Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement

Ndichu, Samuel

Kim, Sangwook

Ozawa, Seiichi

(Citation)
CAAI Transactions on Intelligence Technology,5(3):184-192

(Issue Date)
2020-07-17

(Resource Type)
journal article

(Version)
Version of Record

(URL)
https://hdl.handle.net/20.500.14094/90009107

CAAI Transactions on Intelligence Technology

Research Article

Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement

ISSN 2468-2322
Received on 30th January 2020
Revised on 29th April 2020
Accepted on 8th June 2020
doi: 10.1049/trit.2020.0026
www.ietdl.org

Samuel Ndichu[1] ✉, *Sangwook Kim*[1], *Seiichi Ozawa*[1,2]

[1]*Graduate School of Engineering, Kobe University, Kobe City, Japan*
[2]*Center for Mathematical and Data Sciences, Kobe University, Kobe City, Japan*
✉ *E-mail: sammiendichu01@stu.kobe-u.ac.jp*

Abstract: Obfuscation is rampant in both benign and malicious JavaScript (JS) codes. It generates an obscure and undetectable code that hinders comprehension and analysis. Therefore, accurate detection of JS codes that masquerade as innocuous scripts is vital. The existing deobfuscation methods assume that a specific tool can recover an original JS code entirely. For a multi-layer obfuscation, general tools realize a formatted JS code, but some sections remain encoded. For the detection of such codes, this study performs Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) by function redefinition using a Virtual Machine (VM), a JS code editor, and a python *int_to_str()* function to facilitate feature learning by the FastText model. The learned feature vectors are passed to a classifier model that judges the maliciousness of a JS code. In performance evaluation, the authors use the Hynek Petrak's dataset for obfuscated malicious JS codes and the SRILAB dataset and the Majestic Million service top 10,000 websites for obfuscated benign JS codes. They then compare the performance to other models on the detection of DUD-preprocessed obfuscated malicious JS codes. Their experimental results show that the proposed approach enhances feature learning and provides improved accuracy in the detection of obfuscated malicious JS codes.

1 Introduction

JavaScript (JS) obfuscation is a transformation aimed at generating a JS code that is obscure to the human eyes and undetectable to scanners. Obfuscation hinders the comprehension and analysis of JS code content. This method of code transformation is rampant in both benign and malicious JS codes. Therefore, it would be wrong to naively render a JS code as malicious if it contains some form of obfuscation [1–3]. Obfuscation in software and programs, such as JS codes and HyperText Markup Language (HTML), has advantages of, among others, proprietary code protection, code compression, performance optimisation, and cubing code reverse engineering [4, 5]. However, obfuscation is also used by cyber-attackers to camouflage the JS codes malicious intentions while preserving the overall code behaviour. This property, when used in malicious JS codes, is used to circumvent detection and identification, significantly affecting the performance of network and information security tools such as the intrusion detection systems and anti-virus software.

The use of data and encoding-based obfuscation techniques can reduce the detection rate for signature and heuristics-based tools by ~40 and 100%, respectively [6]. Other languages, such as HTML have a finite number of methods of obfuscation [2, 7], for example, a HTML element obfuscation by transforming an *iframe* to perform evasion attacks. On the other hand, JS code is dynamic. Hence, it is resilient to a variety of obfuscation techniques that are widely used in JS codes, such as variable and function names randomisation, string splitting and concatenation, keyword substitution, dead code insertion, packing, and encoding, such as Base64 [8, 9]. It is also possible to implement customised encoding to obfuscate a JS code [6] where a special function is inserted and used to decode the JS code at runtime. Listing 1 (see Fig. 1) is the encoding function where each character's unicode is increased by one. Listing 2 (see Fig. 2) is the resultant obfuscated JS code which is decoded before execution [6]. The function 'document.write('Hello world!')' in the encoding function is encoded into 'epdvnfou/xsjuf)(Ifmmp!xpsme(*', in the resultant obfuscated JS code.

Therefore, it is vital to accurately detect malicious JS codes that use obfuscation to masquerade as innocuous scripts. There exist several methods to analyse such obfuscated JS codes. These approaches perform either static or dynamic analysis of obfuscated JS codes. Some of the JS code deobfuscation approaches assume that it is possible to recover an original JS code entirely using a specific deobfuscation tool. We apply such a strategy to analyse obfuscated malicious JS codes using general deobfuscation tools such as Dirty Markup [10], Online JS Code Beautifier [11], and Dan's Tools JS Code Formatter [12]. Applying the deobfuscation tools helps in realising a readable and pretty JS code that is easier to edit and analyse. However, as part of our findings, there still exists some sections of a JS code which proves hard to deobfuscate, for example, a JS code with multiple layers of obfuscation.

In our previous work [13, 14], we performed feature learning for regular JS code using Plain-JS codes (Plain-JS) features, abstract syntax tree (AST) features, and paragraph vectors. AST form of JS code (AST-JS) gives a robust property against some perturbation in JS codes. Using AST-JS, detection of obfuscated JS code with obfuscation techniques that do not affect the original semantics of a JS code contents would be possible. However, it is not appropriate for the representation of a JS code that contains packing and or encoding obfuscation techniques that convert JS code contents into ASCII, Unicode, hexadecimal, and encrypted values. Only about 50% of the obfuscated JS code preserves its original semantics [15]. This study focuses on the detection of obfuscated malicious JS codes, precisely packed and encoded JS code, including the ones which contain multi-layer obfuscation.

```
function encode(mystring)
{
var c="";
var i =0;
for(var i=0;i
<mystring.length;i++){
c = c +
String.fromCharCode(mystring.charCode
At(i)+1);
}
return c
}

var c =
encode("document.write('Hello world!')");
document.write(c);
```

Fig. 1 *Listing 1: the encoding function*

```
function decode(c)
{
var mystring="";
var i =0;
for(var i=0;i <c.length;i++){
mystring = mystring +
String.fromCharCode(c.charCodeAt(i)-1);
}
return mystring;
}

var c ="epdvnfou/xsjuf)(Ifmmp!xpsme(⋆";
var mystring = decode(c);

eval(decode(c));
```

Fig. 2 *Listing 2: the obfuscated JS code*

This paper proposes a hybrid-analysis approach by integrating the use of a JS code beautifier and formatter, a virtual machine (VM), a JS code editor, and a python *int_to_str()* function to deobfuscate an obfuscated JS code, unpack a packed JS code, and decode an encoded JS code, respectively. The FastText model, a machine learning model, performs feature learning for DUD-preprocessed JS codes. A classifier model judges the maliciousness of an obfuscated JS code using the learned fixed-length vectors. Our approach automatically learns feature vectors compared to other previous methods that learn a set of manually extracted features. Besides, features learned are of low dimensions hence ensuring faster detection. The organisation 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 the paragraph vector models, the StarSpace model, the long short-term memory (LSTM) model, and the term frequency–inverse document frequency (TF-IDF) model. Finally, we give our discussion and conclusion in Sections 5 and 6, respectively.

2 Related work

There exist several approaches for the detection of obfuscated malicious JS codes. The most basic and widely used approach is the use of JS code beautifier tools and formatters, such as DirtyMarkup JS code beautifier [10], Online JS code beautifier [11], and Dan's Tools JS code viewer, beautifier and formatter [12], among others. These tools make an obfuscated JS code look pretty, readable, editable, and analysable. The tools are intended to reverse the effects of obfuscation for JS code analysis. However, these JS code deobfuscation tools are no match to an obfuscated JS code with multiple layers of obfuscation as some sections of the code remain obfuscated even after a JS code deobfuscation and formatting.

Other studies attempt to perform JS code deobfuscation for the discovery of the original and malicious JS codes by converting the obfuscated JS codes to regular ones for analysis [7, 16, 17]. In other approaches, a set of heuristics is created by analysing specific obfuscator features. However, the heuristics-based approach needs security personnel intervention during code analysis, and it is limited to specific obfuscator tools [15, 18]. Other methods use a set of features such as JS code functions known to be prevalent in obfuscated malicious JS codes such as *eval()*, *unescape()*, and *document.write()*. These syntactic-based features are prone to a high false positive rate [1–3]. Recently, approaches that use machine learning and deep learning to detect obfuscated JS codes are also increasing [15, 17–20].

Skolka *et al.* [15] analyse the extent of application of minification and obfuscation in the web with an emphasis on understanding the most popular tools used for code transformation. The study point towards the development of targeted defence techniques that focus on particular obfuscation tools. They use a tree-based convolutional neural network to classify AST for three classification tasks, namely, TRANS, which determines whether a given piece of JS code has been minified or obfuscated using a tool, OBFUS, which checks for the presence of obfuscation and TOOL-X, which determines the specific tool used to obfuscate a JS code. They, however, do not evaluate obfuscation methods applied in benign, and those in malicious JS code and study of specific obfuscation tools could lead to biased features that cannot be easily generalised to other datasets.

Some deobfuscation techniques have been developed to conduct reverse engineering of an obfuscated JS code. For example, Yadegari *et al.* [16] developed a generic deobfuscation method to extract a simplified code. For this, the paper combined a dynamic analysis with concolic (symbolic) executions in order to collect JS code traces. These traces are simplified in order to reconstruct a control-flow graph (CFG) and data-flow of the target program using taint propagation. Obfuscation adds unnecessary transformations instructions and constructs. These details are discarded, and only the parts essential for functionality are maintained. Some automatic attacks are often specific to particular implementations of obfuscation transformations or individual attacker goals, such as CFG simplification. Some JS code analysis technique requires execution traces that cover all the code of the program being analysed. This process ensures that the CFG is complete with no missing statements, basic blocks, or arcs [21]. However, the proposed system [16] only emphasises on the simplification of a JS code execution traces without performing code coverage. This paper does not make any assumption about the obfuscation technique. Moreover, it is a dynamic unpacker that needs to unpack the code in memory at runtime to obtain the unpacked code.

Lu and Debray [7] proposed a semantics-based approach for automatic deobfuscation of a JS code using dynamic analysis. They collect bytecode execution traces from the target program and use dynamic slicing and semantics-preserving code transformations to simplify the trace automatically, then reconstruct deobfuscated JS code from the simplified trace. The resulting code becomes observationally equivalent to the original program for the execution path considered. Obfuscations are simplified away. This method exposes the core logic of the computation a JS code performs. This method achieves an excellent detection performance. However, the deobfuscation and simplification methods do not identify a JS code maliciousness, as they are designed for the detection of code-unfolding and string-based obfuscations.

Attackers use JS code functions, such as *eval()*, *unescape()*, and *escape()*, to obfuscate malicious JS codes. These functions can be easily detected using signature-based tools. Hence, the functions are mostly hidden. Gorji and Abadi [17] use internal function debugging and map the behaviour of a webpage to a sequence of internal function calls, generating a behavioural signature for each JS code malware family. To identify these hidden functions, they intercept the suspicious function invocations, which can be triggered at runtime using a browser. They use reverse engineering to identify internal function calls. However, obfuscation tools can result in a JS code with behavioural characteristics that are different from the original JS code [15]. Also, they select a limited set of 15 internal function calls of *jscript.dll* and *mshtml.dll* from *jscript* and *mshtml* libraries, which they assume are fully representative for the JS code malware family.

Multiple classifiers have been proposed to detect obfuscation and evaluate the feasibility and accuracy of distinguishing between different classes of a JS code. Tellenbach *et al.* [18] collect a dataset of obfuscated and non-obfuscated JS codes and selects and extracts a set of 45 features from the dataset. The features used include frequency of specific keywords, number of lines, characters per line, number of functions, entropy, among others. This paper assumes that there are a few JS codes that are both benign and obfuscated, and hence, they mostly focus on classifying obfuscated and non-obfuscated JS codes. They also fail to detect JS codes obfuscated with tools not present in the training set, and detection precision depends on the manually specified limited set of features.

For detection of obfuscated malicious JS code instances, Fass *et al.* [19] leverage the use of syntactic units using *n*-grams features from an AST traversal and a random forest classifier. They use specific constructs for accurate detection distinguishing obfuscation from maliciousness. However, they grouped units with the same abstract syntactic meaning, such as *FunctionDeclaration* and *VariableDeclaration* as Declaration node, and *ForStatement* and *WhileStatement* as statement node, therefore losing context information. They also use specific features, namely extracted syntactic units, which translates to limited learning components. In contrast, we use all AST information so that the FastText model can learn the most detailed features.

Algorithms such as J48 decision tree, Naïve Bayes, Logistic Regression, and linear SVM have been proposed [20] to develop a machine-learning based approach to detect obfuscated malicious JS code. The approach dynamically executes the trace of a JS code and extract unordered and non-consecutive sequence patterns. Semantics-based deobfuscation technique [7] is adopted to simplify the trace using deobfuscation slicing. This approach assumes that the deobfuscated trace will be useful for feature extraction that represents the internal characteristics of a malicious JS code. Also, obfuscation tools have been shown to result in codes that are structurally different from the original ones.

These existing approaches adopt either a static or dynamic approach to deobfuscation and detection of obfuscated malicious JS codes. The main issue in static analysis is coping with obfuscation, specifically, packing and encoding obfuscation. Static techniques have also been employed to assess if a malicious JS code is similar to a previously seen variant, without actually performing the costly task of unpacking and decoding. However, with attacks such as zero-day JS code malware using obfuscation, a static analysis technique generates many false positives. Dynamic analysis techniques have been shown to suffer from limited code coverage and the complexity of the JS code runtime environment [22]. Limited code coverage results from cloaking, a technique employed by attackers to fingerprint the victim's web browser. The malicious content can only be visible if certain conditions are met, for example, the use of a specific browser version and a vulnerable plugin. The complexity of the JS code runtime environment results because JS code is in a variety of applications utilising the functionalities of plugin extensions they support. A dynamic analysis set-up will majorly include known browsers and plugins, ignoring less popular plugins and their vulnerabilities.

Most of the JS code dynamic analysis techniques use unfolded code contexts logs and assume that the original code can be retrieved from the deobfuscated one. JS code event callback feature, such as a mouse click event, also makes it challenging for dynamic analysis to trigger a JS code automatically. For example, the deobfuscator by Lu and Debray [7] is a fully dynamic solution. However, the JS code obfuscated part requires localisation as preprocessing the unfold loops during the abstraction stage.

Some of these JS code deobfuscation and detection approaches assume that the transformation applied to a JS code preserves the overall semantics of the code. Benign JS codes may apply simple obfuscation techniques such as string splitting. However, in the case of malicious JS codes, attackers apply sophisticated obfuscation techniques such as packing and encoding in an attempt to perform evasion attacks. In some instances, a single code contains different obfuscation techniques, for example, double packing or packing coupled with encoding. These multi-layered obfuscation techniques dramatically affect the structure of the resultant JS code. In such a case, the resultant AST-JS would not accurately capture the structure of the original JS code. Therefore, to improve the detection performance of machine learning, we propose a combination of the deobfuscation, unpacking, and decoding (DUD-preprocessing) approach to enhance feature learning.

3 Proposed method

Malicious JS codes attempt to trick users using items such as text, function names, function types, variables, and other contents. Obfuscation is employed to evade detection by security devices. This property can also hinder the feature learning process of a machine learning model as such JS codes contain numbers and special characters in addition to JS code words. It is essential to capture JS code details and structure by code structure representation to detect such malicious JS codes effectively. Ndichu *et al.* [14] developed the AST-JS approach for code structure representation and accurately detected some obfuscated malicious JS codes which contain the same semantics and syntax as the original JS codes. However, malicious JS codes employ sophisticated obfuscation methods such as packing, encoding, and multi-layered obfuscation methods. The resulting obfuscated malicious JS codes are characterised by unnatural and unreadable syntax and corrupted code structure that is different from that of the original JS code. To effectively conduct feature learning on such obfuscated JS codes, it is essential to deobfuscate, unpack, and decode the obfuscated JS codes. We propose the use of a JS code beautifier, formatter, VM, JS code editor, and a python *int_to_str()* function to obtain a Plain-JS code from an obfuscated one and the FastText model for feature learning coupled with a classifier for the prediction task.

In the following sections, we explain the preprocessing for transforming an obfuscated JS code into a Plain-JS code and the FastText model that we use for feature learning.

3.1 Preprocessing

Deobfuscation: Deobfuscation is the process of analysing and formatting an obfuscated JS code to make it readable again. It is also referred to as simplification or beautification. Deobfuscation uncovers the actual functionality of a JS code. We use a JS code beautifier to format an obfuscated JS code resulting in a deobfuscated JS code. However, if upon analysis of the resultant deobfuscated JS code, we still find the presence of obfuscation in the dataset, there is still a need to process such an obfuscated JS code further. We identify such JS codes as hard-to-deobfuscate JS codes as sections of the code remain obfuscated even after deobfuscation attempts.

Unpacking: A JS code is packed or encrypted to hide its maliciousness, thereby hindering its interpretation by static

CAAI Trans. Intell. Technol., 2020, Vol. 5, Iss. 3, pp. 184–192

analysis. This concept is also used in benign JS codes for code compression and privacy purposes. JS code packers are diverse and more common with attackers attempting to evade detection. These tools wrap the entire code using the *eval()* function that takes a string and attempts to run it as a JS code. The code is then loaded at runtime via the function.

We evaluate the packed JS codes using oracle VM and a JS code editor. We strip the script tags such that for each JS code; $JS\_code = "'eval(function(p,a,c,k,e,d)...obfuscated\_JS\_code\ldots)"'$, replace *eval()* with *console.log()* and parse the packed JS code to extract function arguments using $Unpacked\_JS\_code = eval\ (\text{'}unpack\text{'} + JS\_code[JS\_code.find(\text{'}\}\text{'}() + 1 : -1])$ regular expression. It is important to disable the drag and drop and shared clipboard options in the VM virtual box manager and use a wifi network to avoid infecting the host personal computer and the local area network with malware. This process results in an unpacked JS code. However, upon analysis of the resultant unpacked JS code, we still find the presence of obfuscation in the dataset. The JS codes that remain obfuscated contain encoding implemented using *basex()* functions.

Decoding: JS code encoding is the process of converting all string literals of the stringArray, for example, by a JS code obfuscator tool, using either Base64 or RC4 [8, 9]. JS code packers also encode strings in a JS code to integers. The tools encode a JS code using a *basex()* function. The resultant JS code is a shorter version that represents the long original one. It is a JS code compression method to reduce download time and to represent a uniform resource locator (URL) anonymously. Encoding methods such as ASCII, Unicode, and HEX affect the semantics of a JS code.

Python allows conversion of a string to an integer of a given base using *class int(x = 0)*, *class int(x, base = 10)* function, which returns an integer object constructed from a number or string given by *x*, or returns 0 if no arguments are given [23]. We evaluate encoded JS codes by performing the inverse of this function. We implement an *int_to_str(num,base)* function in python such that, for a number *num* and given *base*, $int(int\_to\_str(num, base), base) == num$ which returns a string object constructed from a number *num*, or return 0 if no arguments are given. Finally, we parse the encoded JS codes to extract the function arguments using a JS code editor in a VM.

This process results in a Plain-JS code. The Plain-JS code is parsed to AST-JS for feature learning using the FastText model. Listing 3 (see Fig. 3) shows a JS code with encoding obfuscation. The string 'New user' in the hello("New user") function call from Listing 4 (see Fig. 4) had been replaced to a call to a function that retrieves its value at runtime, such as *var _0 × 74f5*.

Fig. 5 shows the overview of the steps to deobfuscate, unpack, and decode an obfuscated JS code. It consists of three steps, namely, deobfuscation or formatting, unpacking, and decoding. During deobfuscation, a JS code beautifier and formatter are used to deobfuscate a JS code to make it pretty, readable, easier to edit, and analyse. Then, the resulting formatted JS code is analysed for

```
var _0x74f5=["\x48\x65\x6C\x6C\x6F\x2C\x20","\x6C
    \x6F\x67","\x4E\x65\x77\x20\x75\x73\x65\x72"
    ];function hello(_0xe170x2){console[_0x74f5
    [1]](_0x74f5[0]+ _0xe170x2)}hello(_0x74f5[2])
```

Fig. 3 *Listing 3: an obfuscated JS code representation using hexadecimal to implement encoding*

```
function hello(name){
console.log("Hello, " + name);
}
hello("New user");
```

Fig. 4 *Listing 4: the original JS code*

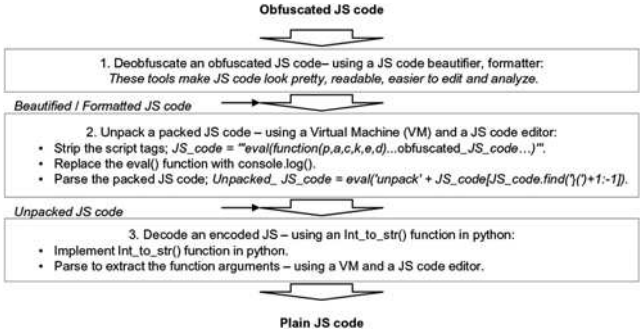

Fig. 5 *Steps to deobfuscate, unpack and decode an obfuscated JS code*

obfuscation. If there is still evidence of obfuscation, a VM and a JS code editor are used to unpack the JS code by function redefinition. The unpacked JS code is further examined for the presence of encoding, and a *int_to_str()* decoding function is applied to realise a Plain-JS code.

3.2 Feature learning by the FastText model

There exist several models that learn continuous representations for word, sentence, and paragraphs in the natural language processing (NLP) tasks. Models such as bag-of-words [24, 25], Word2Vec [26, 27], and Doc2Vec [28, 29] result in a unique vector for each word, thereby ignoring the internal structure of words. Besides, the produced vectors do not share parameters.

The FastText model [30, 31] is an approach based on the continuous skip-gram training algorithm [26]. The model represents each word as a bag of character *n*-grams. The resulting vector representation for a word is the sum of its character *n*-gram vectors taking into account subword information. The model is fast, therefore suitable for feature learning on big datasets and can learn vector representation for out of vocabulary words.

The Word2Vec [26] continuous skip-gram training algorithm learns a vector representation for each word x given a word vocabulary of size X, and each word identified by index $x \in \{1, \ldots, X\}$. The model predicts words in a context given a word. For a large training corpus $\boldsymbol{X}_c = \{\boldsymbol{x}_{t-2}, \boldsymbol{x}_{t-1}, \boldsymbol{x}_{t+1}, \boldsymbol{x}_{t+2}\}$, the model predicts the surrounding words, context $\boldsymbol{X}_c$ given the centre word $\boldsymbol{x}_t$ represented by $\boldsymbol{X}_t$. The objective of the Word2Vec continuous skip-gram training algorithm is to maximise the log-likelihood

$$\sum_{t=1}^{T} \sum_{c \in C_t} \log p(\boldsymbol{X}_c | \boldsymbol{X}_t), \tag{1}$$

where C_t is the set of indices of the context $\boldsymbol{X}_c$. Given a scoring function f which maps pairs of $(\boldsymbol{X}_t, \boldsymbol{X}_c)$ to scores in $\mathbb{R}$, the softmax can be used to define the probability of a context word

$$p(\boldsymbol{X}_c | \boldsymbol{X}_t) = \frac{e^{f(\boldsymbol{X}_t, \boldsymbol{X}_c)}}{\sum_{j=1}^{X} e^{f(\boldsymbol{X}_t, j)}}. \tag{2}$$

This type of model only predicts one context word $\boldsymbol{X}_c$ given a word $\boldsymbol{X}_t$. The task of predicting context words can be viewed as a set of independent binary classification with the objective of predicting presence or absence of $\boldsymbol{X}_c$. For a word $\boldsymbol{X}_t$, all $\boldsymbol{X}_c$ are the positive examples, negatives examples are sampled randomly from the dictionary. The binary logistic loss is used to obtain the negative log-likelihood for $\boldsymbol{X}_c$

$$\log\left(1 + e^{-f(\boldsymbol{X}_t, \boldsymbol{X}_c)}\right) + \sum_{n \in N_{t,c}} \log\left(1 + e^{f(\boldsymbol{X}_t, n)}\right), \tag{3}$$

where $N_{t,c}$ is a set of negative examples sampled from X. The objective function can be given by

$$\sum_{t=1}^{T}\left[\sum_{c\in C_t}\ell\left(f\left(\boldsymbol{X}_t,\boldsymbol{X}_c\right)\right)+\sum_{n\in N_{t,c}}\ell\left(-f\left(\boldsymbol{X}_t,n\right)\right)\right], \tag{4}$$

by denoting the logistic loss function $\ell:x\mapsto\log\left(1+e^{-x}\right)$.

The FastText model introduces a different scoring function f where a bag of character n-gram vectors represent each word x. This scoring function takes into account the internal structure of a word. Prefixes and suffixes are distinguished using <and>, special boundary symbols, added at the start and end of a word. For a word such as *encode* with $n=3$, its character n-grams will be <*en*, *enc*, *nco*, *cod*, *ode*, *de*> and <*encode*> which is a special sequence that is used to learn each words representation. Given a dictionary of size G n-gram vectors and a word x, $G_x\subset\{1,\ldots,G\}$ gives the set of n-gram vectors in x. The scoring function f is given by the sum of vector representations of a words n-grams

$$f(x,c)=\sum_{g\in G_x}\boldsymbol{Z}_g^{\top}\boldsymbol{X}_c, \tag{5}$$

where $\boldsymbol{Z}_g$ is vector representation for each n-gram g. The n-gram vector representations are shared across words. This ensures reliable vector representations are learnt for rare words [32]. A computation of the most similar word to *encode* using the trained FastText model, *model_JS.wv.most_similar("encode")*, results in '*encodedurl*', 0.91, '*encodeuri*', 0.89, '*htmlencode*', 0.89, '*enc_str*', 0.87, and '*decodeuri*', 0.83. This function is used in malicious JS code to obfuscate the contents of a JS code. The words identified by the model are correct because each encoding function is accompanied by a decoding function. The encoding function is used to encode a string or a URL in a JS code and the decoding function decodes it at runtime.

The FastText model implementation [33] creates n-gram vectors for words. We obtain a JS code feature vector by superimposing the n-gram word vectors. Given M as the total number of words in a JS code, the vector representation of a JS code will be given by

$$\left(\boldsymbol{V}_{\text{JS\_code}}\right)=\frac{\sum\boldsymbol{X}_n}{M}. \tag{6}$$

where $\boldsymbol{V}_{\text{JS\_code}}$ is the vector representation for a JS code and $\boldsymbol{X}_n$ is the n-gram vector representation for a word n. A JS code vector is given by the summation of the average value of each component of a word n-gram vectors divided by the total number of words in a JS code.

An implementation [33] of the FastText model is used to learn the feature vectors for a JS code producing fixed-length vectors. Labels obtained from a JS code dataset are used to train the network for the prediction tasks. The vectors are then used to classify a JS code into either malicious or benign. The FastText model uses several parameters for the learning of feature vectors such as, *sg*, which represents the skip-gram training algorithm, *word_ngrams*, which enriches word vectors with subword (n-grams) information, and *min_n* and *max_n*, which control the minimum and maximum length of n-grams respectively. Other parameters include *min_count* which ignore all words with a total frequency lower than a preset one, *size*, which represents the word vectors dimensionality, *iter* which represents the epochs over the corpus and *window*, which controls the maximum distance between the current word and the predicted one.

Fig. 6 shows the overview of DUD-preprocessing of a JS code. The figure also presents the feature learning process by the FastText model and classification of the AST-JS features by a classifier model. It is a two-stage process. During the first stage, an obfuscated JS code is formatted, unpacked, and decoded, and a JS code parser is used to perform syntactic analyses of the resultant Plain-JS code. Second, the AST-JS obtained from the JS

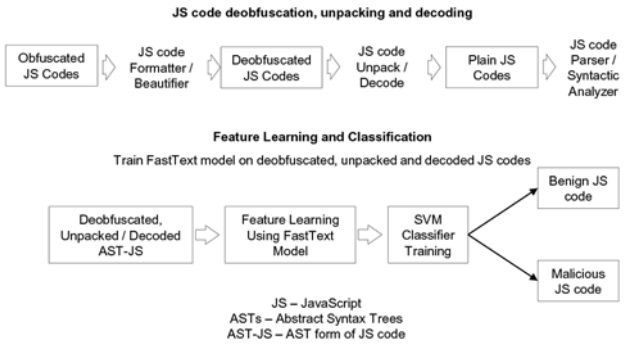

Fig. 6 *Layout for DUD-preprocessing of a JS code and the feature learning and classification for DUD-preprocessed AST-JS features*

code parser is passed to the FastText model for feature learning, and a classifier is used to judge the maliciousness of a JS code using the learned AST-JS feature vectors. Using a training set of the learned feature vectors, we perform cross-validation and classifier learning to find the optimal hyperparameters. We then compare the performance of the FastText model to the paragraph vector models, the StarSpace model, the LSTM model, and the TF-IDF model on obfuscated malicious JS code detection on the same dataset.

4 Experiments

4.1 Experimental setup

In the experimental setup, we use a dataset of 40,000 malicious JS codes and 150,000 benign JS codes. We also add 50,000 benign JS codes by crawling the Internet. The malicious JS codes were obtained separately from the Hynek Petrak's dataset [34] and the benign JS codes are from the SRILAB [35] and the Majestic Million service top 10,000 websites [36] datasets. These datasets are used for both training and test purposes. As shown in Table 1, some malicious and benign JS codes in the dataset could not be successfully parsed because of *Syntax Error*, *Unicode Decode Error*, *Attribute Error*, *Name Error*, *Index Error*, *Not Implemented Error*, *Key Error*, and *Recursion Error*.

We perform two types of feature embedding: The FastText model for regular obfuscated JS codes and the FastText model for DUD-preprocessed JS codes. We conduct feature learning using the datasets for the FastText model and compare its performance to the distributed bag of words version of the paragraph vector model (the PV-DBoW model), the distributed memory model of the paragraph vector model (the PV-DM model), the StarSpace model, the LSTM model, and the TF-IDF model. First, we conduct feature learning using regular obfuscated JS codes with input defined as AST-JS, that is, by learning feature vectors from both obfuscated benign and malicious JS codes represented as an AST-JS. Second, we conduct feature learning using DUD-preprocessed JS codes with input defined as an AST-JS, that is, by performing feature learning on code structure representation for DUD-preprocessed JS codes. We also perform AST-level merging and AST sub-tree realignment [14] on the AST-JS to enhance feature learning. This AST-level manipulation facilitates simulation of a real web setting by ensuring that the ratio of the benign AST-JS features is more compared to that of the malicious ones.

Table 1 JS code label, original JS codes and deobfuscated JS codes showing the success rate of the deobfuscation process

JS code label	Original JS codes	Deobfuscated JS codes
malicious	39,449	32,436
benign	121,702	97,606

CAAI Trans. Intell. Technol., 2020, Vol. 5, Iss. 3, pp. 184–192

The FastText model parameters such as *sg*, *size*, *window*, *min_count*, *word_ngrams*, *min_n* and *max_n* significantly affect the performance of the FastText model. We conducted experiments with the dimensions of the range 100–900 for *size*, 1–8 for *window*, 1–8 for *min_count* and 3–6 for *min_n* and *max_n*. We learned feature vectors of 200 dimensions for the word vectors, with $sg = 1$ to select the skip-gram training algorithm, $window = 8$ for the number of context words, $min\_count = 5$ for the word frequency, $word\_ngrams = 1$ to activate the use of sub-word information and $min\_n = 3$ and $max\_n = 5$ for *n*-gram minimum and maximum length, respectively. These are the parameters that we found optimal for the task of feature learning on DUD-preprocessed JS codes, for the FastText model. For performance evaluation, we used cross-validation with $k = 10$, to get ten folds of feature vectors. The folds translate to ten iterations, divided into nine folds for classifier training and one fold for model evaluation. For feature vectors classification, we used SVM with $kernel = linear$ and $C = 1$ parameters. SVM is a classification method for regression and classification. The algorithm is mostly used as a binary classifier and it constructs a hyper-plane in high dimensional space [37, 38]. For this, we computed precision, recall, and *F*1-score. Precision is ability of a classifier not to identify a benign JS code as malicious and recall is the classifiers ability to identify all malicious JS codes. *F1*-score can be interpreted as a weighted harmonic mean of precision and recall [38]. Precision, recall and *F1*-score are computed as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{7}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{8}$$

$$F1\text{−score} = 2\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}, \tag{9}$$

where TP is the number of malicious JS codes correctly classified as malicious, TN is the number of benign JS codes correctly classified as benign, FN is the number of malicious JS codes classified as benign, and FP is the number of benign JS codes classified as malicious.

For performance visualisation, we used a receiver operating characteristic (ROC) graph plotting the true positive rate against the false positive rate. We calculated area under the curve (AUC) score [38] for the ROC to determine the model performance. We compared the difference in performance between our approach and other feature learning approaches: the PV-DBoW model ($vector\_size = 200$, $window = 8$, $min\_count = 5$), the PV-DM model ($dm = 1$, $vector\_size = 200$, $window = 8$, $min\_count = 5$), the StarSpace model, the LSTM model, and the TF-IDF model.

4.2 Performance comparisons

Precision, recall, and F1-score: The results of the experiments are presented in this section. Table 2 presents the performance of the FastText model, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model when performing feature embedding for regular obfuscated JS codes with input defined as AST-JS similar to the work by Ndichu *et al.* [14]. As shown in the table, the models can detect regular obfuscated malicious JS codes with an error rate of ∼0.0489, 0.0717, 0.0649, 0.0892, 0.1205, and 0.1657 in recall for the FastText model, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model, respectively. AST can bypass obfuscation. Therefore, this approach can detect regular obfuscated JS codes that have similar syntactic and semantic characteristics as the original ones at an abstract level. However, malicious JS codes employ methods such as packing, encoding, and multi-layered obfuscation methods which results in complex codes that hinder the detection of JS-based attacks by signature, heuristics, and machine learning approaches. The performance of paragraph vector models in Table 2 is far from perfect as this would result in many misclassified obfuscated malicious JS codes which translates to multiple successful attacks.

Table 2 Precision, recall, and *F*1-score representing the performance of the FastText model, the paragraph vector models, the StarSpace model, the LSTM model, and the TF-IDF model on a regular obfuscated JS code dataset

Model	Precision	Recall	*F*1-score
FastText	0.9427 (±0.1005)	0.9511 (±0.0477)	0.9459 (±0.0528)
PV-DBoW	0.9412 (±0.1571)	0.9283 (±0.1159)	0.9331 (±0.0868)
PV-DM	0.9313 (±0.1675)	0.9351 (±0.1016)	0.9289 (±0.0873)
StarSpace	0.9160 (±0.0192)	0.9108 (±0.0193)	0.9098 (±0.0121)
LSTM	0.8601 (±0.0155)	0.8795 (±0.0421)	0.8552 (±0.1082)
TF-IDF	0.8471 (±0.0112)	0.8343 (±0.0521)	0.8317 (±0.1325)

PV-DBOW model stands for the distributed bag of words version of paragraph vector, the PV-DM model stands for the distributed memory model of paragraph vectors, the LSTM model stands for the long short-term memory model, and the TF-IDF model stands for term frequency–inverse document frequency model.

Table 3 presents precision, recall, and *F*1-score for the FastText model, the paragraph vector models, the StarSpace model, the LSTM model, and the TF-IDF model trained using a deobfuscated JS code dataset with input defined as an AST-JS and includes the standard deviation. The models achieved an above-average performance with precision, recall, and *F1*-score being above 50%. The FastText model with $sg = 1$ and 200 dimensions of word vectors achieved superior results compared to the paragraph vector models. The FastText model outperformed the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model with 0.0110, 0.0163, 0.0228, 0.0538, and 0.0963 in precision, 0.0041, 0.0099, 0.0502, 0.0769, and 0.1078 in recall and 0.0056, 0.0108, 0.0373, 0.0852, and 0.1124 in *F*1-score, respectively. Compared to the other models, the TF-IDF model achieved the lowest performance for the task of obfuscated malicious JS code detection while applying JS code formatting. Consequently, the FastText model resulted in less false positives and false negatives compared to the other models.

The objective of this study is to improve the performance of machine learning models on the detection of obfuscated malicious JS codes. A model with a high recall rate is desired to achieve this objective. Such a model would result in a few misclassified obfuscated malicious JS codes, that is, low false-negative. JS code deobfuscation results in a formatted code and can do away with only the simple forms of obfuscation. This property explains the reason why there is only a slight improvement between the models trained using regular obfuscated JS codes and those trained using a deobfuscated JS code dataset of ∼0.0115 for the FastText model, 0.0302 for the PV-DBoW model, 0.0176 for the PV-DM model, 0.0016 for the StarSpace model, 0.0062 for the LSTM model, and 0.0205 for the TF-IDF model in the recall. The input for the models in both cases is defined as an AST-JS. The best performing model when using a deobfuscated JS code dataset, the

Table 3 Precision, recall, and *F*1-score obtained by feature learning using a deobfuscated JS code dataset

Model	Precision	Recall	*F*1-score
FastText	0.9498 (±0.0270)	0.9626 (±0.0197)	0.9531 (±0.0288)
PV-DBoW	0.9388 (±0.0498)	0.9585 (±0.0529)	0.9475 (±0.0402)
PV-DM	0.9335 (±0.0372)	0.9527 (±0.0407)	0.9423 (±0.0272)
StarSpace	0.9270 (±0.0216)	0.9124 (±0.0537)	0.9158 (±0.0410)
LSTM	0.8960 (±0.0441)	0.8857 (±0.0386)	0.8679 (±0.0314)
TF-IDF	0.8535 (±0.0110)	0.8548 (±0.0193)	0.8407 (±0.0650)

PV-DBOW model stands for the distributed bag of words version of paragraph vector, the PV-DM model stands for the distributed memory model of paragraph vectors, the LSTM model stands for the long short-term memory model and the TF-IDF model stands for term frequency–inverse document frequency model.

FastText model, has an error rate of ∼0.0374 in the recall. Even though the FastText model is the best performing model here, a lot of obfuscated malicious JS codes will go undetected, which translates to multiple successful attacks. The failure in detection is because sections of the obfuscated malicious JS codes that employ sophisticated obfuscation techniques cannot be easily solved by deobfuscation. Therefore, further processing of the deobfuscated malicious JS codes is needed to realise improved performance.

Table 4 presents precision, recall, and *F1*-score for the FastText model, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model obtained with AST-JS features by feature learning using an unpacked JS code dataset. The FastText model achieves an improvement of over 2.78 and 2.07% in precision, 2.09 and 0.94% in the recall, and 2.54 and 1.82% in *F*1-score, when compared to the same model, trained using a regular obfuscated JS code dataset and a deobfuscated JS code dataset, respectively. The resultant code structure can explain the improvement in performance for the FastText model after deobfuscation and unpacking. Deobfuscation results in a well-formatted JS code, which facilitates the feature learning process of the model. When the unpacking is done for the deobfuscated JS code dataset, the complex code structure which remains after deobfuscation is simplified away. Therefore, the model can learn better features from a deobfuscated and unpacked JS code dataset compared to a regular obfuscated or a deobfuscated JS code dataset.

Table 5 presents precision, recall, and *F*1-score for the FastText model, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model obtained with AST-JS features by feature learning using a DUD-preprocessed JS code dataset. The FastText model, the best performing model, achieves an improvement of over 5.21, 4.5, and 2.43% in precision, 4.2, 3.05 and 2.11% in recall, and 4.14, 3.42 and 1.6% in *F*1-score when compared to the same model trained using a regular obfuscated JS code dataset, a deobfuscated JS code dataset, and an unpacked JS code dataset, respectively. The performance for the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model in this task is also improved. However, the FastText model achieves the highest recall rate for the task of obfuscated malicious JS code detection. This is because deobfuscation, unpacking, and decoding results in a Plain-JS code which has JS code transformations simplified away. Therefore, the resultant JS code provides better AST-JS features during feature learning.

True positive and false positive rate: The improved performance of the FastText model on the task of detection of obfuscated JS codes is also shown in Fig. 7*a*, which presents the ROC graph and AUC results for the model. As shown in Fig. 7*a*, the model achieved high and low rates of true positives and false positives, respectively, compared to the PV-DBoW model, which is presented in Fig. 7*b*, the PV-DM model, which is presented in Fig. 7*c*, the StarSpace model, which is presented in Fig. 7*d*, the LSTM model, which is presented in Fig. 7*e*, and the TF-IDF model, which is presented in Fig. 7*f*. The PV-DBoW model and the PV-DM model achieved the same AUC score. However, the latter model achieved the least AUC score compared to the PV-DBoW during cross-validation, as shown by sections of its ROC curve. The StarSpace model, the LSTM model, and the TF-IDF model achieved the lowest AUC scores with sections of the curves in Figs. 7*d–f* going below the performance threshold. The coloured curves represent the performance of the model for each fold during cross-validation. The paragraph vector models: the PV-DBoW and the PV-DM models ignore the internal structure of words. The FastText model takes into account a word's subword information. The *n*-gram word vectors learned by the FastText model during feature learning are shared across words. This property enables the FastText model to learn better and reliable vector representations for the DUD-preprocessed JS code dataset. Using a trained FastText model, words that do not exist in the model's vocabulary can be inferred from the learned character *n*-gram vectors. The model also results in low dimensional representations for the learned *n*-gram word vectors.

Table 4 Precision, recall, and *F*1-score obtained by feature learning using an unpacked JS code dataset

Model	Precision	Recall	*F*1-score
FastText	0.9705 (±0.0092)	0.9720 (±0.0257)	0.9713 (±0.0235)
PV-DBoW	0.9617 (±0.0087)	0.9574 (±0.0411)	0.9592 (±0.0224)
PV-DM	0.9474 (±0.0399)	0.9468 (±0.0457)	0.9460 (±0.0276)
StarSpace	0.9363 (±0.0339)	0.9226 (±0.0366)	0.9460 (±0.0542)
LSTM	0.9097 (±0.0270)	0.9082 (±0.0444)	0.9178 (±0.0457)
TF-IDF	0.8858 (±0.0237)	0.8729 (±0.0488)	0.8691 (±0.0336)

PV-DBOW model stands for the distributed bag of words version of paragraph vector, the PV-DM model stands for the distributed memory model of paragraph vectors, the LSTM model stands for the long short-term memory model, and the TF-IDF model stands for term frequency–inverse document frequency model.

Table 5 Precision, recall, and *F*1-score obtained by feature learning using a deobfuscated, unpacked and decoded (DUD-preprocessed) JS code dataset

Model	Precision	Recall	F1-score
FastText	0.9948 (±0.0045)	0.9931 (±0.0070)	0.9873 (±0.0152)
PV-DBoW	0.9839 (±0.0046)	0.9841 (±0.0053)	0.9801 (±0.0060)
PV-DM	0.9837 (±0.0084)	0.9802 (±0.0121)	0.9819 (±0.0132)
StarSpace	0.9571 (±0.0199)	0.9584 (±0.0204)	0.9628 (±0.0161)
LSTM	0.9170 (±0.0297)	0.9332 (±0.0360)	0.9234 (±0.0327)
TF-IDF	0.9021 (±0.0352)	0.9009 (±0.0266)	0.9042 (±0.0366)

PV-DBOW model stands for the distributed bag of words version of paragraph vector, the PV-DM model stands for the distributed memory model of paragraph vectors, the LSTM model stands for the long short-term memory model and the TF-IDF model stands for term frequency–inverse document frequency model.

5 Discussion

One of the objectives of obfuscation is to hinder a JS code reverse engineering. Therefore, the resultant Plain-JS codes after performing DUD-preprocessing for the obfuscated JS codes may not result in the same original JS code as before JS code obfuscation was applied. However, as part of our findings in this research, an obfuscated JS code that has been DUD-preprocessed results in regular Plain-JS code. The detection performance for the machine learning models is improved because the resultant JS code has the same semantic and syntactic properties as the original one. After performing deobfuscation, which is the first step in our obfuscated JS code preprocessing, we have to inspect the resultant JS codes for the presence of obfuscation manually. Some approaches have been proposed for the automatic detection of obfuscated JS codes. Some of these approaches focus on specific JS code obfuscator tools that are unique to specific obfuscation techniques. Therefore, given a dataset that contains JS codes obfuscated by a JS code obfuscator tool other than the one focused in the approach would result in misclassification of obfuscated JS codes. The existing detection methods for obfuscated JS code detection cannot be easily applied to other JS code obfuscator tools. Other studies use features such as the structure of the obfuscated JS code and functions such as *eval()* and *unescape()* within a *x* number of bytes of each other. These are heuristic-based approaches that are limited in capability when faced with sophisticated JS code obfuscation techniques. In the case of a packed JS code, single-step unpacking is not enough as the majority of the obfuscated malicious JS codes contain multiple layers of obfuscation.

To address this issue, as future work, (i) there is a need to develop a technique to check for obfuscation in the dataset automatically. This technique should remain independent of specific JS code obfuscator tools or obfuscation techniques. (ii) Recently, studies are focusing on embedding on source code. Both regular and

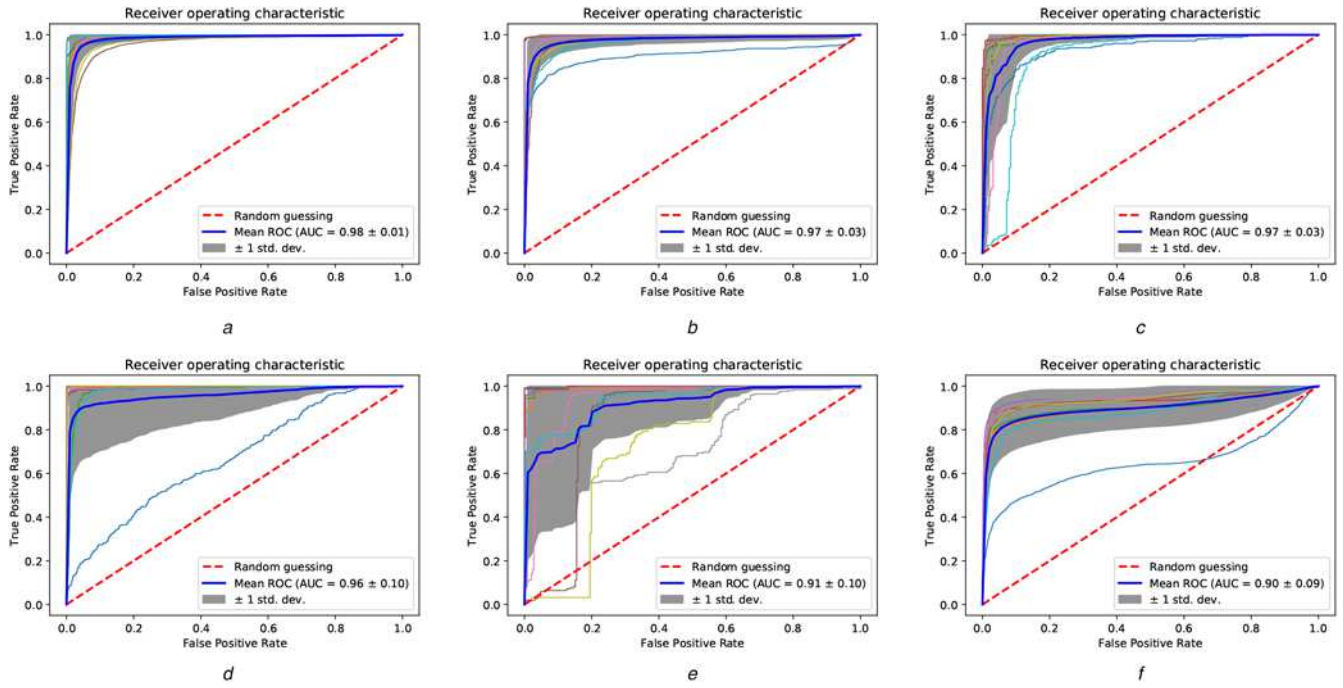

Fig. 7 *ROC curves and AUC that represent the performance of the FastText, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model on obfuscated malicious JS codes detection with the true positive rate plotted against the false positive rate. The dashed diagonal line represents the performance threshold as random guessing. The coloured curves represent the performance of the each model for each fold during cross-validation. Also included is the mean AUC score for each model*

a FastText,
b PV-DBoW,
c PV-DM,
d StarSpace,
e LSTM,
f TF-IDF

obfuscated JS codes follow some code structure rules for sentences, functions, variables, operation procedures, and stop conditions. As such, a word embedding model may lose much of the information in a JS code structure. Embedding on JS source code would be language-specific and share the same concept as the NLP embedding models with a few tweaks. (iii) While the existing word embedding models, such as the FastText and the paragraph vector models, yield high-quality vector representations on programs and source code tasks such as obfuscated malicious JS code detection, they are not well-tailored for obfuscated JS code parts. As such, parsing errors are common when trying to realise a Plain-JS code for static analysis. It is envisioned that directly embedding a JS source code would facilitate the capturing of a JS code structure and avoid such errors.

6 Conclusion

In this paper, we propose DUD-preprocessing of obfuscated malicious JS codes and feature learning using the FastText model. The NLP models have previously achieved good results and resulted in high-quality vector representations for words, sentences, and paragraphs. Consequently, the models have been used for malicious JS code detection using features such as Plain-JS and AST-JS. However, the models struggle to learn useful representations when faced with obfuscated JS codes as the JS codes or sections of the JS code do not maintain appropriate code structure. One expectation is that DUD-preprocessing of the obfuscated JS codes would result in improved performance for machine learning models for the task of obfuscated malicious JS code detection and enable detection of JS codes transformed using sophisticated obfuscation techniques. We have conducted experiments using feature embedding for regular obfuscated JS codes and DUD-preprocessed JS codes. We compared the performance of the FastText model to the paragraph vector models: the PV-DBoW model and the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model on the detection of obfuscated malicious JS codes. We analyse the resultant JS code after DUD-preprocessing to check for the presence of obfuscation. This further processing is essential to simplify away sophisticated JS code obfuscation techniques such as JS code multi-layer obfuscations. We use AST-JS for code structure representation of the resultant Plain-JS code and realise an improvement in performance for the models used for feature learning. AST-JS makes the structural relationship of elements within a JS code distinct while at the same time preserving all the relevant information of a JS code. Our proposed approach provides improved detection performance for the task of obfuscated malicious JS codes contents detection compared to feature learning on regular obfuscated JS code. Besides, there is considerable improvement in performance compared to our previous work [14]. The FastText model is fast and suitable for big datasets and, therefore, well suited for the JS code feature learning. The model can also learn vector representations for JS code words that are not contained in the vocabulary at the time of model training and would, therefore, result in reliable vector representations for future, new and unseen JS codes.

As future work, there is a need for a study on obfuscated URLs and their properties. URLs in JS codes are heavily obfuscated. The obfuscation is even more complicated for URLs in obfuscated malicious JS codes as attackers endeavour to hide their attack methods such as URL redirection attacks to attackers untrusted websites. Such websites employ insecure protocols that would be effortlessly flagged down by implemented security controls such as the firewall. URL obfuscation is stealthily and deliberately applied to evade such detection. A malicious URL detection system must be developed to realise a comprehensive security system against web-based attacks.

7 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.

8 References

[1] Curtsinger, C., Livshits, B., Zorn, B., *et al.*: 'ZOZZLE: fast and precise in-browser JavaScript malware detection'. Proc. of USENIX Security Symp., Berkeley, CA, USA, 2011, pp. 33–48

[2] Howard, F.: 'Malware with your mocha: obfuscation and anti-emulation tricks in malicious JavaScript', Sophos Lab, September 2010, pp. 1–18

[3] Kaplan, S., Livshits, B., Zorn, B., *et al.*: ''NOFUS: Automatically Detecting' + String.fromCharCode(32) + 'ObFuSCateD' .toLowerCase () + 'JavaScript Code''. Microsoft Research Technical Report, MSR-TR-2011, (57), 2011, pp. 1–11

[4] Sebastian, S., Malgaonkar, S., Shah, P., *et al.*: 'A study and review on code obfuscation'. World Conf. on Futuristic Trends in Research and Innovation for Social Welfare (WCFTR'16), Coimbatore, Tamilnadu, India, 2016, pp. 1–6

[5] Schrittwieser, S., Katzenbeisser, S., Kinder, J., *et al.*: 'Protecting software through obfuscation: can it keep pace with progress in code analysis?', *ACM Comput. Surv.*, 2016, **49**, (1), pp. 4:1–4:40

[6] Xu, W., Zhang, F., Zhu, S.: 'The power of obfuscation techniques in malicious JavaScript code: a measurement study'. 7th Int. Conf. on Malicious and Unwanted Software (MALWARE), Fajardo, PR, USA, 2012, pp. 9–16

[7] Lu, G., Debray, S.: 'Automatic simplification of obfuscated JavaScript code: a semantics-based approach'. Proc. of the IEEE 6th Int. Conf. on Software Security and Reliability, Gaithersburg MD, USA, 2012, pp. 31–40

[8] JavaScript Obfuscator: 'JavaScript obfuscator is a powerful encoding, and obfuscation technologies prevent reverse engineering, copyright infringement and unauthorized modification of your code'. Available at https://javascriptobfuscator.com/Javascript-Obfuscator.aspx, accessed on August 2019

[9] Serafim, T., Kachalov, T.: 'JavaScript obfuscator tool', A free and efficient obfuscator for JavaScript (including ES2017) – a Web UI tool to the excellent (and open source) javascript-obfuscator@0.21.0. Available at https://obfuscator.io/, accessed on August 2019

[10] DirtyMarkup: 'JavaScript beautifier'. Available at http://www.dirtymarkup.com/, accessed on November 2019

[11] Lielmanis, E., Newman, L.: 'Online JavaScript beautifier (v1.10.2), beautify, unpack or deobfuscate JavaScript and HTML, make JSON/JSONP readable'. Available at https://beautifier.io/, accessed on November 2019

[12] Dan's Tools: 'JavaScript viewer, beautifier, formatter and editor'. Available at https://www.cleancss.com/javascript-beautify/, accessed on November 2019

[13] Ndichu, S., Ozawa, S., Misu, T., *et al.*: 'A machine learning approach to malicious JavaScript detection using fixed length vector representation'. Proc. of 2018 Int. Joint Conf. on Neural Networks, (IJCNN'18), Rio de Janeiro, Brazil, 8–13 July 2018, pp. 1–8

[14] Ndichu, S., Kim, S., Ozawa, S., *et al.*: 'A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors', *Appl. Soft Comput. J.*, 2019, **84**, (105721), pp. 1–11

[15] Skolka, P., Staicu, C., Pradel, M.: 'Anything to hide? Studying minified and obfuscated code in the web'. the Proc. of the World Wide Web Conf. (WWW'19), San Francisco, CA, USA, 2019, vol. 4, pp. 1–11

[16] Yadegari, B., Johannesmeyer, B., Whitely, B., *et al.*: 'A generic approach to automatic deobfuscation of executable code'. the Proc. of the 36th IEEE Symp. on Security and Privacy (IEEE SP'15), San Jose, CA, USA, 2015, pp. 674–691

[17] Gorji, A., Abadi, M.: 'Detecting obfuscated JavaScript malware using sequences of internal function calls'. the Proc. of the 52nd ACM Southeast Regional Conf. (ACMSE'14), Kennesaw, GA, USA, 2014, pp. 1–6

[18] Tellenbach, B., Paganoni, S., Rennhard, M.: 'Detecting obfuscated JavaScript from known and unknown obfuscators using machine learning', *Int. J. Adv. Secur.*, 2016, **9**, (3–4), pp. 196–206

[19] Fass, A., Krawczyk, R., Backes, M., *et al.*: 'JaSt: fully syntactic detection of malicious (obfuscated) JavaScript'. the Proc. of the 15th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Saclay, France, 2018, pp. 303–325

[20] Pan, J., Mao, X.: 'Obfuscated malicious JavaScript detection by machine learning'. 2nd Int. Conf. on Advances in Mechanical Engineering and Industrial Informatics (AMEII'16), Hangzhou, Zhejiang, 2016, pp. 805–810

[21] Graziano, M., Balzarotti, D., Zidouemba, A.: 'ROPMEMU: a framework for the analysis of complex code-reuse attacks'. Proc. of the 11th Asia Conf. on Computer and Communications Security (ASIA CCS'16), Xi'an, China, 2016, pp. 47–58

[22] Hu, X., Cheng, Y., Duan, Y., *et al.*: 'JSForce: a forced execution engine for malicious JavaScript detection'. Proc. of the 13th Int. Conf. on Security and Privacy in Communication Networks (SecureComm'17), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Niagara Falls, Canada, 2017, vol. 238, pp. 704–720

[23] Python: 'Python 2.7.17 documentation, the python standard library'. Available at https://docs.python.org/2/library/functions.html, accessed on August 2019

[24] Zhang, Y., Jin, R., Zhou, Z.-H.: 'Understanding bag-of-words model: a statistical framework', *Int. J. Mach. Learn. Cybern.*, 2010, **1**, (1–4), pp. 43–52

[25] Goldberg, Y.: 'Neural network methods for natural language processing', 'Synthesis lectures on human language technologies', vol. 37 (Morgan and Claypool, San Rafael, CA, USA, 2017), no. 69, pp. 1–309

[26] Mikolov, T., Sutskever, I., Chen, K., *et al.*: 'Distributed representations of words and phrases and their compositionality'. Proc. of the Advances in Neural Information Processing Systems (NIPS), Lake Tahoe, Nevada, USA, 2013b, vol. 26, pp. 3111–3119

[27] Mikolov, T., Chen, K., Corrado, G., *et al.*: 'Efficient estimation of word representations in vector space'. the Int. Conf. on Learning Representations (ICLR), Workshop Papers, Scottsdale, AZ, USA., 2013a, pp. 1–12

[28] Le, Q., Mikolov, T.: 'Distributed representations of sentences and documents'. Proc. of the 31st Int. Conf. on Machine Learning (ICML-14), Beijing, China, 2014, pp. 1188–1196

[29] Dai, A., Olah, C., Le, Q.: 'Document embedding with paragraph vectors'. Neural Information Processing Systems (NIPS), Deep Learning Workshop, Palais des Congrès de Montréal, 2014, pp. 1–8

[30] Bojanowski, P., Grave, E., Joulin, A., *et al.*: 'Enriching word vectors with subword information', *Trans. Assoc. Comput. Linguist.*, 2017, **5**, pp. 135–146, arXiv preprint 27 arXiv:1607.04606

[31] Joulin, A., Grave, E., Bojanowski, P., *et al.*: 'Bag of tricks for efficient text classification'. Proc. of the 15th Conf. of the European Chapter of the Association for Computational Linguistics (EACL), Short Papers, Valencia, Spain, 2017, pp. 427–431

[32] Wang, B., Wang, A., Chen, F., *et al.*: 'Evaluating word embedding models: methods and experimental results', *APSIPA Trans. Signal Inf. Process.*, 2019, **E19**, (8), pp. 1–13, arXiv:1901.09785

[33] Radim, R., Sojka, P.: 'Software framework for topic modelling with large corpora'. Proc. of the Int. Conf. on Language Resources and Evaluation (LREC'10), Workshop on New Challenges for NLP Frameworks, Valletta, Malta, 2010, pp. 45–50

[34] Petrak, H.: 'JavaScript malware collection – a collection of almost 40.000 JavaScript malware samples'. Available at https://github.com/HynekPetrak/javascript-malwarecollection, accessed on August 2019

[35] Raychev, V., Bielik, P., Vechev, M., *et al.*: 'Learning programs from noisy data'. Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, (POPL'16), New York, NY, USA, 2016, pp. 761–774

[36] The Majestic Million Service: 'The million domains we find with the most referring subnets'. Available at https://majestic.com/reports/majestic-million, accessed on August 2019

[37] Burges, C.J.C.: 'A tutorial on support vector machines for pattern recognition', *Data Min. Knowl. Discov.*, 1998, **2**, (2), pp. 121–167

[38] Pedregosa, F., Varoquaux, G., Gramfort, A., *et al.*: 'Scikit-learn: machine learning in python', *J. Mach. Learn. Res.*, 2011, **12**, pp. 2825–2830