Deep Neural Networks for Malicious JavaScript Detection Using Bytecode Sequences

Muhammad Fakhrur Rozi
Graduate School of Engineering
Kobe University
Kobe, Japan
rozi_mahfud@stu.kobe-u.ac.jp

Sangwook Kim
Graduate School of Engineering
Kobe University
Kobe, Japan
kim@eedept.kobe-u.ac.jp

Seiichi Ozawa
Center for Mathematical and Data Sciences
Kobe University
Kobe, Japan
ozawasei@kobe-u.ac.jp

***Abstract*—JavaScript is a dynamic computer programming language that has been used for various cyberattacks on client-side web applications. Malicious behaviors in JavaScript are injected on purpose as the outputs of web applications, such as redirection and pop-up texts or images. It exploits vulnerabilities by using a variety of methods such as drive-by download or cross-site scripting. To protect users from such cyberattacks, we propose a deep neural network for detecting malicious JavaScript codes by examining their bytecode sequences. We use the V8 JavaScript compiler to generate a bytecode sequence, which corresponds to an abstract form of machine codes. The benefit of using bytecode representation is that we can easily break complex obfuscation in JavaScript. To identify the attacker's malicious intention, We adopt a deep pyramid convolutional neural network (DPCNN) combining with recurrent neural network models, which can handle long-range associations in a bytecode sequence. In our experiment, various recurrent networks are testified to encode temporal features of code behaviors, and our results show that the proposed approach provides high accuracy in detection of malicious JavaScript.**

***Index Terms*—Cybersecurity, Deep learning, Malicious JavaScript detection**

I. Introduction

JavaScript has become the most used client-side programming language that gives many conveniences to develop a web application [1]. It is prevalent because it is lightweight, flexible, and powerful. However, some attacks use a JavaScript code to exploit vulnerabilities in a server, plugin, and other systems. For instance, cross-site scripting (XSS) [2], drive-by-download [3], heap spraying attack [4], or any kind of attack that exploits browser, cookies, and security permission to act. XSS is one of the common ways to exploit the vulnerabilities of web applications using JavaScript. It tries to inject scripting code into the output of the applications that are then sent to the user's web browser [5]. The malicious-injected code is executed and used to access sensitive data stored and transfers it to a server under the attacker's control. This code is camouflaged in some parts of benign web page such as image, text, pop-up, and button.

Benign JavaScript often utilizes obfuscation to protect code privacy or intellectual property and compress the code to make human unreadable code without downgrading the execution performance [6]. On the other hand, malicious JavaScript applies obfuscation to hide known exploits and to evade anti-virus systems [7]. Some obfuscation techniques can be applied to transform a JavaScript code to become unreadable. Those techniques succeed in making malicious a JavaScript code difficult to be recognized by the anti-virus system, yet mixed and multi-level obfuscation are often used to make it more complicated for the system to de-obfuscate the code.

Therefore, machine learning and deep learning exist to address the obfuscation problem for detecting malicious a JavaScript code. Many researchers have proposed using a wide variety of machine learning classifiers such as logistic regression [8], neural networks [9], and the others. Feature extraction takes a significant role when using deep learning model to detect maliciousness of a JavaScript code. According to Ucci et al. [10], there are three kinds of feature extraction approaches that are used in malware analysis, e.g., static analysis, dynamic analysis, and hybrid analysis. Approaches based on static analysis focuses on static features of a JavaScript code such as string code or any content of samples without requiring the execution. Meanwhile, dynamic analysis is the contrary of static analysis, which uses the execution of code to grab the features by using a virtual machine or Sandbox to isolate malicious code in a safe environment. Other than that, the hybrid analysis uses the combination of static and dynamic analysis, which takes advantage of both of them. Static analysis has faster performance than others due to no execution is needed to extract the feature, while dynamic and hybrid analysis need more time to execute the code first before the data can be used to build a machine learning model.

For addressing obfuscation problem, dynamic analysis approach leverages more detailed information related to the target as compared to static analysis even though it takes more time in runtime execution. By executing the code in a safe and reliable environment, output such as opcode, log activity, or byte code will be produced, which represents the instruction code of the program even though the code contains obfuscation part. One of the features from the JavaScript engine that we can use is bytecode. JavaScript uses an engine to compile high-level programming language before it is executed in the machine in order to make a JavaScript code run faster. Bytecode is a sequence of abstract instruction codes that represent the whole process of the program without any redundancy. Compiling bytecode to machine codes will be easier because the bytecode was designed with the same computational model as the physical CPU.

In this work, we use the bytecode sequence as a feature of JavaScript code to detect whether it is malicious or benign. Since a bytecode sequence is obtained only through the execution of JavaScript, we build a safe virtual environment to extract bytecode from a JavaScript code. For this work, we use V8 engine from Google to compile a high-level JavaScript code into bytecode sequence which is similar to work by Fang et al. [11] where jsdom package library of NodeJS is needed to address some of DOM and BOM object in the browser environment that cannot be identified. By using this feature, we do not need the de-obfuscation process to reveal the real source code of the program. We just focus on the analysis of the sequence to capture the process of the program. The bytecode sequence can be a very long and high-frequency sequence that can consist of more than 200,000 codes for one program. Due to this condition, we use Deep Pyramid Convolutional Neural Networks (DPCNN) [12], which deals with long sequence problems by reducing the length of feature to become some of the feature maps. This approach provides considerable performance for detecting malicious JavaScript compared to other approaches. We use the recurrent network at the end of DPCNN in order to get a long-range association of the sequence. Some modifications are applied to DPCNN to fit with the recurrent network.

The organization of this paper is as follows, we highlight the related work in Section 2. After that, we explain our proposed deep learning model in Section 3. We present our experiment result through a comparison to some combination of deep learning methods in Section 4 and Section 5. Lastly, we deliver our conclusions in Section 6.

II. Related work

Cyberattacks with a malicious JavaScript code basis have evolved over the last recent years. Many approaches have been developed to encounter attacks such as signature-based analysis, string pattern analysis [13], and machine learning or deep learning. The approach may depend on the feature that we want to analyze. Some kinds of features can be considered for the analysis of a malware or malicious code based on how the features can be extracted, e.g., static analysis, dynamic analysis, and hybrid analysis.

The approach based on static analysis catches the feature of samples by using the content without necessitating their execution [10]. Many researchers use a static feature, which is faster and easier for analysis. For example, Rafiqul Islam et al. [14] used static features such as function length frequency and printable string information to classify malware data set. They made a vector representation of each feature by counting the number of functions and strings in the file program. Other researchers used plain JavaScript source code and then transformed it into a hexadecimal byte sequence for each character [15]. Furthermore, there is research that used Abstract Syntax Tree (AST) representation for the static features [16] that contains more information than lexical units, and it was able to analyze samples whose behavior is time- or environment-dependent.

Meanwhile, many researchers conducted a dynamic analysis for feature extraction rather than static analysis. These approaches are commonly used due to eliminating obfuscation of the source code by executing the code in a virtual environment such as Sandbox and emulator. Anderson et al. [17] used graphs which were constructed by dynamically collected instruction traces of the target executable. They combined graph kernels to make a similarity matrix between the instructions trace graphs. Besides that, Kawaguchi et al. [18] proposed a method to classify malware's functions from APIs observed by dynamic analysis on the host. They tried to address a threat from malware proliferation, which subspecies of existing malware that have been automatically generated by illegal tools. Despite dynamic analysis, we can use hybrid analysis, which is the combination of dynamic and static analysis for feature extraction. The purpose of using both techniques to avoid obfuscation and execution-stalling [19].

More recently, methods of applying deep learning to malware or malicious JavaScript detection have been proposed. Fang et al. [11] proposed malicious JavaScript classification by applying Long Short-Term Memory (LSTM) to a bytecode sequence from the V8 JavaScript engine. It reported that it showed higher accuracy than the previous methods that had been proposed, i.e., random forest, support vector machine (SVM), and Naïve Bayes. Rhode et al. [20] had introduced similar work that used a Recurrent Neural Network (RNN) as a deep learning model with the feature was a short snapshot of behavioral data. Moreover, Stokes et al. [15] proposed the modification of LSTM by combining it with the Max Pooling layer (LaMP). They also came up with the Convoluted Partitioning of Long Sequences (CPoLS) to address the long sequence problem from byte sequence representation. Differently, Yakura et al. [21] used the combination of Convolutional Neural Network (CNN) and attention mechanism to the imaged binary data. The attention mechanism improves the performance of the CNN model by selecting the essential features of the sample and gave significant performance compared with conventional methods. Zhang et al. [9] used Residual Network (ResNet), which is one of the deep learning models that apply a deep convolutional network to overcome long sequence problems in an opcode feature.

As bytecode or opcode sequence is frequently used as the feature component of the samples, the deep learning model with the capability of addressing a long sequence is needed. To deal with the long-range problem in sequential data is quite challenging. We need to accurately capture the dependencies between symbols or codes that are far apart in the sequence. Commonly, sequential inputs are processed using RNNs. However, vanishing gradient problem is a big issue when training RNNs. Therefore, the feed-forward convolution network is highly useful to process sequential data. Some researches combine convolution and recurrent approaches to deal with the long-range problem in sequential data. Johnson et al. [12] proposed DPCNN architecture for text categorization

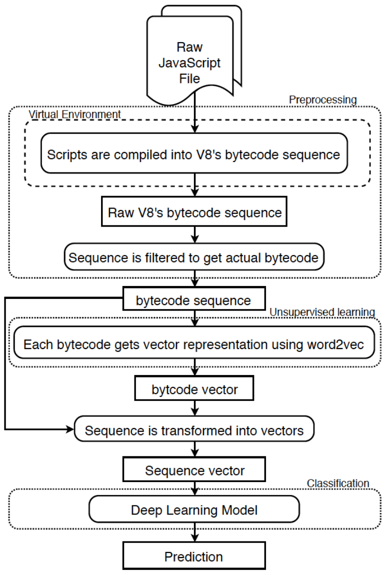

Raw JavaScript File
Preprocessing
Virtual Environment
Scripts are compiled into V8's bytecode sequence
Raw V8's bytecode sequence
Sequence is filtered to get actual bytecode
bytecode sequence
Unsupervised learning
Each bytecode gets vector representation using word2vec
bytecode vector
Sequence is transformed into vectors
Sequence vector
Classification
Deep Learning Model
Prediction

Fig. 1. Overview of our proposed method.

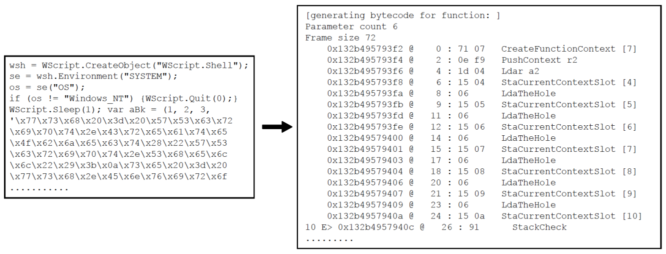
that can efficiently represent long-range associations in text. The critical feature of DPCNN is the downsampling layer without increasing the number of feature maps, which make this architecture shape like '*pyramid*.' These methods can be applied in other data types that are similar to the characteristics of text, such as sequences. Furthermore, Stokes et al. [15] used CPoLS, which is a neural network architecture, to get information hidden deep within a long sequence. They split the input sequence into smaller parts of fixed-length, processed them separately, and then combined them again for further learning. These models can perform on extremely long sequences and can learn a single vector representation of the input.

III. Proposed method

Our full classification framework is illustrated in Fig. 1, which consists of three main components. The first component is a preprocessing step that we use a V8 engine to compile the function of a program into the sequence of V8's bytecodes. From this bytecode sequence, we reduce the unimportant features of the bytecode. After that, we implement unsupervised learning for our code to get the vector representation of the code using the word2vec method [22]. Finally, the vector representation of each code can be used as the input for our deep learning model.

A. V8's Bytecode Generation

The V8 engine is an open-source program provided by Google with a high-performance JavaScript and WebAssembly engine [23]. This engine is used in Chrome and Node.js, which can run standalone or be embedded in C++ applications. It compiles a JavaScript source code into bytecodes through the parse of Abstract Syntax Tree (AST), which is a tree representation of the syntactic structure of JavaScript code. JavaScript is internally compiled by V8 with just-in-time (JIT) compilation to speed up the execution.



```
wsh = WScript.CreateObject("WScript.Shell");
se = wsh.Environment("SYSTEM");
os = se("OS");
if (os != "Windows_NT") {WScript.Quit(0);}
WScript.Sleep(1); var aBk = (1, 2, 3,
'\x77\x73\x68\x20\x3d\x20\x57\x53\x63\x72
\x69\x70\x74\x2e\x43\x72\x65\x61\x74\x65
\x4f\x62\x6a\x65\x63\x74\x28\x22\x57\x53
\x63\x72\x69\x70\x74\x2e\x53\x68\x65\x6c
\x6c\x22\x29\x3b\x0a\x73\x65\x20\x3d\x20
\x77\x73\x68\x2e\x45\x6e\x76\x69\x72\x6f
...........
```

```
[generating bytecode for function: ]
Parameter count 6
Frame size 72
    0x132b495793f2 @    0 : 71 07   CreateFunctionContext [7]
    0x132b495793f4 @    2 : 0e f9   PushContext r2
    0x132b495793f6 @    4 : 1d 04   Ldar a2
    0x132b495793f8 @    6 : 15 04   StaCurrentContextSlot [4]
    0x132b495793fa @    8 : 06      LdaTheHole
    0x132b495793fb @    9 : 15 05   StaCurrentContextSlot [5]
    0x132b495793fd @   11 : 06      LdaTheHole
    0x132b495793fe @   12 : 15 06   StaCurrentContextSlot [6]
    0x132b49579400 @   14 : 06      LdaTheHole
    0x132b49579401 @   15 : 15 07   StaCurrentContextSlot [7]
    0x132b49579403 @   17 : 06      LdaTheHole
    0x132b49579404 @   18 : 15 08   StaCurrentContextSlot [8]
    0x132b49579406 @   20 : 06      LdaTheHole
    0x132b49579407 @   21 : 15 09   StaCurrentContextSlot [9]
    0x132b49579409 @   23 : 06      LdaTheHole
    0x132b4957940a @   24 : 15 0a   StaCurrentContextSlot [10]
10 E> 0x132b4957940c @   26 : 91      StackCheck
.........
```

Fig. 2. V8 engine compiles a obfuscated JavaScript (*left*) into the corresponding V8's bytecode sequences (*right*).

V8 provides a runtime environment for JavaScript execution, and the browser provides the document object model (DOM) and the other Web Platform APIs. Due to this condition, for the simulation purpose of getting the V8's bytecode sequences, we have to make a browser environment in JavaScript so that no error will appear when the code is executed. The solution to this problem is to use the packaging library jsdom of Node.js. Jsdom is the implementation of the pure-JavaScript for many web standards. It uses WHATWG DOM and HTML standards, which can make a simulation of browser rendering engines to define documents into DOM [11]. Besides that, there are some objects that we have to define first before the execution, which is Windows scripting (WScript) and Visual Basic scripting (VBScript). WScript is one of the Microsoft scripting hosts that provides an environment in which users can use it to execute any kind of language that uses a variation of objects models to perform tasks. Almost all JavaScript codes use this scripting to run the string code that is already obfuscated.

Recently, there are two ways to generate bytecode sequences using the V8 engine. The first way is to use a Node.js environment by adding "–print-bytecode" to the command line parameters. The second one is to use a Chrome browser with launching the program from the command line and use "–js-flags='–print-bytecode'" to print. Figure 2 illustrates the output of V8's bytecode sequence generated using Node.Js.

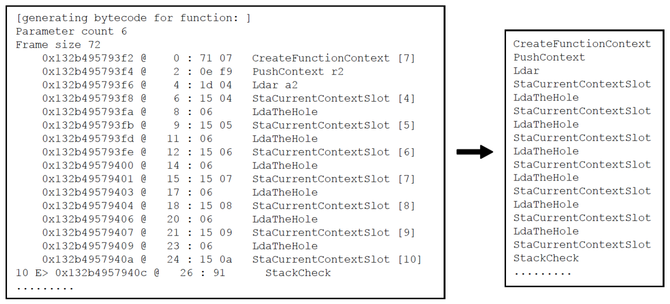
B. Preprocessing

V8's bytecodes can be seen as small building blocks that construct any JavaScript functionality when composed together [24]. It has several hundreds of bytecode which means that the occurrence of bytecode is very high for each sequence with the length can be longer than 100,000 bytecodes.

However, the output of a raw bytecode sequence consists of some elements, such as register number, properties, and values, which are not essential to detect malicious contents. We can ignore all of the unnecessary outputs and just focus on the main body of bytecodes. Figure 3 presents the illustration of preprocessing by taking actual bytecodes that we will use for the learning process.

C. Word Embedding

Word2vec is an unsupervised learning algorithm to make distributed representations of a word in a vector space to



```
[generating bytecode for function: ]
Parameter count 6
Frame size 72
    0x132b495793f2 @    0 : 71 07   CreateFunctionContext [7]
    0x132b495793f4 @    2 : 0e f9   PushContext r2
    0x132b495793f6 @    4 : 1d 04   Ldar a2
    0x132b495793f8 @    6 : 15 04   StaCurrentContextSlot [4]
    0x132b495793fa @    8 : 06      LdaTheHole
    0x132b495793fb @    9 : 15 05   StaCurrentContextSlot [5]
    0x132b495793fd @   11 : 06      LdaTheHole
    0x132b495793fe @   12 : 15 06   StaCurrentContextSlot [6]
    0x132b49579400 @   14 : 06      LdaTheHole
    0x132b49579401 @   15 : 15 07   StaCurrentContextSlot [7]
    0x132b49579403 @   17 : 06      LdaTheHole
    0x132b49579404 @   18 : 15 08   StaCurrentContextSlot [8]
    0x132b49579406 @   20 : 06      LdaTheHole
    0x132b49579407 @   21 : 15 09   StaCurrentContextSlot [9]
    0x132b49579409 @   23 : 06      LdaTheHole
    0x132b4957940a @   24 : 15 0a   StaCurrentContextSlot [10]
10 E> 0x132b4957940c @   26 : 91      StackCheck
.........
```

```
CreateFunctionContext
PushContext
Ldar
StaCurrentContextSlot
LdaTheHole
StaCurrentContextSlot
LdaTheHole
StaCurrentContextSlot
LdaTheHole
StaCurrentContextSlot
LdaTheHole
StaCurrentContextSlot
LdaTheHole
StaCurrentContextSlot
LdaTheHole
StaCurrentContextSlot
StackCheck
.........
```

Fig. 3. Preprocessing for elimination of redundant information from bytecode sequence.

achieve better performance in natural language processing tasks [22]. Mikolov et al. [25] proposed two types of architecture, Skip-gram, and Continuous Bag Of Word (CBOW) models. The Skip-gram model uses each current word to predict words within a certain range before and after the current word. On the other hand, CBOW is the inverse of the Skip-gram model, which predicts the center word given by the context of the words in range before and after the center words [26]. The CBOW is similar to the conventional bag-of-words representation in which we discard order information, and works by either summing or averaging the embedding vectors of the corresponding features.

In the CBOW model, the following loss function $\mathcal{L}$ is minimized so that the prediction probability of a center word w_t can be maximized for given context words $c_1, c_2, ..., c_k$:

$$\mathcal{L} = \sum_{t=1}^{T} \log p(w_t | c_{t-K} ... c_{t+K}) \tag{1}$$

where T is the number of corpora and K is the window length of surrounding words which regarded as the context. The probability is defined by softmax function

$$p(w_O | w_I) = \frac{\exp\left({v'_{w_O}}^{\top} v_{w_I}\right)}{\sum_{w=1}^{W} \exp\left({v'_{w}}^{\top} v_{w_I}\right)} \tag{2}$$

where W is the number of words in vocabulary, and v_w and v'_w are input and output vector representations of w, respectively. The denominator can be approximated via hierarchical softmax or negative sampling [22].

The hidden output of word2vec is a vector representation for each word in the vocabulary and we use it to transform a bytecode into a vector. After that, the vector representation is used as the input in our model. The dimensionality of a vector affects the accuracy of word representation, where a higher dimension will make higher accuracy. Since the dimensionality of vectors has a direct effect on memory requirements and processing time, it is essential to choose a good trade-off between size of model and task accuracy.

In this paper, word2vec is used to obtain vector representation of bytecodes; that is, a bytecode sequence is represented as a sequence of corresponding word vectors. The reason why we introduce word vector embedding into JavaScript code representation is because it can express the semantic meaning of code as a continuous, dense vector. We train the word2vec model to get an embedding matrix for words (bytecodes) and the matrix is used for transforming a bytecode sequence into a list of vectors.

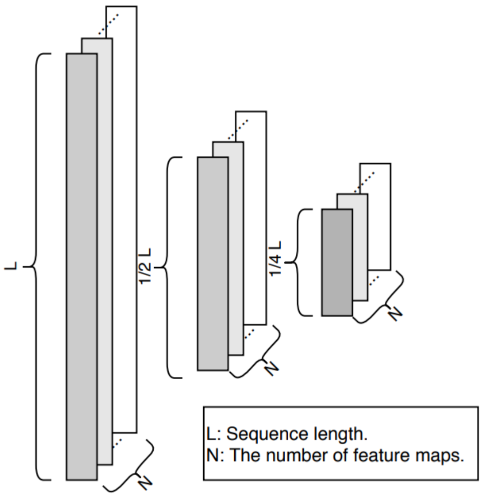


Fig. 4. Pyramidal shape in DPCNN.

D. Deep Pyramid Convolutional Neural Networks

DPCNN is a low-complexity word-level deep convolutional neural network (CNN) architecture introduced by Johnson et al. [12]. It can efficiently represent long-range associations in text for text categorization. This model tries to build an effective and efficient design of deep word-level CNNs, which yields better accuracy yet low-complexity. It can be obtained by increasing the depth but not the order of computation time. As the computation time per layer decreases exponentially in a '*pyramid*,' this model is called *deep pyramid CNN*, which is illustrated in Fig. 4. Due to such a pyramidal shape, the computation time is generally halved for each layer after every pooling.

The architecture of DPCNN consists of the following three components: text region embedding, shortcuts connection with pre-activation and identity mapping, and downsampling blocks. A text region embedding layer with the sequential input is similar to the standard convolutional layer applied to a sequence of vectors representing a sentence or a document. Since the bytecode sequence has much repetitive code, in this paper, we adopt a bigger kernel size (window) than that in the original paper, where we set 100 instead of 3. To reduce the computation time for this network, we set 30 as the stride of the convolutional layer. It also can cover long association among blocks of bytecode in a sequence. Besides that, we use unsupervised embedding to enhance region embedding, which is useful for classification.

Shortcut connection with pre-activation is applied in order to enable training of deep networks which is written as $\mathbf{z} + f(\mathbf{z})$ where f represents the skipped layers [27]. The definition

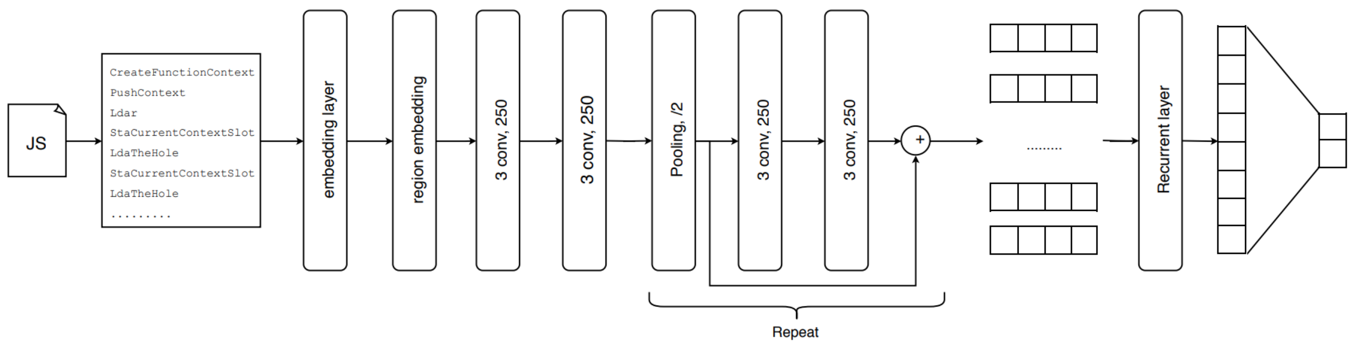


Fig. 5. Architecture of DPCNN with recurrent network for malicious JavaScript detection.

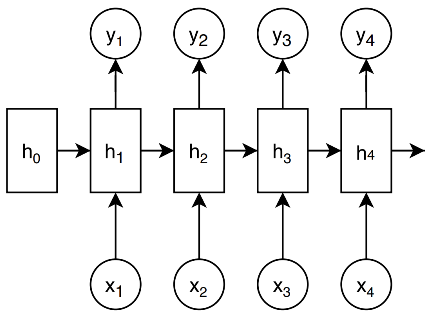


Fig. 6. Schematic structure of RNNs.

of skipped layers $f(\mathbf{z})$ is two convolutional layers with *pre-activation*. The activation function in the proposed model is the rectified liner unit (ReLU) $\sigma(\cdot) = \max(x, 0)$.

One of the characteristics in DPCNN is the downsampling block, which makes this architecture looks like *'pyramid'*. Downsampling with stride two necessarily doubles the adequate coverage of the convolution kernel. Thus, DPCNN is computationally efficient for representing long-range associations for long sequence data. Figure 5 shows the architecture of DPCNN used in this work.

E. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) is one of deep learning models that deals with sequences and stacks such as sentences, documents, and sequences. RNNs allow representing variable-length sequential inputs in fixed-size vectors while paying attention to the structured properties of the inputs [28]. Several architectures such as Simple RNN [28], the Long Short-Term Memory (LSTM) [29], and the Gated Recurrent Unit [30] are proposed to implement RNNs in various perspectives. Fig. 6 describes a graphical representation of RNNs where h_n, x_n, y_n is the hidden state, input, output respectively. The architecture of RNNs can be interpreted as a chained neural network, which allows the previous output to be used as an input while having hidden states. Since RNNs have this architecture, it gives advantages of the model such as sharing of weights across the time, consideration of historical information, and the natural processing of variable-length inputs. However, it makes the computation of this model slower than other neural network architectures like multi-layer perceptrons.

TABLE I
THE WHOLE SET OF JAVASCRIPT CODE SAMPLES USED FOR THE EXPERIMENT.

Data Set	Total Files	#Malicious	#Benign
Training	22,010	12,729	9,281
Validation	2,446	1,407	1,039
Testing	6,115	3,521	2,594
Total	**30,571**	**17,657**	**12,914**

IV. EXPERIMENTS

In this section, a performance comparison is carried out among the proposed malicious JavaScript detection system and some alternatives.

A. Experimental Setup

We conducted experiments on large labeled JavaScript files data set that is described in Table I. The malicious JavaScript files data set consists of three different sources as follows, 1,783 files from anti-malware engineering workshop (MWS) data set 2015 [31], 15,663 files from JavaScript malware collections by Hynek Petrak [32], and 211 files from JavaScript malicious data set in GeeksOnSecurity of GitHub [33]. Moreover, we used 12,914 JavaScript files by depth crawling from Alexa top domain list as benign JavaScript data set. Initially, we split data set into two data set with percentage 80% and 20%. We used the smaller data set for testing the model. Meanwhile, we used 10% of the bigger data set for validation and the rest is used to train the model. Most of the original data set is obfuscated with many varieties of obfuscation styles. For file which is not obfuscated, we applied obfuscation with some techniques such as string and encoding obfuscation.

To compile a JavaScript code into a bytecode sequence, the code should be executed a virtual environment. Jsdom library in Node.js is used to provide DOM objects so that we can execute and compile a code into bytecode sequence in a terminal. WScript objects are also needed to comply with some

TABLE II
PERFORMANCE OF VARIOUS MODELS WHICH WERE EVALUATED FOR THIS WORK

Model	Accuracy (%)	Precision (%)	Recall (%)	F1-score	AUC
LSTM (L=60K)	95.75(±0.1102)	95.90(±0.0986)	96.14(±0.0845)	0.9451(±0.0011)	0.9698(±0.0005)
DPCNN (L=200K)	97.33(±0.1379)	96.64(±0.0678)	97.10(±0.0940)	**0.9684**(±0.0008)[a] 0.00001	0.9949(±0.0002)
DPCNN-RNN (L=200K)	97.15(±0.2525)	**96.69**(±0.3020)	97.03(±0.2649)	0.9684(±0.0029)[a] 0.00117	0.9951(±0.0002)
DPCNN-LSTM (L=200K)	96.87(±0.1867)	96.37(±0.2917)	96.85(±0.2335)	0.9657(±0.0027)[a] 0.00165	0.9937(±0.0005)
DPCNN-BiLSTM (L=200K)	**97.36**(±0.2046)	96.63(±0.2578)	**97.11**(±0.1702)	0.9683(±0.0023)[a] 0.00051	**0.9951**(±0.0001)

[a] p value.

of JavaScript codes that use those scripting in the program. Because the malicious codes have the potential to be active and attack the system, we ran the codes in the virtual machine so that it can be safe from any malicious behaviors.

We conducted tuning of various hyper-parameters for deep learning models and the unsupervised learning model, and the best setting is then set based on the validation error rate. For the word2vec model, we adopt the following parameters: window size=3, embedding size=100, and vocabulary size=203. We applied the embedding matrix as the first layer of our deep learning model before feeding it to the next layer. Besides that, we set stride 30 and kernel size 100 for the region embedding layer, which is a convolutional layer for a region of sequence covering one or more codes. We made different depth of model for the DPCNN-RNNs because we need longer sequences for the input of the recurrent layer. Consequently, DPCNN-RNNs has deeper architecture than DPCNN that have 12 and 8 layers for DPCNN and DPCNN-RNNs respectively. After that, we defined the number of feature maps of the convolution layer in DPCNN by 250 and the kernel size by 3 [12]. We carried out experiments with a variety of recurrent layers such as RNN, LSTM, and Bidirectional LSTM (BiLSTM) with the size of the hidden layer 50, and zero value as the first hidden input. Not only that, we also tuned the hyper-parameters for model that only use recurrent network, which is the previous work's model, LSTM. The final setting of LSTM model that we set for the experiment as follows, the number of layer is 2, the hidden layer dimension is 200 and the drop-out is 0.2. The stochastic gradient descent (SGD) optimizer was used to train all models.

The DPCNN model is designed to address long-range associations of a sequence. However, the maximum length of sequence exhausts the memory capacity and the computation time. The length of bytecode sequence can be longer than 400,000 codes, which is too long to handle on a computer with a limited size of memory. Therefore, we set the length of bytecode sequences to $L = 200,000$ for all models that use DPCNN and $L = 60,000$ for LSTM in order to avoid out of memory problem. Concurrently, we also use the zero-padding technique for a sequence that has the length shorter than the maximum length.

B. Performance Evaluation

In this section, we evaluate performances of our proposed methods. Table II describes the performance metrics of all models. We used common performance metrics such as the accuracy, precision, recall, F1 score, and the area under the receiver operating characteristics (ROC) curve (AUC).

The objective of this work is to find a suitable model that can minimize the false negative (FN) cases. In other words, we have to get a high recall, which indicates how many malicious JavaScript codes labeled by the model are originally malicious (misclassified malicious JavaScript). A low false negative rate is crucial, which translates to a successful attack, that is, a malicious code goes undetected.

Table II shows tour evaluation of four variants of DPCNN with recurrent networks in detecting malicious JavaScript. In addition, we used a previous work's model, LSTM, for comparison. Each model has a different architecture, especially in the recurrent layer, where we try to know the performance of the recurrent layer that fits in our model. Based on the results, we find that the combination of DPCNN and recurrent networks have superior results compared to the only recurrent networks. The model that use DPCNN outperformed the model that does not use DPCNN around 2%. Among all models, DPCNN-BiLSTM has the best performance in accuracy, recall and AUC score. However, DPCNN-RNN and DPCNN give slightly better performance in precision and F1-score.

We evaluated the AUC score and plotted the ROC curve for each model to show the diagnostic ability of binary classifiers. ROC curve shows the trade off between sensitivity (true positive rate) and specificity (true negative rate). Figure 7 shows us the ROC curve for each model in our experiment. LSTM model gives the lowest curve of other models that use DPCNN as the first architecture of our classifier. On the other hand, all models with DPCNN gives better AUC score as well as ROC curve. Therefore, we can conclude that the proposed model has good property as a detector of malicious JavaScript codes.

V. DISCUSSION

The experimental results in Section IV demonstrate that the feature learning in DPCNN works well for extracting representation of bytecode sequences. A pyramid shape could reduce the computation time so that it can make the model

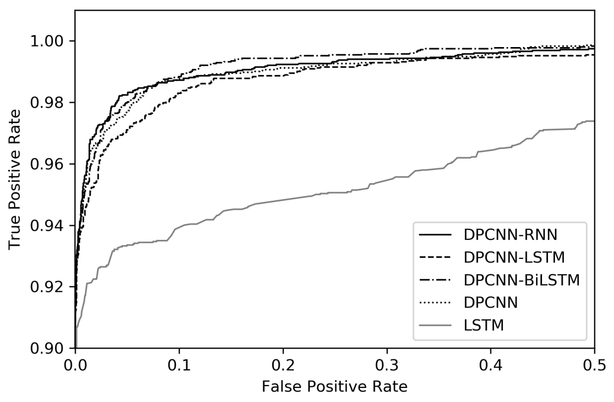


Fig. 7. ROC curves for different DPCNN models zoomed into a maximum FPR = 0.5 and a minimum TPR = 0.9

detection of malicious bytecodes very quickly even for representing long-range associations. Furthermore, the recurrent layer sharpens the accuracy of the model by capturing the dependencies of each feature in a sequence.

The performance results show that the use of DPCNN gives better result than the LSTM model. This is because of the ability of DPCNN to capture the long-range association in the bytecode sequences and reduce the complexity of the network so that it can process longer sequence than previous one. The use of RNNs results in the performance improvement to some extend, which helps to consider historical information of the bytecode sequence. However, the addition of RNNs after DPCNN occasionally makes the information is easily corrupted due to being multiplied many times by small numbers, which causes a vanishing gradient problem. Due to this problem, the DPCNN with RNNs often misclassifies the input and makes the number of FN and FP is more significant than other models that do not use RNNs. Furthermore, the two-tailed paired t-test is conducted to check the difference between the model which use DPCNN and LSTM that does not use DPCNN. The result shows a significant difference in performance where p-value is less than 0.05.

We consider several limitations of the proposed method to detect malicious JavaScript. These include limitations due to the virtual environment, length of a sequence, and adversarial-learning-based attacks.

The first limitation is a virtual environment that is used to compile the high-level JavaScript language into bytecode sequence. We need to set all possibilities DOM object and function, which is needed for execution and the security of the environment. However, it is kind of difficult to provide all the required objects. Malicious codes often use their functions and objects that need connection with their server. Therefore, the virtual environment cannot transform all obfuscated JavaScript codes into bytecode sequences. Also, some malicious codes can detect whether the environment is virtual or not. It makes the malicious code will not show the real code of the malware even though we have already run the program in a virtual environment.

Another limitation is the maximum length of the sequence. Our proposed model has a fixed maximum model as the hyperparameter. For our experiment, we set the maximum length is 200,000 codes for a sequence. The longer maximum length that we set, the memory will increase too and lead to out of memory exceptions. The bytecode sequence can be longer than 200,000, which means that the information of sequence that includes the maliciousness cannot be obtained due to the maximum length of the input sequence. It may be possible to process for an extremely long sequence by using advanced computers that are released in the future, which contain more capacity of memory.

Another thing that we have to concern is the adversarial learning-based attack. The research about adversarial learning inspires other research about new attack schemes for deep learning models such as CNN and RNN. While it is not a direct attack, Papernot et al. [34] have shown that standard RNN cells are vulnerable from adversarial learning-based attack. Therefore, it is essential to run the model in a secure environment.

VI. Conclusion

As described herein, we propose a deep neural network, which is the combination of DPCNN and recurrent network for malicious JavaScript detection by analyzing the bytecode sequence that is a good way to avoid obfuscation in malicious code. The result of our experiment reveals that the proposed method performed well to detect the maliciousness of JavaScript code. We used the unsupervised learning algorithm to create vector representation of sequences, which then becomes the input for the model. Feature learning is done by DPCNN to construct simpler features before going through the classifier. It succeeds in representing the long-range association in the bytecode sequences.

Acknowledgment

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

References

[1] C. Zapponi, “Githut: a small place to discover languages in github,” https://githut.info/, accessed: 2020-05-10.

[2] S. Abaimov and G. Bianchi, “Coddle: Code-injection detection with deep learning,” *IEEE Access*, vol. 7, pp. 128 617–128 627, 2019.

[3] A. K. Sood and S. Zeadally, “Drive-by download attacks: A comparative study,” *IT Professional*, vol. 18, no. 5, pp. 18–25, 2016.

[4] V. L. Le, I. Welch, X. Gao, and P. Komisarczuk, “Detecting heap-spray attacks in drive-by downloads: Giving

attackers a hand," in *38th Annual IEEE Conference on Local Computer Networks*, 2013, pp. 300–303.

[5] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *NDSS*, 2007.

[6] W. Xu, F. Zhang, and S. Zhu, "Jstill: mostly static detection of obfuscated malicious javascript code," in *CODASPY 2013 - Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy*, Feb. 2013, pp. 117–128.

[7] P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques," in *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, 2009, pp. 47–54.

[8] S. Palahan, D. Babić, S. Chaudhuri, and D. Kifer, "Extraction of statistically significant malware behaviors," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013, p. 69–78.

[9] X. Zhang, M. Sun, J. Wang, and J. Wang, "Malware detection based on opcode sequence and resnet," in *Security with Intelligent Computing and Big-data Services*, C.-N. Yang, S.-L. Peng, and L. C. Jain, Eds., 2020, pp. 489–502.

[10] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Computers & Security*, vol. 81, pp. 123 – 147, 2019.

[11] Y. Fang, C. Huang, L. Liu, and M. Xue, "Research on malicious javascript detection technology based on lstm," *IEEE Access*, vol. 6, pp. 59 118–59 125, 2018.

[12] R. Johnson and T. Zhang, "Deep pyramid convolutional neural networks for text categorization," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, vol. 1, Jul. 2017, pp. 562–570.

[13] Y. Choi, T. Kim, S. Choi, and C. Lee, "Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis," in *Future Generation Information Technology*, Y.-h. Lee, T.-h. Kim, W.-c. Fang, and D. Ślezak, Eds., 2009, pp. 160–172.

[14] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *Journal of Network and Computer Applications*, vol. 36, no. 2, pp. 646 – 656, 2013.

[15] J. W. Stokes, R. Agrawal, G. McDonald, and M. Hausknecht, "Scriptnet: Neural static analysis for malicious javascript detection," in *IEEE Military Communications Conference (MILCOM)*, 2019, pp. 1–8.

[16] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "Jast: Fully syntactic detection of malicious (obfuscated) javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Giuffrida, S. Bardin, and G. Blanc, Eds., 2018.

[17] J. N. C. S. B. Anderson, D. Quist and T. Lane, "Graph-based malware detection using dynamic analysis," *Computer Virology*, vol. 7, pp. 247–258, 2011.

[18] N. Kawaguchi and K. Omote, "Malware function classification using apis in initial behavior," in *10th Asia Joint Conference on Information Security*, 2015, pp. 138–144.

[19] B. Anderson, C. Storlie, and T. Lane, "Improving malware classification: Bridging the static/dynamic gap," in *AISec'12 - Proceedings of the ACM Workshop on Security and Artificial Intelligence*, Nov. 2012, pp. 3–14.

[20] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *Computers & Security*, vol. 77, pp. 578 – 594, 2018.

[21] H. Yakura, S. Shinozaki, R. Nishimura, Y. Oyama, and J. Sakuma, "Neural malware analysis with attention mechanism," *Computers & Security*, vol. 87, p. 101592, 2019.

[22] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, vol. 2, 2013, p. 3111–3119.

[23] V8. [Online]. Available: https://v8.dev/

[24] F. Hinkelmann, "Understanding v8's bytecode," https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775, Aug. 2017, accessed: 2020-05-10.

[25] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, 2013.

[26] E. Nalisnick and S. Ravi, "Learning the dimensionality of word embeddings," 2015.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[28] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179 – 211, 1990.

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997.

[30] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, 2014.

[31] M. Hatada, M. Akiyama, T. Matsuki, and T. Kasama, "Empowering anti-malware research in japan by sharing the mws datasets," *Journal of Information Processing*, vol. 23, no. 5, pp. 579–588, 2015.

[32] H. Petrak. Javascript-malware-collection. [Online]. Available: https://github.com/HynekPetrak/javascript-malware-collection

[33] Geeksonsecurity. js-malicios-dataset. [Online]. Available: https://github.com/geeksonsecurity

[34] N. Papernot, P. McDaniel, A. Swami, and R. Harang, "Crafting adversarial input sequences for recurrent neural networks," in *IEEE Military Communications Conference (MILCOM)*, Dec. 2016, pp. 49–54.