Detecting TensorFlow Program Bugs in Real-World Industrial Environment

Chen Liu[†‡], Jie Lu[†*], Guangwei Li[†‡], Ting Yuan[†‡], Lian Li[†‡*], Feng Tan[§], Jun Yang[§], Liang You[§], Jingling Xue[°]
[†] *SKL Computer Architecture, ICT, CAS, Beijing, China*
[‡] *University of Chinese Academy of Sciences, China*
[§] *Alibaba Group*
[°] *University of New South Wales, New South Wales, Australia*
[†]{liuchen17z, lujie, liguangwei, yuanting, lianli}@ict.ac.cn
[§]{tanfeng.tf, muzhuo.yj, youliang.yl}@alibaba.com
[°]jingling@cse.unsw.edu.au

***Abstract*—Deep learning has been widely adopted in industry and has achieved great success in a wide range of application areas. Bugs in deep learning programs can cause catastrophic failures, in addition to a serious waste of resources and time.**

This paper aims at detecting industrial TensorFlow program bugs. We report an extensive empirical study on 12,289 failed TensorFlow jobs, showing that existing static tools can effectively detect 72.55% of the top three types of Python bugs in industrial TensorFlow programs. In addition, we propose (for the first time) a constraint-based approach for detecting TensorFlow shape-related errors (one of the most common TensorFlow-specific bugs), together with an associated tool, SHAPETRACER. Our evaluation on a set of 60 industrial TensorFlow programs shows that SHAPETRACER is efficient and effective: it analyzes each program in at most 3 seconds and detects effectively 40 out of 60 industrial TensorFlow program bugs, with no false positives. SHAPETRACER has been deployed in the PLATFORM-X platform and will be released soon.

***Index Terms*—TensorFlow Bugs, Constraint Solving**

I. INTRODUCTION

Deep learning has been widely adopted in industry. Assisted by open-source frameworks [1]–[3], developers can efficiently design new deep learning models for applications in a wide range of areas [4]–[7], such as image recognition, natural language processing, and self-driving cars. To enable developers to test and train their models effectively, enterprises have built dedicated platforms, such as Google Cloud AI [8], Microsoft Azure Machine Learning [9], and Amazon SageMaker [10]. Those platforms are equipped with rich computational resources including GPUs and AI accelerators, running tens of thousands of deep learning jobs every day.

Like other software applications, deep learning programs are often plagued by bugs. In a real-world industrial environment, these bugs often lead to job failures, wasting seriously resources and time. There are a number of studies [11]–[14] targeting deep learning program errors. Specifically, Zhang et al. [12] conducted an extensive empirical study on program failures of deep learning jobs for Microsoft's *Philly* platform. Sifis et al. [14] developed a new static analysis to detect shape errors in TensorFlow programs, which can effectively detect 11 of 14 shape-related TensorFlow bugs studied in [11].

In this paper, we aim at detecting TensorFlow (the dominant open-source deep learning framework) program bugs in a real-world industrial environment. We perform an extensive empirical study on 12,289 failed TensorFlow jobs submitted to the PLATFORM-X platform by teams in Alibaba Group [15]. Compared to [11], [12], our study focuses on job failures due to TensorFlow program bugs, and targets a different industrial platform. Our findings and actions are:

- *Finding*: Most bugs (63.69%) are common Python bugs, with argument mismatches, undefined variables, and missing attributes as the top three types of bugs.

  Action: We deployed four existing representative static tools, Mypy [16], Pylint [17], Pyflakes [18], and Pytype [19], to detect Python bugs in TensorFlow programs. Our results show that these four tools together detect 72.55% of the top three types of Python bugs.

- *Finding*: *Checkpoint Errors* (17.49%) and *Shape Errors* (8.82%) are the two most common types of TensorFlow-specific bugs. As for the former category (triggered by failing to load a checkpoint file), we are not aware of any existing bug-detection techniques reported in the literature. As for the latter category, a static analysis approach [14] exists and will be evaluated in this paper.

  Action: We have developed SHAPETRACER, a new tool for detecting also shape-related errors in real-world TensorFlow applications. In contrast to the static analysis approach, PYTHIA, described in [14], we adopt a constraint-based approach for the first time. SHAPETRACER traverses program paths and builds a *shape-flow* graph (an abstract dataflow computation graph) for each path. A constraint solver is then employed to solve shape-related constraints (introduced by shape operators) for each shape-flow graph. Finally, a bug is reported if the constraint solver cannot find a feasible solution, and a suggestion is offered as a warning if the user input is

*Corresponding authors.

constrained.

Unlike PYTHIA (based on static analysis), SHAPETRACER (based on constraint solving) enables detecting subtle shape-related errors when the *rank* (number of dimensions) or *dimensions* (dimension sizes) of a shape are completely unknown. PYTHIA formulates the problem of detecting shape-related errors as one of inferring the shapes of tensors from a set of datalog rules and its analysis cannot progress unless an unknown shape rank or dimension can be deduced to be a concrete value (or a finite set of concrete values). However, in real-world applications, many unknown shapes cannot be concretized this way, as illustrated by the program given in Figure 1, for which we are required to solve the constraint "*batch_size*32==batch_size* $\vee$ *batch_size==1*", where *batch_size* is provided as user input. In particular, the shapes of tensors that are statically unknown may be provided as user input by reading from the commandline or files, or by calling unmodeled library functions, making PYTHIA often ineffective, as evaluated in Section VI.

An immediate question arises: does SHAPETRACER suffer from path explosion? The answer is no. TensorFlow programs have simple control-flow structures, making such an approach practical and efficient. We have applied SHAPETRACER to a set of 60 real-world buggy TensorFlow applications. Our experimental results show that SHAPETRACER is efficient (by analyzing a program in at most 3 seconds) and effective (by reporting 40 out of 60 bugs) in detecting real-world TensorFlow bugs.

SHAPETRACER, together with four other tools, Mypy [16], Pylint [17], Pyflakes [18], and Pytype [19], have been packaged as a new tool (publicly released soon) for detecting TensorFlow program bugs and deployed to platform users. Developers are recommended to run this packaged tool against their applications before submitting a job to the platform.

In summary, this paper makes the following contributions:

- We report an extensive empirical study on 12,289 industrial TensorFlow job failures. Our findings show that most failure-triggering bugs (63.69%) are Python bugs, and four existing representative static bug-detection tools can detect 72.55% of the top three types of Python bugs.
- We propose the first constraint-based approach for detecting shape-related errors, one of the most common TensorFlow-specific bugs. Our approach explores program paths systematically and can detect subtle errors when the rank or dimensions of a shape are unknown, by solving the shape-related constraints for each path.
- We have implemented our constraint-based approach as a tool, SHAPETRACER, and applied it to a set of 60 real-world buggy TensorFlow applications. SHAPETRACER is highly efficient and effective, by analyzing each application in at most 3 seconds, and detecting 40 out of 60 shape-related errors, with no false positives. We have also compared SHAPETRACER with PYTHIA [14] to demonstrate the effectiveness of our new approach.

The rest of this paper is organized as follows. Section II gives an overview of TensorFlow programs and the PLATFORM-X platform. In Section III, we report an empirical study with 12,289 real-world TensorFlow job failures and motivate this work. Section IV discusses how existing static tools detect Python bugs. Section V introduces SHAPETRACER. In Section VI, we evaluate SHAPETRACER using both open-source and real-world TensorFlow programs. Section VII reviews the related work and Section VIII concludes the paper.

II. BACKGROUND

A. TensorFlow programs

The Google-born TensorFlow library [1] is the dominant open-source deep learning framework. It adopts the *dataflow* programming model, which represents all the computations as dataflow graphs. In a dataflow graph, its nodes are computation units (i.e., operators) and its edges propagate *tensors* (typed multi-dimensional arrays) from their source nodes to their sink nodes. A dataflow graph is executed on the data provided, with the input data flowing along its edges, which are processed by each node before, and the output results finally produced.

A TensorFlow program, commonly written in Python, consists of two phases: construction and execution. Figure 1 gives a simple example abstracted from a real-world industrial application. In the construction phase (lines 1-14), a computation graph is configured: each operator (e.g., *tf.matmal* at line 3) generates some nodes and edges connecting data between nodes. In the execution phase (lines 15-18), a session object is created to instantiate the graph, which is executed multiple times (*sess.run* in line 18) with data being fed into the placeholders (e.g., *in_x* and *in_y* at lines 10 and 11, respectively).

B. The PLATFORM-X Platform

The PLATFORM-X platform is built by Alibaba Group [15] and deployed in its commercial cloud. PLATFORM-X provides support for a variety of deep learning frameworks including TensorFlow [1], PyTorch [20], and MXNet [21]. As for other platforms, users can submit their deep learning jobs via the commandline or a web interface by specifying resources such as CPU/GPU times required and checking the status of submitted jobs.

Most platform users are production teams within this particular company. Everyday, tens of thousands of deep learning jobs run on the platform. There are a substantial number of job failures, i.e., aborted jobs. The platform will tag each failed job, and record its log messages for further investigation. These job failures not only lead to an expensive waste of resources, but also can take an enormous amount of human efforts to debug.

III. AN EMPIRICAL STUDY

Our objective is to develop an effective failure prevention technique. To this end, we take failed TensorFlow jobs on the PLATFORM-X platform as our study subjects. There are a total number of 12,289 failed jobs sampled in one month.

```
# Construction
1.  def fully_connect(input_op, name, n_in,n_out ):
2.    fc_w = tf.get_variable(name, [n_in, n_out])
3.    return tf.matmul(input_op, fc_w)
4.  def predict(Input_x, class_num):
5.    mp = tf.nn.conv2d(input_x,tf.get_variable('mpc',[5,5,1,32]),strides=[1,1,1,1], padding='SAME')
6.    reshaped = tf.reshape(mp, [-1, 28 * 28])
7.    fc = fully_connect(reshaped, 'fc1', 28 * 28, 128)
8.    logit = fully_connect(fc, 'fc2', 128, class_num)
9.    return logit
10. in_x = tf.placeholder(tf.float32, shape = [None, 28, 28, 1])
11. in_y = tf.placeholder(tf.float32, shape = [None, 10])
12. y = predict(in_x, 10)
13. cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=in_y,logits =y))
14. train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
# Execution
15. train_img, train_lab = read_image( batch_size ,...)
16. with tf.Session() as sess:
17.   for i in range(1000):
18.     sess.run(train_step, feed_dict = in_x:train_img, in_y:train_lab))
```

Fig. 1. A sample TensorFlow program (abstracted a real-world industrial application).

TABLE I
THE TOP FIVE MOST COMMON TYPES OF JOB FAILURES.

Error Type	Example Error Message Patterns
Checkpoint Error	*Assign requires shapes of both tensors to match. lhs shape= <*> rhs shape= <*>* *Key <*> not found in checkpoint*
Module/Attribute missing	*<*><*> has no attribute <*>* *No module named <*>*
Arguments Mismatch	*<*> takes exactly <*> arguments <*> given* *<*> got an unexpected keyword argument <*>*
Undefined Variable	*name <*> is not defined* *local variable <*> referenced before assignment*
Shape Error	*Shape must be rank <*> but is rank <*> for <*> op: <*> with input shapes: <*><*><*><*>* *Cannot feed value of shape <*> for Tensor <*> which has shape <*><*>* *Dimensions must be equal, but are <*> and <*> for <*> (op: <*> with input shapes: <*><*>*

All failed jobs are submitted by different production teams in Alibaba Group. For each failed job, we contacted its corresponding production team to collect related information including source code, execution logs, and job scripts. We are not able to obtain the input data to a job since they are regarded as being highly confidential. Thus, it is difficult to reproduce a failure by rerunning the failed application.

Figure 2 shows the size distribution of studied applications. TensorFlow programs are small, with 407 lines of uncommented code on average. The largest program that we studied has 2,355 lines of uncommented code. Note that the third party libraries packaged in an application are not considered.

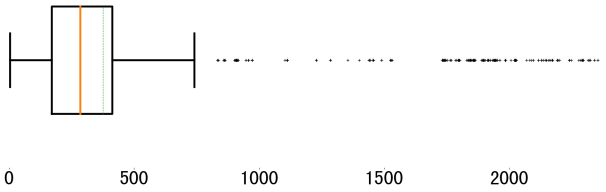

Fig. 2. Distribution of program sizes, in lines of uncommented code.

A. Failure Classification

It is time-consuming to manually analyze the 12,289 applications one by one. Hence, we apply log analysis [22] to group failed applications throwing the same error message pattern together. Figure 3 gives an example. The application throws an error message at line 2, which is parsed into a regular expression *"Input to reshape is a tensor with <*> values, but the requested shape has <*>"*. All the applications throwing the same error message pattern are then grouped together. In the end, there are 968 failed groups. We have sampled 630 applications using the standard sample size calculator with a confidence level of 99%, and confirmed that most applications in the same group fail due to the same root cause.

We have randomly selected two applications in each group, and manually investigated their root causes to failure. Finally, we obtain a total of 17 common root causes. Table I highlights the top five, with some error message patterns highlighted.

```
1. Traceback (most recent call last):
   //other stack traces
2. tensorflow...errors_impl.InvalidArgumentError:
   Input to reshape is a tensor with 1583 values,
   but the requested shape has 1820
```

Fig. 3. The exception trace of a failed job.

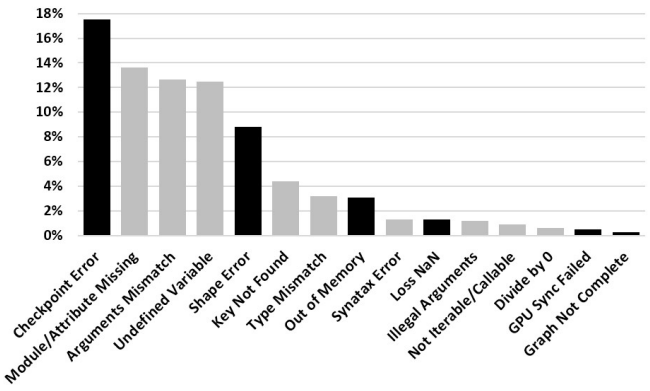

Fig. 4. Bug type distribution in TensorFlow programs. TensorFlow-specific bugs are depicted in dark bars and Python bugs in gray bars.

B. Threats To Validity

First, root cause analysis and failure classification involve manual inspection on application code, which may be subjective. To mitigate this threat, each failed application was examined by two authors separately and the results were cross-validated. Decisions were made only if both authors reached an agreement. For some applications, we also communicated with the original developers to confirm our decisions. Second, our study subjects are all from the PLATFORM-X platform. Hence, some findings may not be applicable to other platforms. To mitigate this threat, we focus on failures caused by program bugs instead of platform-specific issues (e.g., failures related to an execution environment). The PLATFORM-X platform is a widely used platform and the 12,289 studied applications cover a variety of areas, including image and speech recognition, natural language processing, and recommendation systems.

C. Findings

We focus our study on bug-related failures. Out of 12,289 job failures, 1,612 failures are environment-related, throwing error messages such as *"remote file <*> not found"*. In addition, 586 jobs failed due to corrupted input data. Those job failures will not be further discussed. The remaining 10,091 failure-triggering bugs are classified into two categories: Python bugs and TensorFlow-specific bugs. Figure 4 divides these 10,091 bugs into different types of bugs in percentage terms, with the Python bugs shown in gray bars and TensorFlow-specific bugs in dark bars.

***Finding 1*:** 63.69% bugs are Python bugs, which also commonly exist in general-purpose Python applications.

1) Python bugs: Let us examine some Python bugs classified in Figure 4. The most common type of Python bugs is *Module/Attribute Missing* (referencing a non-existent Python class field or function), accounting for 13.61% of all bugs. There are also other common bug types, such as *Argument Mismatch* (invoking a function with an inconsistent number of actual arguments) and *Undefined Variables* (referencing a variable before its definition), accounting for 12.67% and 12.51% of all bugs, respectively.

Several Python bug types are directly related to the dynamic features of Python, e.g., *Type Mismatch* (operating on objects of incompatible types), *Illegal Argument* (arguments not satisfying function specifications), and *Not Iterable/Callable* (iterating over objects of non-collection types). The other Python bug types are common run-time errors such as *Key Not Found* (accessing maps with non-existent keys) and *Divide by Zero* (dividing a value by 0).

2) TensorFlow-Specific Bugs: Checkpoint Error, the most common bug type, accounts for 17.49% of all bugs. Platform users frequently use the checkpointing mechanism to store a trained model to the cloud or to load an already-trained model from the cloud for inference or further training. A checkpoint bug arises when either the model file is missing or the loaded model is inconsistent with the required network structure. The former is related to a particular execution environment and how to deal with the latter is worth a separate investigation.

Shape Error (8.82%) arises when invoking TensorFlow operators with arguments of incompatible shapes (incompatible ranks or dimensions). It is difficult for developers to understand the tricky semantics of thousands of TensorFlow APIs, leading to frequent *Shape Error* bugs in practice. For example, many TensorFlow operators (e.g., *softmax_cross_entrophy_with_logits* at line 13 in Figure 1) support the NumPy "broadcasting" semantics (which "broadcasts" a small array across a relatively large array, by copying leading dimensions of the higher-rank argument and padding any dimension of size 1 to the size of the matching dimension from the other argument), often leading to surprising results.

It can be difficult to debug *Shape Error* bugs. As illustrated in Figure 3, an exception is thrown at line 2 when invoking the *tf.reshape* operator. The *tf.reshape* operator changes the shapes of tensors as long as their sizes (number of elements) stay the same. Hence, it fails to convert a tensor of 1,583 elements to a specified shape of 1,820 elements and throws an exception. However, the *tf.reshape* operator is frequently used. In the example, there are 29 *tf.reshape* operators and it is time-consuming for developers to examine each operator.

***Finding 2*:** *Shape Error* is one of the most common TensorFlow-specific bugs (8.82% of total bugs) and such bugs can be detected effectively as demonstrated in [14].

The other types of TensorFlow-specific bugs include *Out of Memory* (GPU out of memory, commonly fixed by reducing the sizes of tensors), *Loss NaN* (invalid loss values), *GPU Sync Failed* (memory issues in GPU [23]), and *Graph Not Complete* (invalid dataflow graphs).

TABLE II
PYTHON BUGS REPORTED BY FOUR EXISTING STATIC TOOLS [16]–[19].

BugType	Mypy	Pylint	Pyflakes	Pytype	Total
Mod/Att Missing	5.33%	28.00%	8.00%	48.00%	57.33%
Arg Mismatch	1.27%	25.05%	8.28%	18.68%	41.61%
Undef Var	20.85%	71.59%	96.88%	75.70%	98.36%
Total	11.86%	49.78 %	54.98 %	50.95 %	72.55%

IV. DETECTING PYTHON BUGS

There are a number of static tools for finding bugs in Python programs, such as Mypy [16], Pylint [17], Pyflakes [18], and Pytype [19]. We have investigated their effectiveness in detecting Python bugs in industrial TensorFlow programs.

Table II gives the results for the top three Python bug types. Overall, these four tools together have detected 72.55% of all the bugs of these three types. Since these bugs are simple semantic errors, the false positive rates of these four tools are low. Among the four tools, Pyflakes is the best performer, reaching 54.98%. However, all the four tools perform poorly on *Arguments Mismatch* bugs (with Pylint attaining only 25.05% even as the best performer for bug type).

V. DETECTING SHAPE ERROR BUGS

In TensorFlow programs, tensors are the basic data units. A tensor is a multi-dimension array and its *shape* refers to the number of dimensions (rank) and dimensions' sizes. *Shape Error* bugs are the errors incurred when the shape of a tensor does not match the specification of an operator.

In [14], a static analysis, PYTHIA, is introduced for detecting *Shape Error* bugs, by modeling tensor operators in Datalog, so that the shape of a tensor can often be deduced to be a concrete shape (or a set of concrete shapes). PYTHIA can detect 11 out of the 14 shape-related bugs studied in [11]. However, from our study on industrial TensorFlow applications, there are still many cases where the rank or dimension sizes of a tensor are completely unknown. For example, PYTHIA failed to report any error in the 60 real-world applications under testing, due to unresolved unknown shape values. Therefore, in this paper, we introduce a new constraint-based approach, SHAPETRACER, for detecting *Shape Error* bugs, and we will compare it with PYTHIA in our evaluation. In this section, we first use three examples to motivate our approach and then describe it in detail.

A. Motivating Examples

1) Example 1: Figure 5 depicts the computation graph for the program in Figure 1, where each edge is annotated with the shape information of its propagated tensor. The shape of a tensor can be input-dependent: a *placeholder* tensor can set some dimensions (or the whole shape) to *none* and its shape will be instantiated by feeding data to the placeholder (using the *feed_dict* operator, e.g., line 18) when executing the graph.

The computation graph is executed by invoking *Session.run()*, *Tensor.eval()*, or *Operation.run()*. In Figure 1, at line 18, the graph is executed to obtain the results from the operator *train_step* (line 14). The input data *train_image* and *train_lab* (line 15) are fed to the placeholders *in_x* (line 10) and *in_y* (line 11), respectively. Note that the first dimension of input data is configured by an input argument *batch_size* as highlighted by the box in line 15. As a result, the shapes of tensors *in_x* and *in_y* are *[batch_size, 28, 28, 1]* and *[batch_size, 10]*, respectively.

The tensor *in_x* is passed as an actual parameter to the function *predict()* at line 12 and processed by *conv2d* (line 5), a core operator for convolution. The *conv2d* operator is often used to extract intermediate features in complex neural networks. It takes a 4-dimensional *input* tensor, a 4-dimensional *filter* tensor, and a *strides* vector with 4 elements as input. With the "*same*" padding strategy, *cond2d(x, f, s, "same")* will produce a tensor of shape $[x[0], \frac{x[1]}{s_1}, \frac{x[2]}{s_2}, f[3]]$. Hereafter, we use the notation $x[i]$ to represent the ith dimension of tensor *x*'s shape and the notation s_i to represent the ith element of vector *s*. In our example (Figures 1 and 5), the *conv2d* operator produces the tensor *mp* of shape *[batch_size, 28, 28, 32]*.

The *reshape* operator at line 6 changes the shape of the incoming tensor *mp* to a specified shape *[-1,28*28]*, i.e., a 2-dimension array. Here, the special dimension size *-1* denotes that the size of the corresponding dimension needs to be computed dynamically. A tensor can be reshaped correctly if its size (total number of items in the tensor) is the same as the size of the specified shape. At line 6, after reshaping, we have a new tensor *reshaped* of shape *[batch_size*32, 28*28]*.

At line 7, the function *fully_connect* is invoked with the tensor *reshaped* as its actual parameter. Thus, the operators *get_variable* (line 2) and *matmul* (line 3) are included in the computation graph. The *matmul* operator multiply *reshaped* (*[batch_size*32, 28*28]*) with *fc_w* (*[28*28, 128]*), resulting in a new tensor *fc* of shape *[*batch_size*32, 128]. Next, the tensor *fc* is processed by the same function again at line 8. Finally, *logit* (*[batch_size*32, 10]*) is produced and returned as *y*.

The operator *softmax_cross_entrophy_with_logits* (line 13) produces normalized probabilities from input tensors *in_y* (*[batch_size, 10]*) and *y* (*[batch_size * 32, 10]*). It supports the "broadcasting" rule: sizes of matching dimensions must be identical, or one of them is 1 (in which case, the resulting tensor adopts the other size in its corresponding shape dimension). Hence, the operator can succeed only if the sizes of both tensors' first shape dimensions are the same or one of them is 1, i.e., *batch_size*32==batch_size* $\vee$ *batch_size==1*. As the user input *batch_size* is configured to be 200, the application failed with a runtime exception.

2) Examples 2 and 3: Let us look at another two examples given in Figures 6 and 7, respectively. In Figure 6, the tensor *behavior_input* comes from user input and its shape (i.e., its rank and dimension sizes) is completely unknown. At line 1, tensor *tmp_user_profile_cnn* is reshaped to a 4-dimensional tensor *user_profile_cnn*, which is then multiplied with *behavior_input* via the operator *matmul*, suggesting that the rank of *behavior_input* is also 4, since the operator will fail otherwise. At line 3, tensor *attention_weights* is reshaped to a 3-dimensional tensor *tmp_attention_weights*, which is also multiplied with *behavior_input*. Since the input tensor

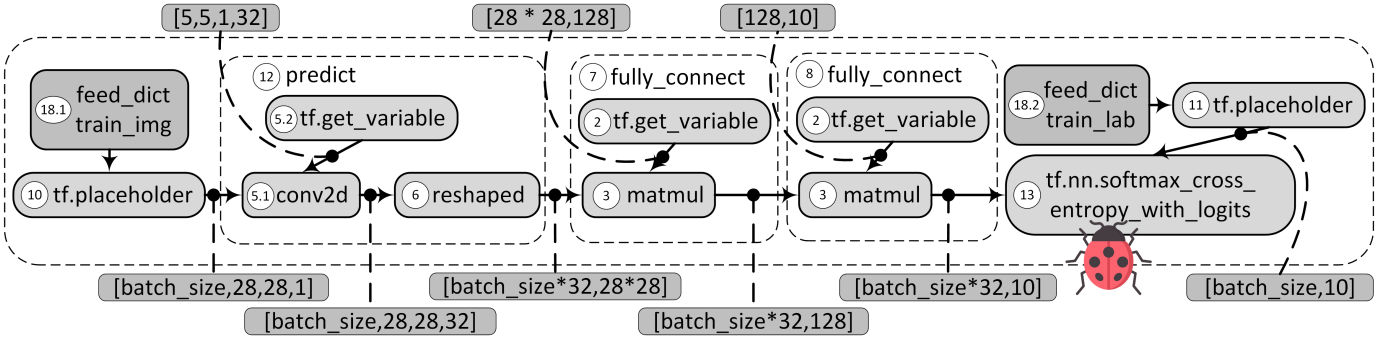

Fig. 5. The computation graph for the program given in Figure 1, where the nodes represent the operators in the program and the tensors (black dots) flow along the graph edges (annotated with the the shape information of their propagated tensors). The bug triggered is highlighted.

```
1. user_profile_cnn = tf.reshape(tmp_user_profile_cnn, shape=[-1, num_behavior_max[behavior_cnt], n_output_behavior, 1])
2. attention_layer_input = tf.matmul( behavior_input,user_profile_cnn)
.....
3. tmp_attention_weights = tf.reshape(attention_weights, shape=[-1, num_behavior_max[behavior_cnt], 1])
4. behavior_output = tf.matmul(tmp_attention_weights,  behavior_input)
```

Fig. 6. Code from an industrial application, where the shape of **behavior_input** is completely unknown. The bug-triggering lines are highlighted in red.

behavior_input cannot satisfy both constraints, the application will always fail on one of the *matmul* operators (at line 2 or line 4).

In Figure 7, the tensor *labels* is a 2-dimensional array and the tensor *pred* is a 1-dimensional array. All their dimension sizes are unknown. The bug is triggered at line 2 when the condition *loss_type* == *"mae"* holds, since the operator *absolute_difference* expects input tensors with the same shape (i.e., same rank and dimension sizes). However, the bug will not be triggered if the other branch is taken (when input parameter *loss_type* is *"logloss"*). The operator *sparse_softmax_cross_entropy_with_logits* allows the rank of the input argument *pred* to be one less than that of *labels*. Hence, it will not trigger a bug.

B. Methodology

In the above three examples, there are tensors with completely unknown shapes (Example 2) or partially unknown shapes (Examples 1 and 3). It is difficult to write Datalog rules (as in PYTHIA [14]) and deduce those unknown shapes to a finite set of concrete shapes. The tensors with unknown or partially unknown shapes can be frequently found in real-world applications. They can come from commandline input, files, or unsupported library functions. Note that it is difficult for developers to manually annotate a tensor from files with unknown shape information. In addition, the TensorFlow library provides thousands of APIs and enterprises often offer their own in-house libraries. It will be a daunting task, if not impossible, to support all libraries APIs, in practice.

Therefore, we are motivated to develop SHAPETRACER, a new tool founded on a constraint-based approach. We represent the shape of a tensor symbolically by introducing symbolic values for unknown ranks or unknown dimension sizes. Constraints can be introduced from tensor operators, scalar variables, and conditional branches. Finally, a constraint solver is applied to check the satisfiability of these constraints.

For instance, for Example 1 (Figure 1), the value of input variable *batch_size* is symbolic, denoted as X. The computation graph will generate a constraint $X * 32 == X \vee X == 1$, together with other constraints. The solution is $X = 1$, which can be provided to users as a warning. In Example 2 (Figure 6), the rank of tensor *behavior_input* is symbolic, denoted as $\overline{X}$. The two *matmul* operators at line 2 and line 4 will introduce their respective constraints, $\overline{X} == 4$ and $\overline{X} == 3$. As both cannot be satisfied together, an error is found.

C. SHAPETRACER

We have implemented SHAPETRACER in WALA [24] and used Ariadane [25] as its front-end to parse Python programs into WALA IR. Figure 8 sketches the architecture of SHAPETRACER. Its three main components are summarized below.

- First, *Builder* traverses program paths and builds a *shape-flow graph* (an abstracted computation graph) for each path.
- Next, *Solver* formulates a shape-flow graph into a list of constraints, which is then solved by Z3 [26], a state-of-the-art constraint solver.
- Finally, an error (warning) is issued if the constraints are not satisfiable (if the user input is constrained). To report precisely the line number where a bug/warning occurs, *Reporter* searches for the first operator introducing unsatisfiable constraints, and reporting it to the user.

Next, we describe these three components in detail.

1) Builder: Builder constructs a shape-flow graph for each program path. This may sound inefficient initially. However,

```
1. if loss_type == ”mae”:
2.    loss = tf.reduce_mean(tf.losses.absolute_difference(labels, pred))
3. elif loss_type == ”logloss”:
4.    loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=pred,labels=labels), 0)
5. optimizer = tf.train.GradientDescentOptimizer(0.05).minimize(loss, ... )
```

Fig. 7. Code from an industrial application, where *pred* is a 2-dimensional array with unknown dimension sizes and *labels* is a 1-dimensional array with an unknown dimension size. The bug-triggering line is highlighted in red.

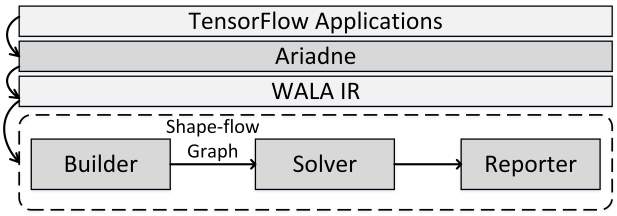

Fig. 8. The high-level architecture of SHAPETRACER.

the control flow structures of TensorFlow programs are usually simple. In general, the number of program paths is 2, rarely reaching 8. In an extreme case, where the neural network is constructed in a loop (Figure 12), the largest number of program paths observed is 256 only.

a) Basic Algorithm: The shape-flow graph of a program is its abstracted computation graph annotated with shape information. To build a shape-flow graph, we slice backwards from an invocation to *session.run()*, i.e., from an output tensor. Since TensorFlow programs commonly propagate values directly through assignments or parameter passing, we slice along the use-def chains of WALA's SSA (single static assignment) representation. During the backward slicing, function calls are inlined: when a function call is met, the return values of the callee function are added to the graph (as new nodes) and we continue slicing backwards from the newly added return values. In the end, all operators (i.e., TensorFlow API invocations), tensors and scalars (e.g., actual parameters of operators), that the output tensor is transitively dependent on, are included in the graph.

Let us explain our basic algorithm using the example in Figure 1. We slice from *sess.run()* at line 18, i.e., the output tensor *train_step*. Since *train_step* is returned from the operator *minimize*, the operator and its operand (i.e., *cross_entropy*) are added to the graph. Similarly, *cross_entropy* is produced by the operator *softmax_cross_entropy_with_logits* (line 13). Hence, the operator and its operands (*in_y* and *y*) are included. From *y*, we inline the function call to *predict* (line 12) and continue slicing from its return value *logit* (line 9). Next, the function *def_fully_connect* is inlined twice at lines 8 and 7, in that order. The final shape-flow graph is given in Figure 5.

b) Graph Duplication: Shape-flow graphs are duplicated at *phi* nodes (control-flow confluence points in SSA). When we encounter a phi node with n incoming values, the graph is duplicated n times and each graph picks a distinct incoming value to continue slicing. Figure 9 gives the shape-flow graphs of Example 2 (Figure 6). In SSA, there exists a phi node at the confluence point of different branches of the *if* statements (lines 1 - 4). Thus, we have two shape-flow graphs, one for each branch.

Loops, although rarely seen in the graph construction phase, are processed by unrolling a loop twice. In our study, there is only one application building neural networks in a loop.

c) Shape Information Collection: Constants and scalar variables propagated directly along use-def chains are recorded straight-forwardly. We try to infer as much concrete information as possible by applying constant propagation and computing concrete shape information according to the documented semantics of TensorFlow APIs. We also consider the following two special cases. First, the shape of a tensor can be set using the *tf.setshape()* function, as shown in Fig. 10 (lines 2 and 4). Hence, for each tensor, we check its uses for a *tf.setshape()* call to the object, and update the shape of the tensor accordingly. Second, in most cases, values are directly propagated. However, when initializing a tensor with a given shape, values are passed into the constructor of the shape as parameters and stored in its corresponding fields. In general, a pointer analysis [27]–[30] is required to compute field-related dependences. However, such fields of a shape object are only stored once, in its constructor (during initialization). Therefore, when encounter a field load, we simply search for a unique store to the corresponding field.

2) Solver: Solver formulates a shape-flow graph as a list of constraints, which are then solved by Z3 [26]. We collect the constraints from tensor operators and scalar instructions in the shape-flow graph. However, we do not consider branch conditions, since shape-related values rarely have data dependences on conditionals in real-world TensorFlow applications.

Figure 11 defines the symbolic representation of shapes and values. Specifically, $T[-1]$ denotes the size of T's last shape dimension. This variable is particularly useful when T's rank is unknown, i.e., $\overline{T}$ is symbolic. By default, we assume that all variables are symbolic unless otherwise specified. If $\overline{T}$ is a constant value C, we introduce a variable for each dimension size of T, by applying the following function to concretize T:

$$Concretize(T, C): \prod_{i=0}^{C-1} T[i] == |T| \wedge T[-1] == T[C-1] \wedge \overline{T} == C$$

This function sets T's rank to C, sets $T[-1]$ to T's last dimension size ($T[-1] == T[C-1]$) , and concretizes T's size to the product of its all dimension sizes ($|T| == \prod_{i=0}^{C-1} T[i]$).

Constraints are introduced for operators according to their documented semantics. For instance, the *C=reshape(A,B)* op-

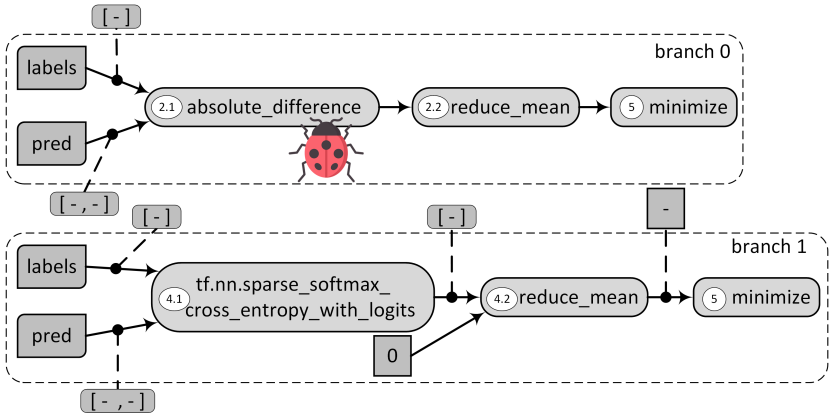

Fig. 9. The two shape-flow graphs (one for each path) of the program given in Figure 7. The oval nodes are operators and square nodes are scalars. The edges are annotated with shape information of their associated tensors (blackdots).

```
1. x = tf.placeholder(tf.float32, [None])
2. x.set_shape([1028178])
3. y = tf.identity(x)
4. y.set_shape([478, 717, 3])
5. X = np.random.normal(0, 0.1, 1028178)
6. Y = sess.run(y, feed_dict=x:  X)
```

Fig. 10. The code example UT-3 from [11].

$T[0], T[-1], T[-], ...$	T's Dimension sizes
$\|T\|$	T's Total size (number of elements)
$\overline{T}$	T's Rank (number of dimensions)
$V_0, V_1, ..., V_{\|V\|-1}$	V's element values
X	X's value

Fig. 11. Symbolic representation of tensor *T*'s shape, vector *V*'s values, and scalar *X*'s value. *V* is a 1-dimensional shape, and its size $|V|$ is a constant.

erator reshapes tensor *A* to tensor *C* of the same size, with the shape specified by vector *B*. Hence, we have:

$$\bigwedge_{0\le i<|B|} C[i] == B_i \wedge \overline{C} == |B| \wedge Concretize(C, |B|) \wedge |C| == |A|$$

Here, the constraint $|C| == |A|$ states that tensor *C* and *A* have the same size (as required by *reshape*), and the remaining constraints specify the shape of *C* according to vector *B*: *C*'s rank is defined by *B*'s size ($\overline{C} == |B|$), and *C*'s dimensions are defined by *B*'s elements ($\bigwedge_{0\le i<|B|} C[i] == B_i$). Note that the size of *B*, i.e., $|B|$, is a constant. Hence, tensor *C* is concretized ($Concretize(C, |B|)$). Except for one element value (e.g., -1), all the other element values are constant. The same list of constraints is applicable to the case when tensor *A*'s rank is constant, i.e., when *A* is already concretized.

Let us examine the operator *softmax_cross_entropy_with_logits*, abbreviated as *C=logits(A,B)*, for supporting Numpy "broadcasting". Before we dive into the details of the tricky broadcasting semantics, we first introduce another helper function $Broadcast(A, B, C, i, j)$. This function represents the constraints on the ith dimension of the higher-ranked input tensor *A*, the ith dimension of output tensor *C*, and the matching jth dimension of the other input tensor *B*, where $j \le i$:

$$Broadcast((A, B, C, i, j) : ((A[i] == B[j] \wedge C[i] == A[i]) \\ \vee (A[i] == 1 \wedge C[i] == B[j]) \vee (B[j] == 1 \wedge C[i] == A[i]))$$

$Broadcast((A, B, C, i)$ holds if one of the following three cases holds: 1) $A[i]$ matches with $B[j]$, producing the same size for *C*'s ith dimension, 2) $A[i]$ is 1, in which case *C*'s ith dimension takes the size from *B*'s jth dimension, and 3) $B[j]$ is 1, in which case *C*'s ith dimension size takes that from *A*'s. The three cases reproduce the semantics of broadcasting one pair of matching dimensions (*A[i]* and *B[j]*).

The list of constraints for *C=logits(A,B)* is given by:

$$(\overline{A} == \overline{B} \wedge Broadcast(A, B, C, 0, 0) \wedge Broadcast(A, B, C, -1, -1)) \vee \\ (\overline{A} > \overline{B} \wedge Broadcast(A, B, C, -1, -1) \wedge A[0] == C[0]) \vee \\ (\overline{A} < \overline{B} \wedge Broadcast(A, B, C, -1, -1) \wedge B[0] == C[0])$$

The above constraints are applied when $\overline{A}$ or $\overline{B}$ is symbolic. In this case, we only introduce the constraints on the first and last dimension sizes of the input and output tensors. In the case of $\overline{A} == \overline{B}$, the constraints are applied to the first and last dimensions of all three tensors *A*, *B*, and *C* ($Broadcast < A, B, C, 0 > \ \wedge \ Broadcast < A, B, C, -1 >$). In the other two cases, the constraints are applied to the last dimension of the three tensors, and the output tensor *C* takes the size from the higher-ranked tensor (e.g., $A[0] == C[0]$ when $\overline{A} > \overline{B}$).

When both $\overline{A}$ and $\overline{B}$ are constants (i.e., *A* and *B* are concretized), *C* can be concretized as follows:

$$(\overline{A} == \overline{B} \wedge Concretize(C, \overline{A}) \wedge \bigwedge_{0\le i<\overline{A}} Broadcast(A, B, C, i, i)) \vee \\ (\overline{A} > \overline{B} \wedge Concretize(C, \overline{A}) \wedge \bigwedge_{0\le i<\overline{A}-\overline{B}} C[i] == A[i] \wedge \\ \bigwedge_{0\le i<\overline{B}} Broadcast(A, B, C, i + \overline{A} - \overline{B}, i)) \vee \\ (\overline{A} < \overline{B} \wedge Concretize(C, \overline{B}) \wedge \dots)$$

In the case of $\overline{A} == \overline{B}$, the output tensor *C* is concretized and each of its dimensions is defined by the $Broadcast$ rule. Otherwise, *C* is concretized with the higher rank (e.g, $Concretize(C, \overline{A})$ when $\overline{A} > \overline{B}$), higher di-

TABLE III
NUMBER OF REPORTED ERRORS/WARNINGS IN INDUSTRIAL AND OPEN-SOURCE (FROM [11]) APPLICATIONS. #TP IS THE NUMBER OF REPORTED TRUE BUGS, #FP IS THE NUMBER OF REPORTED FALSE POSITIVES, AND #FN IS THE NUMBER OF FALSE NEGATIVES.

Tool	Industrial			Open-Source		
	#TP	#FP	#FN	#TP	#FP	#FN
SHAPETRACER	24/16	0	20	9/0	0	5
PYTHIA	0/0	0	60	7/4	1	2
Enhanced PYTHIA	9/0	0	51	7/4	1	2

mensions are copied directly from the higher-ranked tensor ($\bigwedge_{0\leq i<\overline{A}-\overline{B}} C[i] == A[i]$), and the matching dimensions are broadcasted ($\bigwedge_{0\leq i<\overline{B}} Broadcast(A, B, C, i + \overline{A} - \overline{B}, i)$).

Similarly, appropriate constraints are introduced for the other operators, such as *conv2d* and *matmul*. To date, SHAPETRACER provides support for 54 common operators, with an average of 34.8 lines of code for each operator. For tensors returned from unsupported library functions, their ranks are symbolically represented. In the end, the constraints of a shape-flow graph are fed into Z3 [26].

3) Reporter: If Z3 fails to solve a given set of constraints (if a user input is constrained by a constant value), an error (warning) is issued. While errors can always trigger a bug, warnings suggest expected user inputs. To report the bug location precisely, *Reporter* searches for an operator introducing the unsatisfiable constraints found. Conceptually, this is realized by removing each operator one by one (more precisely, by removing the constraints introduced by each operator), in the reverse order of when it is added to the underlying dataflow graph, until the constraints become satisfiable. In practice, we have accelerated this process using binary search.

VI. EVALUATION

Our evaluation addresses the following research questions:

- How effective is SHAPETRACER in detecting *Shape Error* bugs in TensorFlow programs?
- How does SHAPETRACER compare to a state-of-the-art static analysis tool, PYTHIA [14]?
- How efficient is SHAPETRACER?

All experiments were conducted on a laptop equipped with i5-9400 CPU and 32GB RAM.

A. RQ1: Effectiveness

We evaluate SHAPETRACER using a set of 60 buggy industrial TensorFlow programs (randomly picked from our study) and the 14 open source programs studied in [11].

Table III summarizes the results. Overall, SHAPETRACER has successfully detected 40 out of 60 bugs in industrial programs, and 9 out of 14 bugs in open-source applications. There are 33 errors (24 from the industrial programs and 9 from the open-source programs) and 16 warnings (all from the industrial programs). A bug can definitely be triggered for any of the 33 reported errors. The 16 warnings are subject to user input, in which case SHAPETRACER warns on expected input values. The applications that exhibit these warnings can only be correct if the corresponding shape-related constraints

```
1. deep_out = None
2. for idx, info in enumerate(self.deep_info):
3.   if idx == 0:
4.     deep_out= deep_features
5.   else:
6.     deep_out = res_out
7.   deep_out = tf.matmul(deep_out,info["w"])
8.   deep_out = tf.nn.leaky_relu(deep_out)
9.   if idx > 1:
10.    res_out=tf.concat([deep_out,deep_out_list[idx-1]])
11.  else:
12.    res_out = deep_out
13.  if len(deep_out_list) <= idx:
14.    deep_out_list.append(deep_out)
15.  else:
16.    deep_out_list[idx] = deep_out
.....
17.  loss = tf.reduce_mean(
           tf.nn.sigmoid_cross_entropy_with_logits(
           logits=pred, labels=labels))+l2
```

Fig. 12. Code snippet of a false negative example: the neural network is built in a loop. Each loop iteration builds one layer of the network (lines 2-8). The next loop iteration uses the previous two layers as input to build a new layer (line 10). The bug is triggered at line 17 as highlighted in red.

TABLE IV
FALSE NEGATIVES IN THE 20 INDUSTRIAL PROGRAM BUGS EXPLAINED.

#	Reasons for Producing False Negatives
19	Too many shape-related values from user input
1	Constructing neural networks in a loop.

are met. According to our experience, the odds for such a warning to be an error is much higher than that for the shaped-related constraints to be always satisfiable. Thus, these 16 warnings can be considered as errors detected. We will compare SHAPETRACER and PYTHIA later.

1) False Negatives: SHAPETRACER failed to detect 20 industrial program bugs, with the two reasons given in Table IV. 19 bugs were not reported because the corresponding programs define most of their tensor shapes in configuration files, resulting in many unknown shapes. Since SHAPETRACER failed to deduce a constant-constrained input, no error or warning was given. The other false negative occurs in the program illustrated in Figure 12, where a neural network is constructed in a loop. *ShapeTracer* processes a loop by unrolling it twice, which is not sufficient for detecting this particular bug.

There are also 5 open-source program bugs missed by *ShapeTracer*. We will discuss these bugs in Section VI-B2.

2) False Positives: SHAPETRACER did not report any false positives for the programs in Table III. To evaluate its precision further, we have further tested SHAPETRACER using a set of 60 randomly selected correct programs from the PLATFORM-X platform. Again, no false positive was reported.

3) Error and Warning Examples: Figure 13 gives a buggy example, with the bug triggered at line 10. The operator *absolute_difference* requires the incoming arguments of the same shapes, producing constraint $\overline{layer8} == \overline{var\_0}$. Furthermore, SHAPETRACER can infer that *layer8* is of shape [-1,1] (line

```
1. def input_fn():
2.   #other complex operation
3.   var_0.shape = [-1]
4.   var_1 = tf.reshape( var_1, [None, 31])
5.   return var_0,var_1
6. def model_fn(var_1, var_0, reuse):
7.   #other complex operation
8.   #layer7 and w7 comes from other operation
9.   layer8 = tf.matmul(layer7,W7)+b7#now layer8 is [-1,1]
10.  tmp1 = tf.losses.absolute_difference(layer8, var_0)
11.  loss = tf.reduce_sum(tmp1)
12.  return loss,
13.var_0, var_1 = input_fn()
14.loss = model_fn(var_1, var_0, False)
15.c = sess.run([loss])
```

Fig. 13. Code snippet with a bug, abstracted from an industrial program. Sensitive variable names have been replaced with *var_0* and *var_1*.

```
1. def fun_0(batch_size):
2.   #other complex operation
3.   var_1 = cloud_platform_api0(batch_size, 67)
4.   return var_1
5. var_0 = fun_0(batch_size) #batch_size is from user input
6. x = tf.reshape(var_0, [-1, 66])
```

```
#warning reported by ShapeTracer
[[z3][suggest]]
    [batch_size]filename.py(line3)__value = 66
    [anonymous]filename.py(line6)__value = 67
```

Fig. 14. Code snippet of an example, which contains a warning, abstracted from an industrial program. Sensitive variable names have been replaced with *var_0* and *var_1*. The warning message is given in the box.

9) and *var_0* has a shape of [-1] (line 3). Inference details are omitted here due to space limitation. As a result, we derive the constraints $\overline{layer8} == \overline{var\_0} \wedge \overline{layer8} == 2 \wedge \overline{var\_0} == 1$, which are unsatisfiable. Hence, an error is reported. It is worth noting that SHAPETRACER generates 1,485 constraints for this program. However, *Reporter* is able to examine every operator and precisely points out the bug location.

Figure 14 gives an example triggering a warning, together with the warning message reported by SHAPETRACER. The *cloud_platform_api0* at line 3 produces tensor *var_1* of shape $[batch\_size, 67]$ (with $batch\_size$ being unknown), which is propagated to *var_0* (line 5). The operator *reshape* at line 6 reshapes *var_0* to the specified shape [-1, 66], and expects the size of *var_0* to match with the size of the specified shape, i.e., $batch\_size \times 67 == X \times 66$, where X stands for the symbolic value of the vector. Given such constraints, SHAPETRACER will issue a warning as highlighted. Note that the warning and error messages are helpful to developers during both code review or post-mortem debugging.

B. RQ2: Comparing with PYTHIA

Table III also compares SHAPETRACER with PYTHIA [14] (provided by its artifact) on detecting *Shape Error* bugs in industrial and open-source programs. PYTHIA is a state-of-the-art tool for detecting TensorFlow shape-related bugs.

TABLE V
COMPARING SHAPETRACER AND PYTHIA ON OPEN-SOURCE PROGRAMS, WITH ✓(⚠) DENOTING A CORRECTLY REPORTED ERROR (WARNING).

	UT-1	UT-5	UT-12	UT-13	UT-15
SHAPETRACER	✓	✓	-	-	-
PYTHIA	⚠	-	⚠	⚠	⚠

1) Industrial Programs: PYTHIA neither detects any industrial program bugs nor reports any false positives. To figure out why, we have carefully examined the logs and messages printed by PYTHIA and summarized the reasons below:

- PYTHIA failed on 23 industrial programs due to implementation bugs. It throws runtime exceptions in *PYTHON_FACT_GEN* when generating Python facts. We have reported this issue to the PYTHIA developers for further investigation.
- PYTHIA failed on the other 34 industrial programs due to unknown shape values. Although facts were successfully generated, the unknown shape values prevented its analysis to progress further. Unknown shape values come from user inputs or unsupported operators. Note that although there still exist a considerable number of unknown values in SHAPETRACER, our constraint-based approach still enables our analysis to detect many bugs, as demonstrated in our motivating examples (Section V-A).
- PYTHIA can successfully analyze a partially-known shape (with a *None* or special (-1) dimension). However, it cannot analyze both completely unknown shapes (e.g., an unknown rank) and completely unknown dimensions when they cannot be concretized. As a result, it fails to detect any error in the set of 60 real industrial programs. We further investigated on how to extend PYTHIA to deduce as many unknown shapes as possible. An extra *Unknown* tag is introduced for any unknown shape, and new datalog rules are introduced to deduce the rank and dimension values of tagged *Unknown* shapes. For example, for a *matmul* operator, its two parameter shapes must have the identical rank. Thus, we can infer the rank of a completely unknown parameter shape from the other parameter. We have extended PYTHIA with a set of 75 datalog rules (509 LOC). This extension enables PYTHIA to detect 9 shape-related errors. However, it still fails to report the other errors due to unknown shape values or unfixed crashes. For example, three shape-related errors that are missed by PYTHIA are related to complex constraints like *batch_size*32==batch_size* illustrated in Figure 1.

2) Open-Source Programs: PYTHIA performs much better on open-source programs, and we have successfully reproduced their results (as in their paper [14]) using their given artifact. Table V highlights the differences of the two tools on open-source programs. PYTHIA reported warnings for the 3 bugs, UT-12, UT-13, and UT-15, missed by SHAPETRACER. PYTHIA exploits heuristics to report these 3 warnings (e.g., suspicious broadcasting). We did not implement such heuristics in SHAPETRACER because they can lead to many false

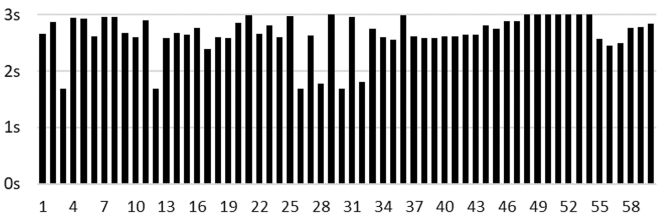

Fig. 15. Analysis times of SHAPETRACER on 60 industrial programs.

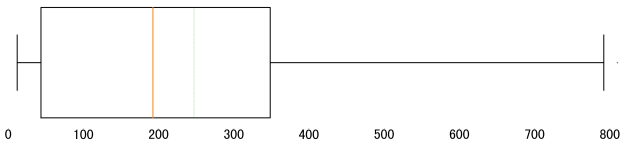

Fig. 16. Size distribution of shape flow graphs for industrial programs.

positives and *developers often ignore such warnings*.

In UT-13, operator *argmax(Y[4,1], axis=1)* will result in a shape of [4] with all values being zero. In UT-15, *y[3,1]-y_[3]* produces a tensor of shape[3,3], due to broadcasting. PYTHIA warns on such suspicious operations since the results look surprising. However, these programs still satisfy the shape-related rules. We attempted to incorporate similar heuristics but were discouraged to do so (by our industry sponsor developing the PLATFORM-X platform) because of spurious reports. UT-12 uses *list slice* in Python to construct the shape of all initial tensors, which is not yet supported by SHAPETRACER.

C. RQ3: Efficiency

Figure 15 summarizes the analysis times of SHAPETRACER on the 60 industrial programs. SHAPETRACER is fast, as it finishes all its analyses in at most 3 seconds for each program on a standard laptop with 32GB RAM. Note that the analysis times include the times in exploring programs paths, collecting and solving constraints, and searching for bug locations.

Figure 16 shows the size distribution of shape-flow graphs. The sizes of shape-flow graphs range from 12 to 810 nodes, with an averages of 246.9 nodes. The average number of constraints for each shape-flow graph is 914.8, which looks seemingly large. However, as most of constraints are constant equality constraints, Z3 can solve them very efficiently.

VII. RELATED WORK

Empirical Studies. Zhang et al. [11] investigated 175 TensorFlow program bugs from Stack Overflow and GitHub. Following [11], Islame et al. [31] performed a more comprehensive study on deep learning program bugs, including 2,716 bugs from applications using five different deep learning libraries (Caffe [2], Keras [32], Tensorflow [1], Theano [33], and Torch [3]). The authors in [12] conducted an extensive empirical study on 4,960 job failures on Microsoft's *Philly* platform. Guo [13] surveyed bugs in deep learning development and deployment. In this paper, we also perform an empirical study focusing on 12,289 industrial TensorFlow job failures on a new platform, motivating us to develop SHAPETRACER, a new static tool for detecting TensorFlow shape errors.

Static Bug Detection. Python is the most popular language in developing deep learning applications [13]. Python bugs [16], [17] in deep learning programs can be detected quite effectively with existing static tools, as confirmed in this paper.

A number of research efforts focus on shape-related bugs. Ariadne [25] is the first static shape analysis tool developed for TensorFlow. However, due to implementation issues (e.g., failing to analyze shapes inter-procedurally), it cannot effectively detect errors in practice [14]. PYTHIA [14] is shown to be able to detect 11 out of 14 open-source program bugs using a Datalog-based static analysis. In this paper, we introduce a new constraint-based approach and a tool, SHAPETRACER, to detect effectively industrial TensorFlow program bugs.

Testing. There is a large body of research [34]–[40] aiming at testing the robustness of deep learning models. How to apply a constraint-based approach to improve testing effectiveness is an interesting topic worth further investigation.

VIII. CONCLUSION

This paper aims at detecting industrial TensorFlow program bugs. We have conducted an extensive empirical study on 12,289 failed industrial TensorFlow jobs. Based on our findings, we have applied four existing representative static tools to detect 72.55% of the top three common Python bugs in TensorFlow programs. To detect TensorFlow-specific bugs, we have introduced the first constraint-based approach for detecting TensorFlow shape-related errors and developed an associated static tool, SHAPETRACER. We have applied SHAPETRACER to a set of 60 industrial TensorFlow programs, showing that SHAPETRACER is both efficient (by analyzing a program in at most 3 seconds) and effective (by detecting 40 out of 60 industrial TensorFlow program bugs, with no false positives). SHAPETRACER is now deployed in the PLATFORM-X platform and will be publicly available soon.

ACKNOWLEDGEMENTS

Thanks to all the reviewers for their constructive comments. Project supported by the State Key Laboratory of Computer Architecture, ICT, CAS, China (Grant No. CARCH5402).

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[2] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.

[3] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Idiap, Tech. Rep., 2002.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[5] A. Vaswani, S. Bengio, E. Brevdo, F. Chollet, A. N. Gomez, S. Gouws, L. Jones, L. Kaiser, N. Kalchbrenner, N. Parmar, R. Sepassi, N. Shazeer, and J. Uszkoreit, "Tensor2tensor for neural machine translation," *CoRR*, vol. abs/1803.07416, 2018. [Online]. Available: http://arxiv.org/abs/1803.07416

[6] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 3104–3112.

[7] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[8] (2020) Ai and machine learning products. [Online]. Available: https://cloud.google.com/products/ai

[9] (2020) Azure machine learning. [Online]. Available: https://azure.microsoft.com/en-us/services/machine-learning/

[10] (2020) Machine learning for every data scientist and developer. [Online]. Available: https://aws.amazon.com/sagemaker/

[11] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on TensorFlow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–140. [Online]. Available: https://doi.org/10.1145/3213846.3213866

[12] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1159–1170.

[13] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 810–822.

[14] S. Lagouvardos, J. Dolby, N. Grech, A. Antoniadis, and Y. 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.

[15] (2020) Alibaba. [Online]. Available: https://www.alibabagroup.com/

[16] (2020) mypy. [Online]. Available: http://mypy-lang.org/

[17] (2020) pylint. [Online]. Available: https://www.pylint.org/

[18] (2020) pyflakes. [Online]. Available: https://github.com/PyCQA/pyflakes

[19] (2020) pytype. [Online]. Available: https://google.github.io/pytype/

[20] 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," *arXiv preprint arXiv:1912.01703*, 2019.

[21] 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," *arXiv preprint arXiv:1512.01274*, 2015.

[22] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 121–130.

[23] (2015) Nvidia responds to gtx 970 3.5gb memory issue. [Online]. Available: https://wccftech.com/nvidia-geforce-gtx-970-memory-issue-fully-explained/

[24] (2006) Wala. [Online]. Available: https://github.com/wala/WALA

[25] J. Dolby, A. Shinnar, A. Allain, and J. Reinen, "Ariadne: analysis for machine learning programs," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 1–10.

[26] (2020) Z3. [Online]. Available: https://github.com/Z3Prover/z3

[27] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009, pp. 243–262.

[28] N. Grech and Y. Smaragdakis, "P/taint: unified points-to and taint analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[29] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: modeling the heap by merging equivalent automata," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 278–291.

[30] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction*. New York: ACM, 2016, pp. 265–266.

[31] 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*, 2019, pp. 510–520.

[32] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[33] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv*, pp. arXiv–1605, 2016.

[34] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[35] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 120–131.

[36] 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*, 2018, pp. 303–314.

[37] 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: Association for Computing Machinery, 2019, p. 146–157. [Online]. Available: https://doi.org/10.1145/3293882.3330579

[38] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic autonomous driving system testing," *arXiv preprint arXiv:1802.02295*, 2018.

[39] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "Deepct: Tomographic combinatorial testing for deep learning systems," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 614–618.

[40] 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.