or fails. In this paper, we consider three test oracles in total. Any failed test case determined by these oracles will be reported.

3.3.1 Oracle1: 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. This is because 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 Oracle2: Result Inconsistency among original high-level IR, optimized high-level IR and mutated high-level IR
It is intuitive that 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 IR.

All mutation examples are shown in Figure 4. 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 $g$ (Refer to Mutated High-level IR-1).
- 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 $g$ (Refer to Mutated High-level IR-2).
- Call a function $f$ and return the call in another function $g$, then substitute all calls to $f$ with calls to $g$ (Refer to Mutated High-level IR-3).

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. 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 Oracle3: 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 Oracle2, 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 Oracle3 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 HirFuzz in detecting bugs of TVM?
- RQ2: Are all the test oracles effective in detecting bugs?
- RQ3: Are bugs found by HirFuzz 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 HirFuzz Implementation

We implement HirFuzz in C++ with around 3K lines of code. Our implementation involves 58 operators 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, HirFuzz includes 58 operators supported by TVM, including 23 binary operators and 35 unary operators. And it is easy to extend HirFuzz with other operators. The following lists are the two groups of operators involved by HirFuzz.

- **Binary operators**: Add, Subtract, Multiply, Divide, Pow, Mod, Floor Mod, Floor Divide, Logical And, Logical Or, Logical Xor, Bitwise And, Bitwise Or, Equal, Not Equal, Less, LessEqual, Greater, GreaterEqual, Maximum, Minimum, Right Shift, Left Shift.

- **Unary operators**: Log, Log2, Log10, Tan, Tanh, Cos, Cosh, Sin, Sinh, Acos, Acosh, Asin, Asinh, Atan, Atanh, Exp, Erf, Sqrt, Rsqrt, Sigmoid, Floor, Ceil, Trunc, Round, Abs, Sign, Negative, Logical not, Bitwise not, Zeros Like, Ones Like, Copy, isNan, isNaN, isFinite, isInf.

4.2.2 Optimization and Compilation Methods.

We select in total 25 high-level optimizations supported by TVM\textsuperscript{2} 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 HirFuzz. Besides these high-level optimizations, it is easy to extend HirFuzz 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\textsuperscript{3}; 2) report the bug on Github Issue\textsuperscript{4} with 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 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. 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 [23]. It generates Keras [24] 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.

\textsuperscript{2} https://github.com/haoyang9804/HirFuzz/blob/experiment/optimizations

\textsuperscript{3} https://discuss.tvm.apache.org/

\textsuperscript{4} https://github.com/apache/tvm/issues
4.5 Metrics
To compare HiRFUZZ with these three baselines in the ability of detecting bugs in high-level optimization stage, we execute them all separately in two days. To be fair, we only execute HiRFUZZ in strict generation mode, since all other three techniques can only generate valid DL models and APIs. As for bug counting, we utilize the informative stack trace and bug text provided by TVM and manually check bug duplication. Following existing works [25], failures with same bug text and same bug trace are regarded as duplicates. We also use this metric in the comparison between coverage-guided generation and no-guide generation in the answer to RQ5.

4.6 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 HiRFUZZ
5.1.1 Summary of Detected Bugs by HiRFUZZ
Until now, HiRFUZZ has found 21 bugs, of which 17 have been confirmed, 12 have been fixed in the main branch of TVM and 10 are previously unknown to developers. Table 2 presents all details about all the confirmed bugs discovered by HiRFUZZ, including their symptoms, root causes, the test oracles detecting them, the fixing status and whether they are previously unknown. 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
We conducted a comparison experiment among HiRFUZZ and other three state-of-the-art DL compiler fuzzers, named TVMfuzz, MT-DLComp and LEMON respectively, to compare their bug detection ability. All tests are executed on TVM v0.9 (commit id: 124813f). This version is the initial TVM version we began our first testing using HiRFUZZ. After two-day execution, HiRFUZZ detected 11 non-duplicate bugs, TVMfuzz detected three non-duplicate bugs, of which only one bug has not been detected by HiRFUZZ, while MT-DLComp and LEMON detected nothing. These bugs have been released in HiRFUZZ repository.

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.

*Oracle1: Crash. Oracle1 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.*

![Fig. 5: The Computational Graph to Trigger Bug1](image)

*Incorrect Numerical Computation. Take Bug1 as an example. The computational graph to trigger this bug is presented in Figure 5. In this graph, 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 pre-calculates 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. Bug11, Bug12 and Bug13 are three bugs of Incorrect Exception Handling. They are detected under disruptive generation. To trigger these bugs, HiRFUZZ must generate computational graphs containing obvious breaks of constraints. Take Bug11 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, HIRFUZZ 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.** Bug14 is the only bug of root cause, named Memory Allocation Problem. Specifically, when HIRFUZZ leverages \( \text{relay}\_\text{shape\_of} \) 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 \([?, ?]\).

**Oracle2:** Result Inconsistency among original high-level IR, optimized high-level IR and mutated high-level IR. Oracle2 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 Bug10 as an example. HIRFUZZ catches this bug because it finds a high-level IR passes compilation while its optimized version fails. Specifically, HIRFUZZ 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.

**Type Problem.** Take Bug8 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 Bug9 for example. This bug is detected because \( \text{relay}\_\text{build}\_\text{module}\_\text{create}\_\text{executor}('vm') \) fails, but compilation in other ways run smoothly. Specifically, after HIRFUZZ transforms high-level executor 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.

**Oracle3:** Result Inconsistency across hardware devices. Oracle3 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 Bug15 as an example. HIRFUZZ creates a simple computational graph containing a right_shift operator node. This operator node takes as input two other variable nodes. Subsequently, HIRFUZZ first generates the corresponding high-level IR, then compiles the IR with \( \text{relay}\_\text{build} \) 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, HIRFuzz 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 HIRFuzz and analyze their relationship with high-level optimization and how the detection of them improve this stage.

\textit{Bug}_2, \textit{Bug}_6, \textit{Bug}_9, \textit{Bug}_{10} are bugs detected in high-level optimization. Bug-triggered pattern for these four bugs are similar: after high-level optimizations, HIR level optimization. Bug-detected pattern for these four bugs is not guided to generate computational graphs. We have conducted three experiments for comparison between these two techniques.

Besides, HIRFuzz finds a total of eight bugs with crash symptom and all of them trigger crash during compilation. Except for \textit{Bug}_{11}, \textit{Bug}_{12} and \textit{Bug}_{13}, all other bugs are directly related to high-level optimization. To improve efficiency, TVM calls \textit{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. Fix for these bugs are actually fix for required IR passes needed by 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. \textit{Bug}_{15}, \textit{Bug}_{16} and \textit{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 better couple TVM with these backends.

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. In this section, we will present three experiments to show the usefulness of these criteria in generating diverse computational graph and finding more bugs. We implement a simplified version of HIRFuzz, saying HIRFuzz\textsubscript{nancov}. HIRFuzz\textsubscript{nancov} is identical to HIRFuzz except that HIRFuzz\textsubscript{nancov} is not guided to generate computational graphs. We have conducted three experiments for comparison between these two techniques.

The first experiment is about the diversity scores achieved by HIRFuzz and HIRFuzz\textsubscript{nancov}. In short, diversity score is a quantification of the coverage achieved by HIRFuzz and HIRFuzz\textsubscript{nancov}. In experiment setting, each new piece of coverage increases diversity score by 1. In this experiment, HIRFuzz and HIRFuzz\textsubscript{nancov} both generated 200 computational graphs, each of which contains 100 operators. Figure 6 presents the experiment result. After all generations, HIRFuzz\textsubscript{nancov}'s score is more than five times than that of HIRFuzz\textsubscript{nancov}. HIRFuzz explores stable amount of new coverage in each computational graph generation, while HIRFuzz\textsubscript{nancov} explores less and less with the increase of the number of graphs.

The second experiment is about the bug detection ability of HIRFuzz and HIRFuzz\textsubscript{nancov} under strict generation mode. In this experiment, we executed these two techniques for two days separately and collect all failing tests. HIRFuzz found 11 bugs out of 24 failing tests, while HIRFuzz\textsubscript{nancov} found 8 bugs out of 37 failing tests. This result shows that 1) HIRFuzz has better bug detection ability than HIRFuzz\textsubscript{nancov} and 2) with coverage-guided generation, HIRFuzz can avoid triggering duplicate failure and focus on finding new bugs.

The third experiment is similar to the second one. In this experiment, we compare the bug detection ability of HIRFuzz and HIRFuzz\textsubscript{nancov} under 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. Therefore, we utilize these two techniques to generate computational graphs that contain only one operator. Figure 7 presents the experiment results. HIRFuzz and HIRFuzz\textsubscript{nancov} both generated 170 bug-triggered compu-
tational graphs, each of which contains unique tuple of \( \text{operator, tensor shape, data type} \). In this figure, HIRFuzz shows more exploratory nature in diversity of graph and thus detects bugs faster. Besides, two techniques both found 3 bugs using these 170 graphs. And the timestamps of bug detection are also marked in this figure, showing that HIRFuzz found bugs faster than HIRFuzz\(_{noncov}\).

These three experiments show that coverage-guided generation help HIRFuzz become more exploratory in the diversity of computational graphs. Therefore, HIRFuzz can detect more bugs and find them in faster speed.

### 6 Discussion

#### 6.1 Threats to Validity

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

The **external** threat 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 in Github. Though HIRFuzz 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 [22]. And we also have one experimental version to support ONNX. ONNX is an open format to represent diverse DL models defined by various DL frameworks and is supported by currently popular DL compilers. Similar to Relay, we can use ONNX’s APIs to easily convert a computational graph into 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 **construct** threat mainly lies in randomness. In computational graph generation, though with coverage guidance, the selection of operator and connection also involves randomness. To alleviate the negative impact of construct threat, we 1) conducted bug detection ability comparison experiment for a sufficient time, and 2) conducted the first and the third experiment in section 5.5 for 10 times to assure the stability of the experimental results.

#### 6.2 Future Work

HIRFuzz can be potentially extended in these three aspects. First, the operators and high-level optimizations supported by HIRFuzz are extensible. We can easily add new operators and high-level optimizations to HIRFuzz in the future.

Second, HIRFuzz only focuses on TVM currently. In the future, we can improve the support for ONNX and test diverse DL compilers with the help of this framework.

Third, since the size of the generated computational graphs could be large by manual setting, the reduction of graph size may cost huge manual effort. Directly submitting a large bug-triggered computational graph is not friendly to developers and may leave the bug ignored. Besides, though TVM provides bug stack trace and bug information, it is also not easy to determine whether the bug is duplicate. And the submission of duplicate bug is harmful for the efficiency in software development. Therefore, we will work on automatically reducing computational graph and even filtering out duplicate bugs of DL compilers in the future.

### 7 Related Work

HIRFuzz is a generation-based fuzzer for DL compiler testing. In this subsection, we introduce two categories of related work in this section, including generation-based fuzzing and DL compiler testing.

**Generation-based Fuzzing.** Generation-based fuzzing is a class of common fuzzing techniques [26], [27], [28], [29], [30]. 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 [31], [32].

Generation-based fuzzing has been widely used in many domains, such as C compilers [30] and so on [27], [28], [29], [33]. However, these techniques cannot be directly 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 cannot cover high-level optimization and thus is incapable of detecting bugs in this most bug-prone stage. Besides, authors of a DL Compiler bug study [8] propose a proof-of-concept prototype also named TVMFuzz (with lowercase f) to test high-level optimization. Specifically, TVMFuzz learns API call chains and parameters needed by these APIs from unit tests of TVM and attempts to generate test scripts by reordering these API calls. It attempts to cover some under-tested paths in this way. However, it is highly dependent on unit test and can not explore complicated high-level IRs to test high-level optimization systematically. Different from these two techniques, HIRFuzz is able to generate complicated and valid computational graphs independently from scratch and then their corresponding high-level IRs with thousands of nodes and diverse data types and tensor shapes. In this way, HIRFuzz can effectively detect bugs related to high-level optimization.
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 technique can be divided into two categories according to their testing focus. MT-DLComp [13] aims at testing the whole workflow of four DL compilers, including TVM, Glow, NNFusion and Tensorflow XLA. This technique performs mutation on existing DL models and involves three test oracles from the spirit of metamorphic testing. During testing, it treats all tested Compilers as a black box and test if DL compilers corrupt the predictive capability of DL models. Different from MT-DLComp, 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 TVM-fuzz focuses on high-level optimization with generation-based approaches. Similar to these techniques, HirFuzz 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. HirFuzz is therefore proposed to fill this gap.

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 HirFuzz, a generation-based fuzzer with effective computational graph generation approach and three test oracles. Different from existing works, HirFuzz can explore more complicated and valid high-level IR and thus detect deeper bugs. Besides, three test oracles in HirFuzz also improve its capability of detecting bugs of various root causes. In total, HirFuzz has detected 21 bugs, of which 17 have been confirmed and 12 have been fixed. 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. Abdurrasool, 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. Bhawandia, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovalko, V. Kumar, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A. Procter, and T. J. Webb, “Intel xgraph: 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. Self, A. Kornhauser, and J. Xiao, “Deepdriving: Learning guidance for driving 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 deep learning compiler fuzzing with joint ir-pass mutation,” arXiv preprint arXiv:2202.08947, 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] 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.

[15] 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.

[16] 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/

[17] 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.

[18] B. Yu, L. Kong, Y. Zhang, and H. Zhu, “Testing java components based on algebraic specifications,” in 2008 1st International Conference on Software Testing, Verification, and Validation. IEEE, 2008, pp. 190–199.

[19] J. W. Nimmer and M. D. Ernst, “Automatic generation of program specifications,” ACM SIGSOFT Software Engineering Notes, vol. 27, no. 4, pp. 229–239, 2002.

[20] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Rio, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, M. Millman, J. K. Vessey, M. Van Mulbregt, and T. A. Peterson, “Scipy 1.7.0 release,” 2020.

[21] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Raison, L. Quinlan, J. Scoltock, M. Abadi, A. Bastoul, and T. J. Webb, “Intel ngraph: An intermediate representation, compiler, and execution for deep learning,” 2018.

[22] “Onnx,” 2022. [Online]. Available: https://github.com/onnx/onnx

[23] 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.

[24] “Keras,” 2022. [Online]. Available: https://keras.io/

[25] 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.

[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 type-checker 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] 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.

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

[32] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” Cybersecurity, vol. 1, no. 1, pp. 1–13, 2018.

[33] P. Amini and A. Portnoy, “Sulley fuzzing framework,” 2010.