A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors

Applied Soft Computing Journal 84 (2019) 105721

Contents lists available at ScienceDirect

Applied Soft Computing Journal

journal homepage: www.elsevier.com/locate/asoc

A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors

Samuel Ndichu [a,*], Sangwook Kim [a], Seiichi Ozawa [b], Takeshi Misu [c], Kazuo Makishima [c]

[a] *Graduate School of Engineering, Kobe University, Japan*
[b] *Center for Mathematical and Data Sciences, Kobe University, Japan*
[c] *SecureBrain, Co., Japan*

ARTICLE INFO

Article history:
Received 13 February 2019
Received in revised form 30 July 2019
Accepted 16 August 2019
Available online 22 August 2019

Keywords:
Cybersecurity
Machine learning
Doc2Vec
Malicious JavaScript detection
Feature learning
Abstract Syntax Tree

ABSTRACT

Websites attract millions of visitors due to the convenience of services they offer, which provide for interesting targets for cyber attackers. Most of these websites use JavaScript (JS) to create dynamic content. The exploitation of vulnerabilities in servers, plugins, and other third-party systems enables the insertion of malicious codes into websites. These exploits use methods such as drive-by-downloads, pop up ads, and phishing attacks on news, porn, piracy, torrent or free software websites, among others. Many of the recent cyber-attacks exploit JS vulnerabilities, in some cases employing obfuscation to hide their maliciousness and evade detection. It is, therefore, primal to develop an accurate detection system for malicious JS to protect users from such attacks. This study adopts Abstract Syntax Tree (AST) for code structure representation and a machine learning approach to conduct feature learning called Doc2vec to address this issue. Doc2vec is a neural network model that can learn context information of texts with variable length. This model is a well-suited feature learning method for JS codes, which consist of text content ranging among single line sentences, paragraphs, and full-length documents. Besides, features learned with Doc2Vec are of low dimensions which ensure faster detections. A classifier model judges the maliciousness of a JS code using the learned features. The performance of this approach is evaluated using the D3M dataset (Drive-by-Download Data by Marionette) for malicious JS codes and the JSUNPACK plus Alexa top 100 websites datasets for benign JS codes. We then compare the performance of Doc2Vec on plain JS codes (Plain-JS) and AST form of JS codes (AST-JS) to other feature learning methods. Our experimental results show that the proposed AST features and Doc2Vec for feature learning provide better accuracy and fast classification in malicious JS codes detection compared to conventional approaches and can flag malicious JS codes previously identified as hard-to-detect.

1. Introduction

Websites and web applications exist widely because of the myriad services and convenience they offer. Adoption of these technologies across various devices and browsers is consequently inevitable. However, even with their practical applications, the rapid development of these technologies coupled with the availability of tools and technologies to exploit vulnerabilities in servers, plugins and other third-party systems has made them interesting targets for cybercriminals. These exploits result in the proliferation of various types of attacks such as drive-by-downloads, pop-up ads, and phishing attacks common with cybercriminals, who use malicious code to perform these attacks for financial gains, social engineering or extortion [1]. JavaScript (JS) is a dynamic and lightweight scripting language used to add more functionality and enhance user experience. It is a ubiquitous technology that allows client-side script to interact with a user and makes websites dynamic and interactive [2], leading to less server–client interaction, immediate feedback, and rich interfaces. Despite its various advantages, JS has been used to perform cyber-attacks. There exist various types of JS-based attacks recently:

- Drive-by-Download attacks which refer to automatically downloading malware on a computer stealthily, without the user's awareness. Malware downloads and automatically installs itself. Infected websites may contain exploit kits (this is a toolkit used to attack vulnerabilities and distribute malware in compromised websites). If any weakness occurs, malware is downloaded to exploit the vulnerability without the user's intention.

* Corresponding author.
E-mail address: sammiendichu01@stu.kobe-u.ac.jp (S. Ndichu).

https://doi.org/10.1016/j.asoc.2019.105721

- Cross-site scripting (XSS) attack is common in web applications without proper user input validation and sanitization, for example, form data and comments. It allows the injection of malicious scripts into a website. Insertion of malicious links into a legitimate website is one of the ways for the attack perpetration. Stealing of user sensitive information such as passwords, cookies, and accounts, is common when a user visits such a website.
- Cross-Site Request Forgery (XSRF) attack is where a user authenticated to a web application is tricked into performing unwarranted actions. This kind of malicious JS code can exploit cookies, browser, and security permissions to perform actions on another website.
- Heap spraying attacks constitute writing of a sequence of bytes at a predetermined process memory location which is then exploited to execute arbitrary malicious code.
- Hidden iframes load JS malware from compromised websites, resulting in stealthily loading of exploits.
- Malvertising or malicious advertising takes advantage of the pervasive and rife nature of advertisements, which are billions on the Internet. Most of these do not adhere to ethical standards and are illegitimate. Google took down close to two billion advertisements which violated its policies in 2016.

When browsing a website, JS files are downloaded in the client-side and executed via a browser. There are differences in possible damage and method for these attacks. However, these attacks generally use JS. For example, XSS vulnerabilities facilitating Drive-by-Download when users open URLs. Malicious JS codes with these various attack types will have the same code properties at an abstract level. These codes are, therefore, detectable with the proposed method. However, due to the limitations of the current dataset, we only verify the detection of Drive-by-Download attacks. Malicious JS codes are difficult to detect [3] because they sneakily exploit vulnerabilities of browsers, plugins, and other third-party applications and attack visitors of a website unknowingly. Therefore, it is important to accurately detect malicious JS codes to protect users from such attacks proactively.

Malicious JS codes perform, among others, keystroke recording, browser cookies theft, hacking, website defacement, and Trojan horses [4]. Besides, it is possible to create a botnet by tricking users into downloading malware by social engineering [5]. Conventional security solutions such as antivirus and Intrusion Detection Systems (IDS) employ signature and heuristic-based approaches to detect attacks. Heuristic-based methods such as file analysis, file emulation, and generic signature detection [1] use a set of expert decision rules. These can detect previously unknown and variants of malware. However, these approaches have many shortcomings. First, knowledge of an emerging attack is essential to update the program for effective detection, which creates a vulnerability window that attackers can leverage to launch zero-day and other types of attacks. Second, they suffer from a slow detection rate due to lengthy analysis and scanning time. Lastly, they are prone to suffer from large numbers of false negatives, which is a drawback to performance. Other conventional approaches for detecting malicious JS are the use of pattern matching [6], low and high interaction client honeypots and maintenance of a URL blacklist [7,8]. Recently, many intelligent intrusion detection systems have been proposed, which mainly focus on anomaly detection.

Recently, application fields of machine learning have been extending enormously including applications such as text classification and clustering [9–17] where a machine learning model carries out tasks such as document retrieval, sentiment analysis, spam filtering, and web search. Text documents are used to train a machine learning model, which in turn is used to predict document classes or categories. A machine learning model learns from data and can give an accurate prediction for a given input after correctly conducting learning. Usually, raw data need to be transformed into a proper feature representation for inputs to attain high-performance prediction. One such representation is a fixed-length vector for text documents. These models have achieved state-of-the-art results for text classification and clustering tasks on various datasets. As such, since a JS code also contains textual data, it is envisaged that machine learning models can classify JS-based attacks.

This paper proposes the use of the AST form of JS code (AST-JS) as a code structure representation, a feature learning approach to malicious JS contents using Doc2Vec and a classifier for AST features classification. This approach provides considerable improvement in performance compared to our previous work with the capabilities of detecting JS previously identified as hard-to-detect. To enhance feature learning, we perform two types of AST-JS manipulations: AST-level merging and AST-subtree realignment. Our approach automatically learns feature vectors compared to other previous methods which employ manually crafted feature sets and subsets. Besides, features learned are of low dimensions hence ensuring a faster detection. The organization for the rest of this paper is as follows. Section 2 highlights the related work. We present our approach in Section 3. Section 4 carries out the performance evaluations through a comparison to some feature learning methods, and finally, give our conclusion in Section 5.

2. Related work

Cyber-attacks that employ malicious JS codes are ever-increasing and conventional security approaches such as signature, heuristic-based approaches, and pattern matching [6] have shortcomings in terms of zero day's attacks, leading to many false negatives coupled with slow detections. It is therefore important to accurately and timely detect these codes to curb this trend. Several other approaches for detecting malicious JS such as using low and high interaction client honeypots and maintenance of a URL blacklist [7,8] exist. However, these are prone to suffer from the risk of infiltration, which is a shortcoming for honeypot implementations, and new URLs easily circumvent the blacklist.

Approaches that use machine learning to detect malicious JS codes have been proposed, such as monitoring the execution of a JS code at run time using a sequence of events to obtain vectors for classification [3]. Another approach is to learn and detect malicious patterns in the structure and behavior of a JS code [18]. In another example, [19] has proposed a static detection approach to extract and analyze JS features, where the assumption is that the number of some special functions is limited in benign cases and vice versa. [20] has also proposed the use of features extracted from a JS code using a three-layer Stacked denoising Autoencoders (SdA), which is a typical neural network and pattern classification. They classify binary features obtained from a JS code. However, when compared to other models such as ADTree and RBF SVM, SdA had the second-best false positive rate. Besides, they use sparse random projection for dimensionality reduction resulting in a set of 480 features, which leads to long training time as compared to other methods.

Another approach is to use a wrapper method and a classifier for feature clustering [1] which generates a feature subset through feature selection. The technique employs a limited set of features, and the wrapper method is prone to suffer from overfitting and long processing time. Misuse (signature-based detection) and anomaly detection to detect malicious websites by integrating both supervised and unsupervised learning have also been proposed [21]. The assumption is that the misuse detection method can identify well-known attack types, but it cannot detect new ones. They use this method to detect malicious websites using the C4.5 decision tree algorithm. They then use anomaly detection to counter the weakness of the misuse detection method. For this, they use SVM to identify anomalies in the websites determined as benign by the misuse detection method. This approach achieved a high detection rate. However, it is also prone to suffer from a high false-negative rate. [22] combines dynamic and static analysis to automatically classify a malicious JS and achieve both scalability and high true positive rate. They employ pattern analysis and build a malicious JS attack model using only JS function calls and textual contents. [23] infer behavior model or typical behavior of malicious JS by active automata learning using Deterministic Finite Automaton (DFA). To identify the malicious traces of the same attack types, they use a JS replay mechanism to analyze data dependency and defense rules.

These conventional approaches have achieved a certain degree of accuracy in the detection of malicious JS contents to some extent. However, they have several limitations; these approaches identify specific attack types using constructed feature sets, subsets, and functions such as $navigator.userAgent.toLowerCase()$, $unescape()$, $eval()$, $setTimeout()$, etc. for detection of malicious JS contents. These would only go as far as the pre-defined features, any variant beyond the defined set of features could result in a false negative. These limitations explain the reason why most of these methods are prone to suffer from a high false-negative rate. Another drawback to these conventional approaches is that they generate high-dimensional representations which translate to slow detection rate. These approaches are also known to lose the order of words in the resulting representations. Previously, we proposed the use of Doc2Vec for plain JS code (Plain-JS) feature learning [24]. This approach automatically learns fixed-length vector representation for a variable length of plain-JS contents with better accuracy and fast detection rate compared to other feature learning methods. However, when performing feature learning for a JS code that contains elements other than JS code words, such as an obfuscated JS code, the model is prone to suffer from false-positive and false-negative cases because an obfuscated JS content does not accurately represent a JS code structure. Even though these misclassified JS cases are a few in paragraph vector models compared to other models, there is a need to further improve the performance for better detection of malicious JS codes. We, therefore, propose the use of AST-JS for code structure representation and Doc2Vec for feature learning of AST-JS. This approach is expected to provide even better accuracy compared to our previous work [24] as AST-JS gives a robust property against some perturbation in codes.

3. Proposed method

Development objectives of benign JS codes differ from those of malicious ones attempting to trick or deceive users using soliciting words [25] such as ‘free,’ ‘money,’ and ‘win.’ This characteristic makes malicious JS codes different from benign ones. In such JS codes, Table 1 shows a list of frequently used function names, function types, possible threats, and intention in malicious JS codes. These have been selected from [1,22]. Learning such specific words on JS would help to enhance the reliability of identifying the maliciousness [24]. In our previous work, a plain form of JS codes is provided to Doc2Vec as inputs and use a small dataset of 80 JS codes for feature learning. However, a JS code contains more than information on its text contents (e.g., function or variable, operational procedures, and stop condition). Therefore, to effectively learn JS code contents, it is also important to take the structure of a JS code into consideration. We extend our previous work [24] in this paper so that essential information in a JS code can be considered to identify malicious intention embedded in JS codes. For this task, we propose the use of Abstract Syntax Tree (AST) [26–28] for code structure representation and Doc2Vec for feature learning. To this end, we perform AST-level tweaks to enhance feature learning for accurate malicious JS detection.

Table 1
A list of frequently used function names in malicious JS codes, function types, possible threats, and intention. This list provides evidence that malicious and benign JS codes contain different frequent words as malicious JS codes would be expected to utilize such function names compared to benign ones heavily.

Function name	Function type	Possible threats (Intension)
location.replace()	Change current URL	Redirect to malicious URL
document.write()	DOM operation	Embed malicious script
getUserAgent()	Check browser	Target specific browser
String.split()	String operation	Hide intent-encode/encrypt
setCookie()	Cookie access	Manipulate Cookie

Let us give a brief explanation of the preprocessing for transforming a JS code into an AST structure (AST-JS) and the Doc2Vec feature learning we used in the following sections.

3.1. AST form of JavaScript code

AST, also called syntax tree, is a simplified parse tree or Concrete Syntax Tree (CST). AST represents the necessary and functional structure of code such as scopes, expressions or declarations while at the same time avoiding redundancy by omitting unnecessary syntactic details [26,27] such as whitespaces, punctuation marks or comments which do not affect the original function of code even after the parsing process. Many of the recent cyber-attacks exploit JS vulnerabilities, in some cases employing obfuscation to hide their maliciousness and evade detection. AST's have been used to accurately represent such obfuscated JS codes [29–32].

The compilation process starts with lexical analysis where a compiler performs scanning and tokenization. Then a parser takes this tokenized input and generates AST through syntax analysis [28]. Listing 1 shows an AST generated through syntactic analysis using 5+(1*12) as the source text. The JS code is represented in JS Object Notation (JSON). AST-JS can then be used as input for machine learning algorithms. These algorithms require input to be represented as feature vectors for classification or clustering tasks. This may result in high dimensional representations depending on the size of the dataset which translate to longer detection time [9]. For faster detection, we propose the use of fixed-length vector representation.

```
{'body': [{'expression': {'left': {'raw': None,
'type': 'Literal',
'value': 5.0}
'operator': '+',
'right': {'left': {'raw': None, 'type': 'Literal',
'value': 1.0}
'operator': '*',
'right': {'raw': None, 'type': 'Literal','value':
    12.0},
'type': 'BinaryExpression'},
'type': 'BinaryExpression'},
'type': 'ExpressionStatement'}],
'type': 'Program'}
```

Listing 1: A sample Abstract Syntax Tree (AST) generated using esprima, a syntactical and lexical analysis tool. This is obtained from a binary expression which contains numeric values (5, 1 and 12), operators (+ and *) and parentheses (()). It is worth noting that some details such as the parentheses, which would otherwise make the AST verbose, are omitted in the resultant AST.

3.2. Preprocessing

Preprocessing of a JS code is an important step to enhance the model performance during feature learning. To define the first type of Doc2Vec input, we use regular expression meta-characters $[a-z]+$, to specify the input range which matches any lowercase letter from a to z, remove symbols and special characters such as $*$, =, !, etc. such that only plain-JS text elements remain as shown in Fig. 1 which is a resultant JS code representation obtained from Listing 2. + indicates the preceding element and its one or more occurrences. In this case, it is used to match one or more characters in a regular expression. Feature learning in Doc2Vec requires a large amount of data. For this reason, we add a *json* file with random text data. This text data is necessary for two primary purposes; one acts as noise for performance validation by ensuring that the model can distinguish noise dataset from the JS code contents during classification. Two, it is used when increasing the dataset by random insertion of text data into the JS codes to create new dummy JS code instances for feature learning. This approach is comparable to the one in our previous work [24].

To define the second type of Doc2Vec input, we employ code structure representation using a parser called esprima [33], ECMAScript scripting-language standard, which allows various code analysis and operations such as editing, transformation, visualization, and validation. We use this tool to perform syntactic analysis, that is, parsing the JS codes into AST's. Esprima produces an AST-JS from a JS code string input as shown in Listing 3 which shows the resultant snippets of an AST-JS after parsing of Listing 2. This tool also checks the JS codes for syntactic errors thereby ensuring that the input is a valid JS program, if the input contains syntax errors, it will not be parsed into AST-JS. Finally, the plain-JS and the AST-JS are given to Doc2Vec for feature learning.

```
function d(a) {
var c = typeof a;
if ("object" == c) {
if (a) {
if (a instanceof Array) {
return "array";
...
var b = Object.prototype.toStrin
if ("[object Array]" == b || "number" == ... && !a
    .propertyIsEnumerable("splice")) {
return "array";
}
...
}
} else {
if ("function" == c && "undefined" == typeof a.
    call) {
return "object";
}
}
return c;
}
```

Listing 2: A snippet of a sample original JS code before preprocessing.

```
{
"type": "Program",
"body": [
{
"type": "FunctionDeclaration",
"id": {
"type": "Identifier",
"name": "d",
"range": [
9,
10
]
```

```
TaggedDocument(['function', 'var', 'this', 'function', 'var', 'typeof', 'if', 'object', 'if',
      'if', 'instanceof', 'array',    'return',  'array', 'if',  'instanceof',  'object',
      'return', 'var', 'object', 'prototype', 'tostring', 'call', 'if',     'object',
      'window', 'return', 'object', 'if', 'object',  'array', 'number', 'typeof',
      'length', 'undefined', 'typeof',  'splice',  'undefined',  'typeof',  'splice',
      'return', 'array', 'if', 'object', 'function', 'undefined', 'typeof', 'call',
      'undefined',  'typeof',  'call',  'return',  'function',  'else',  'return',  'null',
      'else', 'if', 'function', 'undefined',    'typeof',   'call',   'return',   'object',
      'return', 'function',     'var', 'array', 'prototype', 'slice', 'call',
      'arguments', 'return', 'function', 'var', 'slice', 'push', 'apply',.............]
```

Fig. 1. A sample of a plain JS code (plain-JS) after preprocessing using a regular expression. This shows the resultant Plain-JS after removal of symbols and special characters. It is used as Doc2Vec input for the first part of the experiments.

```
],
"body": {
"type": "BlockStatement",
"body": [
{
...
"type": "VariableDeclaration",
"declarations": [
{
...
```

Listing 3: Snippets of a sample AST-JS after preprocessing using a parser. This shows the resultant snippets of AST-JS when parsing is applied to a JS code, it is used as Doc2Vec input for the second part of the experiments.

3.3. Feature learning by Doc2Vec

The bag-of-words approach is a common fixed-length feature that represents a sentence or a document as a simplified, unordered collection of words [34,35]. For document classification, classifier training utilizes features such as word occurrence or count. This task capitalizes on a vocabulary of known words and a measure of the presence of known words. Documents with the same words or text are assumed to be similar, and the words can give the meaning of the document. However, bag-of-words has been known to lose context information and semantic structure. Word vectors [11] represent each word with a vector, concatenated with other word vectors, and the resulting vector predicts other words. However, this model has only been extended up to sentence-level representation while relying on parsing [36]. Paragraph vectors [9–12] learns continuous distributed vector representations for text with variable length of input sequences, for example, documents, paragraphs or sentences. This property is important because each JS code generally has a different length, and the sequence of words differentiate its behavior. Paragraph vectors have advantages of retaining a semantic structure of a sentence, consider word order, and create low-dimensional representation. Word vector model [11] predicts a word given other words in a context and vice versa using the continuous bag of words model or the skip-gram model. Paragraph vectors extend word vectors by adding additional input units that represent documents as additional context. Each additional input unit acts as an ID for each input document.

For a sequence of training words $\boldsymbol{X} = \{\boldsymbol{x}_{t-2}, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+2}\}$, the model predicts the center word $\boldsymbol{x}_t$ represented by $\boldsymbol{X_t}$ and given by,

$$\boldsymbol{U}_i = p(\boldsymbol{X_t}|\boldsymbol{X}, \boldsymbol{D_i}) \tag{1}$$

$\boldsymbol{D_i}$ is the paragraph or document vector which represents unique vector mapping for ith paragraph or document. The model has

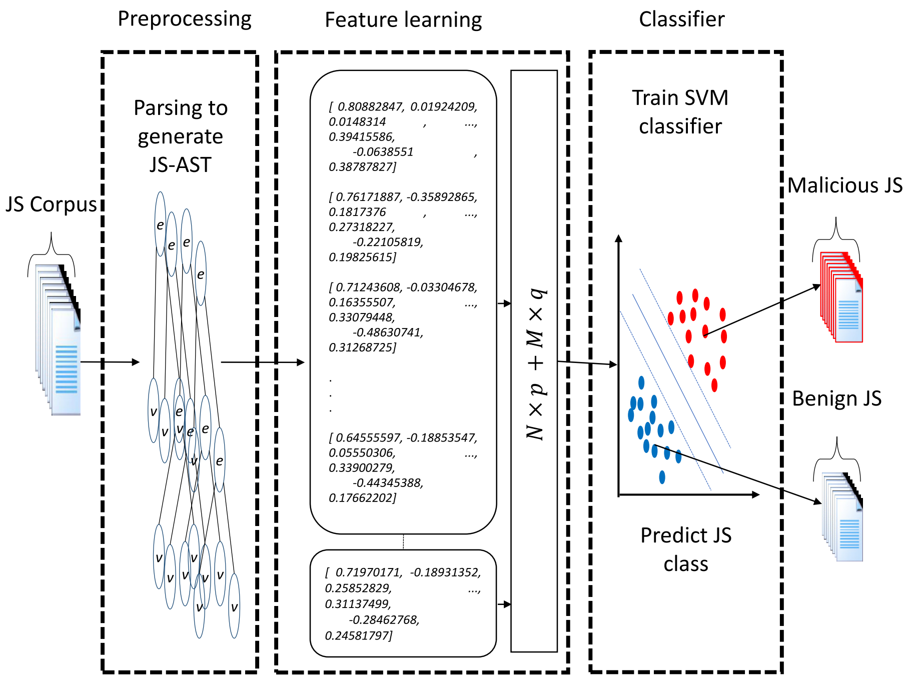


Fig. 2. The layout for learning and classification of malicious JS contents AST features.

three layers. Assume that a word and a document, which is the input for the network is represented by one-hot vectors $\boldsymbol{x_i}$ and $\boldsymbol{d_i}$. Each input layer unit corresponds to $\boldsymbol{x_i}$ and $\boldsymbol{d_i}$ representing the ith word and a document input respectively. All input layer units make the vectors $\boldsymbol{X}$ and $\boldsymbol{D}$ for all M words and N documents in the corpus, used as the model input.

The hidden layer is of size p which represents the paragraph vector dimensionality and chosen during training. Both the input and hidden layers are fully connected with each connection having a weight. Small values are used for random initialization of these weights before training the network. The hidden or embedding layer for this network is very simple for faster computation. A weighted sum of the input layer activation is used as the activation for the hidden layer given by,

$$\hat{\boldsymbol{D}} = \boldsymbol{W}\boldsymbol{D} \tag{2}$$

$\boldsymbol{W}$ is a $p \times N$ matrix with all weights between the input and hidden layer. $\boldsymbol{D}$ contains the concatenation of the word and document vectors $\boldsymbol{X}$ and $\boldsymbol{D_i}$. $\hat{\boldsymbol{D}}$ is the hidden layer activation, this is the paragraph vector and an estimation which represents $\boldsymbol{D}$ in p-dimensions. The paragraph vector $\hat{\boldsymbol{D_n}}$ for document ID n would be given by the nth column in $\hat{\boldsymbol{D}}$.

The output layer corresponds to V words in the vocabulary which gives the number of the output layer units. The output layer is also fully connected to the hidden layer. This represents an estimation of the term or word vector and can be represented as,

$$\boldsymbol{U} = \boldsymbol{W}'\hat{\boldsymbol{D}} \tag{3}$$

$\boldsymbol{W}'$ is a $V \times p$ matrix with all weights between the hidden and output layer. $\boldsymbol{W}'$ has output vectors represented as rows for each word in the vocabulary. A word at index m would be given by $\boldsymbol{W}'_m$. Therefore,

$$\boldsymbol{U_m} = \boldsymbol{W}'_m \cdot \hat{\boldsymbol{D}} \tag{4}$$

After training the network, $\boldsymbol{W}$ will have the word and document embeddings respectively, which will be reused for prediction. When given a document vector, the hidden layer will be used as the output layer giving fixed-size paragraph vectors of p-dimensions. To map $\mathbb{R}$ given by $\boldsymbol{U}$ to a probability distribution, a softmax function is applied,

$$s(\boldsymbol{U_i}) = \frac{e^{\boldsymbol{U_i}}}{\sum_k e^{\boldsymbol{U_k}}} \tag{5}$$

Where $0 < s(\boldsymbol{U_i}) < 1$, $\sum_i s(\boldsymbol{U_i}) = 1$ and $s(\boldsymbol{U_i})$ boosts activation of the units with the highest function where if activation is higher, $s(\boldsymbol{U_i})$ will move towards 1 and 0. The softmax function will result with $\hat{\boldsymbol{t}}$, an approximation of the actual term to be predicted (an estimated probability distribution). During training, $\hat{\boldsymbol{t}}$ is compared to $\boldsymbol{t}$ the target term vector using a loss function which should be low when $\hat{\boldsymbol{t}}$ and $\boldsymbol{t}$ are similar. This can be given by cross entropy as,

$$E(\hat{\boldsymbol{t}}, \boldsymbol{t}) = -\sum_i \boldsymbol{t_i} \log \hat{\boldsymbol{t_i}} \tag{6}$$

Where $E(\hat{\boldsymbol{t}}, \boldsymbol{t}) > 0$, if $\boldsymbol{t_i} = 0$ the ith element of the sum will be 0 and E is low when $\hat{\boldsymbol{t_i}}$ is close to 1. Let y be an index where $\boldsymbol{t_y} = 1$ and $\boldsymbol{t}$ is a one-hot vector, then,

$$E(\hat{\boldsymbol{t}}, \boldsymbol{t}) = -\log \hat{\boldsymbol{t_y}} \tag{7}$$

During training, a set of training predictions are compared to corresponding actual target vectors by minimizing a loss function on a set of training points. With a set of training target vectors $\boldsymbol{T}$ and $\boldsymbol{t_j}$ as the jth target vector, the loss function can be given as,

$$L = \frac{1}{|\boldsymbol{T}|} \sum_{j=1}^{|\boldsymbol{T}|} E(\hat{\boldsymbol{t_j}}, \boldsymbol{t_j}), \tag{8a}$$

$$L = \frac{1}{|\boldsymbol{T}|} \sum_{j=1}^{|\boldsymbol{T}|} -\boldsymbol{W}'_{y_j} \cdot \hat{\boldsymbol{D_j}} + \log \sum_{m=1}^{M} e^{\boldsymbol{W}'_m \cdot \hat{\boldsymbol{D_j}}} \tag{8b}$$

Where E is averaged over a set of training examples. y_j is the index of the actual word for training example j in the target vector $\boldsymbol{t_j}$ and its corresponding output vector is given by $\boldsymbol{W}'_{y_j}$.

Doc2Vec, an implementation [37] of paragraph vectors, is a novel feature learning method for text documents that converts a document into a feature vector. It is an unsupervised algorithm that produces vectors of a fixed-length [9]. We train a small neural network for a prediction task using labels obtained from a JS dataset, and a part of this trained network is reused to give a document or a paragraph vector. A JS code classification task is then performed using these vectors. Doc2Vec uses many hyper-parameters to learn a feature vector. These include *vector_size*, which represent the dimensionality of a feature vector, *min_count*, which ignores all words with a frequency lower than a preset one, and *window*, which is the maximum distance between the current word and the predicted one. There are two approaches to paragraph vectors: Distributed Bag of Words version of Paragraph Vector (PV-DBoW) and Distributed Memory Model of Paragraph Vectors (PV-DM).

Fig. 2 shows the overview of learning and classification of malicious AST-JS features. It consists of three components, which are, preprocessing, feature learning, and classification. During preprocessing, the JS corpus is parsed into AST-JS features by syntax analysis. The AST-JS features are used for feature learning by Doc2Vec. Training of the classifier is performed using the learned fixed-length feature vectors. A new JS code generates an AST-JS through parsing and inference for feature vectors is done from the trained Doc2Vec model vector space. Lastly, the classifier uses the inferred feature vectors for prediction.

We performed cross-validation to find the optimal hyperparameters and classifier learning with a training set of feature vectors. We then compared the performance of Doc2Vec on malicious JS contents detection to other feature learning methods on the same dataset. Algorithm 1 shows the process for the learning of feature vectors from the JS codes in PV-DM, which consists of training on input JS codes for a given corpus. The model learns both word embeddings and document embeddings (lines 4–9), with weights initialized randomly using small values before training the network. These embeddings are of p dimensions (line 8) with consideration for a word frequency higher than *min_count* (line 5). Algorithm 2 shows the process for the learning of feature vectors from the JS codes in PV-DBoW (lines 4–8), the same as in PV-DM. However, this model only learns document embeddings (line 4) while keeping the word embeddings fixed. The updating of learned embeddings for both PV-DM and PV-DBoW through negative sampling (lines 2–11), gradient descent (lines 12–18) and backpropagation (lines 21–24) as shown in algorithm three outputs the feature vectors which are then fed directly to SVM for train and test purposes.

Algorithm 1 PV-DM malicious JS contents detection

```
Input: JS_codes, model parameters W and W′, hyper-parameters
    size, window, min_count.
Output: neu1[s] - Word vectors U, Document vectors D.
    repeat until JS_codes run out.
    // read a document from JS_codes.
    Doc[] = Read_Codes(JS_codes);
    for each tword, tDoc ∈ Doc do
      if tword > min_count then
        // feed forward.
        for each wWord, dDoc ∈ Context(tword, tDoc, window)
        do
          for s = 0 to size do
            neu1[s] = neul[s] + syn0[wWord, dDoc] [s];
          end for
        end for
      end if
    end for
```

Algorithm 2 PV-DBoW malicious JS contents detection

```
Input: JS_codes, model parameters W and W′, hyper-parameters
    size, window.
Output: neu1[s] - Document vectors D.
 1: repeat until JS_codes run out.
 2: // read a document from JS_codes.
 3: Doc[] = Read_Codes(JS_codes);
 4: for each tDoc ∈ Doc do
 5:   // feed forward.
 6:   for each dDoc ∈ Context(tDoc, window) do
 7:     for s = 0 to size do
 8:       neu1[s] = neul[s] + syn0[dDoc] [s];
 9:     end for
10:   end for
11: end for
```

Algorithm 3 Paragraph vectors training

```
Input: neu1[s] - Word vectors U, Document vectors D.
Output: updated syn0 - Feature vectors.
 1: // negative sampling.
 2: for each sample ∈ (nsample || tword) do
 3:   if sample == tword then
 4:     label = true
 5:   else
 6:     // sample ∈ negative samples
 7:     label = false
 8:   end if
 9:   for s = 0 to size do
10:     f += neu1[s] * syn1[sample] [s];
11:   end for
12:   g = getGradient(f, label);
13:   for s = 0 to size do
14:     neu1e[s] += g * syn1[sample] [s];
15:   end for
16:   for s = 0 to size do
17:     syn1[sample] [s] += g * neu1[s];
18:   end for
19: end for
20: // backpropagation.
21: for each wWord, dDoc ∈ Context(tword, tDoc, window) do
22:   for s = 0 to size do
23:     syn0[wWord, dDoc] [s] += neu1e[s];
24:   end for
25: end for
```

4. Experiments

4.1. Experimental setup

For our experiments, a dataset of 5,024 JS codes is used that consists of 2,512 malicious and 2,512 benign JS codes. These JS codes were obtained separately, all malicious codes come from the D3M dataset (Drive-by-Download Data by Marionette) [38,39], whereas the benign JS codes are collected from a generic JS unpacker web-based portal — JSUNPACK [40] and Alexa top 100 websites [41] by depth crawling.

We do performance comparison using the dataset for the two types of feature embedding: Doc2Vec for plain-JS and Doc2Vec for AST-JS. First, we do feature learning using input defined as a regular expression, as shown in Fig. 1, that is, by learning feature vectors from both benign and malicious JS codes represented as a plain-JS. For this, we create 40 dummy data for each original JS code. The resulting dataset is 200,960 JS codes. Second, we do feature learning using input defined as AST-JS as shown in Listing 3, that is, by learning code structure representation. For this, we perform two types of manipulation on the AST-JS to enhance feature learning by Doc2Vec, AST-level merging, and AST subtree realignment. For merging, we generate ASTs from the original JS codes by syntax analysis and then create combinations of the AST bodies to form a new AST-JS. For AST sub-tree realignment, we randomly shuffle the AST-subtrees to change their location in a particular AST-JS. Using these AST-level manipulations, we add two, three, four, and five artificial AST-JS per each original AST-JS. This results in the new datasets of 15,072, 20,096, 25,120 and 30,144 AST-JS. Besides, since the actual web setting generally contains more benign JS codes compared to malicious JS codes, we simulate a real web setting by making ten times more benign AST-JS. This results in the new datasets of 82,896, 110,528, 138,160 and 165,792 AST-JS.

For the paragraph vector models, we found three hyper-parameters to significantly affect the performance of the models. These are the *vector_size* which represents the dimensions of the feature vectors, word frequency given by *min_count* and *window*. For *vector_size*, we did experiments with feature vector dimensions of the range 100–1,000, for word frequency we used the range 1–10 and window size of range 1–8. M and N are mapped to 200 dimensions of p, $min\_count = 5$ for word frequency and a window size of 8. These are the hyper-parameters that we found optimal for effective feature learning and detection of malicious JS contents. We performed cross-validation to evaluate the performance of our approach with $k = 10$ by dividing the feature vectors into 10-folds, where during ten iterations, nine folds are used for classifier training and the remaining one-fold for model evaluation. We used SVM with $kernel = ' linear'$ and $C = 1$ parameters as the classifier for the experiments, a classification method that constructs a hyper-plane in high dimensional space [13,42] for regression and classification. It is mostly used as a binary classifier. For this, we computed precision, recall and F1-score. Precision PRE is ability of a classifier not to label benign a JS code contents as malicious and recall REC is the classifiers ability to find all malicious JS codes contents. F1-score $F1$ can be interpreted as a weighted harmonic mean of precision and recall [42]. With TP as the number of malicious JS codes correctly classified as malicious, TN as the number of benign JS codes correctly classified as benign, FN as the number of malicious JS codes classified as benign and FP as the number of benign JS codes classified as malicious, PRE, REC and $F1$ are computed as,

$$PRE = \frac{TP}{TP + FP} \tag{9}$$

$$REC = \frac{TP}{TP + FN} \tag{10}$$

$$F1 = 2\frac{PRE \times REC}{PRE + REC} \tag{11}$$

A Receiver Operating Characteristic (ROC) graph was used for performance visualization where the true positive rate is plotted against the false positive rate. Area Under the Curve (AUC) score [16] for the ROC was calculated to determine the model performance. We also performed a t-test to check for significant difference in performance between our approach and other feature learning approaches: LDA ($n\_components = 100$, $max\_iter = 5$, $learning\_method = ' online'$, $learning\_offset = 50$. and $random\_state = 0$), LSA ($n\_components = 80$, $algorithm = ' randomized'$, $n\_iter = 5$, $random\_state = None$ and $tol = 0.0$), TF–IDF ($norm = ' l2'$, $use\_idf = True$, $smooth\_idf = True$ and $sublinear\_tf = False$) and Ngram ($ngram\_range = (1, 2)$). This is a statistical test to check for significant difference in model performance by comparing the output. Using the p-value, which is an output of a t-test, one would tell whether our proposed approach truly performed better compared to existing ones. For the experiments alpha value was set to 0.05 therefore if p-value is less than the alpha value, there is a significant difference in performance.

Table 2a
Precision, recall, and F1-score obtained by feature learning using a plain-JS and p value representing significance difference in performance for paragraph vector models compared to other approaches. LDA stands for Latent Dirichlet Allocation, LSA stands for Latent Semantic Analysis, and TF–IDF stands for term frequency–inverse document frequency.

Model	Precision	Recall	F1-score
PV-DBoW	0.95 (± 0.1332)	0.97 (± 0.0429)	0.96 (± 0.0716)
PV-DM	0.95 (± 0.1620)	0.96 (± 0.0473)	0.95 (± 0.0955)
LDA	0.89 (± 0.2456)	0.88 (± 0.1403)	0.87 (± 0.0866) [a]0.0009
LSA	0.85 (± 0.1438)	0.87 (± 0.1320)	0.86 (± 0.1928) [a]0.0011
TF–IDF	0.83 (± 0.3783)	0.84 (± 0.2514)	0.84 (± 0.2140) [a]0.0055
Ngram	0.72 (± 0.0916)	0.74 (± 0.1407)	0.73 (± 0.1213) [a]0.0002

[a]p value.

4.2. Performance comparisons

Precision, recall, and f1-score. This section presents the results of the experiments. Table 2a presents precision, recall, and F1-score for paragraph vector models and other feature learning methods trained using a plain-JS, and includes standard deviation and p-value. The overall performance of the models is generally good as each of the models achieved an average of above 50 percent in precision, recall, and F1-score. Paragraph vector models; PV-DM at 200 dimensions and PV-DBoW at 200 dimensions and $dbow\_words = 1$, achieved superior results compared to the other models. PV-DBoW outperformed other models with 0.06 and 0.23 in precision, 0.09 and 0.23 in recall and 0.09 and 0.23 in F1-score above the best and the worst of the other models.

Ngram with $ngram\_size = 1$ achieved the lowest performance for the task of malicious JS content detection. Consequently, paragraph vector models resulted in less false positives and false negatives compared to the other models. The results of the two-tailed paired t-test to check for a significant difference in performance between paragraph vector models and the other feature learning methods further support the difference in performance for the six models on malicious JS content detection. A comparison of each of the other feature learning methods with PV-DBoW shows a significant difference in performance for malicious JS contents detection where all p-values are less than an alpha value of 0.05.

The objective is to have a high recall, which translates to as fewer false-negative cases (misclassified malicious JS codes) as possible. A low false-negative rate is crucial because a false negative would mean a successful attack, that is, a malicious JS code that goes undetected. Assuming we take 10% of the dataset as the test samples, the models trained with input defined as a plain-JS, shown in Table 2a, would miss 603, 804, 2,412, 2,613, 3,216 and 5,226 malicious JS codes respectively. Even though the paragraph vector models achieve good results with fewer false negatives compared to these other models, there is a considerable need to improve the detection rate of these models further.

Table 2b presents precision, recall and F1-score for PV-DBoW and PV-DM obtained with AST features. PV-DBoW and PV-DM achieves an improvement of over 4% and 3% in precision, 2% in recall and 3% in F1-score when compared to the same models trained using a plain-JS. The significant difference in performance is as a result of JS code representation using AST-JS. AST represents a JS code structure which facilitates capturing of a JS code semantic and syntactic details which would otherwise be lost when input is represented as a plain-JS.

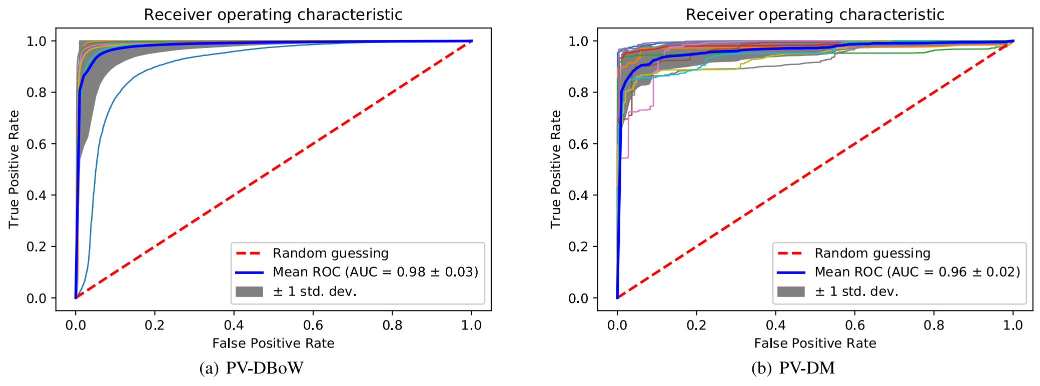


Fig. 3. ROC curves and AUC representing performance of PV-DBoW and PV-DM on malicious JS codes detection with respect to false positive and true positive rates. The dashed diagonal line represents performance threshold as random guessing. The colored curves represent the performance of the specific model for each fold during cross-validation. Also included is the mean AUC score for the individual models.

Table 2b
Precision, Recall and F1-score obtained by feature learning on AST-JS and *p* value representing significance difference in performance for paragraph vector models compared to other approaches.

Model	Precision	Recall	F1-score
PV-DBoW	0.99 (± 0.1413)	0.99 (± 0.0819)	0.99 (± 0.0279)
PV-DM	0.98 (± 0.1271)	0.98 (± 0.0766)	0.98 (± 0.0312)
LDA	0.90 (± 0.2200)	0.92 (± 0.0514)	0.90 (± 0.1388) [a]0.0061
LSA	0.87 (± 0.1091)	0.90 (± 0.1468)	0.87 (± 0.1217) [a]0.0003
TF–IDF	0.85 (± 0.1757)	0.86 (± 0.1077)	0.87 (± 0.3034) [a]0.0229
Ngram	0.76 (± 0.1656)	0.77 (± 0.2165)	0.76 (± 0.1241) [a]0.0005

[a]*p* value.

Table 3
Computational time, representing the training and detection time per JS code obtained by dividing the total computation time with the number of JS code samples. It includes the time for different models.

Model	Training time	Detection time
PV-DBoW	1.4780	0.0019
PV-DM	1.5290	0.0012
LDA	0.2072	0.0190
LSA	0.0826	0.0079
TF–IDF	5.0419	0.8096
Ngram	0.9016	0.1120

True positive and false positive rate. The superior performance of paragraph vector-based models on feature learning for JS contents is also reflected in the ROC graph and AUC results, as shown in Figs. 3 and 4, where paragraph vectors based models achieved high and low rates of true positives and false positives respectively compared to the other models. Ngram achieved the least AUC with a section of the curve being on the threshold. The difference in feature learning and prediction would explain the difference in performance between paragraph vector models and the other models. Ngram predicts x_i based on $x_{i-(n-1)}, \ldots, x_{i-1}$ which limits its effectiveness in detection of malicious JS contents as its main focus is prediction of the next word. TF–IDF works on word level by checking the importance of a word to a document in a corpus and produces a sparse matrix with zero elements. In our experiments, TF–IDF had the longest training time. LSA uses singular value decomposition to reduce matrix row count hence reducing the training time, but LSA has been shown to result with hard to interpret dimensions [14,43]. LDA focuses on word and topic distribution, where each document is assumed to have a mixture of various topics that use a small set of words frequently. This approach is however more suited for information retrieval [15–17]. The use of paragraph and word vectors via Doc2Vec approach, which represents words as feature vectors for a sentence, paragraph, or document preserves word order and produces low dimensionality representations, which are not computationally expensive. As a result, malicious JS contents detection using paragraph vectors and AST's results with a highly accurate and fast model as shown in Table 3 with an average detection time of 0.0019 (± 0.0008) and 0.0012 (± 0.0001) seconds per JS code for PV-DBoW and PV-DM respectively.

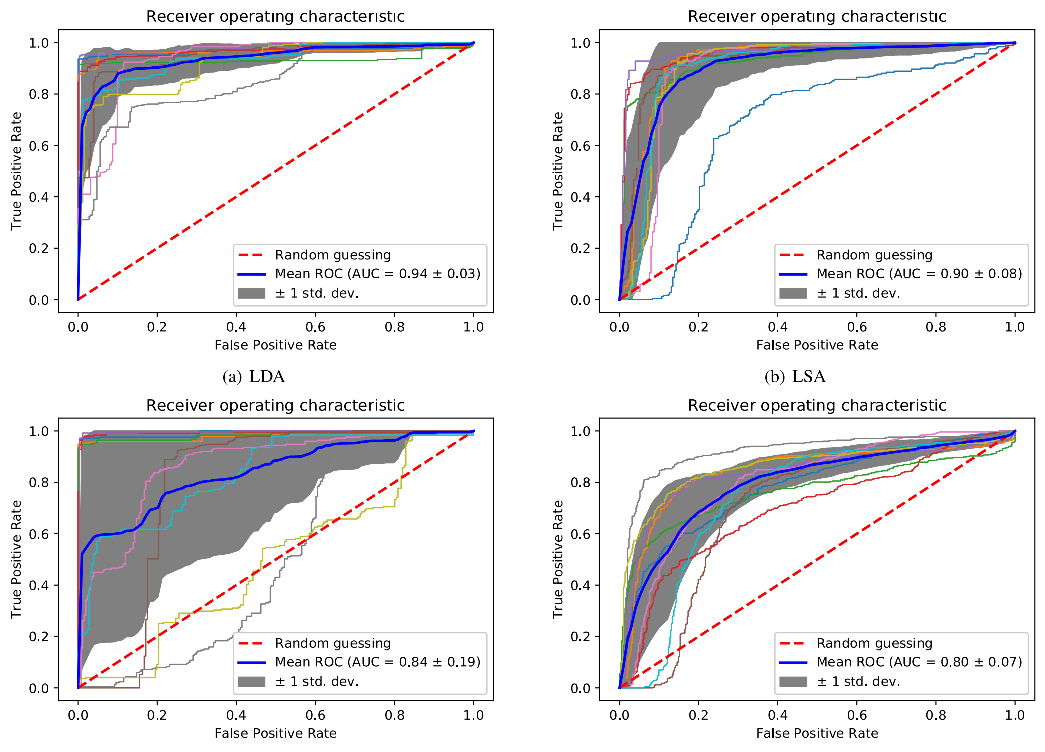
Discussion

As part of our findings in this research, there is a small number of JS codes identified as false positives and false negatives after performing feature learning for a plain-JS. The false positives and negatives cases are largely attributed to obfuscation in some of the JS codes in the dataset, as shown in Fig. 5, which presents hard-to-detect JS codes for both benign and malicious JS codes.

However, obfuscation does not necessarily indicate maliciousness of a JS code [44–46] as a few benign JS codes are often obfuscated to preserve privacy and preventing plagiarism and to fasten code loading by removal of whitespace and comments. On the other hand, malicious JS codes are obfuscated to hide their maliciousness and evade detection and therefore use complicated or combination of obfuscation techniques. Generally, there are four categories of obfuscation techniques [44,45] used in JS codes,

- Randomization obfuscation by changing or inserting elements in JS codes.
- Data obfuscation by string splitting and substitution.
- Logistic structure obfuscation by manipulating a JS code execution path.
- Encoding obfuscation by converting JS codes into ASCII, Unicode, hexadecimal, or by encryption.

Doc2Vec maintains the semantics of words and would therefore accurately detect the first three obfuscation techniques, which do not affect the original semantics of JS code content. Therefore, using Doc2Vec and AST features, most of the code details in the JS codes earlier identified as false positive and false negative cases are now classified accordingly. However, the proposed method has some limitations when faced with obfuscation in the dataset, as shown in Table 4, which shows the performance of paragraph vector models on the obfuscated JS dataset. As shown in the table, our proposed approach can detect obfuscated malicious JS codes with an error rate of approximately 0.08 and 0.07 in recall for PV-DBoW and PV-DM, respectively. The detection is possible because JS codes have syntactic similarities at an abstract level, and therefore, AST can bypass obfuscation. As future work, we will improve on this performance to ensure more accurate detection of obfuscated malicious JS codes. For example, (1) by



(a) LDA (b) LSA

(c) TF-IDF (d) Ngram

Fig. 4. ROC curves and AUC scores representing performance of other models on malicious JS codes detection with respect to false positive and true positive rates. The dashed diagonal line represents performance threshold as random guessing. The colored curves represent the performance of the specific model for each fold during cross-validation. Also included is the mean AUC score for the individual models.

Table 4
Precision, Recall and F1-score representing the performance of paragraph vector models on obfuscated JS codes.

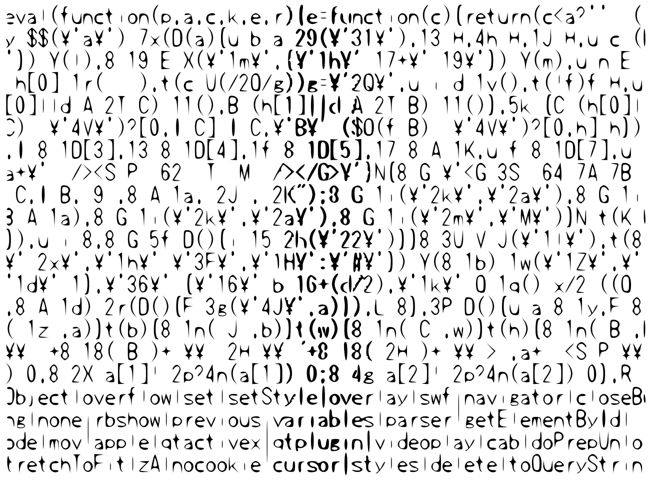
Model	Precision	Recall	F1-score
PV-DBoW	0.94 (± 0.1571)	0.92 (± 0.1159)	0.93 (± 0.0868)
PV-DM	0.93 (± 0.1675)	0.93 (± 0.1016)	0.92 (± 0.0873)

estimating the maliciousness of obfuscated features based on entropy information of string arrays and/or some known signatures of toolkits and (2) by using statistics on obfuscated parts such as a percentage of obfuscated parts in the whole code and/or which parts (e.g., variables and functions) are obfuscated, and so on. To do this, we need to identify obfuscated parts accurately, but this is also a challenging problem at present. Therefore, we left this remedy as future work.

5. Conclusions

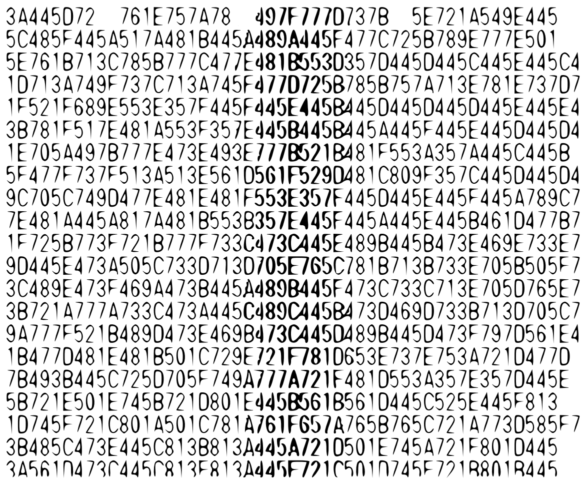
We propose the use of AST-JS for a JS code structure representation and feature learning using Doc2Vec in this paper. Paragraph vector models have previously achieved good results for the task of malicious JS detection using plain-JS features. One expectation is that adopting a JS code structure representation using AST-JS would result in even better performance for the task of malicious JS code detection and enable detection of the previously identified hard-to-detect JS codes. We have conducted experiments using plain-JS, compared the performance of paragraph vector models to LDA, LSA, Ngram, and TF–IDF on the detection of malicious JS codes contents and used a bigger JS dataset. We analyze both the false negative and false positive cases from the results obtained with a JS code input defined as plain-JS. To capture more details of a JS code, we use AST-JS features for code structure representation. AST gives a robust property against some perturbation in codes. The results of our experiments provide a piece of significant evidence that our proposed approach provides good results for the task of malicious JS codes detection. There is a considerable improvement in performance compared to our previous work, which learns a plain-JS as input for Doc2Vec. Besides, our approach eliminates the element of risk brought about by the previously proposed approaches, where a JS code content is evaluated at runtime, exposing the production environment to real-time threats. Also, Doc2Vec automatically learns the feature vectors hence ruling out the need for manually constructing and selecting functions, features (sets and subsets) and rules, which would be a tedious process requiring more resources to achieve. In addition, feature learning by Doc2Vec has the advantage of detecting new and variant malicious JS codes contents as the model has the capability of automatically learning new vector representations contrary to conventional approaches which would require more features, rules, and functions to be defined for effective detection of new malicious JS codes contents not previously defined.

As future work, by use of Generative Adversarial Network (GAN), the Doc2Vec feature learning process can incorporate adversarial machine learning for JS samples. This approach would be used to retrain the model to detect evasion and poisoning attacks performed against machine learning algorithms and at the same time, improve the detection rate. The retraining would achieve even higher accuracy for the task of detection of malicious JS code content using Doc2Vec for a JS code content feature learning. For successful feature learning of malicious JS contents, it is essential to have a fully representative and adequate benign and malicious JS codes dataset, further work on the development of such a dataset is primal.



```
eval(function(p,a,c,k,e,r){e=function(c){return(c<a?''  (
y $$(¥'a¥') 7x(D(a){u b a 29(¥'31¥').13 H.4h H.1J H.u c (l
')) Y(l).8 19 E X(¥'1m¥',{¥'1h¥' 17+¥' 19¥'}) Y(m).u n E
 h[0] 1r(    ).t(c U(/20/g))g=¥'2Q¥'.u  d 1v().t('f)f H.u
[0]lld A 2T C) 11().B (h[1]]|d A 2T B) 11()].5k (C (h[0]l
C)  ¥'4V¥')?[0.l C] l C.¥'B¥'  ($0(f B)  ¥'4V¥')?[0.h] h])
.l 8 1D[3].13 8 1D[4].1f 8 1D[5].17 8 A 1K.u f 8 1D[7].u
a+¥'  /><S P  62  T  M  /></G>¥'}N(8 G ¥'<G 3S  64 7A 7B
 C.l B. 9 .8 A 1a. 2J . 2K");8 G 1l(¥'2k¥',¥'2a¥').8 G 1l
3 A 1a).8 G 1l(¥'2k¥',¥'2a¥').8 G 1l(¥'2m¥',¥'M¥'))N t(K l
]).u l 8.8 G 5f D()(l 15 2h(¥'22¥'))]8 3U V J(¥'1l¥').t(8
¥' 2x¥',¥'1h¥' ¥'3F¥',¥'1H¥':¥'H¥'}) Y(8 1b) 1w(¥'1Z¥',¥'
'1d¥' 1}.¥'36¥' {¥'16¥' b 1G+(d/2).¥'1k¥' 0 1a() x/2 ((0
.8 A 1d) 2r(D(){F 3g(¥'4J¥',a)}}).L 8}.3P D(){u a 8 1y.F 8
( 1z .a))t(b){8 1n( J .b))t(w){8 1n( C .w))t(h){8 1n( B .l
¥¥  +8 18( B )+ ¥¥  2H ¥¥ '+8 18( 2H )+ ¥¥ > .a+  <S P ¥¥
) 0.8 2X a[1]' 2p?4n(a[1]) 0;8 4g a[2]' 2p?4n(a[2]) 0).R
Object|overflow|set|setStyle|overlay|swf|navigator|closeB
ng|none|rbshow|previous|variables|parser|getElementById|
ode|mov|apple|qtactivex|qtplugin|videoplay|cab|doPrepUnlo
tretchToFit|zA|nocookie|cursor|styles|delete|toQueryStrin
```

(a) Malicious JS



```
3A445D72  761E757A78  497F777D737B  5E721A549E445
5C485F445A517A481B445A489A445F477C725B789E777E501
5E761B713C785B777C477E481B553D357D445D445C445E445C4
1D713A749F737C713A745F477D725B785B757A713E781E737D7
1F521F689E553E357F445F445E445B445D445D445D445E445E4
3B781F517E481A553F357E445B445B445A445F445E445D445D4
1E705A497B777E473E493E777B521B481F553A357A445C445B
5F477F737F513A513E561D561F529D481C809F357C445D445D4
9C705C749D477E481E481F553E357F445D445E445F445A789C7
7E481A445A817A481B553B357E445F445A445E445B461D477B7
1F725B773F721B777F733C473C445E489B445B473E469E733E7
9D445E473A505C733D713D705E765C781B713B733E705B505F7
3C489E473F469A473B445A489B445F473C733C713E705D765E7
3B721A777A733C473A445C489C445B473D469D733B713D705C7
9A777F521B489D473E469B473C445D489B445D473F797D561E4
1B477D481E481B501C729E721F781D653E737E753A721D477D
7B493B445C725D705F749A777A721F481D553A357E357D445E
5B721E501E745B721D801E445B561B561D445C525E445F813
1D745F721C801A501C781A761F657A765B765C721A773D585F7
3B485C473E445C813B813A445A721D501E745A721F801D445
3A561D473C445C813F813A445F721C501D745F721B801B445
```

(b) Benign JS

Fig. 5. Example of a hard-to-detect JS code by analysis of false positive and false negative cases in the dataset. These are sample codes for which detection is ineffective while defining input as a plain-JS. These codes also include numbers and special characters in addition to JS code words. Input defined as plain-JS does not adequately capture such JS code details and code structure representation.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.asoc.2019.105721.

Acknowledgments

This research was achieved by the Ministry of Education, Science, Sports, and Culture, Japan, Grant-in-Aid for Scientific Research (B) 16H02874 and the Commissioned Research of National Institute of Information and Communications Technology (NICT), Japan, Grant number 190.

References

[1] N. Khan, J. Abdullah, A.S. Khan, Defending malicious script attacks using machine learning classifiers, Wirel. Commun. Mobile Comput. 2017 (5360472) (2017) 1–9.

[2] O. Mogren, Malicious JavaScript detection using machine learning, Learning 10 (11) (2017) 1–7, http://www.cse.chalmers.se/edu/course/FDAT085.

[3] K. Schutt, M. Kloft, A. Bikadorov, K. Riek, Early detection of malicious behavior in JavaScript code, in: Proceedings of the 5th ACM Workshop on Artificial Intelligence and Security, AISec'12, Raleigh, NC, USA, vol. 12, 2012, pp. 15–24.

[4] J. Grossman, P.D. Petkov Hansen, A. Rager, S. Fogie, XSS Attacks: Cross-Site Scripting Exploits and Defense, Syngress, Burlington, MA, 2007.

[5] M. Cova, C. Kruegel, G. Vigna, Detection and analysis of drive by download attacks and malicious JavaScript code, in: Proceedings of the 19th International Conference on World Wide Web, April 2010.

[6] Y. Choi, T. Kim, S. Choi, C. Lee, Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis, in: International Conference on Future Generation Information Technology, Springer, 2009, pp. 160–172.

[7] Y. Alosefer, O. Rana, Honeyware: A web based low interaction client honeypot, in: IEEE third International Conference on Software Testing, Verification and Validation Workshops, ICSTW, 2010, pp. 410–417.

[8] H. Kim, D. Kim, S. Cho, M. Park, Efficient detection of malicious web pages using high interaction client honeypots, J. Inform. Sci. Eng. 28 (5) (2012) 911–924.

[9] Q. Le, T. Mikolov, Distributed representations of sentences and documents, in: Proceedings of the 31st International Conference on Machine Learning, ICML-14, 2014, pp. 1188–1196.

[10] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in: ICLR Workshop Papers, 2013, pp. 1–12.

[11] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: Proc. Advances in Neural Information Processing Systems, vol. 26, 2013, pp. 3111–3119.

[12] A. Dai, C. Olah, Q. Le, Document embedding with paragraph vectors, in: NIPS Deep Learning Workshop, 2014, pp. 1–8.

[13] C.J.C. Burges, A tutorial on support vector machines for pattern recognition, Data Mining Knowl. Discov. 2 (2) (1998) 121–167.

[14] Y. Shirota, B. Chakraborty, Visual explanation of mathematics in latent semantic analysis, in: 2015 IIAI 4th International Congress on Advanced Applied Informatics, 2015, pp. 423–428.

[15] Y. Wang, J.S. Lee, I.C. Choi, Indexing by latent dirichlet allocation and an ensemble model, J. Assoc. Inform. Sci. Technol. 67 (7) (2016) 1736–1750.

[16] R. Deveaud, E. SanJuan, P. Bellot, Accurate and effective latent concept modeling for ad hoc information retrieval, Doc. Numer. (2014) 61–84.

[17] H. Jelodar, Y. Wang, C. Yuan, X. Feng, Latent dirichlet allocation (LDA) and topic modeling: models applications a survey, 2017, pp. 1–40, arXiv:1711.04305 [cs.IR].

[18] T. Krueger, K. Rieck, Intelligent defense against malicious javascript code, Prax. Inf.verarb. Kommun. 35 (1) (2012) 54–60.

[19] W.-H. Wang, Y.-J. LV, H.-B. Chen, Z.-L. Fang, A static malicious JavaScript detection using SVM, in: Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering, ICCSEE, vol. 40, 2013, pp. 21–30.

[20] Y. Wang, W. d. Cai, P. c. Wei, A deep learning approach for detecting malicious JavaScript code, in: Security and Communication Networks, Wiley, 2016, pp. 1520–1534.

[21] S. Yoo, S. Kim, A. Choudhary, O. Roy, T. Tuithung, Two-phase malicious web page detection scheme using misuse and anomaly detection, Int. J. Reliable Inf. Assur. 2 (1) (2014) 1–9.

[22] J. Wang, Y. Xue, Y. Liu, T.H. Tan, Jsdc: A hybrid approach for javascript malware detection and classification, in: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ACM, 2015, pp. 109–120.

[23] Y. Xue, J. Wang, Y. Liu, H. Xiao, J. Sun, M. Chandramohan, Detection and classification of malicious javascript via attack behavior modelling, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, New York, NY, USA, ACM, 2015, pp. 48–59.

[24] S. Ndichu, S. Ozawa, T. Misu, K. Okada, A machine learning approach to malicious javascript detection using fixed length vector representation, in: Proc. of 2018 International Joint Conference on Neural Networks, IJCNN 2018, 2018, pp. 1–8.

[25] M. Fraiwan, R. Al-Salman, N. Khasawneh, S. Conrad, Analysis and identification of malicious javascript code, Inf. Secur. J. 21 (1) (2012) 1–11.

[26] Y. Zou, K. Kontogiannis, Towards a portable XML-based source code representation, in: ICSE 2001 Work-shops of XML Technologies and Software Engineering, XSE, 2001.

[27] J. Jones, Abstract syntax tree implementation idioms, Pattern Lang. Programs (2003).

[28] J. Misek, F. Zavoral, Mapping of dynamic language constructs into static abstract syntax trees, in: Proceedings of the 2010 IEEE/ACIS 9th International Conference on Computer and Information Science, 2010, pp. 625–630.

[29] C. Curtsinger, B. Livshits, B. Zorn, C. Seifert, Zozzle: low-overhead mostly static JavaScript malware detection, in: Proceedings of the Usenix Security Symposium, 2011, pp. 1–18.

[30] G. Blanc, D. Miyamoto, M. Akiyama, Y. Kadobayashi, Characterizing obfuscated JavaScript using abstract syntax trees: experimenting with malicious scripts, in: Proc. 26th Advanced Information Networking and Applications Workshops, WAINA, 2012, pp. 344–351.
[31] M. Kamizono, M. Nishida, E. Kojima, Y. Hoshizawa, Categorizing hostile JavaScript using abstract syntax tree analysis, Inf. Process. Soc. Japan (2012) 349–356.
[32] T. Johnishi, M. Kamizono, M. Nashida, M. Morii, Classification of hostile javascript based on encoding abstract syntax tree, Comput. Secur. Symp. (2012) 232–237.
[33] A. Hidayat, http://esprima.org/, Esprima Release master, April 15, 2018.
[34] Y. Zhang, R. Jin, Z.-H. Zhou, Understanding bag-of-words model: a statistical framework, Int. J. Mach. Learn. Cybern. 1 (1–4) (2010) 43–52.
[35] Y. Goldberg, Neural Network Methods for Natural Language Processing, in: Synthesis Lectures on Human Language Technologies, vol. 37, Morgan and Claypool, 2017, p. 69.
[36] R. Socher, C.C. y. Lin, A.Y. Ng, C.D. Manning, Parsing natural scenes and natural language with recursive neural networks, in: Proceedings of the 28th International Conference on Machine Learning, ICML- 11, 2011, pp. 129–136.
[37] R. Radim, P. Sojka, Software framework for topic modelling with large corpora, in: Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks, 2010, pp. 45–50.
[38] M. Akiyama, K. Aoki, Y. Kawakoya, M. Iwamura, M. Itoh, Design and implementation of high interaction client honeypot for drive-by-download attacks, IEICE Trans. Commun. E93-B (5) (May 2010) 1131–1139.
[39] M. Hatada, M. Akiyama, T. Matsuki, T. Kasama, Empowering anti-malware research in Japan by sharing the MWS datasets, IPSJ J. Inform. Process. 23 (5) (Sep. 2015) 579–588.
[40] JSUPACK, A generic JavaScript unpacker, 2017, http://jsunpack.jeek.org/.
[41] Alexa, Inc, The top 500 sites on the web, 2018, https://www.alexa.com/topsites.
[42] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[43] B. Pincombe, Comparison of human and latent semantic analysis (LSA) judgements of pairwise document similarities for a news corpus, Tech. rep. DSTO-RR-0278, Information Sciences Laboratory, Defence Science and Technology Organization, Department of Defense, Australian Government, 2004.
[44] W. Xu, F. Zhang, S. Zhu, The power of obfuscation techniques in malicious JavaScript code: A measurement study, in: 7th International Conference on Malicious and Unwanted Software, MALWARE, IEEE, 2012, pp. 9–16.
[45] A. Gorji, M. Abadi, Detecting obfuscated JavaScript malware using sequences of internal function calls, in: Proc. 52nd ACM Southeast Conference, ACMSE'14, Kennesaw, GA, USA, 2014, pp. 1–6.
[46] Y. Choi, T. Kim, S. Choi, C. Lee, Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis, in: Proceedings of the 1st International Conference on Future Generation Information Technology, FGIT, Springer-Verlag Berlin Heidelberg, 2009, pp. 160–172.