Leveraging Documentation to Test Deep Learning Library Functions

Danning Xie
Purdue University
West Lafayette, IN, USA
xie342@purdue.edu

Yitong Li
University of Waterloo
Waterloo, ON, Canada
ytnli95@gmail.com

Mijung Kim†
Ulsan National Institute of Science and Technology
Ulsan, South Korea
mijungk@unist.ac.kr

Hung Viet Pham
University of Waterloo
Waterloo, ON, Canada
hvpham@uwaterloo.ca

Lin Tan*
Purdue University
West Lafayette, IN, USA
lintan@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, IN, USA
xyzhang@cs.purdue.edu

Michael Godfrey
University of Waterloo
Waterloo, ON, Canada
migod@uwaterloo.ca

ABSTRACT

It is integral to test API functions of widely-used deep learning (DL) libraries. The effectiveness of such testing requires DL-specific input constraints of these API functions. Such constraints enable the generation of valid inputs, i.e., inputs that follow these DL-specific constraints, to explore deep to test the core functionality of API functions. Existing fuzzers have no knowledge of such constraints, and existing constraint-extraction techniques are ineffective for extracting DL-specific input constraints.

To fill this gap, we design and implement a document-guided fuzzing technique—*D2C*—for API functions of DL libraries. D2C leverages sequential pattern mining to generate rules for extracting DL-specific constraints from API documents and uses these constraints to guide the fuzzing to generate valid inputs automatically. D2C also generates inputs that violate these constraints to test the input validity checking code. In addition, D2C uses the constraints to generate boundary inputs to detect more bugs.

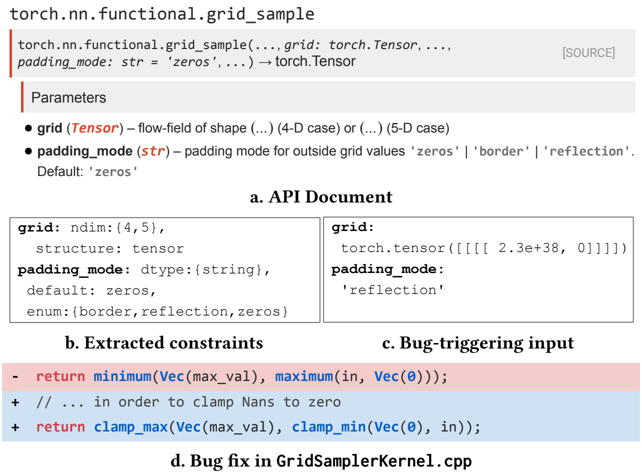
Our evaluation of three popular DL libraries (TensorFlow, PyTorch, and MXNet) shows that D2C's accuracy in extracting input constraints is 83.3–90.0%. D2C detects 121 bugs, while a baseline fuzzer without input constraints detects only 68 bugs. Most (89) of the 121 bugs are previously unknown, 54 of which have been fixed or confirmed by developers after we report them. In addition, D2C detects 38 inconsistencies within documents, including 28 that are fixed or confirmed after we report them.

KEYWORDS

text analytics, testing, fuzzing, deep learning

1 INTRODUCTION

Widely-used deep Learning (DL) libraries (e.g., TensorFlow [10] and PyTorch [48]) contain software bugs [32, 33, 51, 73, 74], which hurt not only the development but also the accuracy and speed of the DL models. Therefore, DL libraries need to be well-tested for better reliability.



torch.nn.functional.grid_sample

```
torch.nn.functional.grid_sample(..., grid: torch.Tensor, ...,
padding_mode: str = 'zeros', ...) → torch.Tensor
```

[SOURCE]

Parameters

- **grid** (*Tensor*) – flow-field of shape (...) (4-D case) or (...) (5-D case)
- **padding_mode** (*str*) – padding mode for outside grid values `'zeros'` | `'border'` | `'reflection'`. Default: `'zeros'`

a. API Document

```
grid: ndim:{4,5},
  structure: tensor
padding_mode: dtype:{string},
 default: zeros,
 enum:{border,reflection,zeros}
```

b. Extracted constraints

```
grid:
 torch.tensor([[[[ 2.3e+38, 0]]]])
padding_mode:
 'reflection'
```

c. Bug-triggering input

```
-  return minimum(Vec(max_val), maximum(in, Vec(0)));
+  // ... in order to clamp Nans to zero
+  return clamp_max(Vec(max_val), clamp_min(Vec(0), in));
```

d. Bug fix in GridSamplerKernel.cpp

Figure 1: PyTorch document helps our tool detect a bug that was fixed after we reported it to PyTorch developers.

A standard practice of testing library API functions is to pass input values to these functions and inspect unexpected behaviors triggered by the given inputs. Fuzzing is a scalable and practical testing technique for this purpose. Fuzzing provides random data as test inputs to a program and monitors if the inputs trigger any error in the program such as a crash [53].

Fuzzing a DL library's API functions is challenging because many of these API functions expect structured inputs that follow DL-specific constraints. If a fuzzer is (1) unaware of these constraints or (2) incapable of using these constraints to fuzz, it is practically impossible to generate *valid* inputs (i.e., inputs that follow these DL-specific constraints) to explore deep and test the core functionality of DL API functions.

†The work was completed when Mijung Kim was at Purdue University.
*Corresponding author.

Specifically, DL libraries' API functions require two types of constraints for their input arguments: (1) data structures and (2) properties of these data structures. First, DL libraries often require their input arguments to be specific *data structures* such as lists, tuples, and tensors to perform numerical computations. For example, the PyTorch API function `torch.nn.functional.grid_sample` has two parameters, `grid`, and `padding_mode` (other parameters are omitted for demonstration purpose). The former has to be a tensor, while the latter has to be a string, as dictated by its API document shown in Fig. 1a. A *tensor* is represented using an n-dimensional array, where *n* is a non-negative integer. Any input that cannot be interpreted as a tensor (e.g., a Python list) is rejected by the function's input validity check. Such invalid inputs exercise only the input validity checking code, failing to test the core functionality of the API function. To test `grid_sample`'s core functionality, a fuzzer needs to generate a tensor object for the `grid` parameter.

Second, API functions of DL libraries require their arguments to satisfy specific *properties* of data structures. Generating a correct data structure with incorrect properties often fails the input validity checking of the DL API functions. They often require two common properties of a data structure—*dtype* and *shape*. Property *dtype* specifies the data type of the data structure (e.g., `int32`, `float64`, and `String`). In Fig. 1a, the *dtype* of the parameter `padding_mode` should be `String`. Property *shape* specifies the length of each dimension of the data structure. For example, a *shape* of 3 × 4 matrix is a 2-dimensional tensor with the first dimension of 3 elements and the second dimension of 4 elements. In Fig. 1a, the parameter `grid` should be a tensor of either 4 dimensions or 5 dimensions. Any inputs that do not follow these *dtype* or *shape* requirements are rejected by the API function. Such inputs would exercise only the input validity checking code of the API function and fail to test the core functionality of the API function.

Effectively testing DL API functions requires such input constraints. Existing fuzzers such as AFL [4], HonggFuzz [56], and libFuzzer [5] have no knowledge of such input constraints, thus are very limited in testing DL API functions. **While existing techniques can extract constraints from code or software text (e.g., comments and documents), they are insufficient for extracting DL-specific constraints.** Specifically, while Pytype [7] infers data types from Python code, it cannot precisely infer types for DL libraries because Pytype cannot analyze across Python and C++ code. In addition, it cannot extract numerical constraints such as *shape* and *range*. Existing techniques that extract constraints from software text extract different types of constraints that are not DL-specific such as exceptions [13, 25, 47, 72], command-line options and file formats [69], lock- and call-relations [38, 57], interrupts [58], nullness [38, 47, 59, 78], resources [76], dependency relations for web service parameters [70], and inheritance relations [78]. Although some [13, 47, 72, 78] can extract constraints related to valid ranges, those are only a small portion of DL-specific constraints (Section 4.1). Techniques such as C2S [72] require pairs of Javadoc comments and formal JML [1] constraints as training data. For DL API functions, such formal constraints are unavailable, which requires a large amount of manual work. Many existing techniques [13, 25, 47, 58, 59, 69, 70, 76] use a handful of manually-designed rules to extract constraints. DL-specific constraints require 239 rules (Section 4.1), which would be difficult to manually design.

1.1 Our Approach

To fill this gap and tackle these challenges, we develop—*D2C*—the *first* technique that extracts constraints from API documentation to guide the fuzzy generation of test inputs for DL API functions. D2C uses the following techniques:

(1) DL-specific constraint extraction: Since API documents are written informally in a natural language, manually extracting constraints from a large number of API documents (e.g., TensorFlow v2.1.0 has 2,334 pages of API documents and 854,900 words) is tedious and inefficient. In addition, since these documents are constantly evolving, it is undesirable and error-prone to manually analyze them each time when the documents update which can be as frequent as every commit. To address these challenges, we leverage sequential pattern mining [26, 30] to automatically mine frequently occurring patterns in API documents and manually transform them into *rules* to extract constraints automatically. This semi-automatic process enables us to extract a large number of rules that are needed for extracting DL-specific constraints.

(2) DL-specific input fuzzing: D2C first analyzes free-form API documentation (e.g, Fig. 1a) to extract DL-specific input constraints (e.g., Fig. 1b). D2C uses these constraints to guide the fuzzer to generate valid inputs (e.g, Fig. 1c) satisfying DL-specific constraints such as tensor-related constraints which the existing techniques cannot. D2C then evaluates the generated inputs by checking if the API runs successfully without failures, e.g., crashes. If a failure occurs with a valid input, the generated test has manifested a bug in the implementation of the API's core functionality.

Fig. 1d shows a previously unknown bug detected by D2C in PyTorch, and its patch that the PyTorch developers committed after we reported the bug to them. According to the document in Fig. 1a, the shape of the parameter `grid` is either 4-D or 5-D. Following the extracted constraints in Fig. 1b, D2C automatically generates the bug-triggering input in Fig. 1c. The four pairs of square brackets indicate that the parameter `grid` is four-dimensional (4-D). The two elements in `grid`, i.e., "`(2.3e+38, 0)`", are the indices to specify a pixel in a given image (the image is another parameter of `grid_sample` not shown for simplicity). The large index value (`2.3e+38`) causes the computation to produce `NaN` (not a number), which leads to invalid array access, resulting in a segmentation fault. This bug is only triggered in the `padding_mode = "reflection"` mode with a large index value in the `grid`'s tensor. A fuzzing technique that randomly generates inputs for this API fails to generate any input to trigger this bug.

(3) Constraint-guided invalid-input fuzzing: In addition to valid inputs, D2C generates invalid inputs that violate the constraints to test the input validity checking code of API functions. Despite invalid inputs, DL API functions should not crash, because anyone can invoke API functions, and may make mistakes due to carelessness, ignorance, or malice. Thus, one expects API functions to report the invalid input (e.g., by throwing an exception or printing an error message) instead of crashing. This point is well confirmed by an API developer after we reported a crash bug detected by D2C "*A segmentation fault is never OK and we should fix it with high priority*". Thus, we also consider that serious failures such as crashes caused by invalid inputs indicate bugs. Different from random fuzzing, D2C generates invalid inputs by violating the constraints of one parameter (invalid parameter) at a time (i.e., other parameters follow their constraints). This way, D2C creates inputs that exercise different paths of the input validity checking code. Such invalid-input fuzzing is impossible without the constraints.

(4) Constraint-guided boundary-based fuzzing: Boundary input values (e.g., 0 and None) tend to cause bugs due to off-by-one errors and lack checking for boundary values [20, 29]. Thanks to the extracted constraints, D2C is capable of mutating the parameter to boundary values of constraints to trigger more bugs. D2C generates both valid and invalid boundary values. For example, D2C extracts two constraints *dtype* of `int` and *range* of `[0,inf)` for variable `num_columns` from `tf.sparse.eye`'s API documents. D2C then generates boundary values following or violating the constraints, such as `-MaxInt`, -1, 0, and `MaxInt`, to test if the target API function handles such boundary input values correctly.

(5) Documentation-bug detection: Since incorrect API documentation provides false information about APIs, which often misleads developers to introduce bugs in code [57], it is important to detect bugs in API documents as well. Different from prior work [57, 59] that detects inconsistencies between documents/comments and code, D2C detects inconsistencies within documents. For example, if the shape of a parameter is dependent on another parameter, which is missing in the API document, the document is inconsistent, indicating a *documentation bug*.

1.2 Contributions

In this paper, we make the following contributions:

- A rule-discovery technique that uses sequential pattern mining to identify patterns from API documentation to help create 239 rules for constraint extraction.
- A document-analysis technique that extracts 19,684 constraints automatically from API documentation with the focus on four categories of input properties in DL APIs: *structure*, *dtype*, *shape*, and *valid values* for 2,476 API functions across the three widely-used DL libraries, TensorFlow, PyTorch, and MXNet [17]. The constraint extraction accuracy is 83.3–90.0%.
- A simple yet effective fuzz-testing technique that is capable of generating DL-specific inputs (i.e., multi-dimensional array inputs, also referred to as *tensors*) and other general inputs, guided by the four categories of constraints. The fuzzer generates both valid and invalid inputs, and is designed to generate boundary inputs to detect more bugs.
- A tool *D2C* that combines the techniques above, and detects 121 bugs in 202 APIs from the three libraries, while a baseline fuzzer that has no knowledge of constraints detects 68 bugs only. Among the 121 bugs, 89 are previously unknown bugs, 54 of which have been fixed (41) or confirmed (13) by the developers after we report them. Out of the 41 bugs that have been fixed after we reported them, 21 are fixed in C++ code, 13 in Python code, 5 in both C++ and Python code, while the rest is unknown. In addition, D2C detects 38 documentation bugs, 28 of which have been fixed (14) or confirmed (14) after we report them.

While it is possible to extend D2C to test general computational libraries, e.g., scikit-learn [49], without major changes, we focus on DL libraries in this paper due to the importance of DL libraries and the lack of available testing techniques for them.

2 APPROACH

2.1 Challenges and Overview

Fig. 2 shows the overview of D2C using an example of the PyTorch API `grid_sample` (document in Fig. 1a). The *constraint extraction phase* takes API documents and extracts constraints for each input parameter. The *fuzzing phase* takes these constraints, generates test inputs either conforming or violating the constraints, and evaluates the generated inputs to return bug-triggering inputs.

Each component comes with its own challenges. A major challenge of the constraint extraction phase is **analyzing free-form API documentation** written in a natural language [13, 47, 59, 69, 76]. We leverage *sequential pattern mining* [26, 30] (SPM) to collect frequently occurring patterns and manually transform them into constraint extraction rules (or *rules* for short). D2C uses these rules to automatically extract constraints from API documents. Our semi-automated process reduces the manual effort required to discover useful patterns in a large number of documents. The constraint extraction works together with the rules, matches certain keywords relevant to the properties of a parameter such as *dtype* and *shape*, and outputs the parameter's constraints.

One challenge for the fuzzing phase is **satisfying constraint dependencies**. Generating inputs that follow such constraints requires D2C to determine the parameters' generation order correctly to ensure that the early-generated parameters do not violate the constraint dependency with later parameters, so that all constraints are satisfied. For example, if parameters `a` and `b` should have the same shape as parameter `c` but they are generated before `c`. Therefore, `a` and `b` may have different shapes because `c` has not been generated yet. Therefore the shape dependency cannot be satisfied. We address this by fuzzing parameters using the topological order of the dependency graph. This graph is a directed acyclic graph that represents the parameter dependencies, where a node represents a parameter and a directed edge represents a dependency.

The constraint extraction phase consists of two components, *Pattern miner* and *Constraint extractor*. The pattern miner automatically finds frequent subsequences from the sentences in the API documents. These subsequences are then manually verified and transformed into rules (up to 15 hours per project). The constraint extractor applies these rules to the API documents to automatically extracts a set of constraints for input parameters.

For example, in PyTorch documents (e.g., Fig. 1a), subsequences "*shape(x, ...)*" often specify the *shape* of a corresponding parameter. Using these frequent subsequences, we create a rule that extracts the *shape* of parameters using regular expressions (as shown in Fig. 1b). For `grid_sample`, the extracted *shape* constraint indicates that `grid` must be a 4-D or 5-D tensor.

During the fuzzing phase, for each DL API function, D2C takes the extracted constraints and iteratively performs two steps: generating an input (*Input generator*) and evaluating that input (*Test case evaluator*). By either following or violating the *extracted* constraints, the input generator generates *Conforming inputs* (CIs) or *Violating inputs* (VIs), respectively. Since the extracted constraints may be incorrect or incomplete, the CIs are not always valid and

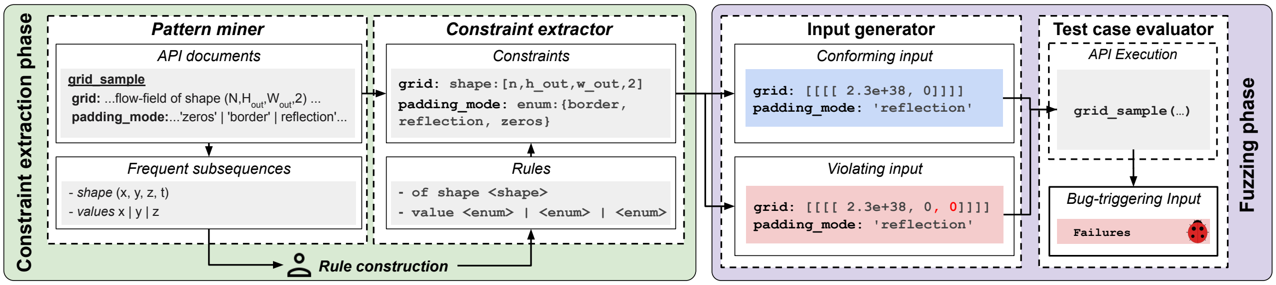

Constraint extraction phase
Pattern miner
API documents
grid_sample
grid: ...flow-field of shape (N,H_out,W_out,2) ...
padding_mode:...'zeros' | 'border' | reflection'...
Frequent subsequences
- shape (x, y, z, t)
- values x | y | z
Constraint extractor
Constraints
grid: shape:[n,h_out,w_out,2]
padding_mode: enum:{border, reflection, zeros}
Rules
- of shape <shape>
- value <enum> | <enum> | <enum>
Rule construction
Input generator
Conforming input
grid: [[[[2.3e+38, 0]]]]
padding_mode: 'reflection'
Violating input
grid: [[[[2.3e+38, 0, 0]]]]
padding_mode: 'reflection'
Test case evaluator
API Execution
grid_sample(...)
Bug-triggering Input
Failures
Fuzzing phase

Figure 2: Overview of D2C

the VIs are not always invalid. In this paper, we consider an input a *valid input* or *invalid input* if it follows or violates, respectively, the ground-truth constraints, as opposed to extracted constraints. The test case evaluator uses the generated input to invoke the API function and returns the bug-triggering inputs that cause severe failures (e.g., segmentation faults).

For the function `grid_sample` (whose API document is in Fig. 1a), the extracted constraints (Fig. 1b) indicate that `grid` must be a 4-D or 5-D tensor and parameter `padding_mode` expects one of three options: `"border"`,`"reflection"`,`"zeros"`. By following these constraints, the input generator creates an input (Fig. 1c and Section 1). The test case evaluator executes `grid_sample` with this input and detects a segmentation fault. After we report this previously unknown PyTorch bug, it has been fixed.

2.2 Pattern miner

Since API documentation is presented informally in natural language, manually extracting rules from the documents is expensive. For example, there are 2,334 pages of API documents and 854,900 words in TensorFlow v2.1.0. It is a daunting and tedious task for developers to manually examine such a large set of API documents to identify constraints. Following prior work [13, 59], we use rules to match potential API documents and extract relevant constraints. Different from the prior work where the authors designed rules manually, we semi-automate this process by using SPM to identify recurring patterns in API documents. In addition, our rules are designed for analyzing DL API documents, which have not been explored by existing work. Specifically, D2C automatically applies SPM on sentences of API documents to find *frequent subsequences*, which provide insight and templates that save manual efforts compared to constructing rules from scratch.

API document collection and preprocessing: Before applying SPM, D2C collects API documents from DL libraries' websites. We focus on two sources in API documents for pattern mining: parameter descriptions and parameter names. Parameter descriptions in API documents often specify useful constraints such as the parameter type. Parameter names often imply additional constraints (e.g., parameter `name` implies its *dtype* to be `String`).

To extract information from API documents, we first parse the HTML documents to obtain function signatures and parameter descriptions using an HTML parsing tool [2]. Since sentences are a natural unit of organizing constraints, we split the description into sentences using regular expressions. Parameter names are extracted from the API signatures. We tokenize the sentences and parameter names into words using white spaces and "_" respectively. These sequences of words are fed into the SPM process as lists of items.

To improve the effectiveness of SPM, we normalize data types (e.g., `int32` and `int64`) as `D_TYPE` and structure types (e.g., `array`, `list`, and `tuple`) as `D_STRUCTURE`. This way, references to different data and structure types could be grouped into the same frequent sequential pattern, reducing the number of frequent subsequences for less manual inspection effort. Fortunately, each library provides a list of supported data types. There are 23, 13, and 13 data types listed for TensorFlow, PyTorch, and MXNet, respectively. Since the lists use the exact data types (e.g., `np.int32`), we add informal variations (e.g., "int32", "integer", and "ints") to match the format of API documents. We also add common *dtypes* that are missing, e.g., `String`. In total, we use 71, 52, and 40 type phrases for TensorFlow, PyTorch, and MXNet, respectively. We manually collect 11 structure types that are shared by all three libraries.

Sequential pattern mining: It is hard to manually discover rules because API documents use many different ways to specify the same content. For example, one common way to specify the *dtype* of a parameter is "*must be one of the following types:* $\texttt{D\_TYPE}_1$, ... $\texttt{D\_TYPE}_n$", which occurs 175 times in all API documents. In addition, the sentence *"If set to `true`,..."* implies that the *dtype* of the parameter is `boolean` (found by the pattern *"set true/false"* which occurs 110 times). In fact, we discover 126 rules to extract *dtype* constraints alone which shows that it would be difficult, tedious, and error-prone for developers to manually design extraction rules.

Therefore, D2C uses SPM to automatically find frequent subsequences from the sentences in the API documents. SPM discovers frequent subsequences within a sequence dataset [40]. For example, an input sequence dataset D can consist of a set of sequences: $D = \{< \underline{a}, b, c, \underline{d} >, < \underline{a}, c, \underline{d}, b >, < \underline{a}, e, c, \underline{d}, f >, < b, \underline{a}, e, \underline{d} >\}$. If subsequence $< a, d >$ appears four times in the dataset, its `support` and `length` are 4 and 2, respectively. From dataset D, an SPM algorithm efficiently finds all subsequences that occur at least `min_support` times and have a length of at least `min_len` [11], while a naive approach that counts the frequencies of all possible patterns does not scale. D2C uses PrefixSpan SPM [30], for its efficient processing. We select the same `min_support` and `min_length` thresholds for all three evaluated DL libraries to demonstrate the generality.

2.3 Rule construction

We manually categorize and convert the frequent subsequences, mined by the pattern miner, into four categories (i.e., *structure*, *dtype*, *shape*, and *valid value*) of rules. We focus on these four

Table 1: Rule examples and the extracted constraints from TensorFlow, PyTorch, and MXNet

Category	No.	Examples of extraction rules	Examples of sentences from API documents	Examples of extracted constraints
structure	1	<*structure*> (list/tuple/...) of <*dtype*>	n: A list of integer.	n: structure={list(int)}
	2	<*structure*> (dict/dictionary) of <$dtype_1$> to <$dtype_2$>	features: Dict of string to `Tensor`.	features: structure={dict(string:tensor)}
dtype	3	of/with type <*dtype*>	audio: A `Tensor` of type ` float32`.	input: dtype={float32},structure={tensor}
	4	<*ndim*>-d/dimensional <*dtype*> tensor	mask: K-D boolean tensor.	mask: dtype={boolean},ndim={k},structure={tensor}
	5	must have the same type/dtype as <*dependency*>	imag: Must have the same type as `real`.	imag: dtype={&real.dtype}
shape	6	<*ndim*>-d/dimension tensor	logits: 2-D Tensor..	logits: ndim={2},structure={tensor}
	7	with/of the same shape as <*dependency*>,	target: A tensor with the same shape as `output`.	target: shape={&output.shape}
	8	of/with shape <*shape*>	weights: ...of shape `[num_classes, dim]`.	weights: shape={[&num_classes,dim]}
	9	tensor of length <*shape*>	rates: A 1-D Tensor of length 4.	rates: shape={[4]}
valid value	10	only <$value_1$>, ... <$value_n$> are supported	data_format: A string. Only `"NWC"` and `"NCW"` are supported.	data_format: dtype={string},enum={"NWC","NCW"}
	11	non-negative <*dtype*>	num_columns: ... non-negative integer...	num_columns:range={[0,inf)},dtype:{int}
	12	must be in the range <*valid range*>.	axis: Must be in the range `[-rank(input), rank(input))`	axis: range={[-&input.ndim,&input.ndim)}

categories because they represent the most common properties of input parameters of API functions in major DL libraries. With these four categories, D2C is able to extract constraints from almost all (96.8%) of the collected API functions. The four categories are:

- *structure*: the type of data structure that stores a collection of values for the input parameter, such as list, tuple, n-dimensional array (i.e., tensor), etc.
- *dtype*: the data type, such as int, float, boolean, String, etc., of the parameter or the elements of *structure*.
- *shape*: the shape or number of dimensions of the parameter. For example, in row 8 of Table 1, weights is of shape [num_classes, dim] (i.e., it is a 2-D array where the sizes of its first and second dimensions are num_classes and dim, respectively).
- *valid value*: a set of valid values (e.g., parameter padding can only be one of "zeros", "border", and "reflection") or the valid range of a numerical parameter (e.g., a float between 0 and 1).

Table 1 shows examples of rules (col. "Examples of extraction rules") and examples of matched sentences (col. "Examples of sentences from API documents"). For example, the first rule "<*structure*> (list/tuple/...) of <*dtype*>" is used to extract the *structure* constraint of a parameter, which could be applied to "n: A list of integer".

We make several reasonable assumptions when constructing the rules. For example, a parameter is assumed to be a 0-dimensional float between 0 to 1 inclusive if the document states it is a *"probability of ..."*. In addition, for optional parameters, we design rules to infer *dtype* from the parameter's default value.

The frequent subsequences extracted from parameter names represent patterns in parameter naming. We manually investigate these patterns and add *dtype* constraints using our knowledge of DL libraries. For example, a frequent subsequence "*shape*" in a parameter name indicates that the parameter represents the shape of a tensor. Since the shape of a tensor is a 1-D array with each element specifying the size of a dimension, parameters with names containing "*shape*" should be a 1-D array of non-negative integers.

2.4 Constraint extractor

The constraint extractor automatically finds matching texts in the parameter descriptions and names, and extracts the relevant constraints according to the rules. In Table 1, for each rule (col. *Examples of extraction rules*), we list one example of the sentences (col. *Examples of sentences from API documents*) that can be matched. The extractor automatically generates the corresponding constraints (col. *Examples of extracted constraints*).

Constraint dependencies: The description of one parameter often refers to the *dtype*, *shape*, and *valid value* of another parameter of the same API function. In such cases, D2C extracts constraints that involve *dependencies* among input parameters. Since most dependencies are direct (i.e., the property of one parameter is the *same* as another parameter), we do not consider less common or implied dependencies (e.g., *"compatible with"* or *"broadcastable to"*). These dependencies are useful not only for generating valid inputs but also for determining the parameters' generation order.

For *dtype* dependencies, D2C uses extraction rules such as "*must have the same dtype as* <other_parameter>" to extract the *dtype* dependencies among parameters. For example, row 5 in Table 1 shows the constraint dependency dtype:{&real.dtype} (& symbol in front of the <other_parameter> indicates that it is a dependency) where imag's *dtype* must be the same as real.

Parameters can also have *shape* dependencies, an example is row 7 of Table 1 (i.e., the parameter target has the same shape as parameter output). The *shape* dependency could also involve sizes of the parameter's dimensions. Example in row 8 indicates that 2-D array weights should have shape [num_classes,dim]. The size of its first dimension is specified as the value of another parameter (num_classes) in the same API function. In addition, The last dimension of weights and another parameter input (with shape [batch_size,dim]), should have the same size.

Valid value dependencies such as *range* dependencies arise when a parameter's elements should be in a certain range using another parameter's constraints. For example, row 12 in Table 1 shows that the value of the parameter axis should be within a range determined by the rank (i.e., number of dimensions) of another parameter input.

Documentation bug detection: As discussed in the Introduction, D2C detects inconsistencies within documents as documentation bugs (e.g., the parameter names in the description do not match the names specified in the function signatures). Since D2C is capable of analyzing constraint dependencies, D2C also detects dependency inconsistencies in documents. For example, in the document of tf.keras.backend.moving_average_update, the description for parameter value is *" ...with the same shape as `variable`,..."*, but the parameter variable is not documented. This documentation bug of unclear constraint dependency has been fixed after we report it.

2.5 Fuzzing process

For each API function, D2C performs fuzzing by iteratively generating an input i.e., a set of values of the API function's parameters, and evaluating that input. Specifically, in each iteration, D2C generates values for all required parameters and some optional parameters. The probability for generating each optional parameter is `optional_ratio`. The value selection of this ratio is discussed in Section 3. We do not fuzz all optional parameters because that increases the chance of generating invalid parameters due to incomplete documentation. Additionally, this could help cover more diverse program behaviors. For example, in `torch.nn.CrossEntropyLoss`, specifying either `size_average` or `reduce` overrides the argument `reduction`. Fuzzing with `size_average` or `reduce` separately will test two different behaviors of this API.

D2C generates two types of input: *conforming input* (CI) and *violating input* (VI). The conforming inputs are designed to test the core functionality of the API function while the violating inputs are designed to test the API function's input validity checking code. In both cases, D2C reports bug-triggering inputs that cause serious crashes (e.g., segmentation fault). D2C tests each API function with `maxIter` number of inputs, and the ratio of inputs allocated to each mode (CI or VI) is determined by the ratio `conform_ratio`.

Input generator: The input generator generates one input for each fuzzing iteration. Given a set of extracted constraints, D2C generates a value for each parameter following the generation order (determined by the constraint dependencies as described in Section 2.1). For a conforming input, all generated arguments satisfy extracted constraints for *structure*, *dtype*, *shape*, and *valid value*. If concrete values are specified (e.g., enumerated values) in the constraints, the input generator chooses from those values. Otherwise, it chooses a *dtype* from the list of *dtypes* specified in the constraints and creates a *shape* following the constraints. If the constraints do not specify a list of valid *dtypes*, D2C selects one from a default list of *dtype* described in Section 2.2. While the input generator is choosing *dtype* and *shape* for a parameter, it ensures they are generated according to the parameter dependencies, if any. For example, parameters often have matching dimension(s), so the input generator needs to ensure such shape consistency.

Once the *dtype* and *shape* are determined, the input generator generates an n-dimensional array with values satisfying the given *dtype*, *shape*, and the range as specified in the constraints, if any. Finally, the *structure* constraints are checked and satisfied. Specifically, if the generated value is 1-dimensional and the constraints explicitly specify the *structure* (e.g., tuple or list) for the parameter, the input generator converts the generated value accordingly.

To generate an invalid input to violate the extracted constraints, the input generator randomly selects one parameter as the constraint violating parameter. For this chosen parameter, D2C generates a value that violates one or multiple relevant constraints. For all other parameters, D2C generates their values in the same way as conforming inputs (i.e., conforming to all constraints).

Constraint-guided boundary-based fuzzing: Constraint-guided boundary-based fuzzing helps D2C find bugs that are triggered by boundary values. To improve the effectiveness of boundary-based fuzzing, D2C chooses boundary values that follow the constraints and boundary values that violate the constraints. For each API, D2C picks one parameter with the probability of `mutation_p` to be mutated to one of the boundary cases. We consider six types of boundary mutators: one constraint-specific (boundary values of constraints) and five generics (`None`, zero, zero dimension, empty list, and empty string). As an example, the mutator "zero dimension" sets the size of one of the dimensions of the parameter's shape to 0 (e.g., it mutates a 3-D tensor of shape `[1,1,1]` to `[1,0,1]`). The value selection of `mutation_p` is discussed in Section 3. To diversify the boundary cases, D2C keeps track of the boundary mutators that have been used on each parameter in previous iterations so that it prioritizes the unused mutators when generating values for that parameter in later iterations. In addition, when applying the "zero dimension" mutator, D2C prioritizes the dimensions that have not been mutated in previous iterations.

Test case evaluator: The test case evaluator invokes the target function with the generated input. If a severe failure occurs, D2C reports the input as a bug-triggering input for both conforming and violating inputs. Specifically, D2C returns those inputs causing a segmentation fault, floating-point exception, abort, bus error, and hang in the C++ backend. Crashes from the C++ backend indicate severe problems because DL libraries use C++ code to handle computationally-intensive tasks. In addition to those potential *bugs* (caused by bug-triggering inputs), D2C detects Not-a-Number (NaN) *warnings*, where the output of an API is NaN, which is commonly caused by invalid operations (e.g., division-by-zero) [3]. NaNs could lead to bugs when the NaNs are used.

Testing layer APIs: Layer APIs represent the computational DL layers in a network (e.g., 3D max pooling layer), and for most of them, we simply invoke the APIs with the relevant inputs to test them. For layer APIs in constructor form (class layer APIs), we need to test it differently. Specifically, to test a class layer API, e.g., `tf.keras.layers.MaxPool3D`, D2C first needs to create the layer object, e.g., `layer = MaxPool3D(...)`, and then evaluates the layer object with a generated input `data`, e.g., `output = layer(data)`.

3 EXPERIMENTAL SETUP

Data collection: We choose three popular DL libraries (TensorFlow 2.1.0, PyTorch 1.5.0, and MXNet 1.6.0) as testing subjects. There are 144,541–854,900 words in the API documents among the subjects. We collect documents for between 529 and 1,021 relevant API functions. A function is considered irrelevant if it (1) is deprecated, (2) is from an old version, (3) is a non-layer class constructor, (4) has no input argument, (5) has API document without a "Parameter" description section, or (6) has a detected documentation bug (e.g., format and mismatch issues).

Constraints extraction: We apply PrefixSpan [30] SPM on parameter names and parameter descriptions with NLTK [12] English stop words excluded. For parameter names, we set $min\_len = 1$ to target short patterns (e.g, *"shape"* or *"name"*). These patterns imply the *dtype* of the parameter (e.g., `String` for *"name"*). For parameter descriptions, we set $min\_len = 2$ to catch more descriptive patterns (e.g., *"positive D_TYPE"* or *"tensor of length ..."*)). We set $min\_support = 5$ for both parameter names and descriptions to cover most patterns with reasonable manual inspection effort. Depending on resources available, one can decrease *min_support* to find more rules with increased manual effort or increase this number for a quicker manual inspection at the risk of missing rules.

Threshold settings: For multi-dimensional arrays (including the input data for class layer APIs), D2C generates shapes of 0–5 dimensions or as specified by the constraints. By trying different values of `optional_ratio` and `mutation_p` on a 10% random sampled APIs from all three libraries, we choose `optional_ratio=0.6` and `mutation_p=0.2` since this setting triggers the most number of bugs. For each generated test case, once a timeout of ten seconds is reached, D2C terminates the evaluation process and moves on to the next iteration.

4 EVALUATION AND RESULTS

This section presents our evaluation results of D2C.

4.1 Constraint extraction results

Approach: We apply D2C to extract constraints in our subjects and study the number and quality of constraints. We randomly sample 5% (596) of input parameters of API functions from each subject. For each sampled parameter, we extract relevant constraints of the four categories (i.e., *structure*, *dtype*, *shape*, and *valid value*) manually to build the ground truth that we use to evaluate the accuracy of our semi-automated constraint extraction. Section 5 discusses other categories which remain as future work. The ground truth constraints of the sampled parameter are extracted independently by two authors who have reached 91% agreement, and the disagreement is resolved with a third author to reach a consensus. The extracted constraints by D2C are then compared against manually extracted ground truth constraints for evaluation.

To estimate the quality of extracted rules, we calculate the *accuracy* of the constraint extraction for the sampled parameters (i.e., the percentage of correctly analyzed parameters over the total number of sampled parameters). We use a very strict definition of accuracy—a parameter is correctly analyzed if D2C accurately extracts all constraints of the four categories for this parameter. For example, parameter `size` of `tf.slice` can be either `int32` or `int64`. The extracted *dtype* constraint `size:dtype={int32,int64}` is deemed correct, while `size:dtype={int32}` is considered incorrect. If a parameter's document contains no constraints of the four categories, the parameter is excluded from this accuracy and subsequent precision and recall computation. While it is reasonable to include such no-constraint parameters in our calculation because D2C can trivially extracts nothing, the accuracy may be inflated if there is a large portion of no-constraint parameters. Among the sampled parameters, the numbers of no-constraint parameters are 13 (7.0%), 10 (11.1%), and 51 (16.0%) for TensorFlow, PyTorch, and MXNet, respectively (details in Extraction result section below).

To show the quality of constraints extracted for each parameter, we compute the precision and recall of the extracted constraints of the sampled parameters for each constraint category. *Precision* is the percentage of the correctly extracted constraints (i.e., extracted constraints that match the ground truth) over the number of all extracted constraints. *Recall* is the percentage of correctly extracted constraints over the total number of all ground truth constraints.

Extraction results: Table 2 shows the quality of extracted constraints. In total, D2C gets 19,684 constraints from the three libraries (row *# constr. extracted*). Specifically, D2C extracts a total of 3,773 and 533 frequent subsequences from 18,754 sentences of parameter descriptions and 10,772 parameter names, respectively. We exclude 56.7% subsequences, 49.9% of which contain no constraints (e.g., functionality description of the API functions), and 6.8% describe constraints out of the scope of D2C, e.g., complex constraints (discussed in Section 5). For example, we exclude subsequence *"cudnn operator"* because some sentences that contain this subsequence, e.g., *"Do not select CUDNN operator, if available."*, do not contain the constraints in our scope. After removing irrelevant subsequences, we keep 1,757 frequent subsequences from parameter descriptions and 108 from parameter names. We then group the relevant subsequences for a total of 239 constraint extraction rules, where each rule is merged from 7.8 subsequences on average. For example, we group *"the input array"* and *"the output array"* together to get the rule *"the input/output array"* which indicates `array` as one of the valid *structure*. It takes 8–15 hours of manual time to convert the subsequences to rules per library. Using these rules, D2C extracts on average 7.5 constraints per API for TensorFlow, 5.7 for PyTorch, and 9.5 for MXNet (row *# constr. per API: Avg (Min-Max)* Table 2). Overall, D2C is able to extract constraints from 95.1%, 95.8%, and 98.9% of relevant APIs (details in Section 3) for TensorFlow, PyTorch, and MXNet, respectively.

Table 2: Quality of constraint extraction

	TensorFlow	PyTorch	MXNet	Total/Avg
# APIs with constr. extracted	959	507	1,010	2,476
# constr. extracted	7,172	2,892	9,620	19,684
# constr. per API: *Avg (Min-Max)*	7.5 (1-44)	5.7 (1-33)	9.5 (1-142)	7.6 (1-73)
# examined param.	187	90	319	596
# examined param. with constr.	174	80	268	522
# examined constr.	380	138	481	999
Accuracy (%)	83.3±5.2	90.0±6.4	89.6±3.6	87.6
Precision/Recall for *structure* (%)	97.6/95.4	97.2/94.6	98.4/99.2	97.7/96.4
Precision/Recall for *dtype* (%)	94.1/91.4	98.0/96.1	96.5/94.4	96.2/94.0
Precision/Recall for *shape* (%)	93.2/94.0	90.7/92.9	94.7/90.0	92.9/92.3
Precision/Recall for *valid value* (%)	91.2/83.8	100/77.8	95.1/95.1	95.4/85.6
Precision/Recall for All (%)	94.4/92.4	95.6/93.5	96.4/94.4	95.5/93.4

For each subject, Table 2 shows the number of manually examined parameters (row *# examined param.*), the number of manually examined constraints (row *# examined constr.*), and the number of examined parameters with at least one constraint (row *# examined param. with constr.*). The *Total/Avg* column shows the total number of examined parameters and constraints as well as the average accuracy, precision, and recall. The confidence intervals of accuracy are computed with 95% confidence level.

Overall, D2C achieves a high accuracy of constraint extraction (87.6%) across all three subjects. D2C is the most accurate in extracting constraints for PyTorch and MXNet with high accuracy over 89%, precision over 95%, and recall over 93%. D2C is less accurate when extracting constraints for TensorFlow with good but lower accuracy (83.3±5.2%). The reason is that sentences in TensorFlow's API documents are longer and more free-form compared to other subjects. Hence, it is much harder for D2C to extract a complete set of constraints for each parameter. However, we can still extract thousands of correct constraints for TensorFlow, which helps generate valid inputs and detect more bugs.

D2C achieves high precision and recall (over 90%) for *structure* and *dtype*, constraints. For *valid value* constraints, we achieved lower recall for TensorFlow and PyTorch because of some missing uncommon patterns due to high `min_support = 5`. Reducing the threshold to extract more rules remains as future work.

Generality of rules: Since all three subjects are DL libraries, there might be similarity among their API documents and extraction rules.

Table 3: Rule overlap across the three libraries

Category	TensorFlow		PyTorch		MXNet		Rule Overlap
	Rules	Constraints	Rules	Constraints	Rules	Constraints	Rules (Ratio)
dtype	41	2,808	38	1,089	47	3,518	12 (31.6%)
structure	13	1,287	12	720	16	2,535	4 (33.3%)
shape	16	2,418	13	894	11	2,778	2 (18.2%)
valid value	15	659	5	189	11	789	0 (0.0%)
Total	85	7,172	68	2,892	85	9,620	18 (26.5%)

Table 3 shows the number of rules (col. *Rules*) and the corresponding extracted constraints (col. *Constraints*) for each category of each library. The last column shows the number of rules that are shared among the libraries. The *overlapping ratio* is the number of shared rules divided by the minimum number of rules in the three libraries. For example, 12 of the *dtype* rules are the same across the libraries, resulting in an overlapping ratio of 31.6%, which is $\frac{12}{min(41,38,47)}$, since at most only $min(41, 38, 47)$ number of rules can be shared across the libraries.

Overall, 18 rules are shared among the three libraries with an overlapping ratio of 26.5%. On one hand, this result indicates the generality of the extracted rules and suggests that a significant portion of the extracted rules can be reused in extracting constraints for new libraries. For example, *“<ndim>-d/dimension tensor"*, one of the shared rules, captures a common way among all three libraries to describe the number of dimensions of an input tensor. Phrases such as *“a list of `strings`"* and *“tuple of floats"* are used in all three libraries to describe the *structure* of an input parameter which can be matched with a shared rule *“<structure> of <dtype>"*.

On the other hand, the 18 shared rules only account for a small portion of the total number of rules for all three libraries. The rest of the rules are unique to one or two libraries. For example, the *valid value* rule *“<enum$_1$>, <enum$_2$> ... are supported"* is unique to TensorFlow and MXNet. To extract similar *valid value* constraints in PyTorch, D2C uses rules such as *“can only be <enum$_1$>, <enum$_2$> ..."*, e.g., *“`signal_ndim` can only be 1, 2 or 3."*, which is unique to PyTorch. Given the diversity of the rules, the proposed semi-automatic process using sequential pattern mining is essential to reduce manual effort in applying D2C to new libraries.

Comparison with grep: While it may appear to be straight forward to use a *grep-like* technique (i.e., matching existing keywords in documents) to extract constraints, such technique can only identify relevant API document sentences. D2C, on the other hand, uses rules to extract constraints automatically. One naive approach could be assigning a constraint to a match, e.g., if a sentence contains the keyword “integer”, the corresponding parameter would be assigned the constraint `dtype=int`. We implement such an approach by searching in the documents for keywords such as “int” and “interger” for *dtype* constraints and “list” and ‘tensor” for *structure* constraints. We manually collect such keywords in the API documents. This approach misses more than half (58.4%) of the constraints that D2C extracts, including all *shape*, all *valid value*, 44.6% of *dtype*, and 10.3% of *structure* constraints. This result shows the importance of D2C’s pattern mining step which helps create rules that correctly extract thousands of constraints including complex dependencies.

Comparison with existing constraint-extraction techniques: Some techniques [13, 47, 72, 78] can extract constraints regarding valid ranges, which belong to the category *valid value*. This type of constraint only covers 6% of the constraints that D2C extracts.

Table 4: Number of verified new / new / all bugs found

Approach		TensorFlow	PyTorch	MXNet	Total
BL		19 / 29 / 44	10 / 11 / 12	7 / 9 / 12	36 / 49 / 68
BL+CSTR	All	19 / 37 / 59	16 / 18 / 22	9 / 19 / 23	44 / 74 / 104
	CI	13 / 30 / 51	15 / 17 / 20	9 / 18 / 22	37 / 65 / 93
	VI	17 / 31 / 50	6 / 6 / 8	8 / 13 / 17	31 / 50 / 75
D2C (BL+CSTR+BDY)	All	24 / 44 / 67	19 / 20 / 25	11 / 25 / 29	54 / 89 / **121**
	CI	16 / 34 / 54	17 / 18 / 23	9 / 21 / 25	42 / 73 / 102
	VI	20 / 37 / 58	12 / 12 / 14	8 / 16 / 19	40 / 65 / 91

4.2 Bug detection results

Approach: We evaluate D2C’s effectiveness in detecting bugs in both API documents and library code, as they both hurt software reliability now or later [57]. For the documentation bugs, we use D2C to detect inconsistencies within API documents when analyzing them. For library code bugs, we use D2C to test API functions that have at least one constraint extracted. Table 2 shows the numbers of these API functions (row *# APIs with extracted constr.*). For each API function, with the `conform_ratio` set to 50%, D2C generates 2, 000 test inputs (1,200 conforming inputs and 800 violating inputs), evaluates them, and returns bug-triggering inputs that cause serious failures (details in Section 2.5). It takes D2C on average 0.14 seconds to generate and test each input. We manually examine those bug-triggering inputs to check if they reveal real bugs. For those inputs that still trigger the same failures in the nightly version, we report the bugs to the developers.

We implement an unguided fuzzer as the *baseline* for comparison. The baseline generates 2,000 random inputs for all parameters without any constraint knowledge. For a fair comparison, we convert the generated array inputs to tensors assuming that the baseline minimally knows which input argument should be a tensor. Without this conversion, non-tensor input arguments are trivially rejected by PyTorch and MXNet, thus very ineffective in exercising the code in depth. Baseline uses the same `optional_ratio = 0.6` as D2C.

We do not compare with existing fuzzers [4, 5, 9, 56] including AFL [4] because they cannot test Python code: the most popular language for DL which occupies 34.1–48.5% of code in the three evaluated libraries, which are in Python and C++. These fuzzers require code coverage, which is currently unavailable across Python and C++. Instead of code coverage, D2C uses constraints extracted from documents to guide the testing of both Python and C++ code, by generating inputs for the Python API functions, in which C++ code is invoked. In addition, existing fuzzers [4, 5, 56] generate inputs in the format of a sequence of byte arrays. Randomly mutating some bytes is unlikely to generate valid DL-specific inputs. Our baseline is similar to AFL with two enhancements: (1) knowledge of tensors and (2) automatically testing Python and C++.

Bugs in libraries code: Table 4 presents the number of *verified new / new / all* bugs found by the baseline (BL), the baseline with constraints (BL+CSTR), and D2C (where BDY represents boundary-based fuzzing). A bug is verified if it has been fixed or confirmed by the developers. A new bug refers to a previously unknown bug that we reported. We count one bug for each required fixing location.

D2C detects 121 bugs including 89 previously unknown bugs, 54 of which have been verified by the developers (41 fixed and 13 confirmed). Of the 41 fixed bugs, 21 are fixed in C++, 13 are fixed in Python, and 5 are fixed in both. The other 4 are fixed silently after we reported them. The 121 bugs cause 202 APIs to fail because one bug can cause failures in multiple APIs but are fixed in one code location. Thus, we count them as 121 instead of 202 bugs. Of the 202 buggy APIs, 51 have parameters with constraint dependencies.

On the other hand, the baseline detects only 68 bugs with 49 new bugs. D2C detects 52 bugs that the baseline cannot, while missing 7 bugs found by the baseline due to the randomness of the fuzzing process. The unverified new bugs are reproducible and waiting for developers' response. Only 3 out of 80 newly reported bugs receive "won't fix" responses from the developers. They claim such inputs are not supported, which is not stated in the document. We do not count these 3 bugs in the results. This suggests that the majority of our bug-triggering inputs are not false alarms and trigger real bugs. Additionally, D2C has also detected 32 (121 − 89) known bugs that have already been fixed in the nightly versions.

Our fuzzer takes the automatically extracted constraints without any manual examination. The few incorrectly extracted constraints could potentially hurt the fuzzer's effectiveness, but in practice, the impact is small as shown by our strong bug detection results. Alternatively, one can manually examine all extracted constraints first, if they would like to trade manual effort of constraint verification for higher bug detection effectiveness. It is possible that documents themselves are incorrect or incomplete, causing incorrect constraints to be extracted, leading the fuzzer to produce false alarms, where the code is correct, but the API documentation is incorrect. Since we focus on severe bugs such as crashes, all detected bugs are in the library code, as well said by a developer after we reported a crash bug "*A segmentation fault is never OK and we should fix it with high priority.*"

Boundary-based fuzzing results: The effectiveness of boundary-based fuzzing is shown by comparing the results of row "BL+CSTR" and "D2C" in Table 4. D2C detects 17 (121 −104) more bugs with boundary-based fuzzing enabled since it generates more inputs with boundary values. However, without the guidance of the constraints, boundary-based fuzzing alone does not outperform the baseline: BL+BDY detects the same number (68) of bugs as BL, since the inputs are likely to be rejected before triggering any bugs.

Conforming and violating inputs: As discussed earlier, D2C generates both conforming inputs (CIs) and violating inputs (VIs). Row "CI" and "VI" in Table 4 presents the breakdown of the bug detection for CIs and VIs with `conform_ratio = 60%`. The results show that the CIs alone (with half of the test inputs of the baseline) finds more bugs (102 bugs) than the baseline (68 bugs), and the VIs alone (with half of the test inputs) also finds more bugs (91 bugs) than the baseline. We manually verify the generated CIs and VIs: out of 102 CI bugs we found, 84 of them are caused by valid inputs conforming to the ground truth constraints. The rest of the CI bugs are caused by invalid inputs generated by conforming to inaccurate constraints; out of 91 VI bugs we found, all of them are caused by invalid inputs violating the ground truth constraints.

Many bugs are detected by both CIs and VIs (comparing the "All" row with the "CI" and "VI" rows in Table 4) because D2C violates the constraints of one parameter only when generating VIs. When a crash is caused by one of the conforming parameters of a VI, it is likely to be triggered by a CI also. However, both CIs and VIs detect their unique bugs, thus both are effective in detecting bugs.

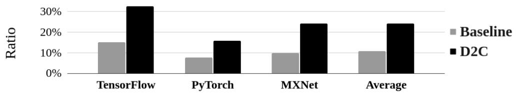


Figure 3: Ratio of passing inputs

We set `conform_ratio` to values between 20%–90% (with a 10% increment) and set `conform_ratio=60%` the value since the fuzzer detects the most number of bugs under the setting. Without the constraints, a baseline is much worse than the results from any of the ratio setups. Table 4 shows that a *key contribution of our work is the ability to extract constraints from documents*. One cannot choose to focus on valid or invalid inputs without knowing the definition of valid inputs for an API. D2C enables this choice by extracting input constraints from DL API documents, and use these constraints to guide fuzzing to find more bugs.

NaN errors: In addition to detecting 121 bugs, D2C detects 34 NaN errors (Section 2.5), 6 of which are fixed by the developers. For example, `tf.nn.avg_pool` outputs NaN when `ksize = 0`, which is fixed by throwing an exception if `ksize = 0`.

Bugs in API documents: D2C detects three types of documentation bugs: (1) formatting bugs (e.g., indentation issue); (2) signature-description mismatch (e.g., the description refers to a parameter that is not specified in the API signature); and (3) unclear constraint dependency (Section 2.4). We detect the first two types while collecting API documents and the third when extracting constraints. D2C detects 38 previously unknown documentation bugs in 53 API functions of the three libraries (11 formatting bugs, 24 signature-description mismatches, and 3 unclear constraint dependencies). We report all 38, 28 of which have been fixed or confirmed by the developers, which indicates that D2C detects documentation bugs that developers care to fix.

Bug examples: We present three examples of bugs detected by D2C that the baseline fails to detect. All of them have been fixed by developers after we report them. **Bug 1** is the previously unknown bug in PyTorch API `grid_sample` discussed the Introduction (Fig. 1). **Bug 2:** When testing API `tf.image.crop_and_resize`, D2C detects a segmentation fault where parameter `boxes` is a 2-D tensor which represents the location of the cropping bounding box. If `boxes` contains a large value (e.g., `1.0e+40`), it gets converted to infinity (`inf`), which causes an invalid access (i.e., segmentation fault). The developers add an `inf` check before the memory access to fix it. The condition to trigger this bug seems trivial but there are complex dependency constraints to follow to pass the input checking code. The size of the first dimension of `boxes` must be identical to the length of parameter `box_indices`. Without the constraints, random fuzzing fails to trigger the bug. **Bug 3:** In the API `tf.nn.avg_pool3d`, D2C triggers a division-by-zero bug by setting one element of `ksize` to 0. To trigger this bug, D2C needs to generate valid values (`"VALID"` or `"SAME"`) for the required parameter `padding`. D2C extracts these valid values from the API document. Without this constraint, a random fuzzer fails to generate the valid values for `padding` to detect this bug.

4.3 Valid-input generation results

Approach: As discussed in the Introduction, generating valid inputs is essential to exercise the core functionality of the API function. While D2C attempts to generate CIs, these CIs may still be invalid if the constraints extracted are incorrect or incomplete. We study the percentage of generated CIs that are valid inputs. We compute the ratio out of 1,000 CIs with the first 1,000 baseline inputs for each API function. Since manually examining the validity of all inputs is impractical and the validity checking of mature projects (e.g., our subjects) is generally reliable, we make a reasonable approximation by counting the number of passing inputs whose executions terminate normally.

Results: Fig. 3 presents the ratio of passing inputs for each subject and the average. On average, D2C outperforms the baseline by generating more than twice the passing inputs. The results suggest that D2C is much more effective in generating valid input than the baseline to test the core functionality code to detect more bugs.

Although D2C outperforms the baseline, the ratio of passing inputs is relatively low (24.2% on average), because API documents are often incomplete. We hope that D2C can convince developers to write more complete API documents after learning that API documents can help them find bugs.

5 THREATS TO VALIDITY

Practicality of generated inputs: D2C generated boundary inputs from a large uniformly distributed value range. However, such corner case inputs are beneficial to the improvement of standalone API function's robustness as they help detect 54 new bugs which have been fixed or confirmed after we report them.

Complex constraints: This work does not use some complex constraints: constraints that require (1) a class object, (2) a pointer of a function, (3) a nested structure, and (4) indirect dependency with the constraints of another parameter (discussed in Section 2.4). However, these complex constraints are uncommon in DL libraries (appeared in only 6.4% of our sampled parameters), thus excluding them should not affect the effectiveness of D2C much.

Manual rule construction: For the three DL libraries, a one-time cost of up to 15 man hours per library is needed to manually construct rules from frequent subsequences. We hope D2C's results can convince developers to write documentation in more consistent formats/expression patterns, so tools such as D2C can more easily extract useful information to help detect bugs in the libraries.

Testing Python and C++ code: Since DL libraries' core computations are in C++, it may appear to be more reasonable to directly test C++ code. However, since Python APIs are the most popular for DL, testing them is testing the common use cases. D2C test Python APIs which invoke the computations in C++, so D2C finds bugs in both Python (21 bugs) code and C++ (24 bugs) code.

6 RELATED WORK

D2C is the first DL library testing technique that extracts input constraints semi-automatically to guide testing.

Testing DL libraries: The handful of DL libraries' testing techniques focus on addressing the test oracle challenge. They leverage differential testing [20, 29, 51, 55, 65, 67] or oracle approximation [42, 75] to obtain oracles. D2C uses only crashes as an indicator of unexpected behaviors and addresses the challenge of obtaining input constraints automatically.

Existing techniques are designed to detect specific types of bugs such as shape-related (e.g., tensor shape mismatch) [20, 34], numerical [20, 29] (e.g., returns `NaN/Inf`), decreased accuracy [20], and performance [61]. On the other hand, D2C finds general bugs that lead to severe crashes by generating input arguments of API functions based on the constraints specified in the documentation.

TensorFlow developers use OSS-Fuzz [6] along with libFuzzer [5] to test only 19 TensorFlow's C++ API functions. It requires developers to manually encode constraints about data structures and properties to reinterpret the sequence of byte-arrays returned by libFuzzer. This would take prohibitive amount of manual effort to test on the same scale that D2C is capable of (i.e., extracting constraints automatically and testing on a total of 2,476 APIs).

Unit test generation and fuzzing: D2C belongs to a large body of work that generates unit tests. Random and search-based techniques [19, 21, 46, 62] generate a sequence of methods as test cases. Our work is a random testing technique that generates API function arguments (not method sequences) for DL libraries that require DL-specific constraints. Dynamic symbolic execution engines for unit testing [24, 27, 54] generate inputs for API functions. However, these tools require heavy program analysis, causing scalability issues. In contrast, we extract constraints from API documents, which are light-weight. While BGRT [18] leveraged a search-based algorithm to detect floating-point errors, D2C focuses on crashes.

The state-of-the-art fuzzers [4, 5, 9] have been adopted to test non-DL libraries [14, 15, 22, 35, 36, 50, 52]. They would not work well for DL libraries (Section 4.2). Test generation tools such as Randoop [46] generate a sequence of function calls to create various states of input objects. However, it works only for a statically-typed language (e.g., Java) and would fail to create valid dynamically-typed objects for Python (the most popular language for DL [8]).

Analyzing software text to detect bugs: Prior work leverages documents [16, 38] and comments [57–59, 76, 78] to detect inconsistency bugs between code and its specifications. Some prior work leverages UML statecharts for test case generation [45] and translates software specifications into assertions [37], verificatino code [38], and oracles [25, 41]. Different from these techniques, D2C uses sequential pattern mining to aid the extraction of constraints from API documents to guide input generation for testing DL libraries.

Testing DL models: Many fuzzing techniques test the robustness of DL *models* instead of DL *libraries* by finding adversarial inputs (e.g., images or natural language texts) for the models[23, 28, 31, 39, 43, 44, 60, 63, 64, 66, 68, 71, 77, 79]. Testing DL models alone is insufficient, as DL libraries contain bugs [32, 33, 51, 73, 74], which hurt the accuracy and speed of the entire DL system [51]. Different from these techniques, D2C tests DL libraries.

7 CONCLUSION

We propose D2C, which leverages sequential pattern mining to extract input constraints from API documents and uses the constraints to guide the testing of DL API functions. The constraints enable D2C to generate valid and invalid inputs including boundary inputs to detect more bugs. D2C aids the generation of 239 rules, which was used to extract 19,684 constraints automatically with 83.3–90.0% accuracy. D2C detects 121 bugs, 89 of which are previously unknown, 54 of which are fixed or confirmed by developers after we reported them. In addition, D2C finds 38 inconsistencies in API documents, 28 of which have been fixed or confirmed by developers.

REFERENCES

[1] 1999. *The Java Modeling Language (JML)*. "https://www.cs.ucf.edu/~leavens/JML/examples.shtml"

[2] 2004. *Beautiful Soup*. https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[3] 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935

[4] 2013. *American Fuzzy Lop*. http://lcamtuf.coredump.cx/afl/

[5] 2015. *libFuzzer – a library for coverage-guided fuzz testing*. http://llvm.org/docs/LibFuzzer.html

[6] 2016. *OSS-Fuzz*. https://github.com/google/oss-fuzz

[7] 2016. *pytype*. "https://github.com/google/pytype"

[8] 2017. What is the best programming language for Machine Learning? https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7.

[9] 2019. *FuzzFactory: Domain-Specific Fuzzing with Waypoints*. "https://github.com/rohanpadhye/fuzzfactory"

[10] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[11] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering*. IEEE, 3–14.

[12] Steven Bird, Edward Loper, and Ewan Klein. 2009. *Natural Language Processing with Python*. O'Reilly Media Inc.

[13] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 242–253.

[14] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing.. In *ACM Conference on Computer and Communications Security*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2329–2344.

[15] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain.. In *ACM Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1032–1043.

[16] Sandeep Chaudhary, Sebastian Fischmeister, and Lin Tan. 2014. em-SPADE: a compiler extension for checking rules extracted from processor specifications. *ACM SIGPLAN Notices* 49, 5 (2014), 105–114.

[17] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 [cs.DC]

[18] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. 2014. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 43–52.

[19] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exp.* 34, 11 (2004), 1025–1050.

[20] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing Probabilistic Programming Systems. *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3236024.3236057

[21] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (feb. 2013), 276 –291.

[22] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing.. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 679–696.

[23] Xiang Gao, Ripon K Saha, Mukul R Prasad, and Abhik Roychoudhury. 2020. Fuzz Testing based Data Augmentation to Improve Robustness of Deep Neural Networks. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing *(PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[25] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th international symposium on software testing and analysis*. 213–224.

[26] Karam Gouda, Mosab Hassaan, and Mohammed J Zaki. 2010. Prism: An effective approach for frequent sequence mining via prime-block encoding. *J. Comput. System Sci.* 76, 1 (2010), 88–102.

[27] Hui Guo and Cindy Rubio-González. 2020. Efficient generation of error-inducing floating-point inputs via symbolic execution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1261–1272.

[28] 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 739–743. https://doi.org/10.1145/3236024.3264835

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

[30] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. 2001. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*. Citeseer, 215–224.

[31] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. 2019. DeepMutation++: A Mutation Testing Framework for Deep Learning Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1158–1161.

[32] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of 42nd International Conference on Software Engineering (ICSE '20)*. ACM.

[33] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[34] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs.. In *ECOOP 2020*.

[35] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage.. In *ASE*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 475–485.

[36] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing.. In *ESEC/SIGSOFT FSE*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 627–637.

[37] Shuang Liu, Jun Sun, Yang Liu, Yue Zhang, Bimlesh Wadhwa, Jin Song Dong, and Xinyu Wang. 2014. Automatic early defects detection in use case documents. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 785–790.

[38] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. 2020. RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1837–1852.

[39] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. 100–111.

[40] Nizar R Mabroukeh and Christie I Ezeife. 2010. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)* 43, 1 (2010), 1–41.

[41] Manish Motwani and Yuriy Brun. 2019. Automatically generating precise Oracles from structured natural language specifications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 188–199.

[42] M. Nejadgholi and J. Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 785–796.

[43] Yixin Nie, Yicheng Wang, and Mohit Bansal. 2019. Analyzing compositionality-sensitivity of NLI models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 6867–6874.

[44] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, California, USA, 4901–4911.

[45] Jeff Offutt and Aynur Abdurazik. 1999. Generating tests from UML specifications. In *International Conference on the Unified Modeling Language*. Springer, 416–429.

[46] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[47] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring Method Specifications from Natural Language API Descriptions. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA,

815–825.

[48] 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. In *Advances in neural information processing systems*. 8026–8037.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[50] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation.. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 697–710.

[51] 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 *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1027–1038. https://doi.org/10.1109/ICSE.2019.00107

[52] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2018. Smart Greybox Fuzzing. *CoRR* abs/1811.09447 (2018).

[53] Noam Rathaus and Gadi Evron. 2007. *Open Source Fuzzing Tools*. Syngress Publishing.

[54] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C *(ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[55] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-Implementation Testing of Supervised Learning Software. In *Proc. AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS)*.

[56] Robert Swiecki. 2015. *Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options*.

[57] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*Icomment: Bugs or Bad Comments?*/. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. ACM, New York, NY, USA, 145–158. https://doi.org/10.1145/1294261.1294276

[58] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1985793.1985796

[59] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 260–269. https://doi.org/10.1109/ICST.2012.106

[60] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

[61] Saeid Tizpaz-Niari, Pavol Černỳ, and Ashutosh Trivedi. 2020. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 189–199.

[62] Paolo Tonella. 2004. Evolutionary Testing of Classes *(ISSTA '04)*. ACM, New York, NY, USA, 119–128. https://doi.org/10.1145/1007512.1007528

[63] Sakshi Udeshi and Sudipta Chattopadhyay. 2019. Grammar Based Directed Testing of Machine Learning Systems. *CoRR* abs/1902.10027 (2019). arXiv:1902.10027

[64] Jonathan Uesato, Ananya Kumar, Csaba Szepesvari, Tom Erez, Avraham Ruderman, Keith Anderson, Krishmamurthy, Dvijotham, Nicolas Heess, and Pushmeet Kohli. 2018. Rigorous Agent Evaluation: An Adversarial Approach to Uncover Catastrophic Failures. arXiv:1812.01647 [cs.LG]

[65] Jackson Vanover, Xuan Deng, and Cindy Rubio-González. 2020. Discovering discrepancies in numerical libraries.. In *Proceedings of the 2020 International Symposium on Software Testing and Analysis (ISSTA 2020)*. 488–501. https://doi.org/10.1145/3395363.3397380

[66] Haohan Wang, Da Sun, and Eric P Xing. 2019. What if we simply swap the two text fragments? a straightforward yet effective way to test the robustness of methods to confounding signals in nature language inference tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7136–7143.

[67] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*.

[68] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 2018. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 408–426.

[69] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. Dase: Document-assisted symbolic execution for improving automated software testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 620–631.

[70] Qian Wu, Ling Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. 2013. Inferring dependency constraints on parameters for web services. In *Proceedings of the 22nd international conference on World Wide Web*. 1421–1432.

[71] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: A Coverage-guided Fuzz Testing Framework for Deep Neural Networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 146–157. https://doi.org/10.1145/3293882.3330579

[72] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*.

[73] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. IEEE/ACM.

[74] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 2020 International Symposium on Software Testing and Analysis (ISSTA 2018)*. 129–140. https://doi.org/10.1145/3213846.3213866

[75] W. Zheng, W. Wang, D. Liu, C. Zhang, Q. Zeng, Y. Deng, W. Yang, P. He, and T. Xie. 2019. Testing Untestable Neural Machine Translation: An Industrial Case. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 314–315.

[76] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 307–318.

[77] Husheng Zhou, Wei Li, Yuankun Zhu, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2018. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. *CoRR* abs/1812.10812 (2018). arXiv:1812.10812

[78] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37.

[79] Zhi Quan Zhou and Liqun Sun. 2019. Metamorphic Testing of Driverless Cars. *Commun. ACM* 62, 3 (Feb. 2019), 61–67. https://doi.org/10.1145/3241979