DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions

***Abstract*—Widely-used deep learning (DL) libraries demand reliability. Thus, it is integral to test DL libraries' API functions. Despite the effectiveness of fuzz testing, there are few techniques that are specialized in fuzzing API functions of DL libraries.**

To fill this gap, we design and implement a fuzzing technique called DocTer for API functions of DL libraries. Fuzzing DL API functions is challenging because many 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 to test the core functionality of API functions. DocTer extracts DL-specific constraints from API documents, and use these constraints to guide the fuzzing to generate valid inputs automatically. DocTer also generates inputs that violate these constraints to test the input validity checking code. To reduce manual effort, DocTer applies sequential pattern mining technique on API documents to help DocTer users create rules to extract constraints from API documents automatically.

Our evaluation on three popular DL libraries (TensorFlow, PyTorch, and MXNet) shows that DocTer's accuracy in extracting input constraints is 82.2–90.5%. DocTer detects 46 bugs, while a baseline fuzzer without input constraints detects only 19 bugs. Most (33) of the 46 bugs are previously unknown, 26 of which have been fixed or confirmed by developers after we report them. In addition, DocTer detects 37 inconsistencies within documents, including 25 fixed or confirmed after we report them.

***Index Terms*—fuzzing, testing, text analytics, deep learning**

I. Introduction

With growing interests in building intelligent systems using Machine Learning (ML) including Deep Learning (DL) techniques, various libraries (e.g., TensorFlow [1] and PyTorch [2]) have been released to allow developers to easily integrate ML algorithms in their applications. These libraries have made developing ML models efficient and convenient.

However, ML libraries contain bugs [3]–[7], which hurt not only the development but also the accuracy and speed of the ML models. Therefore, a DL library needs to be well-tested for better reliability.

A standard practice of testing these API functions is to pass various input values to these functions and inspect any 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 [8].

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 to test the core functionality of DL API functions.

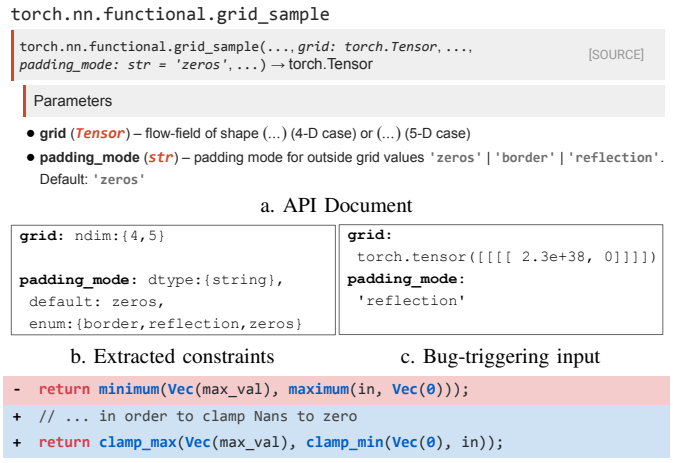

Fig. 1. Documents of PyTorch API function `grid_sample`, which helps our tool detect a bug that was fixed after we reported it to PyTorch Developers.

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 a specific *data structure* 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 `String`) 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 is often insufficient to pass 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.

There is a lack of techniques that are specialized in fuzzing API functions of DL libraries. First, fuzzing techniques in the DL domain have been mostly used to test the robustness of DL *models* instead of DL *libraries* by finding adversarial inputs (e.g., images or natural language texts) for the models [9]–[22]. As discussed earlier, testing DL models alone is insufficient, as a DL library itself contains bugs [3]–[7], which hurt the accuracy and speed of the entire DL system [7].

Second, general-purpose fuzzers that support API fuzzing cannot effectively generate valid inputs for DL API functions. Fuzzers such as AFL [23], HonggFuzz [24], and libFuzzer [25] generate inputs in the format of a sequence of byte arrays. Thus, randomly mutating some bytes in the input is unlikely to generate valid DL-specific data structures (i.e., tensors) following their properties. Other test generation tools such as Randoop [26] generate a sequence of function calls to create various states of objects under which the function is executed. However, this approach works only for a statically-typed language (e.g., Java) where each variable has a static type. It would fail to create valid objects in Python, which is the most popular language used for DL libraries [27], because Python is a dynamically-typed language where variable types are unknown until runtime.

A. Our Approach

To fill this gap and tackle these challenges, we develop a fuzzing technique called *DocTer*, which extracts constraints from API documentation to guide the generation of test inputs for DL API functions. Since DL API functions are dominantly written in more than one programming language, e.g., Python for the user interface code and C++ for the core matrix calculations, DocTer, which generates inputs for the Python API functions, tests both the Python code and the C++ code.

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* [28], [29] to mine frequently occurring patterns in API documents and manually transform them into rules to extract constraints automatically.

DocTer first analyzes free-form API documentation (e.g, Fig. 1a) to extract input constraints (e.g., Fig. 1b). DocTer uses these constraints to guide the fuzzing so that it generates a valid input (e.g, Fig. 1c) that satisfies the constraints. DocTer then evaluates the generated test input by checking if it runs successfully without failures, e.g., crashes. If a failure occurs with a valid input, it is highly likely that 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 DocTer 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 parameter `grid` is either 4-D or 5-D `tensor`. Following the extracted constraints in Fig. 1b, DocTer 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 an 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.

In addition to valid inputs, DocTer 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 DocTer "*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.

Since incorrect API documentation provides false information about APIs, which often misleads developers to introduce bugs in code [30], it is important to detect bugs in API documents as well. Different from prior work [30], [31] that detects inconsistencies between documents/comments and code, DocTer 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*.

B. Contributions

In this paper, we make the following contributions:

- A technique that automatically extracts 17,919 constraints from API documentation with the focus on four categories of input properties in DL APIs: *dtype*, *structure*, *shape*, and *valid values* for 2,273 API functions across the three widely-used DL libraries, TensorFlow, PyTorch, and MXNet [32]. The constraint extraction accuracy is 82.2–90.5%.

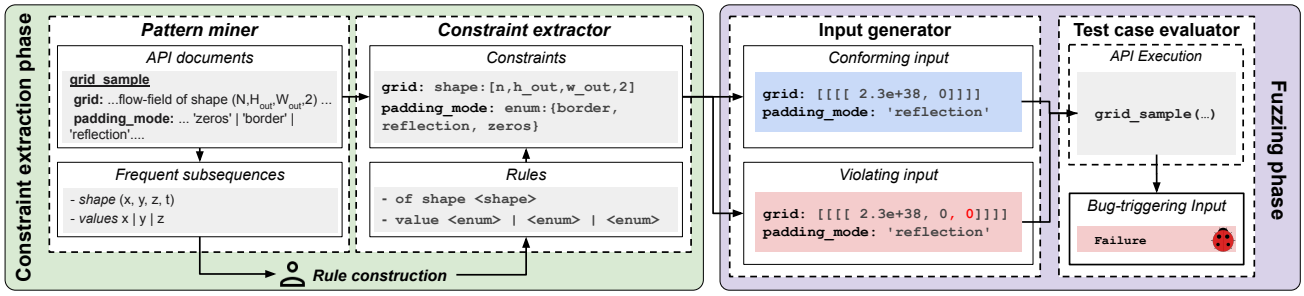

Fig. 2. Overview of DocTer

- Use of sequential pattern mining to help identify patterns from API documentation for constraint extraction.
- A new fuzz-testing technique capable of generating multi-dimensional array object inputs (i.e., tensors) as well as other general inputs, guided by the four categories of constraints above.
- A prototype *DocTer* that detects 46 bugs in the three libraries, while a baseline fuzzer that has no knowledge of constraints detects 19 bugs only. DocTer detected all the bugs that the baseline detected, i.e., 27 $(46 - 19)$ bugs are found by DocTer but not the baseline. Among the 46 bugs, 33 are previously unknown bugs, 26 of which have already been fixed (13) or confirmed (13) by the developers after we report them. In addition, DocTer detects 37 documentation bugs, 25 of which have already been fixed (10) or confirmed (15) after we report them.

II. Approach

A. Challenges and Overview

Fig. 2 shows the overview of DocTer using an example of the PyTorch function `grid_sample` (whose document is 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 [31], [33]–[39]. We leverage *sequential pattern mining* [28], [29] (SPM) to collect frequently occurring patterns and manually transform them into constraint extraction rules (or *rules* for short) 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 the constraint dependencies (i.e., the relation between the properties of different parameters). Generating inputs that follow such constraints requires DocTer to determine the parameters generation order correctly to ensure that the generated values of earlier parameters do not break the constraint dependency with later parameters. We address this by generating the values for 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 semi-automatic 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 manually verified and transformed into rules (up to 15 hours per project). The constraint extractor takes these rules and automatically extracts a set of constraints for each input parameter.

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, DocTer 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 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 (as shown in 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 (as shown in Fig. 1c and explained in the Introduction). The test case evaluator executes `grid_sample` with this input and detects a segmentation fault. This PyTorch bug is previously unknown

and has been fixed after we report it.

B. 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 [31], [34], 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, DocTer automatically applies SPM on sentences of API documents to find *frequent subsequences*. These subsequences provide insight and templates that save manual efforts required to construct rules from scratch.

API document collection and preprocessing: Before applying SPM, DocTer 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 [40]. Since sentences are a natural unit of organizing constraints, we split the description into sentences using regular expressions. Parameter names are extracted from API signatures. We tokenize the sentences and parameter names into words using white spaces and `"_"` respectively. These extracted 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 *dtypes* in these lists 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" and "integer") to match the format of API documents. We also add a common *dtype* that is missing such as `String`. In total, we use 30, 18, and 18 type phrases for TensorFlow, PyTorch, and MXNet, respectively. We manually collected 7 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. Another way is *"tensor type `D_TYPE`"* (occurs 135 times). 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). These examples show that it would be difficult, tedious, and error-prone for developers to manually design extraction rules.

Therefore, DocTer uses SPM to automatically find frequent subsequences from the sentences in the API documents. SPM discovers frequent subsequences within a sequence dataset [41]. 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 a subsequence $< a, d >$ appears four times in the dataset, its `support` and `length` are 4 and 2, respectively. From dataset D, an SPM algorithm finds all frequent subsequences that occur at least `min_support` times and have a length of at least `min_len` [42] efficiently, while a naive approach that counts the frequencies of all possible patterns does not scale. DocTer uses PrefixSpan SPM [29], for its efficient processing. We select the same `min_support` and `min_length` thresholds for all three evaluated DL libraries to demonstrate the generality.

C. 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 categories because they represent the most common properties of input parameters of API functions in major DL libraries. With these four categories, DocTer is able to extract constraints from almost all (97.8%) of the collected API functions in TensorFlow, PyTorch, and MXNet. 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 input parameter. For example, in row 8 of Table I, `weights` has a shape of `[num_classes, dim]` (i.e., it is a 2-dimensional array with the sizes of the first and the second dimension being `num_classes` and `dim`, respectively).
- *valid value*: a set of enumerated values (e.g., parameter `padding` can only take three possible values: `"zeros"`, `"border"`, and `"reflection"`) or the valid range of a numeric parameter (e.g., a float between 0 and 1).

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

TABLE I
RULE EXAMPLES AND THE EXTRACTED CONSTRAINTS

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)}`

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, the `dtype` of a parameter's default value is considered to be one of the valid *dtype* for that parameter.

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 always a 1-D array with each array element specifying the size of a dimension, parameters with names containing "*shape*" should be a 1-D array of non-negative integers.

D. Constraint extractor

The constraint extractor automatically finds matching texts in the parameter descriptions and names, and extracts the relevant constraints according to the rules. For each rule in the "Examples of extraction rules" column in Table I, we list one example of the sentences (column "Examples of sentences from API documents") that can be matched. The extractor automatically generates the corresponding constraints shown in the column "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, DocTer 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 constraint dependencies are useful not only for generating valid inputs but also for determining the parameters' generation order.

For *dtype* dependencies, DocTer 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 I 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. The example in row 7 of Table I shows the dependency to be the shape of parameters (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 `weights` should have shape `[num_classes,dim]` (i.e., a 2-dimensional array). The first dimension of this shape is specified by another parameter `num_classes` in the same API function. The second dimension `dim` is a constant which is used by another parameter `inputs` with shape `[batch_size,dim]`, so parameter `weights` should have the same last dimension size as `input`.

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 I 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, DocTer detects inconsistencies within documents as documentation bugs. For example, the parameter names in the description should match the names specified in the function signature. If they mismatch, there is an inconsistency in the document. Since DocTer is capable of analyzing constraint dependencies, DocTer also detects dependency inconsistencies in documents. For example, in the document of `tf.keras.backend.moving_average_update`, the description for parameter `value` is *"A tensor with the same shape as `variable`,..."*. However, parameter `variable` is not documented, which indicates a documentation bug of unclear constraint dependency. DocTer detects this document inconsistency, which has been fixed after we report it.

E. Fuzzing process

For each API function, DocTer performs fuzzing by iteratively generating an input and evaluating that input. It 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 functions' input validity checking code. In both cases, DocTer reports bug-triggering inputs that cause serious crashes (e.g., segmentation fault). DocTer 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 `conformRatio`.

Input generator: The input generator generates one input for each fuzzing iteration. Given a set of extracted constraints, DocTer generates a value for each parameter following the generation order (determined by the constraint dependencies as described in Section II-A). For a conforming input, all generated arguments satisfy extracted constraints. Specifically, for each parameter, the input generator picks a value that conforms to the relevant *dtype*, *shape*, *valid value*, and *structure* constraints. If concrete values are specified in the constraints, the input generator uses those values. Otherwise, it chooses a *dtype* from the list of *dtypes* specified in the parameter constraints and creates a *shape* following the constraints. If the constraints do not specify a list of valid *dtypes*, DocTer selects a *dtype* from a default list of all possible *dtype* described in Section II-B. 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 based on the given *dtype* and *shape*. If there is a valid range constraint (e.g., must be a value between 0 and 1), the input generator only generates values within the specified range. Finally, the *structure* constraints are checked and satisfied. Specifically, if the generated value is a 1-dimensional array and the constraints specify the parameter should be an explicit *structure* (e.g., a tuple or a list), 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, DocTer generates a value that violates one or multiple relevant constraints. For all other parameters, DocTer generates their values in the same way as conforming inputs (i.e., conforming to all constraints).

Test case evaluator: The test case evaluator invokes the target function with the generated input. If a severe failure occurs, DocTer reports the input as a bug-triggering input for both conforming and violating inputs. More precisely, DocTer returns those inputs causing a segmentation fault, floating-point exception, bad memory allocation, and hang in C++ core-level as bug-triggering inputs. Core-level C++ signals indicate severe problems because DL libraries use C++ code to handle computationally-intensive tasks. DocTer finds many abort signals that occurred due to C++ assertion failures. We do not view this behavior as buggy, but as bad coding practice because users mostly invoke the API functions in Python-level. Therefore, DocTer does not report C++ assertion failures as bug-triggering inputs.

III. 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 949, 415, and 959 relevant API functions in the three libraries, respectively. A function is considered irrelevant if it (1) is deprecated, (2) is from an old version, (3) is a 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 [29] SPM on parameter names and parameter descriptions with NLTK [43] 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 so that we can cover most of subsequences 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.

Fuzzing: Since all three libraries heavily depend on `numpy` package, DocTer uses `numpy` array format to generate input values. In this way, the subject library conveniently adapts the generated input. For each generated input, once a timeout of 10 seconds is reached, DocTer terminates the evaluation process and moves on to the next iteration.

IV. Evaluation and Experimental Results

A. Constraint extraction results

Approach: We apply DocTer to extract constraints in our subjects and study the number and quality of constraints. We randomly sample 5% of input parameters of API functions from each subject, which results in a total of 534 parameters. For each sampled parameter, we extract relevant constraints manually to build the ground truth that we use to evaluate the accuracy of our semi-automated constraint extraction. We extract constraints of the four categories: *dtype*, *structure*, *shape*, and *valid value*, because other constraints are out of scope by design of our approach. Section V 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. For any disagreement, we reach a consensus after discussing it with a third author. The extracted constraints by DocTer are 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 DocTer 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

TABLE II
QUALITY OF CONSTRAINT EXTRACTION

	TensorFlow	PyTorch	MXNet	Total/Avg
# APIs with constr. extracted	915	404	954	2,273
# constr. extracted	6,729	2,201	8,989	17,919
# constr. per API: *Avg (Min-Max)*	7.4 (1-44)	5.4 (1-34)	9.4 (1-143)	7.9 (1-74)
# examined param.	167	71	296	534
# examined param. with constr.	157	63	247	467
# examined constr.	341	108	442	891
Accuracy (%)	82.2±5.8	90.5±7.1	90.3±3.6	87.7
Precision/Recall for *structure* (%)	97.4/96.2	96.8/96.8	98.3/99.1	97.5/97.4
Precision/Recall for *dtype* (%)	93.4/90.4	97.4/97.4	96.9/94.5	95.9/94.1
Precision/Recall for *shape* (%)	92.3/93.2	90.9/93.8	95.9/90.6	93.0/92.5
Precision/Recall for *valid value* (%)	90.3/82.4	100/83.3	94.3/97.1	94.9/87.6
Precision/Recall for All (%)	93.7/91.8	95.4/95.4	96.8/94.8	95.3/94.0

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 DocTer can trivially extract 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 10 (6.0%), 8 (11.3%), and 49 (16.6%) 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 II shows the quality of extraction. In total, DocTer extracts 17,919 constraints from the three libraries (row "*# constr. extracted*"). Specifically, DocTer 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 are that contain no constraints (e.g., functionality description of the API functions), and 6.8% are that describe constraints out of the scope of DocTer, e.g., complex constraints (discussed in Section V). 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 1757 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*. Using these rules, DocTer extracts an average 7.4 constraints per API for TensorFlow, 5.4 for PyTorch, and 9.4 for MXNet (row *# constr. per API: Avg (Min-Max)* Table II). Overall, DocTer is able to extract constraints from 96.4%, 97.3%, and 99.5% of relevant APIs (details in Section III) for TensorFlow, PyTorch, and MXNet, respectively.

TABLE III
RULE OVERLAP ACROSS THE THREE LIBRARIES

Category	TensorFlow		PyTorch		MXNet		Rule Overlap
	Rules	Constraints	Rules	Constraints	Rules	Constraints	Rules (Ratio)
dtype	41	2,662	38	855	47	3,208	12 (31.6%)
structure	13	1,245	12	643	16	2,454	4 (33.3%)
shape	16	2,235	13	623	11	2,584	2 (18.2%)
valid value	15	587	5	80	11	743	0 (0.0%)
Total	85	6,729	68	2,201	85	8,989	18 (26.5%)

For each subject, Table II 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, DocTer achieves a high accuracy of constraints extraction (87.7%) across all three subjects. DocTer is the most accurate in extracting constraints for PyTorch and MXNet with a high accuracy over 90%, precision over 95%, and recall over 94%. DocTer is less accurate when extracting constraints for TensorFlow with a good but lower accuracy (82.2±5.8%). 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 DocTer 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.

DocTer achieves high precision and recall (over 90%) for *structure*, *dtype*, and *shape* constraints. For *valid value* constraints, we achieved lower recall for TensorFlow and PyTorch because we miss some uncommon patterns in the mining stage (due to high threshold `min_support = 5`). Reducing the threshold of our sequential pattern mining could help us extract more rules, which remains as future work.

Generality of rules: Since all three subjects are DL libraries, one may wonder the amount of similarity in their API documents and extraction rules. Table III shows the number of rules (column *Rules*) and the corresponding extracted constraints (column *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

TABLE IV
NUMBER OF **VERIFIED NEW / NEW / ALL** BUGS FOUND BY DOCTER AND THE BASELINE

Approach		TensorFlow	PyTorch	MXNet	Total
Baseline		5 / 8 / 14	4 / 4 / 4	1 / 1 / 1	10 / 13 / 19
DocTer	All	8 / 15 / 25	10 / 10 / 11	8 / 8 / 10	26 / 33 / 46
	CI	6 / 13 / 21	6 / 6 / 6	7 / 7 / 9	19 / 26 / 36
	VI	8 / 13 / 21	6 / 6 / 7	8 / 8 / 9	22 / 27 / 37

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, DocTer 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 DocTer to new libraries.

B. Bug detection results

Approach: We evaluate DocTer's effectiveness in detecting bugs in both API documents and library code, as both hurt software reliability now or later [30]. For the documentation bugs, we use DocTer to detect inconsistencies within API documents when analyzing them. For library code bugs, we use DocTer to test API functions that have at least one constraint extracted. Table II shows the numbers of these API functions (row *# APIs with extracted constr.*). For each API function, with the `conformRatio` set to 50%, DocTer generates $1,000$ test inputs (500 conforming inputs and 500 violating inputs), evaluates them, and returns bug-triggering inputs that cause serious failures (details in Section II-E). 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.

Since there are no state-of-the-art API fuzzers for DL libraries, we implement an unguided fuzzer as the *baseline* for comparison. The baseline generates 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.

Bugs in libraries code: Table IV presents the number of *verified new / new / all* bugs found by DocTer and the baseline. 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 have reported.

DocTer detects 46 bugs including 33 new bugs, 26 of which have been verified by the developers (13 fixed and 13 confirmed). On the other hand, the baseline detects only 19 bugs with 13 new bugs. DocTer detects all 27 $(46-19)$ bugs that the baseline cannot, while missing no bug found by the baseline. The unverified new bugs are reproducible and waiting for developers' response. None of the newly reported bugs has received a 'won't fix' response from the developers, which suggests that our bug-triggering inputs are not false alarms and trigger the real bugs. DocTer has also uncovered 13 (*all* bugs $-$ *new* bugs, $46-33$) 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, 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.*"

Conforming and violating inputs As discussed earlier, DocTer generates both conforming inputs (CIs) and violating inputs (VIs). Table IV presents the breakdown of the bug detection for CIs and VIs with `conformRatio = 50%`. The results show that the CIs alone (with half of the test cases of the baseline) find more bugs (36 bugs) than the baseline (19 bugs), and the VIs alone (with half of the test cases) also find more bugs (37) than the baseline. We manually verified the generated CIs and VIs: out of 36 CI bugs we found, 27 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 37 VI bugs we found, all of them are caused by invalid inputs violating the ground truth constraints.

It is widely known and expected that CIs find more bugs than the baseline, because most CIs are expected to be valid inputs, exploring deeper to find hard-to-detect bugs, while the baseline mostly generates invalid inputs which get stuck at the input validity checking code. The reason that VIs find more bugs than the baseline is that the input validity checking code is often complex. The baseline mostly generates inputs that violate all parameter constraints, thus stops at the first batch of checks of input validity. On the other hand, DocTer violates

the constraints of one parameter only for each VI, thus can pass the checking of all other parameters, going much deeper than the inputs generated by the baseline.

By comparing the "All" row with the "CI" and "VI" rows, we can see that many bugs are detected by both CIs and VIs. This is because DocTer violates the constraints of one parameter only when generating VIs. When a crash is caused by one of the conforming parameters of a VI, is likely to be triggered by a CI also. However, both CIs and VIs detect unique bugs that cannot be detected by the other, thus both types of inputs are effective in detecting bugs.

While the optimal ratio between the CIs and VIs may depend on the type of applications and maturity of code (for example, there may be more bugs in the input validity checking and exception handling code in mature projects as the norm cases have been tested more thoroughly), we study how this ratio (`conformRatio`) affects the testing effectiveness on the three DL libraries. We perform our study by varying `conformRatio` between 0% to 100% with a 10% increment. Overall, the numbers of bugs detected using ratio between 20% to 90% differ by at most three bugs which indicates that DocTer is insensitive to this ratio. Thus, we use `conformRatio = 50%` as the default ratio to be more general.

Although the optimal ratio of conforming and violating constraints may change for different projects, it is crucial to know the constraints. We have observed that without the constraints, a baseline is much worse than the results from any of the ratio setups. Table IV shows that a *key contribution of our work is the ability to extract constraints from documents*. Regardless of what inputs (valid or invalid) to focus on, one cannot make such a choice without knowing the definition of valid inputs for an API. DocTer enables this choice by extracting input constraints from DL API documents, and use these constraints to guide fuzzing to find more bugs.

Bugs in API documents: In addition to the bugs in the library code, DocTer also detects documentation bugs during the constraint extraction phase. DocTer can detect 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 (e.g., as discussed in Section II-D, some properties of one parameter depend on those of a missing parameter). We detect the first two types during the API documents collection and the third when extracting constraints with rules.

DocTer detects 37 previously unknown documentation bugs for 53 API functions in three libraries (including 9 formatting bugs, 25 signature-description mismatches, and 3 unclear constraint dependencies). We report all 37 documentation bugs, 25 of which have been fixed or confirmed by the developers (including all 3 unclear dependencies). The result suggests that, in addition to code bugs, DocTer is effective in detecting documentation bugs that developers care to fix.

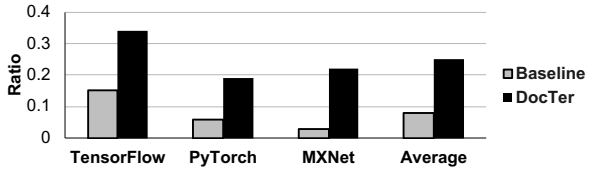

Fig. 3. Ratio of passing inputs

C. Bug examples

We present three examples of bugs detected by DocTer 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: DocTer detects a floating-point exception bug in API `mxnet.ndarray.InstanceNorm`. The bug involves three parameters, `data`, `gamma` and `beta`, which have *shape* dependency on each other: `gamma` and `beta` must have the same length which should be equal to the size of the second dimension of `data`. When at least one of the size of the first two dimensions of `data` is 0, e.g., when it is of shape such as `(0,1,...)` or `(1,0,...)`, a division by zero exception occurs. The fix adds code to check zero before the division.

Bug 3: DocTer detects a segmentation fault in API `torch.cholesky_solve`. This function requires two input tensors to be at least 2-D, and their second to last dimensions should match. The API function takes these two tensors and performs broadcasting (i.e., making their shapes to be compatible for the arithmetic operations). However, the API execution fails to check the shapes for compatibility and attempts to perform broadcasting with the incompatible tensors. This causes a segmentation fault due to an invalid array access. The fix calls a different function that properly checks the shape compatibility before broadcasting.

Discussion: The baseline approach fails to generate valid inputs and cannot trigger any of these bugs. Unless generated inputs conform with the strict constraints, the input validity checking of the API functions rejects the inputs by throwing exceptions. Therefore, the inputs generated by the baseline do not exercise the core functionality, which contains bugs. The inputs generated by DocTer, however, pass the validity check and find the bugs by exploring deeper into the program.

D. 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 DocTer attempts to generate CIs, these CIs may still be invalid if the constraints extracted are incorrect or incomplete. We study what percentage of generated CIs are valid inputs. For a fair comparison, we compare 1,000 CIs with 1,000 baseline inputs. The ratio of valid inputs is computed out of $1,000$ total inputs generated for each API function.

Since manually examining the validity of all inputs for all API functions under test is impractical, we approximate the number of valid inputs by counting the number of passing inputs whose executions terminates without any exceptions or

crashes from both Python and C++ level. Since the validity checking of mature projects (e.g., our subjects) is generally reliable, the passing inputs that have not been rejected by the API functions are likely valid inputs.

Results: Fig. 3 presents the ratio of passing inputs for each subject and the average. DocTer outperforms the baseline approach by generating more than three times of passing inputs that the baseline generates on average. The results suggest that DocTer is likely much more effective in generating valid input than the baseline to test the core functionality code to detect more bugs.

Although DocTer outperforms the baseline for all DL libraries, the ratio of passing inputs is relatively low at 25% on average. This is because API documents are often incomplete. For example, many API documents specify only `ndim` of a tensor and often do not specify the size of each dimension. Thus, even though our input conforms to all constraints given in the document, the source code throws an exception to complain that the input is invalid. Therefore, the effectiveness of generating valid inputs relies on the quality of documents. We hope that DocTer can convince developers to write more complete API documents after learning that API documents can help them find bugs.

V. Threats to Validity

Practicality of generated inputs: Many inputs DocTer generates are boundary inputs that are too large or small because for each dtype, DocTer draws values from the uniformly distributed value range. However, these corner case inputs are beneficial to improve the robustness of standalone API functions by detecting 26 new bugs that developers have already 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 II-D). 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 DocTer much.

Manual rule construction: The extraction rules need to be manually constructed from the frequent subsequences. For the three libraries in this work, it takes one person up to 15 hours per library. The process is also only a one-time cost. We hope DocTer's results can convince developers to write documentation in more consistent formats/expression patterns, so tools such as DocTer can more easily extract useful information to help detect bugs in the libraries.

Python test inputs One may argue that DL libraries' core computations are in C++, so directly testing C++ code is more appropriate. However, since Python APIs are the most popular for DL, testing them is more aligned to the popular use case. DocTer test the Python APIs which invoke the computations in C++, so DocTer can find bugs in both Python and C++ code. DocTer found one bug in Python, which hangs with an infinite loop. Since DocTer focus on severe failures (section II-E), the rest of the 45 bugs DocTer found are in C++ code. Other indicator(s) is need to find more bugs in Python.

VI. Related Work

Testing DL libraries: Among all DL library testing techniques from research and industry, DocTer is the first that extracts input constraints semi-automatically to guide testing. The handful of DL libraries testing techniques focus on addressing the test oracle challenge. They leverage differential testing [7], [44]–[48] or oracle approximation [49], [50] to obtain oracles. DocTer 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) [47], [51], numerical [47], [48] (e.g., returns `NaN/Inf`), decreased accuracy [47], and performance [52]. To do this, they leverage user-specified template to generate numerical values from the specified probability distribution [47], modeling of library semantics with respect to tensor shape [51], different combinations of structure, parameters, weights, and data input of a DNN model [48], and performance clustering [52]. DocTer can find 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 [53] along with libFuzzer [25] to test only 19 TensorFlow's C++ API functions. It requires developers to manually add 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 of APIs that DocTer is capable to do. DocTer extracts input constraints automatically from API documents and use them to generate valid inputs to test on 2,273 APIs across three libraries.

Unit test generation and fuzzing: DocTer belongs to a large body of work that generates unit tests. Random and search-based techniques [26], [54]–[56] generate a sequence of methods as a test case. 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 [57], [58] generate inputs for API functions. However, these tools require heavy program analysis, causing scalability issues. In contrast, we extract constraints from API documents, which is light-weight.

DocTer is a generation-based black-box fuzzer [59]. The state-of-the-art fuzzers such as AFL [23] and libFuzzer [25] are mutation-based grey-box fuzzers guided by code coverage [59] and have been adopted to test various non-DL libraries [60]–[66]. However, they would not work well for DL libraries that enforce DL specific constraints for valid inputs.

Analyzing Software text to detect bugs: Previous work leverages comments [30], [34], [67]–[69] and documents [31], [38], [70] to detect inconsistency bugs between code and its specifications. Some prior work uses software text to extract

input constraints for testing by leveraging UML statecharts for test case generation [71] and translating software specifications into oracles [36], [37], [72]. Different from these techniques, DocTer uses sequential pattern mining to aid the extraction of constraints from API documents to guide input generation for testing DL libraries.

VII. Conclusion

We propose a fuzzing technique called DocTer that tests API functions of deep learning libraries. The technique leverages sequential pattern mining to extract input constraints from API documentation. It uses the constraints to guide the input generation for (1) valid inputs that conform to the constraints to test API function's core functionality, and (2) invalid inputs that violate the constraints to test the input validity checking code. Our results show that DocTer achieves 82.2–90.5% accuracy for the constraint extraction. DocTer detects 46 bugs, 33 of which are previously unknown including 26 of them fixed or confirmed by developers after we report. In addition, DocTer finds 37 inconsistencies in API documents, and 25 of them have been fixed or confirmed by developers. In the future, it would be conceivable to extract other categories of DL constraints and more complex dependency-related constraints from API documents to further fuzz-test DL libraries.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[3] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*. IEEE/ACM, July 2020.

[4] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "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*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 510–520.

[5] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 2020 International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, 07 2018, pp. 129–140.

[6] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, 2020.

[7] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 1027–1038.

[8] N. Rathaus and G. Evron, *Open Source Fuzzing Tools*. Syngress Publishing, 2007.

[9] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "TensorFuzz: Debugging neural networks with coverage-guided fuzzing," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019, pp. 4901–4911.

[10] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "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*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 739–743.

[11] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "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*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 146–157.

[12] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 408–426.

[13] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Commun. ACM*, vol. 62, no. 3, pp. 61–67, Feb. 2019.

[14] J. Uesato, A. Kumar, C. Szepesvari, T. Erez, A. Ruderman, K. Anderson, Krishmamurthy, Dvijotham, N. Heess, and P. Kohli, "Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures," 2018.

[15] S. Udeshi and S. Chattopadhyay, "Grammar based directed testing of machine learning systems," *CoRR*, vol. abs/1902.10027, 2019.

[16] Y. Nie, Y. Wang, and M. Bansal, "Analyzing compositionality-sensitivity of nli models," 2018.

[17] H. Wang, D. Sun, and E. P. Xing, "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," 2018.

[18] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 303–314.

[19] H. Zhou, W. Li, Y. Zhu, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "Deepbillboard: Systematic physical-world testing of autonomous driving systems," *CoRR*, vol. abs/1812.10812, 2018.

[20] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018, pp. 100–111.

[21] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1158–1161.

[22] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*, 2020.

[23] "American fuzzy lop." [Online]. Available: http://lcamtuf.coredump.cx/afl/

[24] R. Swiecki, "Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options," *URl: https://github. com/google/honggfuzz*.

[25] "libfuzzer – a library for coverage-guided fuzz testing." [Online]. Available: http://llvm.org/docs/LibFuzzer.html

[26] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816.

[27] "What is the best programming language for machine learning?" https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7.

[28] K. Gouda, M. Hassaan, and M. J. Zaki, "Prism: An effective approach for frequent sequence mining via prime-block encoding," *Journal of Computer and System Sciences*, vol. 76, no. 1, pp. 88–102, 2010.

[29] J. Han, J. Pei, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth," in *proceedings of the 17th international conference on data engineering*. Citeseer, 2001, pp. 215–224.

[30] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: Bugs or bad comments?*/," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 145–158.

[31] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 260–269.

[32] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," 2015.

[33] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "Dase: Document-assisted symbolic execution for improving automated software testing," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 620–631.

[34] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 242–253.

[35] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang, "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*, ser. ESEC/FSE 2020, 2020.

[36] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 213–224.

[37] M. D. Ernst, "Natural language is a programming language: Applying natural language processing to software development," in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[38] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 307–318.

[39] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 815–825.

[40] "Beautiful soup." [Online]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[41] N. R. Mabroukeh and C. I. Ezeife, "A taxonomy of sequential pattern mining algorithms," *ACM Computing Surveys (CSUR)*, vol. 43, no. 1, pp. 1–41, 2010.

[42] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the eleventh international conference on data engineering*. IEEE, 1995, pp. 3–14.

[43] S. Bird, E. Loper, and E. Klein, *Natural Language Processing with Python*. O'Reilly Media Inc, 2009.

[44] J. Vanover, X. Deng, and C. Rubio-González, "Discovering discrepancies in numerical libraries." in *Proceedings of the 2020 International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, 07 2020, pp. 488–501.

[45] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "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*, ser. ESEC/FSE 2020, 2020.

[46] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, and T. Xie, "Multiple-implementation testing of supervised learning software," in *Proc. AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS)*, 2018.

[47] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems." ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018.

[48] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks." in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2020.

[49] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 785–796.

[50] W. Zheng, W. Wang, D. Liu, C. Zhang, Q. Zeng, Y. Deng, W. Yang, P. He, and T. Xie, "Testing untestable neural machine translation: An industrial case," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 314–315.

[51] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. Smaragdakis, "Static analysis of shape in tensorflow programs." in *ECOOP 2020*, 2020.

[52] S. Tizpaz-Niari, P. Cerný, and A. Trivedi, "Detecting and understanding real-world differential performance bugs in machine learning libraries," 2020.

[53] Oss-fuzz. [Online]. Available: https://github.com/google/oss-fuzz

[54] C. Csallner and Y. Smaragdakis, "Jcrasher: an automatic robustness tester for java." *Softw. Pract. Exp.*, vol. 34, no. 11, pp. 1025–1050, 2004.

[55] P. Tonella, "Evolutionary testing of classes," ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 119–128.

[56] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276 –291, feb. 2013.

[57] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[58] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272.

[59] V. Manes, H. Han, C. Han, s. cha, M. Egele, E. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 10 2019.

[60] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain." in *ACM Conference on Computer and Communications Security*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1032–1043.

[61] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing." in *ACM Conference on Computer and Communications Security*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2329–2344.

[62] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing." in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 679–696.

[63] C. Lemieux and K. Sen, "Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage." in *ASE*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 475–485.

[64] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing." in *ESEC/SIGSOFT FSE*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 627–637.

[65] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation." in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 697–710.

[66] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing." *CoRR*, vol. abs/1811.09447, 2018.

[67] L. Tan, Y. Zhou, and Y. Padioleau, "acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 11–20.

[68] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 27–37.

[69] L. Tan, D. Yuan, and Y. Zhou, "Hotcomments: How to make program comments more useful?" in *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS07)*, May 2007.

[70] S. Chaudhary, S. Fischmeister, and L. Tan, "em-spade: a compiler extension for checking rules extracted from processor specifications," *ACM SIGPLAN Notices*, vol. 49, no. 5, pp. 105–114, 2014.

[71] J. Offutt and A. Abdurazik, "Generating tests from uml specifications," in *International Conference on the Unified Modeling Language*. Springer, 1999, pp. 416–429.

[72] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 188–199.