Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts

Rohan Bavishi and Michael Pradel and Koushik Sen

Technical Report TUD-CS-2017-0296

TU Darmstadt, Department of Computer Science

November, 2017

Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts

Rohan Bavishi
EECS Department
University of California, Berkeley, USA

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany

Koushik Sen
EECS Department
University of California, Berkeley, USA

Abstract

Most of the JavaScript code deployed in the wild has been minified, a process in which identifier names are replaced with short, arbitrary and meaningless names. Minified code occupies less space, but also makes the code extremely difficult to manually inspect and understand. This paper presents Context2Name, a deep learning-based technique that partially reverses the effect of minification by predicting natural identifier names for minified names. The core idea is to predict from the usage context of a variable a name that captures the meaning of the variable. The approach combines a lightweight, token-based static analysis with an auto-encoder neural network that summarizes usage contexts and a recurrent neural network that predict natural names for a given usage context. We evaluate Context2Name with a large corpus of real-world JavaScript code and show that it successfully predicts 60.4% of all minified identifiers. A comparison with the state-of-the-art tools JSNice and JSNaughty shows that our approach predicts 17% and 43% more names than the best existing approaches, while taking only 2.6 milliseconds to predict a name, on average.

Keywords JavaScript, Deobfuscation, Deep Learning

1 Introduction

Developers invest a significant portion of their time in reading and understanding code [33]. This is because programmers need to periodically review their code for defects, to look for regions of code to optimize, extend existing functionality or simply increase their knowledge base [5]. The developer community has come up with various guidelines and styles to be followed while writing programs that can potentially reduce the comprehension overhead. One widely accepted guideline is to use meaningful variable and function names. Ideally, a name should capture its semantic function in a program, effectively acting as an abstraction that developers can use to aid their understanding [15].

In the same vein, however, variable and function names when deliberately designed poorly, can provide a layer of obfuscation that discourages review and inspection. Combined with the removal of formatting, such as indentation and white-spaces, while retaining functionality, can make a program extremely difficult to read for a developer. Code with non-meaningful identifier names is particular common for real-world JavaScript, where most developers apply *minification* before shipping their code. This process replaces all local identifier names with short, arbitrary, and meaningless names. A number of publicly available tools automate minification by *mangling* local names into short, cryptic ones, and by aggressively reusing names in different scopes. The resulting JavaScript files are smaller and thus reduce the download time of, e.g., client-side web application code. In addition, some website might wish to conceal the meaning of the code to protect their intellectual property or to hide their malicious intent.

Minification tools usually produce *source-maps* that map various elements in the minified code back to their original counterparts. Unfortunately, in most cases source-maps are available only to the authors of the JavaScript code as they are not shipped along with the source. To enable external reviewing and security analysis, it is important to develop techniques that attempt to recover the original source code from its minified version, primarily by renaming identifiers to more meaningful names.

This paper addresses the challenge of inferring natural variable names in minified code through deep learning. The key idea in our approach is to capture the syntactic usage context of a variable or function across a JavaScript program and to predict a natural name from this context. To gather usage contexts, we use a lightweight technique that views code as a sequence of lexical tokens and extracts sequences of tokens surrounding each use of a variable. We then use these sequences to train a *recurrent neural network* (RNN) [19] that predicts a natural identifier for the given usage context. Since these sequences can be arbitrarily long, we use a separate *auto-encoder* neural network to generate *embeddings*, which are much smaller in dimensionality, and retain the key features that are sufficient to categorize usage contexts. The RNN used for prediction can be trained much more efficiently with these embeddings.

To train our Context2Name approach, we leverage the huge amount of JavaScript code that is available online. Machine learning techniques exploiting this availability of source code have been used to solve a variety of development tasks, such as code completion [30], fixing syntactic errors [17], code clone detection [38], malware analysis [13], and even generating programs with specific constraints [32].

Deep Learning [21] is a fast-growing field in machine learning that has been very successful in natural language tasks, such as completion, translation and summarization. We show that this effectiveness propagates to the problem of inferring meaningful variable names as well.

We evaluate our technique on a large, publicly available corpus of JavaScript source code, wherein Context2Name is able to recover 60.4% of all minified identifiers *exactly*, i.e., our tool predicts the *same* name that the authors of the original programs had in mind. We also show that our approach is practical. It takes an average of 2.6 milliseconds to predict a name, or 209 milliseconds to process a file, on average.

A comparison with the state-of-the-art tools JSNice [29] and JSNaughty [36] shows that our work significantly improves accuracy without any loss in efficiency. Evaluating the existing tools on the same data set, along with a reasonable time-limit for processing, Context2Name is able to recover 17% and 43% more identifiers than JSNice and JSNaughty respectively. In addition to empirical benefits, our work also contributes conceptually by providing simpler approach that will be easier to adapt to another language than JSNice. The reason is that our work does not perform any program analysis for feature extraction, but instead relies in neural networks to learn which parts of a token stream are relevant for predicting natural names.

In summary, this paper contributes the following:

- A deep learning-based framework to recover natural identifier names from minified JavaScript code.
- A technique for computing vector embeddings of the usage contexts of a variable. The technique makes minimal assumptions about the underlying programming language, and can be adapted to other languages and usage scenarios.
- Empirical evidence that the approach exactly recovers 60.4% of all minified names in a large, publicly available corpus of JavaScript code, which improves over the state-of-the-art by 17%. The average time for processing each file is also well under a second.

The implementation of our approach, as well as all data to reproduce our results, will be made available as soon as the paper gets accepted.

2 Overview and Example

This section presents an informal outline of our approach and illustrates the main ideas with an example. The example is a piece of JavaScript code, shown in the upper-left corner of Figure 1. The code has been minified with UglifyJS, a popular minification tool that replaces all local variable names with short, meaningless, and arbitrarily chosen names. All globally visible names, such as `httpReq`, and property names, such as `responseText`, are not minified to ensure that the code preserves the semantics of the original code. In practice, UglifyJS also removes unnecessary whitespace and indentation, which we preserve here for clarity.

The minification makes it difficult to understand the meaning of individual variables and the overall purpose of the code. The goal of Context2Name is to recover meaningful and natural names that help developers not familiar with the code to reason about it. Ideally, the recovered names are those chosen by the developers in the original code. In general, predicting the exact same names as in the original code is not always possible. The reason is that, even though developers generally tend to write regular and "unsurprising" code [18], developers do not always choose the most natural identifier name, but sometimes choose a semantically equivalent name or a slightly misleading name. To achieve the goal of helping developers understand the code despite minification, it is therefore sufficient to recover some meaningful name, not necessarily the original name, for each local variable.

For the example, the upper-right corner of Figure 1 shows the code with names recovered by Context2Name. Five out of the six local names have been recovered exactly, i.e., as in the original code: `req`, `url`, `method`, `async`, and `callback`. The remaining name, `body`, is also very similar in spirit to the name `data` that was used by the author of the original code. Overall, the recovered code very much reflects the developer's intention, making it easy for another developer to reason about the code.

The key insight used by our approach to recover names is that the code surrounding different occurrences of a local variable offers a lot of information about the meaning of the variable. For example, the tokens surrounding `e` in Figure 1 include the keyword `function`, the global function name `httpReq`, and the property names `open`, `oneadystatechange`, `send`, and `responseText`. An average JavaScript developer can infer from this information that `e` is likely to be an `XMLHttpRequest` object. A more difficult case is the minified name `i`. Its surrounding tokens include `if`, `onreadystatechange`, and `responseText`, which at first glance do not reveal much about the purpose of `i`. A skilled developer who has used `XMLHttpRequest` may learn from the fact that `i` is the third argument of `open` and that `onreadystatechange` is close to the `if` condition where `i` evaluates to true, while `responseText` is close to another `if` condition that involves `i` and a negation. All these features suggest that `i` toggles the asynchronous behavior of an HTTP requests.

Our technique essentially simulates this semantic reasoning based on prior knowledge of JavaScript code. However, instead of relying on an expert JavaScript developer, we exploit the power of advanced machine learning techniques and the availability of large corpora of code. The approach identifies regularities in the way identifier names occur in real code and uses these regularities to predict meaningful names.

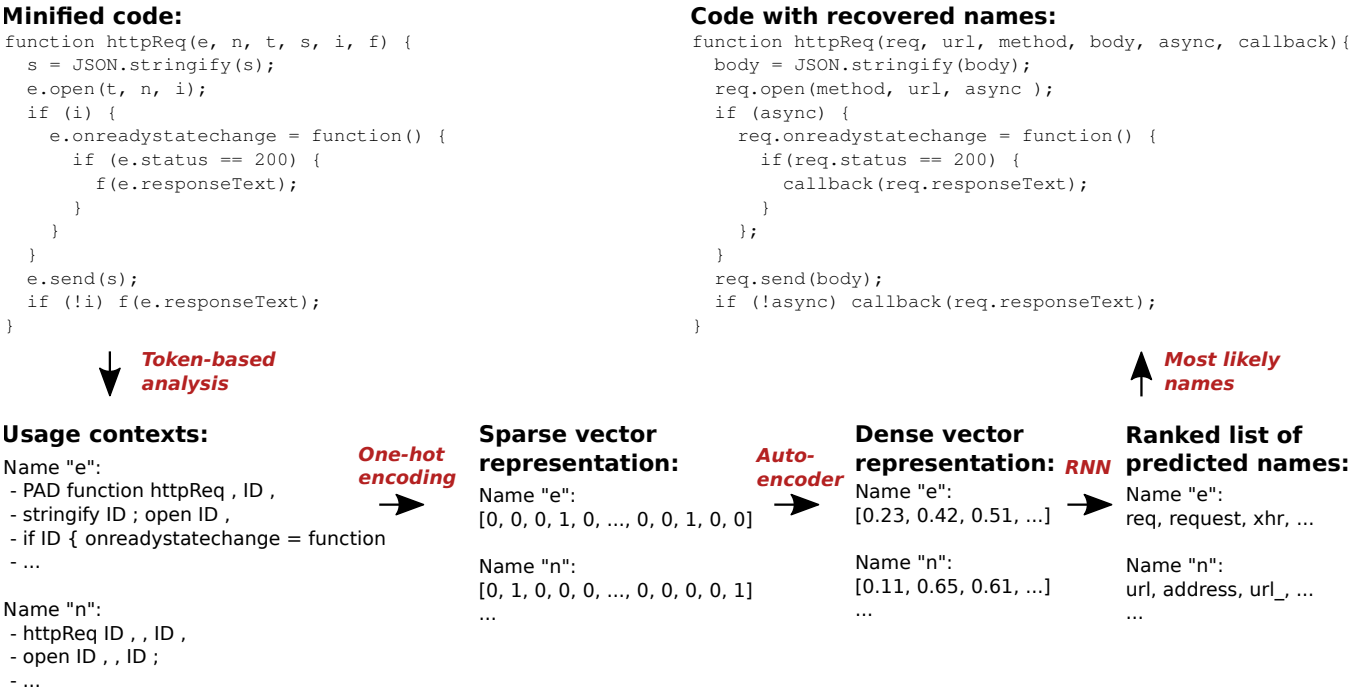

Figure 1. Overview of the Context2Name approach.

Figure 1 illustrates the different steps taken by Context2Name during this process:

1. At first, a simple token-based analysis of the code extracts usage contexts for every minified name. Specifically, the analysis extracts one usage context for each occurrence of the name. The usage context consists the sequence (of length 3 in this case) of lexical tokens before and after the minified name.
2. The next step converts the set of usage contexts of a name into a vector representation. This representation is based on a standard one-hot encoding and yields a sparse, binary vector, where most elements are zero and only few elements are one.
3. To ensure the scalability and efficiency of the overall approach, the next step converts the sparse vectors into a dense vector representation. This step is fully automated by using a sequence auto-encoder [12], i.e., an semi-supervised neural network that searches an efficient encoding.
4. The dense vector representations are fed into a supervised machine learning model, a recurrent neural network (RNN). We train this model to predict a ranked list of meaningful names for a given vector representation of the usage contexts of a variable. A key insight is that the prediction depends only on the usage contexts of the variable, and not on the minified name.
5. Finally, Context2Name greedily selects for each minified variable the name predicted with maximum probability and outputs the code with recovered names.

An important conceptual benefit of Context2Name over the state of the art technique JSNice is that our approach makes very little assumptions about the language of the analyzed programs. Concretely, our approach abstracts program code into a sequence of tokens and assumes to have a way to identify occurrences of the same local variable. In addition to these assumptions, JSNice extracts various hard-coded kinds of relations between program elements from the code, such as between different identifiers involved in the left-hand side and right-hand side of assignments. These relations are specific to the analyzed language, making it non-trivial to adapt JSNice to a different programming language. Instead, Context2Name leaves the task of identifying relevant structural relations between program elements to a machine learning model, making it easier to adapt the approach to other languages.

3 Approach

In this section, we provide the technical details of our approach. The key idea is to approximate the semantic meaning of a variable or function using a sequence of lexical tokens surrounding its different points of usage in the code. We formally describe this notion of a usage summary in Section 3.1. Sections 3.2 and 3.3 then describe two neural networks: One network to reduce the usage summary into an efficient vector representation, and another network to predict a suitable name for a given usage summary. Finally, Section 3.4 presents how to recover all minified names of a program while preserving the semantics of the code.

3.1 Extracting Usage Summaries

The input to Context2Name is the source code of a minified program. As a first step, the approach is to extract a usage summary for a each local variable or locally defined function. The usage summary will be used by later steps of the approach to predict a suitable name for the variable. A usage summary is composed of the different contexts in which an identifier is used. To extract the usage summary, we view the source code as a sequence of lexical tokens $\mathcal{T} =< t_0, t_1, \cdots, t_{len} >$. We drop some tokens, specifically dot punctuators and round parentheses, from $\mathcal{T}$ as we did not find them to be particularly useful. Let $\mathcal{N} = \{n_1, n_2, \cdots, n_k\} \subseteq \mathcal{T}$ be the set of all local names in the code. Because a single name may independently occur in multiple scopes, variables or functions in different scopes having the same name have separate entries in $\mathcal{N}$.

For constructing the context of each occurrence of an identifier, we define a helper function. Let $\mathcal{T}[k]$ denote the k^{th} token in the sequence $\mathcal{T}$. We define a token projection function $g_{\mathcal{T}}$ for $\mathcal{T}$ as follows:

$$g_{\mathcal{T}}(k) = \begin{cases} \mathcal{T}[k] \text{ if } 0 \leq k \leq len \wedge \mathcal{T}[k] \notin \mathcal{N} \\ \texttt{ID} \text{ if } 0 \leq k \leq len \wedge \mathcal{T}[k] \in \mathcal{N} \\ \texttt{PAD} \text{ if } k < 0 \\ \texttt{PAD} \text{ if } k > len \end{cases} \tag{1}$$

The function replaces all local names in the code with a special `ID` token. The reason is that these names are minified in the given code and therefore do not contribute any semantic information. Also note that $g_{\mathcal{T}}$ returns the special padding token `PAD` when its argument is out of range. This case is useful for extracting the context of tokens that appear close to the beginning and end of the analyzed token sequence $\mathcal{T}$.

Based on the helper function $g_{\mathcal{T}}$, we now define the context of an occurrence of a local name. For each occurrence of a local name in $\mathcal{T}$, we extract the q preceding tokens and the q following tokens into a the ***context*** $c(t)$ of a token t in $\mathcal{T}$:

$$\begin{aligned} c(t) =< & g_{\mathcal{T}}(k-q), \cdots, g_{\mathcal{T}}(k-1), \\ & g_{\mathcal{T}}(k+1), \cdots, g_{\mathcal{T}}(k+q) > \text{ if } t = \mathcal{T}[k] \end{aligned} \tag{2}$$

This local context captures the usage of a name at a particular code location. The hyper-parameter q can be configured to adjust the number of tokens extracted as context. We use $q = 5$ as a default in our evaluation.

Finally, for each local name n, we aggregate the contexts for different usages of n in the code into a single sequence of tokens, which we call the ***usage summary*** of the name. We use up to l contexts per name to construct the usage summary, where l is another hyper-parameter that can be configured to adjust the size of summaries. If less than l contexts are available for a particular name, we pad the sequence with the special `PAD` tokens. If more than l contexts are available, we use the first l contexts. We use $l = 5$ as a default in our evaluation. We formally define the usage summary as a function $\mathcal{U}(n)$ where $n \in \mathcal{N}$ as follows ($\circ$ is the sequence composition operator):

$$\begin{aligned} \mathcal{U}(n) = & \, c(t_1) \circ c(t_2) \circ \cdots \circ c(t_l) \\ & \text{where } t_i \;\; \forall 1 \leq i \leq l \;\; \text{are occurrences of } n \end{aligned} \tag{3}$$

For our running example from Section 2, the lower left corner of Figure 1 show the usage contexts of variables `e` and `n` (with parameter $q = 3$ for space reasons).

The set of tokens in the usage contexts, along with their position relative to the variable occurrences, captures the syntactic context of the variable usages. The intuition behind Context2Name is that this context is often sufficient to infer the meaning of the variable.

3.2 Learning Embeddings for Usage Summaries

After extracting usage summaries for each local name in the code, the next step is to summarize them into an efficient vector representation called *embedding*. The motivation for this step is twofold. First, to benefit from a machine model that predicts likely variable names, we need to convert the information extracted from source into a format suitable for machine learning. The neural network model we use here, as many other machine learning models, expects vectors of real numbers as its input. Second, the usage summaries are highly redundant, e.g., because the same kind of token occurs many times and because subsequences of tokens occur repeatedly. To ensure the scalability and efficiency of the overall approach, we compress usage summaries into an efficient vector representation.

One option to convert usage summaries into a compact vector representation would be to manually define a set of features and to create vectors that describe the presence or absence of these features. However, coming up with a suitable set of features that capture the semantics of identifier usages in JavaScript would require significant manual effort. Moreover, manually designed features would tie our approach to a specific programming language, and require additional manual effort to adapt it to another language.

Instead of manually defining how to compress the usage summaries, we use an auto-encoder. An auto-encoder is a supervised neural network model that learns to compress a given vector while preserving as much of the original information as possible. We train a sequence auto-encoder [12] that compresses each context in a usage summary into a compact vector, allowing us to represent the usage summary as the concatenation of these compact vectors.

The first step is to define an input vocabulary $\mathcal{V}_{inp}$ of tokens that the network recognizes. We construct the input vocabulary by picking the most frequent $|\mathcal{V}_{inp}|$ tokens (plus the special `PAD` token) across our training set of code. In our experiments, $|\mathcal{V}_{inp}| = 4,096$. All tokens that are not frequent enough to occur in $\mathcal{V}_{inp}$ are represented by a special `UNK`

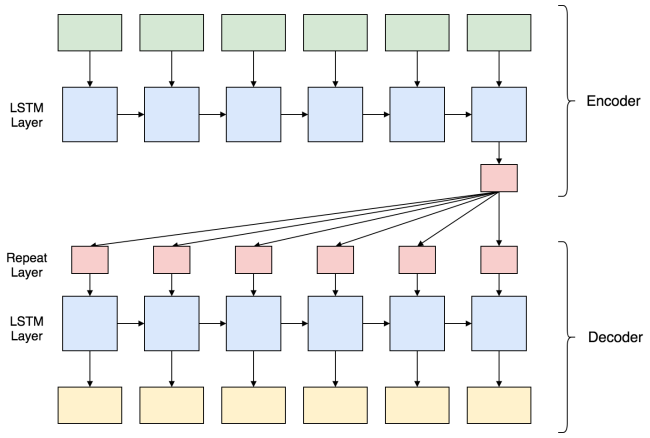

Figure 2. Auto-encoder network that computes an embedding for each context using two RNNs.

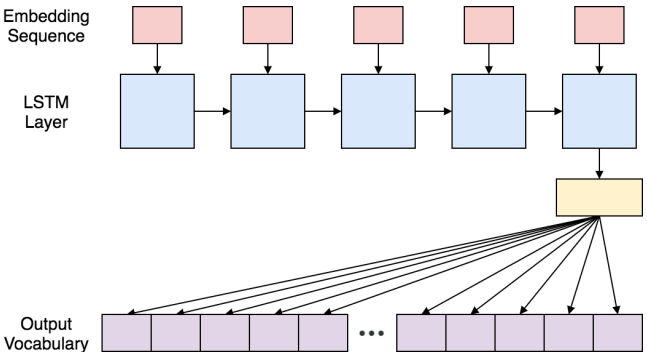

Figure 3. Recurrent neural network to predict likely variable names for a given usage context.

token. The input vocabulary is then used to convert the tokens in the usage summary to their one-hot representations, i.e., binary vectors of size $|\mathcal{V}_{inp}|$ with the i^{th} bit set for the i^{th} token in $\mathcal{V}_{inp}$. The size s_c of the one-hot representation of the context $c(t)$ of a token t is $s_c = |\mathcal{V}_{inp}| \cdot q \cdot 2$ because the context contains q tokens before and q tokens after t, each represented by $|\mathcal{V}_{inp}|$ bits. For our running example, Figure 1 illustrates the one-hot vector representation of the usage contexts of variables e and n.

The one-hot representation of contexts is highly redundant and we next describe how to compress it using a LSTM based sequence auto-encoder. The auto-encoder can be thought of as two jointly learned functions. An encoder function $enc : \{0,1\}^{s_c} \rightarrow \mathbb{R}^{E}_{[0,1]}$, which maps the one-hot representation of a context to a real-valued vector of length E, and a decoder function $dec : \mathbb{R}^{E}_{[0,1]} \rightarrow \mathbb{R}^{s_c}_{[0,1]}$ that maps the real-valued vector back to a binary vector of the original length. The notation $R^{E}_{[0,1]}$ refers to a vector of length E of real-valued numbers in the range $[0,1]$. The goal of training the auto-encoder is to minimize the difference between $dec(enc(c)) = \widetilde{c}$ and the original context vector c. The two functions are trained in tandem to minimize this difference. Once trained, we use the intermediate vector returned by enc for a given context c as the embedding for c. That is, the auto-encoder compresses the input vector corresponding to a context into an embeddings of length E, where E is a hyper-parameter of our approach.

Our implementation of the auto-encoder consist of two jointly trained networks that represent the functions enc and dec. The encoder network consists of a single LSTM layer (a class of recurrent neural network (RNN) models that maintains an internal hidden state and therefore is particularly well-suited to processing sequences of inputs) with an hidden state of size E, which also corresponds to the size of the embedding vectors. The encoder network takes a sequence of $2 \cdot q$ one-hot encoded vectors, denoting a context, and produces an embedding vector of length E. The embedding vector is the final hidden state of the LSTM. We use $E = 80$ as a default value in our implementation. The decoder network consists of a layer that repeats the input $2 \cdot q$ times (the number of tokens in the context), and a single LSTM layer with a hidden state of size $|\mathcal{V}_{inp}|$. Figure 2 provides an illustration of this network.

In summary, we use the encoder component of this network to convert a usage summary for a name $n \in \mathcal{N}$ into a sequence of embeddings:

$$\begin{aligned} &< enc(c_1), enc(c_2), \cdots, enc(c_l) > \\ &\text{where } \mathcal{U}(n) = c_1 \circ c_2 \circ \cdots \circ c_l \end{aligned} \tag{4}$$

For the running example, the "Dense vector representation" part of Figure 1 shows the real-valued vector that results from concatenating the embeddings of each context in the usage summary of each variable.

3.3 Predicting Names from Usage Summaries

Based on the compactly represented usage summaries, we train a second neural network to predict the name of a variable given its usage summary. The intuition behind this idea is that the way a variable is used in code implicitly conveys sufficient knowledge about its meaning to predict a suitable name. We first define an output vocabulary $\mathcal{V}_{out}$ to choose names from. For our experiments, the vocabulary contains the $60,000$ most frequent names encountered in our code corpus.

We then learn a function $\mathcal{P} : \mathbb{R}^{E \times l}_{[0,1]} \rightarrow \mathbb{R}^{|\mathcal{V}_{out}|}_{[0,1]}$. Given a sequence of embeddings that represent a usage summary as in Equation 4, we first reverse it as suggested in [34], so the embeddings constructed out of PAD tokens come first. We then apply the function which yields a probability distribution over the output vocabulary. This probability distribution can be interpreted as a ranked list of predicted names.

To learn function $\mathcal{P}$, we use a recurrent neural network (RNN), i.e., a class of neural models that maintains an internal memory and therefore is particularly well-suited to processing sequences of inputs. The RNN we use consists of a single long short-term memory (LSTM) layer with a hidden state of size h ($h = 3,500$ in our experiments), followed by a softmax layer, which returns a probability distribution. Figure 3 provides an illustration of this network.

The size of the output vocabulary $\mathcal{V}_{out}$ directly corresponds to the range of names our network can output as a prediction. That is, the larger the vocabulary, the higher is the accuracy of name recovery. The trade-off, however, is that the network size increases linearly with $|\mathcal{V}_{out}|$. The vocabulary size we choose for our experiments ($|\mathcal{V}_{out}| = 60,000$) strikes a balance between performance and precision.

In Figure 1, the lower right part shows the names predicted by Context2Name for variables `e` and `n`. As in this example, several of the top-ranked names may convey the semantics of the minified variable.

3.4 Semantics-Preserving Recovery of Names

Our RNN-based predictor outputs a ranked list of possible names for each minified name. The final step is to map each minified name to a single predicted name. This mapping must preserve the semantics of the minified program. Specifically, the same name cannot be mapped as predictions to two different minified variables in the same scope, the predicted name cannot be a keyword, and the predicted name cannot overshadow a name from its parent scope if the name is used in one of its child scopes.

To recover names while respecting these constraints, we compose the ranked predictions for different variables into a single list and then use this list to greedily assign predictions. Algorithm 1 summarizes our approach for recovering names in a consistent and semantics-preserving manner. The procedure PredictNames takes the minified code as input, makes a copy (line 2), and extracts all the minified names using the GetMinifiedNames procedure (line 3). The algorithm then initializes a priority queue, which will use the probabilities of name predictions, as returned by the RNN, as the key for sorting in descending order. The priority queue essentially tracks the minified names yet to be recovered. Lines 6-9 initialize the priority queue with the top predictions for every name along with their corresponding probabilities. To this end, we use a procedure NextPrediction, which for a given minified name, returns a pair where the first element is the next best prediction after the previous invocation for the same name, and the second element is its corresponding probability.

Lines 11-21 greedily replaces all minified names with predicted names, as provided by the priority queue, until all names have been replaced, i.e., until the queue becomes empty. At each iteration of the `while` loop on line 11, the algorithm pops the element with the highest probability from the priority queue (lines 12-14). Then, at line 15, a procedure NoConflicts checks whether the algorithm can replace the minified name with the predicted name without creating conflicts. Specifically, we check whether the name predicted has not already been assigned to a different variable in the same scope, is not a keyword, and does not overshadow a replaced name or global name of the parent scope that is referenced in a child scope. If the check passes, the algorithm replaces the minified name with the prediction in `recoveredCode`. Otherwise, we take the next prediction, and add it to the priority queue. After the loop ends, the algorithm finally returns the recovered code file.

For our running example, the upper right part of Figure 1 shows the code with the names inferred by Context2Name. Even though only five out of the local six names are predicted exactly as in the original code, the code is much more readable than its minified version.

Algorithm 1 Semantics-Preserving Name Recovery

```
1:  procedure PredictNames(minifiedCode)
2:    recoveredCode ← copy(minifiedCode)
3:    minNames ← GetMinifiedNames(minifiedCode)
4:    pQueue ← new PriorityQueue()
5:
6:    for all minName ∈ minNames do
7:      (pred, prob) ← NextPrediction(minName)
8:      pQueue.push((prob, minName, pred))
9:    end for
10:
11:   while pQueue ≠ ∅ do
12:     elem ← pQueue.pop()
13:     minName ← elem.minName
14:     pred ← elem.pred
15:     if NoConflicts(minName, pred) then
16:       Replace minName with pred in recoveredCode
17:     else
18:       (pred, prob) ← NextPrediction(minName)
19:       pQueue.push((prob, minName, pred))
20:     end if
21:   end while
22:
23:   return recoveredCode
24: end procedure
```

4 Evaluation

We have implemented Context2Name in Python using Keras[1] as the deep learning framework. The implementation has 397 lines of Python code. We now present an experimental evaluation of our approach to demonstrate its effectiveness and applicability. Specifically, we attempt to answer the following research questions:

[1] https://keras.io/

RQ1: How effective is our approach at predicting natural names for minified variables and function names in real-world JavaScript code?

RQ2: How does our approach compare to the current state-of-the-art, specifically JSNice [29] and JSNaughty [36]?

RQ3: If the approach efficient enough to be practical and does it scale well to large programs?

To answer these questions, we evaluate Context2Name with a large corpus of real-world JavaScript files. The corpus consist of disjoint sets of training files and validation files. For training, we minify the training files with the popular UglifyJS[2] tool and then train the approach to recover the original names. For validation, we give minified versions of the validation files to Context2Name and then measure the accuracy of recovering the original, unminified names. It is important to note that this accuracy metric gives a *lower-bound* of the effectiveness of any technique, as it is possible that the predicted names may be similar, or even better than the original names. A qualitative evaluation involving the manual inspection of suggested names is beyond the scope of this work.

4.1 JavaScript Corpus

We construct our training and validation corpus using a publicly available data set[3] of JavaScript programs [28]. The data set contains 150,000 non-minified JavaScript files, along with a list of 100,000 files marked for training, and another list of 50,000 files marked for validation. Before our experiments, we clean the corpus by removing 3,150 files that are common between the lists of training and validation files from the list of training files. We then remove duplicates from these lists, followed by the removal of files that cannot be processed by UglifyJS. This cleaning reduces the number of files in the training list to 89,484, and in the testing list to 46,913. For all these files, we run UglifyJS to generate their minified counterparts. During minification, we keep the source maps produced by UglifyJS, which map between the original and the minified names. Source maps are essentially a compact translation from minified to non-minified code, and therefore allow us to map any minified name back to its original for the evaluation.

We present some statistics about the validation corpus. The number of lines of code in the original source files (excluding comments and extra white-space) in the validation corpus ranges from 1 to 76,066 (mean 322.3, median 52). The files contain between 0 to 1,657 unique (per-file, per-scope) local variable names (mean 99.25, median 14), between 0 to 89,065 (mean 441.52, median 48) usages of local variable names, and between 1 and 91,193 (mean 487.77, median 70) usages of all variable names, i.e., both local and global names. Across all files, the number of unique local variable names is 3,802,377. The total number of usages of local variable names is 16,817,737, and the total number of usages of all variable names (global + local) is 18,579,225.

[2]Version 3.1.2, run with the -m (mangler) parameter

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

Parameter	Value
No. q of neighbours used on either side in contexts	5
No. l of contexts used in usage summaries	5
Input vocabulary size $\|\mathcal{V}_{inp}\|$	4,096
Output vocabulary size $\|\mathcal{V}_{out}\|$	60,000
Embedding size E	80
Hidden layer size h	3,500

Table 1. Hyperparameters and values for the evaluation.

4.2 Training

For training Context2Name, we build usage summaries for all minified names in the minified files in the training set, and then train our model by using the original names provided by the source maps as the ground-truth. Across all training files, the total number of unique local variable names is 7,510,539 (i.e. the size of $\mathcal{N}$), which also corresponds to the number of usage summaries used for training. Unique local variable names here excludes any names not minified by UglifyJS to preserve the semantics of the code, such as references to built-in functions and global variables.

4.3 Parameter Selection

The effectiveness and efficiency of Context2Name depends on several hyperparameters. The values of these hyperparameters during the evaluation is provided in Table 1. We found these values to strike a balance between the size of the network and the amount of information used to predict names. In particular, the vocabulary sizes $|\mathcal{V}_{inp}|$ and $|\mathcal{V}_{out}|$ need to be set carefully. Large vocabularies may capture a lot of information and allow many more names to be predicted, but may severely affect performance.

We construct both vocabularies using the training data set, by picking the $|\mathcal{V}_{inp}|$ most frequent tokens across all usage summaries for the input vocabulary, and the $|\mathcal{V}_{out}|$ most frequent names for the output vocabulary. A key finding is that a relatively small number of tokens in the input vocabulary accounts for a significant fraction of all tokens across all usage summaries. The reason is that the frequencies of tokens follow a long-tail distribution: Some tokens, such as `ID` and semi-colons are extremely frequent, whereas many other tokens, such as application-specific literal values, occur only rarely. The situation is similar for the output vocabulary, where a relatively small set of popular identifier names cover the large majority of all occurrences of identifiers.

Tables 2 and 3 show the impact of different vocabulary sizes. The middle column in Table 2 compares the relative size of the input vocabulary to the number of unique tokens

$\lvert\mathcal{V}_{inp}\rvert$	Percentage of unique tokens covered	Percentage of tokens covered
500	0.07	94.54
1,000	0.13	95.55
2,000	0.27	96.38
3,000	0.40	96.79
4,000	0.54	97.06
4,096	**0.55**	**97.08**
5,000	0.67	97.26
6,000	0.80	97.42
7,000	0.94	97.54
8,000	1.07	97.64
9,000	1.21	97.73
10,000	1.34	97.82

Table 2. Impact of input vocabulary size $|\mathcal{V}_{inp}|$. The bold line is our default setting.

$\lvert\mathcal{V}_{out}\rvert$	Percentage of unique names covered	Percentage of names covered
1,000	0.26	63.19
5,000	1.28	76.98
10,000	2.56	81.59
20,000	5.12	85.82
30,000	7.68	88.16
40,000	10.24	89.78
50,000	12.80	91.00
60,000	**15.36**	**92.00**
70,000	17.92	92.80
80,000	20.48	93.48
90,000	23.04	94.04
100,000	25.60	94.58

Table 3. Impact of output vocabulary size $|\mathcal{V}_{out}|$. The bold line is our default setting.

(746,501) seen across all usage summaries in the training set. Note that tokens corresponding to minified names are replaced by the special `ID` token in usage summaries. The right column in Table 2 shows the percentage of all those tokens, seen across all usage summaries, that are present in the input vocabulary. The entries in Table 2 suggest that the input vocabulary, albeit very small in comparison to the set of all *unique* tokens, covers a very high fraction of *all* tokens (97.08% of the tokens from a total of 375,526,950 tokens for $|\mathcal{V}_{inp}| = 4,096$).

Table 3 paints a similar picture for the output vocabulary. The middle column compares the relative size of the output vocabulary to the number of unique non-minified names (390,596) seen in our training set, and the right column shows the percentage of all names covered by the output vocabulary. The entries in bold correspond to the size we chose ($|\mathcal{V}_{out}| = 60,000$). The conclusion is the same as for the input vocabulary: The output vocabulary is small but sufficiently rich (recognizing 92% of the names from a total of 7,510,539 names) to perform naming tasks satisfactorily.

4.4 Setup for JSNice and JSNaughty

To compare Context2Name with the state-of-the-art, we train JSNice on the same training set as ours, enabling an apples-to-apples comparison. We use the JSNice artifact[4], and use the same parameters and arguments as suggested in the accompanying README file. For prediction tasks, we again use the same commandline arguments as provided in the README, with an additional parameter to produce a source map from the minified file to the file recovered by JSNice. We then use this map to compute the accuracy.

Unfortunately, we were unable to train JSNaughty on our training set, as the scripts provided in the docker image[5] failed to process our training data. Therefore, we use the already trained models provided in the image to evaluate the performance of JSNaughty. These models have been trained on a training corpus that exceeds ours significantly (227,000 files for their translation model and 500,000 for their language model), which is likely to give an advantage to JSNaughty. Although this is not an apples-to-apples comparison, it is worth noting that despite the fact that the training corpus used in JSNaughty is significantly bigger than ours, we outperform JSNaughty w.r.t. both accuracy (Section 4.6.1) and efficiency (Section 4.6.3).

For all three tools, we impose a time-limit of 100s for processing a file. If a timeout occurs, we assume that the respective tool failed to recover any names in this file. We believe a time-limit of 100s is reasonable as such tools usually have a web-interface (both JSNice and JSNaughty have a web interface) and response times are critical to satisfactorily serve a large number of users. We have not yet released a web interface for Context2Name to maintain anonymity, but we plan to do so if and when this work is accepted.

4.5 Evaluation Criteria

The evaluation criterion for assessing the effectiveness of the approaches is accuracy, i.e, the ability to recover the original names from minified files. To measure accuracy, we run Context2Name, JSNice, and JSNaughty on each minified file in the validation corpus, and extract a mapping between the minified names and the corresponding predictions made by the tools. We then combine this mapping with the source maps produced by UglifyJS to define a mapping between the original names and the predicted names. Finally, we measure accuracy by computing the percentage of original names that a tool recovers correctly.

[4] http://files.srl.inf.ethz.ch/jsniceartifact/index.html
[5] https://github.com/bvasiles/jsNaughty/DockerFolder

	Predict local names only	Predict all (local + global) names
Count each variable once	Local-Once	All-Once
Count each occurrence of a variable	Local-Repeat	All-Repeat

Table 4. Accuracy metrics used for the evaluation.

Metric	Context2Name	JSNice	JSNaughty	Baseline
Local-Once	60.4%	43.3%	17.65%	0.0%
Local-Repeat	62.8%	47.9%	18.00%	0.0%
All-Once	66.4%	52.3%	31.81%	17.19%
All-Repeat	66.5%	53.4%	26.16%	9.4%

Table 5. Comparison of accuracy (metrics described in Section 4.5) for Context2Name, JSNice, and JSNaughty. Variables in files where JSNaughty times out are counted as not predicted. Baseline represents a tool which does not predict anything.

There are four variants of the "accuracy" metric, which differ in subtle ways. The metrics differ in two dimensions, as illustrated in Table 4. On the one hand, we can measure accuracy either for all variables and functions in the code, which includes global names, or only consider local names. Since minifying global names in a file may break the semantics of the code, minifiers, such as UglifyJS, do not modify these names. Hence, the task of recovering global names is trivial, as they are not minified at all. Arguably, both definitions of accuracy make sense, and therefore we consider both of them.

On the other hand, we can measure accuracy either per variable or per usage of a variable. For example, if a local variable `foo` is referenced three times in the same scope, then the per variable metric counts the prediction for `foo` once, whereas the per usage metric counts the prediction three times. If the variable `foo` appears twice in different scopes in the same file, then the per variable metric counts the prediction for `foo` twice. Again, both definitions make some sense, so we here consider both.

Together, these two dimensions yield four accuracy metrics, Local-Once, All-Once, Local-Repeat, All-Repeat, as shown in Table 4. Metric Local-Repeat can be seen as a weighted computation of metric Local-Once, in that correctly predicting a name for a more frequently used variable would give a higher score. Metric All-Repeat (used by JSNice [29]) allows us to directly gauge the similarity between the original, non-minified file and the recovered file. As a trivial baseline, we also compute a *baseline accuracy*, which gives the percentage of global names only. That is, the baseline accuracy effectively represents the accuracy of a tool that does not predict any names.

4.6 Results

4.6.1 Accuracy

Addressing RQ1 and RQ2, Table 5 provides the four accuracy metrics for all the three techniques. The Local-Once accuracy of Context2Name, i.e., when only unique occurrences of local variable names are considered, is 60.4%, 17% more than JSNice's accuracy of 43.4%. Similarly, Context2Name gains

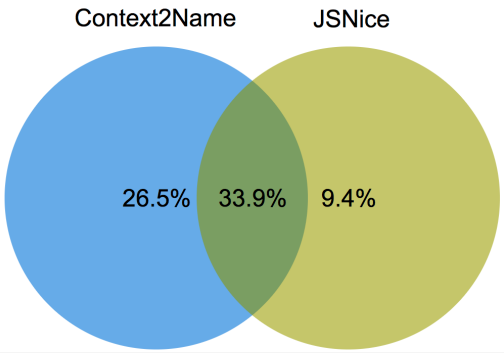

Figure 4. Comparison of predictions by Context2Name, JSNice

14.9%, 14.1% and 13.1% over JSNice when the rest of the metrics are considered.

The comparison with JSNaughty shows our approach to provide a much higher accuracy for all metrics. The main reason is that JSNaughty suffers from an inherent scalability problem: The tool times out on 5,541 files out of the 46,913 files in the validation corpus, and we assume its accuracy to be 0% for variables in these files. The average number of lines of code in these files is 464, and they together constitute a significant fraction (64%) of the total number of unique local variable names across the validation corpus. The authors of JSNaughty appear to be aware of the scalability problem and use only files that have 100 or less lines of code in their evaluation (Section 4.1 in [36]). For the files in our validation corpus where JSNaughty terminates within 100 seconds, JSNaughty achieves a good Local-Once accuracy of 57% (but still smaller than our accuracy). However, taking into consideration all files, it has an overall accuracy of only 14%.

4.6.2 Detailed Comparison

We also analyze the relationships between the sets of names recovered by the respective approaches. For JSNice and JSNaughty, this is captured in the Venn diagram shown in Figure 4. Each approach is represented by a colored circle.

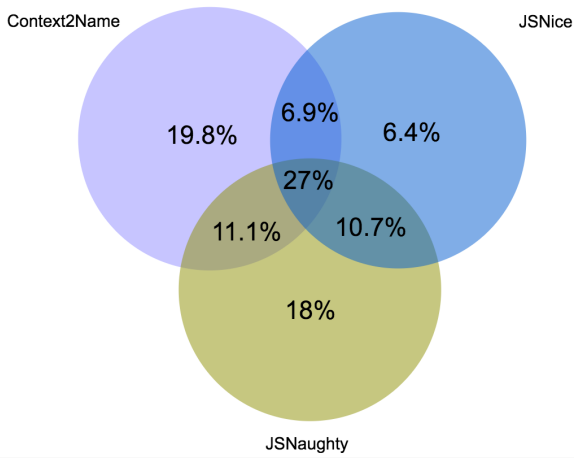

Figure 5. Comparison of predictions by Context2Name, JSNice and JSNaughty

The percentages inside the circle represent the fraction of local minified names for which the original name are recovered (i.e., based on the Local-Once metric). Overall, 26.5% of the names are recovered only by Context2Name, while 9.4% of the names are recovered only by JSNice. 33.9% of the names are recovered by both the tools. It is easy to verify that the sum of percentages inside every circle is equal to the Local-Once accuracies presented in Table 5. The diagram clearly indicates the superiority of Context2Name, in that JSNice outperforms it for only a small fraction of all local names.

We do a similar comparison involving JSNaughty by only considering the subset of files in our validation corpus where JSNaughty does not time out. Across all files in this subset, 19% of the total number of unique local variable names are recovered correctly only by Context2Name, 6.4% only by JSNice and 18% only by JSNaughty. 27% of the names were recovered by all three tools. Overall, the diagram suggests that Context2Name and JSNaughty are complementary to some extent. Therefore it may be possible to create an even better tool by merging the results from Context2Name and JSNaughty, at least for files where JSNaughty does not run into scalability problems.

4.6.3 Efficiency

To address RQ3 about the efficiency and practicality of Context2Name, we measure the time needed for predicting names. We performed our experiments on a 32-core machine with four 2.40 GhZ Intel Xeon processors running Ubuntu 16.04.1 64-bit, with 256GB RAM. We trained and hosted our model on a separate machine with one 4.20 GhZ Intel i7 processor, with one Nvidia 1080Ti GPU, running Ubuntu 16.04.1 64-bit with 48 GB RAM. Training our model on this machine took three days, which is a one-time effort. For prediction tasks, we developed a client running on the first machine that queries the model hosted on the second. Our timing therefore is the sum of processing time on the first machine, and the time taken to query the model loaded on the GPU.

Name	Min. (ms)	Max. (ms)	Mean (ms)	Median (ms)
Our work	0.23	43337	209	48
JSNice	8.23	58900	266	56
JSNaughty	506	99911	19312	11236

Table 6. Timing statistics for Context2Name, JSNice and JSNaughty, computed per file in the testing set. The columns shows the minimum, maximum, mean, and median running time in milliseconds. For JSNaughty, only files recovered within the time limit are considered.

Table 6 shows the per-file timing statistics. Both Context2Name and JSNice perform comparably and are able to process most files in under half a second, making the techniques good candidates for online interactive tools. JSNaughty times out on 5,541 out of the 46,913 files in our testing set (we use a timeout of 100s). On the remaining files, it takes an average of 19.31s, which is quite slow compared to Context2Name and JSNice.

Overall, the results show that Context2Name outperforms the existing techniques both with respect to accuracy and running time.

5 Related Work

5.1 Obfuscation

Program obfuscation has applications in protection of intellectual property [14], resistance against reverse-engineering and software tampering [6, 20, 37] as well as watermarking [31]. An obfuscator is basically a compiler, that takes an input program, and outputs a semantically equivalent program that is mostly unintelligible to a third party without access to the source. Barak et al. [8] showed that obfuscation in general is not realizable. Nonetheless a large body of work has been published on new techniques for obfuscation as it has proved to be practically useful.

Obfuscation is an attractive option in the domain of JavaScript programs as the code is shipped as source, allowing anyone to view and download the original code of the author. But excessive obfuscation can be detrimental to performance w.r.t. bandwidth usage and execution time. Obfuscators that increase the size of the program, or make it significantly slower are not particularly useful. Compatibility is also an issue, as many JavaScript programs rely on external APIs whose usage has to appear in the clear. Minifiers, such as UglifyJS, are an excellent compromise, as the resultant programs are much smaller, and variable renaming is a sufficient deterrence for most adversaries.

5.2 Deobfuscation

Deobfuscation techniques attempt to uncover various aspects of the semantics of the program, which has applications in reverse engineering and malware analysis. Most of the proposed techniques rely on static and dynamic analyses [11, 24, 35], which are more adept at uncovering semantic information about the obfuscated program. Instead, our work focuses on producing friendlier summarizations in the form of variable names, which is useful in the domain of JavaScript programs. Statistical techniques that relate the semantics to the names used in the program are expected to perform better at this task.

5.3 Probabilistic Models for Code

Machine learning has been used in various domains such as natural language, computer vision with tremendous success. These successes have created a lot of interest in applying machine learning to programs, targeting a variety of development tasks such as code completion [10, 27, 30], fixing syntactic errors [9, 17], generating inputs for fuzz testing of programs [16, 22, 26], and generating programs or short code snippets to assist developers or test compilers [4, 32]. Deep learning techniques in particular have been used for code synthesis [7], detecting clones of code fragments [38], and malware detection and signature generation [13]. The core idea behind all these techniques is to exploit the abundance of available source code by mining useful patterns and correlating them to the value of the desired property. In our case, we try to correlate a name to a context comprised of its surrounding tokens, which may include identifiers that have not been minified (global names).

The two closest existing techniques are JSNice [29] and JSNaughty [36], and we show in Section 4 that our approach outperforms both. Conceptually, our work differs from JSNice by not relying on any program analysis to extract features that may be relevant for predicting variable names, but to instead let neural networks decide which parts of the syntactic context matter. A practical benefit of this design decision is that Context2Name is less language-dependent and can be adapted to another language easily. Compared to JSNaughty, an important difference concerns the efficiency and scalability to larger files. As discussed in Section 4.6.1, JSNaughty takes very long for files with only a few hundreds of lines code, which is why the evaluation of JSNaughty in [36] only considers files with up to 100 lines.

Statistical techniques which also use identifiers have been used to tackle a number of problems. JSNice [29], in addition to predicting names, also predicts types for variables in JavaScript. We believe our approach of using sequences of neighbouring lexical tokens can be extended to type prediction as well. Another approach [3] tries to predict concise yet descriptive names that summarize code snippets, which can then effectively serve as method names. Naturalize [1] is a framework that enforces a consistent naming style by providing better names. We expect our number of incorrect predictions to go down after using Naturalize on our testing corpus, as the number of "surprising" and inconsistent names would decrease.

Sequences of lexical tokens are also used by the SmartPaste framework [2] which is designed to assist a developer in extending an existing code base with snippets, by attempting to align the variables in the added snippet with the rest of the code. SmartPaste uses the lexical tokens around a variables in the snippet to compute the probability of it being replaced by some variable in the rest of the code. In addition to using lexically preceding tokens, they also use data flow relations to compute the relevant tokens surrounding the use of a variable. We expect to benefit from using data flow analysis as well.

Embeddings are popular in machine learning-based natural language processing, where they abstract words into vectors [23]. Beyond pure natural language texts, embeddings for code elements have been proposed, e.g., for API elements [25] and for terms that appear both in code and in natural language documentation [39]. Our embeddings differ from these approaches by serving a different purpose, namely to compress usage contexts while preserving their important features, instead of reasoning about individual program elements.

6 Conclusion

This paper addresses the problem of recovering meaningful identifier names in code that has been obfuscated by replacing all local names with short, arbitrary, and meaningless names. We present a deep learning-based approach that exploits and learns from the large amount of available code. The key idea is to predict from the syntactic usage context of a variable what meaning the variable has and to then assign a suitable name. To this end, we combine a lightweight static analysis, an auto-encoder neural network, and a recurrent neural network into a fully automated tool. An evaluation on a set of 150,000 JavaScript files shows that Context2Name clearly outperforms the two state-of-the-art tools JSNice and JSNaughty. Our approach predicts 17% and 43% more names, respectively, then the existing tools, while taking only 2.6 milliseconds per predicted name, on average. Our work contributes a practical tool to help developers understand minified code and shows the power of deep learning to reason about code. Because the approach makes few assumptions about the analyzed programming language and uses a very lightweight, token-based program analysis, it will be easily applicable to other languages.

Acknowledgments

This work was supported in part by NSF grants CCF-1409872 and CCF-1423645, by a gift from Fujitsu Laboratories of

America, Inc., by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, and by the German Research Foundation within the Emmy Noether project ConcSys.

References

[1] 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.

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

[3] 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.

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

[5] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 712–721. http://dl.acm.org/citation.cfm?id=2486788.2486882

[6] Vivek Balachandran and Sabu Emmanuel. 2013. *Software Protection with Obfuscation and Encryption*. Springer Berlin Heidelberg, Berlin, Heidelberg, 309–320. https://doi.org/10.1007/978-3-642-38033-4_22

[7] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). arXiv:1611.01989 http://arxiv.org/abs/1611.01989

[8] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2012. On the (Im)Possibility of Obfuscating Programs. *J. ACM* 59, 2, Article 6 (May 2012), 48 pages. https://doi.org/10.1145/2160158.2160159

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

[10] 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.

[11] Mihai Christodorescu and Somesh Jha. 2003. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=1251353.1251365

[12] Andrew M. Dai and Quoc V. Le. 2015. Semi-supervised Sequence Learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 3079–3087. http://dl.acm.org/citation.cfm?id=2969442.2969583

[13] Omid E. David and Nathan S. Netanyahu. 2015. DeepSign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks, IJCNN 2015, Killarney, Ireland, July 12-17, 2015*. 1–8.

[14] Stephen Drape. 2009. *Intellectual property protection using obfuscation*. Technical Report. University of Oxford.

[15] Edward M. Gellenbeck and Curtis R. Cook. 1991. *An Investigation of Procedure and Variable Names As Beacons During Program Comprehension*. Technical Report. Corvallis, OR, USA.

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

[17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. (2017). https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[18] 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.

[19] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[20] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. 3–9. https://doi.org/10.1109/SPRO.2015.10

[21] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. 521 (27 05 2015), 436 EP –. http://dx.doi.org/10.1038/nature14539

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

[23] 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.

[24] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE Computer Society, Washington, DC, USA, 231–245. https://doi.org/10.1109/SP.2007.17

[25] 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.

[26] 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.

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

[28] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning Programs from Noisy Data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 761–774. https://doi.org/10.1145/2837614.2837671

[29] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

[30] 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.

[31] Christian S. Collberg and Clark Thomborson. 2002. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. 28 (08 2002), 735–746.

[32] Chengxun Shu and Hongyu Zhang. 2017. Neural Programming by Example. *CoRR* abs/1703.04990 (2017). arXiv:1703.04990 http://arxiv.

org/abs/1703.04990

[33] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. 1997. An Examination of Software Engineering Work Practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '97)*. IBM Press, 21–. http://dl.acm.org/citation.cfm?id=782010.782031

[34] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. 3104–3112.

[35] S. K. Udupa, S. K. Debray, and M. Madou. 2005. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*. 10 pp.–. https://doi.org/10.1109/WCRE.2005.13

[36] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar 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)*. ACM, New York, NY, USA, 683–693. https://doi.org/10.1145/3106237.3106289

[37] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. *Software Tamper Resistance: Obstructing Static Analysis of Programs*. Technical Report. Charlottesville, VA, USA.

[38] 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)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

[39] 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.