Deep Learning to Find Bugs

Michael Pradel and Koushik Sen

Technical Report TUD-CS-2017-0295

TU Darmstadt, Department of Computer Science

November, 2017

Deep Learning to Find Bugs

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany

Koushik Sen
EECS Department
University of California, Berkeley, USA

Abstract

Automated bug detection, e.g., through pattern-based static analysis, is an increasingly popular technique to find programming errors and other code quality issues. Traditionally, bug detectors are program analyses that are manually written and carefully tuned by an analysis expert. Unfortunately, the huge amount of possible bug patterns makes it difficult to cover more than a small fraction of all bugs. This paper presents a new approach toward creating bug detectors. The basic idea is to replace manually writing a program analysis with training a machine learning model that distinguishes buggy from non-buggy code. To address the challenge that effective learning requires both positive and negative training examples, we use simple code transformations that create likely incorrect code from existing code examples. We present a general framework, called DeepBugs, that extracts positive training examples from a code corpus, leverages simple program transformations to create negative training examples, trains a model to distinguish these two, and then uses the trained model for identifying programming mistakes in previously unseen code. As a proof of concept, we create four bug detectors for JavaScript that find a diverse set of programming mistakes, e.g., accidentally swapped function arguments, incorrect assignments, and incorrect binary operations. To find bugs, the trained models use information that is usually discarded by program analyses, such as identifier names of variables and functions. Applying the approach to a corpus of 150,000 JavaScript files shows that learned bug detectors have a high accuracy, are very efficient, and reveal 132 programming mistakes in real-world code.

1 Introduction

Automated bug detection techniques are widely used by developers and have received significant attention by researchers. One of the most popular techniques are lightweight, lint-style, static analyses that search for instances of known bug patterns. Typically, such analyses are created as part of a framework that supports an extensible set of bug patterns, or rules, such as Google Error Prone [1] , FindBugs [18], or lgtm [1]. Each of these frameworks contain at least several dozens, sometimes even hundreds, of *bug detectors*, i.e., individual analyses targeted at a specific bug pattern. The term "bug" here refers to a wide spectrum of problems that developers should address, including incorrect program behavior and other code quality issues, e.g., related to readability and maintainability.

Even though various bug detectors have been created and are widely used, there still remain numerous bugs that slip through all available checks. We hypothesize that one important reason for missing so many bugs is that manually creating a bug detector is non-trivial. A particular challenge is that each bug detector must be carefully tuned and extended with heuristics to be practical. For example, a recently deployed bug detector that is now part of the Google Error Prone framework comes with various heuristics to increase the number of detected bugs and to decrease the number of false positives [37].

We propose to address the challenge of creating bug detectors through machine learning. Even though machine learning has helped address a variety of software development tasks [8, 13, 14, 36] and even though buggy code stands out compared to non-buggy code [33], the problem of learning-based bug detection remains an open challenge. A key reason for the current lack of learning-based bug detectors is that effective machine learning requires large amounts of training data. To train a model that effectively identifies buggy code, learning techniques require many examples of both correct and incorrect code – typically thousands or even millions of examples. However, most available code is correct, or at least it is unknown exactly which parts of it are incorrect. As a result, existing bug detectors that infer information from existing code learn only from correct examples [16] and then flag any code that is unusual as possibly incorrect, or search for inconsistencies within a single program [11]. Unfortunately, to be practical, these approaches rely on built-in domain knowledge and various carefully tuned heuristics.

In this paper, we address the problem of automatically detecting buggy code with a technique that learns from both correct and buggy code. We present a general framework, called DeepBugs, that extracts positive training examples from a code corpus, applies a simple transformation to also create large amounts of negative training examples, trains a model to distinguish these two, and to finally uses the trained model for identifying mistakes in previously unseen code. The key idea that enables DeepBugs to learn an effective model is to synthesize negative training examples by seeding bugs into existing, presumably correct code. To create negative examples for a particular bug pattern, all that is required is a simple code transformation that creates instances of the bug pattern. For example, to detect bugs caused by accidentally swapping arguments passed to a function, the

[1] https://lgtm.com/

transformation swaps arguments in existing code, which is likely to yield a bug.

To instantiate the framework, we focus on name-based bug detectors, i.e., a class of bug detectors that exploit implicit semantic information provided through identifier names. For example, suppose a function call `setSize(height, width)` that is incorrect because the arguments are passed in the wrong order. A name-based bug detector would pinpoint this bug by comparing the names of the arguments `height` and `width` to the names of the formal parameters and of arguments passed at other call sites. Such information has been used in manually created analyses that revealed various name-related programming mistakes [23, 31, 37].

However, due to the fuzzy nature of programmer-chosen identifier names, manually creating name-based bug detectors is challenging and heavily relies on well-tuned heuristics. In particular, existing techniques consider lexical similarities between names, e.g., between `height` and `myHeight`, but miss semantic similarities, e.g., between `height` and `y_dim`. To enable DeepBugs to reason about such semantic similarities, it automatically learns embeddings, i.e., dense vector representations, of identifier names. The embedding assigns similar vectors to semantically similar names, such as `height` and `y_dim`, allowing DeepBugs to detect otherwise missed bugs and to avoid otherwise reported spurious warnings.

As a proof of concept, we create four bug detectors for JavaScript that find a diverse set of programming mistakes, e.g., accidentally swapped function arguments, incorrect assignments, and incorrect binary operations. We evaluate DeepBugs and its four instances by learning from a corpus of 100,000 JavaScript files and by searching mistakes in another 50,000 JavaScript files. In total, the corpus amounts to 68 million lines of code. We find that the learned bug detectors have an accuracy between 84.23% and 94.53%, i.e., they are very effective at distinguishing correct from incorrect code. Manually inspecting a subset of the warnings reported by the bug detectors, we found 132 real-world bugs and code quality problems. Even though we do not perform any manual tuning or filtering of warnings, the bug detectors have a reasonable precision (roughly half of all inspected warnings point to actual problems) that is comparable to manually created bug detectors.

In summary, this paper contributes the following:

- A general framework for learning bug detectors by training a model with both positive and negative examples. (Section 2)
- The observation that simple code transformations applied to existing code yield negative training examples that enable the training of an effective bug detector.
- A novel approach to derive embeddings of identifiers and literals, i.e., distributed vector-representations of code elements often ignored by program analyses. The embeddings enable name-based bug detectors to reason about semantic similarities of programmer-chosen identifier names. (Section 3)
- Four name-based bug detectors created with the framework that target a diverse set of programming errors. In contrast to existing name-based bug detectors, they do not rely on manually tuned heuristics. (Section 4)
- Empirical evidence that learned bug detectors have high accuracy, are efficient, and reveal various programming mistakes in real-world code. (Section 6)

Our implementation is available as open-source:
https://github.com/michaelpradel/DeepBugs

2 A Framework for Learning to Find Bugs

This section presents the DeepBugs framework for automatically creating bug detectors via machine learning. The basic idea is to train a classifier to distinguish between code that is an instance of a specific bug pattern and code that does not suffer from this bug pattern. By bug pattern, we informally mean a class of recurring programming errors that are similar because they violate the same rule. For example, accidentally swapping the arguments passed to a function, calling the wrong API method, or using the wrong binary operator are bug patterns. Manually written bug checkers, such as FindBugs or Error Prone, are based on bug patterns, each of which corresponds to a separately implemented analysis.

2.1 Overview

Given a corpus of code, creating and using a bug detector based on DeepBugs consists of several steps. Figure 1 illustrates the process with a simple example.

1. *Extract and generate training data from the corpus.* This step statically extracts positive code examples from the given corpus and generates negative code examples by applying a simple code transformation. Because we assume that most code in the corpus is correct, each extracted code example is likely to not suffer from the particular bug pattern. To also create negative training examples, DeepBugs applies simple code transformations that are likely to introduce a bug. (Step 1 in Figure 1.)
2. *Represent code as vectors.* This step translates each code example into a vector. To preserve semantic information conveyed by identifier names, we use a learned embedding of identifiers to compute the vector representation. (Step 2 in Figure 1.)
3. *Train a model to distinguish correct and incorrect examples.* Given two sets of code examples that contain positive and negative examples, respectively, this step trains a machine learning model to distinguish between the two. (Step 3 in Figure 1.)
4. *Predict bugs in previously unseen code.* This step applies the classifier obtained in the previous step to predict whether a previously unseen piece of code suffers from the bug

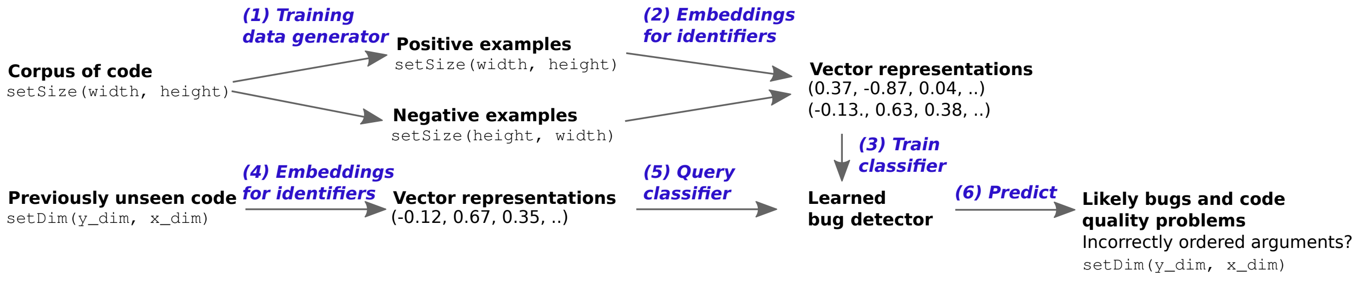


Figure 1. Overview of our approach.

pattern. If the learned model classifies the code to be likely incorrect, the approach reports a warning to the developer. (Steps 4 to 6 in Figure 1.)

2.2 Generating Training Data

An important prerequisite for any learning-based approach is a sufficiently large amount of training data. In this work, we formulate the problem of bug detection as a binary classification task addressed via supervised learning. To effectively address this task, our approach relies on training data for both classes, i.e., both examples of correct and incorrect code. As observed by others [9, 28, 35], the huge amounts of existing code provide ample of examples of likely correct code. In contrast, it is non-trivial to obtain many examples of code that suffers from a particular bug pattern. One possible approach is to manually or semi-automatically search code repositories and bug trackers for examples of bugs that match a given bug pattern. However, scaling this approach to thousands or even millions of examples, as required for advanced machine learning, is extremely difficult.

Instead of relying on manual effort for creating training data, this work generates training data fully automatically from a given corpus of code. The key idea is to apply a simple code transformation τ that transforms likely correct code extracted from the corpus into likely incorrect code. Section 4 presents implementations of τ that apply simple AST-based code transformations.

Definition 2.1 (Training data generator)**.** Let $C \subseteq L$ be a set of code in a programming language L. Given a piece of code $c \in C$, a training data generator $G : c \rightarrow (C_{pos}, C_{neg})$ creates two sets of code $C_{pos} \subseteq C$ and $C_{neg} \subseteq L$, which contain positive and negative training examples, respectively. The negative examples are created by applying transformation $\tau : C \rightarrow C$ to each positive example:
$C_{neg} = \{c_{neg} \mid c_{neg} = \tau(c_{pos}) \; \forall c_{pos} \in C_{pos}\}$

There are various ways to implement a training data generator. For example, suppose the bugs of interest are accidentally swapped arguments of function calls. A training data generator for this bug pattern gathers positive examples by extracting all function calls that have at least two arguments and negative examples by permuting the order of these arguments of the function calls. Under the assumption that the given code is mostly correct, the unmodified calls are likely correct, whereas changing the order of arguments is likely to provide an incorrect call.

2.3 Training and Querying a Bug Detector

Given positive and negative training examples, our approach learns a bug detector that distinguishes between them. Machine learning-based reasoning about programs relies on a representation of code that is suitable for learning.

Definition 2.2 (Code representation)**.** Given a piece of code $c \in C$, its code representation $v \in \mathbb{R}^n$ is a n-dimensional real-valued vector.

As a valid alternative to a vector-based code representation, one could feed a graph representation of code into a suitable machine learning model, such as (gated) graph neural networks [21, 38] or recursive neural networks [39]. To obtain the vector-based code representation, we use a graph representation of code (ASTs, details in Section 3).

Based on the vector representation of code, a bug detector is a model that distinguishes between vectors that correspond to correct and incorrect code examples, respectively.

Definition 2.3 (Bug detector)**.** A bug detector D is a binary classifier $D : C \rightarrow [0, 1]$ that predicts the probability that a piece of code $c \in C$ is an instance of a particular bug pattern.

Training a bug detector consists of two steps. At first, DeepBugs computes for each positive example $c_{pos} \in C_{pos}$ its vector representation $v_{pos} \in \mathbb{R}^n$, which yields a set V_{pos} of vectors. Likewise, the approach computes the set V_{neg} from the negative examples $c_{neg} \in C_{neg}$. Then, we train the bug detector D in a supervised manner by providing two kinds of input-output pairs: $(v_{pos}, 0)$ and $(v_{neg}, 1)$. That is, the model is trained to predict that positive code examples are correct and that negative code examples are incorrect. In principle, the bug detector can be implemented by any classification technique. We use a feedforward neural network with a single hidden layer and a single-element output layer that represents the probability computed by D.

Given a sufficiently large set of training data, the bug detector will generalize beyond the training examples and one can query it with previously unseen code. To this end, DeepBugs extracts pieces of code C_{new} in the same way as extracting the positive training data. For example, for a bug detector that identifies swapped function arguments, the approach extracts all function calls including their unmodified arguments. Next, DeepBugs computes the vector representation of each example $c_{new} \in C_{new}$, which yields a set V_{new}. Finally, we query the trained bug detector D with every $v_{new} \in V_{new}$ and obtain for each piece of code a prediction whether it is incorrect. To report warnings about bugs the a developer, DeepBugs ranks warnings by the predicted probability in descending order. In addition, one can control the overall number of warnings by omitting all warnings with a probability below a configurable threshold.

3 Vector Representations for Identifiers and Literals

The general framework from Section 2 is applicable to various kinds of bug detectors. In the remainder of this paper, we focus on name-based bug detectors, which reason about the semantics of code based on the implicit information provided by developer-chosen identifier names. Name-based bug detection has been shown to be effective in detecting otherwise missed bugs [23, 31] and has recently been adopted by a major software company [37]. For example, the name-based bug detector in [37] detects accidentally swapped function arguments by comparing the identifier names of actual arguments at call sites to the formal parameter names at the corresponding function definition. Besides showing the power of name-based bug detection, prior work also shows that manually developing an effective analysis requires various manually developed and fine-tuned heuristics to reduce false positives and to increase the number of detected bugs.

The main challenge for reasoning about identifiers is that understanding natural language information is non-trivial for computers. Our goal is to distinguish semantically similar identifiers from dissimilar ones. In addition to identifiers, we also consider literals in code, such as `true` and `23`, because they also convey relevant semantic information that can help to detect bugs. To simplify the presentation, we say "identifier" to denote both identifiers and literals. For example, `seq` and `list` are similar because both are likely to refer to ordered data structures. Likewise, `true` and `1` are similar (in JavaScript, at least) because both evaluate to `true` when being used in a conditional. In contrast, `height` and `width` are semantically dissimilar because they refer to opposing concepts. As illustrated by these examples, semantic similarity does not always correspond to lexical similarity, as considered by prior work [23, 31, 37], and may even cross type boundaries. To enable a machine learning-based bug detector to reason about tokens, we require a representation of identifiers that preserves semantic similarities.

DeepBugs reasons about identifiers by automatically learning a vector representation for each identifier based on a corpus of code. The vector representation, also called *embedding*, assigns to each identifier a real-valued vector in a k-dimensional space. Let I be the set of all identifiers in a code corpus. An embedding is a function $E : I \rightarrow \mathbb{R}^k$. A naïve representation is a local, or one-hot, encoding, where $k = |I|$ and where each vector returned by E contains only zeros except for a single element that is set to one and that represents the specific token. Such a local representation fails to provide two important properties. First, to enable efficient learning, we require an embedding that stores many identifiers in relatively short vectors. Second, to enable DeepBugs to generalize across non-identical but semantically similar identifiers, we require an embedding that assigns a similar vector to semantically similar identifiers.

Instead of a local embedding, we use a distributed embedding, where the information about an identifier in I is distributed across all elements of the vector returned by E. Our distributed embedding is inspired by word embeddings for natural languages, specifically by Word2Vec [25]. The basic idea of Word2Vec is that the meaning of a word can be derived from the various contexts in which this word is used. In natural languages, the context of an occurrence of a word w in a sequence of words is the window of words preceding and succeeding w. An obvious way to adapt this idea to source code would be to view code as a sequence of tokens and to define the context of the occurrence of an identifier as its immediate preceding and succeeding tokens. However, we observe that the surrounding tokens often contain purely syntactic artifacts that are irrelevant to the semantic meaning of an identifier. Moreover, viewing code as a sequence of tokens discards richer, readily available representations of code.

3.1 AST-based Context

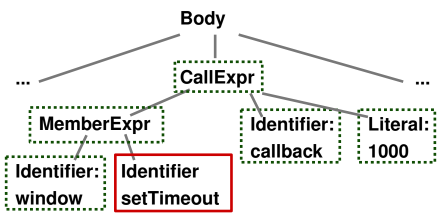
We define the context of an identifier based on the AST of the code. The basic idea is to consider the nodes surrounding an identifier node n as its context. The idea is based on the observation that the surrounding nodes often contain useful context information, e.g., about the syntactic structure that n is part of and about other identifiers that n relates to. Formally, we define the context as follows.

Definition 3.1 (AST context)**.** Given a node n in an abstract syntax tree, let p and g be the parent and grand-parent of n, respectively, and let S, U, C, N be the sets of siblings, uncles, cousins, and nephews of n, respectively. Furthermore, let p_{pos} and g_{pos} denote the relative position of n in the sequence of children of n's parent and grand-parent, respectively. The AST context of n is a tuple $(p, p_{pos}, g, g_{pos}, S, U, C, N)$.

In essence, the AST context contains all nodes surrounding the node n, along with information about the relative positioning of n. We omit the children of n in the definition

```
...
window.setTimeout(callback, 1000);
...
```

(a) JavaScript code.



(b) Abstract syntax tree (slightly simplified). Nodes with dotted boxes are in the context for the node with the red, solid border.

Part of context	Value(s)
Parent p	MemberExpr
Position p_{pos} in parent	2
Grand-parent g	CallExpr
Position g_{pos} in grand-parent	1
Siblings S	{ ID:window }
Uncles U	{ ID:callback, LIT:1000 }
Cousins C	{}
Nephews N	{}

(c) AST context for `setTimeout`.

0..1..0|2|0..1..0|1|0..1..0|0..1..1..0|0..0|0..0

(d) AST context vector.

Figure 2. AST context for identifier `setTimeout`.

because identifiers are leaf nodes in common AST representations.

For example, consider the JavaScript code and its corresponding AST in Figure 2. The table in Figure 2c shows the context extracted for the identifier `setTimeout`.

Each value in the context is a string value. To compute this value from an AST node n we use a function str so that $str(n)$ is

- the identifier name if n is an identifier node,
- the string representation of the literal if n is a literal node, and
- the type of the non-terminal of the abstract grammar that n represents, for all other nodes.

3.2 Learning Embeddings for Identifiers and Literals

To learn embeddings for identifiers n from a corpus of code, our approach proceeds in three steps. At first, it extracts the context $c(n)$ of each occurrence of an identifier in the corpus. Then, it trains a neural network to predict $c(n)$ from n. Finally, the approach derives the embedding $E(n)$ from the internal representation that the neural network chooses for n. The following presents these steps in detail.

Given a corpus of code, the approach traverses the ASTs of all code in the corpus. For each identifier node n, we extract the AST context $c(n)$, as defined above. To feed the pair $(n, c(n))$ into a neural network, we compute vector representations of both n and $c(n)$. For a pair $(n, c(n))$, the vector representation of n is its one-hot encoding. For the context $c(n)$, we compute the vector as follows.

Definition 3.2 (AST context vector)**.** Given an AST context $(p, p_{pos}, g, g_{pos}, S, U, C, N)$, its vector representation is the concatenation of the following vectors:

- A one-hot encoding of p.
- A single-element vector that contains p_{pos}.
- A one-hot encoding of g.
- A single-element vector that contains g_{pos}.
- For each $X \in \{S, U, C, N\}$, a vector obtained by bit-wise adding the one-hot encodings of all elements of X.

For the example in Figure 2, Figure 2d shows the vector representation of the context. The individual subvectors are split by "|" for illustration.

The overall length of the AST context vector is $6 * |V_c| + 2$, where $|V_c|$ is the size of the vocabulary for context strings. We summarize the set of siblings, uncles, cousins, and nephews into four vectors because different nodes may have different, possibly very large numbers of siblings, etc. An alternative would be to concatenate the one-hot encodings of all siblings, etc. However, the possibly very large number of siblings, etc. would either yield very large context vectors, which is inefficient, or require us to omit siblings, etc. beyond some maximum number, which would discard potentially valuable information.

For efficiency during training, we limit the vocabulary $|V_n|$ of identifiers and literals to $10,000$ and the vocabulary $|V_c|$ of strings occurring in the context to $1,000$ by discarding the least frequent strings. To represents strings beyond V_n and V_c, we use a placeholder "unknown".

After computing pairs of vectors (v_n, v_c) for each occurrence of an identifier in the corpus, the next step is to use these pairs as training data to learn embeddings for identifiers and literals. The basic idea is to train a feedback-forward neural network to predict the context of a given identifier. The network consists of three layers:

1. An input layer of length $|V_n|$ that contains the vector v_n.
2. A hidden layer of length $e \ll |V_n|$, which forces the network to summarize the input layer to length e.
3. An output layer of length $6 * |V_c| + 2$ that contains the context vector v_c.

The layers are densely connected through a linear and sigmoid activation function, respectively. We train the network using all pairs (v_n, v_c) with the standard backpropagation algorithm, so that the network becomes more and more accurate at predicting v_c for the given v_n. We use binary cross-entropy as the loss function and the Adam optimizer [20].

Our approach is related to but differs in two ways from the skip-gram variant of the popular Word2Vec embedding for natural languages [25]. First, we exploit a structural representation of source code, an AST, to extract the context of each identifier, instead of simply using the surrounding tokens. Second, our neural network predicts the entire context vector, including all context information, instead of predicting whether a single word occurs in the context of a word, as in [25]. The rationale is to preserve the structural information encoded in the AST context vector, e.g., whether a related identifier is a sibling or an uncle. These design decisions are well-suited for program code because, in contrast to natural languages, it has a precisely defined structural representation.

Since a single identifier may occur in different context, the training data may contain pairs (v_n, v_{c1}) and (v_n, v_{c2}) with $v_{c1} \neq v_{c2}$. Such data forces the network to reconcile different occurrences of the same identifier, and the network will learn to predict a context vector that, on average, gets closest to the expected vectors.

During training, the network learns to efficiently represent identifiers in a vector of length e that summarizes all information required to predict the context of the identifier. Once the network has been trained, we query the network once again for each identifier and extract the value of the hidden layer, which then serves as the embedding for the identifier. The overall result of learning embeddings is a map $E : I \rightarrow \mathbb{R}^e$ that assigns an embedding to each identifier.

4 Example Bug Detectors

To validate our DeepBugs framework, we instantiate it with four bug patterns. They address a diverse set of programming mistakes, including accidentally swapped function arguments, incorrect assignments, and incorrect binary operations. Implementing new bug detectors is straightforward, and we envision future work to create more instances of our framework, e.g., based on bug patterns mined from version histories [10]. Each bug detector consists of two simple ingredients.

- *Training data generator.* A training data generator that traverses the code corpus and extracts positive and negative code examples for the particular bug pattern based on a code transformation. We find a simple AST-based traversal and transformation to be sufficient for all studied bug patterns.
- *Code representation.* A mapping of each code example into a vector that the machine learning model learns to classify as either benign or buggy. All bug detectors presented here build on the same embedding of identifier names and literals, allowing us to amortize the one-time effort of learning an embedding across different bug detectors.

Given these two ingredients and a corpus of training code, our framework learns a bug detector that identifies programming mistakes in previously unseen code.

Expression	Extracted name
`list`	ID:list
`23`	LIT:23
`this`	LIT:this
`i++`	ID:i
`myObject.prop`	ID:prop
`myArray[5]`	ID:myArray
`nextElement()`	ID:nextElement
`db.allNames()[3]`	ID:allNames

Table 1. Examples of identifier names and literals extracted for name-based bug detectors.

All bug detectors presented in this section share the same technique for extracting names of expressions. Given an AST node n that represents an expression, we extract $name(n)$ as follows:

- If n is an identifier, return its name.
- If n is a literal, return a string representation of its value.
- If n is a `this` expression, return "this".
- If n is an update expression that increments or decrements x, return $name(x)$.
- If n is a member expression $base.prop$ that accesses a property, return $name(prop)$.
- If n is a member expression $base[k]$ that accesses an array element, return $name(base)$.
- If n is a call expression $base.callee(..)$, return $name(callee)$.
- For any other AST node n, do not extract its name.

Table 1 gives examples of names extracted from JavaScript expressions. We use the prefixes "ID:" and "LIT:" to distinguish identifiers and literals. The extraction technique is similar to that used in manually created name-based bug detectors [23, 31, 37], but omits heuristics to make the extracted name suitable for a lexical comparison of names. For example, existing techniques remove common prefixes, such as `get` to increase the lexical similarity between, e.g., `getNames` and `names`. Instead, we identify semantic similarities of names through an embedding and by finding related names in similar code examples.

The remainder of this section presents four bug detectors build on top of DeepBugs.

4.1 Swapped Function Arguments

The first bug detector addresses accidentally swapped arguments. This kind of mistake can occur both in statically typed languages, for methods that accept multiple equally typed arguments, and in statically untyped languages, where all calls that pass two or more arguments are susceptible to the mistake.

4.1.1 Training Data Generator

To create training examples from given code, we traverse the AST of each file in the code corpus and visit each call site with two or more arguments. For each such call site, the approach extracts the following information:

- The name n_{callee} of the called function.
- The names n_{arg1} and n_{arg2} of the first and second argument.
- The name n_{base} of the base object if the call is a method call, or an empty string otherwise.
- The types t_{arg1} and t_{arg2} of the first and second argument for arguments that are literals, or empty strings otherwise.
- The names n_{param1} and n_{param2} of the formal parameters of the called function, or empty strings if unavailable.

All names are extracted using the *name* function defined above. We resolve function calls heuristically, as sound static call resolution is non-trivial in JavaScript. If either n_{callee}, n_{arg1}, or n_{arg2} are unavailable, e.g., because the *name* function cannot extract the name of a complex expression, then the approach ignores this call site.

From the extracted information, the training data generator creates for each call site a positive example $c_{pos} = (n_{base}, n_{callee}, n_{arg1}, n_{arg2}, t_{arg1}, t_{arg2}, n_{param1}, n_{param2})$ and a negative example $c_{neg} = (n_{base}, n_{callee}, n_{arg2}, n_{arg1}, t_{arg2}, t_{arg1}, n_{param1}, n_{param2})$. That is, to create the negative example, we simply swap the arguments w.r.t. the order in the original code.

4.1.2 Code representation

To enable DeepBugs to learn from the positive and negative examples, we transform c_{pos} and c_{neg} from tuples of strings into vectors. To this end, the approach represents each string in the tuple c_{pos} or c_{neg} as a vector. Each name n is represented as $E(n)$, where E is the learned embedding from Section 3, i.e., $E(n)$ is the embedding vector of the name n. To represent type names as vectors, we define a function T that maps each built-in type in JavaScript to a randomly chosen binary vector of length 5. For example, the type "string" may be represented by a vector $T(\text{string}) = [0, 1, 1, 0, 0]$, whereas the type "number" may be represented by a vector $T(\text{string}) = [1, 0, 1, 1, 0]$. Finally, based on the vector representation of each element in the tuple c_{pos} or c_{neg}, we compute the code representation for c_{pos} or c_{neg} as the concatenation the individual vectors.

4.2 Incorrect Assignments

The next bug detector checks for assignments that are incorrect because the right-hand side is not the expression that the developer intended to assign, e.g., due to a copy-and-paste mistake.

4.2.1 Training Data Generator

To extract training data, we traverse the AST of each file and extract the following information for each assignment:

- The names n_{lhs} and n_{rhs} of the left-hand side and the right-hand side of the assignment.
- The type t_{rhs} of the right-hand side if the assigned value is a literal, or an empty string otherwise.

If either n_{lhs} or n_{rhs} cannot be extracted, then we ignore the assignment.

Based on the extracted information, the approach create a positive and a negative training example for each assignment: The positive example $c_{pos} = (n_{lhs}, n_{rhs}, t_{rhs})$ keeps the assignment as it is, whereas the negative example $c_{pos} = (n_{lhs}, n'_{rhs}, t'_{rhs})$ replaces the right-hand side with an alternative, likely incorrect expression. To find this alternative right-hand side, we gather the right-hand sides of all assignments in the same file and randomly select one that differs from the original right-hand side, i.e., $n_{rhs} \neq n'_{rhs}$. The rationale for picking the alternative right-hand side from the same file is to create realistic negative examples.

4.2.2 Code representation

The vector representations of the training examples is similar to Section 4.1. Given a tuple that represents a training example, we map each string in the tuple to a vector using E (for names) or T (for types) and then concatenate the resulting vectors.

4.3 Wrong Binary Operator

The next two bug detectors address mistakes related to binary operations. At first, we consider code that accidentally uses the wrong binary operator, e.g., `i <= length` instead of `i < length`.

4.3.1 Training Data Generator

The training data generator traverses the AST of each file in the code corpus and extracts the following information:

- The names n_{left} and n_{right} of the left and right operand.
- The operator op of the binary operation.
- The types t_{left} and t_{right} of the left and right operand if they are literals, or empty strings otherwise.
- The kind of AST node k_{parent} and k_{grandP} of the parent and grand-parent nodes of the AST node that represents the binary operation.

We extract the (grand-)parent nodes to provide some context about the binary operation to DeepBugs, e.g., whether the operation is part of a conditional or an assignment. If either n_{left} or n_{right} are unavailable, then we ignore the binary operation.

From the extracted information, the approach creates a positive and a negative example: $c_{pos} = (n_{left}, n_{right}, op, t_{left}, t_{right}, k_{parent}, k_{grandP})$ and $c_{neg} = (n_{left}, n_{right}, op',$

$t_{left}, t_{right}, k_{parent}, k_{grandP}$). The operator $op' \neq op$ is a randomly selected binary operator different from the original operator. For example, given a binary expression `i <= length`, the approach may create a negative example `i < length` or `i % length`, which is likely to create incorrect code.

4.3.2 Code representation

Similar to the above bug detectors, we create a vector representation of each positive and negative example by mapping each string in the tuple to a vector and by concatenating the resulting vectors. To map a kind of AST node k to a vector, we use a map K that assigns to each kind of AST node in JavaScript a randomly chosen binary vector of length 8.

4.4 Wrong Operand in Binary Operation

The final bug detector addresses code that accidentally uses an incorrect operand in a binary operation. The intuition is that a trained machine learning model can identify whether an operand fits to another given operand and a given binary operator. For example, the bug detector identifies an operation `height - x` that was intended to be `height - y`.

4.4.1 Training Data Generator

Again, the training data generator extracts the same information as in Section 4.3, and then replaces one of the operands with a randomly selected alternative. That is, the positive example is $c_{pos} = (n_{left}, n_{right}, op, t_{left}, t_{right}, k_{parent}, k_{grandP})$, whereas the negative example is either $c_{neg} = (n'_{left}, n_{right}, op, t'_{left}, t_{right}, k_{parent}, k_{grandP})$ or $c_{neg} = (n_{left}, n'_{right}, op, t_{left}, t'_{right}, k_{parent}, k_{grandP})$. The name and type n'_{left} and t'_{left} (or n'_{right} and t'_{right}) are different from those in the positive example. To create negative examples that a programmer might also create by accident, we use alternative operands that occur in the same file as the binary operation. For example, given `bits << 2`, the approach may transform it into a negative example `bits << next`, which is likely to yield incorrect code.

4.4.2 Code representation

The vector representation of the positive and negative examples is the same as in Section 4.3.

5 Implementation

The code extraction and generation of training examples is implemented as simple AST traversals based on the Acorn JavaScript parser.[2] The training data generator writes all extracted data into text files. These files are then read by the implementation of the bug detector, which builds upon the TensorFlow and Keras frameworks for deep learning.[3]. The large majority of our implementation is in the generic framework, whereas the individual bug detectors are implemented in about 100 lines of code each.

Bug detector	Examples	
	Training	Validation
Swapped arguments	1,450,932	739,188
Wrong assignment	2,274,256	1,090,452
Wrong bin. operator	4,901,356	2,322,190
Wrong bin. operand	4,899,206	2,321,586

Table 2. Statistics on extraction and generation of training data.

The following provides some details on the neural networks we use. The classifier that represents the bug detector is a feedforward network with an input layer of a size that depends on the code representation provided by the specific bug detector, a single hidden layer of size 200, and an output layer with a single element. We apply a dropout of 0.2 to the input layer and the hidden layer. We use binary cross-entropy as the loss function and train with the RMSprop optimizer for 10 epochs with batch size 100. The network that learns embeddings for identifiers is also a feedforward network with a single hidden layer that contains the learned embedding of size 200. We use binary cross-entropy as the loss function and train for two epochs with a batch size of 50 using the the Adam optimizer.

6 Evaluation

We evaluate DeepBugs by applying it to a large corpus of JavaScript code. Our main research questions are: (i) How effective is the approach at distinguishing correct from incorrect code? (ii) Does the approach find bugs in production JavaScript code? (iii) How long does it take to train a model and, once a model has been trained, to predict bugs? (iv) How useful are the learned embeddings of identifiers compared to a simpler vector representation?

6.1 Experimental Setup

As a corpus of code, we use 150,000 JavaScript files provided by the authors of earlier work.[4] The corpus contains files collected from various open-source projects and has been cleaned by removing duplicate files. In total, the corpus contains 68.6 million lines of code. We use 100,000 files for training and the remaining 50,000 files for validation. All experiments are performed on a single machine with 48 Intel Xeon E5-2650 CPU cores, 64GB of memory, and a single NVIDIA Tesla P100 GPU.

[2]https://github.com/ternjs/acorn
[3]https://www.tensorflow.org/ and https://keras.io/

	Embedding	
	Random	AST-based
Swapped arguments	93.88%	94.53%
Wrong assignment	77.73%	84.23%
Wrong bin. operator	89.15	91.68%
Wrong bin. operand	84.79	88.55%

Table 3. Accuracy of bug detectors. The last column shows DeepBugs applied to previously unseen code.

6.2 Extraction and Generation of Training Data

Table 2 summarizes the training and validation data that DeepBugs extracts and generates for the four bug detectors. Each bug detector learns from several millions of examples, which is sufficient for effective learning. Half of the examples are positive and negative code examples, respectively. Manually creating this amount of training data, including negative examples, would be impractical, showing the benefit of our automated data generation approach.

6.3 Accuracy and Recall of Bug Detectors

To evaluate the effectiveness of bug detectors built with DeepBugs, we conduct two sets of experiments. First, reported in the following, we conduct a large-scale evaluation with thousands of artificially created bugs, which allows us to study the accuracy and the recall of each learned bug detector. Second, reported in Section 6.4, we apply the bug detectors to unmodified real-world code and manually inspect the reported warnings to assess the precision of the learned bug detectors.

To goal of the first sets of experiment is to study the accuracy and recall of each bug detector at a large scale. Informally, accuracy here means how many of all the classification decisions that the bug detector makes are correct. Recall here means how many of all bugs in a corpus of code that the bug detector finds. To evaluate these metrics, we train each bug detector on the 100,000 training files and then apply it to the 50,000 validation files. For the validation files, we use the training data generator to extract correct code examples C_{pos} and to artificially create likely incorrect code examples C_{neg}. We then query the bug detector D with each example c, which yields a probability $D(c)$ that the example is buggy. Finally, we compute the accuracy is as follows:

$$accuracy = \frac{|\{c \mid c \in C_{pos} \wedge D(c) < 0.5\}| + |\{c \mid c \in C_{neg} \wedge D(c) \geq 0.5\}|}{|C_{pos}| + |C_{neg}|}$$

The last column of Table 3 shows the accuracy of the bug detectors. The accuracy ranges between 84.23% and 94.53%, i.e., all bug detectors are highly effective at distinguishing correct from incorrect code examples.

[4]http://www.srl.inf.ethz.ch/js150.php

The recall of a bug detector is influenced by how many warnings the detector reports. More warnings are likely to reveal more bugs, but in practice, developers are only willing to inspect some number of warnings. To measure recall, we assume that a developer inspects all warnings where the probability $D(c)$ is above some threshold. We model this process by turning D into a boolean function:

$$D_t(c) = \begin{cases} 1 & \text{if} \quad D(c) > t \\ 0 & \text{if} \quad D(c) \leq t \end{cases}$$

where t is a configurable threshold that controls how many warnings to report. Based on D_t, we compute recall as follows:

$$recall = \frac{|\{c \mid c \in C_{neg} \wedge D_t(c) = 1\}|}{|C_{neg}|}$$

Figure 3 shows the accuracy of the four bug detectors as a function of the threshold for reporting warnings. Each plot shows the results for nine different thresholds: $t \in \{0.1, 0.2, \ldots, 0.9\}$. As expected, the recall decreases when the threshold increases, because fewer warnings are reported and therefore some bugs are missed. The results also show that some bug detectors are more likely than others to detect a bug, if a bug is present.

6.4 Warnings in Real-World Code

The experiments reported in Section 6.3 consider the accuracy and recall of the learned bug detectors using artificially created bugs. We are also interested in the effectiveness of DeepBugs in detecting programming mistakes in real-world code. To study that question, we train each bug detector with the 100,000 training files, then apply the trained detector to the unmodified 50,000 validation files, and finally inspect code locations that each bug detector reports as potentially incorrect. For each bug detector, we inspect all warnings reported with threshold $t = 0.99$, which yields a total of 290 warnings. After inspection, we classify each warning in one of three categories:

- A warning points to a *bug* if the code is incorrect in the sense that it does not result in the expected runtime behavior.
- A warning points to a *code quality problem* if the code yields the expected runtime behavior but should nevertheless be changed to be less error-prone. This category includes code that violates widely accepted conventions and programming rules traditionally checked by static linting tools.
- A warning is a *false positive* in all other cases. If we are unsure about the intended behavior of a particular code location, we conservatively count it as a false positive. We also encountered various code examples with misleading identifier names, which we classify as false positives because the decision whether an identifier is misleading is rather subjective.

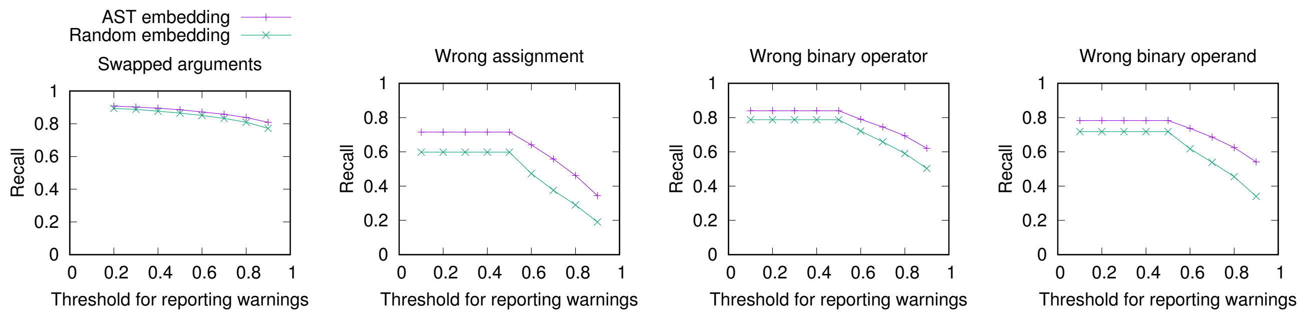


Figure 3. Recall of four bug detectors with different thresholds t for reporting warnings. Each plot contains nine data points obtained with $t \in \{0.1, 0.2, ..., 0.9\}$. The data labeled “AST embedding” corresponds to the DeepBugs approach.

Bug detector	Reported	Bugs	Code quality problem	False positives
Swapped arguments	178	75	10	93
Wrong assignment	24	1	1	22
Wrong bin. operator	50	14	17	19
Wrong bin. operand	38	10	4	22
Total	290	100	32	156

Table 4. Results of inspecting and classifying warnings in real-world code.

Table 4 summarizes the results of inspecting and classifying warnings. The four bug detectors report a total of 290 warnings. 100 of them point to bugs and 32 point to a code quality problem, i.e., roughly half of all warnings point to an actual problem. Given that the bug detectors are learned automatically and do not filter warnings based on any manually tuned heuristics, these results are very encouraging. Existing manually created bug detectors typically provide comparable true positives rates, but heavily rely on heuristics to filter likely false positives. Many of the detected problems are difficult to detect with a traditional, not name-based analysis, because the programming mistakes is obvious only when understanding the semantics of the involved identifiers and literals. We discuss a selection of representative examples in the following.

6.4.1 Examples of Bugs

Buggy call of **`setTimeout`** The following code is from Angular.js and has been fixed (independent of our work) in version 0.9.16 of the code. The first argument of `setTimeout` is supposed to be a callback function, while the second argument is supposed to be the delay after which to call that function.

```
browserSingleton.startPoller(100,
  function(delay, fn) {
    setTimeout(delay,fn);
  });
```

Buggy call of **`Promise.done`** The following code is from Apigee’s JavaScript SDK. The `Promise.done` function expects an error followed by a result, but line 6 passes the arguments the other way around.

```
1  var p = new Promise();
2  if (promises === null || promises.length === 0) {
3    p.done(error, result)
4  } else {
5    promises[0](error, result).then(function(res, err) {
6      p.done(res, err);
7    });
8  }
```

Buggy call of **`assertEquals`** The following code is from the test suite of the Google Closure library. The arguments of `assertEquals` are supposed to be the expected and the actual value of a test outcome. Swapping the argument leads to an incorrect error message when the test fails, which makes debugging unnecessarily hard. Google developers consider this kind of mistake a bug [37].

```
assertEquals(tree.remove('merry'), null);
```

Incorrect operand in for-loop In code from Angular-UI-Router, the developer compares a numeric index to an array, instead of comparing to the length of the array. The bug has been fixed (independent of us) in version 0.2.16 of the code.

```
for (j = 0; j < param.replace; j++) {
  if (param.replace[j].from === paramVal)
    paramVal = param.replace[j].to;
}
```

Incorrectly ordered binary operands The following is from DSP.js. The highlighted expression at line 5 is intended to alternate between `true` and `false`, but is `false` for all iterations except `i=1` and `i=2`.

```
1  for(var i = 0; i<this.NR_OF_MULTIDELAYS; i++){
2    // Invert the signal of every even multiDelay
3    outputSamples = mixSampleBuffers(outputSamples,
4        this.multiDelays[i].process(filteredSamples),
5        2%i==0, this.NR_OF_MULTIDELAYS);
6      /*^^^^^^^*/
7  }
```

Bug detector	Training		Prediction	
	Extract	Learn	Extract	Predict
Swapped arguments	7:46	20:29	2:56	5:19
Wrong assignment	2:40	22:45	1:29	4:03
Wrong bin. operator	2:44	51:16	1:28	12:16
Wrong bin. operand	2:44	51:13	1:28	10:09

Table 5. Time (min:sec) required for training and using a bug detector across the entire code corpus.

6.4.2 Examples of Code Quality Problems

Error-prone binary operator The following code from Phaser.js is correct but using `!==`, instead of <, as the termination condition of a for-loop is generally discouraged. The reason is that the loop risks to run out-of-bounds when the counter is incremented by more than one or assigned an out-of-bounds value, e.g., by an accidental assignment in the loop body.

```
for (var i = 0, len = b.length; i !== len; ++i) {
  ..
}
```

6.4.3 Examples of False Positives

We discuss some representative examples of false positives. Many of them are related to poor variable names that lead to surprisingly looking but correct code. Another common reason are wrapper functions, e.g., `Math.max`, for which our approach extracts a generic name, "max", that does not convey the specific meaning of the value returned by the call expression. We believe that some of these false positives could be avoided by training with an even larger code corpus. Another recurring pattern of false positives reported by the "incorrect assignment" bug detector is code that shifts function arguments at the beginning of an overloaded function:

```
function DuplexWrapper(options, writable, readable) {
  if (typeof readable === "undefined") {
    readable = writable;
    writable = options;
    options = null;
  }
  ...
}
```

The code works around the lack of proper function overloading in JavaScript by assigning the ith argument to the $(i + 1)$th argument, which leads to surprising but correct assignments.

6.5 Efficiency

Table 5 shows how long it takes to train a bug detector and to use it to predict bugs in previously unseen code. The training time consists of the time to gather code examples and of time to train the classifier. The prediction time also consist of the time to extract code examples and of the time to query the classifier with each example. Running both training and prediction on all 150,000 files takes between 31 minutes and 68 minutes per bug detector. The average prediction time per JavaScript file is below 20 milliseconds. Even though this efficiency is, in parts, due to parallelization, it shows that once a bug detector is trained, using it on new code takes very little time.

6.6 Usefulness of Embeddings

To evaluate the usefulness of the embeddings that represent identifiers as vectors (Section 3), we compare these embeddings with a baseline vector representation. The baseline assigns to each identifier considered by DeepBugs a unique, randomly chosen binary vector of length e, i.e., the same length as our learned embeddings. We compare AST-based embeddings (Section 3 with the baseline w.r.t. accuracy and recall. Table 3 shows in the "Random" column what accuracy the bug detectors achieve with the baseline. Compared to the accuracy with the AST-based embedding (last column), the AST-based embeddings yield a more accurate classifier. Figure 3 compares the recall of the bug detectors with the two embeddings. For all bug detectors, the AST-based embeddings increase recall, for some bug detectors by around 10%. The reason is that the AST-based embedding enables the bug detector to reason about semantic similarities between syntactically different code examples, which enables it to learn and predict bugs across similar examples. For example, the bug detector that searches for swapped arguments may learn from examples such as `done(error, result)` that `done(res, err)` is likely to be wrong, because `error` ≈ `err` and `result` ≈ `res`. We conclude from these results that the AST-based embeddings improve the effectiveness of DeepBugs. At the same time, the bug detectors achieve relatively high accuracy and recall even with randomly created embeddings, showing that the overall approach has value even when no learned embeddings are available.

7 Related Work

7.1 Machine learning and Language Models for Code

The recent successes in machine learning have lead to a strong interest in applying machine learning techniques to programs. Existing approaches address a variety of development tasks, including code completion based on probabilistic models of code [9, 34, 36], predicting fixes of compilation errors via neural networks [8, 14], and the generation of inputs for fuzz testing via probabilistic, generative language models [30] or neural networks [7, 12, 24]. Deep neural networks also help recommend API usages [13], detect code clones [44], classify programs into pre-defined categories [26], and adapt copied-and-pasted code to its surrounding code [4]. A recent survey discusses more approaches [3]. All these approaches exploit the availability of a large number of examples to learn from, e.g., in the form of publicly available code repositories, and that source code has regularities even across projects written by different developers [17]. We also exploit this observation but for a different tasks, bug finding, than the above work. Another contribution is to augment the training data provided by the existing code by generating negative training examples through simple code transformations.

Name-based bug detection relates to existing learning-based approaches that consider identifiers. JSNice [35] and JSNaughty [40] address the problem of recovering meaningful identifiers from minified code, using conditional random fields and statistical machine translation, respectively. Another approach summarizes code into descriptive names that can serve, e.g., as method names [5]. The Naturalize tool suggests more natural identifiers based on an n-gram model [2]. Applying such a tool before running our name-based bug detectors is likely to improve its effectiveness.

Word embeddings, e.g., Word2Vec approach [25], are widely used in natural language processing, which has inspired our embeddings of identifiers. Other recent work has proposed embeddings for source code, e.g., for API methods [29], or for terms that occur both in programs and natural language documentation [45]. Our AST-based embedding is targeted at name-based bug detection. Incorporating another embedding of identifiers into our framework is straightforward.

7.2 Machine Learning and Language Models for Analyzing Bugs

Even though it has been noted that buggy code stands out compared to non-buggy code [33], little work exists on automatically detecting bugs via machine learning. Murali et al. train a recurrent neural network that probabilistically models sequences of API calls and then use it for finding incorrect API usages [27]. In contrast to DeepBugs, their model learns from positive examples only and focuses on bug patterns that can be expressed via probabilistic automata. Bugram detects bugs based on an n-gram model of code. Similar to the above, it learns only from positive examples. Choi et al.˜[]Choi2017 train a memory neural network [43] to classify whether code may produce a buffer overrun. Their model learns from positive and negative examples, but the examples are manually created and labeled. Moreover, it is not yet known how to scale their approach to real-world programs. A key insight of our work is that simple code transformations provide many negative examples, which help learn an effective classifier. Finally, Wang et al. use a deep belief network to find a vector representation of ASTs, which are used for defect prediction [41]. Their approach marks entire files as likely to (not) contain a bug. However, in contrast to our and other bug finding work, their approach does not pinpoint the buggy location.

7.3 Bug Finding

A related line of research is specification mining [6] and the use of mined specifications to detect unusual and possibly incorrect code [22, 32, 42]. In contrast to our work, these approaches learn only from correct examples [16] and then flag any code that is unusual compared to the correct examples as possibly incorrect, or search for inconsistencies within a program [11]. Our work replaces the manual effort of creating and tuning such approaches by learning and does so effectively by learning from both correct and buggy code examples.

Our name-based bug detectors are motivated by manually created name-based program analyses [23, 31, 37]. The "swapped arguments" bug detector is inspired by the success of a manually developed and tuned name-based bug detector for this kind of bug [37]. For the other three bug detectors, we are not aware of any existing approach that addresses these problems based on identifiers. Our work differs from existing name-based bug detectors (i) by exploiting semantic similarities that may not be obvious to a lexical comparison of identifiers, (ii) by replacing manually tuned heuristics to improve precision and recall with automated learning from examples, and (iii) by applying name-based bug detection to a dynamically typed language.

7.4 Other Related Work

Our idea to create artificial, negative examples to train a binary classifier can be seen as a variant of noise-contrastive estimation [15]. The novelty is to apply this approach to programs and to show that it yields effective bug detectors. The code transformations we use to create negative training examples are strongly related to mutation operators [19]. Based on mutation operators, e.g., mined from version histories [10], it is straightforward to create further bug detectors based on our framework.

8 Conclusions

This paper addresses the problem of finding bugs by training a model that distinguishes correct from incorrect code. The key insight that enables our work is to automatically create large amounts of training data through simple code transformations that insert likely bugs into supposedly correct code. We present a generic framework for generating training data and for training bug detectors based on them. Applying the framework to name-based bug detection yields automatically learned bug detectors that discover 132 programming mistakes in real-world JavaScript code. In the long term, we envision our work to complement manually designed bug detectors by learning from existing code and by replacing most of the human effort required to create the bug detector with computational effort.

Acknowledgments

This work was supported in part by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the Emmy Noether project ConcSys, by NSF grants CCF-1409872 and CCF-1423645, and by a gift from Fujitsu Laboratories of America, Inc.

References

[1] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. 14–23.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 281–293.

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *arXiv:1709.06182* (2017).

[4] Miltiadis Allamanis and Marc Brockschmidt. 2017. SmartPaste: Learning to Adapt Source Code. *CoRR* abs/1705.07867 (2017). http://arxiv.org/abs/1705.07867

[5] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2091–2100.

[6] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 4–16.

[7] M. Amodio, S. Chaudhuri, and T. Reps. 2017. Neural Attribute Machines for Program Generation. *ArXiv e-prints* (May 2017). arXiv:cs.AI/1705.09231

[8] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. *CoRR* abs/1603.06129 (2016).

[9] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2933–2942.

[10] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas W. Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 511–522.

[11] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 57–72.

[12] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. *CoRR* abs/1701.07232 (2017).

[13] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 631–642. DOI: http://dx.doi.org/10.1145/2950290.2950334

[14] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *AAAI*.

[15] Michael Gutmann and Aapo Hyvärinen. 2010. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 297–304.

[16] Sudheendra Hangal and Monica S. Lam. 2002. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering (ICSE)*. ACM, 291–301.

[17] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 837–847.

[18] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 132–136.

[19] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2011), 649–678.

[20] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[21] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2015. Gated Graph Sequence Neural Networks. *CoRR* abs/1511.05493 (2015).

[22] Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: Mining More Bugs by Reducing Noise Interference. In *ICSE*.

[23] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen Est Omen: Exploring and Exploiting Similarities between Argument and Parameter Names. In *International Conference on Software Engineering (ICSE)*. 1063–1073.

[24] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic Text Input Generation for Mobile Testing. In *ICSE*.

[25] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 3111–3119.

[26] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. 1287–1293.

[27] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Finding Likely Errors with Bayesian Specifications. *CoRR* abs/1703.01370 (2017). http://arxiv.org/abs/1703.01370

[28] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-Based Statistical Language Model for Code. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 858–868.

[29] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 438–449.

[30] Jibesh Patra and Michael Pradel. 2016. *Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data.* Technical Report TUD-CS-2016-14664. TU Darmstadt.

[31] Michael Pradel and Thomas R. Gross. 2011. Detecting anomalies in the order of equally-typed method arguments. In *International Symposium on Software Testing and Analysis (ISSTA)*. 232–242.

[32] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. 2012. Statically Checking API Protocol Conformance with Mined Multi-Object Specifications. In *International Conference on Software Engineering (ICSE)*. 925–935.

[33] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar T. Devanbu. 2016. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016.* 428–439.

[34] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. In *OOPSLA*.

[35] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code".. In *Principles of Programming Languages (POPL)*. 111–124.

[36] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014.* 44.

[37] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting Argument Selection Defects. In *OOPSLA*.

[38] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80.

[39] Richard Socher, Cliff Chiung-Yu Lin, Andrew Y. Ng, and Christopher D. Manning. 2011. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011.* 129–136.

[40] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 683–693.

[41] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016.* 297–308.

[42] Andrzej Wasylkowski and Andreas Zeller. 2009. Mining Temporal Specifications from Object Usage. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 295–306.

[43] Jason Weston, Sumit Chopra, and Antoine Bordes. 2014. Memory Networks. *CoRR* abs/1410.3916 (2014).

[44] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016.* 87–98.

[45] Xin Ye, Hui Shen, Xiao Ma, Razvan C. Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016.* 404–415.