Towards More Complete Constraints for Deep Learning Library Testing via Complementary Set Guided Refinement

Gwihwan Go
Tsinghua University
Beijing, China
iejw1914@gmail.com

Chijin Zhou*
Tsinghua University
Beijing, China
tlock.chijin@gmail.com

Quan Zhang*
Tsinghua University
Beijing, China
zhangq20@mails.tsinghua.edu.cn

Xiazijian Zou
Central South University
Changsha, China
zouxia19@gmail.com

Heyuan Shi
Central South University
Changsha, China
hey.shi@foxmail.com

Yu Jiang
Tsinghua University
Beijing, China
jiangyu198964@126.com

Abstract

Deep learning library is important in AI systems. Recently, many works have been proposed to ensure its reliability. They often model inputs of tensor operations as constraints to guide the generation of test cases. However, these constraints may narrow the search space, resulting in incomplete testing. This paper introduces a complementary set-guided refinement that can enhance the completeness of constraints. The basic idea is to see if the complementary set of constraints yields valid test cases. If so, the original constraint is incomplete and needs refinement. Based on this idea, we design an automatic constraint refinement tool, DeepConstr, which adopts a genetic algorithm to refine constraints for better completeness. We evaluated it on two DL libraries, PyTorch and TensorFlow. DeepConstr discovered 84 unknown bugs, out of which 72 confirmed, with 51 fixed. Compared to state-of-the-art fuzzers, DeepConstr increased coverage for 43.44% of operators supported by NNSmith, and 59.16% of operators supported by NeuRI.

CCS Concepts

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

Keywords

Large Language Model, Fuzzing, DL library

ACM Reference Format:
Gwihwan Go, Chijin Zhou, Quan Zhang, Xiazijian Zou, Heyuan Shi, and Yu Jiang. 2024. Towards More Complete Constraints for Deep Learning Library Testing via Complementary Set Guided Refinement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680364

*Quan Zhang and Chijin Zhou are corresponding authors.

1 Introduction

In a world reshaped by artificial intelligence, Deep Learning (DL) libraries such as PyTorch [30] and TensorFlow [4] play an essential role in the development and deployment of machine learning models. Therefore, the presence of bugs in these DL libraries can have far-reaching implications. For example, a bug in a DL library can cause incorrect decisions in a self-driving system, potentially causing serious personal injury and substantial economic damage. Consequently, there is an urgent and growing need to rigorously test DL libraries to ensure their reliability.

To generate effective test cases, existing research efforts [10, 24, 25, 38, 42, 46] focus on inferring the input constraints for DL operators and use the constraints to guide the generation of test cases. These constraints describe the requirements that the input arguments of an operator must satisfy to bypass the operator's input validity checks. For example, the `MaxPool2d` operator accepts input tensor and operation configurations (*e.g.,* kernel_size, ...) as arguments to calculate the output tensor, i.e.,

$$\text{MaxPool2d}(\text{input}, (\text{kernel\_size}, \text{padding}, \text{stride}, \ldots)). \quad (1)$$

These arguments are tightly constrained: the kernel size should depend on the input shape and padding size, the stride should be greater than zero, and the padding should not be less than zero. As such, the constraint for this operator can be represented as

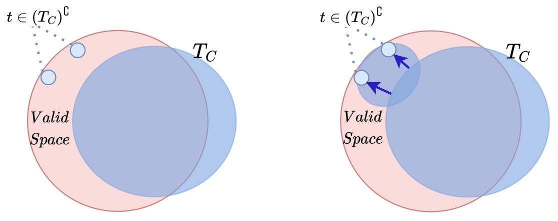
$$\begin{aligned}&(\text{kernel\_size} \leq \text{input\_size} + 2 \times \text{padding}) \wedge \\ &(\text{stride} > 0) \wedge (\text{padding} \geq 0).\end{aligned} \quad (2)$$

If a test case fails to meet this constraint, the operator will abort its operation, thereby preventing effective testing. This is why many techniques [10, 25, 38, 42, 46] have been proposed to automatically infer such constraints for operators. Their focus is on pursuing the *soundness* of the constraints. We call a constraint is *sound* only when every derived test case from the constraint is valid for the target operator.

Constraint Completeness. In contrast to soundness, which can be directly evaluated through the responses of programs to test cases, ensuring the *completeness* of constraints is difficult. Let us consider a program that accepts an integer input within the range $(-\infty, 10]$ and a fuzzer configured with an input constraint of $(-\infty, 0)$. Although all the test cases generated by this fuzzer are valid, they will never cover the range $[0, 10]$. In this case, the fuzzer's constraint is sound but **incomplete**, meaning that it is overly strict and fails to generate test cases that cover the entire input space. Therefore, a constraint is considered complete if it can yield all possible valid test cases for a given operator. DL libraries, in contrast to simple programs, have more complex constraints that are difficult to infer. Prior research [24, 25, 38, 46] has pioneered the use of constraints-guided generation and reports success in testing efficiency. However, although they have made significant progress in ensuring the validity of test cases, they cannot ensure the completeness of their constraints. The difficulty of ensuring completeness arises from the need for an exhaustive understanding of the valid input space. Without this, it becomes impossible to identify areas of the input space that a constraint might have failed to cover. For instance, previous research [25] infers the constraint of the `MaxPool2d` as

$$(\text{kernel\_size} < \text{input\_size}) \wedge (\text{stride} > 0) \wedge (\text{padding} \geq 0). \quad (3)$$

All test cases generated based on the constraint (3) are indeed valid for the `MaxPool2d` operator, as they also satisfy the constraint (2). However, the constraint (3) fails to model the interrelation of `kernel_size` between `padding`. As a result, they overlook valid test cases where $\text{input\_size} \leq \text{kernel\_size} \leq \text{input\_size} + 2 \times \text{padding}$. Without manually inspecting the operator, it is challenging to identify such overlooked areas of the input space. As a result, a large portion of valid input space is unconsciously ignored by existing techniques, leading to incomplete testing for DL libraries.



(a) Identify counter test cases. **(b) Refine the constraint.**

Figure 1: An illustrated example of our method. The red area represents the valid input space of an operator, and the blue area represents the search space of the constraint.

Method. This paper introduces *Complementary Set Guided Refinement*, an approach to enhance the completeness of constraints for DL libraries. Figure 1 illustrates its basic idea. Suppose we have a constraint C for a DL operator, and T_C is the set of test cases generated based on C. The basic idea behind our approach is to identify the complementary set of $T_C^\complement$, and observe if there is a *counter test case* $t \in T_C^\complement$ that is valid for the operator. If such a test case exists, it indicates that the constraint C is incomplete, thereby pointing towards a potential direction for refining the constraint. By gradually refining the constraint with $t \in T_C^\complement$, we can achieve a more complete constraint whose yielded test cases can cover a wider range of the valid input space.

Challenge in Refinement. While the counter test case provides a direction for refining the constraint of an operator, the refinement process remains a significant challenge. The constraint of an operator often contains multiple sub-constraints. Consequently, once we identify a counter test case, it becomes difficult to pinpoint which sub-constraint is responsible for the incompleteness. Consider the constraint (3) as an example, which includes three sub-constraints. During the refinement process, we cannot directly determine which sub-constraint is incomplete and needs refining. A counter test case could potentially violate any of these sub-constraints, and thus, we need to consider all of them as potential candidates to be refined. However, in the case of constraint (3), only the first sub-constraint, i.e., $\text{kernel\_size} < \text{input\_size}$, is incomplete and needs refining, while the other two are complete. Therefore, a mechanism is required to identify the specific sub-constraint responsible for the incompleteness of a given counter test case.

Solution. To address this challenge, we design DeepConstr, a tool that leverages error messages to automatically refine the constraints of DL operators. Our key insight is that each sub-constraint is coupled with a distinct error message. This allows us to divide the input constraint of a given operator into several smaller, independent sub-constraints based on these error messages. Therefore, we can focus solely on refining the specific sub-constraint that corresponds to a particular error message at a time, thereby avoiding the potential interference from other sub-constraints. The refinement of each sub-constraint employs a genetic algorithm. Specifically, we maintain a set of high-quality constraints and generate new ones in each iteration. We then assess the soundness and completeness of them based on the test cases derived from them, and prune the set by removing less-quality constraints. This iterative process ensures the constraint is gradually refined towards a more complete state.

Evaluation. We evaluated DeepConstr's performance on the two mainstream DL libraries, namely, PyTorch and TensorFlow. DeepConstr discovered 84 previously unknown bugs, out of which 72 were confirmed, and 51 were fixed. Most of the bugs reside in operators that were heavily tested by existing fuzzers [24, 25, 46]. Compared to the state-of-the-art fuzzers, DeepConstr has demonstrated significant enhancements in terms of branch coverage for every single operator. Specifically, it achieves greater coverage on 58.27% of operators supported by NeuRI, and 49.15% of operators supported by NNSmith in PyTorch and 62.1% of operators supported by NeuRI, and 38.1% of operators supported by NNSmith in TensorFlow. In summary, this paper makes the following contributions:

- We identify the problem of current constraint-guided testing in DL library testing: insufficient consideration of constraint completeness. To address this, we propose a complementary set based refinement approach to resolve the problem by enhancing the completeness of constraints.
- We design DeepConstr, a practical tool that generates more complete constraints for DL operators. This tool is available at https://github.com/THU-WingTecher/DeepConstr
- We evaluate DeepConstr on PyTorch and TensorFlow. In total, it uncovers 84 previously-unknown bugs with 72 confirmed and 11 with received high-priority, out of which 51 has been fixed. DeepConstr achieves greater coverage of 58.27% of operators supported by NeuRI and 49.15% of operators supported by NNSmith in PyTorch, and 62.1% of operators supported by NeuRI and 38.1% of operators supported by NNSmith in TensorFlow.

2 Background and motivation

2.1 DL Library and Constraint

DL libraries are crucial components in most of DL applications. DL applications usually employ complicated and costable numeric calculations such as convolutional operations and matrix multiplications. These operations are highly influenced by the way they are implemented. DL libraries incorporate DL operators that implement these complex operations in the most optimized manner [4, 30, 37], saving users' efforts in implementing these operations. However, as a trade-off for the optimized implementation, DL libraries restrict the range of accepted inputs, which makes it difficult for fuzzers to generate valid test cases for DL libraries. Existing research [25, 38] has shown that modeling constraints of DL libraries is effective for generating valid test cases.

The constraints of DL libraries indicate the specific requirements for generating valid test cases. If these requirements are not fulfilled, the DL operator will not initiate its operation. Suppose that $S^k = [s_0^k, s_1^k, \ldots, s_{r_k-1}^k]$ describes the shape of the tensor I^k, in which s_i corresponds the i-th dimension value of I^k with the rank of r_k. There are three kinds of constraints that are usually seen in DL libraries. First, a constraint that requests a single input tensor I^k to satisfy, such as $\forall s_i^k, s_j^k \in S^k, i \neq j \implies s_i^k \neq s_j^k$. Second, a constraint that specifies an input tensor I^k and the operator configuration $A = [a_1, a_2, \ldots, a_m]$ to be satisfied. For example, many dimension expansion operators requires that $\forall a_i \in A, s_{a_i}^k = 1$, where A includes the dimensions referred to be expended. Third, a constraint that multiple input tensors I^k, I^j should satisfy. For example, in matrix multiplication operators, the last dimension of one input tensor I^k should be the same as the first dimension of the other input tensor I^j, which means $s_{r_k-1}^k = s_0^j$

This complexity arises from the high-dimensional computations performed, as opposed to those in conventional software. In consequence, randomly generating inputs will seldom pass the internal checks of DL operators [25, 38, 46]. It emphasizes the need for a constraint-guided approach to generate valid test cases.

2.2 Motivation

Overly-strict constraints often lead to insufficient diversity in generated test cases, making fuzzers impossible to completely explore the semantics state of target program. Previous studies [24, 25, 38, 46] have deduced constraints of DL libraries and generated test cases derived from these constraints. However, they cannot ensure the completeness of the constraints they've inferred. This may cause them to miss potential vulnerabilities of DL libraries. Listing 1 is an inconsistency bug that DeepConstr has found on PyTorch[1]. The developers regarded this bug as important and assigned it a high-priority tag, which led it to be fixed very quickly. An operator `torch.diag_embed` is used to embed a tensor into a diagonal matrix. This operator can be invoked with 4 arguments, input tensor `x`, and three integers `dim1`, `dim2`, and `offset`. The bug is triggered when `torch.diag_embed(x, dim1, dim2, offset)` is invoked with negative value assigned to the argument `dim1`, while the positive value passed to the argument `dim2`. To find out this bug, a test

[1]https://github.com/pytorch/pytorch/issues/117019

```python
import torch
class Model(torch.nn.Module):
    def forward(self, x):
        return torch.diag_embed(x, dim1=-1,dim2=1,offset=1)
model = Model()
x = torch.rand([2, 2, 2]) ## Randomly generate tensor
eag = model(x)
opt = torch.compile(model.forward)(x)
torch.allclose(eag, opt) # Different results => bug
```

Listing 1: Minimized code snippet of inconsistency bug.

case should be valid first, which needs to satisfy a constraint, which includes a series of sub-constraints. First, the rank of input tensor `x` should be at least 1. Second, the diagonal dimensions cannot be identical, which means `dim1` should not be equal to `dim2`. Third, the `dim1` and `dim2` should be within the range of $[-\text{x.rank}, \text{x.rank})$. As such, the actual constraint for `torch.diag_embed` would be:

$$(\text{dim1} \neq \text{dim2}) \wedge (-\text{x.rank} \leq \text{dim1}, \text{dim2} < \text{x.rank}) \wedge (\text{x.rank} \geq 1)$$

However, without considering completeness, prior research [25] extracts the the constraints of `torch.diag_embed` as below:

$$\text{dim1} \in \{-2, 0, 1\} \wedge (\text{dim2} = \text{dim1} + 1) \wedge (\text{x.rank} \geq 2)$$

Although the test cases derived from the above constraint are valid, the constraint is actually interrupting the testing. Specifically, within the constraints provided, the feasible candidates for the values of `dim1` and `dim2` are limited: (1). $dim1 = -2, dim2 = -1$, (2). $dim1 = 0, dim2 = 1$, and (3). $dim1 = 1, dim2 = 2$. This limits the fuzzer from finding bugs triggered by a negative value in `dim1` with a positive value in `dim2`.

Generating complete constraints is challenging. Unlike soundness, which can be directly assessed through programs' responses to test cases, completeness is difficult to measure because it is impossible to identify the input space that a constraint may have missed without understanding the program's internal logic. A previous work [38] has noted the importance of completeness of constraint they have inferred. In the absence of alternative methods, they evaluate the completeness of the constraints by manually analyzing the source code of the DL operators. However, this approach is labor-intensive and can only be performed on a small subset of operators, making it unsuitable for an automatic fuzzer. This highlights a critical challenge in constraint refinement: *How can we automatically measure the completeness of constraints and refine them to be more complete?*

3 Complementary Set Guided Refinement

To address the above challenge, we propose a method called *Complementary Set Guided Refinement*. Our method is twofold. First, we divide the complex constraint of an operator into several independent and relatively simple constraints using error messages. Next, we leverage the complementary set of each independent constraint to identify the incomplete areas of this constraint and thus refine it to be more complete.

Formally, let S be a DL library, which includes a set of DL operators $\{O_1, O_2, \ldots\} \in S$. In a system S, a DL operator O contains a set of error messages $E = \{e_1, e_2, \ldots, e_n\}$ that consists of n messages. Once a test case violates the constraint of O, it will trigger an error message e_i. Therefore, we can divide the constraint of O into a set of independent constraints, denoted as *constraint set* $C = \{c_1, c_2, \ldots, c_n\}$, each of which is designed to resolve a specific error message e_i. Based on this, we can independently refine each constraint c_i for better completeness, and subsequently combine them to form a complete constraint for operator O.

To refine a constraint c_i, we use **complementary set** to identify its incomplete areas, i.e., measuring its completeness. Suppose T_{c_i} is a set consisting of all the test cases that are generated under the guidance of c_i. Obviously, T_{c_i} is a subset of the universal input space U for operator O, satisfying that $\forall t \in U$, if t satisfies c_i, then $t \in T_{c_i}$. Let us consider T_{c_i}'s complementary set:

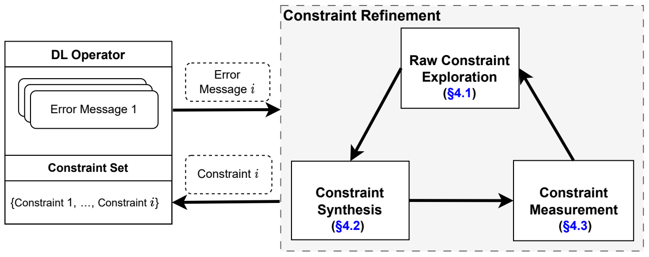
$$T_{c_i}^{\complement} = U - T_{c_i}.$$

This set enables us to identify test cases that can resolve the target error e_i but outside T_{c_i}. Specifically, we can generate a counter test case $t \in T_{c_i}^{\complement}$ that violates c_i and inspect whether it resolves error e_i or not. If O does not return error e_i with $t \in T_{c_i}^{\complement}$, it indicates c_i is not complete, because there exists a counter test case t not satisfies c_i but resolves e_i. As such, we can measure the completeness of c_i by traversing $T_{c_i}^{\complement}$ and minimizing the counter test cases, thereby gradually improving the constraint c_i to make it more complete.

A counter test case only provides information about the incompleteness of a constraint, but how to refine the constraint is still an unsolved problem. To address this, we design a systematic constraint refinement approach. It iteratively refines each constraint by measuring its completeness based on the complementary set. As a result, by refining every constraint c_i in C, we can form a more complete constraint for operator O. We will provide further details in the next section.

4 Approach

In this section, we detail the design of DeepConstr, an automated constraint refinement tool based on the idea of complementary set guided refinement. Our final goal is to generate a constraint set $C = \{c_1, c_2, c_3, \ldots\}$ for operator O. We adopt a divide and conquer approach to generate a constraint set C, wherein we generate and refine each constraint c_i based on the corresponding error message e_i. As illustrated in Figure 2, our approach for refining each c_i follows an iterative approach. Inspired by genetic algorithm [19], we maintain a high-quality constraint pool during the whole iteration, from which we select the final constraint c_i. Our design for refining c_i follows below steps : **(1)**, a raw constraint exploration module that identifies a raw constraint from the given error message; and **(2)**, a constraint synthesis module that enhances the quality of constraint through synthesis with the constraint pool; and **(3)** a constraint measurement module that assesses the constraint and updates the constraint pool; and go back to **(1)** to continue refining the constraint pool, or break the iteration if a complete constraint has been found. After the constraint set C of the operator O is constructed, it will be passed into the fuzzer to effectively generate test cases.



Constraint Refinement
DL Operator
Error Message 1
Error Message i
Constraint Set
{Constraint 1, ..., Constraint i}
Constraint i
Raw Constraint Exploration (§4.1)
Constraint Synthesis (§4.2)
Constraint Measurement (§4.3)

Figure 2: Overview of a constraint refinement process.

4.1 Raw Constraint Exploration

This module aims to generate a raw constraint that can model potential constraints inherent in the DL operator based on an error message. In this section, we first present the grammar designed to effectively model the constraint of the DL operator, and then describe how this constraint is generated from the error message.

Grammar Definition. To model complex constraints inherent in the DL operator, a comprehensive grammar definition is needed. Figure 3 illustrates the grammar definition that DeepConstr supports. The grammar is designed to model all potential input constraints. Therefore, it includes common operations that can be performed on the inputs, such as obtaining inputs' ranks (<rank>) and finding the maximal values (<max>). For example, when the constraint that specified the maximum value of given dimension values `dims` should fall in the range of input tensor `x`, this constraint can be modeled as $-x.rank \leq max(dims) < x.rank$. In addition, to effectively model complex constraints such as broadcasting, the grammar incorporates quantifier expressions (<quan_op>) such as <for_all> or <for_any>. This is necessary because constraints such as broadcasting require that every dimensional value of the tensor size satisfies certain conditions (assuming the two tensors have the same rank).

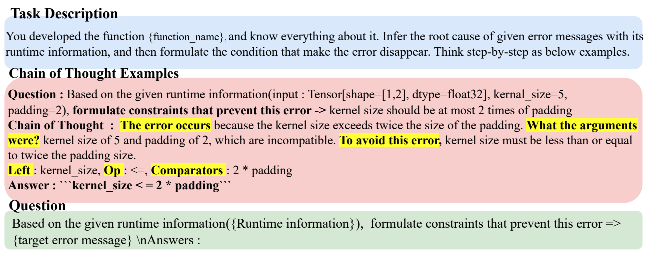
$$\langle expr\rangle ::= \langle expr\rangle\langle bool\_op\rangle\langle expr\rangle \mid \langle var\rangle\langle comp\_op\rangle\langle var\rangle \mid \langle var\rangle\langle com\_op\rangle\langle const\rangle \mid \langle quan\_op\rangle(\langle expr\rangle \implies \langle expr\rangle)$$
$$\langle bool\_op\rangle ::= \langle and\rangle \mid \langle or\rangle \mid \langle not\rangle$$
$$\langle quan\_op\rangle ::= \langle for\_all\rangle \mid \langle for\_any\rangle$$
$$\langle func\rangle ::= \langle min\rangle \mid \langle max\rangle \mid \langle abs\rangle \mid \langle rank\rangle \mid \langle shape\rangle \mid \langle set\rangle \mid \langle dtype\rangle \mid \langle sorted\rangle \mid \langle len\rangle \mid \langle dim\rangle \mid \langle size\rangle \mid \langle slice\rangle$$
$$\langle var\rangle ::= \langle const\rangle\langle bin\_op\rangle\langle var\rangle \mid \langle func\rangle(\langle var\rangle) \mid \langle var\rangle\langle bin\_op\rangle\langle var\rangle$$
$$\langle const\rangle ::= \langle integer\rangle \mid \langle float\rangle \mid \langle str\rangle \mid \langle bool\rangle \mid \langle complex\rangle \mid \langle tensor\_dtype\rangle$$
$$\langle comp\_op\rangle ::= \; < \mid \leq \mid > \mid \geq \mid = \mid \neq \mid \text{is} \mid \text{is not} \mid \text{in} \mid \text{not in}$$
$$\langle bin\_op\rangle ::= \; + \mid - \mid \times \mid \% \mid \div \mid //$$

Figure 3: The grammar definition of constraint.

Error Message. DeepConstr generates a raw constraint from the description of the error message. This error message comes from invalid test cases. For example, assume that we execute the `MaxPool2d` operator with the values that violate the constraint, $k_i <= I_i + 2 \times pad$. In this case, `MaxPool2d` will innerly check the given values first and return the error message, stating `"kernel size should be at most two times of padding"`. We save this error message with the test case that triggers it and use it to generate a raw constraint.

Distillate Semantic Information. We utilize LLMs to infer a raw constraint from an error message that would not re-trigger the given error. Specifically, we leveraged the few-shot Chain of Thought (CoT) prompting technique [43] as illustrated in Figure 4.

Few-shot CoT prompting involves presenting LLMs with a small set of examples that demonstrate a step-by-step approach to problem-solving, guiding the model to follow a logical sequence of thoughts. We built a set of examples that step-by-step infer constraints from error messages. It helps LLMs to identify the root cause, retrieve the runtime information, and subsequently conclude the requirements of constraints. By following the guidance, LLMs would output a constraint, which we call a raw constraint, because it is usually incorrect. DeepConstr takes this raw constraint and moves to the next step. In subsequent steps, this raw constraint will be refined through iterative synthesis and measurement.



Task Description

You developed the function {function_name}, and know everything about it. Infer the root cause of given error messages with its runtime information, and then formulate the condition that make the error disappear. Think step-by-step as below examples.

Chain of Thought Examples

Question : Based on the given runtime information(input : Tensor[shape=[1,2], dtype=float32], kernal_size=5, padding=2), **formulate constraints that prevent this error ->** kernel size should be at most 2 times of padding
Chain of Thought : **The error occurs** because the kernel size exceeds twice the size of the padding. **What the arguments were?** kernel size of 5 and padding of 2, which are incompatible. **To avoid this error,** kernel size must be less than or equal to twice the padding size.
Left : kernel_size, **Op** : <=, **Comparators** : 2 * padding
Answer : ```kernel_size < = 2 * padding```

Question

Based on the given runtime information({Runtime information}), formulate constraints that prevent this error => {target error message} \nAnswers :

Figure 4: The prompt template used by DeepConstr.

4.2 Constraint Synthesis

The constraint synthesis module aims to enhance the quality of constraints. This module is crucial because the raw constraint generated by LLMs may be neither sound nor complete. The synthesis is conducted by combining the newly generated raw constraint with high-quality constraints extracted from the constraint pool that DeepConstr maintains.

Algorithm 1: Constraint Synthesis

```
   Input  : Newly inferred raw constraint $raw$,
            High quality constraints $ConstraintPool$,
            Sound constraint $SoundConstr$,
            Complete constraint $CompleteConstr$
   Output : A synthesized constraint set $EvalSet$
1  Function NeedSynthesis($p, q$):
       // if either $p$ or $q$ is the super-set of the other, then return false
2      return not ($p \supseteq q$ or $q \supseteq p$)
3  $EvalSet \leftarrow \{raw\}$
4  for $constr$ in $ConstraintPool$ do
5      if NeedSynthesis($sound, raw$) then
6          $raw \leftarrow raw \lor sound$
7      end
8      if NeedSynthesis($complete, raw$) then
9          $raw \leftarrow raw \land complete$
10     end
11     if NeedSynthesis($raw, constr$) then
12         $EvalSet \cup \{constr \lor raw, constr \land raw\}$
13     end
14 end
15 return $EvalSet$
```

The synthesis of two constraints is conducted by connecting two different constraints with boolean operators, OR($\lor$) and AND($\land$). As described in Algorithm 1, the synthesis consists of two steps, targeted synthesis (lines 5-10) and naive synthesis (lines 11-13). Before proceeding with any synthesis, DeepConstr first inspects the relationship between two constraints (lines 1-3). Specifically, if one constraint is a super-set of the other, the synthesis process is skipped. This is because synthesizing them with any boolean operator would yield a meaning identical to that of the constraints participating in the synthesis.

The targeted synthesis is conducted on the specific constraint, which has already been proven sound(*SoundConstr*) or complete(*CompleteConstr*). DeepConstr synthesizes raw constraint(refered as *raw*) with thoes constraints using a specific boolean operator. This step aims to guide either sound or complete constraints toward a logically meaningful direction. In specific, synthesizing *sound* constraint with raw constraint using OR($\lor$) will increase the *completeness* of an already *sound* constraint. Similarly, merging a *complete* constraint with the AND($\land$) operator can improve the *soundness* of an already *complete* constraint. On the other hand, naive synthesis is conducted with both boolean operators. The following examples demonstrate the concrete process of naive synthesis. Suppose that LLMs generate a raw constraint `x.rank > 1`, while the ground truth is `x.rank > 1 and x.rank < 4`. The constraint `x.rank > 1` is partially correct because it includes both ground truth and the incorrect range (`x.rank ≥ 4`). Therefore, DeepConstr saves this constraint to the constraint pool and continues refinement. On another iteration, LLMs will generate another partially correct constraint, `x.rank < 4`. This new constraint will be combined with `x.rank > 1` (lines 11-13), resulting in `x.rank > 1 and x.rank < 4` as one of the outputs. This synthesized constraint will score full marks in the fitness function (equation 8) and be selected. In this way, DeepConstr can mitigate the limitation of LLMs that often fail to generate complete constraints.

4.3 Constraint Measurment

The constraint pool is refined based on the result of constraint measurement. This follows the idea of genetic algorithm [19], updating the constraint pool with higher-scoring constraints. This module calculates the scores by defining fitness function, which is designed to consider both soundness and completeness of constraint. The soundness of constraint is measured by examining whether the test case derived from the constraint triggers a error change on DL operator. Completeness is measured by finding counter test cases from a complementary set of constraints.

Error Change. For a DL operator O, let there be a set of n error messages $E = \{e_1, e_2, \ldots, e_n\}$. To address this, we generate a set of constraints $C = \{c_1, c_2, \ldots, c_n\}$, where each constraint c_k is tailored to resolve the corresponding error message e_k. T_k represents the collection of all test cases t that satisfies constraint c_k, denoted as $t_k \vDash c_k$. Thus, T_k is defined as

$$T_k = \{t_i \mid t_i \vDash c_k, i = 1, 2, \ldots, m\}$$

Since the test case t_k satisfies the constraint c_k, the operator O with t_k will not trigger the error message e_k. This indicates that the error state has undergone a change, which means c_k is effective at resolving the error message. Similarly, We assess the quality of a constraint c_k by watching whether c_k has triggered *error_change* on O. Formally, we define *error_change* as below:

Definition 4.1 (Error Change). *For an operator O with input t_k, the error_change for error e_k is defined as*

$$error\_change(O, t_k, e_k) = \begin{cases} 1 & \text{when } O \text{ does not return the error } e_k \\ 0 & \text{when } O \text{ still returns the same error } e_k \end{cases}$$

We consider that the operator O has experienced an *error_change* if O with t_k returns a different error message compared to e_k, or it does not report any error at all.

Soundness. The *soundness* of constraint c_k implies whether the constraint c_k always resolves the target error state e_k. We ensure the *soundness* of c_k, proposed to resolve e_k, by inspecting the *error_change* on e_k with $t_k \models c_k$. Hence, soundness(c_k) is :

Definition 4.2 (Soundness). soundness*(c_k) is true,*

$$if \quad \forall t \in T_k, error\_change(O, t, e_k) \text{ returns } 1 \tag{4}$$

Completeness. Completeness of constraint indicates the constraint c_k does not influence the input space unrelated to resolving e_k. Our insight is that we can assess *completeness* of c_k by utilizing *complementary set* constructed by the negation of the constraint. That is, We ensure the *completeness* by inspecting whether the correct input space is still covered by the $\neg c_k$. We formulate it as follows:

Definition 4.3 (Completeness). completeness*(c_k) is true,*

$$if \quad \forall t \in T_k^{\complement}, error\_change(O, t, e_k) \text{ returns } 0 \tag{5}$$

Note that $T_k^{\complement}$ is the complementary set of test case set T_k. Thus, $T_k^{\complement}$ consists of test case t that does not satisfy c_k, which means that $t \models \neg c_k$. We consider c_k is *complete* when every element $t_k \in T_k^{\complement}$ do not trigger *error_change.*

Fitness Function. Since computational resources are limited, it is infeasible to explore the whole input space. Therefore, we measure the approximate values of *completeness* and *soundness* on constraint c_k by investigating the set $\hat{T_k}$, which contains a considerably large number n of test cases. We introduce equations for $\hat{soundness}$ and $\hat{completeness}$ to compute these approximate values.

First, we estimate the approximate value of *soundness*(c_k) using the following equation. The equation measures the proportion of elements in $\hat{T_k} = \{t_1^k, t_2^k, ..., t_n^k\}$ that actually triggered *error_change.*

$$\hat{soundness}(c_k) = \frac{1}{n}\sum_{i=1}^{n} error\_change(O, t_i^k, e_k) \tag{6}$$

Next, we define the equation that approximately measures the *completeness*(c_k). The Φ is the proportion of element in $T_k^{\complement} = \{\tau_1^k, \tau_2^k, ..., \tau_n^k\}$ that still triggers *error_change.* Φ can be defined as:

$$\Phi = \frac{1}{n}\sum_{i=1}^{n} error\_change(O, \tau_i^k, e_k)$$

The $\hat{completeness}(c_k)$ score reflects the input space that is wrongly excluded by c_k, which is measured by Φ. Inspired by the recall metrics [40], we define the $\hat{completeness}(c_k)$ as follows:

$$\hat{completeness}(c_k) = \frac{\hat{soundness}(c_k)}{\hat{soundness}(c_k) + \Phi} \tag{7}$$

On the top of the definitions of $\hat{completeness}$ with $\hat{soundness}$, we measure the overall quality of constraint c_k by defining a *fitness* function through the harmonic mean [40] of the two metrics.

$$fitness(c_k) = 2 \times \frac{\hat{soundness}(c_k) \times \hat{completeness}(c_k)}{\hat{soundness}(c_k) + \hat{completeness}(c_k)} \tag{8}$$

Refinement. Based on the *fitness* score, DeepConstr maintains a high-quality constraint pool, keeping the top-k constraints and pruning others. Specifically, DeepConstr updates the constraint pool if it finds a constraint with a higher score than the lowest score in the constraint pool. The constraint pool is then used for the constraint synthesis of the next iteration.

End of Iteration. There are two conditions that stop the iteration of constraint refinement. First, if we have found the constraint c_k that achieves a full mark on the *fitness* function, we stop the process as it indicates we have found the perfect constraint. Second, when the number of iterations exceeds a pre-set threshold, we stop refining and return the best constraint from the constraint pool.

5 Implementation

We implemented DeepConstr with approximately 4k lines of Python codes. For raw constraint exploration, we utilized GPT-3.5-turbo [12] and GPT-4 [5]. Based on the grammar definition of DeepConstr, the raw constraint is converted into the corresponding expression of Z3-solver [7], one of the Satisfiability Modulo Theories Solvers. In this way, DeepConstr is able to synthesize constraints and derives test cases from the constraint set utilizing the native support of Z3-solver. For the constraint synthesis module, we retain the top-5 constraints for naive synthesis and top-1 constraint for targeted synthesis, to balance the diversity of the pool and save computational resources. For the constraint refinement module, considering the difference in searching space between soundness and completeness evaluation, we generated 1.5 times more test cases on completeness than we did for soundness when assessment. Specifically, we set n to 500 for $\hat{soundness}$ (equation 6), and 750 for measuring $\hat{completeness}$ (equation 7). When updating the constraint pool, we also considered the length of each synthesized constraint to prevent them from becoming overly long.

Test Case Generation. For efficient test case generation for DL library testing, DeepConstr reused an existing DL library fuzzer, NNSmith [24]. To achieve this, we made the extracted constraints compatible with NNSmith, wrapped the DL operator with them, and then passed the sets of DL operators to the fuzzer.

Oracle. We adopt the bug detection oracles that are in line with previous studies [24, 25, 38, 46]. It involves two types of oracles: differential testing and crash detection. For differential testing, we generate a test case and lower it to be compiled with different compilers, and then compare the outputs to check if they are identical. Given that DL operators handle complex numerical operations, numerical stability is a critical factor in differential testing, which is also mentioned by [45]. To reduce the incidence of false positives, we set the tolerance threshold as 1e-2.

6 Evaluation

We evaluate DeepConstr by answering the following three research questions:

- **RQ1 (§6.1)**: How well does DeepConstr perform compared to other state-of-the-art fuzzers?
- **RQ2 (§6.2)**: What are the soundness and completeness of the constraints refined by DeepConstr?
- **RQ3 (§6.3)**: Can DeepConstr detect previously unknown bugs for real-world DL libraries?

Experiment Setup. We tested the two most popular DL libraries, *i.e.,* TensorFlow [4] and PyTorch [30], with each library supporting approximately over 1,500 operators. We perform our evaluation on an AMD EPYC 7763 with 128 cores running on Ubuntu 20.04. All experiments are conducted in the same environment and repeated five times. As aligned with previous research [25], we compiled TensorFlow with GCC-12.2 and GCOV [1], while PyTorch is compiled with Clang-14 [2].

6.1 Comparative Study

To further investigate the effectiveness of our approach, we evaluate DeepConstr with state-of-the-art DL library fuzzers and a variant of DeepConstr.

- NNSmith [24]: A fuzzer that performs test case generation with approximately 60 DL operators whose constraints are *manually* crafted by domain experts.
- NeuRI [25]: A fuzzer that automatically infers the constraints of a DL operator through real-world *invocations* of the operator, which are collected from various sources, such as developer tests.
- DeepConstr$^-$: A variant of DeepConstr, which removes the completeness objective from the fitness function in the constraint refinement process, and only considers the soundness.

Supported Operators. Table 1 presents an overview of the number of operators supported by different tools. For PyTorch, DeepConstr additionally supports 220 and 784 operators that are not supported by NeuRI and NNSmith, respectively. For TensorFlow, DeepConstr supports 68 and 195 operators that are not supported by NeuRI and NNSmith, respectively. This is because DeepConstr adopts a black-box based approach that does not require manual efforts or external test cases. NNSmith requires human experts' efforts, and thus cannot scale to support a large number of operators. NeuRI, on the other hand, supports a broader range of DL operators based on collected test cases, but the absence of test cases for some operators limits its ability to support additional operators.

DeepConstr does not support 99 and 84 operators that are supported by NeuRI in PyTorch and TensorFlow, respectively. There are two factors contributing to this: (1) element-wise constraints and overly complicated constraints, and (2) incorrect error messages. DeepConstr avoids modeling each tensor element into the constraint because doing so would require excessive computational resources when deriving test cases. This approach aligns with previous works [24, 38, 46]. Consequently, DeepConstr cannot extract constraints from error messages such as "number of elements should be less than n" or "the max value of tensor element should be bigger than 0". Additionally, DeepConstr cannot extract valid constraints for some operators with overly complicated constraints. For example, an operator(`torch.nn.AdaptiveLogSoftmaxWithLoss`) requires its arguments to be an ordered sequence of integers sorted in increasing order. DeepConstr fails to generate valid constraints for such cases. However, NeuRI does not get affected by this issue because it reuses saved test cases when fuzzing if it fails to extract constraints. This limitation prevents DeepConstr from supporting some operators supported by NeuRI.

In addition to this, there are wrong or unclear error messages, which makes it difficult for DeepConstr to refine constraints. Some of these error messages are wrongly implemented, for example, an error message[2] "Padding length too large," returned by the `torch.nn.ConstantPad` operator wrongly describes the root cause of the error. For this case, DeepConstr was unable to generate a constraint that resolves the error. This was because the actual issue was not with the padding value but with the input dimension. This issue with the error message has been fixed to "Input dimension should be at least 3 but got 2," accurately reflecting the underlying problem. For those operators where error messages are correct and precise, DeepConstr is able to support them effectively.

Table 1: The number of DL operators supported by different tools. 'D' denotes DeepConstr, 'R' represents NeuRI, and 'N' refers to NNSmith. 'PT' and 'TF' are PyTorch and Tensorflow, respectively.

Library	D	R	N	D vs R			D vs N		
				$D \cap R$	$D - R$	$R - D$	$D \cap N$	$D - N$	$N - D$
PT	843	722	59	623	220	99	59	784	0
TF	258	274	63	190	68	84	63	195	0

Testing Effectiveness. In this section, to evaluate the effect on testing that comes from the constraints refined by DeepConstr, we compare its testing effectiveness with NNSmith, NeuRI, and DeepConstr$^-$. Specifically, we examine two crucial metrics that are considered significant in testing: branch coverage and the number of test cases. In our experimental setup, we collect branch coverage after each tool executes a single operator for 15 minutes. We then repeat this until we have completed testing for all supported operators. The detailed explanation of metrics is:

- *Branch coverage per operator*: We evaluate each baseline based on the branch coverage it achieves. The better branch coverage indicates that the tool provides better constraints for test case generation. Notably, when comparing each of the two tools, we only include their intersection of supported operators for fair comparison. This is because the broad range of operator support by DeepConstr will interrupt fair comparison. As such, we did not assess the overall coverage, which includes all supported operators at once, because it is significantly influenced by the number of operators supported.
- *The number of test cases*: We assess the efficiency of baselines by counting the number of successful test cases generated within the same time period, which directly reflects the fuzzer's efficiency.

[2]https://github.com/pytorch/pytorch/issues/104508

Table 2: Comparative experiment results on branch coverage and the number of test case.

Library	Tool	#Testcase	#Operators Improved	%Cov Improved
PyTorch	DeepConstr	1954.61	-	-
	NNSmith	14626.52	29/59 (49.15%)	39.00%
	NeuRI	123.31	363/623 (58.27%)	25.94%
	DeepConstr⁻	2660.60	360/843 (42.7%)	1.25%
TensorFlow	DeepConstr	1292.93	-	-
	NNSmith	3674.80	24/63 (38.1%)	-4.17%
	NeuRI	138.11	118/190 (62.1%)	27.26%
	DeepConstr⁻	1459.21	109/258 (42.25%)	6.41%

Table 2 is the experimental result that compares the effectiveness of test case generation with a single operator. "#Testcase" column means the average number of successful test case generations of the same time period, 15 minutes. "#Operators Improved" indicates the number of operators that DeepConstr achieves more coverage than other baselines. The proportion of increased operators is divided by the intersected operators that DeepConstr and another baseline commonly support. "%Cov Improved" indicates the average coverage improvement that DeepConstr achieved compared to another baselines. During the evaluation, we measure only the coverage related to the execution of operators. Coverage that is unrelated, such as importing libraries and launching the framework, is excluded from our coverage collection.

v.s. Manually Crafted Constraints. Based on the result of the "#Testcase" column in Table 2, NNSmith emerges as the most efficient fuzzer that outperforms all baselines, generating 3 to 10 times more test cases on TensorFlow and PyTorch, respectively. This is because NNSmith is designed with manually tailored constraints, eliminating the need to explore the unknown input space. Meanwhile, even compared with these manually-tailored constraints, DeepConstr has managed to improve coverage of 29 operators out of 59 operators for PyTorch, improving averaged 39.00% coverage for each operator. In the case of TensorFlow, DeepConstr improved coverage for 24 operators out of 63 operators. However, it did not achieve an overall increase in coverage, performing -4.17% worse than NNSmith. This is because the constraint set of TensorFlow is more complex than that of PyTorch, making it challenging for DeepConstr to compete with manually-tailored constraints. Overall, the experimental result shows that constraints refined by DeepConstr can achieve comparative testing performance than constraints that are manually crafted by experts.

There are two reasons for these results. First, by iteratively measuring and refining constraints, DeepConstr can generate constraints that achieve a level of soundness and completeness comparable to those crafted by experts. Moreover, manually-tailored constraints still have the possibility of overlooking corner cases due to the complexity of operators. For example, NNSmith has designed logical operators such as `torch.logical_and` and `torch.logical_or` to only accept boolean tensors. However, these operators not only accept boolean tensors but also allow other types of tensors as input (see Listing 3). Furthermore, for the squeeze operators such as `torch.Tensor.squeeze`, NNSmith restricts the rank of its input tensor to be larger than zero, whereas the operator does not have such a constraint. Conversely, DeepConstr generates and refines constraints without any assumption, which allows it to successfully cover the input space that is not covered by NNSmith.

v.s. Automatically Inferred Constraints. As shown in Table 2, DeepConstr achieves higher coverage on 363 operator out of 623 common operators for PyTorch while increased coverage of 118 operators out of 190 operators of TensorFlow. The main reason for this is caused by the overly strict constraints adopted by NeuRI. In many cases, the constraint inferred by NeuRI overfits specific test cases. For instance, NeuRI at most generates three distinct test cases for an operator, `torch.Tensor.cumsum_`. This is because NeuRI failed to infer the accurate constraint from the saved test cases, thereby generating test cases by reusing them. As such, NeuRI adopts a strict constraint on which fuzzers may struggle to produce various test cases. As a result, NeuRI generates 111 test cases for `torch.Tensor.cumsum_` in 15 minutes, while DeepConstr generates 3484 test cases for the same operator at the same time. This explains why NeuRI generally generates fewer test cases than other baselines. For approximately 30% of the operators, NeuRI inferred tight constraints that restrict the test input space. This results in DeepConstr covering more branches for many operators. On average, DeepConstr improves 25.94% and 27.26% branch coverage for PyTorch and TensorFlow, correspondingly.

Table 3: Examples of detailed constraints that is adopted by DeepConstr and DeepConstr⁻.

Target Error Message	Constraints of DeepConstr⁻	Constraints of DeepConstr
Result type Float can't be cast to the desired output type Short	`dtype(input) = dtype(out)` ∧ `dtype(input) = float32`	`dtype(input) = dtype(out)`
Non-empty 3D or 4D (batch mode) tensor expected for input	`rank(input) = 3` ∧ `rank(input) ≠ 5`	`rank(input) = 3` ∨ `rank(input) = 4`
Dimension out of range (expected to be in range of [-1, 0], but got 1)	`-1 < dim` ∧ `dim ≤ 0`	`-dim ≤ len(input)` ∧ `len(input) > dim`

v.s. Constraints without Completeness. DeepConstr⁻ is a variant of DeepConstr that pursues only soundness and does not address completeness when refining constraints. We compare DeepConstr with DeepConstr⁻ to demonstrate that solely considering the soundness does not guarantee higher testing effectiveness, and completeness should be considered during constraint generation. As shown in Table 2, DeepConstr outperforms DeepConstr⁻ in branch coverage, even though it generates fewer test cases within the same time period. Specifically, DeepConstr increases coverage on 360 operators out of 843 operators in PyTorch. In addition, DeepConstr gains better branch coverage on 109 operators out of 258 operators in TensorFlow.

Table 3 shows examples where DeepConstr⁻ and DeepConstr adopt different constraints for the same error message. In detail, when the error message states, "Result type Float can't be cast to the desired output type Short," DeepConstr adopts `dtype(input) = dtype(out)` as a constraint. However, DeepConstr⁻ adopted `dtype(input) = float32` ∧ `dtype(input) = float32`, which overly narrows the input space by restricting the tensor data type to `float32`. Since DeepConstr⁻ only considers soundness when refining, it cannot identify the narrowed input space by the sub-constraint `dtype(input) = float32`, thereby returning it as the best constraint. Similarly, an error message that states, "expected to be in the range of [-1, 0], but got 1" indicates the need for certain arguments (*e.g.,* `dim`) to fall within the range of the tensor rank. DeepConstr^{-} adopts the $-1 < \texttt{dim} \wedge \texttt{dim} \leq 0$ as its constraint because the error message has explicitly referred to the range of $(-1, 0]$. In addition, since the minimum value of tensor rank is zero, setting `dim` to zero will always prevent the error from occurring. However, it cannot yield diverse test cases. Conversely, DeepConstr can successfully identify the narrowed input space of given constraints using the complementary set, thereby finding complete and sound constraints.

6.2 Constraints Assessment

In this section, we evaluate the quality of constraints that have been refined by DeepConstr and DeepConstr^{-} to demonstrate the capability of constraint refinement with the components that we proposed. Specifically, we assess the effectiveness of constraint refinement in DeepConstr and DeepConstr^{-} by analyzing the overall scores of constraints that they have extracted. Two key performance metrics, soundness, and completeness, are evaluated based on previously defined equation 6 and equation 7, respectively. The other metric, *num*, means the number of sub-constraints extracted for an operator.

Table 4: Statistical overview of constraints on PyTorch and TensorFlow. μ represents the mean value, and *mid* denotes the median value. 'PT' and 'TF' indicate PyTorch and Tensorflow, respectively.

	metrics	DeepConstr	DeepConstr^{-}
PT	*num*	μ = 7.61, *mid* = 6	μ = 8.82, *mid* = 7
	soundness	μ = 95.29%, *mid* = 100.00%	μ = 95.54%, *mid* = 100.00%
	completeness	μ = 80.38%, *mid* = 88.24%	μ = 76.32%, *mid* = 76.34%
TF	*num*	μ = 7.74, *mid* = 5	μ = 9.26, *mid* = 8
	soundness	μ = 94.89%, *mid* = 100.00%	μ = 94.45%, *mid* = 100.00%
	completeness	μ = 89.68%, *mid* = 100.00%	μ = 85.34%, *mid* = 95.74%

Table 4 demonstrates that the overall refinement design of DeepConstr using a genetic algorithm is effective in generating higher-quality constraints. In Table 4, "μ" denotes the mean value and "*mid*" indicates the median value. The data reveals that both the DeepConstr and DeepConstr^{-} successfully refine constraints that resolve the error messages. This success is reflected in the soundness columns for both PyTorch and TensorFlow, where both DeepConstr and DeepConstr^{-} achieve a median soundness score of 100, with the mean score of both DeepConstr and DeepConstr^{-} achieves 94.92% to 96.53%. The overall design of DeepConstr contribute this suc This result is due to the fact that the constraint is synthesized iteratively and gradually enhanced, leading to high-quality constraint refinement. The *num* rows also indicate that both DeepConstr and DeepConstr^{-} demonstrate effective sub-constraints finding ability, extracting averaged 7.61 and 8.82 on PyTorch, and 7.74 and 9.26 on TensorFlow.

Moreover, DeepConstr demonstrates significant improvements in completeness for both PyTorch and TensorFlow, with only a slight reduction in soundness compared to DeepConstr^{-}. This is because DeepConstr^{-} did not prioritize the completeness score during the refinement process. As a result, DeepConstr achieved higher coverage on many operators, as shown in Table 2. This demonstrates that considering completeness is more important than soundness for testing thoroughness.

In addition, it is important to note that DeepConstr extracts fewer sub-constraints compared to DeepConstr^{-}. This is because, without considering completeness, one cannot determine whether the newly added sub-constraints would be truly helpful in generating valid input or if they would just decrease the input space. Considering completeness can mitigate this problem. Let's consider the first example in Table 3. Since the ground-truth constraint for the target error message is `dtype(input) = dtype(out)`, one of the sub-constraints that DeepConstr^{-} generate, `dtype(input) = float32`, is not correct. It makes the tensor data type of `input` to be `float32`, decreasing input space without satisfying the ground-truth constraint. However, DeepConstr^{-}, which only considers soundness, cannot determine if this helps to generate valid input or meaninglessly decreases input space. On the other hand, DeepConstr, by considering completeness, can determine whether the sub-constraint is helpful to valid test case generation and, as a result, remove it from the final output. In a similar way, DeepConstr^{-} tends to generate more sub-constraints than DeepConstr. This insight suggests that more sub-constraints may not necessarily help find correct constraints.

6.3 Bug Finding

To evaluate the ability of DeepConstr in bug finding, we intermittently run the prototype of DeepConstr for about a month in two mainstream DL libraries. We conducted tests on the stable versions of PyTorch (2.2.0) and TensorFlow (2.12.0), as well as their nightly versions. We count the bugs on the basis of bug reports, which are classified into two statuses: *fixed*, a patch has been merged to fix the bug; *confirmed*, it has been reproduced/diagnosed as a fault or directly assigned to developers for fixing it.

Table 5: Overview of reported bugs.

	Symptom	Total	Confirmed	Fixed
PyTorch	Inconsistency	30	21	13
	Runtime error	34	34	30
	Others	10	10	4
TensorFlow	Inconsistency	6	5	2
	Runtime error	4	2	2
	Others	0	0	0
Total		84	72	51

Table 5 shows the details of bugs found by DeepConstr. To sum up, we found 84 previously unknown unique bugs, 72 of which were confirmed, out of 51 were fixed. At the time of writing this paper, the remaining bugs are still awaiting confirmation from developers. 11 of PyTorch bugs found by us are assigned high priority because developers regard them as critical vulnerabilities that highly affect the normal function of DL libraries. It is important to note that most of the bugs reside in the operators that were heavily tested by existing fuzzers [24, 25], which means that DeepConstr can find bugs that other tools are not able to find out.

We also analyze the type of bugs found by DeepConstr. It identified various bug types, including 36 inconsistency bugs, 38 runtime error bugs, and 10 other bugs. Inconsistency bugs are usually detected by discrepancies in output. Specifically, these bugs occur when an optimized computation graph yields incorrect results compared to its original graph. Runtime error bugs refer to scenarios where the DL compiler fails to compile the computational graph, indicating issues inherent in the compile system. It includes crash bugs such as core dumped or segmentation fault bugs. Other bugs include errors related to documentation and error messages, and these errors are identified through manual inspection of the training log when we found that DeepConstr encountered failures in constraint extraction.

Responsible Bug Reporting. The number of bugs reported for each library is closely tied to the responsiveness of its developers. Our practice has been to wait for feedback from developers before proceeding with further bug discovery efforts. Thus, a higher count of reported bugs on PyTorch does not necessarily denote that PyTorch is less robust. Instead, it highlights its developers' readiness to engage. In the case of TensorFlow, after not fixing our first 10 bug reports, we decided to stop our reporting to adhere to responsible bug discovery practices[34].

Case Study. In the following paragraphs, we will discuss two distinct error examples missed by the previous fuzzers but found by DeepConstr. All of these errors were found in the operator that *have been already tested* by previous fuzzers. Furthermore, all of these errors have been assigned *high-priority* tags by the developers.

```
import torch
class model(torch.nn.Module):
    def __init__(self, y):
        super().__init__()
        self.y = y
    def forward(self, x):
        z = torch.t(self.y)
        y = torch.abs(input=x, out=z)
        return y
storage = torch.rand([1,2])
input = torch.rand([2,1])
output = Model(storage)(input)
```

Listing 2: Minimized code snippet of bug sample 1.

Bug Sample 1: Inconsistency Error in the torch.abs Operator. Listing 2 presents a simplified Python code snippet that triggers a bug. The code describes an operation that generates output tensors by applying an absolute operation, saving the result to the tensor of `out` argument. The bug's root cause lies in using `torch.abs` when transposed tensor passing to the argument `out`. Specifically, the bug exists in `as_strided()` method implemented under the `torch.abs` when calling an `out` argument. It is designed to create a new view of an existing tensor, specifying its size and stride. When passing the tensor that has been applied, the transpose operation through `torch.t`, `as_strided()` is not applied correctly, leading to inconsistent behavior.

To find out this bug, one must identify the constraints related to the argument `out` of `torch.abs`. However, the previous work failed to model the constraints related to them, leading them to miss this bug. Notably, the argument `out` of `torch.abs` seldom appear in real-world use cases. Hence, for a test case based constraint inference fuzzer, deducing the constraints related to the `out` argument of `torch.t` becomes difficult. In contrast, DeepConstr can deduce constraints that rarely appear in real-world test cases, which could easily be missed by other fuzzers.

```
import torch
class Model(torch.nn.Module):
    def forward(self, i, ot, out):
        return torch.logical_or(input=i, other=ot, out=out)

input = torch.rand([3, 3, 2], dtype=torch.float32)
other = torch.rand([3, 3, 2], dtype=torch.float32)
out = torch.rand([3, 3, 2], dtype=torch.float16)
model = Model()
torch.compile(model.forward,)(i, ot, out)
```

Listing 3: Code snippet of bug sample 2.

Bug Sample 2: Core Dumped Error of Logical Operator. Listing 3 presents a minimized Python code snippet that triggers a crash in PyTorch. The code performs an `OR` operation between two tensors, `i` and `ot`, and saves the result to the tensor `out`. The root cause of bug sample 2 is related to float16 legalization. Specifically, the crash occurs when attempting to store a float32 tensor in an output tensor, `out`, which was previously defined as float16 data type. NNSmith has modeled and tested the operator `torch.logical_or` but failed to detect the bug. The primary reason is that in NNSmith's implementation, `torch.logical_or` is only compatible with boolean tensors. Although restricting the data type to boolean tensors aligns with the intended use of the operator, `torch.logical_or` in practice imposes no such tensor data type limitations, causing NNSmith to overlook potential vulnerabilities. In contrast, DeepConstr does not impose data type restrictions, enabling it to uncover this bug.

7 Discussion

Limitations. While DeepConstr marks a significant advancement in enhancing the completeness of constraint, it does possess inherent limitations. Specifically, DeepConstr can not generate constraints with below cases : *(1) Incorrect Error Messages:* It cannot extract constraints from incorrect error messages, which means the description is not related to the root cause of the error message. *(2) Constraints Related to Tensor Elements:* To optimize resource utilization, we followed the approach of other works [24, 25, 38], and avoided individually modeling each tensor element into the constraint. As a result, DeepConstr does not support constraints that are related to tensor elements. For example, DeepConstr can not extract constraints from the error message such as "number of elements should be less than n" or "the max value of tensor element should be bigger than 0".

Order of Error Message Resolving. The order in which error messages are triaged is important because it affects the efficiency of solving them. This is because a program checks conditions sequentially. If an earlier condition is not met, subsequent condition will not be checked, as the program's attention is captured by the initial condition check. We prioritized error messages based on their frequency of occurrence to approximate the optimal order for handling them. This approach is based on the observation that randomly generated test cases often fail to meet the first condition, thereby being caught more frequently by the earlier input checks. While static analysis methods, such as dominance analysis [33], could potentially offer a more precise analysis of the sequence, these white-box analysis techniques pose greater challenges in adaptation and could reduce the scalability of the tools. In future work, we are interested in further exploring these methodologies to enhance our approach's precision without compromising the tool's scalability. This exploration aims to balance the depth of analysis and the practical applicability in diverse operational environments.

Generalizability. DeepConstr can be applied to other DL frameworks. We spent a few efforts to support NumPy [18]. We have implemented the following steps: **(1)** We generated backend code responsible for converting IR into NumPy native code (69 lines of code). **(2)** We mapped the NumPy native data types (*e.g.,* `numpy.float32, numpy.float64`) to the internal data types that DeepConstr maintains (50 lines of code). In summary, we have integrated the NumPy backend into DeepConstr by adding 119 lines of code. It took less than 2 hours with one graduate student. This demonstrates the generalizability of DeepConstr. We also ran DeepConstr for 5 hours to extract constraints on NumPy operators, and it extracts 184 sub-constraints across the 24 operators.

8 Related Work

As the DL applications expand, the stability of DL libraries becomes increasingly critical. As a result, many works have been proposed for both testing DL models [16, 28, 31, 47, 48] and DL libraries [10, 15, 24, 25, 32, 38, 41, 42, 46]. For testing DL libraries, someone thinks that existing fuzzing techniques [21, 23, 35, 50, 51], Python program fuzzers [22, 26] or API fuzzers [11, 14] can be the solution. However, unlike traditional software, DL libraries more closely resemble a domain-specific language tailored specifically for tensor computation and neural network construction, necessitating specialized testing techniques. Many customized fuzzers are proposed for effectively testing the DL libraries. These are divided into two categories: model-level fuzzers and operator-level fuzzers.

Model Level Fuzzer. The most direct method for testing DL libraries is to reuse generated DL models and execute them on these libraries. Following this, CRADLE [32] first utilizes the open-sourced DL models and uncovers the bugs by differential testing. However, open-sourced DL models are limited, and many are heavily executed, making it hard to explore unknown vulnerabilities. To generate more diverse DL models, LEMON [41] and Audee [17] mutate the existing models with manually designed mutation rules. In addition to reusing existing models, Muffin [15] constructs models from scratch using a series of DL operators, and Luo [27] further enhanced the model generation by leveraging graph coverage strategies. Nevertheless, considering that each DL operator possesses a complicated set of constraints, the aforementioned model level fuzzers are limited to test DL operators in restricted scenarios [25].

Operator Level Fuzzer. FreeFuzz [42] tries to fuzz each operator by mutating existing test cases of DL libraries but struggles to generate valid test cases. To effectively generate valid test cases for DL library testing, many studies adopted constraint-guided generation [10, 24, 25, 38, 46, 49]. NNSmith [24] utilizes constraints written by human experts. However, the extensive manual effort for analyzing each DL operator makes NNSmith only accommodate a limited number of operators, emphasizing the need for automatic constraint generation. Doctor [46] is the first approach that automatically synthesizes constraints from the documentation. However, constraints extracted from documentation make them challenging for valid test case generation. NeuRI [25] tries to infer the constraints from a set of invocations, such as developer test cases. Although it achieves great success in testing a broad range of operators, NeuRI sometimes overfits the offered test cases, limiting its possible input space. AceTest [38] tries to infer the constraints from the source code, but they struggle to identify the user-controllable variables, which is the main part of the constraint. In contrast, DeepConstr can concretely identify constraints by using error messages and validating their completeness using a complementary set.

LLMs for Fuzzing. As LLMs have improved, they can now be used for fuzzing [13, 29, 44]. Some works [8, 9] have used LLMs for testing DL libraries and found many bugs. However, using LLMs for fuzzing presents several challenges [20]. First, having LLMs directly generate test cases would be costly, as fuzzing involves producing a large number of test cases [6, 39]. Secondly, test cases generated by LLMs lack diversity, since LLMs tend to respond similarly when given the same prompt. Finally, LLMs struggle to generate valid test cases for complex programs due to their limited ability to process long texts or examples [36]. In contrast, DeepConstr utilizes LLMs to generate constraints for fuzzing. This approach mitigates the above problems while retaining the advantages of fuzzing.

9 Conclusion

DeepConstr is a novel and general approach that generates and refines a complicated constraint set for DL library testing. Our method is twofold. First, we divide the complex constraint of an operator into several independent and relatively simple constraints guided by error messages. Next, we leverage the complementary set of each independent constraint to identify the incomplete areas of this constraint, and thus refine it to be more complete. We evaluate the performance of DeepConstr on two mainstream DL libraries, *i.e.,* PyTorch and TensorFlow,and compare to state-of-the-art fuzzers such as NeuRI, and NNSmith. DeepConstr discovered 84 previously unknown bugs out of which 72 confirmed with 51 fixed. Compared to state-of-the-art fuzzers, DeepConstr achieves greater coverage of 58.27% of operators supported by NeuRI and 49.15% of operators supported by NNSmith in PyTorch, and 62.1% of operators supported by NeuRI and 38.1% of operators supported by NNSmith in TensorFlow.

Data-Availability Statement

All data and materials supporting the findings of this study are openly available at Zenodo [3].

Acknowledgements

We appreciate the reviewers' valuable and insightful comments. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000) and NSFC Program (No. 92167101, 62021002).

References

[1] 2022. GCOV. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[2] 2022. Source-based Code Coverage — Clang 15.0.0 documentation. https://releases.llvm.org/15.0.0/tools/clang/docs/SourceBasedCodeCoverage.html.

[3] 2024. Artifact for DeepConstr. https://doi.org/10.5281/zenodo.12669927

[4] 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.

[5] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[6] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and reflections. *IEEE Software* 38, 3 (2020), 79–86.

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

[8] 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*. 423–435.

[9] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[10] 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.

[11] Adeel Ehsan, Mohammed Ahmad M. E. Abuhaliqa, Cagatay Catal, and Deepti Mishra. 2022. RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions. *Applied Sciences* 12, 9 (2022). https://doi.org/10.3390/app12094369

[12] Luciano Floridi and Massimo Chiriatti. 2020. GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines* 30 (2020), 681–694.

[13] Jingzhou Fu, Jie Liang, Zhiyong Wu, and Yu Jiang. 2024. Sedar: Obtaining High-Quality Seeds for DBMS Fuzzing via Cross-DBMS SQL Transfer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[14] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API Data Fuzzing. 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, 725–736. https://doi.org/10.1145/3368089.3409719

[15] 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

[16] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 739–743.

[17] Qianyu Guo, Xiaofei Xie, Yi Li, Xiaoyu Zhang, Yang Liu, Xiaohong Li, and Chao Shen. 2020. Audee: Automated Testing for Deep Learning Frameworks. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 486–498.

[18] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[19] John H Holland. 1992. Genetic algorithms. *Scientific american* 267, 1 (1992), 66–73.

[20] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, et al. 2024. When Fuzzing Meets LLMs: Challenges and Opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 492–496.

[21] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. 2023. Continuous fuzzing: a study of the effectiveness and scalability of fuzzing in CI/CD pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 25–32.

[22] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. Pyrtfuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1645–1659.

[23] Jie Liang, Zhiyong Wu, Jingzhou Fu, Yiyuan Bai, Qiang Zhang, and Yu Jiang. 2024. {WingFuzz}: Implementing Continuous Fuzzing for {DBMSs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 479–492.

[24] 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 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 530–543.

[25] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. 2023. Neuri: Diversifying dnn generation via inductive rule inference. *arXiv preprint arXiv:2302.02261* (2023).

[26] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An Empirical Study of Automated Unit Test Generation for Python. *Empirical Softw. Engg.* 28, 2 (jan 2023), 46 pages. https://doi.org/10.1007/s10664-022-10248-w

[27] 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

[28] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).

[29] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. , 8024–8035 pages.

[31] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[32] H. Pham, T. Lutellier, W. Qi, and L. 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[33] Reese T Prosser. 1959. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. 133–138.

[34] John Regehr. 2017. *Responsible and Effective Bugfinding*. https://blog.regehr.org/archives/2037

[35] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[36] Uri Shaham, Maor Ivgi, Avia Efrat, Jonathan Berant, and Omer Levy. 2023. Zeroscrolls: A zero-shot benchmark for long text understanding. *arXiv preprint arXiv:2305.14196* (2023).

[37] Ali Shatnawi, Ghadeer Al-Bdour, Raffi Al-Qurran, and Mahmoud Al-Ayyoub. 2018. A comparative study of open source deep learning frameworks. In *2018 9th international conference on information and communication systems (icics)*. IEEE, 72–77.

[38] Jingyi Shi, Yang Xiao, Yuekang Li, Yeting Li, Dongsong Yu, Chendong Yu, Hui Su, Yufeng Chen, and Wei Huo. 2023. ACETest: Automated Constraint Extraction for Testing Deep Learning Operators. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 690–702.

[39] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.

[40] Cornelis Joost "Keith" van Rijsbergen. 1979. *Information Retrieval* (2nd ed.). Butterworth, London, GB; Boston, MA.

[41] 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

[42] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.

[43] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting

Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022) Main Conference Track*.

[44] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[45] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–28.

[46] 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*. 176–188.

[47] Quan Zhang, Yifeng Ding, Yongqiang Tian, Jianmin Guo, Min Yuan, and Yu Jiang. 2021. Advdoor: adversarial backdoor attack of deep learning system. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 127–138.

[48] Quan Zhang, Yongqiang Tian, Yifeng Ding, Shanshan Li, Chengnian Sun, Yu Jiang, and Jiaguang Sun. 2023. CoopHance: Cooperative Enhancement for Robustness of Deep Learning Systems. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 753–765.

[49] Xufan Zhang, Jiawei Liu, Ning Sun, Chunrong Fang, Jia Liu, Jiang Wang, Dong Chai, and Zhenyu Chen. 2021. Duo: Differential fuzzing for deep learning operators. *IEEE Transactions on Reliability* 70, 4 (2021), 1671–1685.

[50] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. 2023. Towards better semantics exploration for browser fuzzing. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, 604–631.

[51] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: browser API fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 1135–1147.

Received 2024-04-12; accepted 2024-07-03