Testing Deep Learning Libraries via Neurosymbolic Constraint Learning

M M Abid Naziri*
North Carolina State University
Raleigh, North Carolina, USA
mnaziri@ncsu.edu

Shinhae Kim*
Cornell University
Ithaca, New York, USA
sk3364@cornell.edu

Feiran (Alex) Qin
North Carolina State University
Raleigh, North Carolina, USA
fqin2@ncsu.edu

Saikat Dutta
Cornell University
Ithaca, New York, USA
saikatd@cornell.edu

Marcelo d'Amorim
North Carolina State University
Raleigh, North Carolina, USA
mdamori@ncsu.edu

Abstract

Deep Learning (DL) libraries (e.g., PyTorch) are popular in the development of AI applications. These libraries are complex and contain bugs. Researchers have proposed various bug-finding techniques for such libraries. Yet, there is much room for improvement. A key challenge in testing DL libraries is the lack of API specifications. Prior testing approaches often inaccurately model the input specifications of DL APIs, resulting in missed valid inputs that could reveal bugs or false alarms due to invalid inputs.

To address this challenge, we develop Centaur– the *first* neurosymbolic technique to test DL library APIs using dynamically learned input constraints. Centaur leverages the key idea that formal API constraints can be *learned* from a small number of automatically generated seed inputs, and that the learned constraints can be solved using SMT solvers to generate valid and diverse test inputs to test the API.

We develop a novel grammar that represents first-order logic formulae over API parameters and expresses tensor-related properties (e.g., shape, tensor data types, etc.) as well as relational properties between parameters. We use the grammar to guide a Large Language Model (LLM) to enumerate syntactically correct candidate rules, which we then validate using the seed inputs. Further, we develop a custom refinement strategy to prune the set of learned rules to eliminate spurious or redundant rules. We use the learned constraints to systematically generate valid and diverse inputs for the API by integrating SMT-based solving with randomized sampling.

We evaluate Centaur for testing PyTorch and TensorFlow. Our results show that Centaur's constraints have a recall of 94.0% and a precision of 94.0% on average. In terms of coverage, Centaur covers 203, 150, and 9,608 more branches than TitanFuzz, ACETest and Pathfinder, respectively. Using Centaur, we also detect 26 new bugs in PyTorch and TensorFlow, 18 of which are confirmed.

*Equal Contribution

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICSE '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787800

Keywords

Deep Learning Libraries, Software Testing, Specification Inference, Large Language Models

ACM Reference Format:
M M Abid Naziri, Shinhae Kim, Feiran (Alex) Qin, Saikat Dutta, and Marcelo d'Amorim. 2026. Testing Deep Learning Libraries via Neurosymbolic Constraint Learning. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787800

1 Introduction

AI-enabled applications are prolific today. Deep Learning (DL) libraries, such as TensorFlow [1], and Pytorch [32] provide the learning components to build these applications. DL libraries are very complex systems and prone to bugs. Finding bugs in DL libraries is an important and challenging problem that recent prior work has tackled [4, 21, 45, 46].

Despite the impressive advances of recent testing techniques, there still is much room to improve their ability to find *deep bugs*, i.e., bugs associated with the core functionality of the APIs. DL library APIs often do not come with formal input specifications. Hence, most existing techniques are either oblivious to the input constraints of the APIs under test (e.g., DeepREL [11] and FreeFuzz [45]) or are imprecise in modeling input constraints (e.g., DocTer [46] and ACETest [39]). Prior techniques for mining API input constraints use either API documentation (e.g., DocTer) or symbolic code analysis (e.g., ACETest and Pathfinder [22]). Mining constraints from either source can be challenging. API documentation is often incomplete and sometimes specifications can be scattered across multiple documents. On the other hand, symbolic techniques (e.g., by analyzing path conditions) can be costly and imprecise. Due to these challenges, such techniques often mostly reveal shallow bugs, i.e., bugs associated with input validity checkers of the APIs as opposed to bugs related to the deeper functionality of the API. To reveal deep (semantic) bugs, it is important to accurately model the input constraints for each API in DL library.

Our Work. We propose Centaur, the first *neurosymbolic* technique to test DL APIs using dynamically-learned input constraints. Centaur takes as input (1) a DL API, (2) a set of valid inputs for that API and (3) a budget on the number of inputs to generate and reports likely bugs as output. The approach works in three steps. First, Centaur uses a grammar characterizing DL library

`torch.add(input, other, *, alpha=1, out=None) → Tensor`

Adds `other`, scaled by `alpha`, to `input`.

`out[i] = input[i] + alpha × other[i]`

Supports **broadcasting** to a common shape, type promotion, and integer, float, and complex inputs.

Parameters

- `input (Tensor)` – the input tensor.
- `other (Tensor or Number)` – the tensor or number to add to input.

Keyword Arguments

- `alpha (Number)` – the multiplier for `other`.
- `out (Tensor, optional)` – the output tensor.

(a) API documentation (`torch.add`)

Many PyTorch operations support NumPy's broadcasting semantics. See NumPy Broadcasting for details.

General Semantics: Two tensors are *broadcastable* if:

- Each tensor has at least one dimension.
- When iterating over dimensions from the trailing end, ❶ the sizes must either be equal, ❷ one is 1, or ❸ one is missing.

```
>>> x = torch.empty(5, 3, 4, 1)
>>> y = torch.empty( 3, 1, 1)
# x and y are broadcastable.
# 1st trailing dimension: both have size 1
# 2nd trailing dimension: y has size 1
# 3rd trailing dimension: x size == y size
# 4th trailing dimension: y dimension doesn't exist
```

(b) Broadcasting semantics (`torch.add`)

Figure 1: Documentation for PyTorch API and broadcasting semantics.

API constraints related to input parameters. A *rule* is a potential candidate of constraint for a combination of API parameters, e.g., the last dimensions of two tensor parameters should match. Second, CENTAUR learns input *constraints* for an API by mining valid inputs, identifying which of the candidate rules obtained in the previous phase hold. Third, CENTAUR uses an SMT solver (e.g., Z3 [8]) to generate models for abstract API inputs and a sampling strategy to generate concrete values from the abstract inputs.

CENTAUR needs to overcome several key challenges to be effective. First, the space of candidate rules is prohibitively large, which makes simple enumeration and validation of rules impractical. Second, it is difficult to obtain a sizeable number of inputs for each API (e.g., PyTorch has several hundreds of APIs) for learning input constraints. Ideally we need multiple and diverse valid inputs to obtain high quality constraints. Third, the constraints learned from a sample of inputs may not be generalizable to the entire API, and include spurious or incorrect rules. Fourth, typically SMT solvers only output one concrete solution for a given set of constraints, which limits the diversity of inputs and is also time-consuming if we want to generate multiple inputs.

CENTAUR addresses these challenges as follows. First, we design a novel grammar that represents first-order logic formulae and tensor-related properties (such as shape and data type). Then, we use a Large Language Model (LLM) to take this grammar specification and API documentation as inputs and generate candidate rules. Our intuition is that LLMs embody DL API-specific domain knowledge and can generate more relevant properties compared with a simple enumerator like grammarinator [18]. Further, by using a grammar, we can constrain the LLM outputs to syntactically valid rules. Second, we use LLMs to generate a small set of valid inputs for each API for constraint learning, and use custom mutators to diversify the inputs. Third, we develop a refinement strategy to systematically eliminate spurious or redundant rules. Finally, we show how we can use SMT solvers to generate abstract inputs that satisfy the learned constraints. The abstract inputs represent valid input sub-spaces that can be efficiently sampled from. We also develop optimizations to diversify the solver-generated abstract inputs.

Results. We conduct three experiments to evaluate CENTAUR. First, we evaluate the constraints that CENTAUR generates quantitatively and qualitatively. We find that CENTAUR generates more accurate and more precise constraints than prior approaches. Second, we show that CENTAUR outperforms ACETEST, PATHFINDER, and TITANFUZZ on both branch coverage and validity ratio (i.e., ratio of generated inputs that satisfy API-specific validity constraints) – two established metrics adopted in the literature to compare DL API-level fuzzers. For example, CENTAUR achieves a validity ratio of 98.2% for PYTORCH and of 99.24% for TENSORFLOW; these numbers are at least 28.2 and at most 77.7 percentage points higher than the validity ratio of the other comparison baselines. Third, CENTAUR finds 26 bugs in PYTORCH and TENSORFLOW. Of these, 18 bugs have been confirmed by developers at the time of writing this paper.

Contributions. We make the following contributions:

- ★ **Idea.** We propose CENTAUR, the first neurosymbolic API-level fuzzer for DL libraries that combines grammar-driven LLM-assisted constraint learning (the neural part) with SMT-based test input generation (the symbolic part);
- ★ **Evaluation.** We comprehensively evaluate CENTAUR using standard metrics from the literature and compare CENTAUR against SoTA techniques (e.g., ACETEST, PATHFINDER, and TITANFUZZ). Results indicate the superiority of CENTAUR and its ability to reveal deep bugs in DL libraries;
- ★ **Bugs.** Using CENTAUR, we found 26 previously unknown bugs in PYTORCH and TENSORFLOW, 18 of which have been confirmed by developers;
- ★ **Tool.** CENTAUR is publicly-available at https://github.com/ncsu-swat/centaur.

2 Motivating Example

We present a motivating example of how CENTAUR can be used to generate valid inputs for PyTorch's `torch.add` API. Figure 1a presents the documentation for PyTorch's `torch.add` API. The API takes two tensors as input–`input` and `other`–and adds them. The `other` tensor can be multiplied by `alpha`, if that parameter is provided. The resulting tensor can be optionally returned as `out` variable. The two input tensors should satisfy the *broadcasting* semantics [36], which is described by the documentation that Figure 1b shows. Intuitively, we say that two tensors are broadcastable if, starting from the trailing dimension, one of the three conditions is met for all dimensions (see ❶, ❷, and ❸). For example, tensors with shape $(5, 3, 4, 1)$ and $(3, 1, 1)$ are broadcastable as explained in comments. Inputs that do not satisfy this property lead to a runtime error, and notably, this semantics applies to *many* APIs (e.g., `torch.mul`, `torch.sub`, and `torch.div`). For example, if `torch.randn(3, 4)` and `torch.randn(2, 4)` are passed to `torch.mul`, it

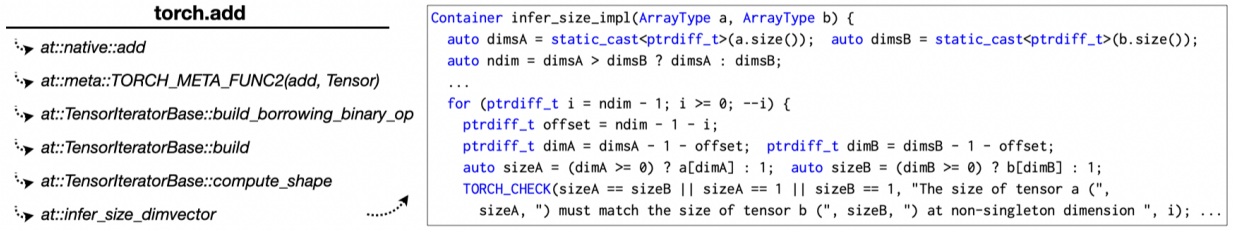


Figure 2: Native code for `torch.add` API.

Rule 15. (Check that the tensors either have the same shape, or are broadcastable)

$\{v\_1 : \texttt{tensor}, v\_2 : \texttt{tensor}\} \models \texttt{if ndim}(v\_1) = \texttt{ndim}(v\_2)$ then
$\forall\, i \in [0, \texttt{ndim}(v\_1)-1] :$ ❶ $\texttt{shape}(v\_1, i) = \texttt{shape}(v\_2, i)$
$\vee$ ❷ $\texttt{shape}(v\_1, i) = 1 \vee$ ❷ $\texttt{shape}(v\_2, i) = 1$
❸ $\texttt{else if ndim}(v\_1) > \texttt{ndim}(v\_2)$ then $\forall\, i \in [0, \texttt{ndim}(v\_2)-1] :$
$\texttt{shape}(v\_1, \texttt{ndim}(v\_1)-\texttt{ndim}(v\_2)+i) = \texttt{shape}(v\_2, i)$
$\vee\, \texttt{shape}(v\_1, \texttt{ndim}(v\_1)-\texttt{ndim}(v\_2)+i) = 1 \vee \texttt{shape}(v\_2, i) = 1$
❸ $\texttt{else} \forall\, i \in [0, \texttt{ndim}(v\_1)-1] :$
$\texttt{shape}(v\_2, \texttt{ndim}(v\_2)-\texttt{ndim}(v\_1)+i) = \texttt{shape}(v\_1, i)$
$\vee\, \texttt{shape}(v\_2, \texttt{ndim}(v\_2)-\texttt{ndim}(v\_1)+i) = 1 \vee \texttt{shape}(v\_1, i) = 1$

Rule 4. (Input and other tensors should have compatible sizes to avoid broadcasting errors ..)

$\{v\_1 : \texttt{tensor}, v\_2 : \texttt{tensor}\} \models \texttt{if ndim}(v\_1) < \texttt{ndim}(v\_2)$ then
$\forall\, i \in [0, \texttt{ndim}(v\_1)-1] :$
$(\texttt{shape}(v\_1, i) = \texttt{shape}(v\_2, i + (\texttt{ndim}(v\_2) - \texttt{ndim}(v\_1)))$
$\vee\, \texttt{shape}(v\_1, i) = 1 \vee \texttt{shape}(v\_2, i + (\texttt{ndim}(v\_2) - \texttt{ndim}(v\_1))) = 1)$ ..

Figure 3: Formal rules CENTAUR learned for broadcasting semantics.

raises an error "size of tensor a (3) must match the size of tensor b (2) at non-singleton dimension 0".

Testing this API is challenging for two reasons. First, the description of properties can be *scattered* throughout the documentation. Note that the API documentation briefly mentions broadcasting semantics; the definition of the actual property appears in a separate documentation. More often, the property may not be explicitly described in any documentation, i.e., properties can be *implicit*. For example, `torch.nn.ConvTranspose2d` has an implicit constraint that `output_padding` should be less than `stride`, which is not described in documentation. Second, properties are described in natural/informal language and often include multiple parameters of an API.

In prior work, Xie et al. [46] extracted constraints from documentation of DL APIs. However, they extracted constraints only from parameter descriptions in API documentation; their approach cannot extract properties that are scattered in the documentation or are implicit Also, their constraint extraction is based on syntactic patterns and each constraint only involves a *single* parameter at a time. For example, the following shows their constraints for `torch.add` API:

`input`: *tensor_t*=`torch.tensor` **`other`**: *tensor_t*=`torch.tensor`
`alpha`: *dtype*=int, *ndim*=0 **`out`**: *tensor_t*=`torch.tensor`

Each italicized keyword corresponds to a constraint: *tensor_t* constrains a tensor type, and *dtype* and *ndim* constrain the data type and number of dimensions of tensor. In total, they have five constraints for the API. As the example shows, they only capture simple tensor type constraints that are explicit in parameter descriptions. Also, their constraints are often imprecise (e.g., wrong *ndim* constraint for the number-typed **`alpha`** parameter) as based on syntactic patterns, and cannot express relational properties over multiple parameters, like broadcasting semantics.

To address the challenges of using only documentation, recent work proposed source code-based constraint extraction [22, 39]. However, their constraints are not as *precise*. ACETEST [39] identifies input validation path by traversing backward from error-handling function calls. For each identified path, ACETEST incrementally builds constraints based on pre-defined rules. However, as Figure 2 shows, error handling code can often be reached after a long sequence of function calls. Also, in this example, the `build` function has multiple function calls internally, and both the `compute_shape` and `infer_size_impl` functions have a loop. This leads to a potential path explosion, and ACETEST fails at identifying any constraints for `torch.abs` API. PATHFINDER [22] uses lightweight concolic testing; it inductively refines path conditions by input executions. However, for the same explosion reason, PATHFINDER fails to construct precise path conditions for broadcasting semantics.

CENTAUR addresses these challenges by leveraging the knowledge of LLMs. Due to the popularity of DL libraries, LLM has sufficient knowledge of valid API usages. We leverage it to generate formal constraints which include those from scattered documentations and implicit properties (§ 3.1). Also, CENTAUR defines a novel grammar (§ 3.1.1) that can express relational properties over an arbitrary number of parameters. Using this novel combination, CENTAUR effectively generates a rule for broadcasting semantics as shown in Rule 15 of Figure 3. As the example shows, the rule precisely captures all three conditions. It not only checks for the cases where two tensors have the same number of dimensions (❶ and ❷), but also check for the cases where one tensor has more number of dimensions (❸). Further, CENTAUR eliminates redundant LLM-generated rules to improve precision. Rule 4 in Figure 3 shows another generated rule that is essentially the same as Rule 15, but in a different representation. CENTAUR dynamically checks redundancy based on validity ratio and filters out such rules [29].

Upon refined rules, CENTAUR automatically converts the grammar-based definitions to Python code which encodes the rules as SMT formulae. CENTAUR uses the Python code to dynamically identify constraints (pairs of a rule and a subset of API parameters that the rule applies to) from the rules. CENTAUR tests whether the generated rules are satisfied by a set of automatically generated valid inputs. If so, CENTAUR saves the pair as a constraint (§ 3.2). Next, CENTAUR uses the learned constraints to generate diverse valid inputs by solving the SMT formulae (§ 3.3.1). Based on this approach, CENTAUR can generate valid inputs efficiently (8.9 milliseconds on average per input), and achieves near 100% validity ratio with no runtime errors.

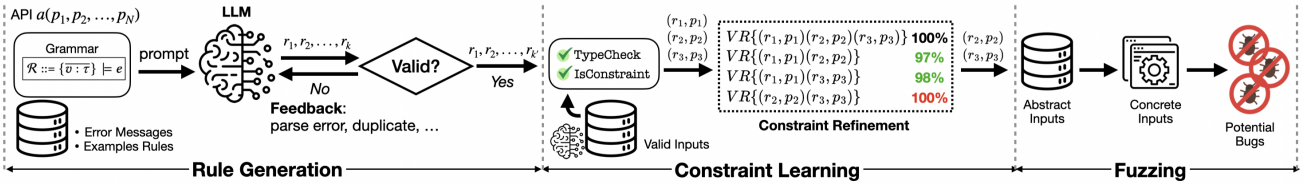


Figure 4: CENTAUR's workflow.

$$\mathcal{R} ::= \{\overline{v:\tau}\} \models e \quad \tau ::= \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \texttt{dtype} \mid \texttt{str} \mid \texttt{tensor} \mid \texttt{list}(\tau) \mid \texttt{tuple}(\tau) \mid \tau \uplus \tau$$

$$e ::= l \mid v_{\texttt{prim}} \mid f(v_{\texttt{tensor}}[, e]) \mid v_{\texttt{tuple}}[[e] \mid \texttt{.len}] \mid e_{\texttt{b}} \mid (e) \mid e \ominus e \mid \texttt{if } e \texttt{ then } e \texttt{ [else } e\texttt{]}$$

$$e_{\texttt{b}} ::= \texttt{true} \mid \texttt{false} \mid e \circledcirc e \mid e \wedge e \mid e \vee e \mid \forall v \in [e, e] : e(v) \mid \exists v \in [e, e] : e(v)$$

$f \in \{\texttt{ndim}, \texttt{shape}, \texttt{dtype\_}, \texttt{min}, \texttt{max}\}$,
$\ominus \in \{+, -, \times, /\}$, $\circledcirc \in \{=, \neq, >, <, \geq, \leq\}$,
$l \in \mathbb{R} \cup \texttt{Bool} \cup \texttt{String}$, $v \in \text{Var} = \{\texttt{i}, \texttt{v}_1, \texttt{v}_2, \ldots\}$

($v_{\texttt{prim}}$: primitive, $v_{\texttt{tuple}}$: tuple/list, $v_{\texttt{tensor}}$: tensor variables)

(a) Grammar of API constraints

$$\frac{\Gamma(v) \in \{\texttt{int}, \texttt{float}, \texttt{bool}, \texttt{str}, \texttt{dtype}\}}{\Gamma \vdash v : \Gamma(v)} \text{ [T-PrimAccess]}$$

$$\frac{\Gamma(v) = \tau_1 \uplus \tau_2, \quad \tau_1, \tau_2 \in \{\texttt{int}, \texttt{float}, \texttt{bool}, \texttt{str}, \texttt{dtype}\}}{\Gamma \vdash v : \tau_1 \uplus \tau_2} \text{ [T-UnionAccess]}$$

$$\frac{\Gamma(v) = \texttt{tuple}(\tau) \text{ or } \texttt{list}(\tau), \quad \Gamma(e) = \texttt{int}}{\Gamma \vdash v[[e]] : \tau} \text{ [T-TupleAccess]}$$

$$\frac{\Gamma(v) = \texttt{tuple}(\tau) \text{ or } \texttt{list}(\tau)}{\Gamma \vdash v.\texttt{len} : \texttt{int}} \text{ [T-TupleLen]}$$

$$\frac{\Gamma(v) = \texttt{tensor}, \quad f \in \{\texttt{ndim}, \texttt{dtype\_}, \texttt{min}, \texttt{max}\}}{\Gamma \vdash f(v) : \texttt{int}} \text{ [T-FuncCall-0]}$$

$$\frac{\Gamma(v) = \texttt{tensor}, \quad \Gamma(e) = \texttt{int}, \quad f = \texttt{shape}}{\Gamma \vdash f(v, e) : \texttt{int}} \text{ [T-FuncCall-1]}$$

(b) Typing rules for expressions and constraints

Figure 5: Grammar and typing rules of CENTAUR.

3 CENTAUR

CENTAUR is a neuro-symbolic API-level fuzzer for DL libraries that takes an API as input and returns a set of potential bug-revealing inputs for that API. CENTAUR uses 1) LLMs for grammar-based constraint learning (the neural part) and 2) SMT solvers for valid test input generation by solving learned constraints per API (the symbolic part).

Figure 4 shows the workflow of CENTAUR. The technique is structured in three stages: Rule Generation (§ 3.1), Constraint Learning (§ 3.2), and Fuzzing (§ 3.3). In the Rule Generation phase, CENTAUR uses an LLM to generate a set of candidate rules for a given API based on a novel grammar representing first-order logic formulae over API input parameters. In the Constraint Learning phase, CENTAUR learns likely constraints by testing the candidate rules on a set of valid API inputs. Each rule represents a first-order logic formula, whereas a constraint is an instantiation of the formula with (a subset of) API parameters. In the Fuzzing phase, CENTAUR executes the API on inputs obtained by solving the learned constraints using an SMT solver. CENTAUR monitors the executions for likely bugs (such as crashes or inconsistent outputs) and reports them to the user.

Algorithm 1 Grammar-based Rule Generation

Inputs: a: API, $\mathcal{G}$: grammar, $\mathcal{R}_{\text{ex}}$: example rules, $\mathcal{M}$: mutators to augment inputs, T: max num. of trials
Output: $\mathcal{R}_a$: ruleset for API a

procedure GENERATERULES
  $\mathcal{R}_a \leftarrow \emptyset$, $c \leftarrow 0$, $F \leftarrow \epsilon$ {ruleset, failure count, feedback}
  $\mathcal{E}_a \leftarrow$ COLLECTERRORS$(a, \mathcal{M})$
  while $c < T$ **and not** timed out **do**
    $r \leftarrow$ LLM(PROMPT_RULE_GEN, $\mathcal{G}, F,$ DOC$(a), \mathcal{E}_a, \mathcal{R}_{\text{ex}})$
    $error \leftarrow$ CHECK(r)
    if $error \neq \emptyset$ **then**
      $F \leftarrow error$; $c \leftarrow c + 1$
    else
      $\mathcal{R}_a \leftarrow \mathcal{R}_a \cup \{r\}$
  return $\mathcal{R}_a$

Inputs: a: API, $\mathcal{M}$: mutators to augment inputs
Output: $\mathcal{E}$: error messages for API a

procedure COLLECTERRORS
  $\mathcal{E} \leftarrow \emptyset$
  $V_a^{\text{valid}} \leftarrow$ LLM(PROMPT_INPUT_GEN, a, DOC(a))
  $V_a^{\text{mut}} \leftarrow$ MUTATE$(V_a^{\text{valid}}, \mathcal{M}, a)$
  $V_a^{\text{rand}} \leftarrow$ RANDOM_INVALID(a)
  for $v \in V_a^{\text{mut}} \cup V_a^{\text{rand}}$ **do**
    $outputs, error \leftarrow$ RUN(a, v)
    if $error \neq \emptyset$ **then**
      $\mathcal{E} \leftarrow \mathcal{E} \cup \{error\}$
  return $\mathcal{E}$

3.1 Rule Generation

In the rule generation phase, CENTAUR uses the domain knowledge of LLMs for valid DL library API usages. More precisely, CENTAUR prompts an LLM with a novel grammar to generate candidate rules for a given API. The grammar represents first-order logic formulae that capture constraints over API input parameters. Sections 3.1.1 and 3.1.2 describe, respectively, the grammar and the rule-generation algorithm.

3.1.1 Grammar for API Input Constraints. To constrain the responses of the LLM and extract information more precisely, CENTAUR provides a grammar of API constraints to the LLM. Figure 5a shows the grammar that CENTAUR uses. The grammar can express first-order logic formulae over multiple input parameters. It is designed to capture relational rules of diverse kinds. A rule $\mathcal{R}$ consists of a set of variable bindings $\{\overline{v:\tau}\}$ and an expression e over those variables. The variable bindings specify to which parameter types the rule applies. For example, the following rule specifies that the second parameter of the API (v_2), of integer type, should be a valid dimension of the first parameter (v_1), of tensor type: $\{v_1 : \texttt{tensor}, v_2 : \texttt{int}\} \models (-1 \times \texttt{ndim}(v_1) \leq v_2) \wedge (v_2 \leq \texttt{ndim}(v_1) - 1)$ Note that DL libraries like PyTorch allows negative indexing [38]. The grammar supports primitive types, compound types, and union

of types ($\tau \uplus \tau$). The union type is necessary to specify constraints for API inputs that admit multiple data types, e.g., `torch.clamp` allows both "number or tensor" types for its parameters [37]. The grammar can express properties for tuple and list types, including indexed access and length. For simplicity, $v_{\texttt{tuple}}$ denotes variables of such types. To support tensor-related properties, the grammar uses five functions: `ndim` denoting number of dimensions in tensor, `shape` denoting the shape of the tensor (e.g., `(3, 1, 1)`), `dtype_` denoting the library-specific data type (e.g., `float32` in PyTorch), `min`, and `max`. Finally, the grammar supports first-order logical operators (e.g., quantifiers), arithmetic operators, and conditionals.

3.1.2 Algorithm. Algorithm 1 shows how CENTAUR prompts an LLM to generate candidate rules for a given API. The algorithm takes as input an API a, a grammar $\mathcal{G}$ (§ 3.1.1), example rules in the grammar $\mathcal{R}_{\text{ex}}$, a set of mutators $\mathcal{M}$, and the number of trials T for rule generation. It returns candidate rules for the given API as output.

The utility function CollectErrors finds a set of *runtime* errors that can be observed by running the API on invalid inputs. The invalid inputs are generated by 1) querying LLM for valid inputs (V_a^{valid}) (Line 3) and then applying nine kinds of mutations (V_a^{mut}). For example, we convert to empty tensors or some of the values to zero or other int/float values (Line 4), and 2) generating random values (V_a^{rand}) (Line 5). The random generation was conducted with 30 seconds per API. In total, we curate a database of 2,642 and 2,822 error messages for 759 PyTorch APIs and 718 TensorFlow APIs, respectively. In addition, we use few-shot learning [3] to improve performance of the LLM [30]; we manually define example rules that span every parameter type and property at least once, and provide to the LLM as few-shot examples.

The function GenerateRules is responsible for generating rules for the input API a. It does so by iteratively querying an LLM with adjusted prompts until it reaches the maximum number of failures or it times out. We use 100 as the failure bound and a one-minute timeout in our evaluation. CENTAUR provides the grammar, API documentation, error messages, and example rules to the LLM. CENTAUR includes all available error messages of the API in a database, which contains an average of 2.54 error messages per API. For each iteration, CENTAUR adjusts the LLM prompt with one of five types of feedback (Line 6,8): 1) format error (e.g., no rule description), 2) redundant variable bindings, 3) duplicated rule, 4) parsing error, or 5) success. If successful, the rule is added to the set of candidate rules (Line 10). When given a one-minute time budget, the average number of iterations is 18.4. Notably, CENTAUR improves efficiency by asking the LLM to return multiple rules in a single iteration. This enables the generation of 93.37 rules on average (max: 213) per API within a minute.

Syntactic Validation. Our grammar uses a parser implemented in Lark [12]. So, CENTAUR precisely validates the *syntactic* correctness of LLM-generated rules. Also, CENTAUR checks for type errors where operators and expected variable types do not match. For example, a rule can define tensor-specific operation on primitive type variables. Figure 5b shows the typing rules that CENTAUR uses to discard the rules with type errors. We implement the type checking as an extension of the Lark Transformer [12].

Algorithm 2 Constraint Learning of CENTAUR

Inputs: $\mathcal{R}_a$: rules for API a, V_a^{valid}: valid inputs for a, T: num. of trials
Output: $\mathcal{I}_a$: constraints for a

1: **procedure** LearnConstraints
2: $\mathcal{I}_a \leftarrow \emptyset$
3: **for** $r \in \mathcal{R}(a)$ **do**
4: **if** TypeCheck$(r) = \bot$ **then**
5: **continue**
6: $k \leftarrow |\text{Vars}(r)|$
7: **for** $P \in \{(p_1, \ldots, p_k) | p_i \in \text{Params}(a)\}$ **do**
8: **if** IsConstraint$((r, P), V_a^{\text{valid}})$ **then**
9: $\mathcal{I}_a \leftarrow \mathcal{I}_a \cup \{(r, P)\}$ {See Def. 3.1}
10: $\mathcal{I}_a^{\text{final}} \leftarrow \emptyset$, $v_{\text{orig}} \leftarrow 0$, $I_a^r \leftarrow \emptyset$
11: **for** $t = 1$ **to** T **do**
12: $x \leftarrow$ Concretize(a, Z3GetModel$(\mathcal{I}_a)$)
13: **if** IsValid(a, x) **then**
14: $v_{\text{orig}} \leftarrow v_{\text{orig}} + 1$
15: **for** $(r, P) \in \mathcal{I}_a$ **do**
16: $\mathcal{I}_a^{\text{test}} \leftarrow \mathcal{I}_a \setminus \{(r, P)\} \setminus I_a^r$
17: $v_{\text{test}} \leftarrow 0$
18: **for** $t = 1$ **to** T **do**
19: $x \leftarrow$ Concretize(a, Z3GetModel$(\mathcal{I}_a^{\text{test}})$)
20: **if** IsValid(a, x) **then**
21: $v_{\text{test}} \leftarrow v_{\text{test}} + 1$
22: **if** $v_{\text{test}} < v_{\text{orig}}$ **then**
23: $\mathcal{I}_a^{\text{final}} \leftarrow \mathcal{I}_a^{\text{final}} \cup \{(r, P)\}$
24: **else**
25: $I_a^r \leftarrow I_a^r \cup \{(r, P)\}$
26: **return** $\mathcal{I}_a^{\text{final}}$

3.2 Constraint Learning

Even if a rule is grammatically correct with no type error, it should be checked 1) if the rule is an actual constraint; LLM can generate *semantically*-incorrect rules, and 2) which API parameters the rule applies to. We call a rule applied to specific API parameter(s), a *constraint*. Algorithm 2 shows how CENTAUR learns constraints $\mathcal{I}_a$ for the input API a (LearnConstraints). If all valid values of some parameter(s) satisfy a rule, it is likely to be a constraint [15]. So, we implement a translator which automatically converts the grammar-based rule definition to Python code. The code encodes constraints as Z3 formulae [8] and checks satisfiability. Then, CENTAUR checks the number of parameters (k) in the rule definition. For each ordered subset of k parameters, CENTAUR runs the satisfiability check by supplying valid values for the parameters. If the rule is satisfied with all the values, CENTAUR adds it as a constraint.

Definition 3.1 (Constraint). Let a be an API with parameter set $\text{PARAMS}(a)$, and let r be a rule with k variables. We say that (r, P) is an *constraint* for a, where $P = (p_1, \ldots, p_k)$ is an ordered list of parameters in $\text{PARAMS}(a)$, if the rule r holds for all valid inputs of a, when the variables in r are assigned to the corresponding values of $p_1, \ldots, p_k$: $\forall v \in V_a^{\text{valid}} \,.\, r(v[p_1], \ldots, v[p_k])$.

In the set of learned constraints, there can be redundant ones. For example, a constraint can imply another, and LLM can generate semantically-equivalent rules in different representations. Redundant constraints can over-constrain the input space and potentially limit the effectiveness of fuzzer. However, exhaustively checking

Algorithm 3 Abstract Input Generation

Inputs: $\mathcal{I}_a$: constraints for API a, N: num. of abstract inputs, p: sampling ratio
Output: C_a: corpus of abstract inputs for API a

1: **procedure** GenerateAbstractInputs:
2: $\mathcal{V} \leftarrow \emptyset$, $C_a \leftarrow \emptyset$
3: **while** not Timeout **do**
4: $Z \leftarrow$ GLOBAL_CONSTRAINTS
5: **for all** $(r, P) \in \mathcal{I}_a$ **do**
6: add $r(P)$ to Z
7: $S \leftarrow$ Sample(Variables(Z), p)
8: **for all** $v_i : \tau_i \in S$ **do**
9: **if** $\mathcal{V}[v_i] \neq \emptyset$ **then**
10: $val \leftarrow$ Sample($\mathcal{V}[v_i]$, 1)
11: $Z \leftarrow Z \cup \left\{\{v_i : \tau_i\} \models v_i \neq val\right\}$
12: $[l, u] \leftarrow$ Sample(Buckets(τ_i), 1)
13: $Z \leftarrow Z \cup \left\{\{v_i : \tau_i\} \models l \leq v_i \leq u\right\}$
14: **if** Z3Solver(Z) = SAT **then**
15: $\mu \leftarrow$ Z3GetModel(Z)
16: $x \leftarrow$ Concretize(a, μ)
17: **if** IsValid(a, x) **then**
18: **for all** $v \in$ Variables(μ) **do**
19: $\mathcal{V}[v] \leftarrow \mathcal{V}[v] \cup \{\mu[v]\}$
20: $C(a) \leftarrow C(a) \cup \{\mu\}$
21: **return** $C(a)$

all redundant rules can be prohibitively expensive (for instance, by checking for implication among all pairs of constraints). So, we implement a heuristic to iteratively eliminate redundant constraints.

Centaur iteratively refines the set $\mathcal{I}_a^{\text{final}}$. Centaur first generates inputs and computes the validity ratio (v_{orig}) (Line 10–14). As we explain in Sec. 3.3, the translated Python code not only enables satisfiability checks but also the generation of satisfiable values. Then, Centaur removes each constraint, one at a time, while keeping all the others ($\mathcal{I}_a^{\text{test}}$), and checks the validity ratio without the constraint (v_{test}) (Line 19). If the ratio decreases, it likely implies that the constraint is an essential one. So, Centaur keeps it in the final constraint set (Line 23), otherwise, discards it (Line 25). Finally, Centaur returns the refined set of constraints (Line 26). This heuristic-based refinement is fully automated and can be easily applied to any library and API.

3.3 Fuzzing

After Centaur learns constraints, the fuzzing phase begins. For efficiency, instead of concrete inputs directly, Centaur generates *abstract inputs* using Z3 and saves them in a corpus. In each fuzzing iteration, Centaur randomly selects an abstract input, generates a concrete input, and executes the target API with the concrete input.

3.3.1 Abstract Input Generation. Intuitively, an *abstract input* defines an input sub-space that satisfies all the constraints, from which multiple concrete inputs can be generated. For example, if an API has a single constraint $\{v_1 : \texttt{tensor}\} \models \texttt{ndim}(v_1) = 1$, the abstract input can generate any one-dimensional tensor. Algorithm 3 shows how Centaur generates a corpus of abstract inputs: $C(a)$ for the input API a (GenerateModels). User can set a timeout and sampling ratio p. For our experiments, we use an 1-hour timeout and sampling ratio of 0.3.

Centaur first creates a constraint set (Z) and adds initial global constraints, which are universal across all APIs. Examples include the maximum numbers of dimensions and their sizes (to suppress unrealistic tensors) and maximum size of tensors in bytes (to avoid memory issue). Also, we constrain the data types and string values to be chosen from predefined sets. Then, Centaur checks each constraint and adds the corresponding logical constraints. This process can be easily done due to how our Python code is designed: it reuses the logical constraints used during constraint learning phase. Centaur first obtains an abstract input μ that satisfies the accumulated constraints by SMT solving. Since an abstract input is an abstraction, it has a chance to be invalid when concretized; the concretization involves randomness (§ 3.3.2) or the constraints may be incomplete. So, Centaur generates a concrete input, and if the input is valid, adds μ to the corpus. This process is repeated until the counter n reaches the target number or the process reaches timeout. The abstract inputs are serialized into JSON format to be stored in and efficiently loaded from the corpus during fuzzing.

Definition 3.2 (Abstract Input). Let a be an API with constraint set $\mathcal{I}_a = \{(r_i, P_i)\}_{i=1}^{n}$, where each $r_i(P_i)$ denotes the Z3 formula obtained by applying rule r_i to an ordered parameter subset $P_i \subseteq \text{PARAMS}(a)$. A model μ is a total assignment to the symbolic variables referenced in the formulas $\{r_i(P_i)\}$, such that: $\forall (r_i, P_i) \in \mathcal{I}_a \,.\, r_i(P_i)(\mu)$.

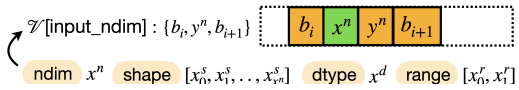


Figure 6: Example abstract input.

Optimizations. By default, Z3 returns the same solution (i.e., abstract input for Centaur) given the same constraints. To improve the input diversity, we develop two tactics. **(1) Blocking:** Centaur adds *blocking* constraints to prevent the same value in the subsequent abstract inputs. For every abstract input generated, Centaur records the value of every Z3 variable ($\mathcal{V}$). It samples a subset of the values (using a sampling ratio p), and adds constraints that block the corresponding variables from being assigned the same values in future iterations (Algorithm 3, Line 11). **(2) Bucketing:** Even with the blocking mechanism, Z3 often returns values closer to the limits specified in the constraint, which limits diversity of inputs (such as tensors). So, to further improve input diversity, we implement a bucketing technique in Centaur. For each data type (integers, floating point, tensors, and booleans), we partition the input space into buckets $[b_1, b_2, \ldots, b_k]$, and add additional constraints $\phi : b_i \leq v \leq b_{i+1}$ that limit the values of a variable v of that type. This approach allows Centaur to uniformly explore the input space. Lines 12 and 13 in Algorithm 3 show how Centaur implements bucketing.

Figure 6 shows an example abstract input for an `input` tensor. A tensor is abstracted based on dimension, shape, data type, and range properties. For the dimension property, a bucket of size four is selected (simplified). As three values are blocked from previous abstract inputs (colored in orange), x^n is chosen for the property.

3.3.2 Concrete Input Generation. Once the corpus of abstract inputs are constructed, Centaur begins fuzzing. Centaur randomly

Table 1: Number of APIs Centaur supports and number of APIs in common between Centaur and each baseline.

	Centaur	∩ ACETest	∩ Pathfinder	∩ TitanFuzz
Pytorch	512	124	192	350
TensorFlow	522	145	157	333

selects an abstract input from the corpus and constructs a concrete input using the values assigned to symbolic variables in the abstract input. These variables represent structural and semantic properties of input parameters and constrain the generation of valid inputs. For instance, Centaur uses the four tensor properties to construct a tensor of the specified dimension, shape, and type, while uniformly sampling its contents from the value range.

This is a repetitive, and notably, the only online phase of Centaur: the rule generation, constraint learning, and abstract input generation are all offline phases which need to be done only once per API. This brings efficiency benefits compared to Pathfinder [22] that computes and solves the path conditions to generate every input. Also, the inputs generated by Centaur have a validity ratio of near 100%. Centaur's separation of stages enables a systematic and balanced exploration of input space: abstract input generation expands the breadth, while concrete input generation explores depth. As a result, Centaur achieves similar or higher coverage compared to previous approaches in most cases.

4 Experimental Setup

Implementation Details. Our tool Centaur is implemented in Python 3.12. We use the Z3 SMT Solver [8], particularly the Python package (z3-solver version 4.14.1) that allows us to construct the constraints and to generate inputs by solving the constraints. We use LLMs to generate the candidate rules, a list of valid inputs for each API to use for validating the constraints and to generate the signatures of the APIs. We use Google Gemini 2.0 Flash [42] via the Google Cloud Platform (GCP) as the LLM in our experiments. For syntactic validation, of the generated rules, we use Lark version 1.2.2. We also use GPT-5, Claude Sonnet 4.5, Gemma 3 27B, and Qwen3-Coder-30B for further analysis (§ 6).

Baselines. For evaluation, we compare with two state-of-the-art constraint-based approaches: ACETest [39] and Pathfinder [40]. They both use constraints, but unlike Centaur which is based on API parameters, their constraints are path conditions for a specific input. We also include a technique that is not constraint-based, but uses LLMs to generate inputs: TitanFuzz [9]. We compare with these tools based on standard metrics such as coverage and validity ratio. Table 1 presents the number of APIs we can support along with the number of APIs common with each SoTA.

Experimental Setup. Our experiments target Pytorch and TensorFlow, the most popular DL libraries. We evaluated Centaur against three state-of-the-art baselines: ACETest [39], TitanFuzz [9], and Pathfinder [22]. While Centaur is compatible with the latest library versions (2.7.1 and 2.19.0), all coverage comparisons were performed on instrumented versions of Pytorch 2.2.0 and TensorFlow 2.16.1 to ensure a fair comparison, as porting the Pathfinder artifact is non-trivial. Table 1 reports the number of APIs we support and the overlap with each baseline. To ensure a fair and uniform evaluation, we use llvm-cov [27] to measure branch coverage for all tools. For ACETest and TitanFuzz, we used monkey patching [19] to isolate the coverage of only the target APIs. For validity ratio computation, we only consider an input valid if it does not raise an exception within the framework, determined using scripts from each tool's artifact.

For bug detection, we use oracles that are standard across the baselines [9, 11, 45]. These are 1) crash oracle: detecting when an input raises a signal (e.g. segmentation fault, aborted) and 2) differential oracle: detecting inconsistencies between outputs by running on different devices (CPU and GPU). Centaur logs any oracle violation, and after manually reviewing the logs, we submit bug reports that are determined by the authors to be true positives. During the review process, we further classify the inconsistencies into three categories: (i) *NaN* when only one platform generates a "Not a Number" value, (ii) *overflow* when only one platform encounters an arithmetic overflow, and (iii) *inconsistent* when there is numerical inconsistency in the outputs indicating potential bugs in implementation.

All experiments were conducted on an Ubuntu 22.04 server with dual AMD EPYC 9684X CPUs and 768GB of RAM. To ensure consistency, we run techniques in Docker containers. We allocate a 180-second time budget per API per tool. We use 30 and 0.3 for the number of trials and sampling ratio in Algorithms 2–3, respectively.

5 Evaluation

We pose the following research questions (**RQs**):

RQ1: How effective is the constraint learning of Centaur?

RQ2: How does Centaur compare with the SoTA in terms of validity rate and coverage?

RQ3: How effective is Centaur in detecting new bugs?

The first question evaluates the quality of constraints Centaur learns based on manually written ground truth. The second question compares Centaur against SoTA fuzzing tools considering the standard metrics in the field. The third question elaborates on the ability of Centaur to reveal deep bugs in DL APIs.

5.1 RQ1: Effectiveness of Constraint Learning

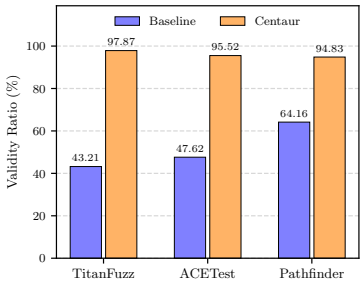
To evaluate the quality of Centaur-learned constraints, we select 15 APIs from PyTorch and TensorFlow each, and manually extract ground truth constraints. We refer to the API documentation and error messages (#E) to understand the input constraints for a certain API and build the ground truth. The APIs are randomly selected under the conditions that 1) at least 25% of random inputs are invalid (the average validity ratio is 13.9% across all APIs), and that 2) Centaur generates less than 15 constraints. The former is to ascertain that we can compare interesting constraints and the later is to balance manual effort. To validate the ground truth correctness, we write rules in our grammar and use the translator to generate Python code. This enables to check if 1) all valid inputs pass the rules, and 2) the ground truth enables valid input generation. We confirm that all the constraints pass with valid inputs, and improve the low validity ratio to 97% at minimum. For the same sets of APIs, Centaur generates 80 and 88 constraints, respectively. We conduct manual analysis to evaluate its recall and precision against the ground truth.

Table 2 shows how many of the ground truth constraints Centaur covers (I^G), and how many of Centaur-generated constraints

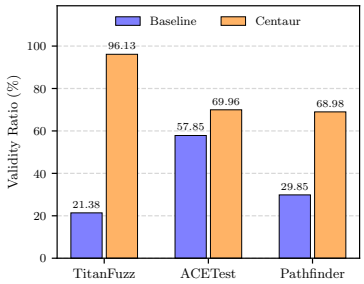
Table 2: Comparison of constraints against ground truth (#E: num. of error messages, V^R: validity ratio of random generation, I^G and I^S: invariants of ground truth and CENTAUR, M^G and M^S: avg. model generation time of ground truth and CENTAUR).

PyTorch							TensorFlow						
API	#E	V^R	I^G (Recall %)	M^G(s)	I^S (Precision %)	M^S(s)	API	#E	V^R	I^G (Recall %)	M^G(s)	I^S (Precision %)	M^S(s)
abs	5	11.3%	2/2 (100.0%)†	0.22	3/3 (100.0%)†	0.08	*experimental.numpy.-*	2	0.0%	1/1 (100.0%)†	0.03	4/5 (80.0%)	0.02
argmax	4	32.0%	3/3 (100.0%)†	0.05	6/6 (100.0%)†	0.03	*eye*	2	0.0%	3/3 (100.0%)†	0.05	8/8 (100.0%)†	0.61
channel_shuffle	4	2.0%	3/3 (100.0%)†	0.08	1/1 (100.0%)†	0.02	*image.ssim*	3	0.0%	6/6 (100.0%)†	0.10	9/10 (90.0%)	0.22
floor	4	4.1%	1/3 (33.3%)	0.18	1/1 (100.0%)†	0.02	*image.transpose*	4	46.0%	2/2 (100.0%)†	0.03	4/4 (100.0%)†	0.04
lcm	5	0.0%	2/2 (100.0%)†	0.07	9/9 (100.0%)†	0.04	*linalg.inv*	3	0.0%	3/3 (100.0%)†	0.02	13/13 (100.0%)†	0.02
linalg.solve_ex	6	0.0%	5/6 (83.3%)	0.64	9/9 (100.0%)†	0.70	*math.invert_permut-*	2	0.0%	3/3 (100.0%)†	0.18	6/6 (100.0%)†	0.05
narrow	4	0.0%	3/3 (100.0%)†	0.75	11/11 (100.0%)†	0.05	*math.sigmoid*	2	48.0%	1/1 (100.0%)†	0.02	4/4 (100.0%)†	0.03
nn.ReLU	3	66.0%	1/1 (100.0%)†	0.04	1/1 (100.0%)†	0.04	*nn.crelu*	3	0.0%	3/3 (100.0%)†	0.03	8/10 (80.0%)	0.02
nn.Softplus	2	23.0%	1/1 (100.0%)†	0.09	4/4 (100.0%)†	0.02	*nn.isotonic_regres-*	2	2.0%	1/1 (100.0%)†	0.05	3/3 (100.0%)†	0.02
nn.functional.alpha-	3	0.0%	2/2 (100.0%)†	0.03	7/7 (100.0%)†	0.03	*signal.fft*	5	24.0%	2/2 (100.0%)†	0.02	2/3 (66.7%)	0.02
*nn.init.sparse_*	5	2.0%	2/2 (100.0%)†	0.04	3/4 (75.0%)	0.50	*linalg.matrix_rank*	2	13.0%	1/1 (100.0%)†	0.38	1/1 (100.0%)†	0.38
linalg.householder	5	0.0%	3/4 (75.0%)	0.18	12/12 (100.0%)†	0.09	*sparse.eye*	2	35.4%	2/2 (100.0%)†	0.01	4/4 (100.0%)†	0.01
*nn.init.eye_*	2	17.0%	1/1 (100.0%)†	0.01	4/4 (100.0%)†	0.01	*make_tensor_proto*	3	6.0%	2/2 (100.0%)†	2.02	5/6 (83.3%)	0.21
view_as_real	2	24.0%	1/1 (100.0%)†	0.01	7/7 (100.0%)†	0.01	*unique*	2	10.0%	1/1 (100.0%)†	0.06	5/6 (83.3%)	0.01
special.expit	2	42.4%	1/1 (100.0%)†	0.15	1/1 (100.0%)†	0.15	*unique_with_counts*	2	10.0%	1/1 (100.0%)†	0.16	3/5 (60.0%)	0.14
Total	56	-	**31/35 (88.6%)**	2.55	**79/80 (98.8%)**	1.80	**Total**	39	-	**32/32 (100.0%)**	3.15	**79/88 (89.8%)**	1.80

* Validity ratio is 97% at minimum for both ground truth and CENTAUR in all APIs.



(a) PyTorch validity ratios



(b) TensorFlow validity ratios

Tool	Valid Inputs		Coverage	
	PyTorch	TensorFlow	PyTorch	TensorFlow
TITANFUZZ	123k	34k	10,012	4,427
CENTAUR	48,320k	56,248k	10,080	4,562
Δ	↑ 48,197k	↑ 56,214k	↑ 68	↑ 135
ACETEST	2,092k	10,556k	9,969	4,414
CENTAUR	34,822k	14,061k	10,129	4,404
Δ	↑ 32,730k	↑ 3,505k	↑ 160	↓ 10
PATHFINDER	5,658k	105k	3,090	1,905
CENTAUR	65,479k	13,367k	10,183	4,420
Δ	↑ 59,820k	↑ 13,262k	↑ 7,093	↑ 2,515

(c) Numbers of valid inputs and coverage

Figure 7: Comparison of validity ratios, valid inputs, and avg. branch coverage (three minutes per API).

are correct (I^S). Across PyTorch and TensorFlow, CENTAUR is accurate and precise: it has a recall of 100% for 27 APIs and a precision of 100% for 22 APIs (†). CENTAUR covers 31 (88.6%) and 32 (100.0%) of the ground truth constraints. For the four missed constraints, we find that CENTAUR either removes the constraints as they do not affect validity ratio, or is unable to generate the rule due to the string literals not supported in the grammar. Overall, we find that 98.8% (PYTORCH) and 89.8% (TENSORFLOW) of the generated constraints are correct, demonstrating that CENTAUR can generate high quality constraints for DL APIs.

5.2 Comparison with SoTA DL Library Fuzzers

5.2.1 Answering RQ2.1: How does CENTAUR compare with the SoTA in terms of validity ratio? Figures 7a and 7b show validity ratios for CENTAUR and the comparison baselines. The validity ratio is calculated as the ratio of valid inputs to the total number of inputs generated by each tool. The results show that CENTAUR achieves a significantly higher validity ratio compared to SoTA tools across both libraries. For example, CENTAUR achieves a validity ratio of 97.87% for PYTORCH, while TITANFUZZ only achieves 43.21%. Similarly, for TENSORFLOW, CENTAUR achieves a validity ratio of 96.13%, while TITANFUZZ only achieves 21.38%. The closest any approach gets to CENTAUR's validity ratio is PATHFINDER's 64.16% for PYTORCH and ACETEST's 57.85% for TENSORFLOW. It is no surprise that both techniques are constraint-based and they do much better than TITANFUZZ. This result further demonstrates the quality of CENTAUR's constraints compared to ACETEST and PATHFINDER.

In addition to the overall validity ratio, we also analyze the statistical significance of the differences between the validity ratio distributions of CENTAUR and each baseline. We use the same standard statistical tests as in the coverage comparison (§ 5.2.2). The p-values we obtain reject the null hypothesis that the distributions have no significant statistical difference. The p-values are 4.12e-153, 4.91e-37, and 2.24e-96 for TITANFUZZ, ACETEST, and PATHFINDER on PYTORCH, respectively. For TENSORFLOW, the p-values are 0, 2.02e-06, and 7.22e-07 for TITANFUZZ, ACETEST, and PATHFINDER, respectively. We also calculate the Cohen's d values to measure the effect size between the distributions. The effect sizes are very large for all comparisons in PYTORCH, and large to medium in TENSORFLOW. Cohen's d values of 3.90, 1.81, and 2.43 for PYTORCH, and 5.11, 0.53, and 0.55 for TENSORFLOW, for TITANFUZZ, ACETEST, and PATHFINDER, respectively.

The second and third columns in Figure 7c show the numbers of valid inputs generated across all APIs. Centaur generates significantly more valid inputs than the baselines for both libraries. For example, Centaur generates 65,479k and 56,248k inputs, whereas the baselines generate 5,658k and 34k inputs for PyTorch and TensorFlow, respectively. TitanFuzz makes an LLM request for every input

Table 3: Summary of reported bugs.

Library	Submitted	Confirmed	Rejected	Duplicates	Pending	Fixed
PyTorch	13	8	3	2	2	1
TensorFlow	13	10	1	0	2	0
Σ	26	18	4	2	4	1

Table 4: Categorization of the reported bugs.

Category	Bugs in PyTorch	Bugs in TensorFlow	Σ
Crash	1	6	7
Inconsistent	9	6	15
Overflow	2	1	3
NaN	1	0	1
Σ	13	13	26

```python
import torch as pt
input = pt.ones(1, 2, 3, 4)
output = pt.native_channel_shuffle(input, groups=3)
# Floating point exception (core dumped)
```

Listing 1: Bug in PyTorch [34].

generation, and the path conditions for ACETest and Pathfinder enable only one abstract input. In contrast, Centaur's constraints are based on API parameters, which enable the generation of multiple abstract inputs. In summary, Centaur's constraints are not only accurate—achieving high validity ratios—but also efficient in enabling input generation.

5.2.2 Answering RQ2.2: How does Centaur compare with the SoTA in terms of coverage? We collect branch coverage for each SoTA per API and compare corresponding distributions. Table 1 contains information on the number of APIs on which we ran this evaluation on Centaur and the number of APIs that intersect with each comparison baseline. The last two columns in Figure 7c present the average number of branches covered by each technique across all APIs using a time budget of 180 seconds. Comparing Centaur with Pathfinder, the differences between the distributions are statistically significant with p-values 0 and 6e-128 for PyTorch and TensorFlow respectively. The effect size is very large, with Cohen's d values of 17.05 and 4.76 for PyTorch and TensorFlow. Pathfinder only focuses on a limited component of the backend, which prevents it from covering branches in other parts of the DL libraries that Centaur can explore. However, for ACETest and TitanFuzz, we do not observe statistical significance between the distributions. The coverage distribution from Centaur have a small effect size when compared to TitanFuzz (0.14 and TitanFuzz 0.16 for PyTorch and TensorFlow), but no measurable differences when compared to ACETest. On average, Centaur demonstrates a slight advantage over ACETest and TitanFuzz, covering 150 and 203 more branches on average per API than ACETest and TitanFuzz, respectively.

5.3 RQ3: Detecting New Bugs in DL Libraries

Table 3 summarizes bugs we report in PyTorch and TensorFlow. So far, we found 26 bugs in total, of which 18 were confirmed. We relied on crash and differential oracles (i.e., the output difference between CPU and GPU) to find bugs. Results demonstrate the effectiveness of Centaur in finding bugs in DL libraries.

Centaur was successful in finding deep bugs in PyTorch and TensorFlow. The issue `#158154` [34] shows one of such cases in the

```python
import torch as pt, tensorflow as tf
a = tf.random.uniform((1,1,3,1), 0, 1, tf.float32)
depth_radius, alpha, beta = 5, 10, -1
pt_cpu = pt.nn.functional.local_response_norm(
  pt.tensor(a.numpy()), depth_radius, alpha, beta)
# tensor([[[[2.7409],[0.6304],[0.7823]]]])
pt_gpu = pt.nn.functional.local_response_norm(
  pt.tensor(a.numpy()).cuda(), depth_radius,
  alpha, beta)
# tensor([[[[2.7409],[0.6304],[0.7823]]]])
with tf.device('/cpu:0'):
  tf_cpu = tf.nn.local_response_normalization(
  a, depth_radius, alpha, beta)
# tf.Tensor([[[[9.8576],[1.3555],[1.8605]]]])
with tf.device('/gpu:0'):
  tf_gpu = tf.nn.local_response_normalization(
  a, depth_radius, alpha, beta)
# InvalidArgumentError: {{function_node...
# cuDNN requires beta >= 0.01, got: -1 [Op:LRN]
```

Listing 2: Example of a deep bug in TensorFlow [43].

API `torch.native_channel_shuffle` (Listing 1). The bug is caused by passing a value to the `groups` parameter that is greater than the size of the second dimension of the input tensor. The input appears valid according to the API documentation. Execution of the test causes a crash causes by a floating point exception. Our artifacts [28] includes the entire list of bugs we reported.

Issue `#97105` [43] (Listing 2) shows another example. The issue demonstrates a discrepancy where a negative `beta` value passed on CPU but triggered a cuDNN-related [5] `InvalidArgumentError` on TensorFlow GPU. Since PyTorch also uses the cuDNN backend yet yielded consistent, error-free results across devices, we concluded the fault lies in TensorFlow's implementation rather than the underlying library and reported the bug.

We note that some bug reports that developers rejected are not false positives. One such example is issue `#158208` [35] in PyTorch. We report a difference of $1.0737e+09$ between the CPU and GPU outputs. The developers rejected the bug report citing expected behavior due to floating point arithmetic differences between numerical libraries like LAPACK [2] and cuSOLVER [31].

Table 4 shows the categories of the bugs we report with Centaur. We find 7 bugs in PyTorch and TensorFlow that caused the program to crash by raising a signal (e.g. Floating Point Exception, Aborted etc.). These bugs are generally more critical since they can crash existing AI-based applications without any meaningful logs to help developers debug. We also find a bug that triggers `NaN` values in the output of the API. This is the only bug that was fixed by the developers at the time of writing. We also found 3 overflow bugs. These cases are typically considered less critical and can be handled by revising input validation checkers in code.

6 Discussion

Generalizability. To evaluate Centaur's generalizability across different LLMs, we used four additional LLMs beyond our default Gemini 2.0 Flash. We chose two popular closed source models (GPT-5 and Claude Sonnet 4.5) and two well-known open source models (Gemma 3 27B and Qwen3-Coder-30B) to cover a diverse range of LLM families. We randomly sampled 100 APIs from PyTorch and 100 APIs from TensorFlow and applied Centaur with each LLM to keep the experiment cost-effective. Table 5 shows the number of APIs successfully executed, average branch coverage, and average validity ratio. For comparison, numbers in parentheses show

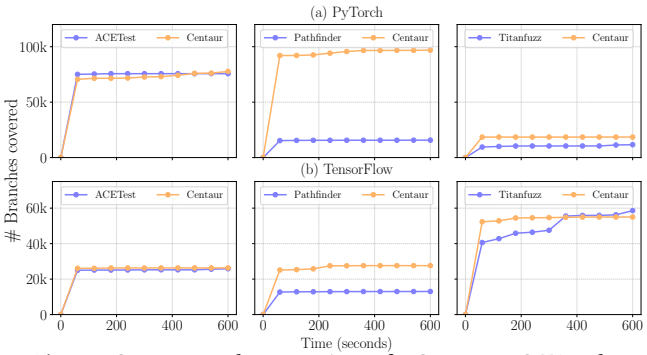


Figure 8: Coverage trend over 10 minutes for Centaur vs SoTA tools

Table 5: Performance of Centaur using different LLMs.

Library	Model	# APIs	Coverage	Validity Ratio
PyTorch	GPT-5	79 (88)	9928.40 (9910.34)	96.39% (95.62%)
	Claude Sonnet 4.5	68 (88)	9946.51 (9922.33)	96.58% (96.62%)
	Gemma 3 27B	70 (88)	9968.88 (10001.22)	96.70% (96.72%)
	Qwen3-Coder-30B	39 (88)	9950.06 (9965.36)	96.97% (96.97%)
TensorFlow	GPT-5	54 (93)	4600.85 (4509.09)	93.64% (92.94%)
	Claude Sonnet 4.5	43 (93)	4410.81 (4548.11)	99.93% (95.39%)
	Gemma 3 27B	72 (93)	4339.52 (4411.10)	92.52% (94.20%)
	Qwen3-Coder-30B	33 (93)	4496.65 (4482.26)	96.82% (96.79%)

Table 6: Results of Ablation Study.

Library	Ablation	# APIs	Coverage	Validity Ratio
PyTorch	w/o DocErr	74/100	9,947.65	94.57%
	w/o Feedback	77/100	9,954.31	94.79%
	w/o ExRules	81/100	9,978.32	97.50%
	w/o All	82/100	9,948.33	95.18%
	Centaur	88/100	9,983.99	96.48%
TensorFlow	w/o DocErr	43/100	4,379.47	92.21%
	w/o Feedback	55/100	4,298.61	95.40%
	w/o ExRules	45/100	4,510.29	97.76%
	w/o All	48/100	4,341.12	89.93%
	Centaur	93/100	4,462.66	94.24%

results for the default LLM, Gemini 2.0 Flash. The data shows that the Gemini 2.0 Flash outperformed the evaluated LLMs in terms of successful execution and that all LLMs but Qwen3-Coder-30B performed well considering the metrics. We find that while Centaur is compatible with multiple LLMs, the choice of LLM has a moderate impact on effectiveness.

Ablation Study. To evaluate necessity of each component of Centaur, we conduct an ablation study that systematically removes one component and evaluates impact on performance. We find three components of rule generation without which the technique still produces results: (1) feedback to the LLM on semantic validity of the rules (w/o Feedback), (2) supplemental information on constraints such as documentation, error messages (w/o DocErr), and (3) example rules to include with the prompt for rule generation (w/o ExRules). We also removed all three together to measure the effect of a technique where rule generation is performed without any assistance (w/o All). We randomly sampled 100 APIs each from PyTorch and TensorFlow to run the ablation configurations on these apis. Table 6 shows that each of these components is essential for Centaur considering the average reduction in measurements of "# APIs" and "Coverage".

Correctness of Rule Refinement. Centaur's refinement based on validity ratio may potentially discard true constraints. To check its correctness, we selected five widely used APIs: `torch.argmax`, `torch.nn.functional.alpha_dropout`, `torch.nn.ReLU`, `tf.math.sigmoid`, and `tf.image.transpose`, and manually validated the discarded constraints. In total, Centaur generated 141 constraints for all five APIs and then discarded 119 among them. Our manual analysis confirms that all 119 are correctly excluded and fall into three categories: (1) duplicates, (2) default constraints provided by Centaur (e.g., parameter types), and (3) incorrect constraints.

Time and Monetary Costs. Centaur uses LLM to generate API signatures, valid inputs, and candidate rules. To compute the time and monetary costs of using LLM, we run Centaur's default model (Gemini 2.0 Flash) for the 100 APIs used in ablation. For signature and input generation, Centaur takes 64.28 and 79.25 seconds, and uses 94k and 61k tokens on average per API for PyTorch and TensorFlow, respectively. This costs an average of $0.013 and $0.009 per API for PyTorch and TensorFlow. Similarly, For rule generation, time limit is configurable. We use one minute per API, and Centaur generates 93.37 rules on average during the time limit (§ 3.1.2). Centaur uses 2,872.5k and 2,234.1k tokens on average per API, which costs an average of $0.30 and $0.23 per API for PyTorch and TensorFlow, respectively. The results show that both time and monetary costs of using LLM are small. More importantly, the generation is a one-time offline process, and the LLM's outputs can be reused later for fuzzing.

Selection of time budget for fuzzing. We chose 180s as our time budget to balance the cost of running a total of 1,034 APIs across all baselines. Centaur can generate tens of millions of inputs in 180 seconds, allowing it to achieve high coverage in a short period. In contrast, in the same time frame, the other approaches generate 100x or 1000x fewer (and less diverse) inputs, and hence achieve lower coverage. For further validation, we calculated the cumulative coverage achieved by Centaur vs SoTA tools over a longer time budget of 10 minutes. Figure 8 shows the coverage trend with a snapshot taken every minute. The results show that Centaur outperforms all SoTA tools over both PyTorch and TensorFlow on completion except TitanFuzz on TensorFlow. On manual analysis, we found that TitanFuzz implicitly knows about hadrdware optimized kernel-specific memory format propagation rules for certain APIs in TensorFlow due to its usage of LLMs. For example, in oneDNN [23], if `dtype = int8`; then the "channel" dimension of a `NCHW` formatted tensor is expected to be aligned to 16 bytes [20]. Such optimization-specific rules are not captured by Centaur's inferred input invariants, leading to lower coverage.

7 Threats to Validity

External Validity. The generality of Centaur is limited by our selection of APIs, the expressiveness of our grammar, and the sample size used in the qualitative study. Considering API selection, we chose APIs from PyTorch and TensorFlow based on their popularity and usage in prior work [9, 22, 39]. It is also worth noting that our tool is extensible; by running the offline phases of Centaur with scripts provided in our replication package [28], users can easily apply Centaur to other APIs in these libraries. Considering our grammar, there may be rules that it cannot express, e.g., when a constraint is conditioned on an optional argument. One example is: "if an argument for the `axis` parameter is given, the value must be a valid dimension of the `input` tensor" concerning

the API `tf.reduce_sum`. We designed our grammar to express a variety of constructs and cover invariants of a large number of APIs. Also, our grammar is based on the Lark format [12] and can be easily extended. Considering the qualitative analysis of learned constraints, the small sample size of APIs and potential errors may have impacted precision and recall. To mitigate this threat, we carefully selected the APIs to cover different functionalities (e.g., `nn`, `math`, `linalg`, and `experimental` modules) and only considered the APIs with clear error messages to facilitate ground truth definition.

Internal Validity. Centaur's oracles may result in false positives. But, we conduct rigorous manual validation for every reported bug and create minimized test cases to reproduce bugs. Centaur is also prone to potential false negatives. However, our evaluation shows that Centaur generates both more valid and diverse inputs compared to other techniques. Also, Centaur has detected deep functional bugs, most of which are confirmed by developers. Using a small set of inputs can cause wrong candidate rules less effectively pruned. To mitigate this, we first obtain a small number of inputs using an LLM and apply nine types of carefully-designed mutations to enrich the set. In the end, we use 117 valid inputs on average per API. Lastly, the LLM's randomness may lead to different results for different runs. However, we believe that its impact is minimized because Centaur (1) uses an iterative approach to generate rules, and (2) applies the approach across all APIs of both libraries. Also, during implementation, we ran the rule generation multiple times and did not observe significant differences in quality.

8 Related Work

Model-level Testing of Deep Learning Libraries. CRADLE [33] was one of the first approaches for testing DL libraries using differential testing by comparing outputs from different execution backends (e.g., CPU vs. GPU) within the same library. More recently, Li et al. [24] used Large Language Models (LLMs) to translate APIs, enabling differential testing between different DL libraries. Similarly, many other approaches have been proposed to test DL libraries using differential testing at the model level [16, 17, 25, 26, 44]. Model-level testing is more limited because the number of APIs that can construct a model by being used together is a small subset of all APIs available in the DL libraries. Thus Centaur focuses on API-Level testing to cover a larger set of APIs.

API-level Testing of Deep Learning Libraries. API-level testing [6, 9, 11, 45] is another complementary approach to model-level testing. It works by generating inputs only for the given API and checking correctness of the outputs via differential testing and checking for crashes or exceptions. Because API-level specifications are incomplete, these tools employ various techniques to generate valid inputs. FreeFuzz [45] addresses this by mining open-source projects to infer API argument types and create valid inputs for mutation. DocTer [46] complements this by extracting API constraints (e.g., types, tensor shapes) directly from technical documentation. To improve testing efficiency, DeepREL [11] identifies APIs with similar behaviors to share test inputs across them, increasing coverage. More recent approaches leverage LLMs. In contrast to these approaches, IvySyn [6] leverages this with a type-aware, black-box mutator and can map low-level C++ crashes back to high-level Python API calls. Unlike Centaur these approaches do not rely on concrete specifications and are hence prone to generating many invalid inputs. While DocTer and DeepREL also use constraints, they are much simpler and do not capture the relational constraints between API parameters that Centaur can infer.

Pathfinder [22] is a gray-box fuzzer that uses program synthesis to approximate path conditions, guiding input generation toward new code paths. ACETest [39] extracts precise API input constraints by analyzing the library's internal validation code, tracing backward from error-handling functions. It then uses an SMT solver to generate valid inputs that satisfy these constraints. While Centaur also uses SMT solving, it does so by analyzing inputs and outputs; it does not require analyzing code and path constraints, which can be computationally expensive.

LLM-based Testing Approaches. Titanfuzz [9] uses LLMs to generate and mutate seed programs for fuzzing, while FuzzGPT [10] generates test code from bug reports and existing code snippets. Unlike these approaches, Centaur uses LLMs to generate API constraints (and some seed inputs for APIs) instead of inputs for fuzzing. This approach allows Centaur to develop an *input generator* that can generate many diverse and valid inputs beyond the LLM's capabilities.

Invariant Learning. Centaur is inspired by Daikon [13–15], a dynamic detection technique to infer likely invariants. Daikon infers invariants from a set of pre-defined patterns and by using a fixed set of execution traces. In contrast, Centaur uses a domain-specific grammar and validates candidate rules using dynamically generated inputs. Conceptually, this enables learning more precise, relevant, and complete constraints. Also, unlike Daikon, Centaur refines the constraints. DySy [7] leverages dynamic symbolic execution and infers invariants based on path conditions. But, Centaur's invariants result in higher validity ratio than those based on path conditions (Sec. 5.2.1). ISLearn [41] is another Daikon-style invariant inference approach for system inputs. However, in constrast to Centaur, ISLearn needs manual template definition and cannot express complex constraints of DL library APIs. Lastly, all the approaches do not use LLMs – an essential source of implicit invariants.

9 Conclusion

We presented Centaur – the first neurosymbolic technique that combines LLMs with SMT constraint solving to learn API input constraints and generate valid test inputs. We showed that Centaur generates high quality constraints for numerous PyTorch and TensorFlow APIs and achieves higher coverage and validity ratio compared to SoTA DL fuzzers. Finally, we showed that Centaur can find new bugs in PyTorch and TensorFlow libraries. We believe that such neurosymbolic techniques for constraint learning are quite promising and general. Such approaches can be further enhanced to find deeper and more diverse set of bugs not only in DL libraries but also other domains when input specifications are sparse. Our replication package is publicly available at https://github.com/ncsu-swat/centaur.

Acknowledgments

This work is partially supported by the United States National Science Foundation (NSF) under Grant Nos. CCF-2349961 and CCF-2319472. We thank Google for the Google Cloud Platform credits and Meta for the Meta LLM Evaluation Research Grant. We also thank the anonymous reviewers for their valuable feedback.

References

[1] 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.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. LAPACK Users' guide. *Society for Industrial and Applied Mathematics* (1999).

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. 1877–1901.

[4] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 135 (sep 2023), 31 pages. doi:10.1145/3587155

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[6] Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and Vasileios P. Kemerlis. 2023. IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2383–2400. https://www.usenix.org/conference/usenixsecurity23/presentation/christou

[7] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. 281–290. doi:10.1145/1368088.1368128

[8] 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*. 337–340.

[9] 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. doi:10.1145/3597926.3598067

[10] 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.

[11] 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. doi:10.1145/3540250.3549085

[12] Noam Elias. 2020–. Lark: Modern Parsing Toolkit for Python. https://github.com/lark-parser/lark. Accessed in July 2025.

[13] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*. 213–224. doi:10.1145/302405.302467

[14] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123. doi:10.1109/32.908957

[15] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (Dec. 2007), 35–45. doi:10.1016/j.scico.2007.01.015

[16] 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. doi:10.1145/3510003.3510092

[17] 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. doi:10.1145/3324884.3416571

[18] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*. 45–48.

[19] John Hunt and John Hunt. 2019. Monkey patching and attribute lookup. *A Beginners Guide to Python 3 Programming* (2019), 325–336.

[20] Intel. 2025. oneDNN Memory Formats. https://www.intel.com/content/www/us/en/docs/onednn/developer-guide-reference/2025-1/understanding-memory-formats.html. oneDNN Documentation.

[21] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935. doi:10.1016/j.jss.2021.110935

[22] Sehoon Kim, Yonghyeon Kim, Dahyeon Park, Yuseok Jeon, Jooyong Yi, and Mijung Kim. 2025. Lightweight Concolic Testing via Path-Condition Synthesis for Deep Learning Libraries. In *Proceedings of the 2025 International Conference on Software Engineering (ICSE)* (Ottawa, Canada). IEEE/ACM.

[23] Jianhui Li, Zhennan Qin, Yijie Mei, Jingze Cui, Yunfei Song, Ciyong Chen, Yifei Zhang, Longsheng Du, Xianhang Cheng, Baihui Jin, et al. 2024. onednn graph compiler: A hybrid approach for high-performance deep learning compilation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 460–470.

[24] Meiziniu Li, Dongze Li, Jianmeng Liu, Jialun Cao, Yongqiang Tian, and Shing-Chi Cheung. 2024. DLLens: Testing Deep Learning Libraries via LLM-aided Synthesis. *arXiv preprint arXiv:2406.07944* (2024).

[25] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23)*. ACM. doi:10.1145/3575693.3575707

[26] Jiawei Liu, Jinjun Peng, Yuyao Wang, and Lingming Zhang. 2023. NeuRI: Diversifying DNN Generation via Inductive Rule Inference. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 657–669. doi:10.1145/3611643.3616337

[27] LLVM Project. 2024. LLVM Coverage Mapping Format. https://llvm.org/docs/CommandGuide/llvm-cov.html.

[28] M M Abid Naziri, Shinhae Kim, Feiran (Alex) Qin, Marcelo d'Amorim, and Saikat Dutta. 2025. Centaur Artifact. https://github.com/ncsu-swat/centaur. GitHub repository, accessed in July 2025.

[29] M M Abid Naziri, Shinhae Kim, Feiran (Alex) Qin, Marcelo d'Amorim, and Saikat Dutta. 2025. Centaur: Candidate Rule (torch.add). https://github.com/ncsu-swat/centaur/blob/f9cb27c005dbf938ce884913163febf35fb67514/rules-torch/torch.add/rules-ebnf. GitHub repository, accessed in June 2025.

[30] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)* (Ottawa, Canada). To appear.

[31] NVIDIA Corporation. 2023. cuSOLVER Library. https://docs.nvidia.com/cuda/cusolver/. NVIDIA CUDA Toolkit Documentation.

[32] 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).

[33] 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. doi:10.1109/ICSE.2019.00107

[34] PyTorch Community. 2025. PyTorch Issue #158154: torch.native_channel_shuffle throws FPE when groups size of the second dimension of the input tensor. https://github.com/pytorch/pytorch/issues/158154. GitHub Issue.

[35] PyTorch Community. 2025. PyTorch Issue #158208: torch.det returns inconsistent results for complex tensor on CPU vs GPU. https://github.com/pytorch/pytorch/issues/158208. GitHub Issue.

[36] PyTorch Developers. 2024. Broadcasting semantics — PyTorch Documentation. https://pytorch.org/docs/stable/notes/broadcasting.html. Accessed in July 2025.

[37] PyTorch Developers. 2024. torch.clamp — PyTorch Documentation. https://pytorch.org/docs/stable/generated/torch.clamp.html. Accessed in July 2025.

[38] PyTorch Team. 2024. PyTorch Documentation. https://pytorch.org/docs/stable/tensors.html#torch.Tensor.size Accessed in July 2025.

[39] 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 690–702. doi:10.1145/3597926.3598088

[40] starlab-unist. 2025. PathFinder. https://github.com/starlab-unist/pathfinder-artifact. GitHub repository, accessed in July 2025.

[41] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 276–288. doi:10.1145/3540250.3549124

[42] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican,

et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[43] TensorFlow Community. 2025. TensorFlow Issue #97105: tf.nn.local_response_normalization returns incorrect output. https://github.com/tensorflow/tensorflow/issues/97105. GitHub Issue.

[44] 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. doi:10.1145/3368089.3409761

[45] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free Lunch for Testing: Fuzzing Deep-Learning Libraries from Open Source. In *Proceedings - International Conference on Software Engineering*, Vol. 2022-May. doi:10.1145/3510003.3510041

[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.