Learning a Static Analyzer: A Case Study on a Toy Language

Manzil Zaheer
Carnegie Mellon University
manzil.zaheer@cmu.edu

Jean-Baptiste Tristan & Michael Wick & Guy L. Steele Jr.
Oracle Labs
jean.baptiste.tristan@oracle.com

Abstract

Static analyzers are meta-programs that analyze programs to detect potential errors or collect information. For example, they are used as security tools to detect potential buffer overflows. Also, they are used by compilers to verify that a program is well-formed and collect information to generate better code. In this paper, we address the following question: can a static analyzer be learned from data? More specifically, can we use deep learning to learn a static analyzer without the need for complicated feature engineering? We show that long short-term memory networks are able to learn a basic static analyzer for a simple toy language. However, pre-existing approaches based on feature engineering, hidden Markov models, or basic recurrent neural networks fail on such a simple problem. Finally, we show how to make such a tool usable by employing a language model to help the programmer detect where the reported errors are located.

1 Introduction

Can programming language tools, such as static analyzers, be learned from data using deep learning? While research projects trying to use machine learning to design better programming language tools are burgeoning, they all rely on feature engineering (Brun & Ernst, 2004; Kolter & Maloof, 2006; Yamaguchi et al., 2012; Tripp et al., 2014; Raychev et al., 2015; Allamanis et al., 2015; Nguyen & Nguyen, 2015; Gvero & Kuncak, 2015; Long & Rinard, 2016). Unfortunately, feature engineering for programs is difficult and indeed the features often seem ad-hoc and superficial.

This raises the question of whether it could be possible to approach a complicated problem such as static analysis – the automated detection of program properties – from almost raw features. In this paper, our goal is to present a very simple experiment that clearly shows that not only feature engineering can completely fail for even the simplest static analysis task, but that deep learning with neural networks can indeed be successful.

The task in which we are interested is simple: we want to ensure that program variables are defined before they are used. We design a toy language to focus on the problem, and indeed our language is so simple that if it satisfies the aforementioned property, then it is semantically valid. Since programs are sequences of tokens, we experiment with different types of sequence learning methods (Xing et al., 2010). We try feature-based methods in which we extract features from the sequence and then use a classifier to decide whether or not the program is semantically valid. We show that they all fail, including methods that compute a sequence embedding. Then, we try different model-based methods (Lipton, 2015): hidden Markov models (HMM), recurrent neural networks (RNN), and long short-term memory networks (LSTM). Our results show that HMM and RNN do poorly (albeit better than random), while an LSTM is almost perfectly accurate. This finding is somewhat surprising as static analysis is essentially a document classification problem and LSTMs are known to perform poorly on related tasks, such as sentiment analysis (Dai & Le, 2015).

The obvious question about such an experiment is: why would we want to learn a static analyzer for a problem that we know of a perfectly fine engineered solution? The answer is that we want to initiate investigation into the use of deep-learning for program analysis, and our broader hopes are two-fold. First, static analyzers are very complicated and often limited by the amount of false positive and false negatives they generate. In cases where false negatives are unacceptable, a learned static analyzer may not be the right approach. But when the goal is rather to find a good balance between false positives and false negatives, learned static analyzers might be more flexible. Second, as we will briefly show in the paper, learned static analyzers have a resilience to small errors that might lead to more robust tools. Indeed, even though our goal is to detect errors in syntactically valid programs, our tool work despite the presence of small syntactic errors, such as the omission of the semicolon. This resilience to errors is in our opinion a very promising aspect of learned methods for the analysis of programs.

Another key problem with static analysis programs is that to be useful, they need to help the programmer understand what is the cause of the error. In that case, models based on recurrent neural networks really shine because they can be trained to provide such information. Indeed, in our experiment, we show how to use a language model to locate the position of erroneous variables in examples classified by the static analyzer as being wrong. This is very important for practical static analysis since a tool that merely reports the existence of an error in a large code file is not useful.

The paper is organized as follows. In section 2 we introduce the programming language of study and the corresponding static analysis task. In section 3, we review how we created the dataset used to learn the static analyzer and the methods that we have tried. In section 4, we explain how we learn to report error messages to help the programmer understand how to fix an error.

2 A Static Analysis Task

Our goal is to study the following static analysis problem: given a program, is every variable defined before it is being used? Because this problem is undecidable for a Turing complete language, programming languages such as `Java` impose constraints on what is a correct variable initialization. For example, a variable may not in general be defined within only one branch of an if-then-else statement and used afterward since it can be impossible to guarantee which branch will be executed for every run.

In order to better understand whether this is feasible and which methods work, we design a toy language. As an example, in this language, we can write a program that computes the 42th Fibonnacci number as follows.

```
1 v0 = 1; v1 = 1;
2 v2 = 0;
3 while(v2 < 42) {
4   v3 = v1;
5   v1 = v0 + v1;
6   v0 = v3;
7   v2 = v2 + 1;
8 }
9 return v1;
```

If we were to invert lines 4 and 6, then not only would the program be incorrect, but it would be semantically invalid since in the first execution of the loop, variable `v3` has not yet been defined.

In order to precisely explain what the task is, we now briefly present the syntax and semantics of our experimental programming language.

2.1 The Language

We present the syntax of the language in Backus-Naur form in figure 1. The symbols delimited by $\langle\rangle$ are non-terminals while the symbols delimited by ‘’ are terminals. Symbol $\langle program\rangle$ is the starting non-terminal. A program is composed of an optional statement followed by an expression. The statement can be a list of statements, control-flow statements like conditionals or iterations, or the binding of an expression to a variable. The expressions are simple arithmetic expressions. For simplicity, the test expressions used in conditional statements are distinct from the other expressions, which is a simple syntactic way to enforce basic type safety. The integers are simple integer values of the form $[0-9]^+$ while the identifiers are of the form $v[0-9]^+$.

The semantics of our experimental programming language is presented as a big-step operational semantics in figure 2. For simplicity, we only present a subset of the rules. It is composed of

⟨*program*⟩ ::= ‘`return`’ ⟨*expression*⟩ ‘`;`’
  | ⟨*statement*⟩ ‘`return`’ ⟨*expression*⟩ ‘`;`’

⟨*statements*⟩ ::= ⟨*statement*⟩
  | ⟨*statement*⟩ ⟨*statements*⟩

⟨*statement*⟩ ::= ⟨*statements*⟩
  | ⟨*identifier*⟩ ‘=’ ⟨*expression*⟩ ‘`;`’
  | ‘`if`’ ‘`(`’ ⟨*test*⟩ ‘`)`’ ‘`{`’ ⟨*statement*⟩ ‘`}`’ ‘`else`’ ‘`{`’ ⟨*statement*⟩ ‘`}`’
  | ‘`if`’ ‘`(`’ ⟨*test*⟩ ‘`)`’ ‘`{`’ ⟨*statement*⟩ ‘`}`’
  | ‘`while`’ ‘`(`’ ⟨*test*⟩ ‘`)`’ ‘`{`’ ⟨*statement*⟩ ‘`}`’

⟨*test*⟩ ::= ⟨*expression*⟩ ‘=’ ⟨*expression*⟩
  | ⟨*expression*⟩ ‘<=’ ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*multiplicative*⟩
  | ⟨*expression*⟩ ‘+’ ⟨*multiplicative*⟩
  | ⟨*expression*⟩ ‘−’ ⟨*multiplicative*⟩

⟨*multiplicative*⟩ ::= ⟨*unary*⟩
  | ⟨*multiplicative*⟩ ‘`*`’ ⟨*unary*⟩
  | ⟨*multiplicative*⟩ ‘`/`’ ⟨*unary*⟩

⟨*unary*⟩ ::= ⟨*atomic*⟩
  | ‘+’ ⟨*unary*⟩
  | ‘−’ ⟨*unary*⟩

⟨*atomic*⟩ ::= ⟨*integer*⟩
  | ⟨*identifier*⟩
  | ‘`(`’ ⟨*expression*⟩ ‘`)`’

Figure 1: Syntax of the language, presented in Backus-Naur form. The symbols delimited by ⟨⟩ are non-terminals while the symbols delimited by ‘’ are terminals. $\langle program \rangle$ is the starting non-terminal.

$$\frac{\Gamma \vdash \texttt{e}_1 \Rightarrow v_1 \qquad \Gamma \vdash \texttt{e}_2 \Rightarrow v_2}{\Gamma \vdash \texttt{e}_1 + \texttt{e}_2 \Rightarrow v_1 + v_2}\ \textsc{Add} \qquad \frac{}{\Gamma \vdash \texttt{i} \Rightarrow i}\ \textsc{Int} \qquad \frac{\texttt{x} \in \Gamma}{\Gamma \vdash \texttt{x} \Rightarrow \Gamma(x)}\ \textsc{Lookup}$$

$$\frac{\Gamma \vdash \texttt{e}_1 \Rightarrow v_1 \qquad \Gamma \vdash \texttt{e}_2 \Rightarrow v_2 \qquad v_1 = v_2}{\Gamma \vdash \texttt{e}_1 = \texttt{e}_2 \Rightarrow \texttt{T}}\ \textsc{Test1} \qquad \frac{\Gamma \vdash \texttt{e}_1 \Rightarrow v_1 \qquad \Gamma \vdash \texttt{e}_2 \Rightarrow v_2 \qquad v_1 \neq v_2}{\Gamma \vdash \texttt{e}_1 = \texttt{e}_2 \Rightarrow \texttt{F}}\ \textsc{Test2}$$

$$\frac{\Gamma, s \overset{*}{\rightarrow} \Gamma'}{\Gamma, s \rightarrow \Gamma'}\ \textsc{Closure} \qquad \frac{\Gamma \vdash \texttt{e} \Rightarrow v}{\Gamma, \texttt{x} = \texttt{e} \rightarrow (\texttt{x}, v) :: \Gamma}\ \textsc{Intro}$$

$$\frac{\Gamma \vdash \texttt{t} \Rightarrow \texttt{T} \qquad \Gamma, \texttt{s} \rightarrow \Gamma' \qquad \Gamma', \texttt{while (t) s} \rightarrow \Gamma''}{\Gamma, \texttt{while (t) s} \rightarrow \Gamma''}\ \textsc{While1} \qquad \frac{\Gamma \vdash \texttt{t} \Rightarrow \texttt{F}}{\Gamma, \texttt{while (t) s} \rightarrow \Gamma}\ \textsc{While2}$$

$$\frac{\emptyset, \texttt{s} \rightarrow \Gamma \qquad \Gamma \vdash \texttt{e} \Rightarrow v}{[\![\texttt{s; return e}]\!] = v}\ \textsc{Program}$$

Figure 2: Semantics of the language, presented as inference rules. The semantics is defined as four predicates formalizing the evaluation of expressions ($\Gamma \vdash e \Rightarrow v$), single statement step ($\Gamma, s \Rightarrow \Gamma$), the reflexive and transitive closure of statements ($\Gamma, s \overset{*}{\rightarrow} \Gamma$), and the evaluation of the program overall ($[\![p]\!] = v$).

Method	Accuracy
Unigram features + logistic regression	.5
Unigram features + multilayer perceptron	.5
Bigram features + logistic regression	.5
Bigram features + multilayer perceptron	.5
Embedding features + logistic regression	.5
Embedding features + multilayer perceptron	.5
Hidden Markov model	.57
Recurrent neural network, classification	.62
Long short-term memory network, classification	.98
Long short-term memory network + set, classification	.993
Long short-term memory network + set, transduction	.997

Table 1: Accuracy of different learning algorithms on the static analysis task. LR stands for logistic regression, MLP stands for multilayer perceptron, HMM stands for hidden Markov Model, RNN stands for recurrent neural network, LSTM stands for Long Short-Term Memory.

four predicates. The predicate $\Gamma \vdash e \Rightarrow v$ denotes the value v resulting from evaluating e in the environment Γ. The environment is simply a list of bindings from variables to their values. We present four rules that define this predicate, ADD, INT, LOOKUP, TEST1, and TEST2. The most important is the LOOKUP rule which states that the value of a variable is the value associated to it in the environment. Note that this is only well-defined if the variable actually is in the environment, otherwise the semantics is undefined. The goal of our static analyzer is to ensure this can never happen.

The predicate $\Gamma, s \Rightarrow \Gamma$ denotes the execution of a statement that transforms the environment by adding variable bindings to it. For example, the INTRO rule shows that a variable assignment adds a variable binding to the environment, The CLOSURE rule states that a possible transition is the reflexive and transitive execution of a single statement $\Gamma, s \overset{*}{\to} \Gamma$. The rules WHILE1 and WHILE2 formalize the execution of a while loop. Finally, the predicate $[\![p]\!] = v$ denotes the evaluation of a complete program into a resulting value.

2.2 The Task

Now that we have presented the language, we can state more precisely the goal of the static analysis. A program such as “v1 = 4; return v1 + v2;”, while syntactically valid, is not well-defined since variable v2 has not been defined. A static analyzer is a function that takes such a program as an input and returns a Boolean value.

$$\texttt{analyze} : \texttt{token sequence} \longmapsto \texttt{Boolean}$$

Function analyze should return `true` only if every variable is defined before it is used. We chose the input to be the sequence of tokens of the program rather than the raw characters for simplicity. It is easy to define such a function directly, but our goal is to see whether we can *learn* it from examples. Note that unlike previous work combining static analysis and machine learning, we are not trying to improve a static analyzer using machine learning, but rather learning the static analyzer completely from data.

3 Learning a Static Analyzer

To learn the static analyzer, we compile a balanced set of examples in which programs are labeled with a single Boolean value indicating whether the program should be accepted or not.

The dataset contains 200,000 examples, half of which are valid programs and half of which are invalid programs. The invalid programs are of two forms. Half of them contain variables that have not been defined at all, the other half contains programs where the order of statements has been swapped and a variable use appears before its definition. Note that this swapping of statements results in documents that have the exact same bag-of-words, but different labels. Of the 200,000 examples, we use 150,000 as the training set and 50,000 as the test set while making sure to respect a perfect balance between valid and invalid programs.

To create this dataset, we have built our own compiler and example generator for our language. The example generator only produces syntactically valid programs. The programs are generated using a variety of random decisions: for example, when trying to generate a statement, we must decide with what probability we want to choose a variable assignment versus a while loop or another type of statement. We vary the probability to try to avoid producing a dataset with a spurious signal, but this is a very delicate issue. We also try our classifiers on hand-written programs.

We apply several different machine learning methods, including LSTM to the problem (described below) and present results in Table 1.

N-grams and classification We attempt to learn the static analyzer using a classic approach of feature engineering followed by classification. We try both unigram and bigrams features and classify the examples using either a linear logistic regression or a non-linear multilayer perceptron. We expect this approach to fail since n-gram features fail to capture statement ordering, and this serves as a test to make sure our dataset does not contain any spurious signal. Indeed, these methods do not perform better than random.

Sequence embedding and classification We also attempt to use an LSTM for our feature engineering. In this case, we first train an LSTM as language model. Then, for classification, we first execute our language model on the example program and use the last hidden state as an embedding. This embedding is used as an input to both a logistic regression and a multilayer perceptron. This approach fails as well and does not perform better than random. It is important to note that we might also consider using an RNN encoder-decoder to produce the embedding but we leave this for future work.

Sequence classication We tried three model-based approaches to sequence classification. First, we tried to use an HMM trained using the Baum-Welch algorithm. Second, we tried to train a vanilla RNN with a cross-entropy loss using stochastic gradient descent (SGD). Third, we tried to train an LSTM with cross-entropy loss and SGD. More precisely, we use the variant of SGD known as RMSProp. In both cases we used the Keras framework.

These sequence classification approaches perform better than the other approaches. However, the HMM and the RNN still perform poorly. Interestingly, the LSTM can achieve an accuracy of 98.3%. The training of the LSTM is very robust, we did not need to do any complicated parameter search to obtain these results. The false negative rate (*i.e.* the program is correct but predicted as faulty) is 1.0% and the false positive rate (*i.e.* the program is faulty but classified as correct) is 2.5%.

Using differentiable data structures The key problem in detecting uninitialized variables is to remember which variables have been defined up to some program point. A solution is to employ a *set* data structure: if we encounter a variable definition, we add the variable to the set; if we encounter a variable use, we test whether that variable is in the set of defined variables. With this in mind, we design a differentiable set data structure to augment an RNN to see if the resulting network can learn (from the training data alone) a policy of how to use the set.

The set is represented by a vector f and intended to be used as a bitmap by the network. The intent is that each possible value correspond to a bit in the vector and is set to 1 if the element is in the set and 0 otherwise. An action a on the set can be either adding an element to the set, or testing if some value is in the set. Values v are indices into the set representation.

Controller update: $$h_t = \mathsf{RNN}([x_t, f_{t-1}], h_t) \tag{1}$$
Action: $$a_t = \sigma(W_a x_t + b_a) \tag{2}$$
Input representation: $$v_t = \mathsf{softmax}(W_{i2}\,\mathsf{tanh}(W_{i1} x_t + b_{i1}) + b_{i2}) \tag{3}$$
Set update: $$f_t = \mathsf{max}\{f_{t-1}, a_t * v_t\} \tag{4}$$
Set test $$p_t = \langle f_t, v_t \rangle \tag{5}$$
Decision: $$y_t = \sigma(W_y h_t + U_y p_t + b_y) \tag{6}$$

RNN can be a simple Ellman network or LSTM. The architecture is shown in Figure 4.

```
v14 = (v14 − 23) ;
# 1 1 1 0  1 0 0 0
v14 = (14 ∗ 93) ;
# 0 0 0 0 0 0 0 0
return  v14 ;
# 0       0   0
```

(a) Classification task. Once an error is detected, the rest of the outputs is meaningless.

```
v14 = (v14 − 23) ;
# 1 1 1 0  1 1 1 1
v14 = (14 ∗ 93) ;
# 1 1 1 1 1 1 1 1
return  v14 ;
# 1       1   1
```

(b) Transduction task. The network gets every output right.

```
v14 = (14 ∗ 93) ;
# 1 1 1 1 1 1 1 1
v14 = (v14 − 23) ;
# 1 1 1 1  1 1 1 1
return  v14 ;
# 1       1   1
```

(c) An example for which both classification and transduction work.

Figure 3: A look inside the prediction of different networks that use an LSTM and a differentiable set data structure. The commented line shows the label attached to each variable by the network. A 1 means a variable is properly used while a 0 means a variable was not initialized.

Unfortunately, training differentiable data structures is sometimes difficult, requiring extensive hyper-parameter tuning and cross validation to find a good weight initialization. Further, LSTMs are often able to learn the training data single-handedly causing the network to learn a policy that ignores the data structure. To circumvent these problem, we annotate the training data with additional intermediate signals: specifically, we annotate each token with a binary label that is true if and only if the token is a variable use that has not been initialized. Note that the additional labels results in a per-token classification problem, but we convert the network back into a program classifier by employing min-pooling over the per-token soft max outputs. We experiment with both per-program (sequence classification) and per-token (sequence transduction) classifiers as described next.

Sequence classification: As in previous experiments, we train using the program as the input sequence and a single Boolean label to indicate whether the program is valid or not. For the network with differentiable set to produce one output we apply min-pooling across all the decisions. This method improves over an LSTM and achieves an accuracy of 99.3%. Note that, we did not need to do any complicated parameter search to obtain these results. The false negative rate (*i.e.* the program is faulty but classified as correct) is 0.8% and the false positive rate (*i.e.* the program is correct but predicted as faulty) is 0.6%.

To understand the behavior of the network, we remove the last minpooling layer, and look at the decision made by the network for each input token. This reveals an interesting pattern: the network correctly identifies the first error location and subsequently emit incorrect outputs. Thus, it is comparable to conventional (non-ML) static analysis algorithms that give up after the first error. For example, in the example in figure 3a the first variable use is correctly identified as invalid but the rest of the output is incorrect. information.

Sequence transduction: Finally, we run an experiment at token level granularity. In this case, the network produce not just a single output but as many outputs as inputs (many-to-many architecture), we refer to this approach as sequence transduction to distinguish from the recurrent networks that produce a single label (many-to-one architecture). The training data also contains the label for each token in the program. This can achieve an accuracy of 99.7%. The training of the transduction task is very robust, we did not need to do any complicated parameter search to obtain these results. The false negative rate is 0.4% and the false positive rate is 0.2%.

Given the token level data, it seems that the network has inducted a use of the set data structure that correspond to what an traditional algorithm would do. Aside from using the set to keep track of defined variables, it correctly handles the tricky case of a statement such as `v1 = v1 + 3;` by making sure that the variable `v1` is introduced in the set only after the statement is finished. For example, in the example presented in figure 3b, the declaration of the variable `v14` utilizes the value of the still undeclared variable `v14` and the network correctly identifies it.

Unfortunately, and interestingly, the accuracy is not perfect. Even though it looks like the correct use of the set has been learned, there are a few rare cases where the network makes simple mistakes. For example, some of the errors happen on some the simplest and shortest programs where the network fails to insert the declared variable into the set.

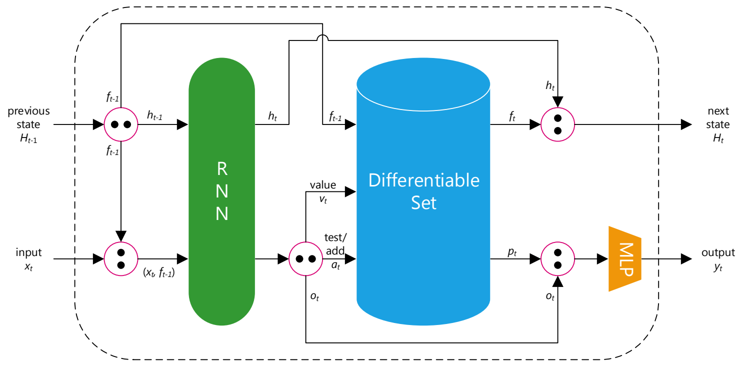

Figure 4: Overview of a network utilizing the differentiable set data structure for the task of static analysis. It consists of a neural controller and a fixed size filter.

Conclusion In conclusion, an out-of-the-box LSTM achieves a promising accuracy on this task, and an LSTM equipped with a differentiable set data-structure has an almost perfect accuracy. Interestingly, none of the other approaches including HMM or RNN, could deliver satisfactory results.

4 Reporting Useful Error Messages

While the above experiment demonstrates that it is possible to learn an accurate static analyzer; practically, such an analyzer is somewhat useless unless we can also help the programmer locate the potential errors. That is, imagine if a tool reported that there is a potential buffer overflow in your code base without any indication of where the problem is: it would not be of much use.

Therefore we train a second LSTM as a language model over true instances of our programming language. That is, we train the LSTM to predict the next character in the sequence, and for every character in the sequence, the model provides the probability of observing this specific character. The idea is that we want to look at all the variable-use in the program and if the probability of this variable use is below a certain threshold, then we report the use as a potential source of error.

We present several such examples in Figure 5. We color a variable-use in blue if its probability is above the threshold and in purple if it is below the threshold and therefore potentially the source of the error.

As we can see from the examples, the method works well. The first four examples show simple cases with only two variables. Note that from the perspective of a bag-of-words classifier, these two programs are identical. Yet the LSTM language model, which takes into account the "word" order is able to model them differently. Examples 5-11 are more complicated in that the variables are used or defined several times. In Example 9, the language model accurately reports the first use of v2 as incorrect and the second use of v2 as correct. This is a somewhat interesting example as the incorrect use of v2 is in the definition of v2 itself. In example 10, we can see that the language model can handle multiple incorrect variable uses; this success crucially depends on the ability of the language model to recover from the error and still accurately model the remainder of the program. Finally, examples 12 and 13 demonstrate robustness. Despite the fact that these two examples are syntactically incorrect, the language model correctly reports the semantic errors. The resilience of the learned tools to small errors is part of what makes them so promising for program analysis.

5 Related Work

There is a growing body of work in employing machine learning to improve programming language tools. In such works, machine learning is used to complement the traditional static analysis methods; further, they rely on extensive feature engineering. In Brun & Ernst (2004), dynamic analysis is used to extract features that are used to detect latent code errors. In Kolter & Maloof (2006), n-

```
1. v1 = 37; v2 = (v1 + 20);
2. v1 = 37; v1 = (v2 + 20);
3. v2 = 37; v1 = (v2 + 20);
4. v2 = 37; v2 = (v2 + 20);
5. v2 = 37; v2 = (v2 + 20); v3 = (v2 + 40);
6. v2 = 37; v2 = (v2 + 20); v2 = (v3 + 40);
7. v2 = 37; v2 = (v2 + 20); v3 = (v1 + v2);
8. v2 = 37; v1 = (v2 + 20); v3 = (v1 + v2);
9. v1 = 37; v2 = (v2 + 20); v3 = (v1 + v2);
10. v1 = 37; v3 = (v2 + 20); v5 = (v3 + v4);
11. v1 = 37; v3 = (v2 + 20); v5 = (v3 + v2);
12. v1 = 37 v2 = (v1 + 20);
13. v1 = 37 v1 = (v2 + 20);
```

Figure 5: Example of programs annotated with variable usage. The use colored in blue are considered to have been properly defined while the use in purple are considered to be faulty. This tool is run when the classifier detects a program error to help the programmer understand what the problem is.

gram features are used to detect viruses in binary code. In Yamaguchi et al. (2012), parts of the abstract syntax tree of a function is embedded into a vector space to help detect functions similar to a known faulty one. In Tripp et al. (2014), various lexical and quantitative features about a program is used to improve an information analysis and reduce the number of false alarms reported by the tool. In Raychev et al. (2015), dependency networks are used with a conditional random field to de-obfuscate and type Javascript code. In Allamanis et al. (2015), the structure of the code is used to suggest method names. In Nguyen & Nguyen (2015), n-grams are used to improve code completion tools. In Gvero & Kuncak (2015), program syntax is used to learn to tool that can generate Java expressions from free-form queries. In Long & Rinard (2016), a feature extraction algorithm is designed to improve automatic patch generation.

6 CONCLUSION

We have shown that it is possible to learn a static analyzer from data. Even though the problem we address is particularly simple and on a toy language, it is interesting to note that in our experiments, only LSTM networks provided a reasonable enough solution. We have also shown that it is possible to make the static analyzer useful by using a language model to help the programmer understand where to look in the program to find the error.

Of course, this experiment is very far from any practical tool. First, dealing with more complicated programs involving memory, functions, and modularity should be vastly more complex. Also, our solution is very brittle. For example, in our language, the space of variable names is very restricted, it might be much more difficult to deal with normal variable names where a specific variable name could not appear at all in the training dataset.

Finally, a fundamental issue are false positives, that is, programs that are wrongly classified as being without error. This is a serious problem that may make such a tool risky to use. However, note that there are useful programming language tools that indeed generate false positive. For instance, a tool that report buffer overflows might not catch every error, but it is still useful if it catches some. Another possibility is to consider approaches were a result is verified by an external tools. For example, in the field of certified compilation, Tristan & Leroy (2008) have shown that it can be acceptable to use an untrusted, potentially bogus, program transformation as long as each use can be formally checked. Also, as exemplified by Gulwani & Necula (2003; 2004; 2005) some static analysis algorithms do trade a small amount of unsoundness for much faster computation, which can be necessary when applying programming tools to very large code base.

REFERENCES

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 38–49, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786849. URL `http://doi.acm.org/10.1145/2786805.2786849`.

Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pp. 480–490, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0. URL `http://dl.acm.org/citation.cfm?id=998675.999452`.

Andrew M. Dai and Quoc V. Le. Semi-supervised sequence learning. *CoRR*, abs/1511.01432, 2015. URL `http://arxiv.org/abs/1511.01432`.

Sumit Gulwani and George C. Necula. Discovering affine equalities using random interpretation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pp. 74–84, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5. doi: 10.1145/604131.604138. URL `http://doi.acm.org/10.1145/604131.604138`.

Sumit Gulwani and George C. Necula. Global value numbering using random interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pp. 342–352, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X. doi: 10.1145/964001.964030. URL `http://doi.acm.org/10.1145/964001.964030`.

Sumit Gulwani and George C. Necula. Precise interprocedural analysis using random interpretation. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pp. 324–337, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040332. URL `http://doi.acm.org/10.1145/1040305.1040332`.

Tihomir Gvero and Viktor Kuncak. Synthesizing java expressions from free-form queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pp. 416–432, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814295. URL `http://doi.acm.org/10.1145/2814270.2814295`.

J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006. ISSN 1532-4435. URL `http://dl.acm.org/citation.cfm?id=1248547.1248646`.

Zachary Chase Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015. URL `http://arxiv.org/abs/1506.00019`.

Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pp. 298–312, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837617. URL `http://doi.acm.org/10.1145/2837614.2837617`.

Anh Tuan Nguyen and Tien N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pp. 858–868, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL `http://dl.acm.org/citation.cfm?id=2818754.2818858`.

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pp. 111–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677009. URL `http://doi.acm.org/10.1145/2676726.2677009`.

Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pp. 762–774, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660339. URL `http://doi.acm.org/10.1145/2660267.2660339`.

Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pp. 17–27, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328444. URL `http://doi.acm.org/10.1145/1328438.1328444`.

Zhengzheng Xing, Jian Pei, and Eamonn Keogh. A brief survey on sequence classification. *SIGKDD Explor. Newsl.*, 12(1):40–48, November 2010. ISSN 1931-0145. doi: 10.1145/1882471.1882478. URL `http://doi.acm.org/10.1145/1882471.1882478`.

Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pp. 359–368, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4. doi: 10.1145/2420950.2421003. URL `http://doi.acm.org/10.1145/2420950.2421003`.