STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 2, 2023
DOI: 10.24193/subbi.2023.2.01

DEOBFUSCATING JAVASCRIPT CODE USING CHARACTER-BASED TOKENIZATION

ALEXANDRU-GABRIEL SÎRBU

Abstract. The JavaScript code deployed goes through the process of minification, in which variables are renamed using single character names and spaces are removed in order for the files to have a smaller size, thus loading faster. Because of this, the code becomes unintelligible, making it harder to be analyzed manually. Since JavaScript experts can understand it, machine learning approaches to deobfuscate the minified file are possible. Thus, we propose a technique that finds a fitting name for each obfuscated variable, which is both intuitive and meaningful based on the usage of that variable, based on a Sequence-to-Sequence model, which generates the name character by character to cover all the possible variable names. The proposed approach achieves an average exact name generation accuracy of 70.53%, outperforming the state-of-the-art by 12%.

1. Introduction

Over the years, naming variables have proven to be one of the most challenging steps during programming that developers face. Choosing a poor variable name decreases the readability and understanding of the code, since their purpose and meaning is not reflected directly by the label assigned [3]. Thus, programmers communicate their intentions via suggestive names, which can serve as a form of documentation within the code itself, helping other developers understand the code without relying heavily on external comments or documentation.

JavaScript is a widely used programming language, primarily used for web development. JavaScript code is typically embedded directly into HTML documents or included as an external script, running client-side, meaning that

Received by the editors: 31 July 2023.

2010 *Mathematics Subject Classification.* 68T05, 68T50.

1998 *CR Categories and Descriptors.* I.2.6 [**Learning**]: Subtopic – *Connectionism and neural nets*; I.2.7 [**Natural Language Processing**]: Subtopic – *Language generation*.

Key words and phrases. JavaScript deobfuscation, variable name prediction, Deep Learning, Recurrent Neural Network, Abstract Syntax Tree.

it executes in the user's web browser, enabling dynamic content and interactivity without requiring server-side processing. Since JavaScript is a scripting language, meaning that the its code is interpreted, rather than compiled, and because the JavaScript code is ran directly into the user's web browser, this code is visible directly in the web browser, thus allowing users to visualize, analyze it and possibly learn from it.

In order to make the JavaScript code more difficult to understand, developers opt to modify the source code in order to make it less readable, while preserving its functionality. Usually, these methods are based on a mapping from the initial variable and function names to short, arbitrary, non-meaningful identifier names, by using code minification tools, mapping which is available only to the developers of the initial JavaScript code. As the name of the tools' type suggests, the main idea behind such a tool is to reduce the size of the JavaScript file, in order to reduce its loading time [13], thus increasing the performance of the web page, while also providing a layer of code obfuscation.

While code obfuscation is often used on the web pages in order to protect the intellectual property of the code, sometimes its deobfuscation is crucial. For example, deobfuscation can be valuable for security analysis and vulnerability assessment, since obfuscated code can hide potential security risks. Thus, by deobfuscating the code, security professionals can more easily identify and understand the underlying security issues, enabling them to assess the risks accurately and propose appropriate mitigation strategies. Moreover, deobfuscation can also be used in the educational process, allowing developers to learn from and understand obfuscated code. In general, experienced programmers can easily understand obfuscated code, but those who lack the experience require deobfuscation tools in order to gain some insights into advanced programming techniques and algorithms.

Although there are multiple deobfuscation techniques, such as Cloning, Static Path Feasibility Analysis and a combination between Static and Dynamic Analyses [14], our main focus will be to rename the obfuscated variables in order to facilitate the analysis of JavaScript files.

In this paper we propose a deep learning approach to deobfuscate JavaScript source files, which reverses the process of code minification by inferring meaningful variable names based on both their initial assignment and their further usages, using character-based tokenization. A Sequence-to-Sequence model will be used for generating the name character by character to cover all the possible variable names. Thus, our trained model does not rely on the labels that it has been initially trained on, offering a solution which gives decent naming suggestions even on unseen training data, while also being able to suggest better names for constant variables. Experiments will be conducted

on a data set containing real world JavaScript code, and the results obtained by our model will be compared to state-of-the-art approaches.

In short, the work aims to answer the following research questions:

RQ1. How to correctly pick a name for a variable using a deep learning approach, without encountering it beforehand in the training data?

RQ2. How would a generative model compare as opposed to a simple, classification model for the problem of JavaScript code deobfuscation?

The rest of the paper is organized as follows. Section 2 will discuss previous ways of generating variable names, including state-of-the-art techniques and how well they performed. Section 3 will present our approach of choosing the best suited variable name, by using a Sequence-to-Sequence architecture, how the data is handled and how to evaluate our constructed model. Section 4 directly compares our approach to the state-of-the-art approach in the literature, while also discussing our achieved results. The conclusions of the paper and directions for future work are outlined in Section 5.

2. Related work

A deep learning approach is proposed by Bavishi et al. [1], which is compared to two state-of-the-art tools JSNice [11] and JSNaughty [15], against a large corpus of real-world JavaScript code, achieving 47.5% name prediction accuracy, outperforming or performing really close to the tools aforementioned. This approach tokenizes the JavaScript code and uses the context of a variable in order to provide a fitting name for it. Thus, for each occurrence of a local variable, it extracts the q preceding tokens and the q following tokens, concatenating them into the context of that respective variable. Since this approach generates highly redundant usage summaries, since sub-sequences of tokens occur repeatedly, an auto-encoder is used in order to compress the given vector. For prediction, a predefined vocabulary of possible variable names is constructed, and the the authors use a Recurrent Neural Network with a single Long Short-Term Memory layer in order to learn a mapping from the variable context to the variable name. The Recurrent Neural Network is used in order to solve conflicts between variable names: if the predicted variable name is a keyword or if it overshadows a name from its parent scope, then another name prediction is made.

In order to improve the performance of code related tasks, Roziere et al. [12] introduced a new pre-training objective based on deobfuscation and outperforms Masked Language Modeling objectives, such as BERT, on tasks such as code search, code summarization and unsupervised code translation, besides deobfuscating fully obfuscated source files. The pre-training objective used is

represented by replacing class, function and variable names with special tokens, after which the model has to recover the original names, similarly to how Masked Language Modelling's objective is to predict a word based on its context. The deobfuscation objective is realized with a seq2seq model, which is trained to map the obfuscated code into a dictionary represented as a sequence of tokens. This model manages to recover 45.6% of the initial identifier names on the Google BigQuery data set. By solving the task of deobfuscation, the authors showed that this pre-trained model achieves better performances than the BERT model on clone detection, code summarization, natural language to code and on code translation from Python to Java and vice-versa.

The problem of variable name generation also arises when copy-pasting code, and the copied code has to be modified in order to match the context into which it was pasted. Liu et al. [9] have discussed this problem of code adaptation, whose importance rises from the adaptation bugs, raised by inconsistent control flow, inconsistent renaming, inconsistent data flow and redundant operations. To solve this task, the authors collect a data set from GitHub repositories with at least 20 stars. Their goal is to compare their approach with various pre-training objective introduced for code related tasks. Thus, the authors compared three Masked Language Modeling approaches by training a RoBERTa model, the aforementioned model in the previous approach and a CodeT5 model, which is supposed to outperform prior methods on understanding tasks such as code defect detection, clone detection and translation tasks. The model the authors proposed is a transformer with two possible implementations: a uni-decoder transformer, which names a variable based on previously predicted names, and a parallel-decoder transformer, which calculates the individual probability without taking into consideration the other predicted symbols. The second type of transformers predicts a name independently from the rest, factorizing the output distribution per-symbol. The results are somehow predictable, as the uni-decoder transformer achieves higher performance than the pre-trained objective-based models, and has slightly better results than the parallel-decoder transformer.

Jaffe et al. [6] have approached this problem in the form of assigning meaningful variable names for decompiled code. Their solution is based on aligning the line-by-line translation of the decompiled code into meaningful code, using a Statistical Machine Translation, Moses. In decompiled code, variables are automatically given general names, such as $v1$ and $v2$, and Statistical Machine Translation model should rename these variables, while keeping the rest of the code identical. The main idea behind such a model is that the model tries to learn the probability distribution of a sentence in target language to

be the translation of a sentence in the source language. Moses is an open-source Statistical Machine Translation toolkit, which automatically estimates the language and translation models given a sentence-aligned parallel corpus, which, in our case, is the decompiled code against the initial pre-compiled code, which contains meaningful names for variables. This approach achieved a 28.6% exact match for the name of each variable.

3. Methodology

This section introduces our methodology for deobfuscating JavaScript code, using a Sequence-to-Sequence model in order to generate the variable names character-by-character, based on their initial value and their usages, with the goal of answering RQ1.

3.1. **Problem statement.** Our problem can be formulated as follows: given a JavaScript file, rename a variable in such a way that its new name reflects intuitively its purpose. For this task, we will convert the JavaScript code into an Abstract Syntax Tree, after which we will extract that variable's assignment and usages and, based on that, the model will suggest a fitting name for it.

The data set for this problem can be constructed directly based on any available source code, be it online or offline, depending if the target is to write more general code or specialized code in some issue, respectively. Once the code has been selected, there are various tools, such as UglifyJS which, although are intended to be used as minification tools, they usually obfuscate variables to reduce the size of the JavaScript file, making them load faster in order to improving a website's performance. Thus, UglifyJS renames variables and function names to shorter, often single-letter names, removes unnecessary white spaces and comments, and perform other transformations to make the code more compact, without altering its functionality. Since we are later converting the obfuscated code directly into an Abstract Syntax Tree, none of the other operations done by UglifyJS should impact the information we gain through that conversion.

3.2. **Proposed approach.** The first step in our approach is to convert the given code into an Abstract Syntax Tree. An Abstract Syntax Tree is a tree-like data structure that represents the abstract syntactic structure of a program, specific to a programming language, which are commonly used when constructing compilers and when analyzing code. This tree representation, when compared to the simple string interpretation of the input, provides a more structured representation of the code, that captures the hierarchical relationships between different elements of the code, such as functions and loops.

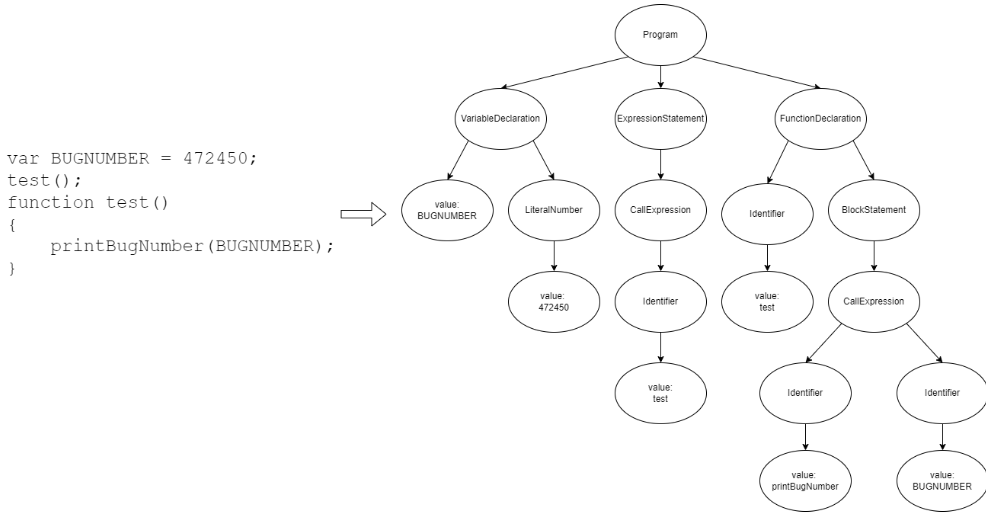

FIGURE 1. JavaScript code example converted into an Abstract Syntax Tree format

Thus, a machine learning algorithm would better understand the code's logical flow and dependencies, which would make it easier to extract meaningful patterns and features. Moreover, Abstract Syntax Trees typically have a lower dimensionality than raw text strings, especially for complex code, whose reduction can lead to both faster training times and better model performance due to reduced complexity [17]. An example of a JavaScript code conversion from plain-text code into an Abstract Syntax Tree format can be seen in Figure 1, where a variable is assigned, a function is called, after which that function is declared.

Usually, when naming a variable, choosing an appropriate and meaningful name is essential for writing clear, maintainable and understandable code. Thus, the chosen name has to clearly describe the meaning and purpose of the variable. The meaning is usually represented by the type of that variable, usually inferred by its initialization. As for the purpose of the variable, it can be inferred directly based on that variable's usages. Programmers tend to rename variables when the purpose of those variables is changed, or if their initial type is completely different. As for experts in code deobfuscation, they can intuitively guess a variable's purpose based on its usages, but still choose to rename them in order to aid them for further deobfuscation. Because of this, we will extract from the constructed Abstract Syntax Tree the initialization of our current variable, and its usages i.e., lines of code where the variable has been used, where the initial variable name is replaced by a special token.

3.2.1. *Deep learning model.* In our task, since we will generate the sequence of characters that determine the name of the variable, we will use a Sequence-to-Sequence model. This model is a type of deep learning model, which is composed on an Encoder and a Decoder. The Encoder is an Recurrent Neural Network which processes the input sequence and generates a fixed-length context vector, also known as the encoding. The Encoder works by reading the input sequence step by step and encoding the information into a context vector, aiming to capture the semantic representation of the input sequence. The second component, the Decoder, is also a Recurrent Neural Network, which takes the context vector produced by the encoder as its initial hidden state, then it generates the output sequence step by step, one token at a time. The Sequence-to-Sequence model architecture allows to handle input and output sequences of different lengths, by compressing the variable-length input into a fixed-length context vector, after which the output sequence is generated token by token based on that context vector.

In our approach, variable names and values are encoded in character-level, in order to accommodate for new values, unseen in our training data, to be handled correctly by the model constructed [10]. Moreover, using a character-level encoding, it is possible to capture sub-word information of, for example, a class name and its characteristics, and may allow the model to understand prefixes, suffixes and stems, thus giving the model the ability to understand word inflections and grammatical variations better. Another advantage is represented by the removal of noise in the form of typos, and the model may recognize similar words based on their character-level similarity, even if there might be some minor spelling differences, while also being more memory efficient because the vocabulary size is substantially reduced. When encoding Abstract Syntax Tree Nodes, we will be liniarizing each node using Breadth First Search in order to maintain the structural equivalence, while also keeping the model relatively more lightweight [5]. Thus, each node type will have a specific label, which will be stored in the vocabulary specific to the JavaScript code, alongside the ASCII characters for the value of these nodes.

The proposed model will follow the Sequence-to-Sequence architecture, which is composed of two components: the Encoder and the Decoder, whose architecture can be seen in Figure 2. The Encoder receives the input tokenized and converts it into a more dense and continuous representation, via the Embedding layer, whose purpose is to capture the semantic relationships between tokens. As a regularization technique, a Dropout will be used, in order to prevent overfitting and to improve the generalization of the model. The Gated Recurrent Unit, which computes the hidden state of the input and forwards it to the decoder as a context, by using both the output of the Decoder and

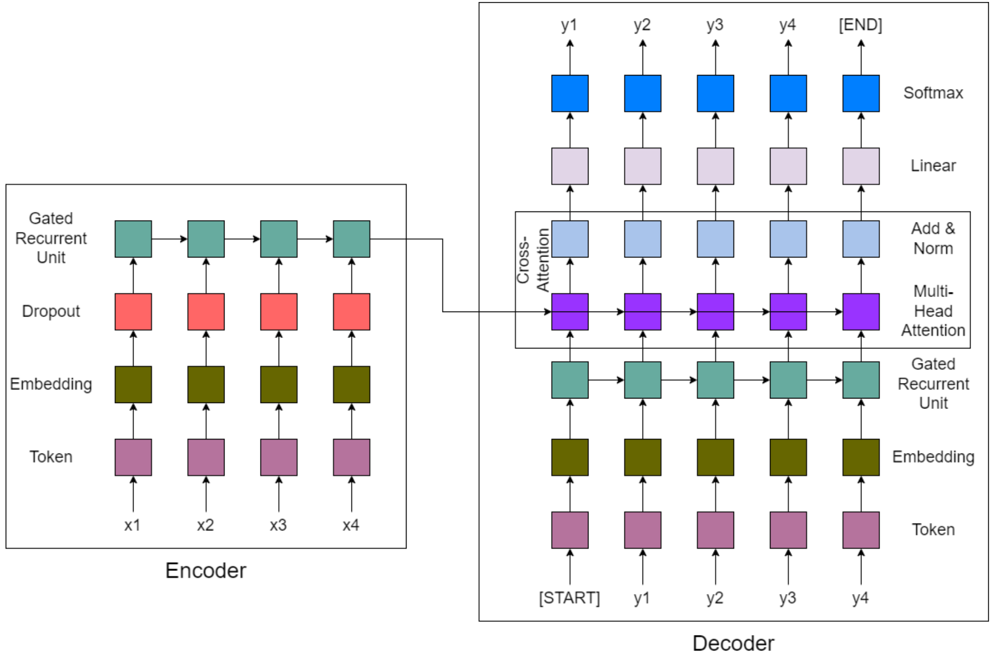

FIGURE 2. Sequence-to-Sequence deep learning model architecture

the output of previous Gated Recurrent Units. The Decoder follows a similar data flow, but this time, the previous output result will be the input to the Embeddings layer, forwarded to a Gated Recurrent Unit. After that, the output of the Gated Recurrent Unit, together with the context resulted from the Encoder will go through a Cross Attention layer, whose purpose is to allow the Decoder to focus on relevant parts of the source sequence, while generating each word of the target sequence [16]. A further processing of the results is done through the Linear layer, and a Softmax layer extracts the best next character for our resulting variable name.

The formula based on which the character at position t is generated is given by:

$$c_t = softmax(f_{seq2seq}([init : usages], s_{t-1})),$$

where $f_{seq2seq}$ is the function the Sequence-to-Sequence model tries to approximate, $init$ represents the embedding of the initialization of the variable, $usages$ is the embedding of the usages of that variable, [:] denotes vector concatenation, and s_{t-1} is the previous hidden decoder step.

3.2.2. *Performance evaluation.* For evaluating our model's performance, we will apply k-fold cross-Validation, splitting the data set into a two components: one for training, and the other for validation and testing. The latter component will be split in half, resulting in 80% of the data set being used for training, 10% for validation and the other 10% for testing, when choosing $k = 5$. Thus, we will be able to give a proper confidence interval, which should give a better performance measure grasp over the data set used [4].

For our task, two evaluation metrics will be used on a testing data set: one evaluation metric which computes the accuracy of each character prediction, since we are using Recurrent Neural Networks, and another evaluation metric which scores the accuracy of each name predicted. The second metric helps us compare to other approaches, in order to see how well the model proposed by us fares against the other approaches proposed.

4. Experimental results

This section presents the experimental results obtained by evaluating the performance of the approach introduced in Section 2 for deobfuscating JavaScript code. In addition, a direct comparison to Context2Name [1], JSNice [11] and JSNaughty [15] is conducted, in order to answer RQ2. Section 4.1 will describe the data set that we are working on, then the experimental setup and the parameters employed for the deep learning model are presented in Section 4.2. A discussion on the obtained results and a comparison to related work is further conducted in Section 4.3.

4.1. **Data set.**

At this step we will construct a data set similar to the one described by Bavishi et al. [1], to be able to compare our model with the state-of-the-art. Thus, from all the files from a publicly available data set[1] of JavaScript programs, which contains 150 thousands non-minified JavaScript files, the duplicate files, the ones very large and the ones that cannot be processed will be removed. After that removal, 97979 files remain which contain 2667804 total variables and 239007 unique names. The minification of JavaScript code is done using UglifyJS, which reduces the file size of those files by renaming variables and removing spaces.

As it can be seen in Figure 3, the most frequent names that the variables have are those with a more generic value, such as *len* or *length*, which usually describe the size of an object or array, *result*, which is usually the returned variable from a function call or operation, and *self*, which referred to the current browser window. There are also many variables names that are either

[1]https://www.sri.inf.ethz.ch/js150

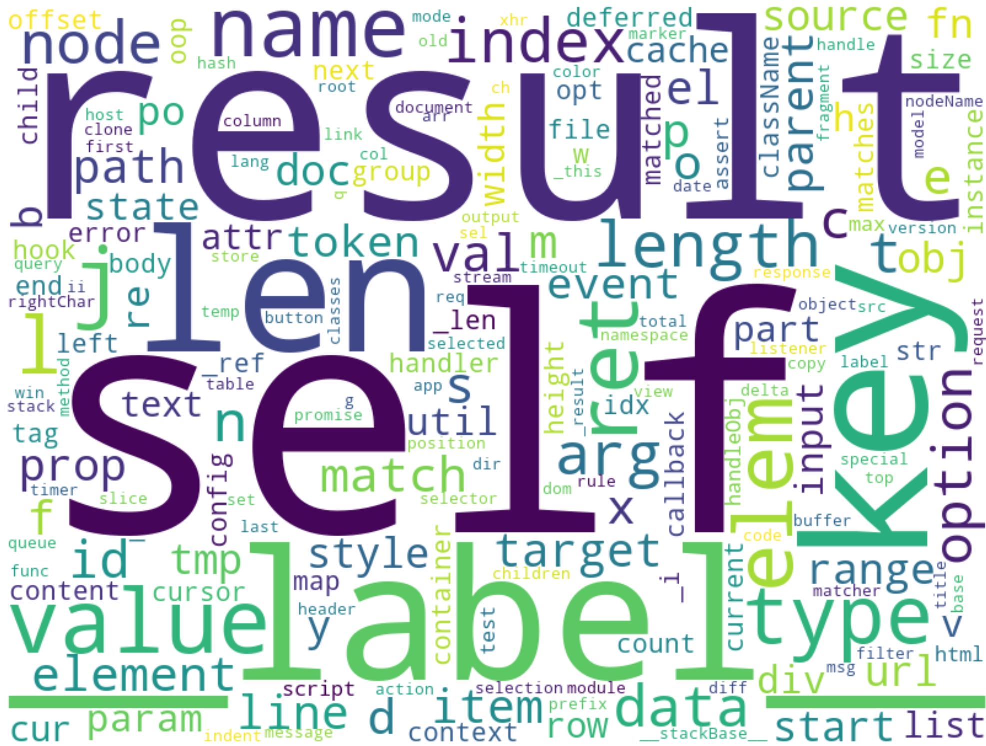

Figure 3. Word cloud over the most frequent variable names found in the dataset

highly generic, such as x, or whose name describes perfectly what it refers to, such as *style* or *error*.

The Recurrent Neural Network architecture highly depends on the size of the output, i.e., in our case, the length of the variables' name. Thus, as it can be seen in Figure 4, although the length can vary infinitely, most of the cases, a name has 4 characters, and the variable names with a length higher than 6 follow a standard exponential distribution.

4.2. **Experimental setup.** The input for our model is represented by both the context of the variable, i.e. its initialization and usages, and the previous hidden state of the Decoder's Gated Recurrent Unit, used for generating the next character for the mentioned variable. Thus, we will require two vocabularies: one for the code component, and one for the name of the variable. From our tests, the vocabulary for code has a size of 158, while the size for the name's vocabulary will be 77. In the data set, all non-ASCII characters have

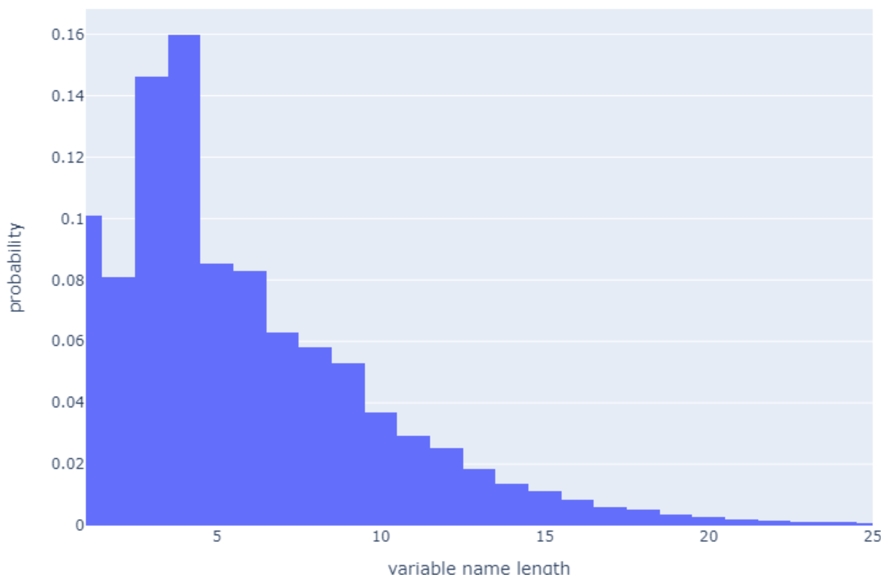

FIGURE 4. The distribution of the variable name length

been removed, since they would hundredfold these sizes, making them more difficult to align and the model more complex.

In our experiment, the architecture used has been described in Section 3.2.1. After constructing the vectors for the initialization and the usages, we decided to keep the first 300 tokens from them, while also picking only the first 3 usages, and concatenating everything, thus obtaining a vector with a size of 1200. The Encoder's Embeddings layer will have an input equal to the size of the code's vocabulary, and an output of 256. It will be followed by a Bidirectional Gated Recurrent Unit, which increases the output size to 512. The Decoder's Embeddings layer will have an input size equal to the size of the name's vocabulary and an output of 256, being followed by an Unidirectional Gated Recurrent Unit, which has the output equal to 256. Thus, the Cross-Attention layer receives a question having the size of 512, and the key equal to 256, having as an output a vector of size 512. The Decoder's Linear layer has an input of size 512 and an output equal to the name's vocabulary size. Thus, the total number of parameters that have to be trained is 738,381.

4.3. **Results and discussion.** After applying a 5-fold cross-validation as described in Section 3.2.2, we obtained an average character generation accuracy of 96.52% and an average name match of 70.53% on our test date sets. For each iteration of the cross-validation algorithm, we obtained the following name match accuracies: 70.81%, 81.16%, 61.70%, 66.30% and 72.68%. Thus, we obtain $[64.13, 76.93]$ name match accuracy percentage as a confidence interval, with a confidence of 95%. The 95% confidence interval (CI) [2] has been computed by using the formula $[\mu - \alpha, \mu + \alpha]$, where μ is the mean of the accuracies obtained during the 5-fold cross-validation and α is the margin of error, computed as

$$\alpha = 1.96 \frac{\sigma}{\sqrt{5}}$$

(σ is the standard deviation of the obtained accuracies). Although our obtained 95% CI (6.4%) is large enough, the *name match accuracy* metric employed is more restrictive than the standard character generation accuracy, mostly because of the variable sequence length of both the input and the output, which might lead the model to learn certain parts of target name at different epochs. Moreover, although our goal is to match the generated name fully to the one predetermined, partial matches might still be valuable, even if the final output does not perfectly match the target sentence, which is not considered by our metric.

A direct comparison to other state-of-the-art approaches can be seen in Table 1, where our approach and its results are marked with bold. The table depicts the name match accuracies for our **Seq2Seq** model compared to the state-of-the-art tools Context2Name [1], JSNice [11] and JSNaughty [15]. This comparison has been made using the results available in the previous work [1], which have been computed on the exact same data set and testing methodology. During the training of our model, validation loss has decreased in conjunction with the training loss, which can be seen in Figure 5, which shows that the model was not overfitted. Moreover, in Figure 6, we can see a direct comparison between the output generated from UglifyJS, where each variable name is obfuscated using a single letter. The output of JSNice, as compared to the output generated from our model provides a level of confusion regarding the variables fb and $Helper$. In this case, our model generates wrongfully only one name, i.e. *adddataService*, as opposed to the instance found in the dataset *addDataService*, which proves that the model correctly learned that lowercase letters have the same meaning as capital letters.

As it can be seen in Figure 7, both most frequent and the most inaccurately predicted type is represented by function calls, whose assigned name can be challenging for programmers, thus the inconsistencies in the dataset, which might lead to incorrect-generated names. One interesting case is represented

Seq2Seq	Context2Name	JSNice	JSNaughty
70.53%	58.1%	56.0%	47.7%

TABLE 1. Results compared to state-of-the-art approaches

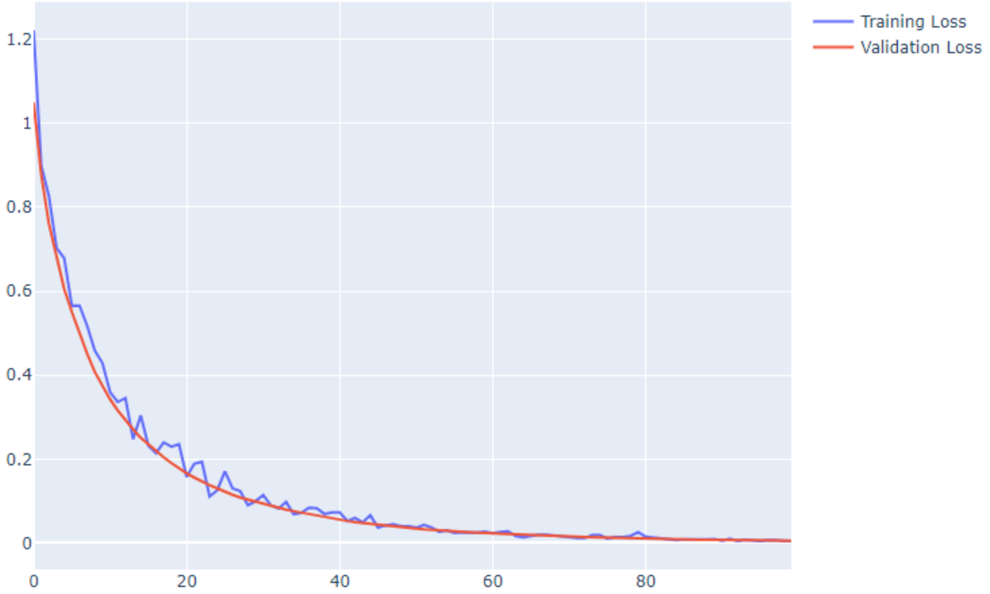

FIGURE 5. Training and validation losses

```
...
var a=new
breeze.config.MetadataHelper;var
n=a.addDataService.bind(a);var
i=a.addTypeToStore.bind(a);var
p=a.setDefaultNamespace.bind(a);
...
```

code resulted using UglifyJS

```
...
var helper = new
breeze.config.MetadataHelper;
  var fn =
helper.addDataService.bind(helper);
  var Helper =
helper.addTypeToStore.bind(helper);
  var addDataService =
helper.setDefaultNamespace.bind(helper);
...
```

code resulted using JSNice

```
...
var helper = new
breeze.config.MetadataHelper();
    var adddataService=
helper.addDataService.bind(helper);
    var addTypeToStore =
helper.addTypeToStore.bind(helper);
    var setDefaultNamespace =
helper.setDefaultNamespace.bind(helper);
...
```

code resulted using our tool

FIGURE 6. Comparison between code generated by UglifyJS, JSNice and the output of our proposed model

by function expressions, which are similar to functions lambda functions assigned with a name, and string literals assignments, where there are fewer and more varied examples to learn from, which might cause the high error rate.

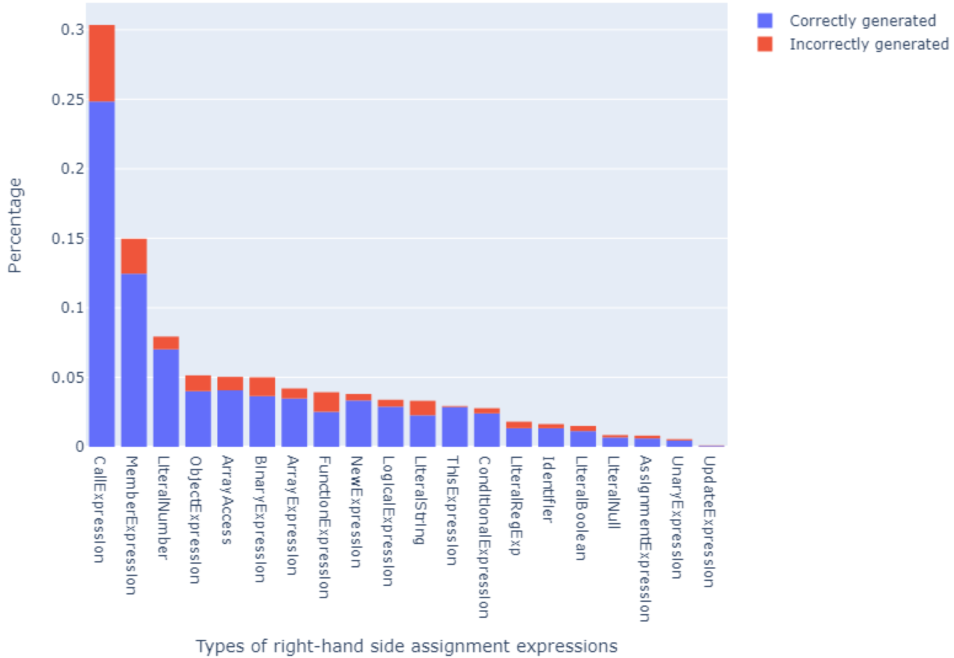

FIGURE 7. Histogram over the variable name types correctly and incorrectly predicted, sorted based on their frequency

The approach introduced in the paper presents both advantages and disadvantages which have to be considered. The main difference between our approach and the previous work in this field is the character-level encoding of non-fixed nodes in the Abstract Syntax Tree, such as the values of strings and function names. This allows us to build an open label vocabulary, which would be much smaller than others and uses strictly the possible types of nodes and the ASCII characters. Thus, the model is able to generate names from input never-before seen, and might generate an output related to that input. Moreover, the code encoding based on Abstract Syntax Tree allows the model to easily learn the relations between nodes and their values as opposed to the traditional token representation of each line of code. As for the disadvantages, as opposed to other methods, this approach cannot handle name collisions: in the previous work presented, name collisions were solved by picking the label with the highest probability, which is not yet existent in the current scope.

Such an approach to name collisions would not provide well-generated labels, but a random sequence of non-intelligible characters.

5. Conclusions and Future Work

This paper addressed the problem of JavaScript code deobfuscation, more generically the problem of assigning names to variables. We proposed a deep-learning generative model, which constructs a fitting name for a variable character-by-character, using their initialization and usages, encoded as Abstract Syntax Trees. After evaluating the model on a data set containing real world JavaScript code, we achieved 70.53% name match accuracy, outperforming state-of-the-art approaches.

Overall, the code obfuscation is a devious task, while also making the code harder for the user to understand it, and sometimes, making him completely unaware of the code that is running on his machine. Code obfuscation was used for malware to propagate through the internet [8], yet there are still organizations that try to protect their code and intellectual property, whose code should be technically safe. There are many other methods of obfuscation besides renaming variables, such as adding code sequences that, when executed, it will have no effect. This technique is used for generating polymorphic code, usually used in malicious code [7]. These techniques provide no real protection from stealing the intellectual property because a professional developer will eventually understand the code, but the described techniques make the whole process more difficult.

To conclude, these results prove that the names in variables are more than a simple label, and they provide a meaning, and their name's characters are similar to words in a sentence.

As for the future work, the presented approach cannot handle well name collisions, as opposed to any of the previous work presented. This approach could be enhanced by adding the current generated name up to that point, but would also require a data set where variables are annotated with multiple possible names. Thus, picking the character with the second-best probability would be a valid solution to this problem, since the model could generate a name properly by using it in future character generations.

The task of generating variable names, in the presented approach, can be adapted to other programming languages as well, where code obfuscation at the level of variable names is predominant, such as decompiled Java and C++. Thus, the only component which has to be changed would be the one that converts the code into the language-specific Abstract Syntax Tree, which can be an area worth experimenting in. Moreover, the task of variable name generation can be integrated in other code-related Natural Language Processing tasks,

such as code generation, where suggestive names have to be recommended based on already-generated logic, or into a code linter, which suggests meaningful variable names based on pre-defined projects to set a naming standard as company policy.

References

[1] Rohan Bavishi, Michael Pradel, and Koushik Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193*, 2018.

[2] George W. Burruss and Timothy M. Bray. Confidence intervals. In Kimberly Kempf-Leonard, editor, *Encyclopedia of Social Measurement*, pages 455–462. Elsevier, New York, 2005.

[3] Raymond PL Buse and Westley R Weimer. Learning a metric for code readability. *IEEE Transactions on software engineering*, 36(4):546–558, 2009.

[4] Tadayoshi Fushiki. Estimation of prediction error by using k-fold cross-validation. *Statistics and Computing*, 21:137–146, 2011.

[5] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[6] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, pages 20–30, 2018.

[7] Xufang Li, Peter KK Loh, and Freddy Tan. Mechanisms of polymorphic and metamorphic viruses. In *2011 European intelligence and security informatics conference*, pages 149–154. IEEE, 2011.

[8] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 47–54. IEEE, 2009.

[9] Xiaoyu Liu, Jinu Jang, Neel Sundaresan, Miltiadis Allamanis, and Alexey Svyatkovskiy. Adaptivepaste: Code adaptation through learning semantics-aware variable usage representations. *arXiv preprint arXiv:2205.11023*, 2022.

[10] Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, 8(67), 2012.

[11] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from" big code". *ACM SIGPLAN Notices*, 50(1):111–124, 2015.

[12] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492*, 2021.

[13] Steve Souders. High-performance web sites. *Communications of the ACM*, 51(12):36–41, 2008.

[14] Sharath K Udupa, Saumya K Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10–pp. IEEE, 2005.

[15] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 683–693, 2017.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[17] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

Babeș-Bolyai University, Faculty of Mathematics and Computer Science, 1 Mihail Kogălniceanu, Cluj-Napoca 400084, Romania

Email address: alexandru.gabriel.sirbu@stud.ubbcluj.ro