NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers

Jiawei Liu*
University of Illinois
at Urbana-Champaign
Champaign, IL, USA
jiawei6@illinois.edu

Jinkun Lin*
New York University
New York, NY, USA
jinkun.lin@nyu.edu

Fabian Ruffy
New York University
New York, NY, USA
fruffy@nyu.edu

Cheng Tan
Northeastern University
Boston, MA, USA
c.tan@northeastern.edu

Jinyang Li
New York University
New York, NY, USA
jinyang@cs.nyu.edu

Aurojit Panda
New York University
New York, NY, USA
apanda@cs.nyu.edu

Lingming Zhang
University of Illinois
at Urbana-Champaign
Champaign, IL, USA
lingming@illinois.edu

ABSTRACT

Deep-learning (DL) compilers such as TVM and TensorRT are increasingly being used to optimize deep neural network (DNN) models to meet performance, resource utilization and other requirements. Bugs in these compilers can result in models whose semantics differ from the original ones, producing incorrect results that corrupt the correctness of downstream applications. However, finding bugs in these compilers is challenging due to their complexity. In this work, we propose a new fuzz testing approach for finding bugs in deep-learning compilers. Our core approach consists of (i) generating diverse yet valid DNN test models that can exercise a large part of the compiler's transformation logic using light-weight operator specifications; (ii) performing gradient-based search to find model inputs that avoid any floating-point exceptional values during model execution, reducing the chance of missed bugs or false alarms; and (iii) using differential testing to identify bugs. We implemented this approach in NNSmith which has found 72 new bugs for TVM, TensorRT, ONNXRuntime, and PyTorch to date. Of these 58 have been confirmed and 51 have been fixed by their respective project maintainers. **Jiawei: Use the clean repo https://www.overleaf.com/6974619962knbhxscyjkgx which has comments and unnecessary files removed.**

*Equal contribution.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575707

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Neural networks**.

KEYWORDS

Fuzzing, Compiler Testing, Deep Learning Compilers

ACM Reference Format:
Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575707

1 INTRODUCTION

Deep learning (DL) compilers such as TVM [11], TensorRT [46], and TensorFlow XLA [1] are increasingly being used to deploy deep neural network (DNN) models in many different applications. These compilers optimize DL models to meet desired performance, energy, and resource requirements, allowing their use by interactive or safety-critical applications deployed on a variety of devices. However, as compiler implementations are complex, we must be vigilant about detecting bugs in these systems. Compiler bugs can result in crashes or generating an incorrect executable that produces different results than those intended by the user-specified input model[1].

In this paper, we develop techniques to *automatically find bugs in deep-learning compilers*. Similar to prior work [31, 35, 59], we adopt a fuzzing and differential testing based approach: we generate random models, compile them using the compiler being tested, and

[1] As deep-learning models use floating-point operations, a correctly compiled executable model can have close but not identical results as those of the input model. We do not regard this case as a bug.

then compare results obtained from the compiled model with those from a reference implementation. This basic approach faces two main challenges, which are not adequately addressed by prior work. First, how do we generate structurally *diverse* and *valid* models? Deep-learning compilers express a model as a computation graph of tensor operators. For better test coverage, we must ensure model diversity, which requires us to generate graphs by combining operators in different ways. However, connecting two arbitrary operators often produces invalid models, which are rejected by deep learning compilers. For example, a compiler will reject any computation graph containing a *MatMul* (matrix multiplication) operator for which the number of rows in the first input differs from the columns for the second. Therefore, for test efficiency, our graph generation method must also ensure the validity of generated models. Second, given a compiled model, what weights/inputs should we use to run it for differential testing? Naively testing generated models with random or default weights/inputs can easily lead to floating point (FP) exceptional values, *i.e.,* NaNs or infinities (Infs). In such cases, we cannot compare the compiled model with its reference implementation. Therefore, to enable equivalence checking, we must be able to generate computation inputs that can avoid FP exceptional values during model execution.

We develop NNSmith, a tester for deep-learning compilers including TVM [11], ONNXRuntime [40], and TensorRT [46], that addresses these two challenges. NNSmith adopts a three-step approach for finding bugs: (i) first, it automatically generates an arbitrary but valid computation graph expressing some model M_I; (ii) it then uses the compiler being tested to produce a compiled model M_O from M_I, and a reference backend to produce an executable model M_R; and (iii) finally it generates random inputs which it passes to M_O and M_R, and compares their outputs.

NNSmith addresses the model and input generation challenges as follows.

Generating diverse and valid computation graphs: The computation graph expressing a deep-learning model consists of tensor operators with attributes attached to both the operators and graph edges. Operator attributes specify parameters such as kernel sizes that impact the operator's semantics, while edge attributes are used to specify input and output tensor types[2]. Before proceeding with the actual compilation steps, deep-learning compilers check the validity of the input computation graph, *e.g.,* whether an operator's output tensor type matches the expected input tensor type of its downstream operators and whether an operator's attributes are valid. In order to produce valid graphs, NNSmith aims to capture and ensure the type matching constraints of a computation graph during its generation. To do so, NNSmith requires that users provide a specification for each operator, which specifies the constraints that must be satisfied by the operator's input tensors/attributes and also indicates the operator's output tensor type, which NNSmith can use to check the validity of generated graphs. During NNSmith's incremental graph generation, it inserts one candidate operator at a time by solving for the satisfiability of its type matching constraints given the existing graph. NNSmith uses an existing SMT solver [42] for constraint solving.

[2]A tensor's type defines its shape and its elements' data type.

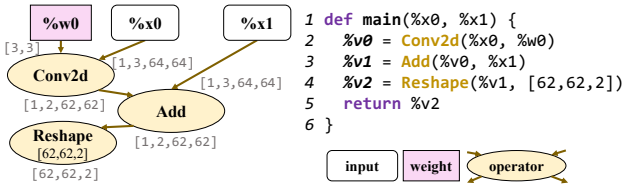


Figure 1: Sample DNN graph.

Executing compiled models without FP exceptional values. In order to meaningfully compare the outputs of a compiled computation graph with those from a reference implementation, NNSmith aims to select computation inputs (aka model weights and inputs) that do not result in NaNs or Infs during execution. Instead of random search, NNSmith uses gradient-guided search to efficiently find viable model inputs/weights for 98% of the generated models with negligible overhead.

In addition to addressing the two main challenges above, we designed NNSmith so it can be easily extended to add support for new operators or to work with other deep-learning compilers. We do so by providing users with a framework for writing operator specifications that are needed to ensure graph validity, and by providing a library of common patterns. In our experience, using this framework and library, users can write new operator specifications in a few lines of code. We evaluated the efficacy of our approach by using NNSmith to identify bugs in TVM, ONNXRuntime, TensorRT, and PyTorch.

Over the last seven months, NNSmith found 72 new bugs in these frameworks. Developers have confirmed 58 and fixed 51 of these bugs. Our coverage evaluation also shows that NNSmith outperforms the state-of-the-art fuzzer by 1.8× for ONNXRuntime and 1.08× for TVM in *total* branch coverage, as well as 32.7× and 10.8× respectively in *unique* branches.

2 BACKGROUND

2.1 The DNN Computation Graph

DL frameworks represent a model's underlying computation as a directed graph of tensor operators. In this work, we focus on DNN inference, where the graph captures the forward NN computation to generate predicted labels or outputs given the model weights and some inputs. For example, the model in Figure 1 is invoked by specifying its inputs (*i.e.,* input variables `%x0` and `%x1`) and the model weights (*i.e.,* input variable `%w0`), and the DNN runtime computes the output tensor (`%v2`) from these inputs.

In what follows, we use the term *tensor type* to refer to both the shape and element type of a tensor. In the DNN computation graph, each edge is marked with the tensor type that corresponds to the output of the edge's upstream operator, as shown in Figure 1. When instantiating an operator, model developers must specify certain additional attributes that dictate its output tensor type. For example, on line 4 in Figure 1, the `Reshape` operator takes `%v1` as an input tensor and `[62,62,2]` as an attribute indicating the output shape. Because each operator expects its input tensors to be of certain types, it is often invalid to connect two arbitrary operators together by an edge: *e.g.,* the reshape operator on line 4 is valid if

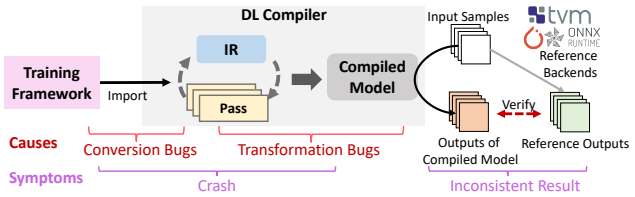


Figure 2: Deep learning compiler workflow and bug finding.

and only if its upstream operator's output (%v1) has 7688 elements ($62 \times 62 \times 2$). This is akin to a "type checking error" in traditional programs. We say that a DNN computation graph is valid if and only if all operators in the graph are valid.

2.2 DL Compilers

State-of-the-art DL compilers turn a user-specified model, expressed as a DNN computation graph, into an executable implementation. As shown in Figure 2, DL compilers process an input DNN model in two stages during its compilation.

First, DL compilers need to convert an input computation graph into their own internal formats. For interoperability, DL training frameworks typically export trained models to a standardized format such as ONNX [2]. DL compilers take ONNX models as input and convert them to a compiler-specific Intermediate Representation (IR) that is easier for the compiler to ompimize.

Next, DL compilers invoke various transformation passes which rewrite the input IR into a more efficient version. These passes include: graph-optimization passes that simplify the graph (*e.g.,* constant folding) or fuse operators (merge *Add* and *Softmax* into *BiasSoftmax*) [47]; low-level passes that optimize computation using arithmetic simplification and loop tiling/fusion, to reduce computational overheads.

Compiler bugs can occur in both the conversion and transformation phases, but bugs in the later phase are generally harder to identify and debug. To comprehensively detect both kinds of bugs, we need to test using models with a diverse graph structure and tensor operators.

2.3 Challenges in Finding DL Compiler Bugs

Differential testing and fuzzing [39] is a promising approach for finding DL compiler bugs. As shown in the right part of Figure 2, this approach requires synthesizing random models for compilation, then running the compiled models with random inputs, and finally comparing the generated results with those from a reference implementation.

There are several challenges facing the basic approach of fuzzing and differential testing, which are not addressed by prior work [35, 59, 61]. Next, we illustrate these challenges using concrete examples.

Challenge #1: Generating graphs with diverse patterns. Finding DL compiler bugs requires generating input graphs that contain a variety of operators and connections. Some prior fuzzers [61] test only using single-operators and thus are too limiting. LEMON [59] and GraphFuzzer [35] generate multi-operator computation graphs, but they are restricted to certain types of operators and connections in order to avoid "type check" errors on the generated graphs (detailed in §6.1). These restrictions limit graph diversity, and compromise test coverage.

Listing 1: DNN patterns that can expose compiler bugs.

```
1 def M0():              # M0 triggers a compiler crash bug!
2   A = Conv2d(...)      # shape: (1,2,1,48)
3   B = Ones(1,1,48)     # shape: (  1,1,48)
4   return A + B
5
6 def M1():              # bug NOT triggered!
7   A = Conv2d(...)      # shape: (1,2,1,48)
8   B = Ones(1,2,1,49) # different shape: (1,2,1,49)
9   return A + B[:,:,:,:48] # slice to match (1,2,1,48)
10
11 def M2():             # bug NOT triggered!
12   A = Conv2d(...)     # shape: (1,2,1,48)
13   B = Ones(1,1,1)    # trivial shape: (1,1,1)
14   return A + B
15
16 def M3():             # M3 can trigger a semantic bug
17   Y = Conv2d(Conv2d(...), ...)  # bug lies here
18   Y = Pow(Y, BIG_NUM) # bug not exposed due to Infs
19   return Y
```

Listing 1 shows an example model (`M0`) generated by NNSmith which has triggered a layout analysis bug in TVM. LEMON cannot generate this model because `M0` contains non-shape-preserving operators (*e.g., Conv2d*) and connections (*e.g.,* broadcasting) which are not supported by LEMON for ensuring graph validity. GraphFuzzer uses a different strategy to guarantee graph validity. Specifically, GraphFuzzer tries to "fix" mismatched tensor shapes in generated graphs through slicing and padding, as illustrated by line 6 in model `M1`. Unfortunately, doing so biases the generated graphs to include many slicing/padding nodes. In our example, the slice operation in `M1` would silence the layout bug found in `M0`.

Challenge #2: Exploring diverse attributes for operators and edges. When generating graphs, it is tempting to ignore the need to explore the operator/edge attribute space and rely on some default values. For example, `M2` of Listing 1 uses trivial attributes (*e.g.,* always 1) to initialize operator `Ones(1,1,1)` (line 13). Unfortunately, the bug found by `M0` will not be triggered by `M2`. Since exploring different attribute values results in diverse output tensor types on edges, it further complicates the task to ensure the validity of generated graphs.

Challenge #3: Running compiled models to produce numerically valid output. Using arbitrary inputs and model weights to test a compiled model can result in FP exceptional values (*i.e.,* NaNs and Infs) during execution for differential testing. Such cases occur when the given inputs to some operator are outside of its expected domain, *e.g.,* feeding *Sqrt* negative values results in NaNs, and feeding *Pow* large base or exponents results in Infs. Larger graphs are particularly prone to encountering FP exceptional values. For example, we have found that NaN/Inf occurs in 56.8% of 20-node models generated by NNSmith when using PyTorch's default weight initializer. Previous testing frameworks did not consider these issues, and consequently up to 41% of their bug reports can be false-alarms because of the undefined/non-deterministic behaviors arising from the NaN/Inf [17].

Clearly we should not compare the output of a compiled model to those of the reference implementation if the results themselves contain NaN/Inf. What about those scenarios with normal final

results (aka without any NaN/Inf) where some internal operator has produced FP exceptional values during graph execution? For example, operator `ArgMax` can output a normal FP value even though one of its upstream operators gives it NaN as input. It is a subtle requirement that we must also exclude these results from differential testing or risk incurring false positives in bug detection. This is because when handling FP exceptional values, otherwise semantically equivalent operators could produce different results. Therefore, to be able to test effectively, we must generate model inputs/weights that avoid FP exceptional values for all operators in the graph. Only then we refer to the model's output as *numerically valid*. Otherwise, we might miss detecting bugs. As an example, Listing 1's model `M3` can trigger a semantic bug. However, this bug is not exposed because the execution results in Inf values which are not used for comparison.

3 NNSMITH'S DESIGN

Overview of the approach. Figure 3 shows an overview of NNSmith's workflow. NNSmith generates random models that are valid, which are then compiled and executed. NNSmith takes as input a compiler's type checking requirements in the form of operator specifications (§3.1), and then uses an SMT solver to generate graphs and operator attributes that meet these constraints (§3.2). Next, when running a compiled model, NNSmith uses a gradient guided search procedure to find benign weights/inputs so that no FP exceptional values are produced at any step of the execution (§3.3). Finally, NNSmith compares the results obtained from multiple deep learning libraries and compilers to those from a reference implementation to identify bugs.

3.1 Modeling DNN Operators

NNSmith generates random DNN models expressed as computational graphs by connecting together different operators. We aim to generate valid graphs that "type check", *i.e.,* graphs where each operator's attributes and input tensor type meet requirements imposed by the compiler.

In order to generate valid graphs, we require users to provide operator specifications that explicitly state the compiler's requirements for each operator and guarantees about its output. An operator's specification codifies rules for checking validity and depends on its inputs and attributes: For example, the 2-D convolution operator (*Conv2d*) has several attributes, including a kernel, and takes an image as input. A model that uses a *Conv2d* operator is valid if the input image is a rank-4 tensor that is larger than the kernel's size.

While our implementation includes specifications for common operators (detailed in §4), we designed NNSmith so that it is easy for users to write specification for additional operators. NNSmith specifications are written using symbolic integers and abstract tensors. An abstract tensor is specified by its data type, rank and shape. In our implementation we specify an abstrat tensor's data type and rank using concrete values, and use symbolic integers to specify its shape. As we will see later in §3.2, NNSmith uses an SMT solver to assign concrete integers to each symbolic integer during graph generation. NNSmith operator specifications provide input and output types (specified using abstract tensors), constraints on inputs and attributes, as well as transfer rules for each operator. Listing 2 shows the operator specification for a 2-D pooling operator (*Pool2d*), and we describe each of part below:

Listing 2: Sample specification for the 2D Pooling operator.

```
1 class Pool2d(AbsOpBase):
2   input_type =   # [(input#1 types), ...]
3       [(AbsTensor(float32, 4), AbsTensor(float64, 4))]
4   output_type =  # [(output#1 types), ...]
5       [(AbsTensor(float32, 4), AbsTensor(float64, 4))]
6
7   def __init__(self, kh, kw, stride, pad):
8     self.kh, self.kw = kh, kw ...
9
10  def requires(self, inputs :List[AbsTensor]):
11    ih, iw = inputs[0].shape[-2:]
12    return [self.kw > 0, self.kh > 0,
13      self.stride > 0, self.pad >= 0,
14      self.kw <= 2 * self.pad + iw, ...]
15
16  def type_transfer(self, inputs :List[AbsTensor]):
17    n, c, iw, ih = inputs[0].shape
18    oshape = [n, c,
19      (h - self.kh + 2*self.pad) // self.stride + 1,
20      (w - self.kw + 2*self.pad) // self.stride + 1]
21    return [AbsTensor(inputs[0].dtype, shape=oshape)]
22
23  def infer_input_type(self, outputs):# §3.2
24    return [AbsTensor(outputs[0].dtype, 4)]
```

Inputs and outputs. An operator's attributes are inferred from the inputs to its `__init__` function. The class variables `input_type` and `output_type` describe the input and output tensor types respectively (Lines 3 and 5). Programmers specify a list of tuples, each tuple says what data types can be used for an input (or provided as output). In the listing, the *Pool2d* operator accepts a single rank 4 tensor of 32-bit or 64-bit floats.

Constraints. The operator's `requires` function (Line 10) returns constraints that its inputs and attributes must satisfy as a list of logical predicates. For example, among other constraints, the *Pool2d* operator requires that the kernel size should be greater than 0 (Line 12).

Type transfer function. The operator uses a *type transfer function* (Line 16) to specify how its output tensor relates to its inputs. For example, on Line 21, *Pool2d*'s type transfer function relates the shape of the operator's output tensor to its kernel size (`self.kw` and `self.kh`) and its input shapes. Observe that the constraints output by the type transfer function become the input constraints on a downstream operator. These constraints are used to to combine constraints from connected operators in a computation graph, thereby allowing NNSmith to generate valid models.

3.2 Model Generation

Given a set of operator specifications, NNSmith generates models that are *topologically diverse* and whose operators use *diverse attributes*. Below we first detail our approach for generating diverse model topologies and then present our binning based approach to assigning diverse attributes.

Generating computation graphs. Our model generation algorithm is designed to ensure that generated computation graphs are fully connected, as is the case with most real-world models. Additionally, it is also designed so that it can generate a rich variety of models, including ones similar to existing multi-modal and multi-task models [3, 22, 44, 53] that can accept multiple inputs and/or produce multiple outputs.

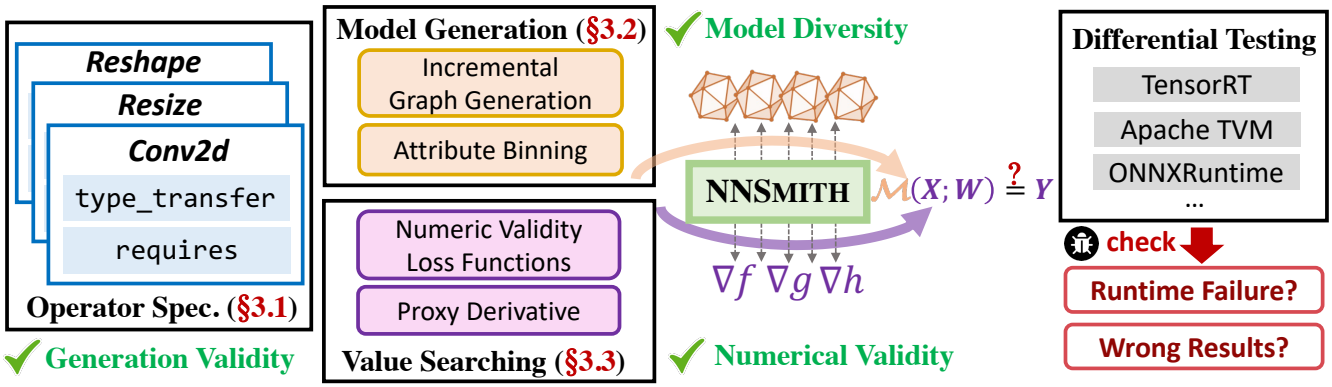


Figure 3: Overview of NNSmith.

NNSmith generates connected computation graph by extending an existing graph while maintaining connectivity. It does so by starting with a graph that contains a single *placeholder* node, and extending it by either (a) adding a new node whose input edges are connected to the output of an existing node (we refer to this as a *forward* insertion) or (b) replacing an existing placeholder node with an operator node whose input edges are connected to one or more placeholder nodes (we refer to this as *backward* insertion). In both cases, the node added by NNSmith is picked at random from the set of symbolic operator specifications (*op*) it is provided. Placeholder nodes have one output, and at the end of the graph generation process they are replaced by input nodes or by weights (which are constant inputs). Algorithm 1 shows our graph generation algorithm. We detail the steps taken when inserting a randomly selected operator (*op*) into an existing compute graph below:

1. *Type matching:* To insert *op*, NNSmith must first find a feasible insertion point in the current graph. When using forward insertion, this means finding an output edge in the graph whose constraints (as provided by the operator that node represents) satisfy *op*'s input constraints. Similarly, when using backward insertion, this means finding a placeholder node whose output is connected to node(s) whose input constraints are satisfied by *op*'s output constraints. To do so we need to check constraint satisfaction, and we use a SMT solver for this. Rather than invoking an SMT solver for all possible insertion points, we use a simple type matching heuristic to filter out nodes that are obviously infeasible because of incompatible data types or ranks. For example, when using forward insertion for *Where(cond, T, F)*, type matching (Lines 7) will filter out any output edges which are not boolean.
2. *Constraint solving:* Next, NNSmith generates constraints for any feasible insertion points that have not been filtered out by its type matching heuristic, and uses an SMT solver to check their satisfiability. NNSmith caches constraints for the current model (in M.`solver`) to reduce constraint generation overheads, and uses incremental solving [5] to reduce time taken for checking constraints (Line 5).
3. *Node insertion:* As we stated previously, we use one of two approaches to insert nodes into the graph: forward insertion and backward insertion:
   - *Forward insertion* (Line 6) selects one group of plausible tensors (v) as the inputs of *op* (Line 8) and inserts *op* as their consumer (Line 10) if the insertion constraints are satisfiable (Line 9).
   - *Backward insertion* (Line 11) replaces an existing placeholder node with *op*. To do so it first determines a placeholder candidate (v) by matching *op*'s output type and the candidate's type (Line 12—13). Next, it infers *op*'s input type from v. If *op*'s input type constraints can be satisfied for the candidate, NNSmith replaces the candidate with *op* and creates new placeholder nodes (of the inferred types) to act as *op*'s inputs (Lines 18 and 19).

Algorithm 1: Computation graph generation.

Input: Graph M; operator to try *op*; global constraints C.

```
1  Function Solve(M, C, op, op's inputs v):
2      C ← C ∪ op.requires(v)
3      for ∀s ∈ op.type_transfer(v) do
4          C ← C ∪ {s.shape_0 ≥ 1, ··· , s.shape_{s.rank−1} ≥ 1}
5      return M.solver.try_add_constraints(C)
6  Function ForwardInsert(M, op):
7      S ← TypeMatch(M.intermediates(), op.input_type)
8      v ← randomly choose one input combination from S
9      if Solve(M, ∅, op, v) then
10         M.insert_consumer(v, op)
11 Function BackwardInsert(M, op):
12     S ← TypeMatch(M.placeholders(), op.output_type)
13     v ← randomly choose one set of placeholders from S
14     // Also see infer_input_type in Line 23 of Listing 2
15     p ← new placeholders w.r.t. op.infer_input_type(v)
16     o ← op.type_transfer(p)
17     if Solve(M, {∀i ∈ [0, |v|], v_i.shape = o_i.shape}, op, p) then
18         M.insert_consumer(p, op)
19         M.replace_placeholder(v, op)
```

Attribute binning. In addition to topological diversity, attribute diversity is also crucial as discussed in §2.3. We use the solution (model) generated by an SMT solver (Z3 [42] in our implementation) when checking satisfiability for the graph's constraints to determine attributes. However, we found that the models produced by SMT solvers tend to pick boundary values for integer constraints. For example, when a tensor shape D is constrained so that $\{d_i \geq 1; d_i \in D\}$, the Z3 SMT solver always returns models where $\{d_i = 1; d_i \in D\}$, limiting the diversity of attributes in generated DNNs.

We address this problem by adding extra constraints (which we call binning constraints) that limit each attribute to a randomly chosen range. Algorithm 2 shows how we generate these binning constraints: We start from an empty set of binning constraints (C_b in Line 8), and iterate over each attribute α of each operator op (Line 9-10) in the graph (M). Note that this algorithm also considers placeholders as operators, and uses placeholder tensor shape as attributes for these operators. The binning constraints for each attribute are generated by randomly choosing from one of k bins, where the i^{th} bin represents the range $[2^{i-1}, 2^i)$ (when $i < k$, the last bin represents the range $[2^{k-1}, \infty)$), and limiting the attribute to a subset of the bin's range. To do so, we randomly pick a bin (Line 12) for each (op, α), and sample two integers (*i.e.,* l and r) from it (Line 13). We then add $l \leq \alpha \leq r$ as the binning constraints for the (op, α) pair. We use bins with exponential ranges (Line 1) because, in practice, systems are more sensitive to changes in smaller values, e.g., changing a variable from 0 to 1 generally has larger effect on the output than changes from 30 to 31. Our approach of dividing ranges in exponential buckets is inspired by how AFL [68] coarsely records the hit counts for execution tuples [19]. We allow operators to provide a different, more specialized strategy, for attribute binning, and in this case we simply use the provided constraints (C^* in Line 16) instead (details in §4).

Adding extra binning constraints to the graph's constraints can produce an unsatisfiable constraint system, leading to a situation where we can find no attributes for a valid graph. We avoid this situation by adding constraints *only* after a graph has been generated (Paragraph 2 in §3.2) and *only* when doing so does not impact satisfiability. Specifically, if the solver fails to find a satisfiable assignment after adding C_b to the graph's constraints, we randomly drop half of the constraints and retry, until it succeeds (Line 18).

Algorithm 2: Attribute binning.

```
Input: Graph M; global constraints C; # bins k.
1  Function SampleFromBin(i, k):
2      if i ≠ k then
3          b, t ~ Uniform(i − 1, i) where b < t and b, t ∈ ℝ
4          return (⌊2^b⌋, ⌊2^t⌋)
5      else
6          return (2^{k−1}, ∞)
7  Function AttrBinning(M, C, k):
8      C_b ← ∅                                  // Binning constraints to apply
9      for op ∈ M do
10         for α ∈ op do
11             if no specialization for (op, α) then
12                 i ← random integer ∈ [1, k]   // Randomly pick a bin
13                 l, r ← SampleFromBin(i, k)
14                 C_b ← C_b ∪ {l ≤ α ≤ r}
15             else
16                 C_b ← C_b ∪ C*(op, α)     // The last paragraph of §4
17     while ¬M.solver.try_add_constraints(C_b) do
18         C_b ← randomly select |C_b|/2 constraints in C_b
```

Operator	Domain	Violation	Loss functions
Asin(X)	$\lvert X\rvert \leq 1$	*NaN*	$\mathcal{L}(\lvert X\rvert - 1 \leq 0)$
Div(X, Y)	$\lvert Y\rvert > 0$	*NaN*	$\mathcal{L}(\lvert Y\rvert > 0)$
Pow(X, Y)	$\begin{cases} X > 0 \\ Y \log(X) \leq 40 \end{cases}$	*NaN*/*Inf*	$\begin{cases} \mathcal{L}(X > 0) \\ \mathcal{L}(Y \log(X) - 40 \leq 0) \end{cases}$
Log2(X)	$X > 0$	*NaN*	$\mathcal{L}(X > 0)$

Table 1: Representative vulnerable operators.

Tensor Ineq.	Loss function $\mathcal{L}$
$f(X) \leq 0$	$\sum_{x \in X} \max(f(x), 0)$
$f(X) < 0$	$\sum_{x \in X} \max(f(x) + \epsilon, 0)$

Table 2: Tensor inequality to loss function conversions.

3.3 Improving Numeric Validity with Gradients

Next, NNSmith generates inputs and weights that can be used to test the generated models. We initially considered using randomly selected numbers. However, we found that the generated graphs produce FP exceptional values, including *NaN* (not a number) and *Inf* (infinite number). For example, when generating 20-operator graphs, FP exceptional values occur in 56.8% of generated graphs if we use random weights and inputs.

This is because some operators, which we refer to as *vulnerable operators* [65], produce real (*e.g.,* $\sqrt{x}$ returns NaN if $x < 0$) or stable (*e.g.,* x^y returns Inf for large x and y) results only for a subset of their input domain. If a vulnerable operator's input lies outside of this domain, the operator outputs an FP exceptional value, which propagates through the model and impacts the model's output, preventing us from comparing model outputs during differential testing. Table 1 lists examples of vulnerable operators we encountered in our evaluation.

One way to address this problem is to use additional heuristics to extend and fix vulnerable operators. For example, changing *Div*(x, y) to *Div*(x, |y|+ϵ) ensures that the *Div* operator is safe. However, this requires changing operator inputs, which limits graph diversity as discussed in §2.3. We thus propose an alternate approach, where we use a gradient-search algorithm to find inputs that ensure that the model's output is numerically valid. Our approach is inspired by GRIST [65], though that work has the *opposite* goal: it aims to find inputs that result in FP exceptional values.

At a high-level, our approach associates a set of loss functions with each operator. When selecting inputs, NNSmith starts with random inputs, and then iteratively refines these inputs so that no operator in the graph produces an FP exceptional value. In each iteration, NNSmith identifies the first operator in the model that produces an FP exceptional value. It then uses the set of loss

functions associated with the operators to compute new model inputs and uses these for the next iteration. The algorithm terminates when no FP exceptional values are found. We provide details below:

Loss functions for avoiding FP exceptional values. NNSmith associates a set of loss functions with each operator, which our input search algorithm (Algorithm 3) uses to update inputs to avoid FP exceptional values. Users can specify loss functions for each operator, and below we describe our approach to producing loss functions.

As we noted above, vulnerable operators produce valid outputs (*i.e.,* outputs that are not FP exceptional values) when inputs are drawn from a particular domain, and this domain can be expressed (or approximated) by the conjunction of a few (usually one or two) inequality predicates on the operator's input. We refer to this conjunction of inequality predicates as the operator's *tensor inequalities*. For example, the $Sqrt(X)$ operator takes a tensor X as input, and is numerically valid if and only if $X \geq 0$ (*i.e.,* all elements of X are positive). Similarly, the $Pow(X, Y)$ operator's numerically valid domain can be under-approximated as $X \geq 0 \wedge Y \log(X) \leq 40$, which requires that all elements of X be positive to avoid NaNs (since Y might contain fractional elements) and bounds $Y \log(X)$ to avoid outputs that are too large (and would be represented by infinity) [3]. We associate a loss function with each predicate in an operator's tensor inequality. We do so by first rewriting each predicate so that it is either of the form $f(X) < 0$ or $f(X) \leq 0$, and then use the formulas in Table 2 to convert this cannonical form to a scalar loss. We show examples of the loss functions produced in this manner in Table 1. When an operator produces invalid outputs, the search algorithm picks which loss function to use by finding a predicate that is violated by the operator's current input and using the loss function associated with it. For simplicity, our design assumes that a loss function is positive if and only if its associated predicate is violated by the operator's input, allowing us to use any positive loss function associated with the operator (Line 8) without evaluating its associated predicate.

Proxy derivative. Given a vulnerable operator's loss, NNSmith uses gradient propagation to compute changes to the model inputs and weights. Doing so requires computing gradients (derivatives) for each operator in the graph (Line 9). However, some operators are either undifferentiable for some inputs (*e.g., Floor, Ceil*, and other operators cannot be differentiated at integers) or have zero gradient in some region (*e.g., ReLU* has gradient 0 for all negative inputs), and this prevents backward propagation. For these functions, we use *Proxy Derivative Functions* [4] instead of actual derivatives during gradient propagation.

Given an operator F whose gradient is 0 in region $\mathbb{U}$ we use $\frac{dF(x)}{dx}(x \in \mathbb{U}) = \alpha$ as the derivative. We set α's sign based on the overall trend of the function, *e.g.,* we use a positive α for *ReLU* because it is monotonic. Similar to *LeakyReLU* [64], we choose a small magnitude for α, thus avoiding large discrepancies between the proxy and the actual derivative. On the other hand, if the operator F cannot be differentiated in the region U, we use the closest left-derivative instead.

[3] We constrain the logarithm instead of directly using the power function to ensure that the loss function does not generate a FP exceptional value.

Algorithm 3: Gradient-guided value search.

```
1  Function GradSearch(DNN M, learning rate μ):
2      ⟨X, W⟩ ← randomly initialized inputs and weights
3      OUTER:while time budget not exhausted do
4          for operator F_i in TopologicalSort(M) do
5              I_i ← input to F_i
6              O_i ← F_i(I_i)
7              if ∃ NaN/Inf ∈ O_i then
8                  ℒ ← first positive loss functions of F_i
9                  ⟨X, W⟩ ← ⟨X, W⟩ − μ∇_⟨X,W⟩ ℒ(I_i)
10                 if ⟨X, W⟩ not changed then // Zero gradients
11                     ⟨X, W⟩ ← randomly initialized values
12                 else if ∃ NaN/Inf ∈ ⟨X, W⟩ then
13                     Replace NaN/Inf with random values
14                 continue OUTER // Go to Line 3
15         return ⟨X, W⟩
16     raise failed to find viable ⟨X, W⟩
```

Search process. The overall input search algorithm (Algorithm 3) proceeds as follows: Given a model M and time budget T, we first randomly initialize inputs and weights $\langle X, W \rangle$ (Line 2) used by the first iteration of the search algorithm (Line 3). In each iteration, we find the first operator (in topological order, Line 4) that produces an FP exceptional value (Line 7). We use its loss function as an optimization objective (Line 8) to tune $\langle X, W \rangle$. If the gradient is neither zero nor a FP exceptional values then we move on to the next iteration (Line 14), otherwise we restart the search with a different initial value (Line 11 and 13). The algorithm throws an exception (Line 16) if it does not terminate within the time budget.

Because loss functions can vary by orders-of-magnitude across operators, we use Adam [25], an adaptive learning rate scheduling algorithm, to set the learning rate. We also reset the learning rate whenever we switch the loss functions used for optimization (as would be the case when an iteration finds a different operator). While this design can lead to a scenario where optimizing for one operator leads to another producing invalid outputs and vice-versa, we found that this to be rare in practice (it occurred less than 1% of the time). We found that the most common reason for the search algorithm failing was that the model has no valid inputs.

4 IMPLEMENTATION

NNSmith is implemented in 5157 lines of Python code. Consistent with Algorithm 1, NNSmith outputs a symbolic graph and its SMT solution for being valid with the help of the Z3 [42] solver. We then concretize the symbolic graph by invoking the materialized PyTorch functors in the topological order, and export the model to the deployment-friendly ONNX [2] format using PyTorch's exporter. We also use PyTorch to implement our algorithm for finding model inputs/weights that result in numerically valid output (§3.3).

Since DL compilers vary in operator and data type support, we infer the set of operators supported by the compiler being tested by trying to compile single-operator models with different data types. We use this information when generating graphs, so as to avoid "*Not-Implemented*" errors.

When verifying outputs from the compiled model, we regard PyTorch's results as the oracle. We use PyTorch as the reference backend over compiler cross-checking because: 1) Obtaining results from PyTorch is a "*free*" lunch which is a by-product of gradient-based value searching; 2) Current DL compilers support different operator sets or data types, implying that cross-checking is limited to the common set of their support matrices; and 3) PyTorch's results are more trustworthy for being a better-tested interpreter, which has been used to produce oracles in many downstream compilers. In rare cases value inconsistency (1 out of 72 in our bug finding) can happen in PyTorch's converter which makes it unclear whether the bug comes from the converter or the compiler. Therefore, for better fault localization, if the compiler and PyTorch disagree on the model outputs, we further compare it with the compiler "O0" mode at an extra cost of re-compilation. If the "O0" mode also disagrees with the optimized model, we then are confident the compiler's optimization must be wrong.

We wrote operator specifications in NNSmith using information obtained from framework documentation [52] and source code [48]. To simplify this task, we implemented several meta types including, unary/binary, reduce and broadcast that further reduce the amount of code needed to specify an operator. Using these, we found that we could implement 59 (out of 73) operator specification within 4 lines of code. Furthermore, even for the most complex specification, which was for *Conv2d*, the `requires` function has 9 inequalities and the `type_transfer` function is only 7 lines of code (formatted by PEP8 [54]) that can be quickly implemented in a few minutes. Furthermore, these specifications can be written once and then shared by all compilers that can accept ONNX models as input.

Regarding the C^* in attribute binning (Line 16 in Algorithm 2), our current implementation uses the following default settings: 1) we add one extra bin that contains only one integer "0" for the "padding" attribute in *Conv2d*, for that padding can also be 0; 2) similarly, for the "padding" attribute in *ConstPad*, *ReplicatePad*, and *ReflectPad*, we add both the 0-bin and negative bins that for supporting zero and negative padding; 3) we specifically handle for the indexing ranges (*i.e.,* "start" and "end" attributes) in *Slice* to make sure the range is valid to its input tensor shape.

5 EVALUATION

5.1 Experimental Setup

Metrics. We mainly target the following metrics for evaluation:

- *Code coverage*: Following prior fuzzing work [7, 8, 61], we trace source-level branch coverage for both the entire systems and their pass-only components, measuring 1) *total* coverage counts all hit branches; and 2) *unique* coverage counts unique branches ("hard" branches) that other baselines cannot cover.
- *Bug counting*: Following prior work [31, 61, 62], we use the number of independent patches as the number of detected bugs, except that we directly count the number of bug reports for closed-source systems (*i.e.,* TensorRT) and unfixed ones.

Baselines. We compare NNSmith with both the state-of-the-art general DNN model generators (LEMON and GraphFuzzer) and fuzzer specifically designed for TVM (*i.e.,* Tzer).

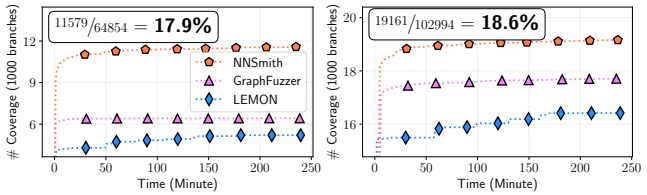
- LEMON [59] is a mutation-based model generator that mutates pre-trained Keras [18] models [60]. We convert Keras models into ONNX, to reuse the same differential testing and evaluation framework of NNSmith for fair comparison;
- GraphFuzzer [35] generates models by randomly connecting nodes from a block corpus. While LEMON is limited to shape-preserving unary operators, GraphFuzzer also supports non-unary operators by aligning input tensor shapes with slicing/-padding and uses specific attributes to create shape-preserving instances for a few non-shape-preserving operators such as *Conv2d*. As its implementation is not open-sourced, for a fair comparison, we reimplemented its main design, *e.g.,* stitching operators via padding/slicing, by replacing NNSmith's specification-based node insertion.
- Tzer [31] is a coverage-guided and mutation-based fuzzer targeting TVM's low-level IR. As DNNs generated by NNSmith can also be lowered to low-level IR, we compare Tzer with NNSmith to see if NNSmith can well cover low-level optimizations as Tzer.

Systems under test. NNSmith finds bugs in the following commonly used compilers:

- ONNXRuntime [47] (by Microsoft) is a graph-optimized DNN library for ONNX models, with over 130 source files on various graph optimizations. Like many runtime-based frameworks (*e.g.,* PyTorch), though ONNXRuntime enables optimizations, the optimized graph will still be directly mapped into pre-compiled kernel functions (*i.e.,* no code generation). To evaluate pass-only coverage, only files under `onnxruntime/core/optimizer` are instrumented;
- TVM [11] is an end-to-end compiler for deploying DNNs on various platforms. In addition to 61 graph-level passes, TVM also performs up to 58 low-level optimizations to generate highly optimized target code. As a front end, ONNX models will be converted into TVM's graph-level IR to perform further optimization. TVM also has a much higher coverage upper limit (*i.e.,* 116k) than ONNXRuntime (*i.e.,* 65k) given its higher capability/-complexity. For pass-only instrumentation, we consider files in all `transfroms` folders.
- TensorRT [45] is a compiler and runtime highly optimized for NVIDIA GPUs and has been used by more than 350k developers across 27.5k companies. Since TensorRT is closed-sourced, we exclude it for coverage evaluation.

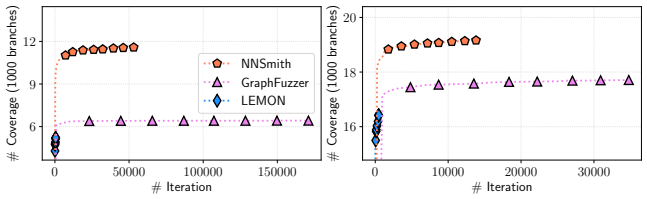
Experimental configuration. The testbed hardware configurations include: 1) Intel 10700k CPU (16 threads); 2) 64 GB memory (3200 Mhz); and 3) 2TB NVMe SSD. The operating system is Ubuntu 20.04 and targeted DL systems are compiled by Clang 14 under release mode. Except that we performed bug findings on various latest compiler versions over the last tenvspace months, the default software versions used in evaluation are: ONNXRuntime v1.12 (`c556f5`), TVM v0.8 (`9ab3a1`), TensorRT v8.4 and PyTorch v1.13 (`dev20220615`).

When evaluating NNSmith, for Algorithm 1 we choose between forward and backward at every insertion randomly with equal probability. For the binning approach we use $k = 7$ bins (§3.2) to ensure a decent amount of attribute diversity while keeping the models small for fuzzing efficiency. For the gradient search,



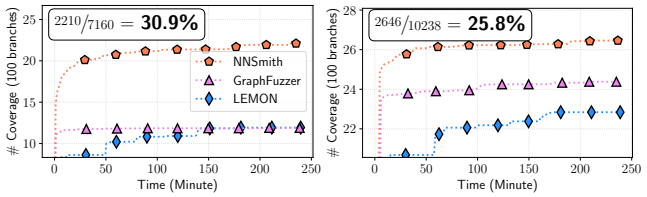
(a) ONNXRuntime (b) TVM

Figure 4: Total branch coverage over time (*all* files).



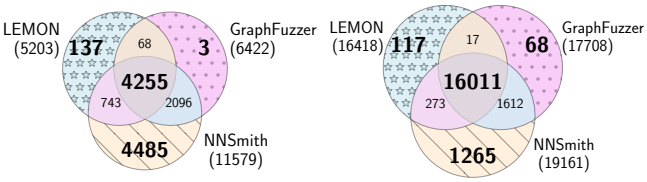
(a) ONNXRuntime (b) TVM

Figure 5: Total branch coverage over test cases (*all* files).



(a) ONNXRuntime (b) TVM

Figure 6: Total branch coverage over time (*pass* files).



(a) ONNXRuntime (b) TVM

Figure 7: Venn diagram of overall coverage (total coverage shown in parenthesis).

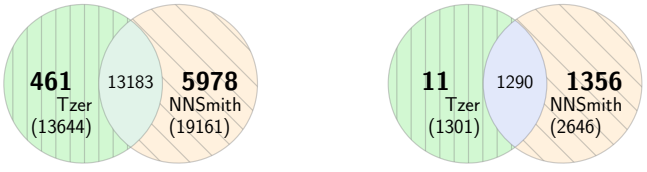
the initial learning rate is set to be 0.5, ϵ in the tensor inequality loss function is set to 10^{-10}. While LEMON does not explicitly control the graph sizes (since it mutates existing models), we set the default generated graph size of NNSmith and GraphFuzzer to be 10. For coverage evaluation, we run fuzzers for 4 hours by default (following Tzer [31]) as we observe that code coverage curves generally converge before that point (*e.g.,* as shown in Figure 4).

5.2 End-to-end Coverage Efficiency

We first compare NNSmith with our graph-level baselines (*i.e.,* GraphFuzzer and LEMON) in terms of code coverage on TVM and ONNXRuntime (since TensorRT is closed-sourced). Figure 4 shows the coverage growth (y axis) over four hours (x axis). As is shown in the Figure, NNSmith beats the 2nd-best baseline (*i.e.,* GraphFuzzer) by 1.8× on ONNXRuntime and by 1.08× on TVM. NNSmith also achieves a decent percentage of total coverage, *i.e.,* 17.9% on ONNXRuntime and 18.6% on TVM [4]. Figure 5 further shows the number of generated test cases (x axis) within 4 hours and their accumulated total coverage (y axis, consistent to Figure 4). We can observe that with fewer test cases generated within the same time limit (mainly due to the overhead incurred by constraint solving), NNSmith can still achieve higher coverage than the 2nd-best baseline (*i.e.,* GraphFuzzer), indicating that NNSmith can generate higher-quality test cases. It is also worth noting that LEMON is the slowest technique (*e.g.,* up to 103× slower than NNSmith). The reason is that LEMON mutates real-world models which can be very costly to run. We also have similar observations on the pass-only coverage. For example, as shown in Figure 6, NNSmith outperforms GraphFuzzer by 1.85× on ONNXRuntime and 1.09× on TVM, showing its effectiveness for testing compiler transformation passes.

Another interesting observation is that NNSmith's coverage improvement on TVM is relatively smaller than that on ONNXRuntime (1.08× v.s. 1.8×). This can be inferred by the difference in their fundamental designs. While ONNXRuntime implements over 130 optimization files targeting various specific graph patterns, TVM's graph-level optimization is more general. For example, TVM's operator fusion does not check specific operator types, but high-level operator properties such as *injective*, *reduce*, etc. Therefore, TVM's coverage is less sensitive to the diversity of generated graph patterns.

To show the *unique* coverage for each studied technique, Figure 7 further breaks down the coverage sets of different fuzzers through Venn diagrams [63]. It shows that NNSmith can achieve much higher *unique* coverage than the 2nd-best baseline (*i.e.,* LEMON), *e.g.,* 32.7× higher on ONNXRuntime and 10.8× higher on TVM. Despite that GraphFuzzer beats LEMON in total coverage, LEMON contrastingly outperforms GraphFuzzer in unique coverage. This is because LEMON has a different design from NNSmith and GraphFuzzer: it mutates existing *real-world* models rather than generating new models from scratch, creating different model patterns. Please note that we omitted the *unique* coverage distribution analysis for pass-only files as it follows a similar pattern as Figure 7.



(a) All files. (b) Pass-only files.

Figure 8: NNSmith vs. Tzer.

[4] Note that it is unlikely for NNSmith to achieve perfect overall coverage as there are many other irrelevant components related to debugging, auto-tuning [12, 71], etc. For instance, existing Linux kernel fuzzers [24] can only achieve 0.8-10.5% coverage.

Figure 8 also compares NNSmith against Tzer on TVM (as Tzer is specifically designed for TVM). On all TVM files, NNSmith as a general graph-level fuzzer, can outperform state-of-the-art IR-level TVM-specific fuzzer by 1.4× in *total* coverage and 13× in *unique* coverage. Interestingly, while other graph-level baselines can at most *exclusively* cover 117 branches (*i.e.,* LEMON in Figure 7b), Tzer has an *unique* coverage of 461. This is because Tzer directly manipulates low-level IR and some low-level operations are not exposed at the graph level. Moreover, in terms of pass-only coverage, NNSmith outperforms Tzer even more, e.g., by 123× in *unique* coverage, demonstrating the superiority of graph-level fuzzing.

5.3 Ablation Study

Attribute binning. Figure 9 evaluates the effectiveness of attribute binning from the perspective of redundancy. Note that for implementation convenience we use the type system from TVM's Relay IR (parsed from ONNX models) to distinguish operators. It shows that within 4 hours, our binning approach achieves 2.07× *unique* operator instances, which are distinguished by input types and operator attributes.

Turning to system coverage, as shown in Figure 10, attribute binning improves the *unique* branch coverage by 2.2× for ONNXRuntime (Figure 10a) and 1.8× for TVM (Figure 10b). The total coverage improvement is relatively subtle (up to 2.3%) as the binning approach aims at covering the hard-to-hit branches whose proportion is expected to be minor. For example, simply importing TVM's libraries with "`import tvm`" can hit 4015 branches but those branches are unlikely to have bugs.

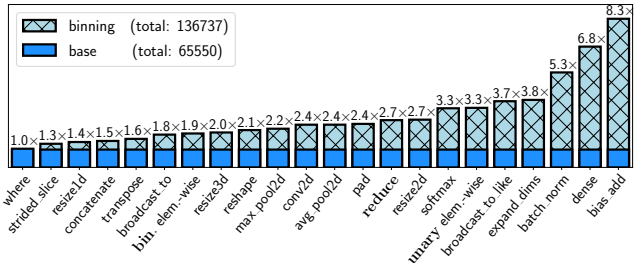
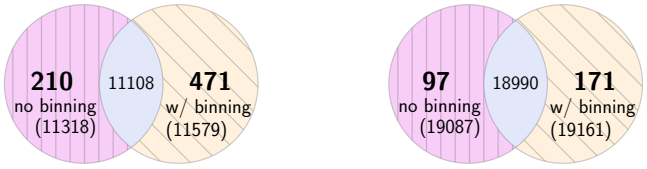


Figure 9: Normalized unique operator instances tested.



(a) ONNXRuntime **(b) TVM**

Figure 10: Impact of attribute binning on coverage.

Gradient guidance. Figure 11 evaluates the effectiveness of three input/weight searching methods: 1) Sampling: randomly initializing test case values; 2) Gradient (Proxy Deriv.): searching values via the full gradient-based approach; and 3) Gradient: method two without proxy derivatives. The experiment is conducted on three model groups, each of which contains 512 models of 10, 20 and 30 nodes respectively. Every model has at least one vulnerable operator. The *Sampling* baseline randomly samples values from the range of $[1, 9]$ which is empirically obtained selecting the best one from various tested ranges. For fairness, all methods run on the same groups of models with the same initial weights/inputs generated by the *Sampling* baseline. We assign different per-model searching timeouts (*i.e.,* $i \times 8$ms where $i \in [1, 8]$) to each method and observe the ratio of models with numeric-valid inputs/weights (y-axis) over group-wide average searching time (x-axis). Figure 11 shows that our full gradient search improves the numerical validity of *Sampling* by 1.16-1.34× as the node size/difficulty grows. Also, the proxy derivative mechanism consistently helps our gradient search achieve higher success rate within shorter amount of time.

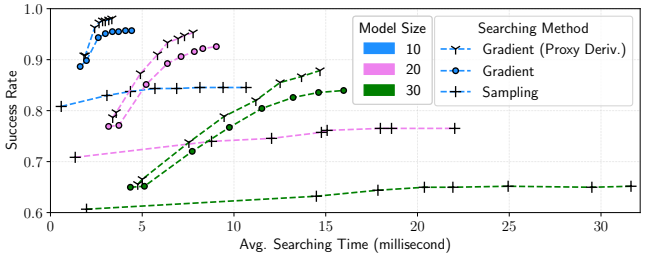


Figure 11: Effectiveness of gradient-based search.

We also observe that searching time is negligible compared with model generation time, *e.g.,* generating a 10-node model costs 83ms on average while our gradient-based searching only takes 3.5ms (**4.2%**) to achieve a success rate of **98%**.

5.4 Bug Study

	Transformation	Conversion	Unclassified	Total
ONNXRuntime	10	0	2	12
TVM	29	11	0	40
TensorRT	4	2	4	10
PyTorch Exporter	–	10	–	10
Total	43	23	6	72
(Crash/Semantic)	(34/9)	(18/5)	(3/3)	(55/17)

Table 3: Bug distribution.

To date, NNSmith has uncovered **72** *new* bugs as shown in Table 3, where **58** have been *confirmed* and **51** have been *fixed*. Others are awaiting developer responses. Interestingly, in addition to compiler bugs, since NNSmith generates models through PyTorch ONNX exporter (§4), it also found 10 conversion bugs in PyTorch as a by-product. Among the bugs we found, 17 are semantic bugs (result inconsistencies with PyTorch) and 55 are crash bugs (segmentation faults or exceptions). In total, there are **43** *transformation* bugs in ONNXRuntime (10), TVM (29) and TensorRT (4), accounting for the majority of the detected bugs [5]. We found that most of

[5]We classify bugs first based on code inspection (when possible); otherwise, we classify a bug as transformation bug if its individual operators cannot reproduce the issue separately.

these were optimization bugs: of the 27 fixed transformation bugs we found, **21** are *optimization* bugs (and the remaining one is an *unclassified* bug in TensorRT whose code is not available).

Of the 72 bugs we found, **49** bugs cannot be triggered using the algorithms implemented by LEMON or GraphFuzzer. Of these 31 are transformation bugs and 14 are conversion bugs. LEMON's algorithms can trigger at most 17 of all bugs we found, while GraphFuzzer's algorithms can trigger at most 23 of these. The core difference is that these prior approaches limit how non-shape preserving operators are connected in the graph, thus limiting graph diversity. In addition to this theoretical analysis, we also evaluated all tools by running them for four hours under the *same* setting (*e.g.,* all on the *default* compiler versions as shown in § 5.1), NNSmith triggers **38** *unique* crashes (by error messages) for ONNXRuntime and **13** for TVM, while LEMON triggers none and GraphFuzzer only triggers 1 crash for each of ONNXRuntime and TVM. For instance, the only ONNXRuntime bug detected by GraphFuzzer is the wrong fusion to a double-precision *ReLU-Clip* connection (element-wise and thus shape-preserving).

We next describe transformation and conversion bugs we found by illustrating prominent bug patterns with examples. We use ★ to denote bugs *exclusively* found by NNSmith.

Transformation bugs. *Wrong expression simplification:* We found 5 such bugs in ONNXRuntime (4) and TVM (1). One bug★ happens in `FuseMatMulScale` when ONNXRuntime optimizes $(s_a \cdot A)@(s_b \cdot B)$ to $(s_a \cdot s_b) \cdot (A@B)$ for scalars s_a, s_b and matrices A, B where @ denotes *MatMul*. However, when B is a 1×1 matrix, ONNXRuntime can mistake matrix B as a scalar and rewrite it into $(s_a \cdot B) \cdot (A@s_b)$, which is illegal as *MatMul* does not accept scalar inputs, causing a compiler exception. Prior work cannot use the non-shape-preserving *MatMul* operator, thus missing such bugs. Wrong expression simplification can also lead to *semantic bugs*, which may lead to wrong decisions in downstream AI applications, introducing security threats in critical scenarios (*e.g.,* self-driving). For example, TVM has a buggy arithmetic optimization pass that switches the order of division and multiplication when rewriting $\lfloor \frac{x \bmod y}{i} \rfloor \times i \bmod z$, simplifying it to $(x \bmod y) \bmod z$ incorrectly.

Wrong layout analysis: Memory layout optimizations in TVM first rewrite layouts of the most beneficial operators (*e.g., Conv2d*) to efficient ones and then let remaining operators adapt changed layouts. We found 7 layout transformation bugs★ in TVM, related to non-shape-preserving operators including broadcasting, reduce and slicing, which cannot be handled by prior work. For example, TVM can rewrite NCHW *Conv2d* to the SIMD-friendly $N\frac{C}{4}HW4c$ layout (NCHW4c for short), by packing every 4 elements on C to the new sub-dimension (4c). However, using this optimization when the *Conv2d* is followed by a *Slice* operator whose stride for C is greater than one causes TVM to crash. GraphFuzzer cannot find this bug because to ensure shape alignment it always uses a stride of 1.

Integer type mismatch: DL compilers, like traditional compilers (*e.g.,* LLVM [26]), leverage IRs to simplify optimization. IR type mismatch can happen if one pass makes wrong assumption for the IR being transformed. This is especially a pain for TVM: we found 9 bugs★ stopping the compilation due to `int32`-`int64` mismatch and one core TVM developer also admits that *"TVM has a pretty fragile system of using i32 vs i64; I personally experienced it a few times before..."*. `int64` is often introduced by shape-related operators (*e.g.,* shape attributes of *Reshape* and *BroadcastTo*), which are not supported by prior work as they cannot handle those complicated shape constraints. Since our first bug report on such issues, there have been 12 fixes (7 from us and **5** from followers) within 5 months to resolve similar issues, one of which even blocked models in production. Interestingly, a bug we found also helped the developers find another bug that had previously been diagnosed as the outcome of a flaky test [34].

Conversion bugs. *Wrong scalar handling*: We found 6 crash bugs★ triggered when TVM imports *reduce*-like operators with a scalar input. Since these operators are not shape-preserving, prior work cannot trigger such bugs. Similarly in PyTorch, when exporting *Log2* with a scalar input, the exporter mistakenly sets its output to a rank-1 tensor instead of a scalar, causing a *semantic* issue. A few days after our report, developers identified **37** other *similar bugs*. Concurrently, NNSmith also identified a subset of these bugs, but in our evaluation we only treat the first bug (*Log2*) as one found by NNSmith.

Wrong broadcasting: Given a 3-way broadcasting operation $Where(C_{1\times1}, T_{3\times1}, F_2)$, a TVM bug★ causes the lower-ranked tensor F_2 being ignored during shape inference, resulting in the wrongly inferred shape 3×1, which should be 3×2. This incurs a compiler failure in later phases. Another TVM bug★ causes an import failure to *MatMul* with single-rank broadcasting (one input is a vector) and notably, one month after our bug report, real-world TVM users also encountered such issues and pushed for its fix, showing that NNSmith can synthesize real-world model patterns. Prior work cannot detect them since their design are incompatible with broadcasting operations.

Data type mismatch: Operators' data type supports vary by ONNX versions, which are often mishandled. For example, PyTorch can mistakenly (and silently) export *Clip* whose data type is `int32` which is not supported by ONNX version 11. Such ill-formed models will be rejected by most compilers; however, it can also be mistakenly compiled by TensorRT, producing unexpected model outputs (*i.e.,* semantic bugs in TensorRT), due to the wrongly interpreted attributes.

False alarms. As we discussed in the introduction, floating point semantics [49, 51] mean that even correct optimizations can lead to scenarios where an optimized model's output differs from the reference output. Consequently, we check output equivalence by checking that the distance between model outputs,when scaled by their overall magnitude is small. However, in some cases valid optimizations can lead to a large relative change in outputs and produce false alarms. For example, when feeding the *Sigmoid* with a large value, the optimized output can be 1 whereas the reference one is less than (though very close to) 1. When it provides the input to a *Floor* operator, the optimized output will differ from the reference output by 1, To reduce false alarms, we use a high error tolerance in comparison. In addition, false positives share similar structural patterns (*e.g., Sigmoid* followed by *Floor* and *Equal* for float-pointing inputs) that we can easily filter. Thus it did not cause large trouble for us.

6 RELATED WORK

Since the first proposal of fuzzing [41], various techniques have been proposed for fuzzing systems of different application domains [6, 15, 30, 32, 36, 37, 43, 57, 58, 72]. In this section, we mainly talk about the most closely related work in DL system fuzzing and compiler fuzzing.

6.1 DL System Fuzzing

In recent years, a number of techniques have been proposed to test DL libraries and compilers. As one of the first techniques in this direction, CRADLE [38] directly runs existing DNN models on different DL libraries to detect potential inconsistencies via differential testing. Later on, AUDEE [21] and LEMON [59] further extend CRADLE by applying search-based mutation strategies on the DNN models and their inputs to cover more library code. While AUDEE mainly focuses on mutating layer parameters and weight/input tensors, LEMON further applies more advanced mutation rules, including layer deletions/additions. Meanwhile, to ensure correctness of generated models, LEMON [59] only mutates type-preserving operators (or blocks of operators) from the real-world models, to avoid handling type constraints. However, there are many non-shape-preserving operator types, *e.g.*, even the commonly used *Conv2d* cannot be completely handled by LEMON. More recently, GraphFuzzer [35] allows a slightly larger operator search space using padding/slicing to align unmatched tensor shapes and also specifically controls the attributes of shape-changing operator types to create shape-preserving instances (*e.g., Conv2d* with kernel size/stride of 1). However, this design still substantially limits model diversity (as demonstrated in §2.3). The very recent (and concurrent) Muffin work [20] shares a similar limitation as GraphFuzzer: it uses "reshaping" layers to align tensor shapes during model generation; in addition, Muffin focuses on finding bugs in traditional DL libraries rather than DL compiler bugs. In this work, we aim to support more diverse/valid model generation for DL compiler fuzzing via a fundamentally different design powered by symbolic constraint solving [10] and gradient-driven search.

To complete DL system testing at the model level, researchers have also proposed DL system fuzzing techniques focusing on directly generating or manipulating the low-level model IRs [31, 50]. TVMFuzz [50] aims to automatically generate arbitrary low-level IRs based on a set of predefined grammar rules for fuzzing the popular TVM compiler [11]. The more recent Tzer work [31] leverages coverage feedback to perform joint mutation of both the low-level IR and optimization passes for TVM. While Tzer has shown promising results over TVMFuzz, the low-level IR mutation adopted by Tzer can hardly test the graph-level optimizations widely adopted by various DL compilers (as shown in §5.2).

In recent years, researchers have also investigated techniques to fuzz each DL library API in isolation. Since DL APIs are usually exposed in Python, a dynamically typed language, prior techniques, such as Predoo [70], require users to manually set up the function arguments, and can only be evaluated on a limited number of APIs. To address this challenge, FreeFuzz [61] dynamically traces API executions from various sources (including documents, developer tests, and model zoos), and generates new tests by mutating traced inputs. More recently, DeepREL [14] aims to automatically infer relational APIs (e.g., APIs that return the same values/statuses when given the same inputs), and then leverages such API relations as test oracle for catching more bugs than FreeFuzz. While such API-level testing techniques are adequate for testing first-generation DL libraries (§2.1), they can hardly find bugs in graph-level optimizations (e.g., 86% of the transformation bugs detected by NNSmith require multiple operators to trigger).

6.2 Compiler Fuzzing

As one of the most widely studied compiler fuzzing approaches in the literature [38], *grammar-based* techniques (such as Csmith [66], jsfunfuzz [55], and LangFuzz [23]) aim to generate syntactically valid input programs acceptable by the underlying compilers. While effective, it is hard for grammar-based techniques to ensure the semantic correctness of the generated programs to cover deep code paths, and highly specialized analyses have to be employed for specific languages. Therefore, various *mutation-based* techniques [16, 27, 28, 56, 69] have also been proposed for fuzzing compilers via mutating existing seed input programs. Moreover, given the advances in DL, researchers have also proposed *learning-based* techniques for compiler fuzzing. DeepSmith [13] and DeepFuzz [33] directly leverage recurrent neural networks (RNNs) to generate test programs from scratch, while Montage [29] performs mutation-based fuzzing, and replaces code snippets of the seed programs with new code fragments generated by RNNs. More recently, researchers have also leveraged the advanced pre-trained language models (e.g., GPT [9]) for more powerful test program generation for compiler fuzzing [67]. Such existing compiler fuzzing techniques can be potentially applied to the low-level IRs (C-like) for fuzzing DL compilers [31]. However, they can be hardly directly applied for graph-level DL compiler fuzzing, and our study has also shown the superiority of NNSmith over state-of-the-art IR-level DL compiler fuzzer.

7 CONCLUSION

NNSmith is a tool for generating diverse and valid test cases for deep learning compilers. It creates abstract operator models to ensure the validity of the generated models, and further utilizes incremental graph generation and attribute binning to ensure its diversity. To avoid false alarms and bug escapes, NNSmith leverages gradient search to find inputs that do not introduce NaN/Inf in the computation. NNSmith is easily extensible to support new operators with few lines of code. Lastly, NNSmith is implemented to generate models in the popular format ONNX and is readily applicable to any systems with ONNX support. To date NNSmith has found 72 new bugs in TVM, TensorRT, ONNXRuntime, and PyTorch, 58 of which have been confirmed or fixed, demonstrating its effectiveness.

ACKNOWLEDGMENTS

We thank the ASPLOS reviewers for their insightful comments. We also thank Yuanyi Zhong, Lingfan Yu and Leyuan Wang for insightful discussions in the early stages of the project, Jinjun Peng for helping open-source the project, and the the NYU IT High Performance Computing group for providing computing resources. This

work was partially supported by the National Science Foundation grants CCF-2131943 and CCF-2141474, a Google research award, a Meta research award, an AMD research award, and a gift from Microsoft Corporation.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains evidence of bug finding, source code of NNSmith's prototype, and user-friendly HTML documentation for re-generating the results. Specifically, it includes (1) links to bugs reported by the authors as real-world bug finding evidence, and (2) scripts and code to re-generate main results in § 5. To make artifact evaluation as simple as possible, our artifact is packaged into a pre-built docker image, along with a detailed and friendly HTML documentation. To fully evaluate the artifact, a X86-CPU platform with docker access is needed, with approximately 21 hours of machine time and 1 hour of manual inspection time.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Linux.
- **Hardware:** X86 CPU.
- **Metrics:** C++ source-level branch coverage.
- **How much disk space required (approximately)?:** 256GB.
- **How much time is needed to prepare workflow (approximately)?:** A few minutes to install and import the docker container image.
- **How much time is needed to complete experiments (approximately)?:** About 21 hours of machine time and 1 hour of manual inspection time.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache-2.
- **Archived (provide DOI)?:** 10.5281/zenodo.7222132

A.3 Description

A.3.1 How to access. The artifact can be downloaded from https://zenodo.org/record/7222132.

A.3.2 Hardware dependencies. A computer with X86 CPU and disk space of over 256 gigabytes.

A.3.3 Software dependencies. Docker.

A.4 Installation

There are two common ways to install the docker image:

To directly install the docker image from the Docker Hub (https://hub.docker.com/repository/docker/ganler/nnsmith-asplos23-ae) repository: `docker pull ganler/nnsmith-asplos23-ae`

To import the image locally from the Zenodo repository (https://zenodo.org/record/7222132):

```
tar xf NNSmith-ASPLOS23-Artifact.tar.gz
export IMG=ganler/nnsmith-asplos23-ae:latest
cat nnsmith-ae.tar | docker import - $IMG
```

A.5 Experiment workflow

Please open `html/index.html` in your browser or use the online documentation (http://nnsmith-asplos.rtfd.io/) for detailed installation and evaluation steps.

The overall logical workflow of experiments includes:

1. Install and run the docker image.
2. Running NNSmith and baselines on instrumented TVM and ONNXRuntime for 4 hours each to get code coverage reports.
3. Visualize coverage in coverage trends and Venn diagram.

A.6 Evaluation and expected results

All instructions and steps to reproduce our experimental results are in the artifact documentation (`html/index.html` or http://nnsmith-asplos.rtfd.io/).

The steps for main experiments are included in the "Evaluating artifact" chapter which re-generates results in § 5.2 and § 5.4. We also include instructions to re-generate non-main experiments (*i.e.,* ablation study in § 5.3) in the "Read more" chapter.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[3] Tadas Baltrušaitis, Chaitanya Ahuja, and Louis-Philippe Morency. Multimodal machine learning: A survey and taxonomy. *IEEE transactions on pattern analysis and machine intelligence*, 41(2):423–443, 2018.

[4] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *ArXiv*, abs/1308.3432, 2013.

[5] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M Wintersteiger. Programming z3. In *International Summer School on Engineering Trustworthy Software Systems*, pages 148–201. Springer, 2018.

[6] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[8] Marcel Böhme, László Szekeres, and Jonathan Metzman. On the reliability of coverage-based fuzzer benchmarking. In *44th IEEE/ACM International Conference on Software Engineering, ser. ICSE*, volume 22, 2022.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[10] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[12] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[13] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 95–105, 2018.

[14] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational api inference. *arXiv preprint arXiv:2207.05531*, 2022.

[15] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 725–730, 2014.

[16] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[17] gbftdlie. Found result inconsistency in graph-based fuzz testing. https://github.com/gbftdlie/Graph-based-fuzz-testing/blob/master/BugDetails_DCF.md, 2020.

[18] Google. Keras, 2015. https://keras.io.

[19] Google. More about afl – detecting new behaviors, 2019. https://afl-1.readthedocs.io/en/latest/about_afl.html#detecting-new-behaviors.

[20] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing. *arXiv preprint arXiv:2204.08734*, 2022.

[21] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. Audee: Automated testing for deep learning frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–498. IEEE, 2020.

[22] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[23] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[24] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[25] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[26] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[27] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[28] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.

[29] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630, 2020.

[30] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[31] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. Coverage-guided tensor compiler fuzzing with joint ir-pass mutation. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.

[32] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. Pmfuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 487–502, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):1044–1051, Jul. 2019.

[34] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 643–653, 2014.

[35] Weisi Luo, Dong Chai, Xiaoyue Run, Jiang Wang, Chunrong Fang, and Zhenyu Chen. Graph-based fuzz testing for deep learning inference engines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 288–299. IEEE, 2021.

[36] Weiyu Luo and Brian Demsky. C11tester: A fuzzer for c/c++ atomics. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*, April 2021.

[37] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[38] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[39] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[40] Microsoft. Onnx runtime: cross-platform, high performance ml inferencing and training accelerator. https://onnxruntime.ai/, 2020.

[41] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[42] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[43] Ian Neal, Andrew Quinn, and Baris Kasikci. Hippocrates: Healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–414, 2021.

[44] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *ICML*, 2011.

[45] NVIDIA. https://nvidianews.nvidia.com/news/nvidia-inference-breakthrough-makes-conversational-ai-smarter-more-interactive-from-cloud-to-edge, 2021.

[46] NVIDIA. Nvidia tensorrt, 2022.

[47] ONNXRuntime. Graph optimizations in onnx runtime. https://onnxruntime.ai/docs/performance/graph-optimizations.html, 2022.

[48] ONNXRuntime. Symbolic shape inference in onnxruntime. https://github.com/microsoft/onnxruntime/blob/master/onnxruntime/python/tools/symbolic_shape_infer.py, 2022.

[49] Michael L Overton. *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001.

[50] David Pankratz. Tvmfuzz: Fuzzing tensor-level intermediate representation in tvm. https://github.com/dpankratz/TVMFuzz, 2020.

[51] Douglas M. Priest. On properties of floating point arithmetics: Numerical stability and the cost of accurate computations. Technical report, UC Berkeley, 1992.

[52] PyTorch. Conv2d — pytorch 1.11.0 documentation, 2021. https://pytorch.org/docs/1.11/generated/torch.nn.Conv2d.html.

[53] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR, 2021.

[54] Guido Rossum, Barry Warsaw, and Nick Coghlan. Pep 8 – style guide for python code, 2013.

[55] Mozilla Security. jsfunfuzz. https://github.com/MozillaSecurity/funfuzz, 2007.

[56] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 849–863, 2016.

[57] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 697–709, 2022.

[58] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. *arXiv preprint arXiv:2102.02308*, 2021.

[59] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 788–799, 2020.

[60] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. The implementation repository of LEMON: Deep Learning Library Testing via Effective Model Generation, 2021. https://github.com/Jacob-yen/LEMON.

[61] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. *arXiv preprint arXiv:2201.06589*, 2022.

[62] Westley Weimer. Patches as better bug reports. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 181–190, 2006.

[63] Wikipedia contributors. Venn diagram — Wikipedia, 2022. [Online; accessed 3-July-2022].

[64] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[65] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 627–638, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[67] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated conformance testing for javascript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 435–450, 2021.

[68] Michal Zalewski. American fuzzing lop (afl). https://lcamtuf.coredump.cx/afl/, 2018.

[69] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 347–361, 2017.

[70] Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. Predoo: precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 400–412, 2021.

[71] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.

[72] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. {TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 489–502, 2021.

Received 2022-07-07; accepted 2022-09-22