JavaScript Programs Obfuscation Detection Method that Uses Artificial Neural Network with Attention Mechanism

Grigory Ponomarenko, Petr Klyucharev
Information Security Department
Bauman Moscow State Technical University
Moscow, Russia
gs.ponomarenko@yandex.ru; pk.iu8@yandex.ru

***Abstract*—In this paper, we consider JavaScript code obfuscation detection using artificial neural network with attention mechanism as classifier algorithm. Obfuscation is widely used by malware writers that want to obscure malicious intentions, e.g. exploit kits, and also it is a common component of intellectual property protection systems. Non-obfuscated JavaScript code samples were obtained from software repository service Github.com. Obfuscated JavaScript code samples were created by obfuscators found on the same service. Before being fed to the network, each JavaScript program is converted to the general path-based representation, i.e. each program is described by the set of paths in an abstract syntax tree. Model proposed in this paper is a feedforward artificial neural network with attention mechanism. We aimed to build a model that relies on AST paths structures instead of statistical features. According to results of experiments, evaluated model potentially can be implemented with some improvements in malicious code detection systems, browser or mobile device fingerprint collection systems etc.**

***Keywords*—obfuscation classification, obfuscated code, obfuscation recognition, Javascript obfuscation, general path-based representation, ECMAScripit obfuscation, AST-based pattern recognition**

I. INTRODUCTION

According to Varnovsky et al. statements [1], obfuscation was firstly implicitly mentioned in 1976 in the famous Diffie and Hellman paper [2], in which they introduced asymmetric cryptography concept. Diffie and Hellman suggested inserting a secret key into the encryption program, and then this secret key initialized encryption program becomes tricky converted so that the secret key extraction would be a very difficult task. The concept of obfuscation was explicitly introduced in 1997 in the Kollberg, Tomborson and Lowe paper [3].

By Han Liu et al. obfuscation is defined as special program transformation whose purpose is to obscure source code or binary code in order to hide implemented algorithms and data structures from being recovered [4]. Obfuscated program is obtained from original after applying obfuscation, and therefore original program is called non-obfuscated [5, 6].

Schrittwieser et al. remark, that at the beginning of the computer era obfuscation was commonly used, in particular, to surprise users by displaying unexpected messages, but today obfuscation is mostly used to protect intellectual property or obscure malicious intentions [7]. Boaz Barak notices, that obfuscation doesn't make protected program invincible, and obfuscated program should be protected from reverse engineering as much as an encryption system shouldn't be broken using any sensible amount of time and computation resources [8].

II. PROBLEM DEFINITION

Obfuscated and non-obfuscated programs distinguishing problem is indelibly linked to the source code properties prediction and various programs classification types. To formalize the problem, we will use the definitions introduced by Silveo Cesar and Yang Xiang in the first chapter of their book "Classification and similarity of programs" [9].

Let r be a property for program p if for all possible execution flaws r is true. A program q is called an obfuscated copy of a program p if q is the result of transformations that preserve the semantics (meaning) of algorithms and data structures. Programs p and q are similar if they are based on the same program.

Let P be the set of source codes of programs, f_1,..., f_k are functions that allocate features from program, i.e. f_i: $P \rightarrow D_i$, where D_i is the i-th set of features. Let $\{p_1, \dots, p_n\} \subset P$ be the training sample, $\{0,1\} = Y$ - class labels (1 is assigned to obfuscated programs, 0 is assigned to non-obfuscated). It is necessary to find the map s: $D_1 \times \dots \times D_k \rightarrow \{0,1\}$ using training sample $\{p_1, \dots, p_n\}$ that classifies all elements of P with the smallest error function value.

III. DATASET PREPARATION

To create a dataset with obfuscated and non-obfuscated Javascript code samples we used software repository github.com. Github.com is one of the largest service platforms that features software projects hosting and collaborative

development. There were downloaded 100 most popular JavaScript projects. To get a list of the most popular projects, we used a special search API (referenced as Github Search API) provided by the service. Further all projects from the resulting list were cloned to the local machine. Downloading was done on March 22th, 2019 and all downloaded projects took up 7.3 Gb of the disk space. 49612 files with the ".js" extension (excluding files with the ".min.js" extension) were retrieved from the obtained data. In order to simplify the further creation of obfuscated code samples, it was decided to retrieve functions from scripts. An example of a simple JavaScript function is shown in fig. 1.

Node.js script that retrieves functions from a Javascript program was written using the Esprima library. With this library abstract syntax tree (AST) can be formed for any Javascript program that complies with the ECMAScript 2016 standard. An abstract syntax tree for a script is formed according to the syntactic rules of the programming language. You can apply the inverse transformation and generate the correct program code from the tree. Unlike to plain source code, ASTs do not include punctuation, delimiters, comments and some other details, but they can be used to describe the syntactic structure of the script along with lexical information [10]. Abstract syntax tree example based on the simple JavaScript program (fig. 1) is shown in fig. 2.

There were built ASTs for all previously downloaded scripts with the extension ".js" (49612 samples) using the parseModule method provided by the Esprima API. Program code for each element of the "FunctionDeclaration" was saved into separate files during the tree traversal. Hereby 126276 files were produced, each file contained a JavaScript function, all files took up 527 Mb of the disk space.

To generate obfuscated code samples, we used special programs that implement JavaScript code obfuscation. On the mentioned above github.com software repository hosting we found 6 obfuscators that fit our needs. They are listed below:

- javascript-obfuscator/javascript-obfuscator
- zswang/jfogs
- anseki/gnirts
- mishoo/UglifyJS2
- alexhorn/defendjs
- wearefractal/node-obf

Obfuscators can work in different ways. Some obfuscators do not significantly change the syntactic structure of the program, e.g. jfogs and UglifyJS2, and mostly rename some identifier and shuffle independent parts. Other obfuscators, such as gnirts or defendjs, completely change the syntactic structure of the scripts.

```
function helloWorldFunction(cityName) {
  var hellWorldString = "Hello, World, from ";
  hellWorldString = hellWorldString + cityName;

  console.log(hellWorldString);

  return hellWorldString;
}
```

Fig. 1. Basic JavaScripit function example

PigeonJS library was used to extract features from the JavaScript programs source codes [11]. It provides an API to get a list of given length paths on the AST.

One path on the AST formed by PingeonJS has following structure called general path-based representation [11]:

$$(vertex_1 \wedge vertex_2 \dots vertex_{k-1} \vee vertex_k) \quad (1)$$

The vertices are separated by v and ^ depending on whether the left vertex is higher or lower on the tree in comparison with right one. One of the paths retrieved from the basic JavaScript program example (fig. 1) is shown in fig. 3.

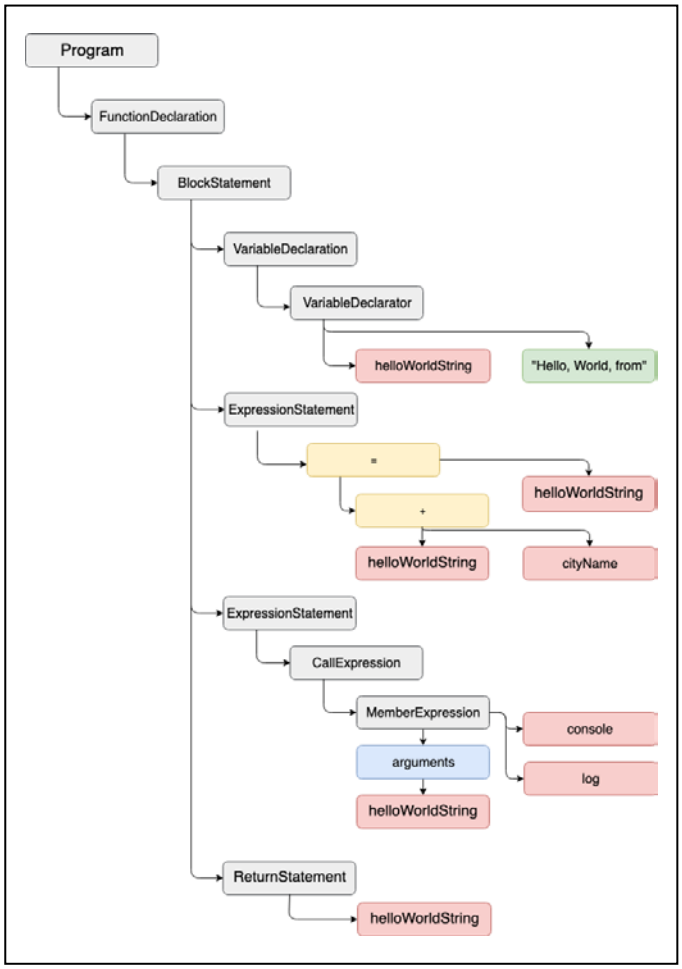

Fig. 2. AST of the basic JavaScript function

```
helloWorldString ∧ VariableDeclarator ∧ VariableDeclaration ∧
∧ BlockStatement ∨ ExpressionStatement ∨ AssignOperation(=) ∨
∨ BinaryOperation(+) ∨ cityName
```

Fig. 3. One of the paths extracted from the basic JavaScript function AST

Not all scripts have been obfuscated with all of obfuscators listed before and contexts were retrieved not from all programs. Main purpose for this was that some of the downloaded JavaScript files contained programs that have nonstandard features and extensions. Besides, some obfuscated scripts took up to 1Gb file storage space although original scripts had size up to 200-300 Kb. We decided to take such samples away from the dataset.

IV. Neural network architecture

The architecture of an artificial neural network was used in this study is based on the network, proposed by Uri Alon et al. in their paper "code2vec: Learning Distributed Representations of Code" [12]. Researchers attempted to create an artificial neural network that predicts method names for programs written in Java. They got excellent results: at the time of the article publication, they had the best percentage of correctly named methods among all known studies — about 60%. So we decided to adapt that network for JavaScript code obfuscation recognition problem solving.

Main objects the network is working on are script contexts. The context $s_i = (x_s, p, x_t)$ is a tuple containing three elements: the start vertex, the path, and the terminal vertex. Start vertex x_s and terminal vertex x_t are elements of the start and terminal vertices set T. Path p is an element of the paths set P. Every JavaScript program (it does not matter, is it obfuscated or non-obfuscated) is described with a set of contexts:

$$s = \{(x_s, p, x_t) \mid x_s, x_t \in T,\ p \in P\} \tag{2}$$

Each element x of the vertices set T has its own vector representation v_x in $Vect_T$ (128-dimensional rational vector). Similarly, each element p of the paths set P has its own vector representation v_p in $Vect_P$ (128-dimensional rational vector). In that way each context has 384-dimensional vector representation that looks like this:

$$c_i = (v_{xt}, v_p, v_{xt}) \in Vect_T \times Vect_P \times Vect_T \subset \mathbb{R}^{128*3} \tag{3}$$

Then each script is described by a tuple of contexts vector representations:

$$c = \langle (v_{xs}, v_p, v_{xt}) \mid v_{xs}, v_{xt} \in Vect_T,\ v_p \in Vect_P \rangle \tag{4}$$

Maximum number of contexts per script was 200. If there were less then 200 contexts for some script then contexts tuple was padded with zero-filled contexts:

$$v_{c0} = (0^{128},\ 0^{128},\ 0^{128}) \tag{5}$$

Artificial neural network architecture used in this research is shown in fig. 4. First of all, there is fully connected layer to which Dropout regularization method was applied. Thanks to this 75% randomly chosen neurons are ignored (not considered during forward pass) on each epoch. This helps to prevent over-fitting of training data and increases model performance on non-observed samples.

Fully connected layer have tanh activation function:

$$d_i = tanh(W * c_i) \tag{6}$$

where $c_i \in \mathbb{R}^{384}, i = \overline{1, 200}$ is a context vector representation, $d_i \in \mathbb{R}^{128}, i = \overline{1, 200}$ is a combined context vector representation

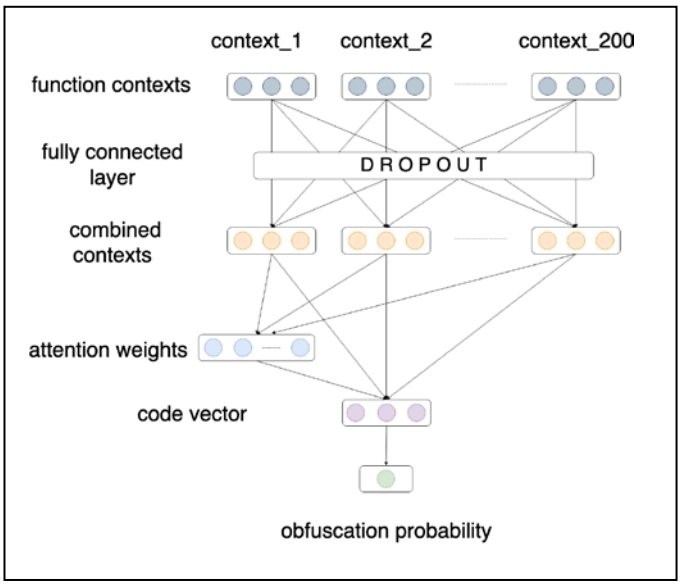

Fig. 4. Neural network ahitecture scheme

and $W \in \mathbb{R}^{128*384}$ is a fully connected layer weights matrix.

Based on the combined contexts $\{d_1, \ldots, d_{200}\}$ and the attention vector α, the attention weights α_i are calculated for each d_i. Vector α is initialized with random variables and updated during the training.

$$\alpha_i = \frac{exp(d_i^T * \boldsymbol{\alpha})}{\sum_{j=1}^{200} exp(d_j^T * \boldsymbol{\alpha})} \tag{7}$$

Obviously, the sum of all α_i equals 1. After that a code vector v is calculated using the attention weights α_i as follows:

$$v = \sum_{i=1}^{200} \alpha_i * \widetilde{c_i} \tag{8}$$

Since all attention weights α_i are nonnegative, and their sum equals to 1, we can consider the calculation of the code vector as the calculation of the weighted average over all combined contexts d_i.

The idea behind the attention mechanism can be described as choosing the most interesting part of the resulting set. The softmax transformation (7) is a key component of several statistical learning models but recently it has also been used to design attention mechanisms in neural networks [13]. Attention mechanisms are used to solve various applied problems with the help of artificial neural networks, e.g. multilingual translation [14], sentiment classification [15], time-series classification [16], vehicle images classification [17] or speech recognition [18].

At the last step the final solution is calculated using 128-dimensional real vectors y_{obf} and y_{notobf}: obfuscated or non-obfuscated script was passed to the network input. Vectors y_{obf} and y_{notobf} are initialized randomly and updated during model training. JavaScript program obfuscation probability $q(v)$ is calculated based on code vector (7):

$$q(v) = \frac{exp(v^T * y_{obf})}{exp(v^T * y_{obf}) + exp(v^T * y_{notobf})} \quad (9)$$

If $q(v) > 0.5$ then script is thought to be obfuscated. Script non-obfuscation probability is estimated as $1-q(v)$ respectively. For one script, the loss function (cross-entropy function) is computed as follows:

$$H(p,q) = -p(v) * log(q(v)) - (1 - p(v)) * log(1 - q(v)) \quad (10)$$

where $p(v) = 1$ for obfuscated scripts and $p(v) = 0$ for non-obfuscated scripts. To minimize the loss function, the method of adaptive moment estimation (Adam) was used as an optimization algorithm.

V. Model Trainig and Evaluation

Model training and evaluation were proceeded on the workstation with the following equipment: Intel Core i7-7700 processor (3.6 GHz) with 8 cores, 16 GB of RAM, NVIDIA GeForce GTX 1080 GPU. The training dataset was formed as follows: 115504 context samples describing non-obfuscated functions and 117990 context samples describing obfuscated functions, among them 36000 randomly chosen from all samples obfuscated with "javascript-obfuscator" , 36000 randomly chosen from all samples obfuscated with "jfogs", 36000 randomly chosen from all samples obfuscated with "UglifyJS2" and 9990 – from samples obfuscated with "defendjs". There were 233494 context samples in sum.

A set T_p ($|T_p|$=776830) of the most popular names of start and final context vertices and a set P_p ($|P_p|$=1008102) were obtained from the training sample so that for each script s at least one context c contains two elements from T and one element from P.

Testing dataset contained 8444 contexts describing obfuscated functions (7655 samples obfuscated with "gnirts" and 789 samples obfuscated with "jfogs") and 8444 contexts describing non-obfuscated functions.

We decided to use precision (10), recall (11) and F1-score (12) as model evaluation metrics explaining model performance.

$$Precision = \frac{tp}{tp + fp} \quad (11)$$

$$Recall = \frac{tp}{tp + fn} \quad (12)$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \quad (13)$$

Model training was 9 epochs long. Precision, recall and F1-score obtained after model training completion are shown in Table 1.

TABLE I. Model scores

Metric	Value
Precision	84.9%
Recall	85.1%
F1	85.0%

Our model showed less well performance than the model proposed by Tellenbach et. al. Their model used features reflecting the frequencies of JavaScript keywords and other statistical statistical calculations and had following evaluations: precision – 95%, recall – 90%, F1-score – 92% [19].

At the same time the presented model has sufficient improvement potential gives rise to further research of obfuscation detection models that do not rely on pre-calculated statistical features. First of all, second fully connected layer and activation function replacement with different one could positively impact model quality scores.

Beyond that, code vector v (7) can be passed to additional classifier input, e.g. SVM-based or Random Forest based. A similar approach Ndichu et al. proposed to solve the JavaScript malware detection problem using feedforward neural network [20]. They divided model training process into two stages: on the first one they trained neural network classifier based on Doc2vec and on the second one they passed fully connected layer output to the SMV. As a result, SVM was trained on the code embeddings [20]. Their model that combines Doc2vec and SMV had following evaluation results: precision – 94%, recall – 92% and F1-score - 93% on the obfuscated samples.

VI. Conclusion

In this paper we explored JavaScript (ECMAScript 2016) code obfuscation detection method that uses artificial neural network with attention mechanism as classifier algorithm.

First of all, a set of samples of obfuscated and non-obfuscated code was obtained using projects and repositories hosted on github.com. Secondly, an artificial neural network model with an attention mechanism was adapted to solve the problem of scripts classification on the obfuscation basis. Thirdly, non-obfuscated dataset could be checked for the presence of obfuscated samples uploaded to github.com repository, downloaded during the dataset preparation stage and erroneously labeled as non-obfuscated.

The characteristics of the obtained model show that the considered method potentially can be implemented with some improvements in malicious code detection systems, browser or mobile device fingerprint collection systems or other software that use obfuscation recognition.

REFERENCES

[1] N.P. Varnovsky, V.A.Zakharov, N.N. Kuzurin, V.A. Shokurov. The current state of art in program obfuscations:definitions of obfuscation security. Proceedings of the Institute for System Programming, vol. 26, issue 3, 2014, pp. 167-198. DOI: 10.15514/ISPRAS-2014-26(3)-9.

[2] Diffie W., Hellman M. New directions in cryptography // IEEE Transactions on Information Theory, IT-22(6), 1976, p.644-654.

[3] Collberg C., Thomborson C., Low D. A Taxonomy of Obfuscating Transformations // Technical Report, N 148, Univ. of Auckland, 1997.

[4] Liu, H., Sun, C., Su, Z., Jiang, Y., Gu, M. and Sun, J., 2017, May. Stochastic optimization of program obfuscation. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) (pp. 221-231). IEEE.

[5] Kozachok A., Bochkov M., Tuan L.M. Indistinguishable Obfuscation Security Theoretical Proof. Voprosy kiberbezopasnosti [Cybersecurity issues], 2016. N 1 (14). P. 36-46.

[6] Markin D., Makeev S. Protection System of Terminal Programs Against Analysis Based on Code Virtualization. Voprosy kiberbezopasnosti [Cybersecurity issues], 2020, N 1 (35), pp. 29-41. DOI: 10.21681/2311-3456-2020-01-29-41.

[7] Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., & Weippl, E. (2016). Protecting Software through Obfuscation. ACM Computing Surveys, 49(1), 1–37.

[8] Barak, B. (2016). Hopes, fears, and software obfuscation. Commun. ACM, 59(3), 88-96.

[9] Silvio Cesare, Yang Xiang. Software Similarity and Classification. Springer-Verlag, 2012.

[10] Zhang, Jian, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. "A novel neural source code representation based on abstract syntax tree." In Proceedings of the 41st International Conference on Software Engineering, pp. 783-794. IEEE Press, 2019.

[11] Alon, Uri, Meital Zilberstein, Omer Levy, Eran Yahav. A general path-based representation for predicting program properties. ACM SIGPLAN Notices, vol. 53, no. 4, pp. 404-419. ACM, 2018.

[12] Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav. Code2vec: Learning Distributed Representationsof Code. Proc. ACM Program. Lang.3, POPL, 2019, 40, P. 1-29

[13] Martins, Andre, and Ramon Astudillo. "From softmax to sparsemax: A sparse model of attention and multi-label classification." In International Conference on Machine Learning, pp. 1614-1623. 2016.

[14] Firat, Orhan, Kyunghyun Cho, and Yoshua Bengio. "Multi-way, multilingual neural machine translation with a shared attention mechanism." In 15th Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL HLT 2016, pp. 866-875. Association for Computational Linguistics (ACL), 2016.

[15] Wang, Yequan, Minlie Huang, and Li Zhao. "Attention-based LSTM for aspect-level sentiment classification." In Proceedings of the 2016 conference on empirical methods in natural language processing, pp. 606-615. 2016.

[16] Du, Qianjin, Weixi Gu, Lin Zhang, and Shao-Lun Huang. "Attention-based LSTM-CNNs For Time-series Classification." In Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems, pp. 410-411. ACM, 2018.

[17] Zhao, D., Chen, Y., & Lv, L. (2017). Deep Reinforcement Learning With Visual Attention for Vehicle Classification. IEEE Transactions on Cognitive and Developmental Systems, 9(4), 356–367.

[18] Kim, Suyoun, Takaaki Hori, and Shinji Watanabe. "Joint CTC-attention based end-to-end speech recognition using multi-task learning." In 2017 IEEE international conference on acoustics, speech and signal processing (ICASSP), pp. 4835-4839. IEEE, 2017.

[19] Tellenbach B, Paganoni S, Rennhard M. Detecting obfuscated JavaScripts from known and unknown obfuscators using machine learning. International Journal on Advances in Security. 2016;9(3/4):196-206.

[20] Ndichu, S., Kim, S., Ozawa, S., Misu, T. and Makishima, K., 2019. A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors. Applied Soft Computing, 84, p.105721.