Static Analysis of Shape in TensorFlow Programs

Sifis Lagouvardos
University of Athens, Greece
sifis.lag@di.uoa.gr

Julian Dolby
IBM Research, Yorktown Heights, NY, USA
dolby@us.ibm.com

Neville Grech
University of Athens, Greece
me@nevillegrech.com

Anastasios Antoniadis
University of Athens, Greece
anantoni@di.uoa.gr

Yannis Smaragdakis
University of Athens, Greece
smaragd@di.uoa.gr

Abstract

Machine learning has been widely adopted in diverse science and engineering domains, aided by reusable libraries and quick development patterns. The TensorFlow library is probably the best-known representative of this trend and most users employ the Python API to its powerful back-end. TensorFlow programs are susceptible to several systematic errors, especially in the dynamic typing setting of Python. We present Pythia, a static analysis that tracks the shapes of tensors across Python library calls and warns of several possible mismatches. The key technical aspects are a close modeling of library semantics with respect to tensor shape, and an identification of violations and error-prone patterns. Pythia is powerful enough to statically detect (with 84.62% precision) 11 of the 14 shape-related TensorFlow bugs in the recent Zhang et al. empirical study – an independent slice of real-world bugs.

2012 ACM Subject Classification Theory of computation → Program analysis; Software and its engineering → Compilers; Software and its engineering → General programming languages

Keywords and phrases Python, TensorFlow, static analysis, Doop, Wala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.15

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at `https://doi.org/10.4230/DARTS.6.2.6`.

Funding We gratefully acknowledge funding by the European Research Council, grant 790340 (PARSE), and by the Hellenic Foundation for Research and Innovation (project DEAN-BLOCK).

1 Introduction

Machine learning has seen widespread use in recent years, for an enormous variety of application domains, from vision to language processing to programming tasks [3, 23, 39] and well beyond, into mainstream science and engineering. The TensorFlow library [1], originally developed by the Google Brain Team, is the dominant open-source framework for modern machine learning applications. TensorFlow has received significant attention and impressive adoption, continually extending its dominance over other frameworks. Current statistics (as

of Jan.08, 2020) show the TensorFlow GitHub repository with over 140K stars and 79.4K forks, with other popular open-source frameworks for machine learning lagging far behind (PyTorch [37] at 35.2K stars and 8.8K forks, Theano [2] at 9K stars and 2.5K forks).

As might be expected, TensorFlow programs are not free of defects ("bugs"). In high-level code, such as TensorFlow clients, bugs are commonly due to misunderstandings of the guarantees offered and obligations imposed by increasingly layered software. At the same time, such bugs have increasing real-world importance, as machine learning makes advances in widespread adoption. In a recent empirical survey, Zhang et al. [58] collect and classify a variety of TensorFlow program bugs from StackOverflow QA page and GitHub projects, by examining documentation, informal posts, commit and pull-request messages, and issue discussions. Many of these bugs are semantic in nature: they can only be ascertained by inspecting the outcome or the performance of the underlying computation. Others are bugs that may admit automatic detection: they signify API misuse, often (but not always) triggering assertions during execution.

TensorFlow, as many other popular machine learning frameworks, is mostly used from Python: a dynamic language that offers significant flexibility and ease of adoption. The dynamic nature of Python implies that there is no static tracking of types that can be used to ensure compatibility of values and operations. Furthermore, the static analysis tools available for Python are less advanced than those in statically-typed languages, focusing more on local code issues rather than whole-program properties. One reason for this has been a lack of underlying general analysis frameworks (analogous, e.g., to WALA [50], Soot [52], or Doop [9] in the Java world) that deploy whole-program technology and support Python. (For instance, we have failed to find a publicly available library for points-to analysis of Python programs.)

In this work, we focus on a class of TensorFlow bugs that relate to the *shape* of tensors, i.e., the number of their dimensions and the dimensions' sizes. Checking that the shape of tensor arguments is compatible with the expectations of library operators is a key validation technique. Shape checking can prevent a large and important class of real-world TensorFlow programming errors, including the 14 shape-related bugs identified in StackOverflow questions by Zhang et al. [58].

Our approach tracks the shape of tensors using static analysis of the Python program and appropriate modeling of the TensorFlow API. In addition to the dynamism of the Python language, static analysis or type checking of TensorFlow code is also hindered by the inherent dynamism of the library itself. The design philosophy of the library (much in line with its common use from a dynamic language) is that of being very resilient to incomplete data. The API exhibits multiple instances of dynamic padding, reshaping, unknown dimensions, partially-known shapes (to be filled in dynamically), and more. Our analysis follows the flexibility of the library operators and attempts to closely model what is a permitted and expected behavior vs. what will produce a run-time error or is very likely a logical error and should induce a warning.

The work offers both application-level and technical-level contributions:

- We define Pythia, a state-of-the-art static analysis for the modeling of tensor shape through TensorFlow API calls. The analysis combines several elements: a relatively complete front-end translating Python source code into the IR of the WALA framework; a translation of the WALA IR into a relational representation for defining analyses using declarative Datalog rules; a whole-program context-sensitive value-flow and points-to analysis for Python; and a shape analysis of tensor values that carefully captures the flexibility of library operators.

- We provide the first concrete demonstration of the applicability of static analysis in the TensorFlow domain, by showing that our tool can find real bugs in real TensorFlow programs. We validate the effectiveness of the analysis by applying it to the 14 shape-related bug examples (and their fixed versions) in the [58] study. Pythia correctly finds 11 of these bugs with a precision of 84.62% and recall of 78.6%. (Importantly, of the missed bugs, all but one are undetectable with static information alone.)
- We present insights on the design of static shape checking for Python/TensorFlow programs. In particular, we argue that an effective such analysis is best classified as a static analysis and not a type checker, due to its desired features (extensional, non-modular behavior, context sensitivity).

2 Background

We next present background useful in later sections, on TensorFlow and Datalog program structure.

TensorFlow

TensorFlow is the most widely-adopted open-source machine learning library. The library performs computations using symbolic data-flow graphs. Operators form the vertices of the graph and tensors are flowing along the edges. TensorFlow invocation from Python code typically follows a two-stage pattern.[1] Initially the data-flow graph representing the computations is constructed. The entire graph is in place before dynamic data have been read. This graph or a number of its sub-graphs can then be executed multiple times with different input data.

During the construction phase of the graph the information about each tensor's shape may vary. It may range from fully-known or *concrete*, to partially-known (where one or more dimensions is unknown, represented as `None`) to completely unknown. The static analysis we describe is based on retrieving as much shape information as possible from the program text, and propagating it through TensorFlow operators, which require careful modeling with respect to their shape transformations. Therefore, *the analysis is crucially based on common TensorFlow programming patterns*. These encourage encoding known shape information in the program text, while leaving unknown (dynamic) shape information undefined.

Datalog in Program Analysis

The Datalog language has been often used to declaratively specify static analysis algorithms [8, 18, 20, 25, 27, 30, 33, 36, 46, 53, 54, 57]. We use Datalog in our analysis, both in the high-level description and in its implementation, in order to seamlessly combine the results of several separate analyses (constant-flow, points-to, tensor-shape), with each one appealing to others.

A Datalog program is a set of logical inference rules, operating over initial facts and producing more inferences until fixpoint. A rule "C(z,x) ← A(x,y), B(y,z)." means that if A(x,y) and B(y,z) are both true, then C(z,x) can be inferred. We shall use syntactic shorthands in the rules, such as multiple rule heads ("H1(...), H2(...) ← ..."), which are equivalent to repeating the rule for each head, and disjunction (operator ";") in the rule body, which is equivalent to replicating the body for each disjunct.

[1] This description, as well as all of our work and presentation, applies to TensorFlow v.1.X, the most widely deployed version of the framework. TensorFlow v.2 was released in late 2019 and includes a radical (and incompatible) reworking of the programming model. Both our core analysis and the engineering scaffolding need to be reworked to apply to TensorFlow v.2, which will likely give rise to related but not identical kinds of bugs. This is a potentially promising future work direction.

3 Illustration: TensorFlow Shape Tracking

The concept of a tensor's shape is straightforward and mostly well-understood: every tensor has a list of dimensions, each with a size. Tensor operations are well-defined when the arguments' dimensions match the operator's expectations. We shall see in Section 4 a more complete mathematical modeling of tensor shapes, but a simple, well-known example is the 2-dimensional tensor (matrix) multiplication operator (TENSOR i j represents a tensor of shape $i \times j$):

$$mul2d : \text{TENSOR } i\ j \to \text{TENSOR } j\ k \to \text{TENSOR } i\ k$$

The complexity of modeling tensor shape in practice is much greater, however. The issue is precisely the dynamism that the TensorFlow library (as well as the Python language) affords. Our analysis seeks to capture this flexibility while closely modeling shape transformations through the TensorFlow API. We next consider several examples that illustrate a) how placeholder tensors, reshaping operations, implicit padding, and subtle semantic differences affect shape reasoning; b) which behaviors cause crashes and which can be reasonably considered likely bugs, and should, therefore, also elicit a warning; c) what flavor an analysis should adopt to capture such bugs in realistic programs.

Example 1: Placeholders

A first example helps demonstrate "placeholder" tensors.

```
import tensorflow as tf
import numpy as np
data1 = np.random.normal(0, 0.1, [20, 50])
data2 = np.random.normal(0, 0.1, [50])
a = tf.placeholder("float", shape=[None, 50])
b = tf.placeholder("float")
y = tf.matmul(a,b)
with tf.Session() as sess:
    print(sess.run(y, feed_dict={a:data1,b:data2}))
```

A placeholder tensor is a tensor that will be fed data at runtime. At instantiation of a placeholder tensor, some dimensions (or the whole shape) can be set to `None`, as in tensors `a` and `b` in our example. Feeding data to a placeholder can be done using the `feed_dict` optional argument to `Session.run()`, `Tensor.eval()`, or `Operation.run()`. When one or more dimensions are set to `None`, the data fed to this tensor has to match the shape of the placeholder, meaning that the number of dimensions has to be the same and the sizes of all explicit-sized dimensions should be equal.

The most common pattern is to set a dimension that represents the "batch number" of the data to `None`, to support placeholder tensors with a variable batch size: the structure of each instance is known, but the total number of instances is a run-time variable. In the code snippets we will be showcasing throughout the paper, the arguments of TensorFlow and NumPy[2] operations that affect the output shape will be highlighted in red. Such is the case in our example, where the call to `np.random.normal()` results in `data1` pointing to a NumPy array object with shape `[20,50]`. Consequently, feeding `data1` to the placeholder `a` with shape `[None,50]` is successful. In our static analysis, this placeholder operation will produce two different modeled result values: `[None,50]` and `[20,50]`.

[2] NumPy is the dominant Python scientific computing package. Our modeling also covers parts of NumPy that are particularly relevant to TensorFlow operations.

Placeholder tensors with no initial shape can be fed data of any shape, as long as the types are compatible. To model these, we take advantage of the shape of the data fed to the placeholder whenever it is available. In our example, the `b` placeholder tensor gets the shape of `data2` which is `[50]`. The call to `tf.matmul()` will fail with an error due to the two argument tensors having different number of dimensions. Our static analysis will issue an error, since no combination of the modeled abstract values for tensors `a` and `b` yields a compatible pair.

Example 2: Reshaping

Not all tensorflow bugs will result in run-time exceptions/assertion failures, yet strong evidence may exist that the code contains an error. An example is below, also illustrating the `tf.reshape()` operator.

```
import tensorflow as tf
import numpy as np
a = tf.placeholder(tf.float32, [None,784])
data = np.random.normal(0, 0.1, [36, 784])
b = tf.reshape(a, [-1,24,24,1])
with tf.Session() as sess:
    print(sess.run(b,feed_dict={a:data}).shape)
```

The `tf.reshape()` function attempts to reshape a tensor, given as input the dimensions specified by its second (`shape`) argument. In order for it to succeed, the product of the elements of the shape list of the input tensor (p_{in}) and the product of the elements of the shape list of the output tensor (p_{out}) should be equal. A very common special case concerns argument shape lists with a single allowed `-1` dimension, as in the `reshape` call of the example. The size of that dimension is then computed so that the reshape operation succeeds, provided that the product of explicit (i.e., not `-1`) dimensions of the shape argument is a divisor of p_{in}.

In the above example, just as in the earlier Example 1, the placeholder tensor `a` has originally one `None` dimension, corresponding to the batch size. The tensor, with shape `[None,784]`, is fed data with shape `[36,784]`. Dynamically, this value is compatible with the `reshape` operation, with attempted shape `[-1,24,24,1]`: the resulting shape of tensor `b` is `[49,24,24,1]`, since $49 \times 24 \times 24 \times 1 = 36 \times 784$. However, there is already a strong hint that the reshaping should only affect the second dimension, with size 784 (i.e., that the programmer expects that 784 should be divisible by 24): the batch size is a volatile attribute of the current input and not an inherent part of the tensor structure, as the explicit `[None,784]` shape suggests.

Our analysis keeps both abstract values, `[None,784]` and `[36,784]`, for tensor `a` and, since one of them is incompatible with the `reshape` operation, it emits a warning. Generally, when the input tensor of a `reshape` has one `None` dimension, we compute the products of the elements of the two shape lists excluding `None` and `-1` and if they are not equal we report a warning.

Example 3: Padding in Broadcast Operations

The distinction between analysis-reported errors and warnings is more generally meaningful for operations that are probably valid, yet likely to have surprising semantics. The most common such case is the "broadcasting" semantics of NumPy arrays. We discuss the behavior in Section 5 but the example below illustrates briefly.

```
import tensorflow as tf
x = tf.constant([[1.0, 1.0], [1.0, 2.0],
                 [1.0, 3.0]], dtype=tf.float64)
y_ = tf.constant([1.0, 2.0, 3.0], dtype=tf.float64)
w = tf.truncated_normal(shape=[2,1], stddev=0.1, dtype=tf.float64)
y = tf.matmul(x, w)
diff = y - y_
error = tf.reduce_mean(tf.square(diff))
```

In this example, tensor `x` has a shape of `[3,2]` and tensor `w` has a shape of `[2,1]`. Their product, tensor `y` has a shape of `[3,1]`. Tensor `y_` has a shape of `[3]`. The difference of `y` (`[3,1]`) and `y_` (`[3]`) has a shape of `[3,3]`, which is highly surprising to many users! (Broadcasting semantics copy leading dimensions of the higher-rank argument[3] and match the rest one-to-one, expanding any dimension with size 1 to the size of the matching dimension from the other argument.)

Pythia models and correctly propagates the effect of broadcasting on shape. However, it produces a warning when array broadcasting results in the expansion of the dimensions of a tensor. This can help prevent errors caused by mechanics that can easily confuse a user.

4 Basic Tensor Shape Modeling

Practical analysis applications that yield realistic benefits need to devote considerable modeling effort to support the idiosyncrasies of different environments – in our case, TensorFlow's operations. Much of the complexity of this modeling is due to technicalities employed for usability, to the sheer number of operators, or to the way data values are introduced from the host language. There is, however, a core set of operations that are representative of many more and whose basic shape modeling can be cleanly expressed in closed-form mathematical formulas, much as the reader might expect. We discuss the "clean" modeling of such operators in this section, and postpone discussing the more operational aspects of our analysis until Section 5. Therefore, this section is purposely simplifying, in order to ensure that the core model is clear to the reader. For instance, we omit tensors of partially-known shape (with `None` dimensions), special (`-1`) dimensions in reshaping, modeling of broadcasting, and other such complexities.

Every tensor operation is modeled mainly in terms of the output shape in relation to the inputs supplied to its formal parameters, and of the data type of individual tensor elements. Complexity mostly arises out of the former, so our design is influenced by this consideration. Tensor operations broadly consist of (i) shape pass-through functions, e.g., `identity` ; (ii) convolution and pooling functions ; (iii) conversions and reshapings from tensors or tensor-like objects (e.g., NumPy arrays).

Shape types of tensors are modeled using the following vocabulary for tensor types and tensor shapes.

$$
\begin{array}{rcl}
\tau, \upsilon \in \mathit{TensorType} & ::= & \textsc{Tensor}\ T \\
T, U \in \mathit{DimensionType} & ::= & T\ i \\
 & \mid & \textsc{nil} \\
i, j \in \mathbb{N} & &
\end{array}
$$

[3] The "rank" of a shape is its number of dimensions.

We typically omit NIL for conciseness. Hence an example of a two dimensional tensor shape type is TENSOR i j, i.e., a tensor of shape $i \times j$.

Note that the variables in the above syntax are meta-variables, used for conceptual modeling. In concrete instances, inside our analysis, all shapes and their dimensions are concrete (i.e., sequences of integers or single integers, respectively). Conceptually, however, the logic of the analysis does use such meta-variables, since it handles any concrete numbers found in the program text. For instance, we can model the understanding of the analysis regarding the core TensorFlow operator MUL as follows.

$$\text{MUL} \quad : \quad \text{TENSOR } U\ i\ j \to \text{TENSOR } U\ j\ k \to \text{TENSOR } U\ i\ k$$

MUL generalizes standard two-dimensional tensor (matrix) multiplication, by adding arbitrary (but identical, in both arguments and in the result) leading dimensions.

Similarly, the core operator IDENTITY takes a tensor of any shape and returns a copy of it with the same shape.

$$\text{IDENTITY} \quad : \quad \text{TENSOR } T \to \text{TENSOR } T$$

RESHAPE is another core TensorFlow operator. It takes a tensor of any shape T and tries to return a tensor of another shape U, supplied as argument.

$$\text{RESHAPE} \quad : \quad \text{TENSOR } T \to U \to \text{TENSOR } U$$

The RESHAPE operation succeeds if the product of all elements in T is equal to the product of all elements in U:

$$\prod_i T_i = \prod_i U_i$$

To get a glimpse of more complex and versatile shape modeling, still easy to express in a closed-form formula, we can consider the CONV2D operator – a core operator for convolution. Convolution is often used to create complex neural networks that can extract intermediate features (typically from an image), as part of an intermediate layer. Convolution takes a 4d `input` tensor, where the middle 2 dimensions represent the data, a `filter` tensor, and a `strides` shape. Furthermore, there are two padding strategies for convolution: `same` and `valid`. Essentially, the former pads the tensor with off-boundary data so that the convolution filter is still applicable on the edges, while the latter avoids padding and applies only up to the point where the filter still retrieves data from the input tensor. If the padding strategy is `same` the shape type of CONV2D is defined as:

$$\text{CONV2D} : \quad \text{TENSOR } *\ i\ j\ * \to \text{TENSOR } *\ k\ l\ * \to *\ s_1\ s_2\ * \to \text{TENSOR } \lceil \tfrac{i}{s_1} \rceil\ \lceil \tfrac{j}{s_2} \rceil$$

(The stars denote any, ignored, integer values.)
Otherwise, if the padding strategy is `valid`, the convolution shape type is defined as:

$$\text{CONV2D}: \quad \text{TENSOR } *\ i\ j\ * \to \text{TENSOR } *\ k\ l\ * \to *\ s_1\ s_2\ * \to \text{TENSOR } \lceil \tfrac{i-k+1}{s_1} \rceil \lceil \tfrac{j-l+1}{s_2} \rceil$$

5 Analysis Structure

Our analysis emphasis is on shape modeling, which is the main element of this work. However, given the dearth of static analysis infrastructure for Python, our analysis had to develop several techniques and combine them in a coherent whole: a Python front-end (parser, IR

generator) that produces intermediate code using the WALA framework [50], a generator of relational tables for declarative program analysis in the Doop framework [9], a points-to, constant-flow and call-graph analysis for Python.

We start our presentation from these underlying analyses, and proceed with representative fragments of the declarative modeling of shape transformations through TensorFlow operators.

5.1 Substrate: WALA and Declarative Value-Flow Analysis

Pythia is expressed declaratively, as Datalog rules for both value-flow and tensor-shape reasoning. For Python support, we extended the parser and intermediate-representation generator of the Ariadne system [12], which produces WALA IR statements from Python source. The past WALA front-end for Python was largely a proof-of-concept implementation, therefore several elements needed to be added to tackle realistic programs, for example:

- correct handling of the global scope of Python programs
- complete modeling of collections
- complete modeling of list comprehensions
- modeling of list slicing
- modeling of parameter initial values
- handling of constant values.

The resulting intermediate representation using WALA data structures is used to output tables for relational processing by Datalog-based analyses. We integrate the input relations generation and subsequent analysis with the Doop framework [9], which already features a WALA front-end and a declarative analysis scaffolding. Doop is a framework for analysis of Java bytecode – to add Python support, we implement a whole-program, context-sensitive value-flow analysis on the Python IR.

The form of this analysis is largely conventional, expressed using a standard declarative approach (e.g., [49]) over an SSA intermediate language (for flow sensitivity on local variables). The analysis propagates constants and object values inter-procedurally, maintaining precision using call-site sensitivity [47, 48]. (In the default setting, a 1-call-site-sensitive analysis with a context-sensitive heap is used, after experimentation with options to balance performance and precision.) A call-graph is inferred based on the values of receiver objects at method calls. The analysis is complete for the static features of Python, but several dynamic features (e.g., decorators, non-trivial list comprehensions, `eval`/`input`, `getattr`) will interrupt the propagation of values.

5.2 Declarative Modeling of Shape Transformations

The main analysis logic is expressed as rules that appeal to the substrate analysis of value-flow throughout the Python program. In general, the declarative model of the analysis helps in having simple, independent rules, mutually recursive with other sub-analyses. We illustrate two sample sets of rules, next, capturing shape reasoning for broadcast operations and reshape operations. As hinted in earlier examples, much of the complexity is due to the close modeling of the flexibility afforded by TensorFlow operators.

5.2.1 Broadcast Reasoning

Example 3 in Section 3 discussed array/tensor broadcasting. Array/tensor broadcasting is a mechanism to allow element-wise operations between arrays of different shapes. Under some restrictions, the smaller array is "broadcast" across the larger one, provided that

their dimensions match. Broadcasting operations can either be overloaded arithmetic binary operations, or calls to tensorflow functions (for example `tf.add()`, `tf.multiply()` or `tf.equal()`).

Consider an example to illustrate different cases:

```
import tensorflow as tf
op1 = tf.ones(shape=[4,3,1])
op2 = tf.ones(shape=[3,2])
res = tf.add(op1, op2)
```

The shape of the resulting tensor will be `[4,3,2]`: leading dimensions are "inherited" from the higher-rank tensor, and dimensions equal to 1 for either argument are expanded to the size of the corresponding dimension for the other argument.

Our analysis logic first creates a new value for a broadcasting operation, encoding (as a 3-tuple) the instruction and tensor argument values. We choose to have the operand with the higher rank as the first operand. The difference of the ranks is computed as the operation's *offset*. In our example, the operation's offset will be 1.

```
BroadcastingOp(bcastOp, offset) ←
  Invocation(insn, fun),
  BroadcastingFunction(fun),
  ActualParamValue(insn, "x", tensor1),
  ActualParamValue(insn, "y", tensor2),
  TensorRank(tensor1, rank1),
  TensorRank(tensor2, rank2),
  rank1 >= rank2, offset = rank1 - rank2,
  bcastOp = [ insn, tensor1, tensor2 ].
```

The rule checks the preconditions of broadcasting and packages all relevant information for further processing. The relations in the rule body are produced by syntactic processing of the program text or by the global value-flow/points-to analysis: Invocation(*insn, fun*) recognizes a call to *fun* in instruction *insn* (such invocation resolution requires global value-flow reasoning); BroadcastingFunction matches TensorFlow API functions that support broadcasting, such as `add` in our example; ActualParamValue(*insn, var, val*) computes the (abstract) value for the actual parameter (*var*, identified by name) of a call at instruction *insn*; TensorRank retrieves the rank of a tensor value (i.e., number of dimensions in its shape).

In words, the rule says that if two tensor values, *tensor1* and *tensor2*, are used as arguments of a broadcasting call, the call instruction, the tensor values, and the offset to be used to match the tensors' dimensions are packaged in predicate BroadcastingOp.

Armed with the above, we can encode the different cases of shape propagation through broadcasting operators. The ResultShapeDimension predicate represents the contents of the shape list for each dimension of a broadcasting operation's result.

For dimensions only in the higher-rank argument (i.e., below "*offset*") the result inherits the size of the higher-rank argument's dimension:

```
ResultShapeDimension(bcastOp, index, dim) ←
  BroadcastingOp(bcastOp, offset),
  bcastOp = [ _, tensor1, _ ],
  index < offset,
  TensorShape(tensor1, tensorShape1),
  ShapeDimension(tensorShape1, index, dim).
```

Predicate TensorShape holds the shape of a tensor value, while ShapeDimension(*shape, i, dim*) holds the size, *dim*, of the *i*-th dimension of the shape value.

In our example, the above rule will produce the dimension with size 4 in the result.

In order to attempt a match of dimension sizes that are expected to match during a broadcasting operation, we introduce a convenience predicate, ArgumentsShapeDimensions that recalls both sizes at positions at least equal to *offset*:

```
ArgumentsShapeDimensions(bcastOp, index, dim1, dim2) ←
  BroadcastingOp(bcastOp, offset),
  bcastOp = [ _, tensor1, tensor2 ],
  index >= offset,
  TensorShape(tensor1, tensorShape1),
  ShapeDimension(tensorShape1, index, dim1),
  TensorShape(tensor2, tensorShape2),
  ShapeDimension(tensorShape2, index - offset, dim2).
```

For fully matching argument dimensions, the common size becomes the size of the output dimension, as well (as in the second dimension of the output in our example):

```
ResultShapeDimension(bcastOp, index, dim) ←
  ArgumentsShapeDimensions(bcastOp, index, dim, dim).
```

There are two more cases and they elicit a warning or an error report: The first computes an output shape of the resulting tensor for dimensions that are not equal but can match due to broadcasting. For two different dimensions to match in this way, at least one would need to be 1. This rule produces the result of the third dimension of our earlier example. In this case we also produce a warning, detecting the use of broadcasting mechanics that could confuse the user.

```
Warning(bcastOp),
ResultShapeDimension(bcastOp, index, dim) ←
  ArgumentsShapeDimensions(bcastOp, index, dim1, dim2),
  dim1 != dim2,
  ((dim1 = 1, dim = dim2) ;   (dim2 = 1, dim = dim1)).
```

The final rule produces an error in the case of dimensions that cannot match, i.e., they are not equal and neither of them is 1.

```
Error(bcastOp) ←
  ArgumentsShapeDimensions(bcastOp, index, dim1, dim2),
  dim1 != dim2,   dim1 != 1,   dim2 != 1.
```

5.2.2 Reshape Reasoning

Example 2 in Section 3 discussed the complexity of modeling the `reshape` operator in a realistic setting, unlike the core, closed-form modeling of Section 4. Tensors of partially-known shape (with `None` dimensions) and special reshape dimensions (of size `-1`) need to be accounted for in an analysis that aims to be useful for real-world bug detection. The Datalog rules we present next reflect these considerations.

First, as in broadcast operations, we identify calls to `reshape` and encode each instance of the operation as a new value, consisting of a 3-tuple of the instruction, tensor, and shape arguments:

```
ReshapeOperation(rshpOp) ←
  Invocation(insn, "reshape"),
  ActualParamValue(insn, "tensor", tensorVal),
  ActualParamValue(insn, "shape", dimListVal),
  rshpOp = [insn, tensorVal, dimListVal ].
```

We also store the products of dimension sizes for the tensor and the shape argument (with the result of the multiplication over all indexes computed separately – from rules not shown – into DimensionsProduct):

```
ProductsOfShapes(rshpOp, tensorProd, dimListProd) ←
  ReshapeOperation(rshpOp),
  rshpOp = [ _, tensorVal, dimListVal ],
  TensorShape(tensorVal, tensorShapeVal),
  DimensionsProduct(tensorShapeVal, tensorProd),
  DimensionsProduct(dimListVal, dimListProd).
```

We can then distinguish different cases of reshapings. A concrete-dimension tensor (i.e., with no `None` dimensions) reshaped into a concrete shape (i.e., with no `-1` dimensions) will succeed if the products of dimension sizes are equal and will produce an error otherwise. Predicate ReshapeConcreteToConcrete is used to cache intermediate results for use in the two later rules. "!" designates negation in a rule and in this case is used to establish the two shape concreteness conditions.

```
ReshapeConcreteToConcrete(rshpOp, tensorProd, dimListProd) ←
  ProductsOfShapes(rshpOp, tensorProd, dimListProd),
  rshpOp = [ _, tensorVal, dimListVal ],
  TensorShape(tensorVal, tensorShapeVal),
  !ShapeDimension(tensorShapeVal, _, "None"),
  !ShapeDimension(dimListVal, _, -1).

TensorOperationProducesOutput(rshpOp) ←
  ReshapeConcreteToConcrete(rshpOp, tensorProd, tensorProd).

Error(rshpOp) ←
  ReshapeConcreteToConcrete(rshpOp, tensorProd, dimListProd),
  tensorProd != dimListProd.
```

Accordingly, we can handle the case of a concrete tensor resized to a special shape – i.e., one that has a `-1` dimension. (Other rules, omitted, enforce that there can be at most one `-1` dimension.) We first collect the products of sizes into a convenience predicate that also enforces the rest of the preconditions:

```
ReshapeConcreteToSpecial(rshpOp, tensorProd, dimListProd) ←
  ProductsOfShapes(rshpOp, tensorProd, dimListProd),
  rshpOp = [ _, tensorVal, dimListVal ],
  TensorShape(tensorVal, tensorShapeVal),
  !ShapeDimension(tensorShapeVal, _, "None"),
  ShapeDimension(dimListVal, _, -1).
```

Subsequently, we distinguish the case of a correct reshaping, when the two dimension-size-products are divisible, from the error case, when they are not:

```
TensorOperationProducesOutput(rshpOp) ←
  ReshapeConcreteToSpecial(rshpOp, tensorProd, dimListProd),
  quot = tensorProd/dimListProd,
  tensorProd = quot * dimListProd.

Error(rshpOp) ←
  ReshapeConcreteToSpecial(rshpOp, tensorProd, dimListProd),
  tensorProd % dimListProd != 0.
```

In a largely similar fashion, we need to handle the case of a tensor of partially-known shape (with a `None` dimension) being reshaped to a shape with a special (`-1`) dimension. We first compute the products of concrete dimension sizes:

```
ReshapePartialToSpecial(rshpOp, tensorProd, dimListProd) ←
  ProductsOfShapes(rshpOp, tensorProd, dimListProd),
  rshpOp = [ _, tensorVal, dimListVal ],
  TensorShape(tensorVal, tensorShapeVal),
  ShapeDimension(tensorShapeVal, _, "None"),
  ShapeDimension(dimListVal, _, -1).
```

Then, we distinguish the case of a correct reshaping, when both products match vs. one that elicits a warning, when they do not. This is the case of the earlier Example 2, commonly corresponding to a programming error. In the example, our analysis correctly infers the product of the input tensor to be 784 and that of the given shape argument to be 576, successfully reporting the appropriate warning. Remember that, for both shapes, their products are the products of the explicit dimensions (i.e. no `None` and `-1` dimensions).

```
TensorOperationProducesOutput(rshpOp) ←
  ReshapePartialToSpecial(rshpOp, tensorProd, tensorProd).

Warning(rshpOp),
TensorOperationProducesOutput(rshpOp) ←
  ReshapePartialToSpecial(rshpOp, tensorProd, dimListProd),
  tensorProd != dimListProd.
```

The final case is that of a tensor of partially-known shape reshaped into a concrete shape list, with no special dimensions. We again enforce the preconditions and cache the products of dimension sizes in a convenience predicate:

```
ReshapePartialToConcrete(rshpOp, tensorProd, dimListProd) ←
  ProductsOfShapes(rshpOp, tensorProd, dimListProd),
  rshpOp = [ _, tensorVal, dimListVal ],
  TensorShape(tensorVal, tensorShapeVal),
  ShapeDimension(tensorShapeVal, _, "None"),
  !ShapeDimension(dimListVal, _, -1).
```

Subsequently, we distinguish the case of a correct reshaping from that of an error:

```
TensorOperationProducesOutput(rshpOp) ←
  ReshapePartialToConcrete(rshpOp, tensorProd, dimListProd),
  dimListProd % tensorProd = 0.

Error(rshpOp) ←
  ReshapePartialToConcrete(rshpOp, tensorProd, dimListProd),
  dimListProd % tensorProd != 0.
```

5.3 Tensor Value Representation

The presentation of the analysis so far has ignored the exact nature of abstract values that arise for tensors. In our previous rules, abstract tensor values are treated as black-box representations that have at least a type and a shape (a list of constants), which the rest of the analysis looks up. The exact abstraction of values, however, has significant implications on precision and performance (even up to non-termination, as Section 5.4 discusses). Our full analysis features two different configurations for tensor value creation: one very coarse and one highly precise.

The coarse value abstraction, which we term *simple-tensor-precision*, creates a single value for each function invocation instruction that resolves to a modeled tensor operation. The problem of this approach is that it can exacerbate the – sometimes unavoidable – imprecision of a static analysis. The snippet below provides a minimal example.

```
import tensorflow as tf
if(CONDITION):
    reshapeTo = [-1,28,28,1]
else:
    reshapeTo = [-1,14,14,4]
a = tf.placeholder(tf.int32, [None, 784])
a = tf.reshape(a, reshapeTo)
```

After the `reshape` operation, variable `a` points to one tensor value with one corresponding shape value. However, having a single value (for all dynamic instances of the operation) entails having a single shape list. This shape list has two possible values for each dimension except the 0th, combining the possible dimension values of the two run-time shape lists to a total of 8 possible shapes.

In contrast, the precise value abstraction of Pythia, which we call *full-tensor-precision*, represents each tensor value as the concatenation of all the values of arguments of the operation that creates it and the function invocation instruction that resolves to the operation.

For the above example, after the reshape operation, variable `a` points to two tensor values – one for each possible value of `reshapeTo` – but each with definite shape: the full shapes (`[-1,28,28,1]` vs. `[-1,14,14,4]`) of the `reshapeTo` arguments are kept in the two abstract values representing the operation's results.

As discussed next, it is easy to switch between the two abstractions to implement interesting hybrid algorithms that give a balance of precision and scalability.

5.4 Analysis Termination

Termination is an interesting question regarding our analysis. There are new tensor shapes produced for several TensorFlow operators, e.g., by replacing `-1` dimension sizes with positive integers in the `reshape` operation. Also, even though the analysis deals with concrete dimensions, it remains a static analysis: a single variable can have many potential abstract values. These do not necessarily reflect dynamic values – they could arise due to control-flow or data-flow imprecision, i.e., because of an over-approximation. Therefore a program with no threat of non-termination can still possibly give rise to a non-terminating static analysis.

To ensure termination of our analysis, we first need to bound the new shapes that can be created. Doing so immediately establishes that our analysis will always terminate when running under the *simple-tensor-precision* value abstraction. We then show how a run of our analysis in the *simple-tensor-precision* configuration can ensure the termination of our analysis with the *full-tensor-precision* value abstraction for the same input program.

5.4.1 Finite shapes

The first challenge for establishing the finiteness of shapes is to show that the integer constants that arise (for each shape dimension independently) are finite.

Conveniently, tensor operations by themselves (without arithmetic in the Python program) cannot create an infinite number of dimension sizes. The dimension sizes for a new shape are either sizes of an existing tensor shape's dimension (as in the case of tensor multiplication), or smaller dimension sizes (as in the case of convolution or reshaping operations, which take quotients of existing shape sizes).

Still, the above observation does not help bound the overall dimension sizes due to Python arithmetic. The finiteness property is actually one that the analysis needs to artificially enforce, since we propagate integer constants (corresponding to tensor dimension sizes) through arbitrary arithmetic operations. For instance, a tensor operation such as:

```
n_input = train_X.shape[1]
```

means that the number of inputs (which will later be used as the dimension size of a tensor) comes from the dimensions of another tensor. With arithmetic over the `n_input` variable and a looping construct, the potential dimension sizes become infinite. Therefore, we artificially bound the "complexity" (i.e., number of intermediate arithmetic operations) of the computed integer constants. (For instance, in our implementation, this bound is a generous 50.)

We additionally need to bound the maximum number of dimensions of a tensor, since operations such as `tf.expand_dims` can increase the number of dimensions.

5.4.2 Termination for Different Value Abstractions and Maximizing Precision

Based on the finiteness of integer dimensions in shapes, the analysis will always terminate for the *simple-tensor-precision* value abstraction: there is a finite number of values, each (by definition) has a single shape list, the shape list has a finite number of dimensions, and each dimension can have a finite set of values for its potential size.

The case for the *full-tensor-precision* value abstraction is more complicated. In principle, this abstraction is not finite: new tensor values can keep arising, even if they have the same shape. As an example, consider a `transpose(x, [0, 2, 1])` operation – permuting the dimensions of tensor argument `x` according to the list given as the second argument – with the output of the operation feeding back to the input tensor (due to a loop or recursion). In the *full-tensor-precision* abstraction, with output values of tensor operations being represented by the concatenation of all their input values, this would result in the creation of a `transpose(x, [0, 2, 1])` value feeding back to the argument of the operation, resulting in a `transpose(transpose(x, [0, 2, 1]), [0, 2, 1])` value, and so on.

Therefore, we employ the full-tensor-precision abstraction in our analysis only over tensor operations with no cyclic dependencies (on themselves). Concretely, Pythia first runs under a *simple-tensor-precision* abstraction, while also propagating values to detect circularity in the inference. For each tensor operation we compute the set of tensor values that flow to it and the corresponding operations that created them. In this way, we can detect the existence of cyclic dependencies. The simple-tensor-precision abstraction is less (i.e., at most as) precise than full-tensor-precision, therefore any cycles arising in the latter will definitely arise in the former.

Subsequently, we enable an analysis with full-tensor-precision only when no evaluation cycles have arisen. In this way, we can leverage higher precision in the common case of TensorFlow programs that do not employ recursion or looping at the Python level, instead delegating complex computation to library operators.

6 Discussion

Tensor shapes are reminiscent of types. It is, therefore, interesting to consider the relationship between our analysis and type checking, as well as the overall potential for a static type system for TensorFlow functionality.

Although the boundary between static analysis and type checking is not always clear, our static checker is best classified as a static analysis. Key factors in this classification are the whole-program and extensional nature of the analysis, as well as the intended soundness in reasoning.

Extensional Representation

The analysis represents value sets extensionally, i.e., by listing all their contents, instead of trying to abstract over them. For instance, if a tensor variable `t` is inferred to hold possible shapes `[4, 3, 3, 2]`, `[None, 45]`, and `[30]`, the analysis will maintain the three different shape values explicitly, instead of trying to unify them in a single, more abstract, shape. This is a property more commonly found in static analyses than in type systems – the latter typically summarize values, at least at the level of program modules (e.g., functions). Static analyses also employ abstraction, but only do so based on the properties of the values themselves (e.g., when two values join in an abstract lattice).

Modular vs. Whole-Program

A type system typically emphasizes modular reasoning, forcing the summarization of values at the function boundary. In contrast, a context-sensitive whole-program static analysis will re-analyze a function under its different calling contexts. To maintain precision for different clients of a function, a type system employs polymorphism instead of context sensitivity: it expresses the type of a function in terms that may employ type variables, i.e., symbolic types that may assume multiple type values, instead of constants.

Sound vs. Best-Effort Reasoning

A static type system aims for soundness in certifying correct code, i.e., guarantees no false negatives. This implies that a type system has to be conservative in certifying correct code, yielding many false positive warnings. In contrast, a static analysis can strike any balance between true/false positive and true/false negative warnings as it deems appropriate for maximum usefulness.

TensorFlow Analysis

With the above factors in mind, it is interesting to consider the static checking of TensorFlow programs longer than toy examples. The example in Figure 1 is a slightly simplified version of one of the programs in the Zhang et al. [58] study. There is a bug in the last line of the program (the reshaping of `h_pool2`) which our analysis correctly warns about, but the main difficulty is in tracking shapes precisely in earlier program statements.

```
import tensorflow as tf
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='VALID')
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

x = tf.placeholder(tf.float32, shape=[None, 784])

W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
x_image = tf.reshape(x, [-1, 28, 28, 1])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])
...
```

Figure 1 Short but realistic example of the need for context sensitivity: program Unaligned-Tensor-1 (UT-1).

Context-sensitive reasoning is essential for this example. Functions `conv2d` (line 8) and `max_pool_2x2` (line 10) are each called twice (lines 18, 23, and 19, 24, respectively), each time with different shapes. (Pooling can be thought of as an operator analogous to convolution in terms of shape transformation.) Even for a small example such as this, a context-insensitive analysis would produce a highly imprecise result, with many false positives and negatives. A context-sensitive analysis considers, e.g., function `conv2d` twice, once for each argument shape, and can reason highly precisely about the effects of convolution on shape when all arguments (`x`, `W`, `strides`, as well as the padding strategy, `VALID`) have known values.

Conversely, consider what type signature (to capture shape) one might assign to function `conv2d` *modularly*, i.e., without knowing its arguments, `x` and `W`. Precise treatment of this function would require a polymorphic type system with considerable expressive power (e.g., integer arithmetic), as the modeling of Section 4 shows. This would more likely employ dependent typing, requiring significant human guidance for deciding interesting properties. It is interesting to further consider possible type signatures for a) the library method `tensorflow.nn.conv2d`, which has different shape behavior depending on the padding strategy; and b) library operations that employ broadcast. The current flexibility of these TensorFlow operators seems to require full algorithmic expressiveness (e.g., see our Datalog rules of Section 5.2.1) to capture well.

We conclude that the shape transformation of current TensorFlow operations requires a highly-expressive vocabulary, unlikely to be supported by a fully automatic type system. A statically-typed TensorFlow-like system would either require significant programmer assistance for sound reasoning, or curtail the flexibility of operators, to permit assigning them closed-form types.

7 Evaluation

Pythia runs at interactive speeds and has a Language Server Protocol integration with most popular IDEs. Therefore, although the analysis is applicable for development at any granularity, it mostly targets interactive feedback at development time. Our evaluation is set up accordingly.

Specifically, we evaluate the analysis against complete pre-existing programs from the recent study by Zhang et al. [58] on TensorFlow bugs. This gives us a curated dataset, collected independently, from real-world settings, and with ground truth relative to both the presence and the absence of bugs.[4]

The Zhang et al. study collects bugs from StackOverflow questions, and categorizes them based on their root causes and symptoms. One of the root cause categories is *Unaligned Tensor (UT)*, which maps exactly to shape violations detected by our analysis. There are 76 bugs that Zhang et al. manage to reproduce from StackOverflow questions[5]. Importantly, these 76 bugs map the entire, broad space of all TensorFlow bugs, most of which are out of the scope of our shape analysis. For instance, this includes low-accuracy computations, low-performance behavior, bugs related to API changes, and more.

There are 14 Unaligned Tensor (i.e., shape-related) bugs in the Zhang et al. study, and the study also provides fixed versions of the same programs, for a total of 28 test subjects, which form the universe set of our evaluation.

Executive Summary

With an 1-call-site-sensitive analysis and the *full-tensor-precision* option enabled, the analysis successfully detects 11 out of 14 bugs, with a single false positive repeated twice in the buggy and fixed version of UT5, for a precision of 84.62% (91.67% if repeat bugs are counted once) and recall of 78.6%. The average analysis time per program is under 1sec. Unless specified, the above analysis configuration is used. The effect of different analysis configurations on analysis precision is discussed in Section 7.3.

7.1 Classification of bugs

Table 1 summarizes the number of bugs reported. The bugs can be classified in the following categories:

- Operation Error: Tensor operation would throw a run-time error due to incompatible arguments provided.
- Incompatible fed data Error: Data fed to a placeholder tensor do not match the shape of the tensor.
- Broadcast/Reshape/Other Warning: Possibly confusing tensor operation behavior that would not cause a run-time error, as described in Section 5.

Table 2 serves as a detailed reference for each input program (including code URLs).

[4] Links to all input programs can be found in Table 2. Pythia is part of the Doop repository (`https://bitbucket.org/yanniss/doop/`). A snapshot is contained in the artifact that accompanies this paper, together with detailed instructions for setting up and running Pythia.

[5] The Zhang et al. study also collected a second dataset: 75 bugs from GitHub commits. We did not consider that dataset for reasons of engineering: a large number of these full Github programs use several external libraries, in addition to TensorFlow, as well as the full TensorFlow API. Modeling all of the required functionality, so that the potential of the approach is accurately evaluated, would require much more manpower than that of a research project. In contrast, the StackOverflow Zhang et al. benchmarks are well-isolated TensorFlow code patterns, which fit well the local, incremental nature of our approach and its implementation inside IDEs.

Table 1 Detected bugs.

Bug type	Number of bugs
Operation Error	5
Incompatible fed data Error	2
Broadcast/Reshape/Other Warning	4
Total	11

7.2 Effectiveness and Efficiency

Overall, among the 14 input programs containing bugs, we successfully identify the bug in 11 programs. Our analysis produces a false positive in both the buggy and fixed versions of UT5 achieving 84.62% (11 of 13) overall *precision* – 100% for errors and 66.6% for warnings. The false positive appearing in both versions of UT5 is a warning for a reshape operation of a tensor with partially known shape into a shape with a special (`-1`) dimension, as described in section 5.2.2. In this case, the use of the reshape operation in a way we consider possibly confusing does not result in a bug.

The 3 false negatives produced by the analysis are a result of either API calls that we have not modeled or reliance on dynamic information to identify these bugs, leading to 78.6% *recall*. Of the 3 bugs our analysis could not detect, two (UT5 and UT10) are not detectable with static information alone. In both of them, the dataset is produced from information read from an external file. The code itself does not provide any hints about the shape of the dataset after it is read. As a result, the analysis cannot identify the incompatibility between the shape of the dataset and the tensor that will hold that data at run-time.

The analysis is compiled by the Soufflé [24] Datalog engine into an optimized C++ program and binary executable. The analysis code comprises several hundred non-trivial Datalog rules, therefore optimizing compilation is time-consuming, at 680sec. (All timings are from a single thread of a laptop with an Intel Core i7-3612QM 2.10GHz CPU, with 16GB of RAM.) Compilation is only performed once per analysis configuration, however, and the resulting analysis is highly efficient. For the input programs, the average analysis running time is just 0.26sec (median: 0.18sec, max: 0.49sec for UT4).

7.3 Precision

Pythia contains many precision enhancements – e.g., levels of context sensitivity, and a more detailed value abstraction. We already saw earlier, in Figure 1 an example of the impact of precision enhancements. We demonstrate the effect on the input programs of the evaluation set in Figure 2.

The figure shows seven input programs whose analysis precision changes for different configurations. (Input programs not shown either show no imprecision for any configuration or are the 3 for which our analysis misses the bug.) Precision is captured in three metrics: instances of imprecise tensor arguments (compared to the full achievable precision), false positives in analysis warnings, and instances of imprecise shapes. A check mark in the figure implies no imprecision for any metric. The four configurations of the analysis for each benchmark are:

- Configuration 1: context-insensitive (insens)
- Configuration 2: 1-call-site-sensitive (1call)
- Configuration 3: 1-call-site-sensitive + context-sensitive heap (1callH)
- Configuration 4: 1callH + full-tensor-precision.

Table 2 "Unaligned Tensor (UT)" input programs (each entry corresponds to 2 programs: a fixed and a buggy version) and analysis reports. No UT14 exists in the input set. For the analyses reports, "—" designates an analysis terminating but reporting no bugs, "X" designates an analysis not terminating due to an exception. The URLs to access the programs are obtained by concatenating the following URL prefixes with the suffix for each program shown in the table's second column.
Github: https://github.com/ForeverZyh/TensorFlow-Program-Bugs/blob/master/StackOverflow/
StackOverflow: https://stackoverflow.com/q/

Case Study	URLs	Description	Pythia Report	Ariadne Report
UT1	GitHub: UT-1/38167455-buggy/mnist.py StackOverflow: 38167455	Reshape Operation	Warning	X
UT2	GitHub: UT-2/43067338-buggy/multiplication.py StackOverflow: 43067338	`matmul` Incompatible Dimensions	Error	—
UT3	GitHub: UT-3/35451948-buggy/image_set_shape.py StackOverflow: 35451948	Invalid call to `set_shape`	Error	—
UT4	GitHub: UT-4/44124668-buggy/experiment.py StackOverflow: 44124668	Fed data don't match shape	Error	X
UT5	GitHub: UT-5/43676638-buggy/mnist.py StackOverflow: 43676638	Fed data don't match shape	—	X
UT6	GitHub: UT-6/35295191-buggy/word_representation.py StackOverflow: 35295191	`matmul` Incompatible Dimensions	Error	—
UT7	GitHub: UT-7/34079787-buggy/playing.py StackOverflow: 34079787	Variable's `initial_value` has unspecified shape	—	—
UT8	GitHub: UT-8/34908033-buggy/multiply.py StackOverflow: 34079787	`matmul` Incompatible Dimensions	Error	X
UT9	GitHub: UT-9/40574552-buggy/neural.py StackOverflow: 34908033	Incorrect operand shapes in `softmax_cross_entropy_with_logits`	Error	X
UT10	GitHub: UT-10/36343542-buggy/tflin.py StackOverflow: 36343542	Fed data don't match shape	—	X
UT11	GitHub: UT-11/41192992-buggy/image.py StackOverflow: 41192992	Fed data don't match shape	Error	—
UT12	GitHub: UT-12/43285733-buggy/mnist.py StackOverflow: 43285733	Reshape Operation	Warning	X
UT13	GitHub: UT-12/42191656-buggy/linear.py StackOverflow: 42191656	Misuse of `argmax` operation	Warning	—
UT15	GitHub: UT-15/38447935-buggy/fitting.py StackOverflow: 38447935	Broadcasting operation	Warning	—

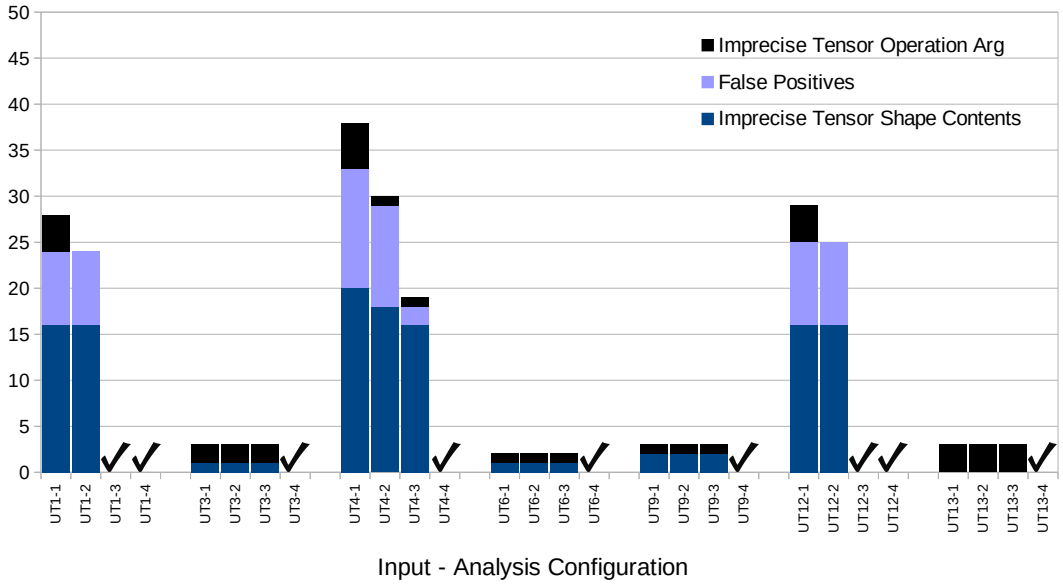

Figure 2 Chart: Imprecision metrics under different configurations. Lower is better, check mark is perfect precision. The y axis shows cumulative instances of three imprecision metrics: instances of imprecise tensor arguments, false positives in analysis warnings, and instances of imprecise shapes.

Among the programs presented in the chart we notice similar behavior for programs UT-3, UT-6, UT-9 and UT-13. For these and for the first 3 configurations, we notice minor imprecision but still no false positives. This is because the programs do not feature any calls to user-defined functions, but imprecision is introduced by other features, such as the use of `set_shape`. The introduction of full-tensor-precision removes any imprecisions.

The 3 remaining programs present large imprecision when using our less precise analysis configurations, resulting in many false positives. This is because, similar to the bug featured in Figure 1, the neural network is built using user-defined wrapper-functions, making context sensitivity necessary in order to achieve a highly-precise analysis. (These are also among the longest programs at around 100 or more lines.)

For instance, in UT-4,[6] Pythia can correctly deduce the shapes of the tensors generated by two separate calls to function `generate_unit_test`, shown below: (This also showcases the analysis support for list comprehensions.)

```
def generate_unit_test(length):
  return [np.random.normal(0, 0.1, [56, 56, 3])
    for _ in range(length)],
    [random.randint(0, 9) for _ in range(length)]
```

In this input program, a false positive warning persists until the full-tensor-precision value abstraction is employed.

[6] `https://github.com/ForeverZyh/TensorFlow-Program-Bugs/blob/master/StackOverflow/UT-4/44124668-buggy/experiment.py`

7.4 Other bugs found and missed

We next discuss selected cases of bugs reported or missed in the input dataset. Several interesting cases are already represented in our earlier examples, so we will not discuss them further. Namely, the reshape operation warning in case study UT1 has already been covered by the example of Section 6. The broadcasting operation warning in case study UT15 is analogous to Example 3 in Section 3. The incompatible dimensions error in `matmul`, appearing in case studies UT2, UT6 and UT8, is captured by Example 1 in Section 3.

Case Study UT3

Invalid call to `set_shape`.

```
import tensorflow as tf
import numpy as np

x = tf.placeholder(tf.float32, [None])
x.set_shape([1028178])
y = tf.identity(x)
y.set_shape([478, 717, 3])
X = np.random.normal(0, 0.1, 1028178)
```

The `set_shape` operation is used to provide additional, more concrete information about the shape of a tensor. In UT3, initially the shape of `x` is `[None]`. The first call to `set_shape` succeeds and establishes that the shape of `x` is `[1028178]`. The call to `identity` produces a tensor of the same shape as `x` and assigns it to `y`. The next call to `set_shape` is erroneous for two reasons. First, it attempts to specify an already established concrete shape. Secondly, even if the shape of `y` had not been already established by the first call to `set_shape`, the call would still fail since the dimensions of `[None]` and `[487, 717, 3]` are incompatible.

Case Study UT9

Incorrect operand shapes in `softmax_cross_entropy_with_logits` call.

```
import tensorflow as tf
import numpy as np
import random

n_feature = 10
n_data = 500
data = np.random.normal(0, 0.1, [n_data, n_feature])
label = [[random.randint(0, 1) for _ in range(n_data)]]

sizeOfRow = len(data[0])
x = tf.placeholder("float", shape=[None, sizeOfRow])
y = tf.placeholder("float")

prediction = neuralNetworkModel(x)
# using softmax function, normalize values to range(0,1)
error = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(
    logits=prediction, labels=y))
...
```

In UT9 `softmax_cross_entropy_with_logits` is applied, which performs two operations. First, it applies the `softmax` function to denormalized log probabilities, i.e, the `logits` tensor, of shape [*batch_size*, *number_of_labels*], to produce linear probabilities normalized to 1. It then computes the cross-entropy error of discrete classification based on the results of `softmax` and the ground truth classification of the training set that is held in the `labels` tensor, which is expected to have the same dimensions as `logits`. In this case, the shape of `logits` is `[500, 2]`, since there are 500 entries in the dataset and 2 labels (1 and 0), while the shape of `labels` is `[1,500]`.

Case Study UT7

Variable's initial value has unspecified shape. The bug in UT7 is representative of confusion regarding the TensorFlow execution model. It is also one that our analysis fails to capture, due to lack of modeling of the corresponding calls.

```
import tensorflow as tf
import random
import numpy as np

class Play:
  def __init__(self, input_data, labels):
    # the input shape is (batch_size, input_size)
    input_size = tf.shape(input_data)[1]

    # labels in one-hot format have shape (batch_size, num_classes)
    num_classes = tf.shape(labels)[1]
    stddev = 1.0 / tf.cast(input_size, tf.float32)

    w_shape = tf.stack([input_size, num_classes])
    normal_dist = tf.truncated_normal(w_shape, stddev=stddev, name='normaldist')
    self.w = tf.Variable(normal_dist, name='weights')
    print(self.w)

n_feature = 10
n_classes = 7
play = Play(tf.placeholder(tf.float32, [None, n_feature]),
    tf.placeholder(tf.int32, [None, n_classes]))
```

Recall from Section 2 that TensorFlow programs work by first setting up a data-flow pipeline of operators, and then executing it by feeding data. The Python code effectively *generates* a TensorFlow pipeline, before evaluating it. In case study UT7, the programmer incorrectly uses `tf.shape(input_data)` and `tf.shape(labels)`, while probably intending to use `input_data.get_shape()` and `labels.get_shape()`.

That is, the programmer intends to retrieve the shape of the dynamic data that will be fed into the TensorFlow pipeline. Instead, the erroneous calls retrieve the shape of the yet-unpopulated variables `input_data` and `labels`. The Python dynamic typing and TensorFlow tolerance conspire to propagate this error until it results in a shape mismatch later: each of the two erroneous calls returns an unevaluated one-dimensional tensor, which when dereferenced (via `[1]`) returns a to-be-evaluated integer. This integer is considered to be a zero-dimension tensor (`[]`), which becomes the value of `input_size` (and similarly for `num_classes`). TensorFlow then deduces that shape `w_shape` has value `[None, None]` as it is

the result of a `tf.stack` operation on two zero-dimension tensors, producing a 1-dimension tensor with shape `[2]`. However, the `tf.Variable` operation does not allow an unspecified shape as input, thus causing a crash.

Case Study UT11

Fed data don't match shape. The next input program indicates the handling of other operators (namely, `transpose`) as well as the ease with which a programmer can lose track of tensor shapes.

```
from tensorflow.contrib.keras.api.keras.preprocessing import image
import tensorflow as tf
import numpy as np

x = image.load_img(img_path, target_size=(250, 250))

x = image.img_to_array(x)
x_expanded = np.expand_dims(x, axis=0)
x_expanded_trans = np.transpose(x_expanded, [0, 3, 1, 2])

X = tf.placeholder(tf.float32, [None, 250, 250, 3])
sess = tf.Session()
sess.run(tf.global_variables_initializer())
print(sess.run(X, feed_dict={X: x_expanded_trans}))
```

In UT11, initially `x` is an image of shape `[250, 250, 3]` (because an image has 3 color channels). It is then converted to a NumPy array of the same shape, which is subsequently expanded to an array of shape `[1, 250, 250, 3]`, essentially creating a fake `batch_size` dimension. The array is then transposed to an array of shape `[1, 3, 250, 250]`. Finally, the code feeds the transposed array to a placeholder tensor of a different incompatible shape, `[None, 250, 250, 3]`, thus causing an error.

Case Study UT13

Misuse of *argmax* operation. Input program UT13 provides another example of a warning by our analysis that does not correspond to a run-time error, yet is highly likely to be a bug (as it is, in this case).

```
import tensorflow as tf

Y = tf.placeholder(tf.float32, shape=[4, 1], name='y')
...
Z = tf.argmax(Y, axis=1)
...
```

In UT13 `argmax` is applied to a tensor with shape `[4, 1]`. The `argmax` operation returns the index with the largest value along the specified `axis`. However, the second dimension of tensor `Y` is 1. In this case `argmax` returns a tensor with shape `[4]` with all 4 values being 0, since the *dimension size* in `1` is just 1. This is likely not the intended use of the `argmax` operation so we issue a warning, predicting that this promotion of values is not what the user aimed to accomplish.

7.5 Comparison with the state-of-the-art

The recent Ariadne tool [12] is, to our knowledge, the only static analysis tool that attempts to find shape bugs in TensorFlow code. We ran the latest version of Ariadne[7] on our setup using the Language Server Protocol client for the Sublime text editor.

Table 2 shows the results of both tools for our dataset. The Ariadne tool reports 0 bugs. Furthermore, for half of the programs in our dataset, the Ariadne analysis ends with an exception, while for the other half it terminates successfully, reporting other information using the LSP protocol (such as call-site information) but no warning. These results can be explained by Ariadne's limited support for tensor operations and by its not performing whole-program value-flow reasoning. For instance, Ariadne supports operators `reshape`, `set_shape`, convolution, and "node", of which only `reshape` works fully. Pythia supports many more operations, such as `equal`, `add`, `multiply`, `matmul`, `argmax`/`argmin`, `transpose`, `expand_dims`, several pooling operations, and many shape pass-through operations. In Ariadne, tensors can be created using the `tf.placeholder` function. We also support `tf.constant`, `tf.Variable`, `tf.ones`, etc.

7.6 Threats to Validity

The largest threat is to external validity. Our findings may not generalize to other TensorFlow programs, especially of larger size. However, the benchmarks we examined are a prior and independently-identified set, collected from real-world reports. The programs are already large enough for context sensitivity and heap modeling to matter (as shown in Section 7.3).

8 Related Work

The space of checkers for machine learning programs is mostly populated by testing techniques [38, 51, 55]. Other approaches aid in the debugging and validation of machine learning programs. PALM [26] produces simplified decision-tree based meta models to facilitate the mapping of failed predictions to subsets of the training data. On the other hand LAMP [31] produces quantitative measurements that maps the impact of each input to each output in graph machine learning algorithms in an efficient way using partial derivatives. MODE [32] applies similar techniques for measuring the impact of each feature in Neural Networks.

The recent Ariadne tool [12] demonstrates an application of static analysis technology to TensorFlow, but neither models many TensorFlow operators, nor performs whole-program value-flow reasoning. This limits Ariadne's applicability to artificial examples, with manually-planted bugs, and to Python input programs of very limited form – e.g., as discussed in Section 7.5 and shown in table 2, the system cannot run or produce useful results on *any* of our input benchmarks.

General program analysis tools for Python have been developed. These mostly aim to find type errors. Invariably, such frameworks restrict the features of Python, since the language is highly dynamic and its full static analysis with good precision is impossible. For instance, even determining which file is imported when an import statement is executed can be undecidable. RPython [5] is a statically typed subset of the Python language designed for writing partially evaluated interpreters. All metaprogramming features (including `eval` and metaclasses) may be used during the initialization of the Python classes. RPython is best compared with a statically typed version of Python. Retrofitting type systems to dynamic languages is a fairly common strategy, and examples include preemptive type

[7] Downloaded from the official site: `https://wala.github.io/IDE/`.

checking for Python [16], DRuby for Ruby [14] or a type system for Erlang [34]. JavaScript has probably attracted the most attention in this space and there are many more examples of type systems for it [10, 11, 19, 22, 29]. These systems are used either for speeding up JavaScript implementations or for type checking during development. Due to the complexity of the underlying problem, many authors (including ourselves) have found it more fruitful to concentrate on type checking or bug finding for specific domains. Related examples in the wild include a system [4] for Ruby on Rails or the work of [28] for static detection of JQuery errors in JavaScript by identifying inconsistencies between the actual page structure and query expectations.

The space of static analysis tools for Python is relatively sparse. Python Taint [45] is a static analysis tool for detecting security vulnerabilities. It uses standard data-flow techniques, and can do some interprocedural analysis. However, its interprocedural reasoning is limited: it looks for a definition of a function for a call using its name, rather than handling function pointers and object semantics, as needed even for simple realistic examples.

Gorbovitski et al. [15] developed a context-sensitive, flow-sensitive alias analysis for Python for program optimization. They offer several significant insights on the precision needed for dynamic languages. The analysis appears sophisticated but we have not found an available implementation for reuse.

Other tools are shallow code quality checkers or lint tools; examples are Pylint [44], pycodestyle [41], pyflakes [43], Flake8 [13], pydocstyle [42], jedi [21], bandit [7] and mccabe [35]. Prospector [40] combines several of these tools. These tools are all local analyses, for instance, mccabe focuses on the syntactic code complexity of single functions and others focus on code style issues.

Another bug detection approach includes analyses that are dynamic, yet generalize from concrete executions. Xu et al. [56] developed such a predictive analysis for Python, detecting more general bugs, such as Attribute Errors and Type Errors, and Unicode Encode/Decode Errors which are specific to web applications.

Finally, although the work we describe is applied to TensorFlow, the principles described may apply to other scientific computing languages and extensions such as SAC [17] or LAPACK [6].

9 Conclusions

We presented a static analysis approach for detecting shape bugs in TensorFlow programs. The analysis models value-flow in Python programs and closely tracks the rich shape-transformation semantics of TensorFlow operators. The result is the first concrete demonstration of the applicability of static analysis for detecting realistic bugs in the TensorFlow domain. The analysis is highly efficient and very effective over an independently-collected set of input programs that sample the universe of real-world TensorFlow bugs.

References

1 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: `https://www.tensorflow.org/`.

2 Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frederic Bastien, Justin Bayer, Anatoly Belikov, and others . Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint*, January 2016. arXiv:1605.02688.

3 Miltiadis Allamanis, Earl Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51, September 2017. doi:10.1145/3212695.

4 Jong-hoon An, Avik Chaudhuri, and Jeffrey S. Foster. Static Typing for Ruby on Rails. In *Proceedings of ASE*, pages 590–594, November 2009. doi:10.1109/ASE.2009.80.

5 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. Rpython: A step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM. doi:10.1145/1297081.1297091.

6 Ed Anderson, Zhaojun Bai, Jack Dongarra, A. Greenbaum, A. McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, Christian Bischof, and Danny C. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Washington, DC, USA, 1990. IEEE Computer Society Press.

7 bandit. https://github.com/openstack/bandit. Accessed: 2020-01-06.

8 Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proc. of the 18th International Symp. on Software Testing and Analysis*, ISSTA '09, pages 1–12, New York, NY, USA, 2009. ACM. doi:10.1145/1572272.1572274.

9 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009. ACM.

10 Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. Type inference for static compilation of JavaScript. *SIGPLAN Not.*, 51(10):410–429, October 2016. doi:10.1145/3022671.2984017.

11 Wontae Choi, Satish Chandra, George C. Necula, and Koushik Sen. Sjs: A type system for JavaScript with fixed object layout. In Sandrine Blazy and Thomas Jensen, editors, *SAS*, volume 9291 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2015. URL: http://dblp.uni-trier.de/db/conf/sas/sas2015.html#ChoiCNS15.

12 Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. Ariadne: Analysis for machine learning programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 1–10, New York, NY, USA, 2018. ACM. doi:10.1145/3211346.3211349.

13 Flake8. https://github.com/PyCQA/flake8. Accessed: 2020-01-06.

14 Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of OOPSLA*, pages 283–300, 2009. doi:10.1145/1639949.1640110.

15 Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay K. Tekle. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 27–42, New York, NY, USA, 2010. ACM. doi:10.1145/1869631.1869635.

16 Neville Grech, Bernd Fischer, and Julian Rathke. Preemptive type checking. *Journal of Logical and Algebraic Methods in Programming*, 101:151–181, 2018. doi:10.1016/j.jlamp.2018.08.003.

17 Clemens Grelck and Sven-Bodo Scholz. SAC – a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, August 2006. doi:10.1007/s10766-006-0018-x.

18 Salvatore Guarnieri and Benjamin Livshits. GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In *Proc. of the 18th USENIX Security Symposium*, SSYM' 09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1855768.1855778.

19 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM. doi:10.1145/2254064.2254094.

20 Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with Datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, ECOOP '06, pages 2–27. Springer, 2006.

21 jedi. https://github.com/davidhalter/jedi. Accessed: 2020-01-06.

22 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-03237-0_17.

23 Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's multilingual neural machine translation system: Enabling zero-shot translation. *Transactions of the Association for Computational Linguistics*, 5:339–351, 2017. URL: https://transacl.org/ojs/index.php/tacl/article/view/1081.

24 Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

25 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '13, New York, NY, USA, 2013. ACM.

26 Sanjay Krishnan and Eugene Wu. Palm: Machine learning explanations for iterative debugging. In *Proceedings of the 2Nd Workshop on Human-In-the-Loop Data Analytics*, HILDA'17, pages 4:1–4:6, New York, NY, USA, 2017. ACM. doi:10.1145/3077257.3077271.

27 Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proc. of the 24th Symp. on Principles of Database Systems*, PODS '05, pages 1–12, New York, NY, USA, 2005. ACM. doi:10.1145/1065167.1065169.

28 Benjamin S. Lerner, Liam Elberty, Jincheng Li, and Shriram Krishnamurthi. Combining form and function: Static types for jquery programs. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 79–103, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-39038-8_4.

29 Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Tejas: Retrofitting type systems for JavaScript. *SIGPLAN Not.*, 49(2):1–16, October 2013. doi:10.1145/2578856.2508170.

30 Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proc. of the 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993567.

31 Shiqing Ma, Yousra Aafer, Zhaogui Xu, Wen-Chuan Lee, Juan Zhai, Yingqi Liu, and Xiangyu Zhang. Lamp: Data provenance for graph based machine learning algorithms through derivative computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 786–797, New York, NY, USA, 2017. ACM. doi:10.1145/3106237.3106291.

32 Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. Mode: Automated neural network model debugging via state differential analysis and input selection. 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 175–186, New York, NY, USA, 2018. ACM. doi:10.1145/3236024.3236082.

33 Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proc. of the ACM SIGSOFT International Symp. on the Foundations of Software Engineering*, FSE '13, pages 499–509. ACM, 2013. doi:10.1145/2491411.2491417.

34 Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proceedings of ICFP*, pages 136–149, August 1997. doi:10.1145/258949.258962.

35 mccabe. https://pypi.python.org/pypi/mccabe. Accessed: 2020-01-06.

36 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM. doi:10.1145/1133981.1134018.

37 Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

38 Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 1–18, New York, NY, USA, 2017. ACM. doi:10.1145/3132747.3132785.

39 Simon J. D. Prince. *Computer Vision: Models, Learning, and Inference*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.

40 Prospector. https://prospector.readthedocs.io/en/master/. Accessed: 2020-01-06.

41 pycodestyle. http://pep8.readthedocs.org/en/latest/. Accessed: 2020-01-06.

42 pydocstyle. https://github.com/PyCQA/pydocstyle. Accessed: 2020-01-06.

43 pyflakes. https://launchpad.net/pyflakes. Accessed: 2020-01-06.

44 Pylint. http://www.pylint.org/. Accessed: 2020-01-06.

45 Python Taint. https://github.com/python-security/pyt. Accessed: 2020-01-06.

46 Thomas W. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.

47 Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program flow analysis: theory and applications*, chapter 7, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

48 Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, may 1991.

49 Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015. doi:10.1561/2500000014.

50 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 196–232. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_8.

51 Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 303–314, New York, NY, USA, 2018. ACM. doi:10.1145/3180155.3180220.

52 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proc. of the 1999 Conf. of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 125–135. IBM Press, 1999. URL: http://dl.acm.org/citation.cfm?id=781995.782008.

53 John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, pages 97–118. Springer, 2005. doi:10.1007/11575467_8.

54 John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the 2004 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM. doi:10.1145/996841.996859.

55 Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *J. Syst. Softw.*, 84(4):544–558, April 2011. doi:10.1016/j.jss.2010.11.920.

56 Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. Python predictive analysis for bug detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 121–132, New York, NY, USA, 2016. ACM. doi:10.1145/2950290.2950357.

57 Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in Datalog. In *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '14, pages 239–248, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594327.

58 Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 129–140, New York, NY, USA, 2018. ACM. doi:10.1145/3213846.3213866.