SkipFuzz: Active Learning-based Input Selection for Fuzzing Deep Learning Libraries

Hong Jin Kang, Yunbo Lyu, Pattarakrit Rattanukul, Truong Giang Nguyen, Stefanus Agus Haryono, Chaiyong Ragkhitwetsagul, Corina Pasareanu, David Lo

Abstract—Many modern software systems are enabled by deep learning libraries such as TensorFlow and PyTorch. The reliability of deep learning libraries is therefore a key concern. For finding crashing inputs to the libraries, the libraries can be tested using the inputs already present in their test suite, but randomly selected inputs are unlikely to uncover bugs. Existing approaches propose strategies to select only valid inputs. However, they do not have methods to explore enough of the input domain. Moreover, to efficiently trigger buggy behavior, a fuzzer should decrease redundancy by selecting inputs that less likely to trigger the same behavior as previously used inputs.

We propose SkipFuzz, a new approach for fuzzing deep learning libraries. SkipFuzz systematically selects inputs to explore the library's input domain and uses fewer redundant inputs. SkipFuzz refines its model of the input constraints of each API function using active learning over the information gathered during fuzzing. The input constraints are leveraged to guide input selection towards valid inputs. To reduce redundancy, SkipFuzz maintains *categories* that group together inputs with common characteristics, e.g. they are tensors of a certain shape. Inputs in one category are distinguished from other categories by the input constraints they would satisfy. Inputs from different categories are used to invoke the library to check if they satisfy a function's input constraints. Our experiments indicate that SkipFuzz generates more crashing inputs than prior approaches. SkipFuzz found 53 crashes, of which 28 were confirmed to be previously unknown. The rest were already known but not fixed yet. 23 unique CVEs were assigned.

Index Terms—Fuzzing, Deep Learning Libraries, Active Learning, Input Selection

1 Introduction

Deep learning is now prevalent, including in safety and security-critical domains. As such, there have been increasing concerns about vulnerabilities in deep learning systems, which can have a severe impact. These vulnerabilities may be exploited, including for denial-of-service attacks on deep learning applications. The development of approaches that effectively fuzz deep learning libraries, such as TensorFlow and PyTorch, is crucial.

Challenges. Fuzzing deep learning libraries is difficult due to the problem of selecting suitable inputs. The space of inputs is very large. Selecting appropriate inputs is, therefore, challenging. The first challenge is that the majority of randomly selected inputs would be invalid and are rejected by validation checks in the library. Thus, fuzzers should avoid generating inputs that execute only the library's validation checks. Prior approaches [11], [40], [41] developed strategies for identifying and selecting valid inputs from a set of seed inputs. However, this may have an unintended effect of narrowing down the input space too quickly, and cause the fuzzers to miss inputs that could be invalid but could crash the library. This calls for approaches that do not overlook and can select inputs that are invalid, but sufficiently resemble valid inputs to pass the libraries' input validation checks. The second challenge is that inputs may be *redundant*, and thus not useful for uncovering new bugs. The selection of similar inputs would test the same library behavior, leading to wasted effort in fuzzing. This calls for approaches that avoid inputs that may test the same behavior. Given the observations of prior test outcomes (e.g., an `InvalidArgumentError` exception thrown given a `float` tensor), a fuzzer should select inputs (e.g., an `int` tensor) to test behaviors that are different from the already observed test outcome.

Existing approaches. Off-the-shelf fuzzers, such as AFL [1], do not encode knowledge of input constraints, and fail to generate valid inputs. Other existing works propose methods to select valid test inputs. `DocTer` [41] infers the input constraints from API documentation, then fuzzes the library based on the inferred constraints. `FreeFuzz` [40] mines valid inputs of functions from open source code. `DeepRel` [11], building on `FreeFuzz`, identifies pairs of similar functions for sharing valid inputs. `DocTer` and `DeepRel` rely on API documentation, which may not always be available or well-maintained. All of the approaches above have a goal of selecting valid inputs but do not have strategies of avoiding redundant inputs, which causes them to miss some crashing inputs.

Our approach. In this paper, we propose an approach (embodied in a tool), SkipFuzz, that selects inputs for inferring a model of the input constraints of each function during fuzzing. SkipFuzz does not require existing specifications. Instead, it learns the input constraints automatically during fuzzing. SkipFuzz uses the hypothesis that inputs sharing the same domain-specific properties lead to the same outcomes. To infer the constraints, SkipFuzz has to select inputs that are not redundant (otherwise, it gains no additional information), and systematically explore the space of inputs (identifying an input constraint requires SkipFuzz to check both valid and invalid inputs). SkipFuzz employs *active learning*, which learns by interactively querying an

oracle. The active learner queries the test executor with inputs that satisfy different possible input constraints. In SKIPFUZZ, the test executor takes the role of the oracle; based on the queries, it constructs test cases using inputs. The *test outcomes* (indicating if the input is valid, invalid, or crashing) are provided back to the active learner, which refines its model of the input constraints. Once inferred, the input constraints enable the selection of valid inputs.

Specifically, SKIPFUZZ distinguishes test inputs by a set of input *properties*, whose design is inspired by the input constraints and root causes of bugs identified in prior studies analyzing deep learning libraries [20], [21], [22], [41]. SKIPFUZZ reduces redundancy by assuming that inputs with the same properties would satisfy the same input constraints; if an input fails a validation check, then other inputs with the same properties will fail the same check. With this assumption, SKIPFUZZ avoids the selection of inputs that it can already determine to lead to the same outcome. The input properties differentiate inputs by their structure, types, shapes, and values, corresponding to the possible input constraints. Selecting inputs with different properties allows for reduced input redundancy and may provide new information about the input constraints.

The active learner identifies a hypothesis for the input constraints that is *consistent* with the observed outcomes. A consistent hypothesis is one where the behavior of the program under the hypothesis matches that of the actual program [9]. Given the observations of the test outcomes indicating if each input was valid or invalid, an ideal hypothesis is a set of properties that matches all valid inputs but excludes all invalid inputs. SKIPFUZZ quantitatively assesses the consistency between a hypothesis and the observed test outcomes by measuring the overlap between the expected outcomes given the hypothesis against the observed outcomes. After finding an adequately consistent hypothesis, SKIPFUZZ selects only inputs satisfying it, enabling a high proportion of valid selected inputs.

In our experiments, SKIPFUZZ detects crashes in 108 functions in TensorFlow and 58 functions in PyTorch. After grouping the crashes with similar root causes, the new crashes have been reported to the libraries. 23 TensorFlow vulnerabilities and 6 PyTorch bug reports have been fixed. SKIPFUZZ can trigger up to 63% of the crashes found by the prior approaches, `DocTer` and `DeepRel`, while the majority of the crashes found by SKIPFUZZ were not found by them. After inferring an input constraint, SKIPFUZZ selects valid inputs most of the time (>80%) of the time, indicating that the inferred constraints are of good quality. Overall, through the process of inferring a model of the input constraints, SKIPFUZZ finds more crashing inputs than prior approaches.

We present the following contributions:

- To fuzz deep learning libraries, we leverage domain knowledge of the libraries and categorize inputs using properties corresponding to possible input constraints. This enables **input constraint inference**.
- We implement SKIPFUZZ, which employs **active learning** for input constraint inference. In this process, fuzzing is guided towards inputs that resemble valid inputs, and reduces redundancy in fuzzing the libraries as it selects inputs to obtain information for refining its model of the input constraints.
- In our experiments, SKIPFUZZ outperforms prior works in finding crashes by generating fewer redundant inputs. From the new crashes found, 23 CVEs have been assigned.

TABLE 1: Glossary

Active Learning: An algorithm that learns by interactively querying an oracle.
Consistency: The extent to which executions under the inferred hypothesis matches the actual program.
Input constraints: The validation checks performed by the library on its inputs.
Input properties: Predicates which describe inputs.
Input categories: Conjunction of input properties.
Hypothesis: A model of the input constraints as inferred by SKIPFUZZ. A disjunction of the properties associated with a set of input categories.

The rest of the paper is structured as follows. Section 2 discusses the background of our work. Section 3 presents SKIPFUZZ. Section 4 describes its implementation details. Section 5 analyzes the experimental results. Section 6 provides more discussion of our results and threats to validity. Section 7 presents related work. Section 8 concludes the paper and mentions future work.

2 PRELIMINARIES

2.1 Deep Learning Libraries

Deep learning libraries are employed by deep learning systems. Library vulnerabilities widen the attack surface of the software systems that depend on them [35]. They may, for example, allow denial-of-service attacks on software systems using them [5].

Input domain. Inputs to deep learning libraries includes tensors and matrices. Library functions may impose input constraints, for example, requiring tensors of a specific size (e.g. a 3x3 matrix). Prior work categorized the input constraints of the library functions by their *structure* (e.g., a list), *type* (e.g., tensor with 'float' values), *shape* (e.g., a 2-d tensor), and valid *values* (e.g., positive integers) [41].

Fuzzing. For effective fuzzing, a fuzzer should not select inputs that fail the input validation checks of library functions (i.e., it should select inputs that are valid, or resemble valid inputs) [27], [41]. To address this, `DocTer` [41] exploits the libraries' consistently structured documentation to extract input constraints such that the functions can be automatically invoked. `DeepRel` [11] and `FreeFuzz` [40] use seed inputs collected from publicly available resource, including the library developers' test suites. `FreeFuzz` invokes library functions for which a valid invocation was observed from the resources. Building on `FreeFuzz`, `DeepRel` selects valid inputs for functions without seed inputs to transfer inputs from test cases between pairs of similar functions. The existing approaches propose strategies for selecting valid inputs, but may miss invalid inputs that lead to bugs and do not have a method to reduce input redundancy.

2.2 Active Learning

We apply active learning for input constraint inference. Table 1 presents a glossary of terms used in the active

TABLE 2: Examples of input property templates. X refers to the input. C1 and C2 refer to constant values, which will be replaced with concrete values to instantiate a property. SKIPFUZZ uses a total of 92 property templates, which can be viewed on the artifact website [4].

Property Group	Example	Description
Type/Structure	isinstance(X, type)	type of the input (e.g. a list).
	X.dtype = type	type (e.g. int) of elements in a tensor.
Value	X < C1	ranges of values.
	all(X > C1)	ranges of values of elements in a tensor/data structure.
	X[C1] = C2	value of an element.
Shape	len(X) < C1	length/size of a data structure.
	X.shape.rank > C1	rank of a matrix.
	X.shape[C1] == C2	size of a specific dimension.

learning phase of SKIPFUZZ. To infer and refine a model of a library function's input constraints, our approach selects inputs that provide new information when they are used to invoke a function.

In active learning [6], [7], a learner sends queries to an oracle who responds with some feedback (e.g., the ground truth label of a given data instance). When active learning is employed for inferring a model of a program, a *hypothesis* is a possible model. A hypothesis is *consistent* if the behavior expected from the model matches the actual behavior of the program.

Input constraint inference. Our approach, SKIPFUZZ, uses active learning for inferring models of the *input constraints* of the functions in the deep learning libraries' API. Input constraints refer to the conditions on the inputs that are expected to be fulfilled for the function to be successfully invoked. Code in the library typically performs validation checks on the inputs, ensuring that the constraints are satisfied before executing the libraries' core functionality. SKIPFUZZ refines a *hypothesis* of the functions' input constraints as its active learner component poses *queries* to the test executor. The test executor determines the answers to the queries by checking if an input with properties corresponding to the query satisfies the input constraint (i.e., if it is *valid*, *invalid*, *crash*) determined by observing if the function invocation completes without error (*valid*), rejects the input through an exception (*invalid*), or crashes the program (*crash*). A crashing input causes the library to terminate in an unclean manner (e.g. a segmentation fault from an illegal memory access).

2.3 Categorizing inputs by their properties

SKIPFUZZ characterizes inputs to the deep learning libraries by *input properties*. The properties, manually designed based on the findings of prior work [41], are descriptions of inputs for distinguishing them. Some examples of the property templates are given in Table 2. From the templates, SKIPFUZZ synthesizes properties, including those considered in the prior work [41].

Each input can satisfy multiple input properties. SKIPFUZZ characterizes each input with the properties that it satisfies. As a pre-processing step of fuzzing, SKIPFUZZ groups inputs that satisfy the same properties into the same *input category*. An input category is associated with the inputs that satisfy a conjunction of properties.

```
# generate inputs
input1 = tf.constant([1, 2, 3])
shape = tf.constant(dtype=tf.qint8,
    value=np.array([1]))
# invoke the target function
tf.placeholder_with_default(input1, shape)
```

Fig. 1: Example of a test case produced for `placeholder_with_default`. Inputs for each argument (e.g. `shape`) are selected.

We assume that the *true* input constraints of the function correspond to a set of input categories, i.e., a collection of properties describing valid inputs. For example, all inputs in the category associated with { `isinstance(X, Tensor)`, `X.shape = (2,2)` } are tensors of the same shape and will satisfy input constraints requiring tensors of this shape. The execution of multiple test cases selecting inputs from different categories provides information about the function's true input constraints. A valid input shows that its properties satisfy the true input constraints, while an invalid input shows that its properties do not satisfy the input constraints.

SKIPFUZZ assumes that inputs with the same properties (i.e., in the same category) would satisfy the same input constraints; if an input fails a given validation check, then the other inputs with the same properties (i.e., in the same category) will fail the same check.

2.4 Motivating Example

Figure 1 shows an example of a test case produced for a function, `tf.placeholder_with_default`. To select valid inputs that satisfy the function's input constraints, the inputs should have the right type, shape, and range of values. If `shape` is a list, there are other constraints such as the type or range of values of its elements. If provided an *invalid* input that does not satisfy these constraints, the library throws an exception.

To discover its input constraints, SKIPFUZZ has to invoke the function multiple times with different values of `shape` and observe their outcomes. A successful invocation indicates that the selected input satisfies the input constraints, and an unsuccessful invocation (i.e., the library throws an exception) indicates otherwise. All inputs with the same wrong shape will fail the same validation check on the input shape, resulting in the same exception. Thus, inputs with the same wrong shape are redundant as they provide no new information for learning the input constraints. SKIPFUZZ skips past the inputs in the same category to inputs from other categories, invoking the function with inputs that match different properties.

After observing several outcomes of invoking the function, SKIPFUZZ forms a hypothesis regarding the constraints of `shape`. A hypothesis is expressed as a disjunction of properties to capture input constraints that are a union of

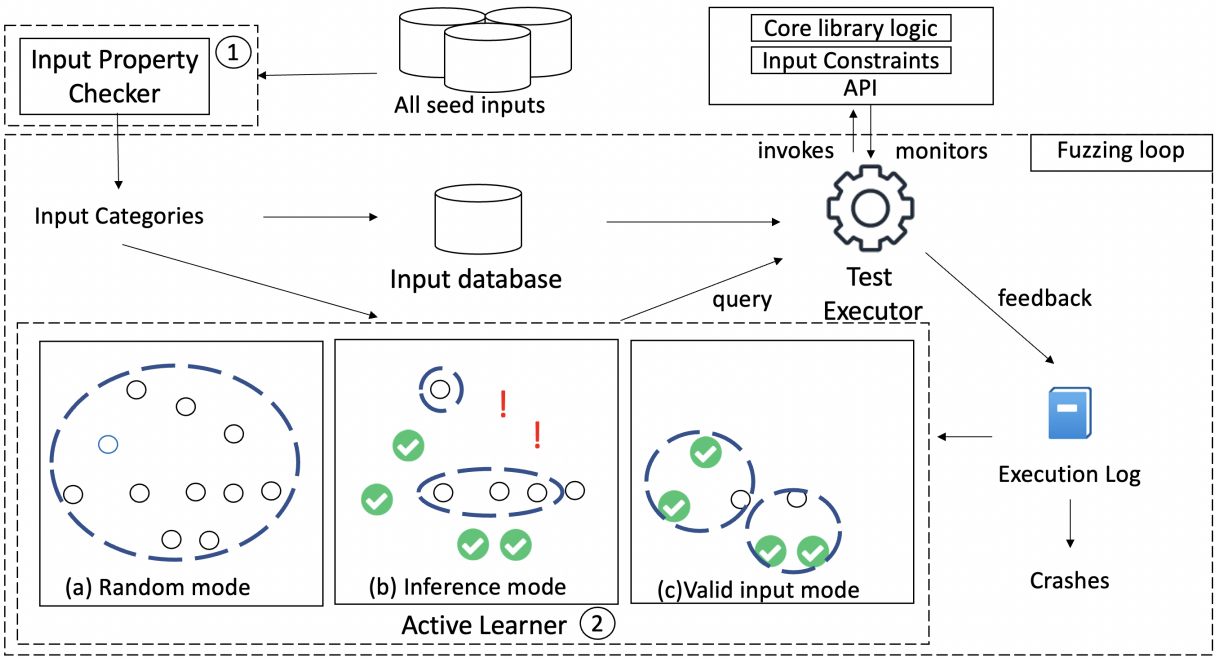

Fig. 2: Overview of SkipFuzz. Step (1): Seed inputs (e.g. collected from the library's test suite) are grouped into input categories. Step (2): As the library is fuzzed, the active learner has 3 modes that determine what queries are posed to the test executor. It selects input categories from the input space. Initially, in its (a) *random generation* mode, it randomly selects input categories (denoted as circles). Next, in its (b) *inference* mode, it selects input categories to refine its hypothesis of the input constraints, which are formed based on the execution log which indicates the outcomes (check marks denote *valid* inputs and exclamation marks (!) denote *invalid* inputs) of prior queries. Finally, once a hypothesized input constraint is accepted, in its (c) *valid input generation* mode, it selects only inputs satisfying the hypothesis. Each blue, dashed ellipse shows the narrowing space of categories considered in each mode.

constraints. The actual constraints of the `shape` parameter permits inputs typed `list` or `TensorShape`. SkipFuzz expresses different constraints for inputs typed `list` and `TensorShape`, treating them as two input categories.

Once SkipFuzz finds its model of the input constraints sufficiently consistent, it stops refining its model. Then, SkipFuzz selects inputs expected to be *valid*, i.e., invoking the function without error, by sampling inputs following its model. This allows SkipFuzz to select inputs that pass the input validation checks.

In the example in Figure 1, if `shape` is a quantized tensor, then the library's kernel code accesses an illegal memory location and triggers a segmentation fault. In other words, a quantized `shape` is a crashing input. SkipFuzz discovers this crash when refining its model of the function's input constraints as the active learner steers fuzzing away from inputs that it is certain is invalid.

3 SkipFuzz

3.1 Overview

Figure 2 shows the overview of SkipFuzz. In the first step ((1) in Figure 2), SkipFuzz collects inputs from the execution of the library's test suite and associates them with properties that they satisfy (Section 3.2). Then, each input is grouped into input categories with other inputs satisfying the same properties. These inputs form the input space considered by SkipFuzz. In the second step ((2) in Figure 2), SkipFuzz fuzzes the deep learning libraries. This involves the generation of test cases by selecting inputs to use as arguments in invoking the API functions. The selection of inputs involves an active learning algorithm that infers the input constraints of a target API function. The active learner constructs queries to check if an input category is a member of the input domain, i.e., its inputs are valid. The test executor has the role of the oracle; to respond to the query, it invokes the library with appropriate inputs sampled from the queried category, checking if they satisfy the actual input constraints (i.e., the function invocation does not lead to an exception or crash).

The test executor constructs test cases by sampling inputs associated with the target input categories. As it constructs and executes a test program, the invocation of the library is monitored for *crashes* (errors in the C++ code of the libraries that may be exploited by an attacker, e.g., segmentation faults) and exceptions thrown by the library are caught. The observation (i.e., query and the outcome of the test execution, *valid*, *invalid*, or *crash*) is written to the execution log. Considering these observations, the active learner refines its hypothesis and constructs more queries.

During fuzzing, SkipFuzz employs active learning to learn the input constraints of the API function. The fuzzing loop involves an active learner and a test executor. The active learner maintains a hypothesis of the input constraints of a given API function. To check the hypothesis, it passes

queries to the test executor. Each query is one input category. On receiving the query, the test executor samples an input that satisfies the input category and constructs a Python program that invokes a function from the library's API. Each constructed Python program consists of code that selects the inputs (e.g., the variable, `shape`, in Figure 1) using program fragments (e.g., invocation of `tf.constant`) collected from the developer test suite. After the inputs are selected, they are passed as arguments to the function under test (e.g., `tf.placeholder_with_default`).

When fuzzing each target function, there are three phases (described in detail in Section 3.3). Initially, as there is no history to support a hypothesis, SKIPFUZZ's randomly selects input categories from the entire input space ((a) in Figure 2). Afterward, the active learner begins to pose queries to the test executor for input constraint inference (described in Section 3.4). These queries are selected based on the hypotheses ((b) in Figure 2). Each query corresponds to one input category. Finally, once an adequately consistent hypothesis is found, then SKIPFUZZ selects only inputs that satisfy the input constraints indicated by the hypothesis ((c) in Figure 2).

SKIPFUZZ maintains a list of crashing test cases. After SKIPFUZZ is terminated, the crashes are output for further inspection.

3.2 Step 1: Input property checking and input category construction

SKIPFUZZ requires seed inputs before it begins categorizing them. In our experiments, we use the developer test suite, which is readily available from the deep learning libraries' repositories, and execute them to obtain seed inputs. SKIPFUZZ instruments the API functions. As the test cases are executed, the inputs passed as arguments to the functions of the APIs are traced and recorded.

SKIPFUZZ enumerates the possible properties for each recorded input, checking if the input satisfies each input property. SKIPFUZZ categorizes the inputs by the properties they satisfy. A mapping from the categories to their inputs is maintained by SKIPFUZZ for efficient sampling of the inputs.

SKIPFUZZ leverages the input categories to reduce redundancy. As inputs from the same categories share the same properties, they will satisfy the same input constraints corresponding to these properties. By selecting inputs from different categories, SKIPFUZZ aims to avoid constructing multiple test cases with similar inputs that fail the same validation checks, as it does not gain information necessary for refining its hypotheses. By avoiding the use of inputs that are similar to previously selected inputs, each test case is more likely to provide new information about the true input constraints. Hence, using inputs from different categories leads to the use of fewer redundant inputs.

3.3 Step 2: Active Learning-driven fuzzing

In the second step, SKIPFUZZ begins fuzzing the deep learning libraries. This is done through three phases.

(a) Random inputs generation. SKIPFUZZ begins generating test cases for each target function by selecting inputs from random categories. This phase ends once SKIPFUZZ successfully produces a test case with valid inputs (i.e., the function executes without error using the inputs).

Algorithm 1: The fuzzing loop of SKIPFUZZ which involves the active learner posing queries to the test executor.

```
   function fuzz(input_categories):
1  history = []
2  while not done do
3  |   selected_category ← active_learner(input_categories,history)
4  |   if selected_category == null then
5  |   |   random_category = sample(input_categories)
6  |   |   input ← sample(random_category)
7  |   else
8  |   |   input ← sample(selected_category)
9  |   end
10 |   tc ← construct_test_case(input)
11 |   outcome ← execute(tc)
12 |   update(history, selected_category, outcome)
13 end
```

(b) Input constraint inference. Once a valid input has been identified, SKIPFUZZ is able to form hypotheses of the input constraints (later described in Section 3.4). SKIPFUZZ tests the hypothesis that is most consistent with the observations by selecting queries based on the hypothesis. It selects input categories from which inputs should be valid according to the hypothesis, as well as categories from which invalid inputs should be produced. Through interacting with the test executor, the active learner refines the hypothesis.

The active learner forms hypotheses of the correct input constraint, assessing them by quantitative measures of consistency. These measures are computed using the number of observed valid and invalid inputs that correctly and incorrectly satisfy the hypothesized input constraints.

(c) Valid input generation. Once SKIPFUZZ considers a hypothesis adequately consistent, SKIPFUZZ begins to construct test cases with inputs that are valid according to the hypothesized input constraints. This is done by sampling inputs from the input categories that are part of the hypothesis.

The procedure for selecting one argument given an API function is given in Algorithm 1. Initially, SKIPFUZZ begins the fuzzing campaign with purely random inputs as the active learner is not able to construct queries without previously observed executions (lines 4–6). After one valid input is observed, the active learner begins to pose queries to the test executor, which constructs test cases based on the queries (line 3). When SKIPFUZZ has entered its valid input generation mode, the active learning only poses queries to guide the selection of inputs that are expected to be valid according to the hypothesis. Given the input category in a query, the fuzzer selects a random input that is associated with the input category (line 8). With the selected input, a test case is constructed and then executed to invoke the library (lines 10–11). The outcome of the test execution is written to the execution log (line 12), $history$, which is used

in the next iteration by the active learner to pose a new query.

3.4 Input constraint inference

The key novelty of SKIPFUZZ is that it selects inputs used for fuzzing to gain information to infer the input constraints (in (b) of step (2) in Figure 2). Through the interaction of the active learner with a test executor, the active learner records the test outcomes in the execution log. These observations guide the formation of hypotheses of the input constraints and for the active learner to pose queries.

Selecting queries based on a hypothesis. We refer again to Table 1, the glossary of terms used in the active learning phase of SKIPFUZZ. The active learner in SKIPFUZZ poses *queries* to the test executor to check if its *hypothesis* of the actual input constraints indeed match the true input constraints. If the hypothesis is a match, then inputs satisfying the hypothesis should be accepted by the library while inputs that do not satisfy the hypothesis should be rejected by the library. Hence, we expect that inputs from the input categories of the hypothesis should lead to *valid* outcomes. Conversely, inputs that are missing at least one property in a category of the hypothesis should be rejected. We expect that these queries should result in *invalid* outcomes. If these queries lead to valid outcomes, then it implies that the hypothesis is less permissive than the true input constraints.

As such, for one hypothesis, the active learner constructs several queries by selecting input categories with respect to the input categories that compose the hypothesis. One set of queries checks that inputs satisfying the hypothesis indeed satisfy the actual input constraints (i.e., the test constructed will be executed without error). Another set of queries checks if the inputs that do not satisfy the hypothesis do not satisfy the input constraints (i.e., the test constructed results in an error when executed).

At the beginning of the fuzzing campaign, SKIPFUZZ selects random inputs. As the fuzzing proceeds, the active learner begins to pose queries to the test executor. The active learner considers the execution log to select input categories to form a hypothesis, and selects input categories as queries. It optimizes for the confirmation of possible hypotheses of the input constraints of the API function.

After the test executor component evaluates a test case, the query (i.e., choice of input category) and test outcome are written in the execution log. If a test case results in an exception thrown by the library, then the input is *invalid*. If the library invocation succeeds without any exceptions, then the input is *valid*. If the test case crashes the library, then the input is a *crash*ing input.

Measuring consistency. At any given time, there may be multiple hypotheses that are considered by the active learner. The active learner selects the hypothesis that is the most consistent with the observations in the execution log. To do so, it quantitatively measures the proportion of valid and invalid inputs that are consistent with the hypothesis. Given a perfectly consistent hypothesis, all valid inputs will be included in an input category in the hypothesis. Conversely, all invalid inputs should not satisfy the hypothesized input constraints.

Within SKIPFUZZ, we do not expect that the hypothesis will perfectly match the input constraints. We compute quantitative measures of a hypothesis' *consistency*. Each hypothesis proposed by the active learner is assessed on its consistency with regard to the observations in the execution log; valid inputs should satisfy the input constraints in the hypothesis and invalid inputs should not. SKIPFUZZ assesses each hypothesis and selects one that is the most consistent with the observed executions. A good hypothesis includes input constraints that cover a large part, if not all, of the valid observations. Additionally, it should not incorrectly cover invalid inputs. SKIPFUZZ uses precision and recall to assess the quality of a hypothesis. Out of `all` inputs selected, given that `covered(valid, hypothesis)` represents the number of valid inputs that fall within the hypothesis, and `covered(all, hypothesis)` represents the number of inputs, both valid and invalid, that fall within the hypothesis. The precision, P, and recall, R, are computed as follows:

$$P = \frac{covered(valid, hypothesis)}{covered(all, hypothesis)}$$

$$R = \frac{covered(valid, hypothesis)}{|valid|}$$

Precision is the proportion of valid inputs that fall within the hypothesized input constraints out of all the observed inputs. Recall is the proportion of valid inputs that fall within the hypothesized input constraints out of all the observed valid inputs. The two metrics measure the adequacy of the hypothesized input constraint. A hypothesis is adequately consistent if the precision and recall exceed a threshold set at the start of the fuzzing campaign.

4 IMPLEMENTATION

In this section, we discuss the implementation of SKIPFUZZ.

Building the input database. SKIPFUZZ takes the API and the developer test suite as its input. We obtain the input values used in the library test suite as the seed inputs for SKIPFUZZ, the Python library code is instrumented to track the invocation of every function call to record their argument inputs. When a function is invoked, SKIPFUZZ records the values of its inputs, and traces backwards through the code in the test case to record how each input was constructed. The functions to construct the inputs, the returned values of their invocations, and the input properties satisfied by the inputs are stored in the database. Subsequently, inputs are selected by fetching and invoking the functions. The library is only instrumented for collecting inputs. Once SKIPFUZZ has constructed the input database, it no longer uses the instrumented library, and fuzzing is performed on the uninstrumented library as the properties of input can be determined without the library.

Crash Oracle. SKIPFUZZ is implemented with a crash oracle. Test programs ran by the test executor are monitored for crashes. Inputs that crash the library are written to the execution log. These crashes are later investigated manually to identify unique crashes before we report them to TensorFlow and PyTorch. Crashes are weaknesses considered as security vulnerabilities [5] (e.g. segmentation faults).

Active Learning. The active learner takes the execution log as input and produces a series of queries to be posed

to the test executor. The queries are constructed based on the subset of input categories in the hypothesis. The construction of a hypothesis and the selection of queries are obtained through the execution of a logic program. Using a logic program allows us to declaratively express the desired characteristics of a hypothesis and optimize the selection of input categories against a criteria. The active learner selects an appropriate hypothesis while maximizing the number of valid inputs that match the hypothesis, minimizing the number of invalid inputs that are incorrectly matched by the hypothesized input constraint, and favouring simpler hypothesis by minimizing the number of input categories used in the hypothesis. In this way, SKIPFUZZ assesses each hypothesis on its consistency with the observed test outcomes.

SKIPFUZZ accepts a hypothesized input constraint considering if its precision and recall exceed a threshold. In our experiments, we set a low threshold for both precision and recall at 0.25. This enables the input constraints to be inferred for a large proportion of the API. As our goal is to fuzz the API thoroughly, we find allowing the fuzzer to focus on a broad region of inputs that include the valid domain of inputs of the functions is more beneficial than precisely identifying the valid domain of inputs. The low thresholds set at this stage do not adversely impact the proportion of valid inputs selected by SKIPFUZZ when using the hypothesized input constraints to select valid inputs. This is because the logic program already optimizes the selection of hypotheses for a high level of consistency.

Interleaving of target functions. The active learner SKIPFUZZ employs `clingo` [14] to execute the logic programs used to select the next set of inputs. Logic programs take a significant amount of time to be executed to produce their output. To allow time for the logic program to be executed, SKIPFUZZ interleaves the construction of test cases for different API functions, coming back to the same function only after completing a test case for other API functions. This provides ample time for the logic program to be run before the same function is tested again, minimizing the impact of their overhead on the fuzzing process.

5 EVALUATION

5.1 Research Questions

Our experiments aim to answer the following questions.

RQ1. Does SKIPFUZZ produce crashing inputs?

SKIPFUZZ's primary objective is to find crashing inputs. We count the number of new crashes that have not been previously reported, which are reported to the library developers for validation. We also assess if SKIPFUZZ can trigger the crashes found by prior approaches, and if they trigger the crashes found by SKIPFUZZ.

RQ2. Does SKIPFUZZ sample inputs with less redundancy?

Active learning should enable SKIPFUZZ to reduce redundancy during fuzzing by selecting a wide range of input categories. We compare the inputs selected by SKIPFUZZ and the baselines in fuzzing the functions known to crash.

RQ3. Does SKIPFUZZ sample valid inputs after learning an input constraint?

If the input constraints are accurately learned, then the inputs selected by SKIPFUZZ are expected be valid if they satisfy the inferred constraints. We investigate if this is true.

RQ4. Which components of SKIPFUZZ contribute to its ability to find crashing inputs?

SKIPFUZZ's selection of inputs is less redundant, and the active learning steers guides input selection towards valid inputs. We perform an ablation study to determine the contributions of SKIPFUZZ's components.

5.2 Experimental Setup

Baselines. We compare SKIPFUZZ against the fuzzers that selects inputs from a closed set, `DeepRel` [11] and `DocTer` [41]. From their replication packages, we run the tools and analyze the list of reported bugs.

`DeepRel` builds on top of `FreeFuzz` [40], using the same strategy of selecting inputs for each function. `DeepRel` and `FreeFuzz` use inputs collected from resources such as publicly available models and the library test suite. `DeepRel` improves over `FreeFuzz` by identifying pairs of similar functions, allowing inputs known to be valid for one function to be shared and used when fuzzing the other function. As `DeepRel` and `FreeFuzz` uses the same input generation strategy, we only compare SKIPFUZZ against `DeepRel`.

`DocTer` extracts input constraints from the library documentation. Then, it selects inputs from a collection of manually constructed inputs to invoke the libraries considering the extracted input constraints.

Environment. We run experiments on TensorFlow 2.7.0 and PyTorch 1.10, the same version of the libraries used in the experiments of DeepRel [11]. We collect a list of all APIs of TensorFlow and PyTorch. It is used in our initial experiments, where we run the approaches on every function. Subsequently, we focus our analysis on the ability of the fuzzers to find inputs that trigger the crashes found by the other approaches.

We configured and ran the fuzzers for up to 48 hours. In the prior experiments of the baseline fuzzers [11], [40], [41], the tools were allowed up to 1,000 [11], [40] or 2,000 [41] executions for each function. To generate 1,000 test cases, we executed `DeepRel` and it took 172 hours and 43 hours to complete generating test cases for TensorFlow and PyTorch. `DocTer` took 16 hours for TensorFlow and 25 hours on PyTorch. Therefore, to use the same budget for a fair comparison, we tweaked the number of test cases generated by the baseline fuzzers to fit in 48 hours and reran the fuzzers. Apart from rerunning the tools, we also considered the original set of bugs reported, linked from their replication packages.

Our experiments used a machine with an Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz, 205G, Tesla P100. While our fuzzer does not directly use the GPU, some library functions may use the GPU.

5.3 Experimental Results

5.3.1 RQ1. Crashes detected

Existing crashes. We perform an analysis of the capability of the approaches in detecting existing crashes. In this analysis,

TABLE 3: The number of unique new vulnerabilities of TensorFlow reported in this study and prior studies.

Approach	# new vulnerabilities
DocTer	1
FreeFuzz	7
DeepRel	1
SKIPFUZZ	23

we consider all crashes found by the approaches. To perform this analysis, we consider that a crash was not detected if its corresponding API is not covered by the tool, if the tool does not report the bug although test cases were generated for the function, and if the the set of bugs reported linked from their replication packages does not include the vulnerability. Within the scope of this study, we consider only bugs that led to crashes, as we wanted to prevent overwhelming the library developers. Although prior studies have reported other types of bugs, bugs found using other test oracles may not always be appreciated by the library developers [2].

SKIPFUZZ detects a total of 166 crashing functions, 108 in TensorFlow version 2.7.0 and 58 in PyTorch version 1.10. From the 108 TensorFlow functions, we grouped related crashes and reported 43 vulnerabilities. From the 58 PyTorch functions, we reported 10 vulnerabilities. After corresponding with the library developers, they confirmed that 23 of the TensorFlow vulnerabilities and 5 of the PyTorch vulnerabilities were previously unknown. The remaining crashes were confirmed as vulnerabilities, but they were already known by the developers (although the fixes were not released yet). From these reports, 23 CVEs were assigned.

Next, we analyze the extent to which SKIPFUZZ, `DocTer`, and `DeepRel` detect the same vulnerabilities. Of the 108 vulnerable TensorFlow and 58 vulnerable PyTorch functions found by SKIPFUZZ, `DocTer` was able to successfully selected crashing inputs to 6 of the 108 vulnerable functions in TensorFlow and 12 of the 58 vulnerable functions in PyTorch. Overall, `DocTer` detects just 18 (18/166, or 11%) of the 166 vulnerable functions detected by SKIPFUZZ.

Next, we investigate the crashes found by DocTer, provided in its replication package. DocTer found bugs in 174 API functions (including both crashes and documentation errors), and of these 174 functions, 120 of them lead to crashes. When executed on the versions of libraries before these crashes were fixed, SKIPFUZZ is able to detect 88 (73%) out of the 120 crashing functions in TensorFlow detected by `DocTer`. On PyTorch, SKIPFUZZ is able to detect 7 (23%) out of the 31 crashing functions detected by `DocTer`. Overall, SKIPFUZZ detects 95 (95/151, or 63%) out of 151 crashing functions detected by `DocTer`.

Next, we compare SKIPFUZZ against `DeepRel` and `FreeFuzz`. Among the crashing functions found by SKIPFUZZ, we found that `DeepRel` and `FreeFuzz` was able to detect crashes for only 9 of the 108 TensorFlow functions and 7 of the 58 PyTorch functions (16/166, 10%). We investigate the bugs found by DeepRel and FreeFuzz. The original experiments done to evaluate `FreeFuzz` [40] and `DeepRel` [11] resulted in 39 bug reports on TensorFlow, of which 10 involved crashes, and 72 bug reports on PyTorch, of which 7 involved crashes. A total of 17 crashing functions were found in their original experiments. When executed on versions of the libraries before the crashes were fixed, SKIPFUZZ detects 8 (80%) of the 10 crashes on TensorFlow and 3 (43%) of the 7 crashes on PyTorch (11/17, or 65%).

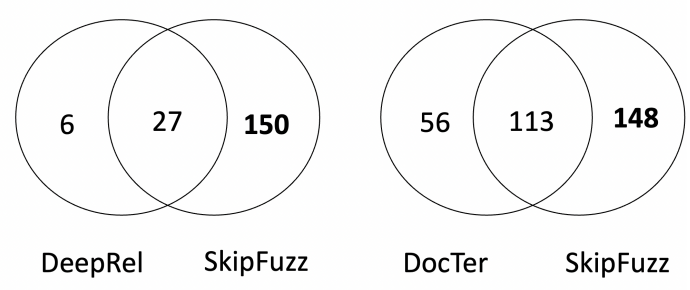

Fig. 3: Crashes found by the fuzzers. SKIPFUZZ detects 95/151 crashing functions reported by `DocTer`, and `DocTer` detects 18/166 crashing functions by SKIPFUZZ. SKIPFUZZ detects 11/17 crashing functions reported by `DeepRel`, and `DeepRel` detects 16/166 crashing functions found by SKIPFUZZ.

Figure 3 shows two Venn diagrams summarizing the number of functions that each approach produces crashing inputs to. Out of the 317 crashing functions (166 crashing functions found by SKIPFUZZ and 174 crashing functions found by `DocTer`), there are 113 crashes found by both SKIPFUZZ and `DocTer`. 148 crashing functions were found only by SKIPFUZZ, and 56 were found only by `DocTer`. Out of 183 crashing functions (166 by SKIPFUZZ, 17 by `DeepRel/FreeFuzz`), 133 crashing functions were found by both approaches. 150 crashing functions were found only by SKIPFUZZ, and 6 were found only by `DeepRel`.

New vulnerabilities in TensorFlow. Table 3 shows the number of new crashes found in the experiments of `DocTer`, `FreeFuzz`, and `DeepRel`. On TensorFlow, we determine if a crash is new by going through the list of TensorFlow vulnerability reports and comparing the referenced bug reports against the bug reports referenced by the replication packages of the prior approaches. Of the several hundred crashing functions detected by SKIPFUZZ, we analyzed the crashes, minimizing the test cases and grouping crashes we suspected had the same root cause. Then, we reported the unique crashes to reduce the workload of the library developers. Finally, we reported 33 vulnerabilities. 23 CVE IDs assigned were assigned in TensorFlow. In comparison, `DocTer` [41] found 1 newly discovered vulnerability. `FreeFuzz` [40] and `DeepRel` [11] found a total of 8 crashes. We do not perform this analysis for PyTorch as its developers do not assign CVEs to potential security weaknesses. However, we note that the PyTorch developers have fixed the 6 crashes that we have reported.

Apart from the baseline approaches discussed, TensorFlow is a fuzz target in the OSS-Fuzz project [3], which employs coverage-guided fuzzers. OSS-Fuzz has found over 30,000 bugs in open source projects, but only 6 security bugs in TensorFlow[1]. Evidently, finding new vulnerabilities is challenging.

1. Issues tagged "Bug-Security" on https://bugs.chromium.org/p/oss-fuzz/issues/list?sort=-opened&can=1&q=proj:TensorFlow

TABLE 4: Coverage of input properties

Approach	% input properties covered
DocTer	15%
DeepRel	16%
SKIPFUZZ	**31%**

Answer to RQ1: SKIPFUZZ finds new crashes, with 23 CVEs assigned. Up to 90% of the crashes were not found by the baselines. SKIPFUZZ detects up to 63% of crashes found by the baselines.

5.3.2 RQ2. Reducing input redundancy.

We investigate the selection of inputs by SKIPFUZZ, First, we assess if selecting inputs sharing the same properties as a previously used input is indeed redundant. Then, we evaluate the reduction in redundancy by measuring how many input properties match at least one input during fuzzing. A greater coverage of input properties imply less redundancy.

Same properties-same outcome assumption. SKIPFUZZ relies on the assumption that inputs matching the same properties execute the same behavior, therefore, they are redundant. We assess this assumption by counting how often using inputs from the same input category result in the same outcome. If the assumption holds, then the vast majority of inputs of the same category should result in the same outcome. Otherwise, they would lead to different outcomes. We analyze the generated test cases. On average, across all API functions, when inputs shared the same input properties, the invocation of the function resulted in the same outcome (e.g., same type of exception thrown) 96.3% of the time. This result suggest that our assumption is appropriate and can be adopted to reduce input redundancy.

Input Redundancy. We investigate the number of properties that were satisfied by an input passed to the libraries as a proportion of all input properties observed in the experiments. A higher coverage of input properties indicates a greater diversity of inputs and a low amount of redundancy in input generation. Conversely, a low coverage may indicate that similar inputs was selected over and over again, which implies a high level of redundancy as the inputs may be triggering the same library behavior (e.g., failing the same validation checks). We focus on the test cases generated to target the crashing functions to investigate the reason for SKIPFUZZ's stronger ability to select crashing inputs.

Table 4 shows the experimental results. The inputs used by SKIPFUZZ in its test cases cover two times more input properties than the inputs used in the test cases generated by DeepRel and DocTer. While the inputs selected by DeepRel and DocTer cover 16% and 15% of the possible input properties, SKIPFUZZ achieves an average of 37% property coverage. This confirms that SKIPFUZZ was able to select a wide range of inputs that were less likely to result in the same outcome.

Answer to RQ2: Yes, SKIPFUZZ covers more input properties than prior approaches.

TABLE 5: Proportion of valid inputs selected

Approach	% of valid test cases
Random selection of inputs	1%
DocTer	13%
SKIPFUZZ	24%
DeepRel	77%
SKIPFUZZ (valid input mode)	**80%**

5.3.3 RQ3. Generating valid inputs

SKIPFUZZ performs input constraint inference. We study if the inferred input constraints are precise enough for producing valid inputs.

DocTer selects just valid inputs 13% of the time, underperforming SKIPFUZZ which produces valid inputs 24% of the time (and 80% after inferring the input constraints). This validates our initial intuition for using active learning. The better performance of SKIPFUZZ in generating valid inputs indicates that active learning may be more successful in inferring input constraints than DocTer's use of the API documentation, which may be incomplete [41]. The improvements of SKIPFUZZ over DocTer shows that the selection of inputs should use information about both the validity and the redundancy of the inputs.

DeepRel uses FreeFuzz as its test generator, and therefore, will produce the same output as FreeFuzz. Their input generation strategy is to select a different input with the same type and value from the seed test cases. Through this strategy of narrowing down the choices of inputs, the vast majority of inputs selected by FreeFuzz and DeepRel are valid. However, as discussed in the previous section, a large proportion of valid inputs may imply that the fuzzer is using similar inputs repeatedly, leading to redundancies. From Table 4, DeepRel covers fewer input properties, which may have led to a lower chance of generating crashing inputs (Table 3). In contrast, SKIPFUZZ is able to better explore the space of invalid inputs and find crashing inputs.

Table 5 shows the proportion of valid inputs selected. We compare SKIPFUZZ against DocTer as well as a simple baseline that randomly selects inputs used in the libraries' test suite. While SKIPFUZZ selects valid inputs 24% of the time considering all three input generation modes, SKIPFUZZ produces valid inputs 80% of the time in its valid input generation mode (after inferring the input constraints). This is higher than the proportion of valid inputs selected by both DocTer and DeepRel, demonstrating the benefit of the active learning for input constraint inference

Answer to RQ3: Yes, 80% of inputs selected by SKIPFUZZ are valid after inferring an input constraint. This is a 6x improvement over DocTer, which extracts constraints from documentation.

5.3.4 RQ4. Ablation analysis

We perform an ablation study on SKIPFUZZ. SKIPFUZZ^{-} refers to a version of SKIPFUZZ where inputs are sampled from the input categories, but there is no active learner posing queries and no input constraint inference (removing (2) in Figure 2). SKIPFUZZ^{--} refer to a version of SKIPFUZZ

TABLE 6: Ablation analysis of the components in SKIPFUZZ. % valid is the proportion of valid inputs that are selected. SKIPFUZZ$^-$ removes active learning. SKIPFUZZ^{--} removes the use of input properties and active learning.

Approach	Property coverage	% valid	# crashes
SKIPFUZZ	31%	**24%**	**168**
SKIPFUZZ$^-$	**93%**	1%	112
SKIPFUZZ^{--}	84%	1%	52

where inputs are selected randomly (removing both (1) and (2) in Figure 2).

Table 6 shows the results of the ablation analysis. Without using active learning to infer input constraints, the number of crashes found by SKIPFUZZ$^-$ drops from 168 to 122, a 26% decline. Without using active learning, SKIPFUZZ$^-$ does not drive the test executor toward valid inputs. While it covers a higher proportion of properties (93%), it never steers fuzzing towards valid inputs. The low proportion of valid inputs generated (1%) suggests that many crashes are not discovered through the generation of valid inputs. SKIPFUZZ is able to find many crashes because they are discovered through the selection of invalid inputs that resemble valid inputs such that they pass the input validation checks. During fuzzing, SKIPFUZZ selects these inputs as they are required to refine SKIPFUZZ's model of input constraints.

Without the input properties, SKIPFUZZ^{--} selects inputs entirely at random. The number of detected crashes substantially drops to just 52, less than half found when skipping over redundant inputs, and one third of the total crashes found by SKIPFUZZ. Overall, both a lack of input redundancy and the active learning-guided selection of inputs are essential to finding crashing inputs.

Answer to RQ4: Our experimental results suggest that the input properties are essential to SKIPFUZZ and that active learning substantially boosts SKIPFUZZ's effectiveness. The consideration of input properties reduces redundancy (and finds 2x more crashes). The use of active learning allows the discovery of crashing inputs that are invalid but pass input validation checks (and finds 3x more crashes over the baseline fuzzer).

6 Discussion and Threats to Validity

Our experiments demonstrate that SKIPFUZZ outperforms prior approaches in generating crashing inputs to TensorFlow and PyTorch. Our analysis suggests that SKIPFUZZ is effective due to reduced redundancy in its selected inputs, which stem from the use of active learning in input constraint inference and avoiding the use of inputs with the same set of input properties.

Inferred Input Constraints. Compared to the constraints indicated in the documentation, the constraints learned through active learning are more permissive. The input constraints can be compared by the proportion of selected valid inputs. The constraints inferred by SKIPFUZZ enable 80% of selected inputs to be valid. In prior work [41], input constraints extracted from documentation enable only 33% of selected inputs to be valid, which suggests that the documentation does not provide a complete set of constraints.

Limitations of generating only valid inputs. Prior approaches had strategies for finding valid inputs, but our experiments suggest that a fuzzer generating only valid inputs would not detect a large number of crashes. Even without active learning and the input properties, SKIPFUZZ outperforms both DocTer [41] and DeepRel [11] in finding crashing inputs. Our experiments imply that fuzzers should not omit the selection of invalid inputs. Both the generation of valid and invalid inputs are important.

Reducing input redundancy. While DocTer [41] could also generate invalid inputs that did not match the constraints inferred from documentation, it did not have a systematic strategy of preventing the selection of redundant inputs. In comparison, SKIPFUZZ's use of input properties led to a lower input redundancy, allowing the selection of inputs that passed input validation checks, but were not correctly handled by the library.

Threats to construct validity include threats related to the experimental metrics. We use standard metrics from prior studies [11], such as API coverage. Moreover, we assess SKIPFUZZ on the discovered vulnerabilities, which were confirmed by library developers. Hence, there are minimal threats to construct validity.

Threats to external validity are concerned with the generalizability of our approach. Our experiments focused on TensorFlow and PyTorch, which are the most commonly-used deep learning libraries. As the library developers have indicated that assumptions of differential testing approaches may not always be valid [2], our work focuses on finding crashes as they are considered more important by the developers (e.g. they may assign CVEs to the crash). Hence, there are minimal threats to external validity.

7 Related Work

Bugs in deep learning libraries. Prior studies [20], [21], [22] have investigated bugs in deep learning programs. Jia et al. [21] reported that common root causes of bugs within TensorFlow include *type confusion* (incorrect assumptions about a variable type), *dimension mismatches* (inadequate checks of a variable's shape), and unhandled *corner cases* (usually related to incorrect handling of a specific variable's value, e.g. division by zero errors). The root causes match the input properties considered by SKIPFUZZ.

Fuzzing deep learning models and systems. Researchers have proposed approaches fuzz either deep learning models [12], [16], [42] or larger systems that use deep learning [8], [13], [18], [34], [43], [46]. Other approaches use static analysis [24], [26]. SKIPFUZZ fuzzes deep learning libraries rather than individual models or systems.

Fuzzing deep learning libraries. Several works [15], [17], [29], [37], [39] generate or mutate deep learning models for testing deep learning libraries. Subsequently, the experiments of `FreeFuzz` [40] showed that API-level testing of deep learning libraries is more effective. Some approaches perform metamorphic and differential testing [17], [29], [37], [39], [42]. Predoo targets precision errors in TensorFlow [45].

ExAIS [30] uses specifications of the deep learning layers for fuzzing. As it requires manual analysis, its scalability is limited. These studies overlook the systematic selection of inputs for minimizing redundancy. The closest approaches to SKIPFUZZ are `DocTer` [41] and `DeepRel` [11], which have been discussed and used in our experiments.

Input validation. SKIPFUZZ addresses the problem of generating valid inputs through input constraint inference. Several approaches [10], [19], [25], [28], [38] use static analysis to address the problem. DriFuzz [33] proposes a method of generating high-quality initial seed inputs. Unlike these approaches, SKIPFUZZ uses active learning to learn the input constraints to generate valid inputs.

Active Learning. Our approach uses active learning [6], [7], [9], [31], which queries an oracle and learns from its feedback. In classification tasks, active learning is used to query for labels of informative data instance when labeling every instance is too difficult [23], [31], [44]. Recent work uses active learning to learn models of programs, and then regenerate programs using the models to remove undesired behaviors [32], [36]. SKIPFUZZ applies active learning for a novel task of learning models of input constraints that are used in fuzzing.

8 CONCLUSION AND FUTURE WORK

In this study, we propose SKIPFUZZ, an approach for testing deep learning libraries. SKIPFUZZ uses active learning to infer input constraints for the libraries' API. SKIPFUZZ has two advantages over existing approaches. First, it infers the input constraints without the use of documented specifications. Second, it guides the selection of a less redundant set of inputs during fuzzing. Our experiments demonstrate that both the use of the input properties and active learning are important. In future work, we plan to synthesize the property templates based on analysis of the library's source code, rather than manually writing them.

The replication package of SKIPFUZZ is available at https://zenodo.org/record/7600936 and https://github.com/skipfuzz/skipfuzz.

REFERENCES

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[2] TensorFlow issue 58131. https://github.com/tensorflow/tensorflow/issues/58131\#issuecomment-1289406316.

[3] Github repository of OSS-Fuzz - continuous fuzzing for open source software. https://github.com/google/oss-fuzz, 2022. Accessed: 2022-10-10.

[4] SkipFuzz's GitHub repository. https://github.com/skipfuzz/skipfuzz, 2022.

[5] TensorFlow security policy. https://github.com/tensorflow/tensorflow/security/policy, 2022. Accessed: 2022-04-20.

[6] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[7] Dana Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.

[8] Muhammad Hilmi Asyrofi, Zhou Yang, Imam Nur Bani Yusuf, Hong Jin Kang, Ferdian Thung, and David Lo. Biasfinder: Metamorphic test generation to uncover bias for sentiment analysis systems. *IEEE Transactions on Software Engineering (TSE)*, 2021.

[9] José P Cambronero, Thurston HY Dang, Nikos Vasilakis, Jiasi Shen, Jerry Wu, and Martin C Rinard. Active learning for software engineering. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 62–78, 2019.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, pages 2123–2138, 2017.

[11] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. Fuzzing deep-learning libraries via automated relational API inference. In *2022 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*, 2022.

[12] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. DeepStellar: Model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*, pages 477–487, 2019.

[13] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, pages 118–128, 2018.

[14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo. 2008.

[15] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*, pages 1418–1430, 2022.

[16] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang 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 (ESEC/FSE 2018)*, pages 739–743, 2018.

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

[18] Pinjia He, Clara Meister, and Zhendong Su. Testing machine translation via referential transparency. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021)*, pages 410–422. IEEE, 2021.

[19] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, et al. SoFi: Reflection-augmented fuzzing for javascript engines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CSS 2021)*, pages 2229–2242, 2021.

[20] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh 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 (ESEC/FSE 2019)*, pages 510–520, 2019.

[21] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. An empirical study on bugs inside TensorFlow. In *International Conference on Database Systems for Advanced Applications*, pages 604–620. Springer, 2020.

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

[23] Hong Jin Kang and David Lo. Active learning of discriminative subgraph patterns for API misuse detection. *IEEE Transactions on Software Engineering (TSE)*, 48(8):2761–2783, 2021.

[24] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. Static analysis of shape in TensorFlow programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[25] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. FANS: Fuzzing Android native system services via automated interface analysis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 307–323, 2020.

[26] Chen Liu, Jie Lu, Guangwei Li, Ting Yuan, Lian Li, Feng Tan, Jun Yang, Liang You, and Jingling Xue. Detecting TensorFlow program bugs in real-world industrial environment. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*, pages 55–66. IEEE, 2021.

[27] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*, pages 329–340, 2019.

[28] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (S&P 2018)*, pages 697–710. IEEE, 2018.

[29] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019)*, pages 1027–1038. IEEE, 2019.

[30] Richard Schumi and Jun Sun. ExAIS: Executable AI semantics. In *2022 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2022)*, 2022.

[31] Burr Settles. Active learning literature survey. 2009.

[32] Jiasi Shen and Martin C Rinard. Active learning for inference and regeneration of applications that access databases. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(4):1–119, 2021.

[33] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1275–1290, 2022.

[34] Ezekiel Soremekun, Sakshi Sunil Udeshi, and Sudipta Chattopadhyay. Astraea: Grammar-based fairness testing. *IEEE Transactions on Software Engineering (TSE)*, 2022.

[35] Xin Tan, Kai Gao, Minghui Zhou, and Li Zhang. An exploratory study of deep learning supply chain. In *Proceedings of the 44th International Conference on Software Engineering*, pages 86–98, 2022.

[36] Nikos Vasilakis, Achilles Benetopoulos, Shivam Handa, Alizee Schoen, Jiasi Shen, and Martin C Rinard. Supply-chain vulnerability elimination via active learning and regeneration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS 2021)*, pages 1755–1770, 2021.

[37] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. Eagle: Creating equivalent graphs to test deep learning libraries. In *2022 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2022)*, 2022.

[38] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy (S&P*, pages 497–512. IEEE, 2010.

[39] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. Deep learning library testing via effective model generation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, pages 788–799, 2020.

[40] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *2022 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2022)*, 2022.

[41] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W Godfrey. DocTer: documentation-guided fuzzing for testing deep learning api functions. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*, pages 176–188, 2022.

[42] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon 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 (ISSTA 2019)*, pages 146–157, 2019.

[43] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software (JSS)*, 84(4):544–558, 2011.

[44] Xueqi Yang, Zhe Yu, Junjie Wang, and Tim Menzies. Understanding static code warnings: An incremental AI approach. *Expert Systems with Applications*, 167:114134, 2021.

[45] Xufan Zhang, Ning Sun, Chunrong Fang, Jiawei Liu, Jia Liu, Dong Chai, Jiang Wang, and Zhenyu Chen. Predoo: precision testing of deep learning operators. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*, pages 400–412, 2021.

[46] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE 2020)*, pages 347–358. IEEE, 2020.