Ariadne: Analysis for Machine Learning Programs

Fabian Schiebel

University of Paderborn, Institute for Computer Science,
fabianbs@mail.uni-paderborn.de

Seminar: Advanced Techniques in Software Analysis

June 17, 2020

Abstract. Most programs for machine learning are written in dynamic languages like *Python*. Due to their lack of strong static typing, many errors are found only at runtime or not at all. Such errors often appear very late in the software development lifecycle or even after deployment. To overcome this, a static code analysis can be used. Thus, this report presents a static code analysis that can be used to find errors in machine learning programs written in *Python* which use the *TensorFlow* library. The analysis consists of a custom type system for tracking tensors and a dataflow analysis to verify their usage.

1 Introduction

Machine learning is an emerging area in computer science. Therefore, many applications like Google, Facebook, etc. rely on the correctness of machine learning programs. This leads to two general problems. At first, most machine learning programs are written in dynamic languages like *Python*. Hence, most errors are only discovered at runtime, when the erroneous code part gets executed. Moreover, machine learning involves much nondeterminism due to probabilistics that can be executed no matter if it makes sense in the context of its execution. This can lead to legal results, which are nevertheless wrong. Those errors cannot be detected by the *Python* infrastructure and need extensive testing to be found.

Thus, errors can only be found, if the machine learning program is fully implemented and executable, which is very late in the software development lifecycle where fixing errors is very expensive. So, Dolby et al. [4] developed the static analysis tool *Ariadne* for finding errors in machine learning programs early. They focus on programs written in *Python* which use the commonly used machine learning library *TensorFlow* [1]. For specifying the static analysis they use and extend the WALA [3] analysis framework and developed a custom type system for tracking tensors, that are mathematical constructs used for machine learning (Refer 2.1). There are several other static analysis frameworks, like Soot [7] for example, but they do not support analyzing dynamically typed languages like

Python. In contrast, WALA supports efficient and precise analysis of JavaScript code. Hence, Dolby et al. [4] extend the existing WALA framework to support a subset of *Python* that is used for writing machine learning programs.

In the following sections, necessary background information is given (2) and after that the approach of Dolby et al. [4] is presented in detail.

2 Background

To understand the concepts of the presented static analysis *Ariadne*, some background information is required. This section summarizes the most important background aspects.

2.1 TensorFlow

TensorFlow [1] is a widely used framework for writing machine learning programs in various languages e.g. *Python*. A commonly used tool in *TensorFlow* are so-called tensors. They are mathematical constructs, very similar to n-dimensional matrices. Among other features, *TensorFlow* offers many possibilities to compute with tensors, for example to change the dimensions using the `reshape` operation or to add tensors together, convert them to an array, and others.

2.2 WALA

T.J. Watson Libraries for Analysis (WALA) [3] is a framework written in *Java* which provides static code analysis solvers for analyzing programs written in an intermediate representation (WALA IR). WALA already provides an API to convert source code from several programming languages e.g. *Java*, *JavaScript*, etc. into WALA IR and thus to analyze programs written in those languages.

To analyze the IR, WALA provides interprocedural dataflow analyses using an extension of the IFDS framework [9] which makes it precise and high performant. Furthermore, it provides several helper analyses like pointer analysis and callgraph analysis. The helper analyses' results can be used by the IFDS solver to enhance the precision of the dataflow analysis results.

2.3 Pointer Analysis

Pointer analysis is used to decide whether two variables may refer to the same object in memory, i.e. they alias. If two variables x and y alias, then every change in x will be automatically reflected in y and vice versa. Moreover, every dataflow into x also implicitly flows into y; thus dataflow analyses need to have pointer information in order to be sound.

Unfortunately, pointer analysis is not distributive which makes it impossible to

use efficient analysis frameworks like IFDS [9] directly. So, many pointer analyses use information from the language's type system to reduce the problem space, as variables of incompatible types cannot alias. However, in dynamic languages like *Python*, those optimizations cannot be applied directly, since the type of a variable is only available at runtime.

2.4 Callgraph Analysis

Call graph analysis is very similar to pointer analysis, but it focuses on resolving dynamic callsites. Any interprocedural dataflow analysis eventually needs to deal with function calls. It mostly does so by resolving the callee function, analyzes it and then uses the results to compute the dataflow facts in the caller. Hence, it is necessary to be able to find all callee functions even if the callsite contains a virtual method which may get overwritten by subclasses in an object oriented environment.

In *Python*, callgraph analysis is particularly difficult because there are not only virtual functions, but also mutable function-pointers and higher-order functions. Moreover, most callgraph construction algorithms use the language's static type system to make the callgraph more precise and to speedup the analysis. However, as for pointer analysis, this is hard for dynamically typed languages like *Python*.

3 Ariadne

This section explains the static code analysis *Ariadne* in detail. It starts with a description on how *Python* programs are mapped to the WALA framework, followed by the definition of the tensor typesystem and finally the dataflow analysis.

To explain how these constructs work, a shortened version of the example program (refer Listing 1.1) from the paper [4] is used. The main functionality of this program snippet is the `reshape` operation at the bottom. It takes a tensor and transforms it to a new shape while preserving the total size. In order to correctly perform this operation, the input tensor `x` needs to be compatible with the shape parameter `[-1, 28, 28, 1]`.

Since *Python* is a dynamically typed language, this requirement can only be checked by the developers by carefully reading the comments. The goal of the analysis is, to automatically verify that this `reshape` operation (and other *TensorFlow* APIs) is legal on all execution paths.

Listing 1.1. Example: Snippet of a program training for image recognition

```
import tensorflow as tf
...
def conv_net(x_dict, n_classes, dropout, reuse, is_training):
# Define a scope for reusing the variables
with tf.variable_scope('ConvNet', reuse=reuse):
x = x_dict['images']
# MNIST data input is a 1-D vector of
#   784 features (28*28 pixels)
# Reshape to match picture format
#   [Height x Width x Channel]
# Tensor input become 4-D:
#   [Batch Size, Height, Width, Channel]
x = tf.reshape(x, shape=[-1, 28, 28, 1])
...
```

3.1 Mapping Python to WALA

For being able to analyze *Python* programs with WALA, the program under analysis first needs to be transformed into a representation which WALA can work with. This representation is the WALA Intermediate Representation and can be automatically generated from a WALA interface called Common Abstract Syntax Tree (CAst). So, the *Python* program must first be converted to an equivalent CAst representation which then gets automatically transformed to WALA IR. The CAst interface is written in *Java*. Luckily, there exists a *Python* interpreter *Jython* [6] written in *Java*. So Dolby et al. [4] use *Jython* to parse the *Python* program under analysis and transform the *Jython* AST to the CAst interface.

This transformation is often very easy, because the most constructs like **`if`**, **`for`**, etc. are already directly representable in CAst with special nodes. However, some *Python* constructs like **`import`** statements are not directly representable in the CAst, so Dolby et al. [4] extended the CAst with custom nodes representing those constructs.

Finally, WALA does not perform the analysis directly on the CAst, but on an intermediate representation (IR). So, before starting the analysis, WALA transforms the input CAst to its IR. Since the CAst of the *Python* program under analysis contains custom nodes and since other constructs have special semantics, Dolby et al. [4] needed to extend the WALA IR too. Especially method calls were hard to model. This is, because in *Python* methods are assignable objects even if they are bound to a receiver object. To resolve this, the authors have used so called trampoline instructions to create a function-pointer which captures the **`self`** reference.

Listing 1.2. Different ways of calling a function in *Python* (vgl. [4])

```
class Foo:
  def f(self, x):
    ...

x = Foo()
# Call x.f the normal way
x.f(3)
# Call x.f through a function-pointer
y = x.f
y(3)
# Call x.f explicitly
Foo.f(x, 3)
# Reassign x.f and call it explicitly
x.f = Foo.f
x.f(x, 3)
```

In the example 1.2, the first two calls would be represented in the WALA IR by calling the trampoline which automatically binds `self` to `x`. The last two calls instead would be represented by calling the normal non-trampoline function `Foo.f`.

3.2 Tensor Typesystem

To enable a dataflow analyses of *Python* programs that use the *TensorFlow* framework, the analysis needs to understand, what variables hold tensors and which dimensions those tensors have. Thus, Dolby et al. [4] developed a type system for *Python* which is specialized on tensors.

The types π are defined inductively with the normal *Python* types as base case including record types and function types. Then the term $[d_1, \ldots, d_n \ of \ \pi]$ defines a tensor type with elements of type π and the dimensions $d_1, \ldots, d_n$ for a $n \in \mathbb{N}$

In the example (Listing 1.1), the variable `x` after the assignment `x = x_dict['images']` is of the tensor type $[batch, y(28) * x(28) \ of \ channel]$ provided that the parameter `x_dict` is of a record type $\{images : [batch, y(28) * x(28) \ of \ channel]\}$. Here, *batch* and $y(28) * x(28)$ are the two tensor dimensions and *channel* is a numeric type. The second dimension is special in that it is composed of two other dimensions y and x which are both of length 28.
As shown in the example, each dimension can be attached with an optional label; in this case *batch*, x and y. This label can be used to give each dimension a meaning in order to prevent unintentional reordering of dimensions.

This enables the analysis to verify that the operation
`x = tf.reshape(x, shape=[-1, 28, 28, 1])` is valid and correctly decomposes the second dimension resulting in $[batch, x(28), y(28), 1\ of\ channel]$ as type for `x` after the reshape operation. Would the shape instead be `[-1, 28, 29, 1]`, the types would not match, since the original size of the second dimension is $28 \cdot 28 = 784$ which does not equal the total size of the new decomposed dimensions $28 \cdot 29 \neq 784$.

Finally, the type system contains an artificial $\top$ type which is used for all irrelevant types. Variables of type $\top$ are ignored by the dataflow analysis.

3.3 Dataflow Analysis

The actual analysis consists of several parts. It starts with a WALA callgraph- and pointer analysis which produce a dataflow graph $\mathfrak{G}$. Its nodes V are the set of program variables. Two nodes $x, y \in V$ are connected by an edge, iff there is a dataflow from y to x, denoted by $x \prec y$. Additionally, the dataflow graph contains points-to information $S(v)$ for each variable $v \in V$ representing all objects which can be held by v.

This dataflow graph is used to define the typing analysis which assigns every variable $v \in V$ a set of possible types $T(v)$. The analysis considers several cases: For input variables, e.g. function parameters, the type cannot be deduced, so it must be declared explicitly. Those type information are the seeds of the analysis. Of course, type information propagates with every dataflow. Thus, given two variables $x, y \in V$ with a dataflow $x \prec y$. Then x can only be of a type which is valid for y. Hence, it holds $T(x) \subseteq T(y)$

Finally, the analysis needs to deal with the library calls into *TensorFlow*, e.g. calls to `reshape`, etc. The problem is that usually, the source code of *TensorFlow* is not available to the analysis. So, the analysis needs to model each relevant API function of *TensorFlow* explicitly. Given the example in Listing 1.1, there is a call to the *TensorFlow* function `reshape`.
To model the influence of `reshape` on the typing analysis, it is necessary to define which tensor types can be reshaped into which other types. Let t_1, t_2 be tensor types. Then it holds $t_1 \doteq t_2$, if and only if tensors of type t_1 can be reshaped to tensors of type t_2. Assuming, the second argument to the `reshape` operation represents a tensor type, its influence on the typing analysis can be defined as follows: Let $x \in V$ be a tensor variable which is used as first argument to a call to `reshape` with dimension argument z and $y \in V$ a variable with a dataflow from the result of the reshape operation to y ($y \prec \texttt{reshape}(x, z)$). It can be observed that the new tensor shape can be deduced from the second argument z; concretely, from its possible values $S(z)$. So it holds $T(y) \subseteq \{t \in S(z) \mid T(x) \doteq t\}$.

4 Evaluation

Dolby et al. [4] have modeled the *TensorFlow* functions `reshape`, `conv2d`, `conv3d` and `placeholder` in their dataflow analysis. To check whether the tool *Ariadne* works as expected, they tested it on six *Python* programs for image recognition.

It turned out that *Ariadne* was able to analyze all of these programs and verified that they used the analyzed *TensorFlow* APIs correctly. The analysis did not give any false positives, so it was very precise. However, they do not make any assumptions on the false negative rate which is probably very high as *Ariadne* was only modeled for a very small fraction of the *TensorFlow* API.
Furthermore, Dolby et al. [4] were able to run the analysis in only a few seconds. Hence, *Ariadne* works very well on the parts it was modeled for.

5 Related Work

There are several tools which provide static analyses of *Python* programs. Most of them, like *Pylint* [2] only perform code quality checks and do not perform a whole-program analysis.
Furthermore, there is one tool *Python Taint* [8] which can run dataflow analyses for finding security vulnerabilities in *Python* code. However, their interprocedural analysis is highly unsound because they perform function resolution by name rather than by function pointers, as is done in Ariadne.
Moreover, the tool *Nagini* [5] can automatically verify non-trivial properties of *Python* programs that are annotates with special `assert` statements. Those statements are used to formulate pre- and postconditions of functions and loop invariants. Unlike *Ariadne*, *Nagini* requires the whole program to be type annotated, but does not support tensor types. So it covers different uses cases than *Ariadne*.
There are also other tools for static analysis, but they use machine learning in order to implement the analysis, instead of analyzing machine learning code.

6 Conclusion

Dolby et al. have developed a static analysis to statically typecheck *Python* programs that use the machine learning framework *TensorFlow*. For this analysis, they first modeled the syntax and semantics of *Python* programs in the WALA IR. They created a custom type system for tracking tensors in *Python* and finally defined the actual dataflow analysis.

They have evaluated the analysis on several machine learning programs and verified (for a small subset of the *TensorFlow* API) that these programs used the APIs correctly. As the analysis was very precise and efficient, it can be said

that this concept of verifying the correctness of machine learning programs using static analysis works very well. Thus, it is worth to model more *TensorFlow* APIs and even more machine learning frameworks in this way in order to find errors early in the software development lifecycle where fixing them is cheap.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems. 2015.
[2] Python Code Quality Authority. Pylint.
[3] IBM T.J. Watson Research Center. Watson libraries for analysis.
[4] 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*, pages 1–10, 2018.
[5] Marco Eilers and Peter Müller. Nagini: a static verifier for python. In *International Conference on Computer Aided Verification*, pages 596–603. Springer, 2018.
[6] Jim Hugunin and Barry Warsaw. Jython: Python for the java platform.
[7] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. Soot-a java bytecode optimization framework. In *Cetus Users and Compiler Infrastructure Workshop*, pages 1–11, 2011.
[8] Stefan Micheelsen and Bruno Thalmann. Pyt: A static analysis tool for detecting security vulnerabilities in python web applications.
[9] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.