NeuRI: Diversifying DNN Generation via Inductive Rule Inference

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

Jinjun Peng*
Columbia University
New York, USA
jinjun.peng@columbia.edu

Yuyao Wang*
Nanjing University
Nanjing, China
yuyao6@outlook.com

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

ABSTRACT

Deep Learning (DL) is prevalently used in various industries to improve decision-making and automate processes, driven by the ever-evolving DL libraries and compilers. The correctness of DL systems is crucial for trust in DL applications. As such, the recent wave of research has been studying the automated synthesis of test-cases (*i.e.,* DNN models and their inputs) for fuzzing DL systems. However, existing model generators only subsume a limited number of operators, for lacking the ability to pervasively model operator constraints. To address this challenge, we propose NeuRI, a fully automated approach for generating valid and diverse DL models composed of hundreds of types of operators. NeuRI adopts a three-step process: (i) collecting valid and invalid API traces from various sources; (ii) applying inductive program synthesis over the traces to infer the constraints for constructing valid models; and (iii) using hybrid model generation which incorporates both symbolic and concrete operators. Our evaluation shows that NeuRI improves branch coverage of TensorFlow and PyTorch by 24% and 15% over the state-of-the-art model-level fuzzers. NeuRI finds 100 *new* bugs for PyTorch and TensorFlow in four months, with 81 already fixed or confirmed. Of these, 9 bugs are labelled as *high priority* or *security vulnerability*, constituting 10% of all high-priority bugs of the period. Open-source developers regard error-inducing tests reported by us as "high-quality" and "common in practice".

CCS CONCEPTS

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

KEYWORDS

Fuzzing, Compiler Testing, Deep Learning Compilers

---

1 INTRODUCTION

The rise of Deep-Learning (DL) libraries and compilers has enabled emerging AI applications, such as AI chatbots [36], art generators [37] and autonomous driving, powering hundreds of millions of users. These complex systems have become increasingly adopted and ever evolving. For example, PyTorch [38] and TensorFlow [9], the most popular DL systems with 62k and 171k GitHub stars respectively, are moving toward their next major version (*i.e.,* PyTorch 2 [5] and TensorFlow 3 [2]), aiming at better model compilation support. However, taking PyTorch's new compiler [41] as an example, since birth (*i.e.,* 17 months) it is insufficiently tested by a test suite in *eight* thousand LoC. Consequently, it is crucial to harness the correctness of DL systems via extensive and automated testing. **The test-case generation problem** for DL systems is to synthesize a DNN model and its computational inputs. Additionally, generating *diverse* and *valid* models is essential for making high-quality tests.

(1) *Model diversity*: Effective DL system testing asks for model diversity coming from the variety of APIs, as well as the way they are composed. Additionally, to test the complicated DL compilers, it is important to generate models with multiple operators of various types for practicing the compiler passes [29].

(2) *Validity*: DNN models are programs [47] – for well-formedness they need to comply with validity constraints. Arbitrarily constructing and composing operators, such as creating pooling operators with negative kernel sizes or "connecting" operators with unwanted tensor shapes, oftentimes violate the constraints for constructing a well-defined model. As a result, argument errors (for DL libraries) or parser errors (for DL compilers) are raised before deeper system behaviours are tested.

Motivation. The model diversity primarily depends on the comprehensiveness of operators, which are the building blocks to a model. Prior work on single-API testing [14, 53] can generate a large body of API invocations (including both operator and utility APIs) via mutation or generation which comply with high-level type constraints or the plausible value sets. Can we directly apply such high-level information to generate valid DNN models? Unfortunately, it is impractical. Because
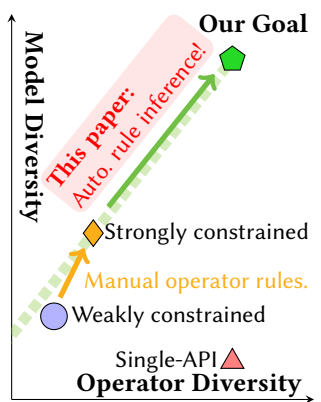

Figure 1: Test-case diversity.

constructing a valid API invocation further requires satisfying fine-grained constraints between operator attributes (*i.e.,* non-tensor arguments such as kernel sizes and strides) and input tensor types[1]

---

[1]Following prior work [29], a tensor type is a tuple of its shape and data type.

(particularly shapes). For example, while single-API testers may understand that conv2d accepts an image and a weight tensor of floating-points (*i.e.,* type constraints), their *newly* created conv2d invocations are not guaranteed to have the channel dimension of the image matching that of the weight (*i.e.,* shape constraints). Consequently, such attributes violate the validity properties required by conv2d and lead to invocation failures. Without understanding such fine-grained constraints, it is unlikely to correctly compose various APIs for constructing well-formed and diverse models. Intuitively, in Figure 1 prior single-API testers ▲ can achieve ideal API diversity when APIs being validly constructed. However, the diversity hardly extends at model-wise which requires multiple APIs to be constructed and "connected" correctly simultaneously.

Meanwhile, there are two categories of proposals for constructing valid models. *Weakly constrained model generation* ◯ [19, 32, 51] limits the use of APIs to those with simple and straight-forward constraints. For example, LEMON [51] only uses shape-preserving operators that have no input constraints. Consequently, such operators can be arbitrarily constructed and added to build a model. More recent work [19, 32] additionally inserts "reshaping" layers such that reshaped output tensors can stay in compatible shapes. However, it still may not construct operators with valid attributes (non-tensor arguments). Even worse, using such "layer wrappers" compromises the structural diversity of the models, which can overlook compiler passes activated by specific patterns. To support more diverse APIs correctly, NNSMITH [29], as a *strongly constrained* ◆ approach, defines a specification for describing input constraints and shape propagation (elaborated in §2). Nonetheless, it requires manual efforts for specifying those rules. For example, while a DL framework, *e.g.,* PyTorch, can define over two thousand APIs, only about sixty are supported by NNSMITH after its first-year development. Hence, it can take years for NNSMITH [29] to completely support a framework, *i.e.,* from ◆ to ⬠, which is unscalable.

Insight. Can we scale the diversity of model generation by enabling more operators (*e.g.,* by hundreds) *fully automatically*? We start to answer this question from two insights: *(i)* Empirically we observed that most operator rules are simple, *e.g.,* consisting of arithmetic expressions for shape computation and if-else branches for handling conditions incurred by some attributes. As a result, it is feasible to search a program that functions as operator rules, given the size of the problem is acceptable. Specifically, by instrumenting DL API invocations, we can obtain a set of input-output examples, with which the inference of operator rules can be regarded as a inductive program synthesis problem [28, 52]. *(ii)* Can an operator still be used for model generation even if its operator rule is not available? We find it *feasible* by inserting a "concrete" operator initialized by recorded invocation traces. To make use of both symbolically and concretely obtained operators, we can apply a *concolic* model generation approach to construct models with both sources.

Summary. This work makes the following contributions:

- In this work, we present the urgency for improving API diversity of model generation and formally introduce the essential properties for generating valid DNNs – *operator rules.* Furthermore, we open the first proposal of automatically inferring operator rules for diversifying and scaling valid model generation.

```
avg_pool2d(input, (kh=2, kw=2), padh=0, …) → output
```
Figure 2: The symbolic view of `avg_pool2d`.

- We build NEuRI (NEUral Network Synthesis via Rule Inference), a fuzzer for testing DL systems with three steps: (i) an instrumenter that collects and augments API invocations from various sources; (ii) an optimized rule synthesizer that efficiently infers operator rules with inductive program synthesis; and (iii) a hybrid model generator that compiles both symbolic and concrete information for producing valid and diverse DNNs.
- We extensively and rigorously evaluated NEuRI. Within four months, NEuRI finds 100 *new* bugs for PyTorch and TensorFlow, with 81 fixed or confirmed. 9 of the PyTorch bugs are labelled as *high priority* or *security vulnerability*, constituting around 10% of all high-priority bugs in PyTorch's bug tracker of the period. By evaluating branch coverage, NEuRI improves the state-of-the-art model-level fuzzer by 15% (PyTorch) / 24% (TensorFlow).

2 OPERATOR RULES

The functionality of a deep-learning model (*i.e.,* DNN) can be represented as a list of operations, each of which transforms one or multiple input tensors (*i.e.,* multi-dimensional arrays) to output tensors. Accordingly, a test-case in DL systems constructs a DNN and is evaluated over some computational inputs, expecting the model can be successfully executed and produce correct results.

For generating effective test-cases automatically, it is crucial to generate and diversify valid DNN models. State-of-the-art NN-SMITH [29] constructs valid DNNs with operator constraints and shape propagation rules. With SMT solvers, such rules can help statically construct an operator which can be safely inserted to a given model. Because in DL frameworks such operator rules are implicitly defined and cannot be exported directly, they are manually specified in NNSMITH. However, crafting them from scratch is unscalable. For example, in the first-year development of NNSMITH, only around sixty operators are implemented with rules, despite the fact that many rules are even repetitive. As a result, for diversifying operators being used and saving manual effort of domain experts, we aim at inferring those operator rules *automatically*. We now formalize and elaborate the operator rules:

Symbolizing operators. As is shown in Figure 2, an operator is a function that takes input tensors (*e.g.,* input) and configurations (*e.g.,* kh) as arguments. The configurations, also known as operator attributes, describe high-level semantics for performing an operation and can impact the operator rules. For example, kw and kh define the size for applying the "avg" filter over the input image which must be no smaller than the kernel size (assume no padding). For being evaluated *statically*, operator rules only leverage and symbolize an operator's compile-time information: (i) operator type (*e.g.,* avg_pool2d), (ii) I which is a list of input shape vectors, and (iii) A as the set of operator attributes. Runtime information such as the detailed element values inside the input tensors is too costly to be modeled. Meanwhile, we use O to denote the output shapes produced by the shape propagation rule.

Rule #1: Input constraints. The input constraints of an operator are a set of *predicate functions* $C = \{c_1, c_2, \cdots\}$ over $A \cup I$. For

example, constraints in avg_pool2d require the kernel size to be no larger than the padded image size (*e.g.,* $i_h + 2 \times a_{padh}$), namely:

$$c_k \left(A = \{a_{kh}, a_{padh}, \cdots\}, I = [[i_c, i_h, i_w]] \right) = a_{kh} \leq i_h + 2a_{padh}$$

The arguments of c_k consist of the attributes to avg_pool2d and the shape list with the shape of the only input tensor (*i.e.,* $|I| = 1$). It is worth noting that for clarity we assume the input is a non-batch image with only three dimensions (*i.e.,* the channel, height, and width); however, in practice, 2d-pooling also accepts batched inputs with an extra batch dimension. Meanwhile, some operators could take a variable length of inputs (*e.g.,* concatenate) or outputs (*e.g.,* split). For being general, a predicate may not assume the *tensor signature, i.e.,* # of input/output tensors and their ranks, to be fixed. Thus, operator rules with such patterns may need to be described with conditional branches (*e.g.,* the syntax of PyTea [25]). We later in §3.2 introduce how to leverage *partial operators* to simplify such branches which are difficult to handle in rule inference.

Rule #2: Shape propagation. Because evaluating Rule 1 requires knowing the dimensions of operator inputs, which are outputs of other operators from the DNN under construction, output shapes of operators also need to be evaluated. The shape propagation rule for an operator can be described by a function $\mathcal{P}$ over $A \cup I$, which returns a list of propagated output shapes as O. For instance, the shape propagation for avg_pool2d can be described as:

$$\mathcal{P} \left(A = \{a_{kh}, a_{padh}, \cdots\}, I = [[i_c, i_h, i_w]] \right) = [[o_c, o_h, o_w]]$$

$$\text{where} \begin{cases} o_c = & i_c \\ o_h = & \left\lfloor \frac{i_h + 2 \times a_{padh} - a_{kh}}{a_{stride}} + 1 \right\rfloor \\ o_w = & \left\lfloor \frac{i_w + 2 \times a_{padw} - a_{kw}}{a_{stride}} + 1 \right\rfloor \end{cases} \tag{1}$$

For example, given input shape of [3,3,3], we can tell the corresponding output shape for operator in Figure 2 (assuming a_{stride} is 1) is [3,2,2] without invoking it.

3 APPROACH

Figure 3 shows the overview of NEURI's workflow.

- NEURI improves the search space of model generation by making use of concrete invocation. To collect those *desired* invocations of tensor APIs, we instrument various sources such as developer tests. Next, we filter out invocations that do not meet properties such as determinism, to facilitate rule inference and bug detection in later phases. For the convenience of rule inference, we summarize the invocation records to a simplified structure and further augment data diversity via mutation (§3.1).
- These records are discrete data points, with which we can inductively synthesize arithmetic expressions in their corresponding operator rules. The inductive program synthesis problem is, however, an NP-hard [24] problem, whose complexity rests with the grammar under enumeration. For affordability, we split a complete operator rule into multiple sub-rules, in order to be describable by a simple arithmetic grammar. Furthermore, we prune the enumeration space over *equivalence* and *rarity*, and as a shortcut, reuse rules when possible. Additionally, redundant input constraints are removed for runtime efficiency.

- Next we apply *Hybrid DNN Generation* which performs DNN generation over (i) symbolic operators, *i.e.,* those with operator rules; and (ii) concrete operators, *i.e.,* those whose rules are not inferred but with concrete invocation records. To achieve this, we perform *concolic operator insertion* where symbolic operators are inserted with SMT solving while concrete ones are inserted by searching a compatible tensor type (*i.e.,* shape and data type).
- Lastly the generated model, after materialization, is cross-checked between the interpreter and compiler via oracles in §3.4.

3.1 Instrumentation

Invocation collection. Following prior work [14], we instrument the desired APIs to store *successful* invocations locally. In contrast to FreeFuzz which comprehensively instruments APIs from both tensor and high levels, we focus on *tensor APIs* since high-level APIs can be decomposed to a series of tensor operations. Additionally, these tensor APIs must be deterministic and value-independent (detailed in §4). We perform instrumentation at Python level (*e.g.,* over the Python test-suite) since Python is commonly used as the front-end of DL frameworks. Meanwhile, we simplify the disorganized and superfluous raw invocation (Figure 4) by dropping concrete tensor values and thereby only preserve the tensor types, functor and other arguments. For instance of avg_pool2d (Figure 2), the layout of its simplified record[2] is illustrated in Figure 4.

Data augmentation via mutation. The robustness of inferred rules depends on the quantity and quality of records. Unfortunately, by only using instrumented records, each rule on average only shares 5-7 records, which are insufficient. More importantly, all collected records are *passing examples*; however, *counter examples* are also required for inferring input constraints (§3.2). Consequently, we further diversify the records by mutating existing records, where valid mutants are used as *passing examples* (denoted by $\mathcal{R}^{\checkmark}$) and invalid ones are used as *counter examples* (denoted by $\mathcal{R}^{\times}$). Specifically, we perform three phases of mutations over input shape dimensions and attributes of records (*i.e.,* $A \cup I$):

(1) *Offset-based*: The goal of offset-based mutation is to quickly build a large set of (preferably) passing examples. To achieve this, we enumerate subsets over $A \cup I$, for each of which we increment the elements by 1 until a desired number of records (*e.g.,* 100 in our experiments) or time budget runs out. The hypothesis behind is *validity locality*: oftentimes validity is preserved after a light-weight mutation (*i.e.,* increment by 1).

(2) *Swapping*: Exchanging two values from $A \cup I$ can quickly verify simple inequalities. For example, assuming $a > b$ holds in all collected passing examples, we invalidate the inequality if the record is still valid after exchanging the values in a and b.

(3) *Special values*: Lastly, we randomly assign attributes with special values (*e.g.,* 0/-1) in order to test attributes' negativity.

Meanwhile if no counter examples are produced after sufficient mutation, it means probabilistically it has no (or extremely weak) input constraints. Consequently, for counter-example-free operators, we directly assign an empty set as its input constraints (*i.e.,* no need to infer input constraints with inductive synthesis in §3.2).

---

[2]For clarity, we use "record" to represent "simplified invocation records" from now on.
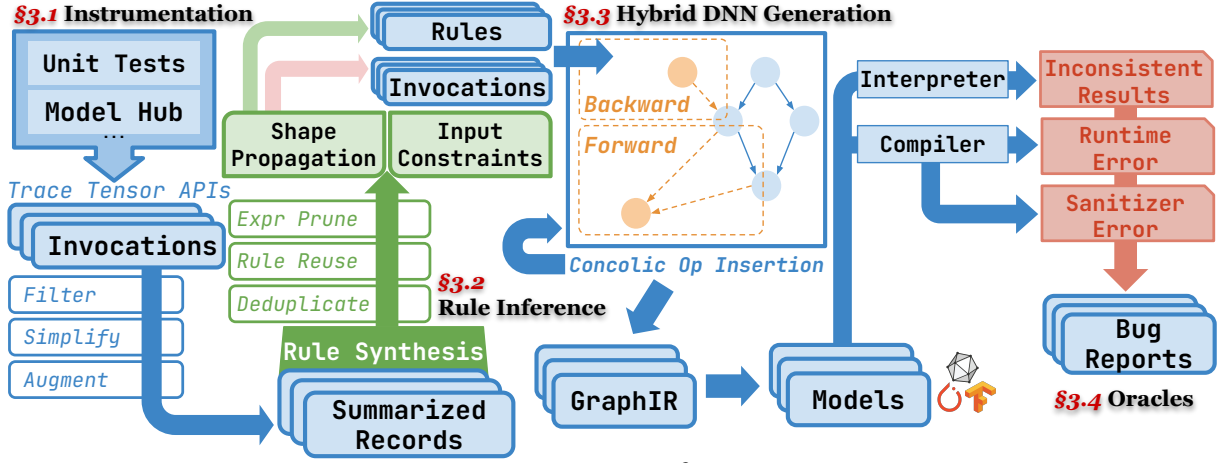
Figure 3: Overview of NEURI

Figure 4: Layout of records before and after simplification.

3.2 Rule Inference

We now explain how to perform rule inference via inductive program synthesis and make it affordable with a line of optimizations. **Inductive synthesis of operator rules.** Input constraints and shape propagation rules can be described by functions (over $A \cup I$) whose bodies are oftentimes arithmetic expressions (§2). Specifically, we can define such an arithmetic grammar $\mathcal{G}$ as follows:

$$
\begin{aligned}
\langle\text{expr}\rangle &::= \langle\text{op}\rangle \langle\text{expr}\rangle\langle\text{expr}\rangle \mid \langle\text{item}\rangle \\
\langle\text{op}\rangle &::= + \mid - \mid \times \mid \div \mid \min \mid \max \mid \mod \\
\langle\text{item}\rangle &::= \langle\text{symbol}\rangle \mid \langle\text{constant}\rangle \\
\langle\text{symbol}\rangle &::= \textit{Symbols from } I \textit{ and } A \\
\langle\text{constant}\rangle &::= \textit{Constant integers}
\end{aligned}
$$

With the grammar, we can infer an operator rule via inductive program synthesis, *i.e.,* by enumerating $\mathcal{G}$ to find an expression that matches inputs/outputs of the records, in certain time budget and program size. Because expressions in operator rules tend to be short, we perform *bottom-up* enumerative search [10, 49] which first constructs small terms (*e.g.,* ⟨item⟩) and compose them gradually for generating larger ones. For clarity we denote the set of enumerated expressions to be $\mathcal{E}$. Meanwhile, there are a few hypotheses to consider, *e.g.,* $|A|$, $|I|$ and $|O|$ are assumed be fixed. We will detail them later in the "partial operator" paragraph.

Formula 2 describes a shape propagation rule with a set of expressions for computing corresponding output dimensions. To infer the propagation expression for the i-th output dimension (*i.e.,* o_i), we enumerate $\mathcal{E}$ until $\text{expr}_k \in \mathcal{E}$ is found to match all records (Formula 3). Otherwise, we say the rules are not inferred and will not be used for inserting *symbolic operators* during model generation.

$$\mathcal{P} \cong \{o_1 = \text{expr}_1(I, A), \cdots, o_n = \text{expr}_n(I, A) \mid \text{expr}_i \in \mathcal{E}\} \quad (2)$$

$$\exists \text{expr} \in \mathcal{E}, \forall\langle A^\star, I^\star, O^\star\rangle \in \mathcal{R}^\checkmark, \text{expr}[A^\star \cup I^\star/A \cup I] = o_i[O^\star/O] \quad (3)$$

Similarly, input constraints C are predicates of equalities and inequalities, which can be normalized to $0 = \text{expr}$ and $0 < \text{expr}$ respectively. Algorithm 1 illustrates how C, starting with an empty set (Line 2), is inferred. By enumerating predicates oriented from $\mathcal{E}$ within the time limit (Line 3-4), we find predicates that are satisfied by all passing examples (Line 5). Specifically, if any passing example does not match the predicate under enumeration (*i.e.,* c) (Line 6), c is then undesired, and consequently we restart the loop for the next predicate (Line 7). Otherwise, we include c in C (Line 8). Meanwhile, for soundness input constraints should reject invalid inputs (if any). Consequently, C should reject all (if any) counter examples (Line 9-10). Otherwise, the rule is not inferred (Line 11).

Algorithm 1: Inference of input constraints C

1 **Function** InferInputConstraints($\mathcal{E}, \mathcal{R}^\checkmark, \mathcal{R}^\times$):
2      $C \leftarrow \emptyset$
3   LABEL: **for** $c \in \{0 = \text{expr}, 0 < \text{expr}; \forall \text{expr} \in \mathcal{E}\}$ **do**
4   **if** *timeout* **then break**
5   **for** $\langle A^\star, I^\star\rangle \leftarrow \mathcal{R}^\checkmark$ **do**
6   **if** evaluate($c[A^\star \cup I^\star/A \cup I]$) *is false* **then**
7   **continue** LABEL // Go to next c at Line 3
8          $C \leftarrow C \cup \{c\}$
9   **for** $\langle A^\star, I^\star\rangle \leftarrow \mathcal{R}^\times$ **do**
10   **if** evaluate($\bigwedge C[A^\star \cup I^\star/A \cup I]$) *is true* **then**
11   **raise** Inference failure
12   **return** C

Partial operator. In NNSMITH, operator rules are directly written in Python, whose grammar is much more complicated than $\mathcal{G}$. Running inductive program synthesis over such a complex grammar,

Figure 5: Examples of similar but distinct partial operators. "f32[3]" stands for a float32 tensor variable whose rank is 3.

though being more capable, is impractical. To preserve a grammar as simple as $\mathcal{G}$, we split an operator into multiple *partial operators*, by "fixing" components whose variation incurs a more complicated grammar. We can empirically summarize such components for defining partial operators. For example, branches in operator rules are used for handling variable lengths of inputs or input ranks. Additionally, rules of operators with dimension-sensitive attributes, *e.g.,* dim in max(x, dim), often requires array operations. With Figure 5, in addition to API names (*e.g.,* ❶ and ❷), we identify a partial operator with the following properties:

(1) *Tensor signature*: Recall §2 that an operator could take and return a variable length of tensors in various ranks. Because incorporating such variability is costly, we let each partial operator have a fixed tensor signature (and thus a fixed form of I and O). For example, ❷ and ❸ have the same API name but are different partial operators for having different input/output ranks (*i.e.,* 3 versus 4). It is also worth noting that we do not distinguish partial operator over the data types of input/output tensors which are often orthogonal to the operator rules.

(2) *Symbolic attributes*: Besides input tensors, we regard other arguments that can be symbolized to *symbolic integers* as *symbolic attributes*, which are the free variables in operator rules (*i.e., A* in §2). Therefore, invocations with different sets of symbolic attributes are associated with different partial operators.

(3) *Other arguments*: We further classify the rest of arguments (*i.e.,* non-tensor and non-symbolic-integer) into two categories: *(i)* rule-orthogonal arguments (*e.g.,* float-point scalars such as "bias") and *(ii)* (likely-)rule-dependent arguments (*e.g.,* image layout in "NCHW" or "NHWC"). Only *(ii)* will be used for identifying partial operators for its potential impact on operator rules. In general, the sub-category of an *other argument* is determined by its type and value. For instance, in Figure 5, the ceil argument, as a boolean, falls into the *(ii)* category, which makes ❸ and ❹ different partial operators. In fact, ceil impacts the shape propagation rule of avg_pool2d, where being true makes the output height and width rounded by *ceil* instead of *floor* (see Formula 1). Further details will be elaborated in §4.

Pruning. $\mathcal{E}$ can be too large to enumerate. For efficiency, we prune semantic-equivalent duplicates (*i.e., equivalence*), as well as those that are uncommon in operator rules (*i.e., rarity*). Specifically, we list the pruning methods in their order of being applied:

(1) *Bound*: Without constraints, $\mathcal{E}$ is infinite. Therefore, we bound $\mathcal{E}$ by limiting the number of $\langle op \rangle$ and the set of constant literals.

For example, in our default experimental setting the maximum number of $\langle op \rangle$ being used is 5 and we use $\langle constant \rangle \leftarrow \{1, 2\}$. Additionally, expressions describing inequality are further limited to have at most one $\langle op \rangle$ because *(i)* inequalities are oftentimes simple; and *(ii)* a larger upper limit will lead to many false-positives as inequalities are more flexible than equalities.

(2) *Rarity*: We empirically prune expressions with the same symbols occurring more than once, which are uncommon. Those with constant sub-expression (*i.e.,* $\langle op \rangle \langle constant \rangle \langle constant \rangle$) are also pruned for being constant foldable.

(3) *Equivalence*: We find semantically equivalent expressions in $\mathcal{E}$ and only keep the simplest one. Specifically, we leverage a two-pass approach inspired by Ruler [34]: First, for the expressions with the same free variables, we *quickly* evaluate them over a number of randomly generated assignments and group them according to the outputs (*i.e.,* often known as *characteristic vectors* or *finger-prints*). For each group, we then rigorously find equivalents by applying an SMT prover.

$\mathcal{E}$ may vary for different $A \cup I$. While pruning $\mathcal{E}$ for each partial operator is costly, we make it one-time effort by: (i) pruning $\mathcal{E}_\square$ with "holes" (*i.e.,* symbol placeholders); and (ii) extending $\mathcal{E}_\square$ to $\mathcal{E}$ by replacing the holes with actual symbols of each partial operators. For example, assume that $\mathcal{E}_\square$ is the pruned set of expressions with holes of $\{\square_1, \square_2, \square_3\}$, *i.e.,* $\langle symbol \rangle ::= \square_1 \mid \square_2 \mid \square_3$. To infer an operator rule with $A \cup I \leftarrow \{s_1, \cdots, s_4\}$, we get the actual $\mathcal{E}$ by extending $\mathcal{E}_\square$, by mappings $\{s_1, \cdots, s_4\}$ to $\{\square_1, \square_2, \square_3\}$ in various ways. More specifically, for each expr with h holes (symbols), we select h symbols from $A \cup I$ to "fill" the holes (*i.e.,* substitution). Because of the *one-time-occurrence* hypothesis, each symbol in $A \cup I$ will not be selected to fill multiple holes (*i.e.,* injective). In addition, the mapping from the selected symbols (in $A \cup I$) to holes is determined according to the relative order of indices. For example, for $\square_1 \div \square_2$, by selecting $\{s_1, s_2\}$ we only get $s_1 \div s_2$. Why not consider permutation over the mappings? Consider if we allow $s_2 \div s_1$ as an extension, when extending $\square_2 \div \square_1$ (indices swapped) we get the same duplicated expression. As a result, for each expr with h holes, we can extend $\binom{|A \cup I|}{h}$ expressions. Meanwhile, when $|A \cup I|$ is smaller than the maximum number of holes, we only consider extending expr where $h \leq |A \cup I|$.

Rule reusing. Partial operators can share equivalent rules. Before running inference from scratch, we can first test if the records can be matched by already inferred rules (if they share the same form of A, I and O). If an existing rule can be matched, we can simply "copy and paste" it for the new Partial operator as a short-cut; otherwise, we can still run inference from scratch. Furthermore, since this optimization is orthogonal to the grammar, we can also reuse those expert-crafted rules from NNSMITH (Python grammar).

Algorithm 2: Predicate deduplication for C

---

1 **Function** Deduplicate(C):
2   **repeat**
3   **for** $c \in C$ **do**
4   **if** Prove($\bigwedge [C] \Leftrightarrow \bigwedge [C - \{c\}]$) **then**
5                 $C \leftarrow C - \{c\}$
6   **until** C *unchanged*

Deduplication. The inferred input constraints could have many redundant predicates, which slows down SMT solving when fuzzing online and makes it less readable. Therefore, Algorithm 2 deduplicates the predicates obtained from Algorithm 1. We each time remove a predicate c if C is equivalent to that without c (Line 4-5). We run the algorithm until a fixed point when no predicates from C are removed after an iteration.

3.3 Hybrid DNN Generation

Following the Algorithm 1 in NNSmith [29], the generation of a DNN can be regarded as a problem of inserting a *valid* operator *correctly* to an already *valid* DNN model. In NNSmith, since all operators has their corresponding rules implemented by domain experts, a DNN model is synthesized *symbolically*, *i.e.,* all shape dimensions and attributes are viewed as symbols during construction and later materialized by a set of assignments offered by the SMT solver. For NeuRI, we adopt a *concolic* [17, 43] style of DNN generation in order incorporate both symbolic and concrete operators[3].

Specifically, the hybrid DNN generator inserts operators concolicly such that each operator, after insertion, is *immediately* concretized by the model from the solver, instead of deferring it when a full DNN is built. Therefore, the DNN under construction is always concrete, *i.e.,* all shape dimensions, data types and attributes are concrete at construction time, which makes it applicable to insert a concrete operator. There are a few benefits with a concrete DNN: *(i)* the insertion of concrete operators can be efficiently implemented by looking up compatible tensor types between traced records and the model under construction; and *(ii)* theoretically SMT solving is incurred less intensively (thus faster) with less symbols. In NNSmith, an operator can be inserted in two directions: 1) *forward insertion* that inserts an operator which consumes existing values; and 2) *backward insertion* that lets an operator be a producer by occupying existing placeholders. Next, we elaborate how symbolic and concrete operators are *forward* inserted and for clarity omit the details for backward insertion, which can be regarded as a reversed version over the placeholders (instead of arbitrary tensors).

Inserting a symbolic operator. Both the manually written operator rules (*i.e.,* NNSmith) and inferred ones (*i.e.,* NeuRI) can be used to insert symbolic operators. Operators in both groups are selected separately in equal amount of probability. Each time, to insert a selected operator ϕ, we first enumerate arity-sized combinations of tensor variables as the input candidates to ϕ, where each of them must respect the data type and rank requirements of ϕ. Taking the batched 2D-convolution as an example, whose input tensor must have four dimensions, any other tensors whose rank is not four will not be taken into the enumeration. Next, each of the input candidate tuples is checked by the input constraints C_ϕ until one satisfies C_ϕ and thus becomes the tensor inputs I to ϕ. Furthermore, we ask the SMT solver to provide a model from the input constraints and use the assignments of operator attributes $A^\star$ to initialize $\phi^\star \leftarrow \phi[A^\star/A]$. We then insert $\phi^\star$, taking I as inputs, to the DNN under construction. We also propagate its output tensor types with the shape propagation rule (Formula 2) for making future insertion feasible (§2). Of course, if none of the candidates

can make the rule satisfiable, the insertion of ϕ will be discarded and the algorithm will re-try another operator.

Inserting a concrete operator. The feasibility of inserting a concrete operator is as simple as finding an intersection of tensor types between inputs (*i.e., I*) in records and visible variables in the working DNN. For example, given a `avg_pool` which has input tensor type of `float32[1,3,224,224]` in the records, if there is a tensor variable with a shape of `[1,3,224,244]` and a data type of `float32` in the DNN under construction, we can safely insert it to the target place. However, because of the large volume of records, checking the satisfiability operator by operator and record by record is inefficient. Instead, we can build a mapping from tensor type (*i.e.,* shape plus data type) to a set of partial operators, any of which has an input tensor of such a type. Then, by accumulating the set of partial operators mapped from tensor types available in the working DNN, we get a reduced set of operator candidates which exclude those with unsatisfiable input types. Consequently, we only need to enumerate records of reduced sets of partial operators. Once a record if found to be matchable, we initialize the partial operators with the record and insert it to the DNN under construction.

3.4 Test Oracle

In this section, we list three test oracles for manifesting bugs.

Result inconsistency. In addition to running the DNNs *eagerly* with the pre-compiled library functions (*i.e., interpreter*), TensorFlow and PyTorch can further optimize the models via *compilation* for better performance. Hence, we cross-check the results obtained by running the same model and inputs from the *interpreter* and *compiler*, where a subtle floating-point error is allowed.

Runtime error. We identify a runtime-error bug if the compilation or execution of a model aborts unexpectedly. The corresponding symptoms include a crash or an unexpected Python exceptions not incurred by incompatibility (*e.g.,* "Not-implemented" error). Furthermore, interpreter exceptions are not considered as bugs as it could be caused the use of an incorrect operator rule.

Sanitizer error. We also enrich the test oracles with sanitizers, such as ASan [44] (memory error), UBSan [7] (use of compiler's undefined behavior), and CSan [3] (CUDA error). Sanitizer errors are reported by sanitizer-injected checkers at runtime. Without sanitizers, bugs may not manifest themselves via a crash (*e.g.,* buffer overflow) or occur at a late stage, making debugging challenging.

4 IMPLEMENTATION

NeuRI implements three components including an instrumentation tool, a rule synthesizer and a fuzzer, via a total effort of 14.9k LoC.

API instrumentation. The instrumentation tool is implemented with 1.8k LoC in Python. The instrumentation is performed by inserting an API-hijacking code snippet in the `__init__` files of DL framework packages. Specifically, we run the instrumentation over the developer tests from the open-source repositories of PyTorch and TensorFlow. As soon as the DL packages are imported, the API-hijacking code adds a function wrapper to all functions within the package. During execution (*i.e.,* running the regression tests of DL frameworks), the invocation snapshots to desired APIs are serialized for reproduction. In post-processing, a filter is applied to remove invocations that do not comply with *determinism* and *value*

---

[3]A *symbolic operator* is an operator with inferred rules; while a *concrete operator* does not have rules successfully inferred but still has its corresponding validated records.

independence. To detect *determinism,* each invocation is replayed for three times for checking output consistency. For testing *value independence, i.e.,* the operator rules are independent to the values/elements in the input tensor, each API is tested by three groups of random inputs (initialized from -10^6 to 10^6) and is expected to output the same tensor types without runtime failure. This component also includes utilities for parsing and composing/replaying a DL API, in order to construct new invocation.

Rule synthesizer. We implemented the synthesizer in 1.5k LoC in Python. Before the actual rule synthesis, we first apply data augmentation for enriching the records on demand (*i.e.,* until 100 records for each partial operator). In §3.2 it is found that not all arguments impact the rules, consequently we identify *rule-orthogonal* arguments in a partial operator through the argument type: for a floating-point argument (*e.g.,* bias) we assume it does not impact operator rules. Furthermore, the arithmetic expressions are represented as binary trees and for memory efficiency smaller trees are re-used to compose larger trees via pointers in the bottom-up enumerative search. We constrain the maximum number of ⟨op⟩ to 5 (*i.e.,* 6 symbols at most). As a result, as a one-time effort we first enumerate $\mathcal{G}$ by regarding the symbols as 6 "holes" and prune it on the fly to get $\mathcal{E}_\square$. Meanwhile, we also leverage commutativity of $\{+, \times, \min, \max\}$ to skip the enumeration of operand swapping and associativity over $\{+, -, \times, \div, \min, \max\}$, in order to accelerate the equivalence-based pruning. With $\mathcal{E}_\square$ obtained as a one-time effort, for any new operator rule under inference, we can quickly extend $\mathcal{E}_\square$ to $\mathcal{E}$ by filling its actual symbols into the "holes" as discussed in §3.2. Specifically, for each partial operator we set a timeout of 1000 seconds to infer the shape propagation or input constraints.

Fuzzer. The fuzzing engine of NeuRI is built by extending the NNSmith prototype [1] with 11.6k new LoC (and removing 7.9k old LoC). Major efforts are spent to improve the extensibility and debuggibility of the original NNSmith for benefiting algorithm prototyping and bug finding. Previously NNSmith uses directed multi-graphs in networkx [21] for describing DL models internally. However, the graph data structure is not suitable for manipulating model structures and being translated to real-world model formats. Additionally, DL models are fundamentally programs [47] which is not necessarily always pure data-flow graphs. For example, in-place operators for reproducibility require a total order during execution whereas traversing a graph cannot guarantee. As a result, we build an SSA-based intermediate representation, namely GraphIR, to describe DNN structures. Following the LLVM [27] interface style, DNN manipulation is made safe and convenient via three fundamental APIs of insert, remove_unused, and replace_alluse. Thanks to the extensibility, five graph generation strategies used in this paper, including three NeuRI variants and two NNSmith variants, are implemented in merely 1.1k LoC.

5   EVALUATION

We evaluate NeuRI by asking following research questions:

- **RQ1 (§ 5.2)**: How does NeuRI compare against state of the art in DL compiler fuzzing in terms of code coverage?
- **RQ2 (§ 5.3)**: How many APIs, partial operators and records are collected and eventually inferred with operator rules? How

efficient and effective is our rule synthesizer compared with general-purpose program synthesis tools such as Rosette [48]?
- **RQ3 (§ 5.4)**: How effective is NeuRI when detecting previously unknown bugs for real-world DL compilers?

5.1   Experimental Setup

Systems under test. We test the emerging *compilers* of the most popular DL frameworks, *i.e.,* TensorFlow [9] and PyTorch [38], which for clarity are denoted by "TF" and "PT" respectively.

(1) *TensorFlow XLA* compiler converts a TensorFlow model (*e.g.,* SavedModel) to its graph-level IR (*i.e.,* HLO) for running various optimization passes. TensorFlow defines over 1500 operators. Of these, around 450 are supported by XLA. This is because DL compilers often focus on a small set of primitive operators, from which other high-level operators can be composed.

(2) *PyTorch JIT*, the PyTorch's equivalent of XLA, supports around 1310 APIs (including alias, *e.g.,* torch.max(a) and a.max()) out of a total of over 2000 PyTorch operators.

Metrics. We evaluate NeuRI over various metrics. Specifically, we explain the most important two here and defer the others.

- *# Found bugs*: We count the bugs at the basis of bug reports, which are classified to four statuses: 1) *fixed*: A patch has been effectively applied to fix the bug; 2) *confirmed*: In addition to fixed bugs, we *conservatively* (*i.e.,* lower bound) identify a confirmed bug *iff* it has been reproduced/diagnosed as a fault or directly assigned to developers for fixing it; 3) *won't fix*: Developers claim the potential of not fixing it (*i.e.,* upper bound); and 4) the rest of bugs are all triaged but require further investigation.
- *Branch coverage*: Following [29, 31], we evaluate fuzzers with *branch* coverage, a stronger criterion (than line coverage) for test adequacy [57], over DL frameworks' C++ source code.

Baselines. In end-to-end benchmarks, we compare NeuRI with the state-of-the-art model-level fuzzers (namely NNSmith [29] and Muffin [19]) and the state-of-the-art operator-level fuzzer (*i.e.,* DeepREL [15]). For ablation study we also evaluate NeuRI's variants.

- NNSmith performs model generation with over 60 operators whose rules are *manually* crafted by domain experts. Specifically, the official NNSmith performs *pure symbolic* generation where the all symbols are materialized together when all operators in the graph are symbolically inserted. In this paper we also propose concolic generation (§3.3), consequently we also implemented a concolic version of NNSmith which immediately materializes the symbols per insertion. Because the concolic variant performs similarly as the pure-symbolic version, for clarity we omitted the results in evaluation. One reason can be that concolic insertion does not bring more operator supports in NNSmith as NeuRI.
- Muffin based on 11 seed models including DenseNet [23] and LSTM [22], performs mutation-based model generation. Similar to NNSmith, it supports over 60 operators by hand-crafting the shape inference rules. However, Muffin-created models are not guaranteed to be valid for the lack of input constraints. Notably, Muffin is only implemented on TensorFlow. As a result, we only compare Muffin against others on TensorFlow.
- DeepREL is an operator-level mutation-based fuzzer similar to FreeFuzz. As an improvement to FreeFuzz whose seeds purely

come from instrumentation, DeepREL extends the seed invocations by matching similar APIs and exchanging their arguments.

- $NeuRI^r$ is a variant of NeuRI where the use of inferred operator rules is disabled. In other words, $NeuRI^r$ constructs DNNs by either using concrete operators determined by collected records or symbolic operators from the original NNSmith and both methods share equal probability for being selected.
- The $NeuRI^i$ variant disables concrete operators and only uses symbolic operators from automated inference or NNSmith.

In addition, in RQ3 we also compare our rule synthesizer with Rosette, a solver-aided programming system, which supports inductive program synthesis. Specifically, we give Rosette $\mathcal{G}$ as the grammar under a bit vector theory. The number of bits for the data and operations is 32 given that the maximum number in records is $2^{32} - 1$ (*i.e.,* INT_MAX). Next, for each partial operator we let its records be the constraints and run Rosette with a 1000-second time budget. More specifically, we only compare with Rosette over the inference of shape propagation, *e.g.,* given an partial operator with k output dimensions, both Rosette and NeuRI will run k times each of which trying to search $expr_k$ that matches records of o_k. We did not infer input constraints for Rosette since multiple matched predicates can be returned in one pass while Rosette directly terminates when the first matched predicate is found.

Configuration. We run all experiments on Ubuntu 22.04 powered by a 64-thread AMD Threadripper CPU, 256 GB of memory, and 4 TB of PCIe-4 SSD. Our approach is evaluated over the up-to-date frameworks and versions: TensorFlow v2.12-nightly (git: 5a6fc06bf8) and PyTorch v2.1-nightly (git: f7520cb51e). Due to the different tool-chain flavours, we compiled TensorFlow with GCC-12.2 and GCOV [4], while PyTorch is compiled with Clang-14 and its source-code based coverage tool [6]. To precisely measure the test adequacy of compilation, for TensorFlow we instrument files in tensorflow/compiler (over 800k LoCs), and for PyTorch, non-kernel-function files under pytorch/csrc and aten/ are conservatively instrumented since PyTorch's passes are "everywhere". Following prior work [29, 31] we by default run fuzzing for four hours and generate models with five operators to balance between efficacy and debuggability (Figure 6c). For detecting result inconsistency, our oracle uses absolute error of 10^{-3} and relative error of 1%. With models generated as small as five nodes, we did not see false-positives brought by propagated floating-point errors.

5.2 RQ1: Evaluating Coverage

Overall coverage. Figure 6a and 6b show the coverage growth (y-axis) in four hours (x-axis). Among all *model-level* fuzzers, NeuRI improves the prior SOTA (*i.e.,* NNSmith) by 24% on TensorFlow and 15% on PyTorch. The results indicate that NeuRI can synthesize more diverse model structures to exercise various compiler passes. In addition, NeuRI also outperforms the SOTA *operator-level* fuzzer DeepREL by 7% on TensorFlow and 2% on PyTorch. Though existing operator-level fuzzers [14, 15, 53] is known for constructing invocations for thousands of APIs, NeuRI shows that by only supporting a smaller but essential set of operators, similar and even better code coverage can be achieved through model-level generation. Despite the marginal coverage improvement over DeepREL, NeuRI, via model-level fuzzing, can explore various system

behaviors that are not covered by DeepREL. Specifically, 62.9% of the PyTorch bugs and 35.7% of TensorFlow bugs found by NeuRI are manifested by models of multiple operators which cannot be covered by single-operator fuzzers like DeepREL. Overall, in four hours NeuRI automatically covered 10.8% and 17.4% of the compiler code branches in TensorFlow and PyTorch respectively. It is worth noting that achieving 10-18% of *total branch* coverage is non-trivial, given the complexity of DL frameworks. As a reference, fuzzers [26] for Linux (*i.e.,* also a million-LoC project) commonly achieve around 10% of block coverage. The missing branches can come from passes designed for other hardware targets (*e.g.,* GPU whereas CPU is used in our experiments) and unused experimental compilation pipelines (*e.g.,* MLIR).

Ablation study. By comparing NNSmith with $NeuRI^r$ and $NeuRI^i$, we show that both symbolic and concrete operators are effective for improving coverage (18.8 ∼ 20.9% for TensorFlow and 12.8 ∼ 13.5% for PyTorch). By combining both of them together (*i.e.,* NeuRI), $NeuRI^r$ and $NeuRI^i$ can be further improved to cover 2.2% / 1.3% more branches for TF / PT. Though this improvement might look marginal, we argue that the additionally covered branches are harder-to-reach ones, as we later show that a certain number of bugs are exclusively contributed by inferred rules. Table 1 shows that NeuRI has a slightly lower validity rate (< 5.2%) and runs 17.6% (TF) / 67.7% (PT) slower than NNSmith. The validity rate is lower as some inferred rules are partially correct. The speed of NeuRI is of course slower as its model generation tackles over 75× more symbolic operators than NNSmith and even much more concrete operators. However, NeuRI still achieves better coverage as it generates test-case of higher quality over quantity. Note that an non-prefect validity rate does not introduce false-positives, as we identify and immediately discard an invalid test-case if it raises exceptions in eager mode.

Speed. The time costs for generating, evaluating and saving a test-case are shown in Table 2. It shows that constrain-based model generation is fast, taking an average of 88ms (TF) / 69ms (PT) for creating and running a test-case, despite the single-thread nature of our model generator. Specifically, the generation phase on average takes 53% (TF) / 86% (PT) of the time, which is overall dominated by constraint solving. Notably, the SMT solving time is long-tailed: the P99 shows that for 1% of test-cases, the solver time deteriorates by over 12.5× (TF) / 15.5× (PT) compared to the average.

Impact of model size. NeuRI and NNSmith by default generate models with 5 nodes. How about other sizes? Figure 6c shows that the model size used in NeuRI impacts the coverage on PyTorch (TensorFlow is omitted for clarity but shares similar trends). For example, only generating one-operator models (*i.e.,* single-API) gets worse coverage, since compiler passes look for multiple-operator patterns (*e.g.,* operator fusion). On the contrary, the benefits converges when model size gets larger and larger (*e.g.,* 0.1% coverage difference in # node 5-13). Consequently, because smaller models are easier to diagnose, we by default use a model size of 5 in NeuRI.

5.3 RQ2: Evaluating Rule Inference

Statistics. Table 3 displays the statistics of APIs, partial operators and records at different phases. The "Collected" row indicates that the developer tests (*i.e.,* the instrumented code) incorporate 758
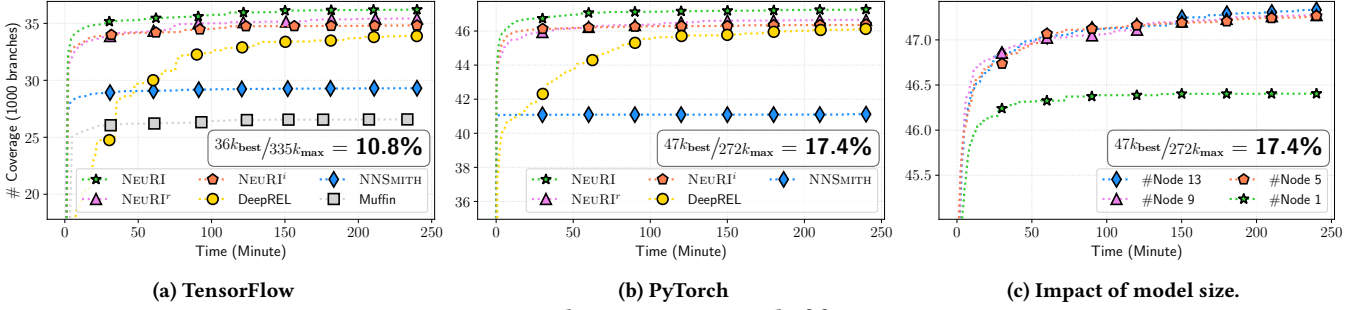
(a) TensorFlow **(b) PyTorch** **(c) Impact of model size.**

Figure 6: 4-hour coverage trend of fuzzing.

Table 1: Number of valid tests generated in 4 hours.

	TensorFlow		PyTorch	
	%Validity	# Tests	%Validity	# Tests
NᴇᴜRI	94.8%	108,572	98.9%	206,486
NᴇᴜRI^i	94.2%	78,083	98.6%	134,912
NᴇᴜRI^r	98.1%	125,059	99.8%	409,872
NNSᴍɪᴛʜ	100%	131,799	100%	639,434
Muffin	95.9%	302	—	—

Table 2: Testing time breakdown (millisecond).

		Gen. (SMT)	Eval.	Save	Total
	Avgerage	67 (33)	21	38	126
TF	P90	74 (37)	29	42	145
	P99	673 (411)	58	59	733
	Percentage	53% (26%)	17%	30%	100%
	Avgerage	60 (38)	9	0	70
PT	P90	45 (26)	11	0	56
	P99	930 (631)	16	1	942
	Percentage	87% (55%)	13%	0%	100%

Table 3: # API/partial operator/record at different stages.

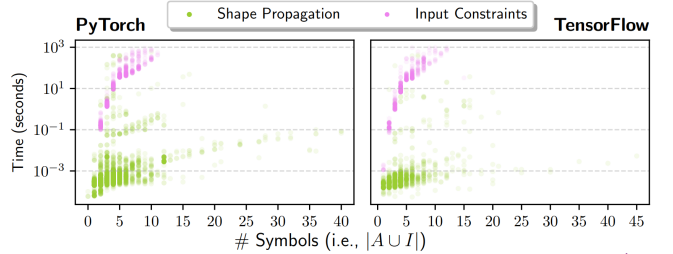|	#API		#Partial Op.		#Record	
	PT	TF	PT	TF	PT	TF
Collected	758	248	—	—	63,136	33,973
Filtering	681	214	5,875	1,799	29,589	12,908
Augment.					1,041,459	303,314
Inference	620	192	4,475	1,507	—	—
Fuzzing$^\top$	604	185	4,186	1,434	—	—
Fuzzing$^\perp$	582	176	4,144	1,415	—	—

Figure 7: Inference time and #symbols of inferred rules[4].

Table 4: Number of inferred shape propagation rules in 1000s.

	Inferred				Timeout	Unsat.
	<1s	<10s	<100s	<1000s		
NᴇᴜRI	4,660	4,700	4,716	4,758	994	123
Rosette	0	83	2,832	4,461	1,414	0

(out of 1310) APIs supported by PyTorch JIT and 248 (out of 450) for XLA respectively. 42-45% APIs are not collected due to the lack of tests (*e.g.,* untested aliased APIs) or being non-tensor APIs (*e.g.,* image encoder and decoder APIs). By focusing on these, around 63k / 34k records can be collected for PyTorch / TensorFlow. After filtering out unwanted records (§4), 47% (PT) / 38% (TF) of the records remained for 90% (PT) / 86% (TF) of the APIs. Furthermore, data augmentation improves the unique records by 15× (PT) / 7.7× (TF), out of which 57% (PT) / 67% (TF) are counter examples.

Within the 1000-second budget per rule, NᴇᴜRI can infer 76% (PT) / 84% (TF) of rules at the partial operator level and 91% (PT) / 90% (TF) of rules at the API level (Table 3). To estimate the usefulness of rules, we use "fuzzing$^\top$" to denote the number for APIs that are used during fuzzing (*i.e.,* NᴇᴜRI^i in one node). To indicate correctness, "fuzzing$^\perp$" denotes those in "fuzzing$^\top$" that always construct valid usages. It turns out 97% (PT) / 90% (TF) APIs out of the inferred ones can be *used* and 96% (both) tend to be used *validly*, showing the overall effectiveness of rule inference.

Scalability. Figure 7 depicts the distribution of inferred rules in terms of the inference time and symbol size. It shows 95% of inferred partial operators have less than 10 (PT) / 11 (TF) symbols, indicating the problem size is small overall. Additionally, inferring shape propagation rules is highly affordable (*i.e.,* 95% of them can be solved within 17ms). However, input constraint inference tends to be more costly, since predicate candidates are thoroughly enumerated in Algorithm 1 while shape propagation terminates when the first feasible solution is found. Meanwhile, the inference time of input constraints grows with # symbols because partial operators with more symbols incur more expressions to validate (Table 5).

Table 4 compares NᴇᴜRI and Rosette in inferring shape propagation for partial operator in PyTorch. It shows that 79% of partial operators are inferred by NᴇᴜRI within one second, while Rosette takes over 10 seconds for 99% of cases. Specifically, the "unsat"

---

[4]The data points of input constraints are fewer because some partial operator have an empty set of input constraints and are not visualized.

Table 5: Sizes of expressions after pruning.

$\|A \cup I\|$	None	Equiv.	Rarity	Both
1	4.77×10^8	5.35×10^6	3.54×10^5	78
2	2.88×10^9	9.04×10^7	5.90×10^6	1.25×10^5
3	1.11×10^{10}	5.18×10^8	5.22×10^7	2.28×10^6
4	3.32×10^{10}	1.91×10^9	3.13×10^8	1.94×10^7
5	8.36×10^{10}	5.49×10^9	1.40×10^9	1.06×10^8
6	1.86×10^{11}	1.33×10^{10}	4.66×10^9	3.89×10^8
Percent.	100%	7.15%	2.51%	0.21%

Table 6: Overview of reported bugs in four months.

	Symptom (§3.4)	Total	Confirmed (Fixed)	Won't Fix
	Inconsistency	19	19 (17)	0
PT	Runtime error	43	32 (25)	2
	Sanitizer error	20	13 (7)	0
	Inconsistency	7	6 (0)	1
TF	Runtime error	7	7 (0)	0
	Sanitizer error	1	1 (0)	0
	Total	100	81 (51)	3

presents the number of cases where no expression is applicable after the *full* enumeration. Rosette has zero "unsat" rules because it fails to finish the enumeration within 1000s.

Impact of pruning. Table 5 shows the number of expressions after being pruned by different methods. For example, for an partial operator with six symbols (*i.e.*, $|A \cup I| = 6$), without any pruning there are hundreds of billions of expressions to verify (*i.e.*, 1.86×10^{11}, recall that we have at most 5 operations). Our pruning method (*i.e.*, the "both") is able to prune the search space smaller by 478×. The ablation study shows that both *equivalence* and *rarity* can sufficiently contribute to shrinking the search space even if applied independently. Interestingly, we found that our method prunes even better pruning ratio for fewer symbols, *e.g.*, $1.64 \times 10^{-5}\%$ *un*pruned for one symbol and 0.21% *un*pruned for six symbols. This is a good news since from Figure 7 we see the number of symbols tend to be small, *i.e.*, approximately clustered within 1 to 10.

Examples. We found most partial operators have very simple rules being inferred, *e.g.*, 70% (PT) / 51% (TF) of which only have expressions with at most one symbol. Nevertheless, we show some complicated representatives to indicate the capability of NᴇᴜRI's rule synthesizer. Given an arbitrary input tensor x, say its shape is $[i_1, i_2, i_3, i_4]$, both x.flatten() and torch.ravel(x) flatten the tensor into a 1-D array whose shape is $[i_1 \times i_2 \times i_3 \times i_4]$, which can be inferred by us. For more complicated cases such as x.unfold(dim, size, step), the shape propagation of $o_{\text{dim}} = 1 + \frac{(i_{\text{dim}} - \text{size})}{\text{step}}$ can also be correctly learnt[5]. There are also cases where partially correct rules are learnt. Consider avg_pool3d(x, ksize=[kT, kH, kW], pad=[pT, pH, pW]) where both the input and output have five dimensions (*i.e.*, $[i_N, i_C, i_T, i_H, i_W] \rightarrow [o_N, o_C, o_T, o_H, o_W]$). While NᴇᴜRI correctly inferred that $o_T = \frac{i_T + 2pT}{kT}$, it overfits the H and W dimension with $o_H = \frac{i_H}{kH} + \min(1, \text{pH})$, due to the lack of records where $\frac{i_H}{kH} + \min(1, \text{pH}) \neq \frac{i_H + 2\text{pH}}{kH}$. However, in our 4-hour fuzzing the corresponding partial operators were used and did not lead to any invalid models. This is because this incorrect expression still gets a good chance of being valid. For example, NNSᴍɪᴛʜ [29] reveals that SMT solvers like Z3 [11] tend to return boundary models, *e.g.*, pH = 0 which makes the two equations equivalent. There are of course uninferred operator rules. For example, one partial operator of torch.stack takes hundreds of input tensors, where NᴇᴜRI cannot handle hundreds of oriented symbols in 1000s.

5.4 RQ3: Bug Finding

Overview. In four months, NᴇᴜRI has found **100** *new* bugs, with 51 fixed and 81 confirmed (Table 6). Links to all bug reports in this

work are included in our artifact[6]. Of these, 76 are found during fuzzing and 24 are byproducts (*e.g.*, crashes by counter examples in argumentation). Among the 85 PyTorch bugs, 8 have been labelled with *high priority*, constituting **10%** (8 / 83) of all high-priority bugs for the entire PyTorch issue tracker in four months. Besides, one PyTorch bug has been tagged with a CVE number when there was only one other CVE published in PyTorch. PyTorch developers say:

> ❝ ...the bugs you've reported are *high quality*, and ... don't look like specially fuzzed set that's impossible to see in practice. They *did* reveal a few common themes that are easy to encounter in practice... ❞

Meanwhile, to report bugs responsibly [42], we discontinued bug reporting to TensorFlow when none of our first 15 reports (14 confirmed) were fixed in a month. Hence, the "15" bugs should be regarded as a lower bound for bug finding efficacy in TensorFlow.

Unique bugs. We illustrate the patterns of exclusive PyTorch bugs (since we discontinued TensorFlow bug finding) found by NᴇᴜRI during fuzzing (*i.e.*, byproducts not included). Among these 62 fuzzing bugs in PyTorch, 39 bugs (62.9%) are only manifested by models with multiple operators – these are not able to be detected by prior single API fuzzers [14, 15, 53]. Meanwhile, 41 (66.1%) of the bugs would not be covered by NNSᴍɪᴛʜ for its limited API supports[7]. For example, torch.reciprocal(torch.dstack([1, 1, 1])) "concatenates" the only input and gets the reciprocal, which should have returned [1, 1, 1]. However, after compilation the third output element becomes non-deterministic. This is confirmed to be a miscompilation bug[8] (now fixed) where the C kernel function generated by PyTorch has erroneous pointer aliases for input and output buffers. This inconsistency bug is neither detectable by single-API testers nor NNSᴍɪᴛʜ (unsupported APIs).

In addition, 17 bugs (27.4%) are exclusive to NᴇᴜRI^i, *i.e.*, the shapes and attributes of the bug-inducing models are not directly obtained from the records, but by solving constraints from inferred rules. It shows that enabling rule inference, though not bringing surprising coverage improvement (§5.2), does help find more bugs. For instance, a *high-priority* bug detected by NᴇᴜRI[9] (now fixed) requires the input shape of torch.histogramdd to be specifically [5, 6] for triggering a compiler failure. The input shape, *i.e.*, [5, 6], comes from the solver-provided model and none of the six collected records of torch.histogramdd can trigger the bug. As another

---

[5]Division demonstrated in this paragraph is floor division, *i.e.*, for integers.

[6]https://github.com/ise-uiuc/neuri-artifact/blob/main/docs/rq3-bug-reports.md
[7]Muffin shares similar limitations as NNSᴍɪᴛʜ in terms of limited API supports, and is not directly comparable here because it only supports TensorFlow.
[8]https://github.com/pytorch/pytorch/issues/93078
[9]https://github.com/pytorch/pytorch/issues/93274

example, the *-unfold-abs_ model pattern ("*" means any operators and the "_" in abs_ means it is an in-place operation) can manifest a result inconsistency bug[10] (now fixed) since the graph functionalization in the PyTorch compiler was not able to identify certain operator patterns that have memory overlapping. Specifically, it is detected by using a set of solver-provided arguments, *i.e.*, tensor.unfold(1, 3, 2). Notably, both examples here require operators that are unavailable in NNSᴍɪᴛʜ, showing that operator diversity further powers model diversity to detect more bugs.

Furthermore, one heap-overflow bug is assigned by PyTorch with a CVE identification number (GHSA-6655-44g2-4gc8) due to its security impact. This bug is induced by searchsorted(arr, val, sorted_idx), which aims to binary search val in arr (*e.g.*, an array) where sorted_idx are the sorted indices of val. Specifically, boundary checks for sorted_idx were absent in the previous implementation. Therefore, a large enough index, if "lucky", can lead to a segmentation fault, terminating the program without further impact. However, a carefully designed index allows attackers to access and steal data from other memory addresses besides the array range when performing the binary search.

"Won't-fix" bugs. Three of our reports are rejected or deprioritized. For example, both an inconsistency bug in tf.cast [46] and a crash bug in torch.linalg.eigvals [40] were rejected for using NaNs as input, incurring undefined behaviours. Another PyTorch JIT bug was deprioritized because developers suggested us to use and test the new compiler [41] (and consequently we did).

6 RELATED WORK

In recent years, fuzzing [33] has been extensively studied for testing DL libraries and DL compilers, which can be mainly categorized into operator and model levels. Operator-level techniques [14, 15, 53] aim to test each DL API in isolation. FreeFuzz [14], a fully automated operator-level fuzzer for testing DL libraries (such as TensorFlow and PyTorch), collects DL API traces from sources such as developer tests and model zoo, and further mutates the traced inputs to generate additional valid/invalid test-cases for fuzzing each operator. Similarly, DocTer [53] also aims to test each DL operator individually, by extracting their input constraints from documentation, incurring manual inspection of the mined rules for 30% of the API arguments. Nonetheless, the input constraints for each argument are defined by DocTer as a potential set of types and values, which cannot model fine-grained shape constraints. Additionally, recent work has also been improving the oracles of DL system testing via API relation [15] and gradient checking [54]. Though effective in bug finding, operator-level fuzzing hardly uncover bugs induced by multiple operators together, *e.g.*, bugs in DL compilers.

Model-level fuzzing techniques generate DL models with multiple operators. The pioneer Cradle [39] directly runs pre-built DL models programmed in Keras [18] and cross-check results from various backends. Built on Cradle, LEMON [51] and Audee [20] generate models via pre-defined mutation rules. Muffin [19] performs layer-by-layer model generation for testing both training and inference. Recently, NNSᴍɪᴛʜ [29] annotates each operator with input constraints and shape transformation, and generates well-formed models (following Csmith [55]) aided by constraint

solving. Furthermore, GenCoG [50] simplifies such annotations via a domain-specific language which still needs manual annotation. While they complement operator-level fuzzing, the model mutation/generation rules are restrictive, *e.g.*, they typically only target naive shape-preserving operators [19, 32, 51], or require certain manual annotations [29], leading to a limited set of operators being used. This work proposes to infer such operator rules, and then apply them to generate valid models with all possible operators. Our work can cover as many operators as operator-level fuzzing while creating valid models with covered operators *fully automatically*, *i.e.*, a step forward for bridging the gap between operator- and model-level fuzzing for DL systems.

More recently, there has been concurrent work [12, 13] on directly leveraging large language models (LLMs) to synthesize Python programs to construct valid DL models. Such techniques aim to *implicitly* solve the validity constraints via directly learning from valid samples. Compared with LLM-based model generation, despite the technical complexity, by *explicitly* solving the constraints, NᴇᴜRI provides stronger validity guarantee within the covered model space and is more affordable (*i.e.*, <100ms per model on CPU). Meanwhile, LLMs, trained over billions of lines of codes, can be used to easily test beyond the model space carefully crafted and exhaustively explored by NᴇᴜRI. Therefore, these two approaches can be further combined for maximized fuzzing in the future.

Lastly, program synthesis has been used by related areas such as synthesizing user-facing tensor-manipulation programs [35, 45, 56]. Similar to NNSᴍɪᴛʜ, they also require manual operator specifications. This paper applies inductive program synthesis [28, 52] to infer such specifications, and can potentially improve all methods targeting model generation, *e.g.*, DL system fuzzing, tensor-manipulation program synthesis and neural architecture search [16].

7 CONCLUSION

We present NᴇᴜRI, a novel approach to automatically infer operator rules for diversifying model generation in order to test DL systems. NᴇᴜRI generates test-cases by creating structurally valid models using a diverse set of operators for exercising deep system behaviours. The primary source of the diversity comes from our automated rule inference and concolic model generation. The rule inference engine inductively and efficiently discovers operator rules for generating valid models symbolically. Meanwhile, our concolic model generator further uses concrete operator invocations in combination with the symbolic operators to maximize the model diversity. As a result, NᴇᴜRI finds many *high-priority* and *-quality* bugs appreciated by DL-framework developers. We show NᴇᴜRI is promising for long-term fuzzing – high-quality test-cases can be generated in milliseconds on a single thread and new operators can be automatically integrated. To date, NᴇᴜRI has already detected 100 *new* bugs for PyTorch and TensorFlow, with 81 fixed or confirmed.

DATA AVAILABILITY

Our artifact is available at Zenodo [8] and GitHub [30].

ACKNOWLEDGMENTS

---

[10]https://github.com/pytorch/pytorch/issues/98143

REFERENCES

[1] 2022. *ASPLOS2023 Artifact for "NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers"*. Zenodo. https://doi.org/10.5281/zenodo.7222132

[2] 2022. Building the Future of TensorFlow — The TensorFlow Blog. https://blog.tensorflow.org/2022/10/building-the-future-of-tensorflow.html.

[3] 2022. "Compute Sanitizer :: CUDA Toolkit Documentation". https://docs.nvidia.com/cuda/compute-sanitizer/index.html.

[4] 2022. GCOV. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[5] 2022. PyTorch 2.0 | PyTorch. https://pytorch.org/get-started/pytorch-2.0/.

[6] 2022. Source-based Code Coverage — Clang 15.0.0 documentation. https://releases.llvm.org/15.0.0/tools/clang/docs/SourceBasedCodeCoverage.html.

[7] 2022. "UndefinedBehaviorSanitizer — Clang 16.0.0git documentation". https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[8] 2023. *ESEC/FSE'23 Artifact for "NeuRI: Diversifying DNN Generation via Inductive Rule Inference"*. Zenodo. https://doi.org/10.5281/zenodo.8319975

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

[10] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 319–336.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[12] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 423–435. https://doi.org/10.1145/3597926.3598067

[13] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. *arXiv preprint arXiv:2304.02014* (2023).

[14] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing Deep-Learning Libraries via Automated Relational API Inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 44–56. https://doi.org/10.1145/3540250.3549085

[15] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational api inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 44–56.

[16] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *The Journal of Machine Learning Research* 20, 1 (2019), 1997–2017.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

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

[19] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. 2022. Muffin: Testing Deep Learning Libraries via Neural Architecture Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1418–1430. https://doi.org/10.1145/3510003.3510092

[20] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2021. Audee: Automated Testing for Deep Learning Frameworks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 486–498. https://doi.org/10.1145/3324884.3416571

[21] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[23] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[24] Susmit Jha and Sanjit A. Seshia. 2017. A Theory of Formal Synthesis via Inductive Learning. *Acta Inf.* 54, 7 (nov 2017), 693–726. https://doi.org/10.1007/s00236-017-0294-5

[25] Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. 2022. A Static Analyzer for Detecting Tensor Shape Errors in Deep Neural Network Training Code. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 337–338. https://doi.org/10.1145/3510454.3528638

[26] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[28] Tessa A. Lau and Daniel S. Weld. 1999. Programming by Demonstration: An Inductive Learning Formulation. In *Proceedings of the 4th International Conference on Intelligent User Interfaces* (Los Angeles, California, USA) *(IUI '99)*. ACM, New York, NY, USA, 145–152. https://doi.org/10.1145/291080.291104

[29] 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. https://doi.org/10.1145/3575693.3575707

[30] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. 2023. Artifact for ESEC/FSE'23 paper "NeuRI: Diversifying DNN Generation via Inductive Rule Inference". https://github.com/ise-uiuc/neuri-artifact.

[31] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 73 (apr 2022), 26 pages. https://doi.org/10.1145/3527317

[32] Weisi Luo, Dong Chai, Xiaoyue Run, Jiang Wang, Chunrong Fang, and Zhenyu Chen. 2021. Graph-Based Fuzz Testing for Deep Learning Inference Engines. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 288–299. https://doi.org/10.1109/ICSE43902.2021.00037

[33] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[34] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

[35] Ansong Ni, Daniel Ramos, Aidan Z. H. Yang, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2021. SOAR: A Synthesis Approach for Data Science API Refactoring. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 112–124. https://doi.org/10.1109/ICSE43902.2021.00023

[36] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/.

[37] OpenAI. 2022. DALL·E 2. https://openai.com/dall-e-2/.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[39] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[40] PyTorch. 2022. Numerical accuracy — PyTorch master documentation. https://pytorch.org/docs/master/notes/numerical_accuracy.html#linear-algebra-torch-linalg.

[41] PyTorch. 2022. TorchDynamo and TorchInductor Tutorial. https://pytorch.org/tutorials/intermediate/dynamo_tutorial.html.

[42] John Regehr. 2020. Responsible and Effective Bugfinding. https://blog.regehr.org/archives/2037.

[43] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. Association for Computing Machinery, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[44] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 309–318.

[45] Kensen Shi, David Bieber, and Rishabh Singh. 2022. TF-Coder: Program Synthesis for Tensor Manipulations. *ACM Trans. Program. Lang. Syst.* 44, 2, Article 10 (may 2022), 36 pages. https://doi.org/10.1145/3517034

[46] TensorFlow. 2022. tf.cast | TensorFlow v2.11.0. https://www.tensorflow.org/api_docs/python/tf/cast.

[47] TensorFlow Contributors. 2023. Using TensorFlow Securely – TensorFlow models are programs. https://github.com/tensorflow/tensorflow/security/policy#tensorflow-models-are-programs [Online; accessed 3-Jan-2023].

[48] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (Indianapolis, Indiana, USA). Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[49] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. *SIGPLAN Not.* 48, 6 (jun 2013), 287–296. https://doi.org/10.1145/2499370.2462174

[50] Zihan Wang, Pengbo Nie, Xinyuan Miao, Yuting Chen, Chengcheng Wan, Lei Bu, and Jianjun Zhao. 2023. GenCoG: A DSL-Based Approach to Generating Computation Graphs for TVM Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 904–916. https://doi.org/10.1145/3597926.3598105

[51] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 788–799. https://doi.org/10.1145/3368089.3409761

[52] Patrick H. Winston. 1970. *Learning Structural Descriptions From Examples*. Technical Report. Cambridge, MA, USA.

[53] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 176–188. https://doi.org/10.1145/3533767.3534220

[54] Chenyuan Yang, Yinlin Deng, Jiayi Yao, Yuxing Tu, Hanchi Li, and Lingming Zhang. 2023. Fuzzing Automatic Differentiation in Deep-Learning Libraries. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1174–1186. https://doi.org/10.1109/ICSE48619.2023.00105

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[56] Zhanhui Zhou, Man To Tang, Qiping Pan, Shangyin Tan, Xinyu Wang, and Tianyi Zhang. 2022. INTENT: Interactive Tensor Transformation Synthesis. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 89, 16 pages. https://doi.org/10.1145/3526113.3545653

[57] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (dec 1997), 366–427. https://doi.org/10.1145/267580.267590