Computer Security Symposium 2020
26 - 29 October 2020

Deep Pyramid Convolutional Neural Networks for Detecting Obfuscated Malicious JavaScript Codes Using Bytecode Sequence Features

Muhammad Fakhrur Rozi[1,a)] Sangwook Kim[1] Seiichi Ozawa[1,2]

Abstract: The cyber attackers generally inject obfuscated malicious scripts to exploit the vulnerabilities by using various methods. To break the complexity of obfuscated JavaScript code, we use JavaScript's bytecode sequences, which representing the instruction of the program. Using this feature, we examine a deep neural network to detect malicious behavior based on the sequence pattern. However, due to the super long bytecode sequence problem, a deep pyramid convolutional neural network (DPCNN) is adopted to address long-range associations of the sequence. A pyramid shape in DPCNN helps to halve the computation and the complexity of the model where it can handle very long input. We combine DPCNN with recurrent neural networks (RNNs) to improve the system's performance that helps considering the historical information of bytecode sequences. The experiment result provides the high accuracy of the proposed model that outperforms the previous model in detecting malicious JavaScript.

Keywords: Cybersecurity, Deep learning, Malicious JavaScript detection

1. Introduction

The improvement of the obfuscation technique helps most programmers or software engineers protect their code from reverse engineering. However, cyber attackers also utilize this technique to hide their intention in malicious code/script and evade them from the anti-virus system. We can find this obfuscated malicious code in the form of JavaScript language, which is indeed one of the most used client-side programming languages to develop a web application [19]. Using the obfuscated malicious JavaScript code, the attackers try to inject scripting code into the output of the applications and then sends it to the user ' s web browser [18]. It exploits the vulnerabilities of the application so that the malicious-injected code is executed and used to access sensitive data stored and transferred to a server under the attacker ' s control. These codes hide on some parts of the web page, such as image, text, pop-up, and button. The obfuscation techniques succeed in making malicious a JavaScript code hard to be recognized by the anti-virus system, yet mixed and multi-level obfuscation can make it more complicated for the system to de-obfuscate the code [11].

Due to the complexity of the obfuscation problem, the researchers came up with a solution that used various JavaScript code features. One of the approaches is to transform the JavaScript code into the sort of representation that we can analyze it effectively. We can obtain the output, such as opcode [20], log activity [16], or byte code sequence [3], by executing the code in a safe environment that represents the program's activity. In this research, we use bytecode sequences as the feature of JavaScript code to break the obfuscation part that the attackers might use to avoid the anti-virus system. There is an engine in JavaScript that compiles an abstract syntax tree (AST) into a sequence of instruction codes to boost the performance of running time, where it is similar to the machine code. However, this feature is hard to analyze due to the super long sequence of bytecode, even though the program is simple. More complex the obfuscation technique, the bytecode sequence will be longer.

In this work, we examine the deep learning method to solve the super long sequence problem in bytecode sequences. We use Deep Pyramid Convolutional Neural Networks (DPCNN) [10], which solve long sequence problems by reducing sequence features to become some of the feature maps. This approach gives significant performance for detecting malicious JavaScript compared to other approaches. We apply the recurrent network at the end of DPCNN to get a long-range association of the sequence. Some adjustments are applied to DPCNN to fit with the recurrent network.

Our motivating hypothesis is that implementing the convolutional neural network with a pyramid shape structure can extend the capacity of the model to capture the information of the bytecode sequence. We hypothesize that analyzing a more extended bytecode sequence can give more

[1] Graduate School of Engineering, Kobe University
[2] Center for Mathematical and Data Sciences, Kobe University
[a)] rozi_mahfud@stu.kobe-u.ac.jp

insight information on the JavaScript program. DPCNN makes the features are easier to analyze, providing the important feature of the bytecode sequence.

Our main contributions in this paper are as follows:

(1) We thoroughly evaluate the efficacy of the DPCNN model for obfuscated malicious JavaScript code detection and propose more extension by combining with RNNs.

(2) We compare the performance of the DPCNN model, some alternative combination with RNNs, and RNNs for detecting malicious JavaScript code based on bytecode sequences.

2. Related Works

Attempt to develop a malicious code detection system has recently undertaken by introducing machine learning or deep learning models. The approach may depend on the feature that we want to analyze based on how the features can be extracted, e.g., static analysis, dynamic analysis, and hybrid analysis.

The static analysis approach makes the feature of samples by using the content without necessitating their execution. The work from Rafiqul Islam et al. [9] used the 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. In the other works, the researcher transformed plain JavaScript source code into a hexadecimal byte sequence for each character. Fass et al. [4] enriched the feature to use Abstract Syntax Tree (AST) representation that contains more information than lexical units, and it was able to analyze samples whose behavior is time- or environment-dependent.

Recently, some works have proposed applying deep learning model for malware or malicious JavaScript code detection. Fang et al. [3] proposed malicious JavaScript classification by implementing Long Short-Term Memory (LSTM) to a bytecode sequence analysis of the V8 JavaScript engine. The experimental results showed that the proposed model has higher accuracy than the previous methods, i.e., random forest, support vector machine (SVM), and Naive Bayes. Stokes et al. [17] introduced similar work that they proposed the modification of the LSTM model 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. They conducted experiments to compare both models and evaluate the efficacy of both models. Furthermore, Zhang et al. [20] used Residual Network (ResNet), one of the deep learning models that apply a deep convolutional network to handle long sequence problems in an opcode feature.

To deal with the long-range problem in sequential data is quite challenging. Some researches combine convolution and recurrent approaches to deal with the long-range problem in sequential data. Johnson et al. [10] proposed DPCNN

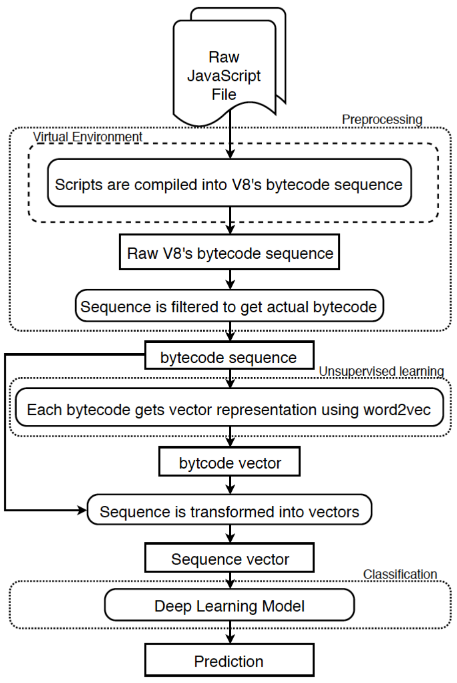

Raw JavaScript File
Preprocessing
Virtual Environment
Scripts are compiled into V8's bytecode sequence
Raw V8's bytecode sequence
Sequence is filtered to get actual bytecode
bytecode sequence
Unsupervised learning
Each bytecode gets vector representation using word2vec
bytcode vector
Sequence is transformed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

Fig. 1 Overview of our proposed method.

architecture for text categorization 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. Due to this layer, the architecture of this model is like a pyramid shape. We can use this method in other data types similar to the characteristics of text, such as sequences.

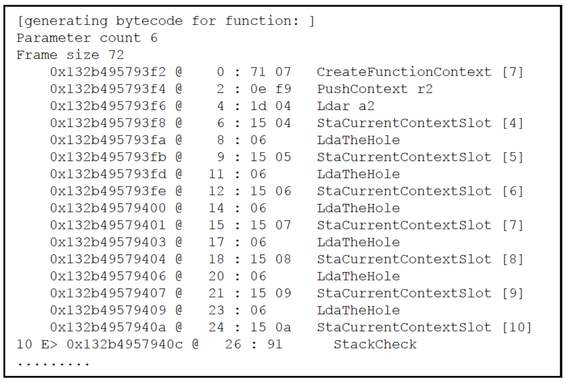
3. Proposed Method

The full detection framework for malicious JavaScript code is illustrated in **Fig. 1**. Firstly, a preprocessing step that we use a V8 engine to compile a program's function into the sequence of V8's bytecode sequences. Then, we remove the unimportant features of the bytecode sequences. After that, we implement bytecode embedding for every code to get the vector representation. Finally, We can input each code's vector representation for a sequence to the deep learning model.

3.1 Bytecode Sequences

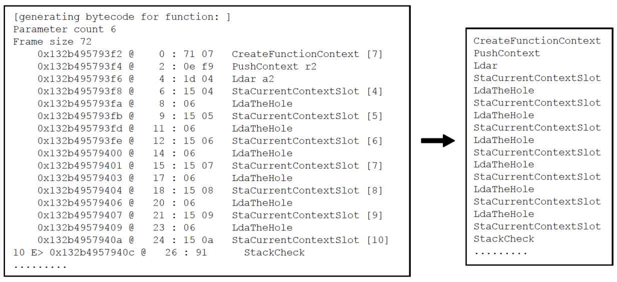
Bytecode sequence is an abstract machine sequence code that corresponds to the program's representation without any redundancy [7]. It has the same computational language model as the physical CPU, for thereby, it has a very long sequence.

There are many kinds of JavaScript engines that can compile bytecode by an interpreter or the Just-In-Time (JIT) compilers. One of them is V8 engine, which is an open-source JavaScript engine by Google. When V8 compiles JavaScript code, the parser in this engine generates an abstract syntax tree (AST) that represents the syntactic structure of the JavaScript code, and then the interpreter generates bytecode from this syntax tree. We can find this



```
[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's bytecode sequences.



```
[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.

bytecode in many applications such as Chrome and Node.js.

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 "–jsflags='–print-bytecode'" to print. **Fig. 2** illustrates the output of V8's bytecode sequence generated using Node.Js.

3.2 Preprocessing

We can assume the V8's bytecodes as small building blocks that construct any JavaScript functionality. V8's bytecode consists of several hundred words or codes. Therefore, the frequency of the occurrence of the bytecode is very high for each sequence. 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 these unnecessary outputs and focus on the main body of bytecodes. **Fig. 3** presents the illustration of preprocessing by taking actual bytecodes that we will use for the learning process.

3.3 Bytecode Embedding

To make a feature representation of bytecode, we encode the bytecode into embedding space. Assuming that the bytecode sequence is a word embedding problem, we can use the Word2Vec model [13]. It is an unsupervised learning algorithm to make distributed representations of a word in a vector space to achieve better performance in natural language processing tasks. Mikolov et al. [12] 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 specific range before and after the current word [14]. On the other hand, CBOW is the inverse of the Skip-gram model, which predicts the center word given by the words' context in range before and after the center words. 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 [13].

In this paper, a sequence of word vectors represents a corresponding bytecode sequence. We introduce word vector embedding into JavaScript code representation because it can express the semantic meaning of code as a continuous, dense vector.

3.4 Deep Pyramid Convolutional Neural Network

Johnson et al. [10] introduced a low-complexity word-level deep convolutional neural networks (CNN) that can efficiently represent long-range associations in text for text categorization, called deep pyramid convolutional neural networks (DPCNN). The name pyramid is from the architecture shape of this model. Due to this pyramid shape, it can halve the computation time per layer of the model that yields better accuracy yet low-complexity.

Generally, DPCNN has three components: text region embedding, shortcuts connection with preactivation and identity mapping, and down-sampling blocks. Among those components, down-sampling blocks are one of the components that are characteristic of the DPCNN model. It shapes the architecture of the model looks like a pyramid. The down-sampling with stride two necessarily doubles the adequate coverage of the convolutional kernel. Figure 4 shows us how the DPCNN model looks like, where the feature map stays at the same length. Still, the model reduces the length of the sequence to get a shorter representation of the feature. For the full architecture of this model is showed in **Fig. 5**.

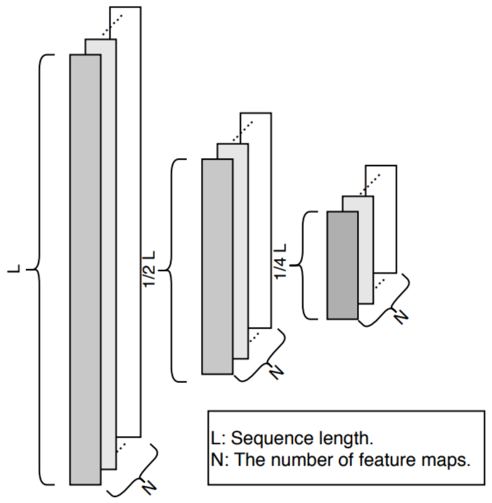


Fig. 4 Pyramidal shape in DPCNN.

3.5 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) is a class of neural networks that allow previous outputs to be used as inputs while having hidden states [2]. RNNs deal with sequences and stacks such as sentences, documents, and sequences. It allows representing variable-length sequential inputs in fixed-size vectors while paying attention to the inputs' structured properties. There are several types of RNNs model architectures, such as simple RNN [2], the Long Short-Term Memory (LSTM) [8], and the Gated Recurrent Unit [1].

Fig. 6 describes a graphical representation of RNNs where h_n, x_n, y_n is the hidden state, input, and output, respectively. We can interpret the architecture of RNNs as a chained neural network. Since RNNs have this architecture, it gives advantages to the model, such as sharing 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.

3.6 Detection Mechanism

Fig. 1 explains our proposed mechanism to detect obfuscated malicious JavaScript code. It starts with transforming the JavaScript code into the bytecode sequence representation by using the V8 JavaScipt engine. This engine's background is parsing the JavaScript code into an abstract syntax tree (AST) such that the internal interpreter transforms it into the bytecode sequence. After we finish getting the sequence, we use the unsupervised learning model to create the low-dimensional space representation for each bytecode in a sequence. In this work, we assume the bytecode sequence as the same as the word in natural language processing (NLP). We maximize the similarity between bytecode using the Word2Vec model to have a proper vector representation of bytecodes.

Next, we have our full architecture of the model in Fig. 5. In the figure, the model consists of two models: the DPCNN and the RNNs. We use the embedding layer to transform the bytecode sequence into a distributed vector representation sequence based on the bytecode vocabulary of the Word2Vec model. By having this sequence, we can pass it to the DPCNN model. This model has two main components: text region embedding and downsampling block. Text region embedding generalizes commonly used word embedding to the embedding of text regions covering one or more words. It is followed by stacking of convolution blocks (two convolution layers and a shortcut) interleaved with pooling layers with stride 2 for downsampling [10]. We stop the downsampling block until a certain length before we compute it into the RNNs model. Finally, we implement the RNNs model and fully connected layer for the rest process to get the final output.

4. Experiments

We conduct experiments to evaluate our proposed method's performance and compare it with the previous work, i.e., LSTM and other alternatives combined with the RNNs model.

4.1 Experimental Setup

Table 1 describes our experiments on the extensive labeled JavaScript files data set. Our malicious data set consists of three different sources: anti-malware engineering workshop (MWS) 2015 [6], JavaScript malware collections by Hynek Petrak [15], and GeeksOnSecurity of GitHub [5]. Moreover, we acquired 12,914 JavaScript files by depth crawling from Alexa top domain list as benign JavaScript data set. Initially, we split data set randomly into two set with a percentage of 80% and 20%. We used the 20% data set for testing the model, and we took 10% of the other data set for validation. Then, we used the rest (72%) as a training data set.

Most of the original data set are the obfuscated code with many varieties of obfuscation technique styles. We applied obfuscation with some techniques such as string and encoding obfuscation for the non-obfuscated JavaScript code.

Meanwhile, we executed the JavaScript code in a virtual environment to compile a JavaScript code into a bytecode sequence. The virtual machine keeps the experiment safe from any malicious behaviors because our data have the potential to be active. Jsdom library in Node.js provides DOM objects so that we can run and compile code into bytecode sequence in a terminal. WScript objects are also needed to comply with some of JavaScript codes that use those scripting in the program.

In our experiments, We tuned hyper-parameters of our proposed models to choose the best setting 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. For the region embedding layer, we set stride=30 and kernel size=100. DPCNN-RNNs have more

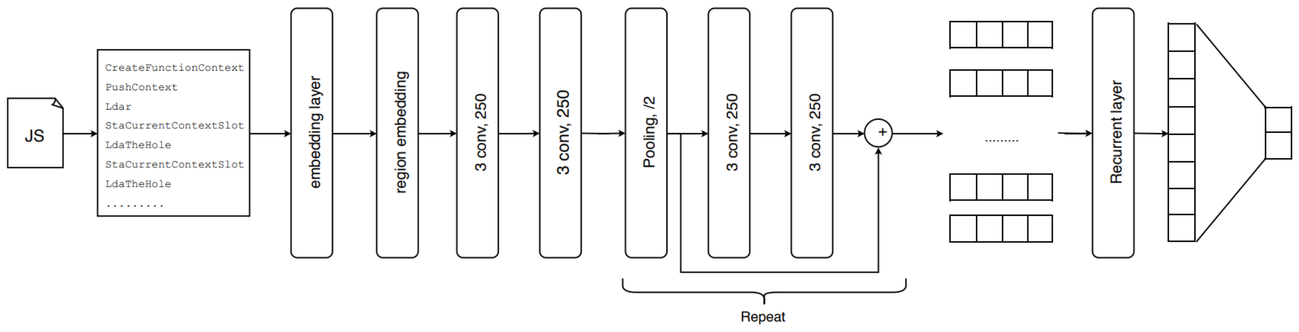


Fig. 5 Architecture of DPCNN with recurrent network for malicious JavaScript detection.

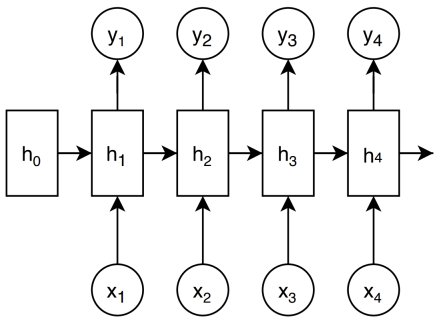


Fig. 6 Schematic structure of RNNs.

Table 1 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

profound architecture than the only DPCNN. Because of that, the number of layers of DPCNN-RNNs is 8, while the model with only DPCNN has 12 layers. We assigned the number of feature maps of the convolution layer in DPCNN by 250 and the kernel size by 3.

We compared our proposed method with the work from Fang et al. [3], i.e., LSTM. We did our experiments with the combination of DPCNN and other 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. The final setting of the LSTM model that we set for the experiment is as follows: the number of layers is 2, the hidden layer dimension is 200, the drop-out is 0.2, and the optimizer is SGD. Due to memory capacity, we set the length of the sequence to 200,000 for DPCNN model, and 60,000 for the LSTM model. Zero-padding is used for the sequence that has shorter and we truncate the sequence if it is longer.

4.2 Performance Evaluation

Table 2 shows the result of the performance of each model. There are five metrics of evaluation: accuracy, precision, recall, F1-score, and the area under the receiver operating characteristics (ROC) curve (AUC). Our objective is to find a suitable model that can minimize the false negative (FN) cases, which indicates that JavaScript file that is actually malicious but the system classify it as benign.

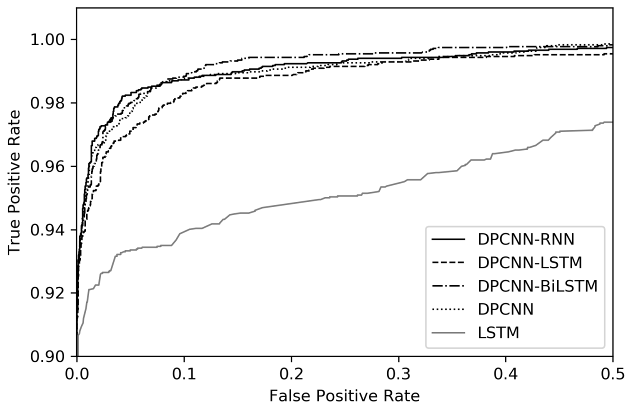


Fig. 7 ROC curves for different DPCNN models zoomed into a maximum FPR = 0.5.

Our experimental results show that the combination of DPCNN and RNNs has superior results than the only RNNs. It clearly shows that the DPCNN model outperformed other models that do not use DPCNN (97.36% vs. 95.75%). If we look at the combined model DPCNN and RNNs, DPCNN-BiLSTM has the best accuracy, recall, and AUC score. Besides that, DPCNN-RNN gives a slightly higher performance in precision and F1-score.

We assessed the diagnostic ability of binary classifiers by plotting the ROC curve. This curve shows the trade-off between the true-positive rate and the false-negative rate. We can see the ROC curve of the result in **Fig. 7**. It explains that the LSTM model has the lowest curve of other models that use DPCNN as the first architecture. Besides that, all models with DPCNN give a better AUC score as well as the ROC curve. Therefore, we can infer that the proposed model has suitable property as a detector of malicious JavaScript codes.

5. Discussion

We demonstrate how the feature learning in DPCNN works for extracting representation of bytecode sequences in Section IV. The notion of a pyramid shape architecture can reduce the model's complexity so that it can detect ma-

Table 2 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)	0.9949(±0.0002)
DPCNN-RNN (L=200K)	97.15(±0.2525)	**96.69**(±0.3020)	97.03(±0.2649)	0.9684(±0.0029)	0.9951(±0.0002)
DPCNN-LSTM (L=200K)	96.87(±0.1867)	96.37(±0.2917)	96.85(±0.2335)	0.9657(±0.0027)	0.9937(±0.0005)
DPCNN-BiLSTM (L=200K)	**97.36**(±0.2046)	96.63(±0.2578)	**97.11**(±0.1702)	0.9683(±0.0023)	**0.9951**(±0.0001)

licious bytecodes efficiently, even though for representing a very long sequence. Also, the recurrent layer sharpens the accuracy of the model by capturing each feature's dependencies.

This work introduces the DPCNN model to solve the long-range association problem in the bytecode sequence. We compare the model with the LSTM model to see the efficacy of both models to handle long input of the bytecode sequence. The result shows that the DPCNN gives a significant improvement compared to the LSTM model. The reason is because of the ability of DPCNN to reduce the complexity of the network but still capture the essential feature of a sequence so that it can process longer sequences. On the other hand, the LSTM model uses a memory cell to save temporary information from the previous state. However, this model cannot work on a super long sequence because it requires a large amount of memory bandwidth as the sequence become longer.

Moreover, we introduce the combination of the DPCNN with some alternatives of RNNs. The use of RNNs results in performance improvement to some extend, which helps to consider historical information of the bytecode sequence. However, the addition of RNNs after DPCNN occasionally causes a vanishing gradient problem, which affects the model's performance.

The length of the bytecode sequence affects the model's ability to capture the essential features of the sequence. More extended input that we can use, the better performance we will get. It is because the bytecode sequence consists of code blocks that make up any JavaScript functionality when composed together.

We consider several limitations of our proposed method as follows: the virtual environment, length of a sequence, and adversarial learning-based attacks. We use a virtual environment to compile the high-level JavaScript language into the bytecode sequence. Therefore, We need to set all possibilities DOM object and function for execution and the environment's security. However, it is hard to provide all the required objects that make many JavaScript files not fully executed due to error function or unknown objects. Besides that, we have to concern about securing the deep learning model from the adversarial learning-based attack. For that reason, it is essential to run the model in a secure environment. Furthermore, because of the computer's memory capacity, we have to limit the length of the bytecode sequence to get into our model. This limitation affects the performance of the detection system to capture all features in the bytecode sequence.

6. Conclusion

In this paper, we introduced a new method to solve obfuscated malicious JavaScript code detection. We examined a deep neural network, the DPCNN, and RNNs to detect malicious intention on the JavaScript's bytecode sequences feature that helps to break the obfuscation parts in source code. The model has a pyramid shape architecture that can compress the model's complexity into more straightforward features. It succeeds in representing the long-range association in the bytecode sequences. Moreover, the combination with the RNNs model slightly improves the accuracy of the model. The experimental results demonstrated that our proposed method performed well to detect the maliciousness of JavaScript code.

7. Acknowledgement

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] Chung, J., Gülçehre, Ç., Cho, K. and Bengio, Y.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling, *CoRR* (2014).

[2] Elman, J. L.: Finding structure in time, *Cognitive Science*, Vol. 14, No. 2, pp. 179 – 211 (1990).

[3] Fang, Y., Huang, C., Liu, L. and Xue, M.: Research on Malicious JavaScript Detection Technology Based on LSTM, *IEEE Access*, Vol. 6, pp. 59118–59125 (2018).

[4] Fass, A., Krawczyk, R. P., Backes, M. and Stock, B.: JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript, *Detection of Intrusions and Malware, and Vulnerability Assessment* (Giuffrida, C., Bardin, S. and Blanc, G., eds.) (2018).

[5] Geeksonsecurity: js-malicios-dataset, , available from ⟨https://github.com/geeksonsecurity⟩ (accessed 2019-12-28).

[6] Hatada, M., Akiyama, M., Matsuki, T. and Kasama, T.: Empowering Anti-malware Research in Japan by Sharing the MWS Datasets, *Journal of Information Processing*, Vol. 23, No. 5, pp. 579–588 (2015).

[7] Hinkelmann, F.: Understanding V8 ' s Bytecode, , available from ⟨https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775⟩ (accessed 2020-05-10).

[8] Hochreiter, S. and Schmidhuber, J.: Long Short-Term Memory, *Neural Comput.*, Vol. 9, No. 8, p. 1735–1780 (1997).

[9] Islam, R., Tian, R., Batten, L. M. and Versteeg, S.: Classification of malware based on integrated static and dynamic features, *Journal of Network and Computer Applications*, Vol. 36, No. 2, pp. 646 – 656 (2013).

[10] Johnson, R. and Zhang, T.: Deep Pyramid Convolutional Neural Networks for Text Categorization, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, Vol. 1, pp. 562–570 (2017).

[11] Likarish, P., Jung, E. and Jo, I.: Obfuscated malicious javascript detection using classification techniques, *2009 4th*

International Conference on Malicious and Unwanted Software (MALWARE), pp. 47–54 (2009).

[12] Mikolov, T., Chen, K., Corrado, G. S. and Dean, J.: Efficient Estimation of Word Representations in Vector Space, *CoRR* (2013).

[13] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. and Dean, J.: Distributed Representations of Words and Phrases and Their Compositionality, *Proceedings of the 26th International Conference on Neural Information Processing Systems*, Vol. 2, p. 3111–3119 (2013).

[14] Nalisnick, E. and Ravi, S.: Learning the Dimensionality of Word Embeddings (2015).

[15] Petrak, H.: JavaScript-Malware-Collection, , available from ⟨https://github.com/HynekPetrak/javascript-malware-collection⟩ (accessed 2019-12-28).

[16] Rhode, M., Burnap, P. and Jones, K.: Early-stage malware prediction using recurrent neural networks, *Computers & Security*, Vol. 77, pp. 578 – 594 (2018).

[17] Stokes, J. W., Agrawal, R., McDonald, G. and Hausknecht, M.: ScriptNet: Neural Static Analysis for Malicious JavaScript Detection, *IEEE Military Communications Conference (MILCOM)*, pp. 1–8 (2019).

[18] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Krügel, C. and Vigna, G.: Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis, *NDSS* (2007).

[19] Zapponi, C.: GitHut: a small place to discover languages in Github, , available from ⟨https://githut.info/⟩ (accessed 2020-05-10).

[20] Zhang, X., Sun, M., Wang, J. and Wang, J.: Malware Detection Based on Opcode Sequence and ResNet, *Security with Intelligent Computing and Big-data Services* (Yang, C.-N., Peng, S.-L. and Jain, L. C., eds.), pp. 489–502 (2020).