SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2025, Vol. 68, Iss. 9, 192104:1–192104:20
https://doi.org/10.1007/s11432-023-4301-6

Effective random test generation for deep learning compilers

Luyao REN[1,2], Ziheng WANG[1,2], Li ZHANG[3], Guoyue JIANG[3], Yingfei XIONG[1,2] & Tao XIE[1,2*]

[1]*School of Computer Science, Peking University, Beijing 100871, China*
[2]*Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University), Beijing 100871, China*
[3]*Sophgo Technologies Ltd., Beijing 100176, China*

Received 14 February 2023/Revised 25 February 2024/Accepted 5 August 2024/Published online 18 August 2025

Abstract Deep learning compilers help address the difficulties of deploying deep learning models on diverse types of hardware. Testing deep learning compilers is highly crucial, because they are impacting countless AI applications that use them for model optimization and deployment. To test deep learning compilers, random testing, the testing method popularly used for compiler testing practices, faces the challenge of generating semantically valid test inputs, i.e., deep learning models that satisfy the semantic model specifications (in short semantic specifications). To tackle this challenge, in this paper, we propose a novel approach named Isra, including a domain-specific constraint solver that resolves the constraints from the semantic specifications without backtracking. We implement and apply our approach to three popular real-world deep learning compilers including TVM, Glow, and a commercial compiler named SophGo. The evaluation results show that Isra is more effective than the state-of-the-art approaches and the baseline approaches on constructing valid test inputs for compiler-bug detection, and Isra successfully finds 24 previously unknown bugs in released versions of the three compilers. These results indicate Isra's effectiveness and practical value.

Keywords random testing, test generation, deep learning compilers, compiler testing, constraint solving

Citation Ren L Y, Wang Z H, Zhang L, et al. Effective random test generation for deep learning compilers. Sci China Inf Sci, 2025, 68(9): 192104, https://doi.org/10.1007/s11432-023-4301-6

1 Introduction

In recent years, deep learning has been widely used in software systems from various domains, such as autonomous driving, e-commerce, and smart cities. Given that deep learning models become increasingly large and complicated, there are emerging needs to deploy versatile deep learning models on different types of hardware such as graphics processing unit (GPU), field programmable gate array (FPGA), and tensor processing unit (TPU) [1]. To reduce the burden of optimizing deep learning models and to address the difficulty of model deployment on hardware, deep learning compilers have been developed, such as TVM [2], Glow [3], and XLA [4]. These deep learning compilers have been widely used for the optimization and deployment of deep learning models, especially those with critical performance requirements.

Testing a deep learning compiler is vital for two main reasons. First, if a deep learning compiler used by AI applications contains bugs, the deployed AI applications can exhibit serious failing behaviors. For example, a critical bug of TVM's SPIRV codegen led to incorrect results for a TVM-optimized model's output, which affected all users who use TVM for their model deployment on the Nvidia Vulkan backend[1)]. Second, the highly sophisticated compilation process in a deep learning compiler is error-prone. According to a very recent study [5], during a time period of 15 months, there are 845 bug-fixing pull requests on the TVM project, including 318 bugs in total.

A deep learning model, as the input of a deep learning compiler, needs to satisfy semantic specifications in two aspects; otherwise, it will be rejected by the deep learning compilers at an early stage before

* Corresponding author (email: taoxie@pku.edu.cn)

1) https://github.com/apache/tvm/pull/8102.

invoking the actual core functionality of the compilers. First, a deep learning model is a neural network arranged as a directed and acyclic graph. Second, the model also needs to satisfy certain constraints, which are required by the operations in the model. For example, within a model, a `MatMul` operation (denoting matrix multiplication) with two input matrices (2-D tensors) requires that the number of columns in the first matrix is equal to the number of rows in the second matrix.

To test core compiler parts (achievable by only valid inputs), one can indeed adopt random testing (the most popularly used technology for compiler testing practices [6]) in a straightforward way: generating possible inputs and filtering them by checking against the semantic specifications[2)], also called as declarative-style random test generation [7–9]; however, this style suffers from two main issues. First, random testing in the declarative style has a fairly low probability of satisfying the semantic specifications, especially for complicated operations (as shown in our experimental results in Section 4), wasting testing the budget on generating and checking many invalid inputs. Second, valid inputs generated by this random testing strategy tend to be simple, as complex inputs have an even lower probability of satisfying the semantic specifications, whereas it is highly critical to also generate complicated models in order to achieve various testing goals.

To better conduct random testing on a deep learning compiler, we take the semantic specifications of a deep learning model as logic constraints; in this way, test generation is equivalent to finding solutions to the constraints. However, we face a challenge due to complex constraints, i.e., those related to both the graph structure of the model and operations within the model. Furthermore, within the constraints, the involved first-order/second-order logic (as shown in Section 2) is undecidable in general [10] and causes existing solvers not to be able to encode, or perform efficiently [11, 12].

To address the preceding challenge, we propose a novel approach named Isra based on the following insight: the constraints on a deep learning model have certain properties, allowing us to iteratively resolve and simplify the constraints to effectively find solutions, by following a proper instantiation order. We design two strategies in the core part of Isra, a novel domain-specific constraint solver. Our solver conducts instantiation with an order for gradually resolving and simplifying constraints. Based on the consistency among the constraints, Isra, with our domain-specific constraint solver, is able to find semantically valid inputs without backtracking, while ensuring both soundness (the generated inputs are semantically valid) and completeness (no loss for the probability of generating any valid model).

To evaluate Isra, we implement it and empirically compare it with five baselines: (1) the aforementioned approach of random test generation, named as declarative-style generation, (2) test generation based on the idea of feedback-guided test generation [13], named Randoop-like generation, (3) a state-of-the-art tool named Muffin [14] implementing a generation-based approach for deep learning models, (4) a mutation-based approach, TVMFuzz [5], toward testing deep learning compilers, and (5) a state-of-the-art generation-based approach named NNSmith [15] toward testing deep learning compilers. Our evaluation results show that our Isra approach substantially outperforms the baselines on the metrics of generated valid test inputs under the same setting, for demonstrating our approach's effectiveness. Furthermore, to investigate the bug detection capability, when used to test the same benchmark (TVM, Glow, and SophGo), Isra detects 33 unique bugs in total (with 18 on TVM, 4 on Glow, and 11 on SophGo), performing better or as well than the baselines.

In addition, among the bugs found by Isra, there are 24 previously unknown bugs. After these previously unknown bugs were reported to compiler developers, 19 were confirmed and 16 were already fixed upon our bug reporting so far. The positive feedback from the developers also shows Isra's high value in practice. The source code, experimental results, and bug reports are publicly available at the websize[3)].

In summary, this paper makes the following contributions.

- An effective test generation approach named Isra for testing deep learning compilers, based on instantiation-based constraint solving, working in a backtrack-free way, with the guarantee of soundness and completeness.
- A domain-specific constraint solver toward finding solutions to the constraints specified from the semantic specifications of a deep learning model, with two novel strategies for resolving complex constraints.
- Implementation and evaluation of our approach for showing its high effectiveness and high practical value, including outperforming state-of-the-art approaches on coverage metrics, achieving comparable and

2) Checking against the semantic specifications can be in the form of a boolean checker such as `repOK` [7].

3) https://github.com/israProj/isra.

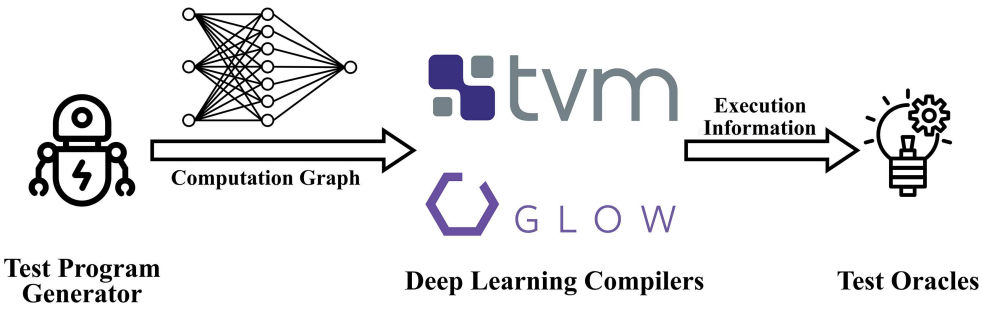

Figure 1 (Color online) Pipeline of testing deep learning compilers.

complementary results on the bug detection, and successfully finding 33 unique bugs in three popular real-world deep learning compilers.

2 Background and overview

Figure 1 shows our overall pipeline of random testing for deep learning compilers. In the stage of test generation, we use our test program generator to generate random computation graphs that are semantically valid. For the test oracle, we use both differential testing and the crash status of the compiler under test [6]. Before formally describing our approach in detail (as shown in Section 3), we first take an overview of the background with specific examples, then illustrate our approach in a nutshell.

2.1 Computational graph for deep learning

A deep learning model, as the input of deep learning compiler, can be regarded as a computation graph, where an edge represents a tensor (an N-D array), denoting the flow of data between nodes, and a node represents (1) an operation (denotes a mathematical function) whose inputs are the incoming edge(s) and outputs are the outcoming edge(s), or (2) a tensor placeholder where we need to represent creation of a user-defined variable such as the input tensor of computational graph. The computation graph can be executed by feeding the specific data into tensor placeholders. The formal definitions of the computation graph are shown in Section 3.

As an example, Figure 2 shows a deep learning model with two operations. The first operation is `Add`. It takes two tensors p and q as its input and outputs a tensor r as their sum. The second operation is `Concat`. It accompanies an attribute axis (denoting the dimension of the axis to concatenate on) and takes two tensors r and s as its input, and outputs a tensor t as their concatenated results. The edge in the computation graph represents the dataflow that gets transferred between the nodes. For example, r, as the output of `Add` operation, could be transferred to the input of `Concat` operation.

A computation graph is directed and acyclic, specifying the order of computation. In the example, you need to compute `Add` first in order to compute `Concat` because the output of `Add` (i.e., tensor r) flows to the input of `Concat`. Except for the acyclicity of the graph, for each operation, the number of incoming edges should be aligned with the number of input arguments defined by the corresponding mathematical function that the operation denotes. For example, `Concat` requires two or more input arguments, so the number of incoming edges should be more than or equal to two. We called those semantic specification of computation graphs as graph-level constraints.

Besides graph-level constraints, each operation in the computation graph holds its internal semantic specification, specified from the definition of the mathematical function that the operation denotes, which we call operation-level constraints. In our example, as the input of `Add` operation, tensor p and q should have the same shape. Similarly, as for `Concat` operation, tensor r and s must have the same shape, except for the dimension of the axis to concatenate on (defined by an operation's attribute axis). Particularly, by explicitly denoting the structure of tensors, operation-level constraints of the computation graph in Figure 2 could be specified as follows ($\dim_a$ denotes the dimension size of tensor a, and $a[i]$ denotes the length of the i-th dimension of tensor a):

$$\dim_p = \dim_q = \dim_r, \tag{1}$$

$$\forall i \in [1, 2, \ldots, \dim_p],\ p[i] = q[i] = r[i], \tag{2}$$

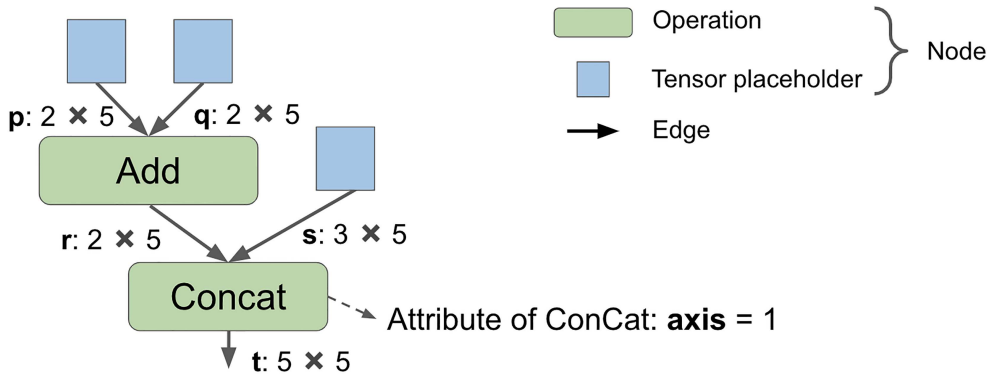

Figure 2 (Color online) Example of a computation graph.

$$\dim_r = \dim_s = \dim_t, \tag{3}$$

$$\forall i \in [1, 2, \ldots, \dim_r] \wedge i \neq \text{axis},\ r[i] = s[i] = t[i], \tag{4}$$

$$t[\text{axis}] = r[\text{axis}] + s[\text{axis}], \tag{5}$$

$$1 \leqslant \text{axis} \leqslant \dim_r. \tag{6}$$

2.2 Challenges

The complicated semantic specifications of the computation graph, which consist of both the graph-level constraints and operation-level constraints, result in the sparsity of semantically valid inputs (compared to syntactically valid inputs). Thus, random test generation suffers the issues on effectiveness. Specifically, the approach that randomly generates possible computation graphs, and then filters invalid ones by checking against the semantic specification, holds fairly low possibility to satisfy the semantic specification. As our previous example, for two tensors of the input of an `Add` operation, assume that the range of the tensor's dimension size is $O(D)$ and the length of each dimension is $O(L)$, the generation holds $O(L^{-D})$ possibility of producing valid ones due to (2). The possibility diminishes with a larger deep learning model, wider range, or more complex specifications.

In order to better conduct random testing on deep learning compilers, instead of taking semantic specification as a blackbox checker, we explicitly specify the semantic specifications of computation graph as constraints, i.e., take semantic specification as a whitebox, and test generation is equivalent to find solutions that satisfy those constraints.

However, existing practices of constraint solving are limited due to our complex constraints which are expressed in first-order/second-order logic instead of propositional logic due to the following reasons: (i) the acyclicity of computation graph; (ii) the existence of quantifiers such as $\forall i \in [1, 2, \ldots, \dim_p]$ in (2); (iii) the existence of unknown functions such as $r[\text{axis}]$ in (5) (it is called unknown function because we actually need to construct a function that maps to the length of each dimension of a tensor that we may not know the dimension size). Compared with propositional logic that is decidable, solving first-order/second-order logic (with quantifiers and unknown functions) is challenging because theoretically the first-order/second-order logic is undecidable in general [10] and also, in practice, quantifiers and unknown functions cause existing solvers unable to encode, or perform inefficiently [11, 12, 16].

2.3 Instantiation-based constraint solving

Instantiation [17–21] is a widely used technique for solving constraint satisfaction problem (CSP) such as satisfiability modulo theories (SMTs). By assigning the values to the variables in the constraints, we could get an instantiation of the constraints.

An instantiation-based solver starts from an empty instantiation and extends instantiation (mostly in an iterative way) to find solutions. An instantiation could be extended by assigning the values to the variables that were not assigned in the instantiation before. In the meanwhile, by replacing the variables with their assigned specific values in the constraints, also, with the help of solver, it is possible to simplify constraints and reduce the domain of unassigned variables (it is called as constraint propagation [17]). For example, for the constraint $S \subset T$ (S and T are set variables), if we first instantiate T as $\{1, 2\}$, then

it is easy to simplify the constraints by solving constraints and simultaneously deducing the domain of S, i.e., $S \subset T \Rightarrow S \subset \{1,2\} \Rightarrow S \in \{\emptyset, \{1\}, \{2\}\}$.

With instantiation, the elimination technique has a chance to be applied for the simplification on constraints that are originally encoded in first-order logic (or higher-order logic). Inspired by quantifier elimination [12,22], if we already determine the domains of quantifiers or unknown functions in constraints according to the instantiation, we could then simplify the constraint by rewriting the constraints to a quantifier-free and unknown-function-free form. For example, assume S is a set variable, for the constraint $\forall x \in S, P(x)$, if we already instantiate the domain of the universal quantifier x as $S : \{1,2,3\}$, then we could eliminate the quantifier by rewriting the constraints as follows: $\forall x \in S, P(x) \Longrightarrow P(1) \wedge P(2) \wedge P(3)$.

We call an instantiation consistent if it could be extended to a solution, otherwise it is inconsistent. For example, in the constraint $S \subset T$, if we first instantiate T as $\emptyset$, then the instantiation is inconsistent. Generally, instantiation-based solvers may backtrack to try other instantiations if they find instantiations are inconsistent. Backtracking decreases the solver's efficiency on finding solutions.

2.4 Isra in a nutshell

To effectively generate semantically valid computation graphs, we propose an effective random test generation approach, named Isra, including a domain-specific constraint solver with two strategies: graph-level and operation-level constraint resolving, based on our key idea that constraints are able to be simplified with a well-designed instantiation order. Next, we introduce a running example to briefly illustrate how Isra works.

2.4.1 *Graph-level constraint resolving*

Our generation follows a topological order of operations in the computation graph. We say node a precedes node b, or b succeeds a, if there exists a path, where a appears earlier than b in the path. Each time we generate a node, this node does not precede any existing node. For example, as the graph shown in Figure 2, followed by a topological order, i.e., operation `Add` then operation `Concat`, our approach instantiates operations one by one. For each operation, we first instantiate its type and the number of incoming edges.

In this way, our approach resolves the constraints by partitioning them into several subparts. Each subpart corresponds to a single operation and its related edges. Furthermore, because the output of operations could be determined only by the input and attributes, we rewrite the constraints by substituting the output as a function of the input and attributes. In our example, the constraints are as follows, where $\mathrm{Spec}_{\mathrm{op}}(V)$ is defined as a set of constraints on V that specifies from the specification of an operation with type op, f_{op} denotes the mathematical function of the operation with type op:

$$S_1 : \mathrm{Spec}_{\mathrm{Add}}(p,q) \wedge (r = f_{\mathrm{Add}}(p,q)), \tag{7}$$

$$S_2 : \mathrm{Spec}_{\mathrm{Concat}}(\mathrm{axis}, r, s) \wedge (t = f_{\mathrm{Concat}}(\mathrm{axis}, r, s)). \tag{8}$$

After resolving constraints with instantiating operations in the graph by their topological order, our goal turns to instantiate each single operation by solving constraints related to the operation (in our example, they are $\mathrm{Spec}_{\mathrm{Add}}(p,q)$ and $\mathrm{Spec}_{\mathrm{Concat}}(\mathrm{axis}, r, s)$), as shown in the next part.

2.4.2 *Operation-level constraint resolving*

The instantiation of a new operation includes assigning the value to its operation type, attributes, input (incoming edge(s)) (output of the operation is excluded as explained before). As a concrete example, assume we already instantiate a computation graph with a single `Add` operation (with p and q as its input as shown in the red part of Figure 3), now we extend the instantiation by appending a new operation into the computation graph.

Assume we instantiate the type of the new operation as `Concat`, and the number of incoming edges of the new operation as two in our example. We now set symbol variables for the input and attributes of the operation. In the example of `Concat`, we denote variable ax for the attribute axis, and variable x and y as two incoming edges (note, ax, x and y are just symbol variables, which will be assigned with values later, such as assigning r to x).

For each incoming edge, i.e., each tensor in the input, instead of instantiating the whole tensor, we focus on instantiating the structure of the tensor first due to that the semantic specifications are only

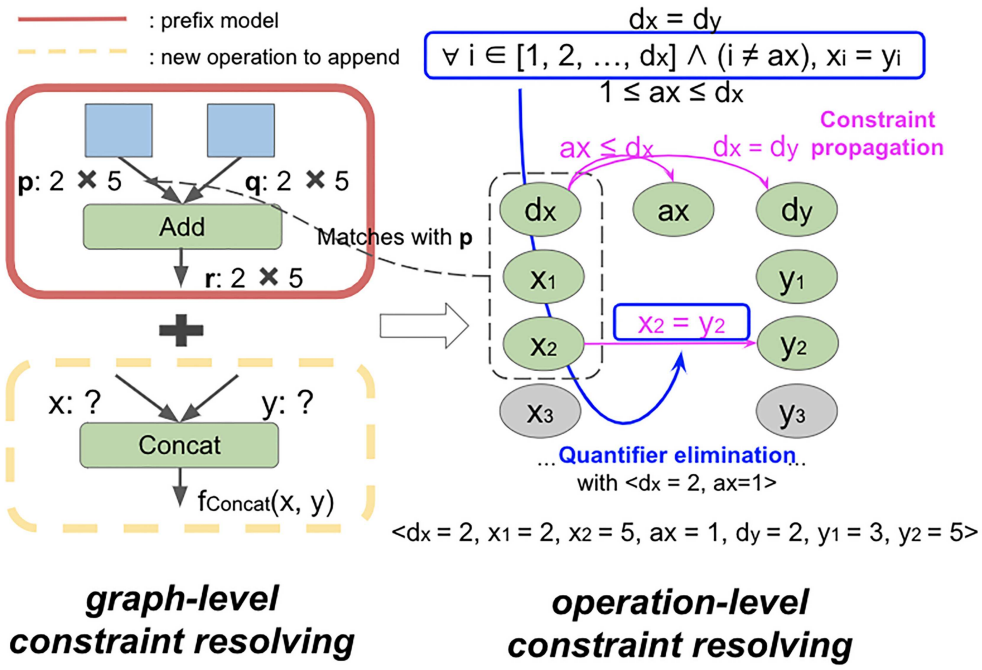

Figure 3 (Color online) Running example of Isra.

related to tensor structure. For an edge x, we setup a set of symbol variables to substitute x in the constraints, including d_x (denoting the dimension size of x, i.e., $\dim_x$) and x_i (denoting each dimension's length of x, i.e., $x[i]$). In this way, the constraints could be further specified as follows:

$$\bigwedge_{k\in\{1,2,3\}} C_k(ax, d_x, d_y, x_i, y_i), \tag{9}$$

$$C_1 : d_x = d_y, \tag{10}$$

$$C_2 : \forall i \in [1, 2, \ldots, d_x] \wedge (i \neq ax),\ x_i = y_i, \tag{11}$$

$$C_3 : 1 \leqslant ax \leqslant d_x. \tag{12}$$

To sample a random solution to the above constraints, our approach instantiates variables in a well-designed order: first are variables related to x; then attribute axis; finally variables related to y. In our example, the order is as follows: d_x; x_i; axis; d_y; y_i. Note, in the order, variables related to the same tensor are ordered together in a group, also, within the group, instantiation of the dimension size (e.g., d_x) is ahead of the length of each dimension (e.g., x_i). The detailed illustration and explanation of the order are shown in Subsection 3.3.

Following this order, we are able to simplify the constraints to quantifier-free and unknown-function-free by the elimination technique mentioned in Subsection 2.3; also controllably choose ways for instantiating unassigned tensors, i.e., instantiating the tensor as an instantiated one such as r for x; or as the output from a tensor placeholder such as s for y. Details are shown in Section 3.

With the above simplification, the constraints belong to propagation logic which is decidable. Thus, we are able to conduct constraint solving by constraint propagation [17] to find solutions. In addition, the constraint propagation will not produce an empty domain (which causes the instantiation inconsistent) due to the good property as explained in the next part, resulting in the overall process being backtrack-free.

2.4.3 *Properties of Isra*

Based on the graph theory and the theory of CSP [17], our instantiation-based solver holds some good properties due to the characteristics of constraints on deep learning models. We draw main conclusions here, formal definitions and detailed explanations are shown in Section 3.

The first property is called graph-level directional consistency. For any semantically valid computation graph with a topological order on operations as $(O_1, O_2, \ldots, O_i)$, we can consistently extend the instan-

tiation to include O_{i+1} (with topological order as $(O_1, O_2, \ldots, O_i, O_{i+1})$) as long as ensuring satisfaction of constraints related to O_{i+1}.

The second property is called operation-level global consistency. For constraints related to each single operation, after determining operation's type and the number of tensors in the input, by taking attributes as a variable and each tensor in the input of the operation as a variable, this CSP, which consists of constraints on the operation level, is globally consistent: any consistent instantiation of a subset of the variables can be extended to a consistent instantiation of all of the variables without backtracking [23].

3 Detailed approach description

3.1 Notations and definitions

3.1.1 *Concepts in the computation graph*

A tensor t is a generalized vector, like N-D array (N is a positive integer). The structure of tensor t is defined as a set Str_t, denoting the structural information of the tensor. Str_t includes (1) a numerical value $\dim_t$ that denotes t's dimension size and (2) a variable-length array that denotes each dimension's length of tensor t (i.e., $t[i]$ as the i-st dimension's length).

An operation n is defined as a function with tensor(s) as input and output. It has some parameters: we denote Op_n as its type ($\mathrm{Op}_n \in \mathrm{AllOps}$, AllOps is a universal set which contains all types of operations), Attr_n as a set which contains attributes of the operation (such as strides in `Conv` operation, and axis in `SoftMax` operation). Also, $\mathrm{Input}_n = \{\mathrm{in}_1, \mathrm{in}_2, \ldots\}$ is a set that contains one or more tensors as the input of n, and $\mathrm{Output}_n = \{\mathrm{out}_1, \mathrm{out}_2, \ldots\}$ as the output of n. Specially, Attr_n contains a special attribute $\mathrm{Indegree}_n$ as the number of tensors in Input_n, i.e., $\mathrm{Indegree}_n = |\mathrm{Input}_n|$.

A tensor placeholder *tp* is simply a variable, denoting a tensor to which the specific data will be assigned later. A tensor placeholder that denotes tensor p could be created with merely Str_p, without the need of specific data.

A computation graph G is defined as an ordered triple (V_G, E_G, ψ_G), where the set V_G denotes the nodes, the set E_G denotes the edges. An element in V_G is either an operation or a tensor placeholder. An element in E_G is a tensor. ψ_G is called an incidence function which maps an edge into a pair of nodes (i.e., a mapping from $E(G) \to V(G) \times V(G)$), denoting the structure of the graph.

3.1.2 *Constraint satisfaction problem*

A constraint C is a limitation placed on the values of variables. Consider a finite sequence of variables $S := \{x_1, x_2, \ldots, x_k\}$, with respective domains $D(x_1)$, ..., $D(x_k)$ associated with them. So each variable x_i ranges over the domain $D(x_i)$. By a constraint C on S we mean a subset of $D(x_1) \times D(x_2) \times \cdots \times D(x_k)$.

An instantiation Q is defined as a set of tuples, written as $\langle x_1 = v_1, x_2 = v_2, \ldots, x_m = v_m \rangle$, denoting that a specific value v_i that has been assigned to the variable x_i.

A CSP $P : (V, D, C)$, where V is a set of variables, D is the set of domains of values for each variable, C is a set of constraints. A solution to a problem P is an instantiation Q, containing the assignment of all variables in V, which satisfies all of constraints in SC. Namely, an instantiation Q is a solution of $P : (V, D, C)$ only if it satisfies with all of constraints in C.

3.1.3 *Local consistency and global consistency*

Let $X = (X_1, X_2, \ldots, X_n)$ be a set of variables in a CSP, and let $X' = (X'_1, X'_2, \ldots, X'_m)$ be a subset of variables from X. A partial instantiation of variables $\langle X'_1 = x_1, X'_2 = x_2, \ldots, X'_m = x_m \rangle$ is locally consistent if it satisfies all the constraints in X'. A globally consistent CSP is one in which any locally consistent partial instantiation of variables can be extended to a consistent full instantiation. Globally consistent CSPs have the property that a solution can be found without backtracking [23].

3.2 Graph-level constraint resolving

Based on the acyclic trait of the computation graph, for any computation graph, there always exists a topological order of operations, i.e., for every two operations x and y in the computation graph, if there is a tensor that is both the output tensor of x and the input of y, then operation x comes before operation y in the ordering.

Our approach works as a top-down way to incrementally instantiate a computation graph by iteratively instantiate a new operation and appending it into the computation graph, as shown in Figure 3. Specifically, we follow the topological order of operations to generate them in the computation graph. When to generate a new operation x, we need to instantiate OP_x, Attr_x and Input_x, with ensuring the satisfaction $\mathrm{Spec}_{\mathrm{OP}_x}(\mathrm{Attr}_x, \mathrm{Input}_x)$ (the same with the definition in Subsection 2.4). We leave the instantiation of edges in Output_x later (when we instantiate another operation y with an edge from x to y) because the value of Output_x could be determined only by OP_x, Attr_x and Input_x, i.e., $\mathrm{Output}_x = f_{\mathrm{OP}_x}(\mathrm{Attr}_x, \mathrm{Input}_x)$.

After finishing the instantiation for current the operation (with its attributes and incoming edges), we iterate the same process for instantiating the next operation if the number of operations in the computation graph has not exceeded to the parameter we set (named numop_G as the number of operations in generated computation graph).

Taking each operation as a variable (a set variable V_x, containing Op_x, Input_x, Attr_x), constraints of the computation graph could be rewritten as a binary CSP, consisting of two kinds of constraints: first are unary constraints on each variable, and second are binary constraints between two variables (i.e., the output of an operation equals to the input of another operation). According to the definition, an instantiation Q is locally consistent on V_x if Q satisfies all the constraints in V_x.

Because our instantiation follows the topological order of a computation graph, with only instantiating edges that are from previous nodes (variables that are ahead of V_x in the order) to the current variable V_x, the instantiation on V_x will not affect local consistency of previous variables (based on the property of topological order). Thus, for each x, as long as we are able to instantiate Q with its local consistency on V_x every time, then we could include V_x in the end of topological order, and instantiation is still consistent. Also, because we instantiate edges with the direction followed by the topological order, the overall graph is loop-free. Thus, following with the topological order, ensuring local consistency on each operation leads to backtrack-free instantiation on the graph level.

3.3 Operation-level constraint resolving

To instantiate a new operation x and append it in the existing computation graph, we need to instantiate (1) OP_x, (2) Attr_x, (3) Input_x (i.e., incoming edges). We take the instantiation of these items into the following steps.

We first determine the OP_x by random sampling from AllOps. Thereafter, the constraints can be specified. To model constraints as a CSP as $P_x(V, D, C)$, we define a variable set V_x which contains all the numerical items from Attr_x and Input_x, i.e., $V_x = \mathrm{Attr}_x \cup \bigcup_{t \in \mathrm{Input}_x} \mathrm{Str}_t$, and also, specify the semantic specification of the operation $\mathrm{Spec}_{\mathrm{OP}_x}(\mathrm{Attr}_x, \mathrm{Input}_x)$ as a set of constraints C on the variables. A specific example is shown in (10)–(12).

To solve the above CSP, based on constraint propagation [17], our approach works as follows: iteratively extending the instantiation by picking an unassigned variable and assigning the value to the variable from its domains; after each turn's instantiation on a variable, with the help of the solver, our approach will conduct constraint propagation among unassigned variables throughout the constraints to reduce the domain of unassigned variables.

During the above process, there are three main issues as follows.

First, the existence of quantifiers and unknown functions in the constraints introduces the difficulty of conducting constraint propagation. For example, in (11): $C_2 : \forall i \in [1, 2, \ldots, d_x] \wedge (i \neq \mathrm{axis}), x_i = y_i$, without instantiating the value of axis, we do not know whether C_2 implies $x_1 = y_1$.

Second, how to instantiate ψ_G (the structure of the graph) during the solving process. A straightforward way (in a 'generate-then-match' style) works as follows: first, instantiate all symbol variables in Input_x, and then matches each tensor in Input_x with instantiated tensors (i.e., outcoming edges of instantiated nodes) by comparing their structures. However, because the domain of the tensors under instantiation is large, the probability of exactly matching instantiated tensors is fairly small. For example, in the example of Section 2, to instantiate tensor x and y (i.e., incoming edges of `Concat`), the number of possible solutions is exploded, which leads to fairly low probability of the equivalence between $\mathrm{Str}_x/\mathrm{Str}_y$ and structures of instantiated tensors (such as Str_p). Thus, this straightforward way will lead to the result that the generated computation graph tends to be simple and scattered.

Third, constraint propagation might produce an empty domain, causing backtracking of the solver (i.e., inconsistence of the instantiation), which affects the effectiveness of constraint solving. For example, for

the constraint $(x = y) \wedge (x = z) \wedge (y \neq z)$, if the instantiation is $\langle x = 1 \rangle$, after constraint propagation, y holds an empty domain.

To address those issues, we tailor a well order that successfully (1) simplifies the constraints to be quantifier-free and unknown-function-free, and (2) enables to controllably select choices for instantiating unassigned tensors; with the guarantee that our propagation-based instantiation is backtrack-free (keeps consistency during the whole process).

In the following parts, we will first describe the order, and then explain our reasons for that, finally, illustrate the property of operation-level constraint resolving in our approach.

3.3.1 *Order of the instantiation*

For an operation x, our order contains several groups, arranged as follows: $G_1, G_2, G_3, \ldots, G_{|\text{Indegree}_x|+2}$. The first group consists of one variable: $G_1 = \{\text{Indegree}_x\}$. The second group is the variables related to the first tensor in the input (first incoming edge of this operation), i.e., $G_2 = \text{Str}_{\text{in}_1}(\text{in}_1 \in \text{Input}_x)$. The third group is the attributes of operations (except for Indegree_x), i.e., $G_3 = \text{Attr}_x \setminus \{\text{Indegree}_x\}$. For the rest, each group is a variable related to the next tensor in the input (next incoming edge of this operation), i.e., $G_{i+2} = \text{Str}_{\text{in}_i}(\text{in}_i \in \text{Input}_x)$. In the groups related to each tensor (assume the tensor is t), the variable that denotes the dimension size of the tensor is ahead of the variables that denote the length of each dimension of the tensor, i.e., d_t (denotes dim_t) is ahead of t_i (denotes $t[i]$).

3.3.2 *Reason I: quantifier and unknown function elimination*

Specifically, there are two forms that cause the existing constraint solving or sampling approaches [11,24] hard to handle. First is the quantifier in the constraints, which hold the forms as $\forall i \in f(x), C(i)$, where f is a function that returns a set, x is a variable, $C(i)$ is a constraint whose form is dependent on the value of i. Second is the unknown function. Constraints may contain terms such as $t_{f(x)}$, where $f(x)$ is a function that returns an integer number and $t_{f(x)}$ is the variable that denotes the $t[f(x)]$ ($f(x)$-st dimension's length of tensor t).

Our instantiation could eliminate the quantifiers and unknown functions in constraints with the following reason. As quantifiers in the form of $\forall i \in f(x), C(i)$, for all of variable v whose existence in $C(i)$ depends on the domain of $f(x)$, we call that v depend on x. As constraints with an unknown function such as $t_{f(x)}$, we call t_i depends on x. For constraints on deep learning models, the above dependencies among constraints are loop-free. Our instantiation order is actually a topological order satisfying those dependencies, i.e., ensuring that for any dependencies that x depends on y, y is ahead of x in the order. Thus, followed by the instantiation order, with satisfying the precondition of eliminating quantifiers and unknown functions by instantiation, we could simplify the constraints to a decidable propositional logic for constraint propagation.

3.3.3 *Reason II: on-demand instantiation*

The reason why we put the variables related to the same tensor in a group, i.e., instantiate the tensor(s) one by one, is due to the consideration of instantiating ψ_G.

We design an on-demand policy for instantiating the tensor, with consideration of the instantiation for ψ_G in the meanwhile. For each tensor t in the input of x ($t \in \text{Input}_x$), our on-demand policy instantiates Str_t with two choices: (1) reusing the structure Str_s of an existing tensor s as long as they are consistent with the instantiation, in other words, we set t's structure the same as an instantiated tensor s, which is from the output of an instantiated node ex, i.e., instantiating $t = s$ with $\psi_G(t) = (ex, x)$ (if there are more than one satisfied tensor, we will randomly pick one of them; if no such tensor, we choose the second way); (2) creating a new tensor placeholder as a node n, and set its output as t, in other words, instantiating a tensor t with $t \in \text{Output}_n$ as well as $\psi_G(t) = (n, x)$.

We select the choice of instantiating tensors according to a Bernoulli distribution, as a common way to produce random boolean decisions. Any other distributions are also allowed. The distribution is controlled by a parameter picking rate. The higher the picking rate is, the higher the chance that our approach would select the first choice (i.e., reusing an existing tensor). To favor generating more complicated computation graphs, we set the picking rate relatively high in practice. We will further explain the effect of this parameter in Section 4.

3.3.4 *Property of operation-level global consistency*

We illustrate that our propagation-based instantiation will not lead to inconsistence due to the property we call operation-level global consistency. For the constraints related to each single operation such as x, after determining the type (OP_x) and the number of tensors in the input ($\mathrm{Indegree}_x$), we could take attributes as a variable and each tensor in the input of the operation as a variable. With good properties of specification on deep learning operations, this CSP is globally consistent [23]. Thus, any order for instantiation, including our delicately-designed instantiation order in our approach, will not lead to inconsistence.

3.4 Properties of Isra

Overall, Isra in our approach is backtrack-free, sound and complete. Based on the property of graph-level directional consistency and operational-level global consistency, we are able to instantiate without backtracking. And also, soundness is guaranteed because the instantiation is always consistent, leading to the satisfaction of final solutions. Completeness is due to that we do not lose the probability of generation of any instantiation during the whole process, in other words, any instantiation that satisfies the constraints has the possibility to be generated.

4 Evaluation

To evaluate the effectiveness of our approach, we compare our approach with five baselines, including three state-of-the-art approaches Muffin [14], TVMFuzz [5], and NNSmith [15]. Also, we evaluate them on three popular real-world deep learning compilers to investigate their bug detection capability. We construct the computation graph based on the ONNX [25] standard. Our implementation is on Python 3, supporting the generation of 65 operations in total [26]. We address the following three research questions with an evaluation conducted on Ubuntu 20.04 of a laptop computer with Intel® Core™ i5-1135G7 CPU @ 2.40 GHz and memory of 8 GB. More details are included on our project website [26].

RQ1: Is Isra effective for generating test inputs for testing deep learning compilers?

RQ2: How effective and practical are the generated tests in revealing bugs in popular real-world deep learning compilers?

RQ3: Does our approach outperform state-of-the-art approaches in terms of testing deep learning compilers in terms of coverage and bug detection capability?

4.1 Compared work

To assess the effectiveness of Isra, we first design and implement two baselines as the representative of another two types of test generation techniques, named DeclGen and Randoop-Gen. In addition, we also compare Isra with three state-of-the-art approaches that can be applied to testing deep learning compilers. More specifically, we include the following representative techniques in our evaluation.

4.1.1 *DeclGen*

Declarative-style generation constructs deep learning models only based on the syntax grammar, in short as DeclGen. When determining the shape of tensors, it just randomly generates from all choices. After the construction of the input, i.e., a whole computation graph, this approach directly feeds input into the compiler under test, and relies on the compiler's running to check whether the model is satisfied with its semantic specifications.

4.1.2 *Randoop-like generation*

Inspired by feedback-directed random test generation [13], this approach conducts random generation for operation construction, i.e., randomly constructs a new operation to append it into the model, and checks whether the model satisfies the semantic specifications or not. This generation way can avoid generating invalid models at early stages, leading to the improvement of overall effectiveness, while the generation for a single operation to satisfy its semantic specifications is still ineffective.

4.1.3 *Muffin*

Muffin [14] is a state-of-the-art generation-based testing approach, proposed originally to test deep learning libraries such as Keras [27], generating models based on two model structure templates (chain structure with skips and cell-based structure) with deep learning library APIs. To satisfy with tensor structural constraints, Muffin hardcodes additional `Reshaping` layers to reshape the input/output tensors for the connection between layers. Though Muffin is not designed for testing deep learning compilers, we can retrofit it to barely achieve the goal by converting the models constructed by high-level APIs into computation graphs with existing tools [28].

4.1.4 *TVMFuzz*

TVMFuzz [5] is a fuzzer which specifically targets testing TVM [2]. It randomly generates and mutates tensor-level IR (TIR) expressions (the intermediary language of TVM) for fuzzing test, mainly towards the type-related bug detection.

4.1.5 *NNSmith*

NNSmith [15] is a generation-based approach for testing deep learning compilers, as a parallel research work with ours. NNSmith constructs the computation graphs based on the specifications of deep learning models: it tries possible choices for types, attributes, and input/output of operations by iteratively adding constraints from specifications and leveraging existing constraint solver Z3 [24] to check the satisfaction; if constraints are not satisfied, it will backtrack and try different choices to pick.

4.2 Study subjects and settings

We choose TVM, Glow (two popular open-source compilers), and SophGo (a state-of-practice commercial compiler) as our study subjects. For TVM and Glow, we download their officially released versions from GitHub: TVM as v0.7 (commit id 728b829), Glow[4)] (commit id a2036bd). For SophGo, we attain its latest released version from the company that developed it.

For test oracles, we use two types of oracles: (1) runtime failure (including error/crash behaviors) of compilers, i.e., when a computation graph causes the exception of compilers (excluding invalid inputs that violate the specifications); (2) differential testing by feeding the same random inputs and comparing the outputs of compiled models from different compilers. In differential testing, we set the relative error as 10% (we set this value relatively large in order to avoid many false positives) for automatically comparing results from different compilers under the same input.

For Isra, we set the upper bound on the tensor dimension as 5, and the upper bound on the length of each dimension as 5 (which is aligned with the default settings of Muffin). For picking rate, we set it with 0.97 based on our sensitivity experiments. Except for the above parameters which remain the same among all of the experiments, we set the lower bound and upper bound of operation number in the graphs (named lb and ub) according to the experiments, the numop_G is uniform sampling between lb and ub.

For Muffin, we obey its default settings. For alignment with comparisons on coverage metrics, we convert the Keras models generated by Muffin to ONNX format with TF2ONNX [28]. Because Keras API is more higher level than ONNX, the converting leads to the growth of model size. For fairness, in the experiment on coverage measurement, for every model Muffin generated, we adopt Isra to generate a model with the number of operations same as Muffin, named Isra*. Also, we ensure that both approaches have a same operation set (of size 36, which is the number of overlapped operations).

For TVMFuzz, we obey its default parameters. Due to the difference on the type of generated inputs (Isra generates computation graphs, while TVMFuzz generates programs in TIR, an IR for the TVM compiler), the metrics that we design for the work in this paper are not applicable for TVMFuzz. So we are unable to measure our coverage metrics on TVMFuzz. TVMFuzz is compiler-dependent, so we are unable to test TVMFuzz on compilers except for TVM.

For NNSmith, we obey its default parameters. For fairness, in the experiment on coverage measurement, we align the settings of Isra with those from NNSmith, named Isra**. For every model NNSmith generated, Isra** generates a model with the same number of operations. Also, we ensure that both approaches have a same operation set (of size 40, which is the number of overlapped operations).

4) Glow does not have a release version on GitHub. Thus we directly download its latest version.

For the experiment on coverage measurement, we run each approach to generate a fixed number of models, such as 10000. For the parameters of operation number in the graphs, we set lb as 1 and ub as 200 for Isra as well as DeclGen and Randoop-Gen in the experiment. To eliminate randomness, we run our approach and baselines separately for five times and calculate the average number. According to our results of the experiments, the coverage metrics of the three approaches are all saturated or converged, so it is a reasonable setting for our evaluations.

For investigating bug detection capability, aligned with the setting in [14], for each method, we generate 300 computation graphs for testing compilers. Due to the difference of supported operations for each compiler, we run the generators with filtering the generated graphs that contain unsupported operations until the graph number reaches 300. To reduce manual work on deduplication, we set the generated model relatively small with lb as 1 and ub as 10.

To save manual effort for checking duplicated bugs, we automatically check and filter the bugs with the same error messages and stack traces as produced before. For the rest of the bugs, the two authors manually check their root causes through error messages, stack traces, and generated inputs to further eliminate duplicated bugs and false positives.

4.3 Metrics

In order to evaluate the effectiveness and practicability of different approaches, we investigate on their bug-detection capability, by counting the number of distinct bugs they detect within a fixed number of test inputs when used to test real-world deep learning compilers.

Furthermore, to better measure various aspects of generated inputs, we inherit the coverage criteria from previous research work [29] and expand upon them to measure the diversity of computation graphs, including types, tensor shapes, and attributes of operations as well as connections between operations. The design of these coverage criteria is motivated by the fact that (1) existing traditional code coverage criteria are ineffective in deep learning testing [29], and neuron coverage criteria [30] are also invalid in our evaluation because compiler testing involves different input models, and (2) type and shape problems are major root causes of deep learning compiler bugs as demonstrated in a recent study [5], also, (3) a lot of high-level optimizations in deep learning compilers, which are error-prone due to continuous and frequent development and modifications [5, 31], could only be triggered by different types of specific connections between operations in the computation graphs [32].

Specifically, we design 11 types of metrics for measuring coverage among input space. The metrics are of two major categories: graph-level metrics and operation-level metrics. For operation-level metrics, we mainly follow the work by [29]. For graph-level metrics, we design them with analogy of concepts in structural code coverage and combinatorial testing. Besides the preceding metrics, we also count the frequency of occurrences for operations and calculate the distribution of operations.

4.3.1 *Graph-level metrics*

Graph-level metrics are designed for measuring various aspects of a single generated model. Let a test set be the set of models generated by an approach. Given a test set I, which contains n_I graphs, the graph-level metrics are defined as follows.

Number of operations (NOO) of a graph g is defined as the total number of operations in g. Number of operation types (NOT) of a graph g is defined as the number of different types of operations in g. Number of operation pairs (NOP) of a graph g is defined as the number of different edges in g that connect two operations together. Number of operation triples (NTR) of a graph g is defined as the number of different pairs of adjacent edges that connect three operations together in g. Number of shapes and attributes (NSA) of a graph g is defined as the total number of different shapes of input tensors and attributes (except for its input degrees) for each operation in graph g. These graph-level metrics for test set I are defined as the average of each of them among all the graphs in I: $\mathrm{GLM}(I) = \frac{\Sigma_g \mathrm{GLM}_g(g)}{n_I}$, where $\mathrm{GLM} \in \{\mathrm{NOO}, \mathrm{NOT}, \mathrm{NOP}, \mathrm{NTR}, \mathrm{NSA}\}$.

4.3.2 *Operation-level metrics*

Operation-level metrics are designed for measuring various aspects of operations among the whole test set. An operation corpus is a set of operations with their attributes including the operation name and the possible number of different input degrees. Given an operation corpus C containing n_C different

Table 1 Results on graph-level and operation-level metrics. The best results are in bold.

	Isra	DeclGen	Randoop-Gen	Isra*	Muffin	Isra**	NNSmith
Time (s)	**320.1713**	9011.9172	4721.8233	**20.1031**	25847.6804	**111.8615**	880.7737
OTC (%)	**100**	97.536	98.46	**100**	**100**	**100**	**100**
IDC (%)	**92.95**	90.178	89.966	**91.85**	88.52	**89.54**	88.71
ODC	**11.848**	4.928	10.616	**8.75**	4.22	**6.925**	6.225
SEC (%)	**98.27**	67.804	88.08	**98.15**	35.49	**99.38**	78.50
DEC (%)	**90.208**	2.126	45.324	**57.7**	4.95	**84.30**	28.70
SPC	**3001.938**	227.018	1509.356	**1192.22**	556.44	**2822.9**	1641.725
NOO	**100.8766**	2.8783	100.1231	**10.4236**	**10.4236**	**16.3999**	**16.3999**
NOT	**45.237**	2.76	33.0021	**8.6589**	5.3289	**13.1333**	10.5113
NOP	**103.7621**	1.4856	98.6460	**7.8243**	6.399	**14.5851**	10.4880
NTR	102.9130	0.6042	**105.5774**	4.6481	**6.0766**	**13.065**	7.6740
NSA	**26.6252**	1.5533	10.0211	5.9253	**11.3604**	**10.8192**	10.1459

operations and a test set I, we first calculate the metric on each type of operator o in I, denoted as $\mathrm{XXC}_{\mathrm{op}}(o)$, then we have the operation-level metric of I as the average of the operation-level metric on different operators, i.e., $\mathrm{XXC}(I) = \frac{\Sigma_o \mathrm{XXC}_{\mathrm{op}}(o)}{n_C}$, where $\mathrm{XXC} \in \{\mathrm{OTC}, \mathrm{IDC}, \mathrm{ODC}, \mathrm{SEC}, \mathrm{DEC}, \mathrm{SAC}\}$. We simply explain the meanings of these six metrics as below, and detailed definitions of these operation-level metrics are shown in our project website [26].

Operation type coverage (OTC), input degree coverage (IDC), output degree coverage (ODC) show the diversity of operations types, and the diversity of the input and output degrees of them in the test set I respectively. Single edge coverage (SEC) shows the diversity of edges between the operations in I. Double edge coverage (DEC) shows the diversity of pairs of edges that are adjacent, which is actually the coverage of different triples of operations that are connected in a graph in the test set. Shapes and attributes coverage (SAC) indicates the diversity of attributes of the operations (except for input degrees) and their input tensor shapes in the test set.

4.4 RQ1: evaluation results of generated inputs

4.4.1 *Operation-level metrics*

The result of the experiment on coverage measurement is shown in Table 1. With alignment on the number of generated inputs, we can find that Isra outperforms the baselines greatly with respect to the amount of time, showing the efficiency of our approach.

For operation-level metrics, we find that Isra is able to cover all kinds of operations that we have chosen and all kinds of input degrees of each type of operation. Compared with the two baselines, Isra has higher coverage on all of operation-level metrics, especially for SEC, DEC, and SAC.

4.4.2 *Graph-level metrics*

We find that the NOO of our approach and Randoop-Gen are closer to the average of the lower bound and upper bound of the operation numbers that we set (consistent with our uniform sampling for the number of operations), while DeclGen's NOO remains at a significantly lower level. The reason is that DeclGen holds less probability to satisfy the semantic specifications, which leads to generating rather simple models and bad performance on the graph-level metrics. Randoop-Gen's NOP and NTR are comparable with our approach, however, the operation types (NOT) of Isra are more diverse, making it outperform Randoop-Gen at coverage of operation pairs (SEC) and triples (DEC). The results of graph-level metrics indicate that Isra are capable of generating diverse, large and complex models.

4.4.3 *Distribution of operation frequency*

As shown in Figures 4–6, Isra is able to generate different operations in a relatively uniform distribution, which is consistent with our uniform sampling for picking the type of operation. In contrast, all of baselines fail to generate a sufficient number of operations with relatively complicated semantic specifications such as `Conv` and `Gemm`. It shows that all of baselines have a limitation that the diversity of models generated by them is weak. This is because the constraints of some operations are relatively complicated and less possible to satisfy. Those complex operations are less possible to be chosen in the valid output by the

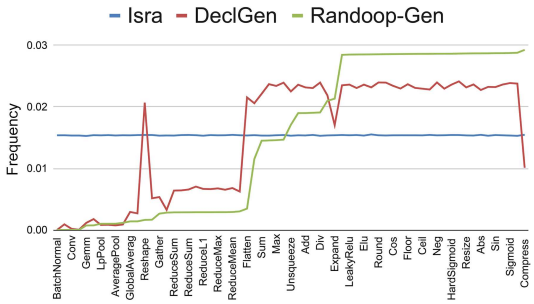

Figure 4 (Color online) Distribution of operation frequency of Isra, DeclGen, and Randoop-Gen (captioning only parts of operations in x-axis for a clear presentation, numbers at y-axis are normalized with the total number).

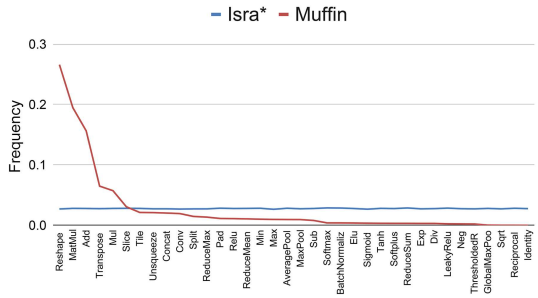

Figure 5 (Color online) Distribution of operation frequency of Isra* and Muffin.

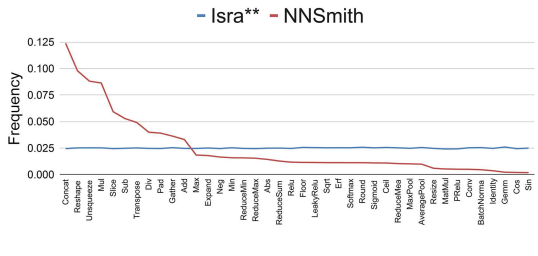

Figure 6 (Color online) Distribution of operation frequency of Isra** and NNSmith.

Table 2 Results of sensitivity experiment of picking rate. The best results are in bold.

Picking rate	Time (s)	NOP	NTR	NSA
0.5	**215.29**	51.07	23.73	**62.86**
0.8	260.47	82.91	63.99	42.05
0.9	269.98	94.34	83.30	33.84
0.95	270.01	100.17	94.19	29.54
0.97	**267.24**	**102.56**	**98.77**	**27.77**
0.99	266.36	**104.94**	**103.57**	25.93

filter of the iterative checking process. Besides, among models generated by Muffin, `reshape` operation appears most frequently because Muffin forces the insertions of `reshape` operation before many operations to ensure semantic specifications.

4.4.4 *Picking rate*

To evaluate the effect of the picking rate parameter, we also compare the results of Isra with the setting of different picking rates. The settings are the same as the experiment on coverage measurement, except that we set the operation number in the graphs as a fixed number of 100. The result is shown in Table 2. If the picking rate is relatively high, the operations are more likely to be matched with existing tensors, leading to NOP, NTR going high. In the meanwhile, the shape of newly created tensors is more likely to be equal to the shape of previous tensors, leading to the result that NTR goes down as the picking rate increases. We finally choose its value as 0.97, as a trade-off.

4.5 RQ2: evalution results on bug-detection capability

We investigate the effectiveness of Isra and baselines in terms of distinct bugs detected in the same version of compiler subjects. After analysis of deduplication, Isra detects 33 unique bugs in total, as shown in Table 3, performing better or as well than baselines on all of three benchmarks. The capability on bug detection shows the high effectiveness of Isra. To further evaluate bug-detection capability of Isra, we conduct a study on the bugs Isra detected.

Table 3 Number of unique bugs detected on different compilers. The number of NNSmith's detected unique bugs on TVM is different from the results in the NNSmith's paper [15] due to the different settings on the compiler versions and testing amount.

	Isra	DeclGen	Randoop-Gen	Muffin	TVMFuzz	NNSmith
TVM	18	8	14	12	5	21
Glow	4	2	2	2	–	4
SophGo	11	3	3	9	–	7
Total	33	13	19	23	5	32

4.5.1 *Bug study*

Among all of the unique bugs detected by Isra, we categorize them into two types based on the test oracles detecting them. (1) Error bug (29 of 33): the given input triggers an error, crash or unexpected exception during compilation. (2) Inconsistency bug (4 of 33): the compiler executes the given input and terminates normally but the compiled model produces different outputs according to differential testing.

After we report found bugs to the corresponding community, compiler developers give responses for most of them, all with positive feedback. Among all of the bugs found by Isra, there are 24 previously unknown bugs. Until now, a majority of our detected bugs (27 of 33) have already been fixed by developers, with 16 bugs directly owing to our bug reporting/pull requests, benefiting all the compiler users and their applications. One of TVM core developers replies with the following message for bugs reported by us[5]: "These two errors that you generated were excellent real bugs with the importer and were very easy to understand and replicate with your post. If they are being auto-generated they look excellent!" The feedback from real-world developers is also strong evidence showing that Isra is practical for testing real-world deep learning compilers and able to detect critical bugs in practice.

We list 24 previously unknown bugs (as well as bug reports) detected by Isra in Table 4, including the stage of root causes, the time it takes to have the bug confirmed and fixed, and the GitHub issue ID[6] for reference. The details of confirmed/fixed bugs are as follows.

TVM-1 is a bug in the compiler backend, caused by a pass named `DynamicToStatic` which should not be defined at optimization level 3. It will lead the internal error when the deep learning model contains operations such as `MatMul`, `Dropout`. After our reporting, developers reordered passes in the backend and lowered `DynamicToStatic` optimization level to fix it[7].

TVM-2 is a bug in the compiler frontend. The TVM developer has explained the cause of the bug and fixed it[8]: "It is due to a bad assumption made in `PRelu` conversion about the input layout ... Our current PRelu converter assumes that incoming data is in NCHW format and that the slope will have C total elements. Neither of these are actual requirements for ONNX `PReLu`."

TVM-3 and TVM-4 are bugs in the compiler frontend. Some of the parameters in `LpPool` and `LeakyRelu` operation in ONNX standard allow default value but it was not supported in TVM.

TVM-5 is a bug in the compiler backend, which catches an edge case of the compiler's type checker as explained by a TVM developer[9].

TVM-6 and TVM-7 are bugs in the compiler frontend. The former is due to that one of the input tensors in `Gemm` operation can be optional by the standard but the TVM does not allow that behavior. The latter is due to the fact that "`Split` operation is not dynamic inputs" as explained by the compiler developer.

TVM-8 is a bug in the compiler backend due to inconsistent specifications between different versions on `Squeeze` operation, causing erroneous implementation in the compiler.

TVM-9 is a bug in the compiler backend. The erroneous backend implementation causes the inconsistent outputs on a model containing `Flatten` and `ReduceL1` operation. Developers have not confirmed this bug, but it has been fixed in the next released version of the compiler.

Glow-1 is a bug in the compiler backend due to that `ReduceSum` operation in the input model contains multi axis inputs and attributes.

SophGo-1 is a bug in the compiler frontend. The compiler does not support that the weight tensor of `Conv` operation is an input tensor. Developers do not fix this bug due to the reason that they suppose

5) https://github.com/apache/tvm/pull/7208#issuecomment-754406762.

6) For reference, the URLs of TVM's and Glow's bug reports are https://github.com/apache/tvm/issues/#IssueID, and https://github.com/pytorch/glow/issues/#IssueID. The URLs for SophGo's bug reports are internal and not publicly accessible.

7) https://github.com/apache/tvm/pull/7213.

8) https://github.com/apache/tvm/issues/7202#issuecomment-754372403.

9) https://github.com/apache/tvm/issues/7262#issuecomment-911968074.

Table 4 Previously unknown bugs detected by Isra.

ID	Type	Location	Confirmed	Fixed	Issue ID
TVM-1	Error	Backend	<1 day	10 days	7200
TVM-2	Error	Frontend	<1 day	<1 day	7202
TVM-3	Error	Frontend	<1 day	6 months	7241
TVM-4	Error	Frontend	<1 day	<1 day	7244
TVM-5	Error	Backend	7 months	7 months	7262
TVM-6	Error	Frontend	42 days	42 days	7263
TVM-7	Error	Frontend	<2 days	10 months	8889
TVM-8	Error	Backend	<2 days	10 months	8890
TVM-9	Inconsistency	Backend	–	10 months	7270
TVM-10	Inconsistency	Unknown	–	–	12734
TVM-11	Error	Unknown	–	–	12735
Glow-1	Error	Backend	13 days	–	5995
Glow-2	Error	Unknown	–	–	5991
SophGo-1	Error	Frontend	<3 days	–	*
SophGo-2	Error	Backend	<3 days	<7 days	*
SophGo-3	Error	Backend	<3 days	<7 days	*
SophGo-4	Error	Frontend	<3 days	<7 days	*
SophGo-5	Error	Backend	<3 days	<7 days	*
SophGo-6	Error	Backend	<3 days	<7 days	*
SophGo-7	Inconsistency	Backend	<3 days	<7 days	*
SophGo-8	Error	Frontend	<3 days	<7 days	*
SophGo-9	Error	Frontend	<3 days	<7 days	*
SophGo-10	Error	Frontend	<3 days	–	*
SophGo-11	Error	Backend	<3 days	<7 days	*

users take this parameter as a constant because this parameter usually does not change after deployment.

SophGo-2 is a bug in the compiler backend. If the inputs of `Mean` operation are the same tensor, then the calculation of the mean operation will throw an exception due to the naming error.

SophGo-3 is a bug in the compiler backend. `ReduceProd` operation will register a buffer, but buffer size has not been assigned which leads to the incorrect parameters in calculation and further causes the final result wrong.

SophGo-4 is a bug in the compiler frontend due to erroneous parsing for `Pooling` operation.

SophGo-5 is a bug in the compiler backend due to incorrect implementation on `Split` operation.

SophGo-6 is a bug in the compiler backend. The compiler assumes the second input of `PRelu` holds a specific shape that is not consistent with the specification.

SophGo-7 is a bug in the compiler backend. The compiler backend incorrectly covers the input data for the calculation on `Cos` operation, causing the wrong results of the model's output.

SophGo-8, SophGo-9, and SophGo-10 are bugs in the compiler frontend due to erroneous parsing for `Unsqueeze` operation, `Split` operation and `Gemm` operation. ONNX standard of some operations has changed after version 13. The compiler only implements the old version and is not compatible with the latest standard. Developers do not fix SophGo-10 due to the same reason as SophGo-1.

SophGo-11 is a bug in the compiler backend due to incorrect implementation on `Resize` operation.

4.5.2 *False positives*

Though our test generation approach is sound (i.e., the models generated by our approach ensure the validity), false positives may be introduced by floating point errors [33,34]. For differential testing, one of the test oracles we used, when it comes to checking output equivalence, false alarms may be produced due to floating point errors/inaccuracies. To reduce false alarms, we use a high error tolerance in comparison of models' outputs (the relative error is set to 10% as mentioned in Subsection 4.2), which is similar to a parallel work [15]. In our evaluation, Isra did not find false positives caused by the floating-point errors.

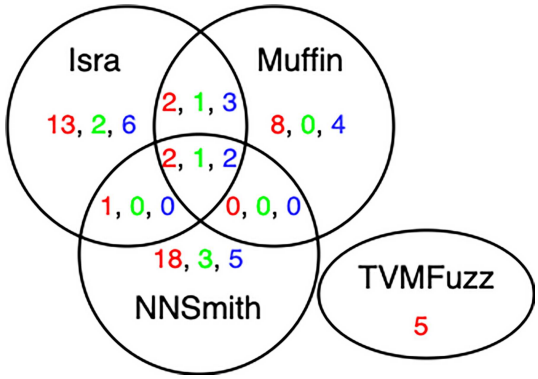

Figure 7 (Color online) Overlaps of detected bugs among different approaches on testing three compilers (red for TVM, green for Glow, blue for SophGo).

4.6 RQ3: comparison with state-of-the-art approaches

We compare Isra with three state-of-the-art approaches (i.e., Muffin [14], TVMFuzz [5] and NNSmith [15]) in terms of coverage and bug detection capability.

For the comparison on coverage, as result shown in Table 1, (1) under the same settings, Isra* outperforms Muffin on most of the coverage metrics except for only NTR and NTR, and (2) under the same settings, Isra** consistently outperforms NNSmith on all of coverage metrics. The reasons why Muffin achieves higher NTR and NTR than Isra* are as follows. For NTR, this is mainly because Muffin contains a cells mode, which will favor generating a dense graph structure that contains many triples, but its DEC is still significantly low compared with Isra*, due to the fewer types of operations in per graph (NOT). Muffin's higher coverage on NTR is because Muffin inserts `Reshape` layers between adjacent layers for patching tensor structural constraints in a hardcode way, which leads to changing tensor shape frequently. However, Muffin is still significantly lower than Isra on SAC, which is the coverage of NTR among the whole test set.

For the comparison on bug detection capability, as shown in Table 3, Isra detects more bugs than Muffin on three compilers, with 1.5x, 2x, 1.22x respectively, in total 33 versus 23. Also, Isra significantly outperforms TVMFuzz on detecting bugs on TVM with 18 versus 5. Isra achieves comparable results compared to NNSmith (33 versus 32) on three compilers. In addition, we also investigate the overlaps of detected bugs among Isra, Muffin, TVMFuzz, and NNSmith as shown in Figure 7. There are overlapping bugs among them as well as distinct non-overlapping bugs, indicating that these approaches are complementary. There is no overlapping bug between TVMFuzz and other approaches, primarily because TVMFuzz fuzzes on low-level Relay IR instead of computational graphs.

Overall, as shown in evaluation results, compared to state-of-the-art approaches under the same settings, Isra outperforms those approaches on various coverage metrics, and also achieves comparable and complementary results on bug detection. The results indicate that (1) Isra is more effective and more efficient than existing approaches for test generation, (2) Isra is effective and complementary to existing approaches for detecting bugs in real-world compilers.

5 Related work

Random testing and test generation. Random testing [35] simply constructs test inputs in a random manner. For example, Randoop [13] and EvoSuite [36] aim to generate JUnit tests for classes by incrementally combining sequences of method calls. Besides many aspects of advantages, random testing still faces problems including inefficiency, imprecision, and lack of available information to guide test generation. To test deep learning compilers, our work conducts random testing by enhancing the effectiveness of test generation. Another test generation way is bounded-exhaustive testing. For example, UDITA [37] uses bounded-exhaustive testing to enumerate the paths through the generator with various optimizations. For deep learning models, the space of the computation graph and the shape of tensors in it can be super large, and the valid space is very sparse; thus, it is intolerable to enumerate all kinds of inputs by searching.

Grammar-based fuzzing. Fuzzing is a common approach for randomly generating inputs to test software. It may generate inputs from scratch, or do mutation on a series of valid seed inputs. Without any knowledge of the input of the software under test, generating valid inputs randomly is ineffective, especially for the software such as compilers whose inputs are highly-structured. To improve it, grammar-based fuzzing [38,39] is proposed, which relies on a grammar specification to generate structured inputs, usually in context-free forms. Deep learning models with semantic specifications fail to be represented as a context-free grammar. Recently Padhye et al. [40] proposed Zest, which is based on coverage-guided fuzzing, targeting at producing semantically valid inputs. However, Zest still requires developers to manually design a generator that can construct syntactically valid test programs. Different implementations for the generator could highly affect the effectiveness of test generation, especially for languages with complicated specifications such as deep learning models.

Testing deep learning toolkits. Deep learning toolkits include deep learning libraries (frameworks) and deep learning compilers. The differences between them lie in their primary functions and interfaces provided to users, which result in divergent emphases in designing corresponding testing approaches.

Deep learning libraries, such as Keras and TensorFlow, are primarily used for simplifying the implementation of deep learning models, they provide high-level APIs and abstractions of pre-defined layers/models as well as optimizers, loss functions and other utilities that allow users to easily define and train deep learning models. To test deep learning libraries, LEMON [41] and Muffin [14] focus on generating parameters and call sequences of high-level APIs.

Deep learning compilers, such as TVM, are designed to transform deep learning models into efficient low-level code for deployment and execution on different hardware devices, and they focus on optimizing the computational graph of the models to improve execution efficiency. To test deep learning compilers, one direction is to directly generate inputs of deep learning compilers, i.e., computation graphs, including research work GraphFuzzer [29] and MT-DLComp [34]; another direction is to fuzz low-level intermediate representation of the compiler (e.g., TVM's compiler-specific intermediate representation), including research work TVMFuzz [5] and Tzer [42]. Our approach, Isra, as well as its two parallel works NNSmith [15] and HirGen [32] belong to the former, i.e., test generation of computation graphs.

To generate test inputs for deep learning toolkits, the validity of test inputs is a critical challenge: invalid test inputs will largely diminish the effectiveness and efficiency of testing. To address it, different techniques are proposed by existing research work. For example, Muffin [14], GraphFuzzer [29] and HirGen [32] are all restricted to certain types of operations and connections for generating computation graphs, which will bias the generated graphs. Specifically, Muffin [14] ensures the semantic specification by inserting the reshaped layers between adjacent layers in origin models. Unfortunately, doing so biases the generated computation graphs to include many `Reshape` operations as shown in our evaluation. GraphFuzzer [29] and HirGen [32] try to adjust mismatched tensor shapes through slicing and padding. They will also bias the generated computation graphs to include many `Slice` and `Padding` operations.

NNSmith [15], as a parallel work with Isra, addresses the validity challenge by leveraging the existing constraint solver Z3 [24]. However, it leads to a further problem which is a lack of diversity due to that existing constraint solvers tend to pick boundary values for constraints. To relieve this problem, NNSmith tries to iteratively add extra constraints (named "attribute binning" [15]). When extra constraints produce an unsatisfiable one, NNSmith will randomly drop some of the constraints and retry, until it succeeds. It results in following disadvantages: (1) extra overhead for constraint solving due to the iteratively retrying; (2) a biased distribution of operations and parameters in the generated models, the models tend to contain more "simple" operations such as `Add` and `Sub`, and less "complicated" operations such as `Conv` and `Gemm` (as seen in both of the results in NNSmith's paper [15] and our evaluation).

Compared with related work, our test generation approach overcomes the validity challenge by the domain-specific constraint solver proposed in this paper. It offers several advantages as follows.

- The computation graphs generated by our approach are more diverse because our domain-specific constraint solver can sample diverse operations/parameters without bias, as evidenced in our evaluation.
- Our approach is more efficient due to lower computational costs compared to other solutions such as repeatedly calling the external constraint solver (as NNSmith did), as evidenced in our evaluation.
- Our approach is more scalable because our domain-specific constraint solver is lightweight and backtrack-free, without inherent limitations on the type and the size of generated computation graphs, which is potentially beneficial for other scenarios such as generating extreme test cases for stress testing.

Besides test generation for valid computation graphs, existing research work also proposes other techniques to enhance the effectiveness of deep learning compiler testing, which are orthometric to our ap-

proach. NNSmith [15] conducts value searching for improving numeric validity with gradients. HirGen [32] proposes "disruptive generation" to generate computation graphs containing obvious breaks of specifications for detecting incorrect exception handling bugs. In addition, mutation-based approaches such as TVMFuzz [5] and Tzer [42], conduct a series of heuristic-based mutation rules on seed inputs (i.e., existing models) at the compiler's low-level intermediate representation. Our approach is complementary to these techniques.

6 Conclusion

In this paper, to construct diverse and semantically valid computation graphs for testing deep learning compilers, we proposed a new approach named Isra, including a novel domain-specific solver for effectively resolving constraints on computation graphs. We have implemented and evaluated our approach against five baselines, and also applied Isra to test three real-world deep learning compilers. The evaluation results show that (1) Isra outperforms the baselines including two state-of-the-art approaches (Muffin [14] and NNSmith [15]) on coverage metrics, demonstrating Isra's effectiveness in generating diverse computation graphs; (2) Isra performs better or as well than state-of-the-art approaches on bug detection, the result of Isra is also complementary to those from existing approaches; (3) Isra detects 24 previously unknown bugs in the released versions of the three compilers, demonstrating its high value in practice.

Acknowledgements This work was partially supported by National Natural Science Foundation of China (Grant No. 62161146003) and Tencent Foundation/XPLORER PRIZE. We would like to thank Haiyue MA, Zhengkai WU, and Chenxi LI for their help in improving the presentation of this work.

References

1 Jouppi N P, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, Toronto, 2017. 1–12
2 Chen T, Moreau T, Jiang Z, et al. TVM: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, 2018. 578–594
3 Rotem N, Fix J, Abdulrasool S, et al. Glow: graph lowering compiler techniques for neural networks. 2018. ArXiv:1805.00907
4 Leary C, Wang T. Xla—TensorFlow, compiled. 2017. https://www.tensorflow.org/
5 Shen Q, Ma H, Chen J, et al. A comprehensive study of deep learning compiler bugs. In: Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, 2021. 968–980
6 Chen J, Patra J, Pradel M, et al. A survey of compiler testing. ACM Comput Surv, 2021, 53: 1–36
7 Boyapati C, Khurshid S, Marinov D. Korat: automated testing based on java predicates. SIGSOFT Softw Eng Notes, 2002, 27: 123–133
8 Elkarablieh B, Marinov D, Khurshid S. Efficient solving of structural constraints. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, Seattle, 2008. 39–50
9 Gligoric M, Gvero T, Lauterburg S, et al. Optimizing generation of object graphs in Java Pathfinder. In: Proceedings of the 2nd International Conference on Software Testing Verification and Validation, Denver, 2009. 51–60
10 Turing A M. On computable numbers, with an application to the entscheidungsproblem. Proc London Math Soc, 1937, s2-42: 230–265
11 Dutra R, Bachrach J, Sen K. SMTSampler: efficient stimulus generation from complex SMT constraints. In: Proceedings of the International Conference on Computer-Aided Design, San Diego, 2018
12 Grädel E, Kolaitis P G, Libkin L, et al. Finite Model Theory and its Applications. Berlin: Springer, 2007
13 Pacheco C, Lahiri S K, Ernst M D, et al. Feedback-directed random test generation. In: Proceedings of the 29th International Conference on Software Engineering (ICSE'07), 2007. 75–84
14 Gu J, Luo X, Zhou Y, et al. Muffin: testing deep learning libraries via neural architecture fuzzing. In: Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, 2022. 1418–1430
15 Liu J, Lin J, Ruffy F, et al. NNSmith: generating diverse and valid test cases for deep learning compilers. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2023. 530–543
16 Reynolds A, Tinelli C, Goel A, et al. Quantifier instantiation techniques for finite model finding in SMT. In: Proceedings of the 24th International Conference on Automated Deduction, Lake Placid, 2013. 377–391
17 Apt K R. Principles of Constraint Programming. Cambridge: Cambridge University Press, 2003
18 Kumar V. Algorithms for constraint-satisfaction problems: a survey. AI Mag, 1992, 13: 32
19 Prosser P. Hybrid algorithms for the constraint satisfaction problem. Comput Intell, 1993, 9: 268–299
20 Reynolds A, Tinelli C, de Moura L. Finding conflicting instances of quantified formulas in SMT. In: Proceedings of the Formal Methods in Computer-Aided Design (FMCAD), 2014. 195–202
21 Reynolds A, Tinelli C, Goel A, et al. Quantifier instantiation techniques for finite model finding in SMT. In: Proceedings of International Conference on Automated Deduction, 2013. 377–391
22 de Moura L M. Bjørner N S. Efficient E-matching for SMT solvers. In: Proceedings of the 21st International Conference on Automated Deduction, Bremen, 2007. 183–198
23 Dechter R. From local to global consistency. Artif Intell, 1992, 55: 87–107
24 de Moura L M, Bjørner N. Z3: an efficient SMT solver. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Budapest, 2008. 337–340
25 Microsoft. ONNX Github repository. 2017. https://github.com/microsoft/onnxruntime
26 Ren L. Isra project repository. 2022. https://github.com/israproj/isra
27 Keras Team. Keras project repository. 2022. https://github.com/keras-team/keras
28 ONNX Team. Tensorflow-onnx project repository. 2022. https://github.com/onnx/tensorflow-onnx
29 Luo W, Chai D, Run X, et al. Graph-based fuzz testing for deep learning inference engines. In: Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, Madrid, 2021. 288–299

30 Pei K, Cao Y, Yang J, et al. DeepXplore: automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, 2017. 1–18
31 Du X, Zheng Z, Ma L, et al. An empirical study on common bugs in deep learning compilers. In: Proceedings of the 32nd International Symposium on Software Reliability Engineering (ISSRE), 2021. 184–195
32 Ma H, Shen Q, Tian Y, et al. Fuzzing deep learning compilers with hirgen. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, 2023. 248–260
33 Zou D, Wang R, Xiong Y, et al. A genetic algorithm for detecting significant floating-point inaccuracies. In: Proceedings of the 37th IEEE International Conference on Software Engineering, 2015. 529–539
34 Xiao D, Liu Z, Yuan Y, et al. Metamorphic testing of deep learning compilers. Proc ACM Meas Anal Comput Syst, 2022, 6: 1–28
35 Orso A, Rothermel G. Software testing: a research travelogue (2000–2014). In: Proceedings of Future of Software Engineering Proceedings, 2014. 117–132
36 Fraser G, Arcuri A. EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011. 416–419
37 Gligoric M, Gvero T, Jagannath V, et al. Test generation through programming in UDITA. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, 2010. 225–234
38 Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008. 206–215
39 Holler C, Herzig K, Zeller A. Fuzzing with code fragments. In: Proceedings of the 21st USENIX Security Symposium, Bellevue, 2012. 445–458
40 Padhye R, Lemieux C, Sen K, et al. Semantic fuzzing with zest. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019. 329–340
41 Wang Z, Yan M, Chen J, et al. Deep learning library testing via effective model generation. In: Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. 788–799
42 Liu J, Wei Y, Yang S, et al. Coverage-guided tensor compiler fuzzing with joint IR-pass mutation. Proc ACM Program Lang, 2022, 6: 1–26