SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2016; **9**:1520–1534
Published online 11 February 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1441

RESEARCH ARTICLE

A deep learning approach for detecting malicious JavaScript code

Yao Wang*, Wan-dong Cai and Peng-cheng Wei

Department of Computer Science and Technology, Northwestern Polytechnical University, Xi'an, China

ABSTRACT

Malicious JavaScript code in webpages on the Internet is an emergent security issue because of its universality and potentially severe impact. Because of its obfuscation and complexities, detecting it has a considerable cost. Over the last few years, several machine learning-based detection approaches have been proposed; most of them use shallow discriminating models with features that are constructed with artificial rules. However, with the advent of the big data era for information transmission, these existing methods already cannot satisfy actual needs. In this paper, we present a new deep learning framework for detection of malicious JavaScript code, from which we obtained the highest detection accuracy compared with the control group. The architecture is composed of a sparse random projection, deep learning model, and logistic regression. Stacked denoising auto-encoders were used to extract high-level features from JavaScript code; logistic regression as a classifier was used to distinguish between malicious and benign JavaScript code. Experimental results indicated that our architecture, with over 27 000 labeled samples, can achieve an accuracy of up to 95%, with a false positive rate less than 4.2% in the best case. Copyright © 2016 John Wiley & Sons, Ltd.

KEYWORDS

JavaScript attacks; static analysis; deep learning; SdA; logistic regression; random projection

***Correspondence**

Yao Wang, Department of Computer Science and Technology, Northwestern Polytechnical University, Xi'an, China.
E-mail: wangyao@mail.nwpu.edu.cn

1. INTRODUCTION

According to a recent report by Symantec [1], there are millions of victims attacked by malicious JavaScript on the Internet per day. JavaScript code embedded in webpages provides a platform for attacks, such as a drive-by download that unnoticeably redirects the victim's browser to load content and malware from a remote server when the webpage is visited in any ordinary browser [2]. Malicious JavaScript code often probes and exploits vulnerabilities in the browser's environment.

Most regrettably, compared with other types of network attacks, malicious JavaScript code is hard to detect. The reason is that there are too many ways an attacker can conceal malicious attacks in his code, and the code is immediately interpreted by the user's browser without precompilation, that is, without converting the script code into system-related machine code. This characteristic expediently allows JavaScript attacks to dynamically inspect and exploit different types of vulnerabilities in the browser's environment.

One traditional approach of detecting malicious code is getting past the black list and checking whether the virus signature is listed. The black list is a list of virus signatures that are identified as malicious from the user's perspective. The black list for malicious code is created on the host computer system, and the virus signature file is updated from the external security server regularly. However, malicious JavaScript code can hardly be detected by this conventional method because of concealment, complexities, and a lack of up-to-date signature files [3]. Another approach is the heuristic detecting method, which is based on rules established by security experts who define a benign behavior or a malicious behavior in order to search for malicious code; however, its major drawback is that it can only detect known malicious code but cannot detect an unknown malicious code. Moreover, dynamic analysis methods have been widely used in malware detection [4]. In dynamic analysis, the malware is executed in a simulated environment, such as a virtual machine, emulator, and sandbox. By making use of a program debugging tool, such as Process Explorer, RegShot, or SysAnalyzer, malware researchers identify the working process of the

malcode. Dynamic analysis can easily detect the new malware by observing its behavior, but this technique is more time-consuming and cannot represent the real application because the behavior of malware changes with different triggering conditions [5].

As such, the detection of malicious JavaScript has received the attention of security researchers. Subsequently, several machine learning-based approaches have been put forward that automatically distinguish between malicious and benign activity, such as the detection frameworks in the works of Likarish *et al.* [6] and Fraiwan *et al.* [7] as well as the detectors in the works of Curtsinger *et al.* [8] and Rieck *et al.* [9]. Although the techniques can be applied to JavaScript detection, it requires much time and manpower to manually craft features for both malicious and benign JavaScript code. The detection mechanism of these methods is mainly based on feature design. Their efficiency relies mostly on the comprehensiveness of feature selection; that is to say, it depends on whether selected features can really represent a sample well. Because JavaScript code segments may have multiple links to other pages and concealment is easy by utilizing data encapsulation, code reordering, rubbish strings insertion, and other techniques, it is difficult to manually design obvious features for malicious JavaScript code. Furthermore, with the scheme of machine learning-based methods, all the manually designed features have to be specified in the implementation, and it is hard for implementers to add new features because the framework has to be reconstructed and there is no doubt it is time-consuming.

In this paper, in order to simplify the aforementioned process of tedious and time-consuming feature selection, we introduce a deep learning-based method that can extract features for JavaScript code automatically with very little manual intervention. In addition, a machine learning system with multiple layers would extract more abstract features of JavaScript code and therefore is considered to have the capacity for yielding higher classification accuracy than the shallow models like the support vector machine (SVM) and logistic regression. Our approach makes use of multiple-layer stacked denoising autoencoders (SdA) to learn the deep features of JavaScript code. By using those learned features, a logistic regression classifier can efficiently detect JavaScript code and has sufficient capacity to uncover malicious code. The key contribution of this research could be framed as follows:

- Learning-based detection. We introduce techniques of deep learning into generating detection models for static JavaScript code analysis, which can spare us from manually crafting features as in existing machine learning-based methods.
- Zero-day attacks detection. The model is able to extract the intrinsic features of the attacks; therefore, our approach has the capability of identifying malicious code even in the case of zero-day or previously unknown attacks.
- Evaluation. We evaluated our model on a larger-scale dataset, which contains over 27 000 samples, to show its applicability in malicious JavaScript code detection.

The remainder of this paper is structured as follows. Related work is first discussed in Section 2. Section 3 then introduces our deep learning framework for malicious JavaScript code detection. Section 4 details the experimental results of our method and the analysis of these results. And last, conclusions and discussions are presented in Section 5.

2. RELATED WORK

Malicious JavaScript code constitutes a serious threat to personal computer systems and has become a security research hotspot. Many methods have been proposed for detecting JavaScript attacks in recent years; for example, in [10], the authors used high-interaction honeypots, and low-interaction honeypots [11] were also used for malicious detection. Many researches concerned about the particular attack types, such as drive-by downloads [2,12] and heap spraying [13], relied on recognizing specific attacks. In addition to these methods, some tools have been developed for analyzing JavaScript code [7–9,14]. Although these approaches are effective and reduce the threat of attacks to a certain degree, they can hardly detect the evolving JavaScript attacks and often consume a large amount of time analyzing JavaScript code. The terminology, evolving attacks, refers to zero-day malware or unknown malware. They can modify the original codes to produce offspring copies, which have the same functionalities but with different signatures. Unfortunately, the detection of evolving attacks is beyond the capability of the aforementioned methods; nevertheless, machine learning techniques can identify such attacks with some degree of success [15].

Therefore, we have studied how to apply machine learning for automatic JavaScript identification. Huda *et al.* [16] proposed a framework for malware detection by choosing application program interface call statistics as malware features and using the SVM as the classifier; similarly, Alazab [17] extracted features from the sequences of application program interface calls and employed the *k*-nearest neighbor algorithm to classify malware behaviors; AL-Taharwa *et al.* [18] provided a JavaScript obfuscation detector, which is a mining and machine learning approach to detect obfuscated codes; Soska and Christin [19] constructed a set of features from the aspects of traffic statistics, file system structure, and webpage contents, which is followed by the process of adopting a C4.5 decision tree algorithm to determine the maliciousness of a target website. Although these traditional machine learning-based methods can predict unknown new malicious JavaScript code, the consumption of testing time would potentially affect their efficiency. In our experiment, the method we proposed provides a median testing run-

time of 0.34 s with 5423 samples in the testing set, while the testing time of SVM, alternating decision tree (ADTree), the RIPPER rule learner, and naïve Bayes algorithm are 42.36, 9.73, 24.62 and 6.83 s, respectively. In the case of fewer test samples, time consumption would not attract our attention; however, with the advent of the big data era for information transmission and the development of web technology, information transmission would be more frequent, and interaction between people and information would be more and more common; therefore, with large test samples, it would be very time-consuming.

Our research used the technique of deep learning, which is a class of machine learning algorithms, for detecting malicious JavaScript code. The method we proposed can automatically learn the features of JavaScript code and discriminate between benign and malicious code with an accuracy that outperforms the advanced machine learning-based approaches. Moreover, aiming at the problem of the low speed of traditional machine learning-based methods, our method has the superiority of a faster speed on the test set. As a result, our method achieved a detection accuracy of 94.82% and a running time of 0.34 s on the testing set, which contains 5423 samples.

3. DETECTION METHODOLOGY

We first propose a deep learning framework for malicious JavaScript code detection in this section. Then, we elaborate on how to analyze JavaScript code by using our learning framework, thereby distinguishing between malicious and benign JavaScript code.

3.1. Deep learning

Deep learning is an emerging research field of machine learning; it attempts to hierarchically learn high-level representation of data with deep neural networks. Each layer of the network is first layer-wise pre-trained [20] via unsupervised learning, and then the entire network carries out fine-tuning in a supervised mechanism. In this manner, high-hierarchy features can be learned from low-hierarchy ones, and the appropriate features can be applied to pattern classification in the end.

According to the universal approximation theorem of neural networks [21], deep models have a better ability to represent the nonlinear functions than shallow ones; therefore, deep models can achieve better results on large-scale training data. Furthermore, from a feature recognition and classification point of view, the deep learning framework incorporates a feature extractor and classifier into one framework, which can automatically learn feature representations (often from unlabeled data), thus avoiding spending substantial effort to manually design features.

Typical deep neural networks include convolutional neural networks [22], stacked auto-encoders [20], SdA [23], deep Boltzmann machines [24], and deep belief networks [25]. In this research, in order to learn more useful features with unsupervised pre-training, we relied on the SdAs for two reasons: first of all, SdA is suitable for the application of text classification [26], especially when the input is binary form. Secondly, according to the results described by Vincent *et al.* [23], the SdA would yield better results than other unsupervised methods when the input is binary high-dimensional data.

3.2. Denoising auto-encoder (dA)

The dA is an extension of a classical auto-encoder. In real applications, many problems like missing data and data noise would cause the theoretical method to be impractical. In order to force the hidden layer to learn more robust features, we trained the auto-encoder to reconstruct the input from corrupted input data. To convert the auto-encoder into a dA, we needed to introduce a stochastic noise (i.e., a stochastic corruption step operation) into the input layer.

A dA is an extension of auto-encoder, which is used to reconstruct the inputs from a noisy version of it by minimizing the reconstruction loss. Figure 1 gives an illustration of a dA; note that once a training sample $\boldsymbol{x}$ is presented, where $x \in R^d$, a corrupted version of $\boldsymbol{x}$ is produced according to the distribution $\boldsymbol{qD}(\bar{\boldsymbol{x}}|\boldsymbol{x})$, which is a stochastic corruption process that randomly sets some of the input to zero. A dA first encodes $\bar{\boldsymbol{x}}$ to a hidden representation $\boldsymbol{y}$ through a deterministic mapping (1), where $y \in R^h$, and then decodes representation $\boldsymbol{y}$ back into a reconstruction $\boldsymbol{z}$ of the same shape as $\boldsymbol{x}$ through a similar transformation (2), where $z \in R^d$. Hence, the dA is attempting to reconstruct the original inputs $\boldsymbol{x}$ from the corrupted values. Mathematically, these two affine transformations are formulated as

$$\boldsymbol{y} = \boldsymbol{s}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{1}$$

$$\boldsymbol{z} = \boldsymbol{s}(\boldsymbol{W}'\boldsymbol{y} + \boldsymbol{b}') \tag{2}$$

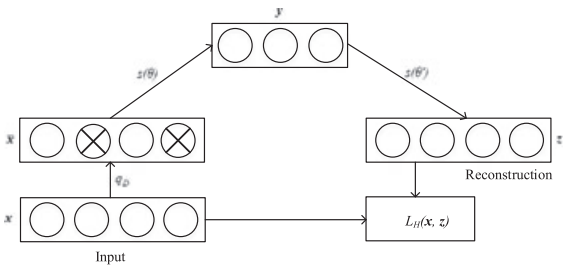


Figure 1. Single-layer denoising auto-encoders for JavaScript code classification. An input $\boldsymbol{x}$ is stochastically corrupted to $\bar{\mathrm{x}}$ (via q_D). The model then learns a hidden feature $\boldsymbol{y}$ from $\bar{\mathrm{x}}$ via encoder $s(\theta)$ and tries to reconstruct $\boldsymbol{x}$ via decoder $s(\theta')$, generating reconstruction $\boldsymbol{z}$. Reconstruction loss is evaluated by the loss function $L_H(x, z)$.

where s is a nonlinearity like sigmoid function or hyperbolic tangent function, $\boldsymbol{W}$ is an encoding weight matrix, $\boldsymbol{W'}$ is a decoding weight matrix, $\boldsymbol{b}$ is an encoding bias vector, and $\boldsymbol{b'}$ is a decoding bias vector; theoretically, the decoding weight matrix $\boldsymbol{W'}$ of the reverse mapping is equal to the transpose of $\boldsymbol{W}$ (i.e., $\boldsymbol{W'} = \boldsymbol{W}^{\boldsymbol{T}}$).

There are many ways to define the reconstruction loss function, depending on the different data distribution of the input layer. For instance, the conventional squared loss function $L(x,z) = \|x - z\|^2$ and absolute value loss function $L(x,z) = |x - z|_1$ can be used. By transforming the JavaScript code into bit vectors in data preprocessing in Section 4.2, we can obtain the model parameters by minimizing the cross-entropy of the reconstruction loss function as

$$\underset{W,b,b'}{argmin}\, L(\boldsymbol{x},\boldsymbol{z}) = \underset{W,b,b'}{argmin} -\sum_{i=1}^{n}[\boldsymbol{x}_i log \boldsymbol{z}_i + (1-\boldsymbol{x}_i) log(1-\boldsymbol{z}_i)] \quad (3)$$

Thus, the simplest approach to update the parameters is using the iteration of the gradient descent as follows:

$$W = W - \alpha \frac{\partial L_H(x,z)}{\partial W} \quad (4)$$

$$b = b - \alpha \frac{\partial L_H(x,z)}{\partial b} \quad (5)$$

$$b' = b' - \alpha \frac{\partial L_H(x,z)}{\partial b'} \quad (6)$$

where the parameter $\alpha > 0$ is called the learning rate. Every time after the update, the gradient is reassessed, and this process is repeated. When the network training is completed, the output of the hidden layer can be regarded as the learned feature, which afterwards can be used for classification or used as the input of a deeper layer to create a high-order feature.

3.3. Stacked denoising auto-encoders (SdA)

An SdA model is created by integrating multiple dAs to form a deep learning network as illustrated in Figure 2. Every hidden layer in the network is trained as a dA by optimizing equation (3) through unsupervised pretraining, and we can take the output of a dA on a previous layer as the input of the next. More clearly, the first hidden layer is trained as a dA with JavaScript code vectors as input, and after finishing training the first hidden layer, we use the output of the first hidden layer as the input of the second hidden layer. Similarly, once the k-th hidden layer is trained, the $(k+1)$-th layer can be trained using the output of the k-th hidden layer as the input of the $(k+1)$-th hidden layer to compute the latent representation. In this scheme, multiple dAs can be stacked hierarchically.

To use the SdA model for malicious JavaScript code detection, a logistic regression classifier is applied to the output of the last hidden layer to distinguish between malicious and benign JavaScript code. We then adjust parameters throughout the entire network by making use of target class labels. This is the fine-tuning stage on which we will elaborate in Section 3.5.

The network structure of our deep framework is similar to that of multilayer perceptron, also known as feedforward neural network, but they are different in essence. The multilayer perceptron is a classifier itself; however, our framework is composed of a feature extractor that aims to learn a

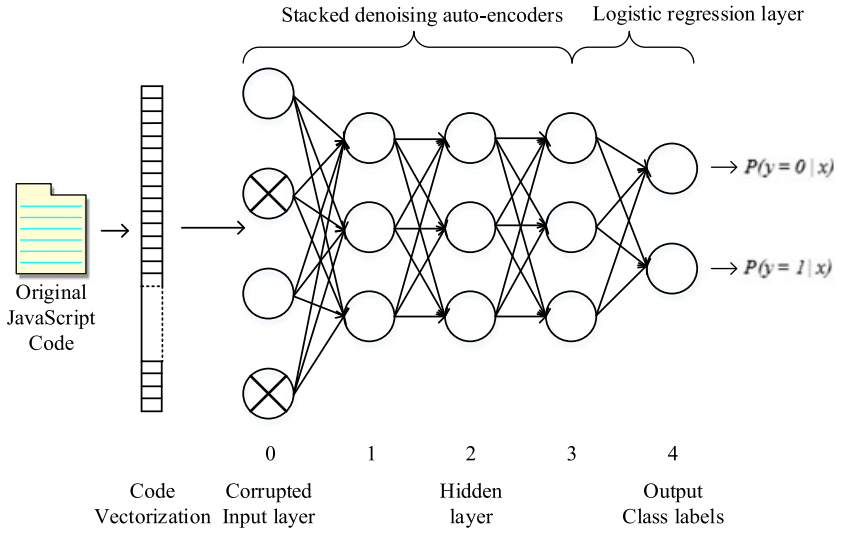


Figure 2. Deep framework for JavaScript code classification. It has five layers: one input layer, three hidden layers of denoising auto-encoders, and an output layer. A stacked denoising auto-encoder model is used to extract deep JavaScript features, and an output layer of logistic regression is used for classification.

compressed representation of the input and a classifier. Conceptually, the output of the SdA network is replaced with logistic regression.

3.4. Unsupervised layer-wise pre-training

In Section 3.3, we outlined a hierarchical approach to learn the deep features of JavaScript code by pre-training an SdA model. Next, we present the detailed derivation about training a dA on each layer.

In Equations (1) and (2), we set s to be the sigmoid function as an activation function in both the encoding and decoding processes, and it takes the form

$$s(x)=\frac{1}{1+e^{-x}} \tag{7}$$

and its first-order derivative is given by

$$s'(x)=s(x)\Big(1-s(x)\Big) \tag{8}$$

which will be useful in the process of deriving the parameters W, b, and b'.

In our implementation, because sigmoid activation is used and data are normalized in data preprocessing, we use the cross-entropy given by Equation (3) as the loss function; moreover, we take the mini-batch rule as the updating strategy for a large-scale dataset; thus, the loss function is actually computed on a mini-batch of input data, so that

$$L(x,z)=-\frac{1}{m}\sum_{b=1}^{m}\sum_{n=1}^{d}(x_{bn}log(z_{bn})+(1-x_{bn})log(1-z_{bn})), \tag{9}$$

where m is the mini-batch size and d is the input size. x_{bn} represents the input value of element n in the b-th mini-batch. In a similar way, z_{bn} denotes the reconstruction value of element n in the b-th mini-batch.

To train a dA, we make use of stochastic gradient descent with mini-batches to optimize the loss function (9). Specifically, the computation that a dA represents is given by

$$V_i^{(2)}=\sum_{n=1}^{d}W_{ni}^{(2)}x_{bn}+b_i^{(2)} \tag{10}$$

$$A_i^{(2)}=s\Big(V_i^{(2)}\Big) \tag{11}$$

$$V_j^{(3)}=\sum_{i=1}^{h}W_{ij}^{(3)}A_{i^{(2)}}+b_j^{(3)} \tag{12}$$

$$A_j^{(3)}=s\Big(V_j^{(3)}\Big) \tag{13}$$

$$\begin{aligned} z_{bn}=A_j^{(3)}&=s\left(\sum_{i=1}^{h}W_{ij}^{(3)}s\Big(V_i^{(2)}\Big)+b_j^{(3)}\right)\\ &=s\left(\sum_{i=1}^{h}W_{ij}^{(3)}s\left(\sum_{n=1}^{d}W_{ni}^{(2)}x_{bn}+b_i^{(2)}\right)+b_j^{(3)}\right) \end{aligned} \tag{14}$$

where $V_{i^{(2)}}$ denotes the input value of unit i in layer 2 (i.e., the hidden layer) while $A_{i^{(2)}}$ denotes its activation. Similarly, $V_j^{(3)}$ is the input of unit j in layer 3 (i.e., the reconstruction layer), and z_{bn} is its output value. Our model has parameters $\left\{W_{ni}^{(2)},b_i^{(2)},W_{ij}^{(3)},b_j^{(3)}\right\}$, where $W_{ni}^{(2)}$ denotes the weight associated with the connection between unit n in layer 1, and unit i in layer 2. Also, $b_i^{(2)}$ stands for the bias term of unit i in layer 2. The same applies for $W_{ij}^{(3)}$ and $b_j^{(3)}$.

In accordance with Equations (4) through (6), we obtain partial differentials of the loss function (9) with respect to parameters $W_{ni}^{(2)}$, $b_i^{(2)}$, and $b_j^{(3)}$ by using the chain rule for derivatives, and it takes the form

$$\begin{cases} \dfrac{\partial L(x,z)}{\partial W_{ni}^{(2)}}=\dfrac{\partial L(x,z)}{\partial z_{bn}}\dfrac{\partial z_{bn}}{\partial W_{ni}^{(2)}}\\ \dfrac{\partial L(x,z)}{\partial b_i^{(2)}}=\dfrac{\partial L(x,z)}{\partial z_{bn}}\dfrac{\partial z_{bn}}{\partial b_i^{(2)}}\\ \dfrac{\partial L(x,z)}{\partial b_j^{(3)}}=\dfrac{\partial L(x,z)}{\partial z_{bn}}\dfrac{\partial z_{bn}}{\partial b_j^{(3)}} \end{cases} \tag{15}$$

and the partial derivative of the loss function (9) over z_{bn} is computed by the expression

$$\frac{\partial L(x,z)}{\partial z_{bn}}=-\frac{1}{m}\sum_{b=1}^{m}\sum_{n=1}^{d}\frac{x_{bn}-z_{bn}}{z_{bn}(1-z_{bn})}. \tag{16}$$

We can compute the partial derivatives of the reconstruction z_{bn} (14) over parameters $W_{ni}^{(2)}$, $b_i^{(2)}$, and $b_j^{(3)}$ more compactly as

$$\begin{cases} \dfrac{\partial z_{bn}}{\partial W_{ni}^{(2)}}=s'\Big(V_j^{(3)}\Big)W_{ij}^{(3)}s'\Big(V_i^{(2)}\Big)x_{bn}\\ \dfrac{\partial z_{bn}}{\partial b_i^{(2)}}=s'\Big(V_j^{(3)}\Big)W_{ij}^{(3)}s'\Big(V_i^{(2)}\Big)\\ \dfrac{\partial z_{bn}}{\partial b_j^{(3)}}=s'\Big(V_j^{(3)}\Big) \end{cases}. \tag{17}$$

Putting the formulas (16) and (17) together, and making use of (8), we can compute the derivative of the overall loss function (9) as follows:

DOI: 10.1002/sec

$$\begin{cases} \dfrac{\partial L(x,z)}{\partial W_{ni}^{(2)}} = -\dfrac{1}{m}\sum_{b=1}^{m}\sum_{n=1}^{d}\dfrac{x_{bn}-z_{bn}}{z_{bn}(1-z_{bn})}s\left(V_j^{(3)}\right)\left(1-s\left(V_j^{(3)}\right)\right)W_{ij}^{(3)}s\left(V_i^{(2)}\right)\left(1-s\left(V_i^{(2)}\right)\right)x_{bn} \\ \qquad\qquad = -\dfrac{1}{m}\sum_{b=1}^{m}\sum_{n=1}^{d}\dfrac{x_{bn}-z_{bn}}{z_{bn}(1-z_{bn})}A_j^{(3)}\left(1-A_j^{(3)}\right)W_{ij}^{(3)}A_i^{(2)}\left(1-A_i^{(2)}\right)x_{bn} \\ \dfrac{\partial L(x,z)}{\partial b_i^{(2)}} = -\dfrac{1}{m}\sum_{b=1}^{m}\sum_{n=1}^{d}\dfrac{x_{bn}-z_{bn}}{z_{bn}(1-z_{bn})}s\left(V_j^{(3)}\right)\left(1-s\left(V_j^{(3)}\right)\right)W_{ij}^{(3)}s\left(V_i^{(2)}\right)\left(1-s\left(V_i^{(2)}\right)\right) \\ \qquad\qquad = -\dfrac{1}{m}\sum_{b=1}^{m}\sum_{n=1}^{d}\dfrac{x_{bn}-z_{bn}}{z_{bn}(1-z_{bn})}A_j^{(3)}\left(1-A_j^{(3)}\right)W_{ij}^{(3)}A_i^{(2)}\left(1-A_i^{(2)}\right) \\ \dfrac{\partial L(x,z)}{\partial b_j^{(3)}} = -\dfrac{1}{m}\sum_{b=1}^{m}\sum_{n=1}^{d}\dfrac{x_{bn}-z_{bn}}{z_{bn}(1-z_{bn})}s\left(V_j^{(3)}\right)\left(1-s\left(V_j^{(3)}\right)\right) \\ \qquad\qquad = -\dfrac{1}{m}\sum_{b=1}^{m}\sum_{n=1}^{d}\dfrac{x_{bn}-z_{bn}}{z_{bn}(1-z_{bn})}A_j^{(3)}\left(1-A_j^{(3)}\right) \end{cases} \tag{18}$$

Substituting the derived formula (18) into equations (4) through (6), the desired updating rules are determined.

To train our model, we first initialized the weight values of a hidden layer for implementing the first iteration of the mini-batch gradient descent, which should be chosen from a symmetrical interval that depends on the activation function [27]. For our sigmoid activation function, the interval is given by

$$\left[-4\sqrt{\frac{6}{num_{in}+num_{out}}}, 4\sqrt{\frac{6}{num_{in}+num_{out}}}\right]$$

where num_{in} is the input layer size and num_{out} is the hidden layer size. These initial values make certain that information from the activation function can be propagated both forward and backward. Then we can iteratively take steps of the stochastic gradient descent with mini-batches to approximately minimize the loss function.

After training our model, we neglect the reconstruction activation and extract the hidden activation to be the learned feature. Namely, the hidden activation is regarded as a compression of input data, which is an abstract representation. The following layers are trained in a similar way, as is discussed in Section 3.3, and we recursively substitute the output of the former layer for the input of this layer. In this fashion, the SdA network is constructed with the dA layer by layer.

3.5. Supervised fine-tuning and classification

After pre-training the SdA, we implement the supervised fine-tuning operation on the whole network. We make use of a logistic regression classifier fixed at the end of the network for classification by using the learned features from the SdA where we utilize the softmax function as its output activation. Mathematically, the probability that an input vector x belongs to a class i of a random variable Y can be represented as

$$\begin{aligned} P(Y=i|\boldsymbol{x},\boldsymbol{W},b) &= softmax_i(\boldsymbol{W}\boldsymbol{x}+b) \\ &= \frac{e^{\boldsymbol{W}_i\boldsymbol{x}+}b_i}{\sum_{j=1}^{k}e^{\boldsymbol{W}_j\boldsymbol{x}+}b_j} \end{aligned} \tag{19}$$

where the parameters $\boldsymbol{W}$ and b denote the weight matrix and bias term, respectively. We can estimate the probability of class labels in each of the k different possible values, so the output will be a k-dimensional vector whose results sum to 1. Classification is performed by choosing the corresponding prediction class whose probability is maximal, so that

$$y_{pred} = argmax_i P(Y=i|\boldsymbol{x},\boldsymbol{W},b) \tag{20}$$

In this paper, we just have two class labels, malicious and benign. Thus, in the special case where $k=2$, the logistic regression hypothesis results take the form

$$\begin{aligned} P(Y=i|\boldsymbol{x},\boldsymbol{W},b) &= \frac{1}{e^{\boldsymbol{W}_1\boldsymbol{x}+b_1}+e^{\boldsymbol{W}_2\boldsymbol{x}+b_2}}\begin{bmatrix} e^{\boldsymbol{W}_1\boldsymbol{x}+b_1} \\ e^{\boldsymbol{W}_2\boldsymbol{x}+b_2}\end{bmatrix} \\ &= \begin{bmatrix} \dfrac{1}{1+e^{(\boldsymbol{W}_2-\boldsymbol{W}_1)\boldsymbol{x}+(b_2-b_1)}} \\ \dfrac{e^{(\boldsymbol{W}_2-\boldsymbol{W}_1)\boldsymbol{x}+(b_2-b_1)}}{1+e^{(\boldsymbol{W}_2-\boldsymbol{W}_1)\boldsymbol{x}+(b_2-b_1)}}\end{bmatrix} \\ &= \begin{bmatrix} \dfrac{1}{1+e^{\boldsymbol{W}'\boldsymbol{x}+b'}} \\ 1-\dfrac{1}{1+e^{\boldsymbol{W}'\boldsymbol{x}+b'}}\end{bmatrix}. \end{aligned} \tag{21}$$

Softmax regression has as a property that its hypothesis is over-parameterized [28], and by taking advantage of this property, we can subtract $\boldsymbol{W}_1$ and b_1 from the two parameters.

After computing the probability of each class, fine-tuning is then implemented via a supervised mini-batch gradient descent of the negative log likelihood loss function. In other words, it uses the Back Propagation (BP) algorithm with gradient-based optimization to adjust the entire network's parameters from the top layer to the

Table I. Data labels and sample size of each set.

Class label	Name	Training set size	Validation set size	Testing set size
0	Benign	7392	2464	2464
1	Malicious	8868	2956	2959
Total		16 260	5420	5423

bottom. The derivation process is similar to that in Section 3.4.

4. EXPERIMENTS

4.1. Data description

The dataset was composed of 12 320 benign and 14 783 malicious JavaScript samples, which are labeled 0 and 1. By using a web crawler named Heritrix [29], these JavaScript code samples were acquired from the Internet over a period of 24 days. The dataset contains both obfuscated and plain JavaScript samples. For collecting benign samples, we visited reputable web sites listed by Alexa's Top.[†] But even that does not guarantee these samples are attack free; we have to make extensive verification to rule out potentially malicious instances of attacks.

For collecting malicious samples, we visited antivirus research organizations VX Heaven,[‡] a site that provides a library of malicious JavaScript code samples and Malicious Website Labs,[§] a site that releases up-to-date malicious URLs. Similar to the benign dataset, all the malicious samples that we obtained were not guaranteed to be malicious and therefore the verification process was of essential importance.

The two verification procedures we applied are outlined in the following:

(1) In order to improve the quality of benign samples, we discarded blacklisted URLs from the URLs we collected from Alex's Top[1] websites. We used the Google Safe Browsing service,[¶] which could offer a regularly updated list of dangerous URLs to accomplish this task effectively. The main aim of this step is to abandon known dangerous JavaScript codes from the benign dataset.
(2) By using the Google Safe Browsing service,[4] we analyzed all malicious URLs to ensure the webpage that a URL pointed to was indeed malicious. Then we made use of Dr. Web[††] and ClamAV[‡‡] to scan all the malicious samples we collected. The main goal of this step is to reserve the bad codes in malicious dataset.

The dataset was divided into three parts with a partition ratio of 3:1:1. That is to say, we choose three-fifths of the dataset as a training set, one-fifth as a validation set, and the rest as a testing set. The method we proposed in Section 3 was applied to this dataset. We made use of the training set to learn the weights and bias terms of each unit in the deep learning network; the validation set was used to adjust hyper-parameters such as the hidden layer sizes and hidden unit sizes, and the testing set was used to predict the classification results. The sample size of each set is detailed in Table I.

4.2. Data preprocessing

In many cases using deep learning for natural language processing, neural language models are used to represent the text samples in the form of word vectors, but for JavaScript code, especially for malicious JavaScript code, it usually uses nonstandard encoding, and there exist massive meaningless strings and Unicode symbols. Therefore, it is not suitable for adopting word vectors. What is more, the process of word vector training would be incredibly time-consuming and tedious. For the aforementioned reasons, we took a method of converting a JavaScript code sample to binary feature vectors as the input of our deep learning model. That is, according to ASCII table, we converted every letter in JavaScript code sample into eight bit binary codes, and we kept all the JavaScript code segments in the form of binary file. Figure 3 shows two examples of JavaScript code snippet. In Figure 3(a), function navigator.userAgent.toLowerCase() acquires user's agent information. After finding out which version of Internet Explorer the user is using, the malicious JavaScript code could be able to obtain the corresponding vulnerabilities of the web browser by making use of the iframe method of document.write() class. In Figure 3(b), function unescape() is used by hackers to decode an encoded string, and then the result can be written to the content of the webpage via function document.write(). The eval() function can accept a JavaScript expression as a parameter and return the result. The conversion process is shown in Figure 4; because of limited space, only the underlined part in Figure 3 is converted to binary vectors. By this means, we convert every JavaScript code sample into a binary file.

However, making use of this method would generate over 20 000 feature dimensionalities, so for the purpose of reducing the high-dimensional input data to an

[†] Alexa's Top Sites, http://www.alexa.com/topsites
[‡] VX Heaven, http://vxheaven.org
[§] Malicious Web Site Labs, http://www.mwsl.org.cn
[¶] Google Safe Browsing, http://developers.google.com/safe-browsing
[††] Dr. Web AntiVirus, http://www.drweb.com
[‡‡] Clam AntiVirus, http://www.clamav.net

```
<script type = "text/javascript" >
function whQd8(){
        var pcss = navigator.userAgent.toLowerCase();
        var kxkx = deconcept[ "SWFOb" + "jectU" + "til" ][ "getPlay" + "erVer" + "sion" ]();
        if (((kxkx[ 'major' ] == 10&&kxkx[ 'minor' ] <= 3) && kxkx[ 'rev' ] <=183||(kxkx[ 'major' ] == 11
                &&kxkx[ 'minor' ] <=1&&kxkx[ 'rev' ] <= 102&& ((pcss.indexOf( 'msie 6.0' ) >0 )
                || (pcss.indexOf( 'msie 7.0' ) >0 )|| (pcss.indexOf( 'msie' ) == -1)))))
        {
                document.writeln( "<iframe src = ff.html><\/iframe>" );
        }
        ...
}</ script>
```

(a) Malicious JavaScript code sample.

```
<script> document.write(unescape( "%3C%73%63%72%69%70%74%
                        20%6C%61%6E%67%75%61%67%65%
                        3D%67%62&73%72%69%70%74%3E" ))
</script>
...
eval( "arrNum = 0:ReDimTempStr(0):strLength
        =Len(NbjHrXYekZCCM..." )
...
```

(b) Obfuscated JavaScript code sample.

Figure 3. Example scripts.

appropriate dimensionality that decreases computational overhead, we next proposed a dimensionality reduction method, the sparse random projection, which is described in [30].

Sparse random projection is an effective method for dimension reduction. It projects the high-dimensional raw data to a much lower dimensional target data by using a sparse random matrix that allows for faster calculation of the projected data. Elements in the random matrix are taken from a distribution over $\{-1, 0, 1\}$; the probabilities of elements -1 and 1 are equal to $1/2\sqrt{d_r}$, and the probability of 0 is $1 - \left(1/\sqrt{d_r}\right)$, where d_r is the dimensionality of the raw data.

After reducing the feature dimensionality of the original dataset, our reduced target dimensionality resulted in the final 480 dimensionalities.

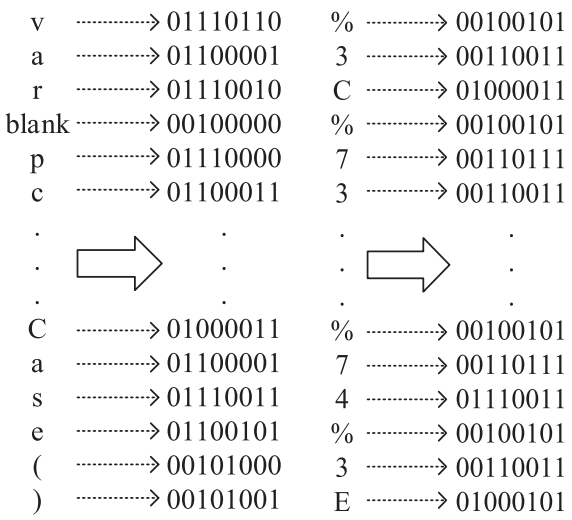


Figure 4. Examples of converting code snippet to binary vectors.

4.3. Structure of SdAs

There were several parameters to be determined for generating our model. The depth of hidden layers plays an important role in the classification accuracy, and we set the depth in the range of 1 to 5 with an increment of 1. As a matter of convenience, unit size in each hidden layer was set the same, and it was chosen from the set {100, 150, 200, 250, 300, 350, 400}. The epochs of pre-training and fine-tuning are also important if the number of epochs is too small; the model cannot achieve the highest accuracy, and our model would overfit. We set the pre-training epochs range from 200 to 1600 with a 200 increment, and fine-tuning epochs ranged from 4000 to 8000 with a 500 increment. We chose the optimal parameters configuration from 2520 stochastically independent runs, and the configuration to obtain the best classification was as follows: layer size = 3, units in hidden layers = 250, corruption level = 0.3, pre-training epochs = 1000 with a learning rate of 0.01, and fine-tuning epochs = 6000 with a learning rate of 0.1. Then a random search with a

predetermined parameter was used in our experiments, and for the simplicity of computation, we tested the influence of different parameters on our model while fixing the other parameters on an Intel Xeon E5-2420 2 Cores × 1.9 GHz computer with 64 GB of RAM.

First, we tested the effect of different hidden layer sizes on the validation set; the error rates and running time are shown in Table II. Here, we fixed the unit size of each hidden layer to 250. The performance improved greatly with an increase in the number of hidden layers from 1 to 3; however, more hidden layers than 3 do not give any better performance. The running time shows the time complexity of various structures; it nearly increased linearly with the increase of hidden layer size. The following experimental values are average results, which were obtained after the test was repeated 10 times.

Next, by fixing the hidden layer size to 3, we tested the effects of a different number of units in the hidden layer. Table III shows that for an input layer of 480 dimensions, 250 units in each hidden layer presents the best result. Fewer units would cause the model to learn improper features, and more units would lead to overfitting. Taking the running time complexity and classification error into consideration, our deep learning model with 3 layers and 250 units was the optimal choice.

Then we studied the influence of various pre-training epochs on the classification accuracy. As with the aforementioned method, if we kept fixed the fine-tuning epochs, the curve in Figure 5(a) shows that the accuracy on the validation set grows with an increase in the number of pre-training epochs. When the number of epochs increased to 1000, the accuracy can reach up to 94.63%, but continuing to increase the number of pre-training epochs would put unnecessary burdens on our model and negatively impact the overall performance, although it improved the accuracy on the validation set slightly. Therefore, 1000 pre-training epochs was the best choice.

Finally, we kept fixed the pre-training epochs and tested the impact of different fine-tuning epochs on the classification accuracy, as illustrated in Figure 5(b). From what has been discussed in the preceding text, for the same reason, we choose 6000 as the number of fine-tuning epochs. In Figure 6, we give examples of features vector extracted by the ready-built model. After preprocessing the code snippets, 250 abstracted features can be extracted from the JavaScript code examples in Figure 3 by our deep learning model.

Table II. Effect of the number of hidden layers.

Hidden layers	Classification error (%)	Time (s)
1	22.77	780
2	11.26	1149
3	5.37	1503
4	9.43	2227
5	14.02	2769

Table III. Effect of the number of hidden layer units.

Number of units	Classification error (%)	Time (s)
100	19.71	734
150	12.26	987
200	5.95	1074
250	5.37	1503
300	6.62	1786
350	7.63	1815
400	9.48	2392

4.4. Comparing with other methods

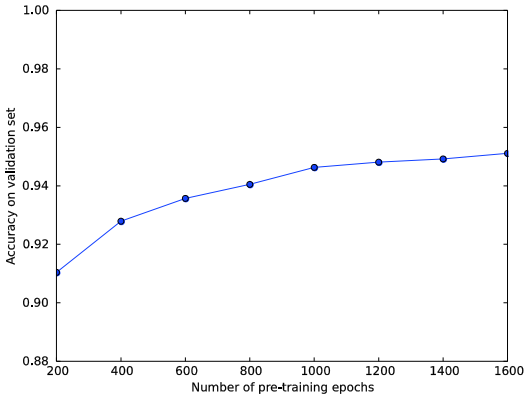
In this section, we made use of the testing set to predict the final classification results and compare our SdA with logistic regression (SdA-LR) model with several other widespread approaches. These models, which were used for comparison, were trained and tested using the same data set, and the same experimental setup as was used for our SdA-LR model.

We first illustrate the performance of our SdA-LR model in the form of a confusion matrix in Table IV. A confusion matrix is a specific table that allows visualization of the performance of a supervised learning algorithm [31], which makes it easier to understand whether the system is confusing. In a confusion matrix, TN represents that a benign JavaScript was correctly labeled as benign, TP denotes that a malicious JavaScript was correctly identified as malicious, FN indicates that a malicious JavaScript was incorrectly identified as benign and FP means that a benign JavaScript was incorrectly labeled as malicious. As for the performance metrics, we are mainly concerned with the TP rate, FP rate, and the overall classification accuracy defined as follows:

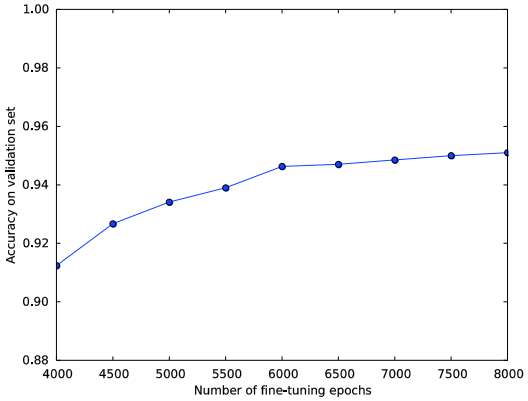
- $\text{TP } rate = \dfrac{\text{TP}}{\text{TP} + \text{FN}} \times 100\% \approx 93.95\%$
- $\text{FP } rate = \dfrac{FP}{FP + TN} \times 100\% \approx 4.13\%$
- $\text{Overall } accuracy = \dfrac{TP + TN}{TP + TN + FP + FN} \times 100\% \approx 94.82\%$

Then we compared the SdA model with other feature extraction models, including principle component analysis [32], independent component analysis [33], and factor analysis (FA) [34]. Using classification accuracy as a criterion, we verified the performance of these feature extractors.

After reducing the dimensionality of the original data by these feature extraction models, we classified the target features by using a logistic regression with the same parameters set to have a learning rate of 0.1 and 10 000 epochs. Here, in our SdA model, the hidden layer size



(a) Accuracy of the validation set with different pre-training epochs.



(b) Accuracy of the validation set with different fine-tuning epochs.

Figure 5. The influence of various epochs on accuracy.

```
[9.90682793e-04  9.99336241e-01  9.92261680e-01  5.01879089e-04  9.99385540e-01
 1.18120459e-02  9.99257750e-01  3.43191949e-04  3.95161265e-04  4.36968543e-01
 9.98021749e-01  9.96634902e-01  7.82693288e-05  9.96998713e-01  1.93621546e-04
       .               .               .               .               .
       .               .               .               .               .
       .               .               .               .               .
 9.93641393e-01  1.71676805e-02  9.99658479e-01  9.99806075e-01  9.94026511e-01
 2.03975693e-04  9.99999600e-01  1.21451086e-03  9.99056086e-01  1.40886278e-01]
```

(a) Extracted features vector from figure 3(a).

```
[3.68170333e-03  6.12466225e-01  9.99912902e-01  9.99695549e-01  3.83223964e-03
 1.11650597e-05  9.99870921e-01  6.15642558e-04  9.99994052e-01  1.94073033e-05
 2.66642828e-05  2.25878246e-04  3.32627717e-05  9.99784614e-01  9.89511400e-01
       .               .               .               .               .
       .               .               .               .               .
       .               .               .               .               .
 5.63008341e-06  5.12197820e-06  2.81083472e-06  8.42276454e-07  3.05349560e-07
 4.49301930e-05  9.99932156e-01  2.97061715e-06  4.37891859e-05  4.29155416e-10]
```

(b) Extracted features vector from figure 3(b).

Figure 6. Examples of extracted features vector.

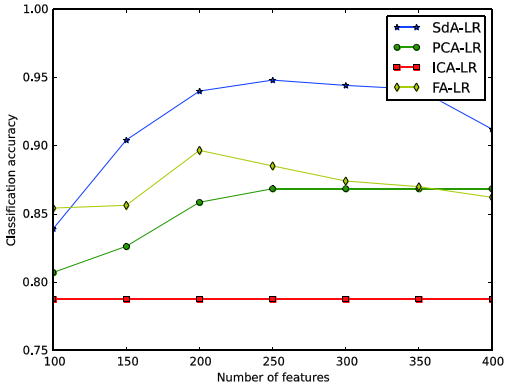
was set to 3. Experimental results in Figure 7(a) shows that by applying a logistic regression classifier, the classification performance of the SdA-LR is optimal compared with the other four models. The only exception was that FA-LR outperformed SdA-LR more than a little when the number of features was 100.

Table IV. Experimental results on the confusion matrix.

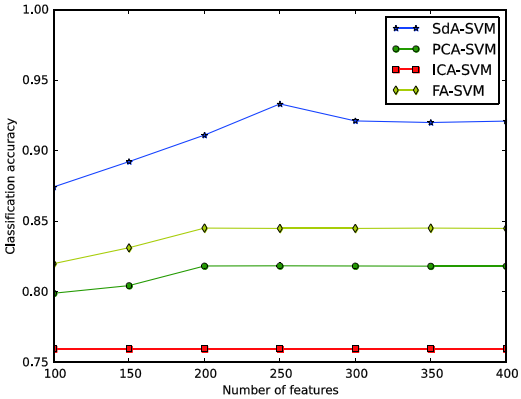
	Predicted as malicious	Predicted as benign	Total
Malicious	TP = 2780	FN = 179	2959
Benign	FP = 102	TN = 2362	2464
Total	2882	2541	5423

For strict verification, we added the SVM to the aforementioned models to verify whether the SdA has more advantages in the classification. From the results in Figure 7(b), we can see that the classification accuracy of the SdA-SVM exceeds the others by a large margin. Consequently, we can draw the conclusion that the SdA helps to improve the accuracy of classification.

Finally, we compared our approach with those of machine learning-based malicious JavaScript code detection methods in Likarish *et al.* [6]. We first made use of the feature extraction method proposed by Likarish *et al.* to convert our dataset into feature vectors (i.e., they used a total of 65 features, 50 of them are JavaScript keywords and



(a) Using logistic regression as the classifier.



(b) Using the support vector machine as the classifier

Figure 7. Comparing the four different feature extraction methods with (a) a logistic regression classifier and (b) the support vector machine. The vertical axis stands for the classification accuracy, and the horizontal axis represents the number of features we extracted from the original data. SdA, stacked denoising auto-encoder; PCA, principle component analysis; ICA, independent component analysis; FA, factor analysis; SVM, support vector machine.

symbols; the other 15 features are selected of their choice by domain knowledge) and then applied their proposed classifiers to produce an output that identified whether a JavaScript sample was benign or not. The whole process was carried out according to the literature (i.e., Likarish *et al.*).

In Figure 8, we can see that our SdA-LR model without manually designed features outperforms the four existing techniques in terms of average accuracy on the testing set, especially the model Radial Basis Function (RBF) SVM, which is a mature model commonly used in the application of malicious detection, yet our SdA-LR model boosted the accuracy of over 1%. Deep models contain more parameters and require more computing resources, thus resulting in long training time and slow convergence speed; we admit that deep learning algorithms take much more time to train models than some other machine learning algorithms such as RBF SVM, ADTree, RIPPER, and

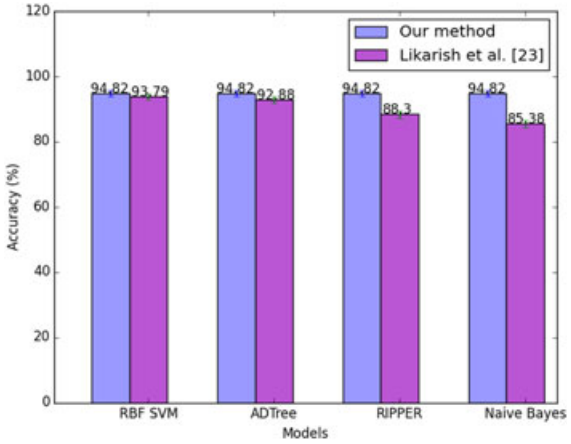


Figure 8. The overall accuracy of our proposed method in comparison with the work of Likarish *et al.* [6]. SVM, support vector machine; ADTree, alternating decision tree.

naïve Bayes as shown in Table V; however, after establishing the model, deep learning algorithms have a superiority because of their fast analysis on the testing set. In Table V, the running time on the same testing set demonstrates that the SdA-LR runs much faster than the other detection frameworks. Table V provides the results in terms of the machine learning assessment criteria for each of the five classifiers defined in the following. Results in Table V are weighted average test values for both malicious and benign samples in the testing set.

- Precision: Precision is the number of true positives divided by the number of true positives and false positives. In our experiment, it is the number of malicious (benign) code samples that is labeled correctly divided by the total number of samples that is labeled as malicious (benign). A low precision can also indicate a large number of false positives. Our model has the highest precision value (0.949) among the five models.
- Recall: Recall denotes the probability that an accurate item is retrieved. Put another way, precision can be thought of as the exactness of a classifier, and recall can be considered as the completeness of a classifier. As we could expect, our model has a better recall value (0.948) than the other four models.
- F-measure: The F-measure is the harmonic mean of precision and recall, which conveys the balance between precision and recall. The value ranges from 0 and 1; the closer the F-measure is to 1, the better the classifier would be; on the contrary, the closer the F-measure is to 0, the worse the performance would be. Therefore, the highest F-measure value (0.948) suggests that the SdA-LR model is the one to beat.
- Root mean squared error (RMSE): It is a measure of the differences between a predicted value and an observed value. A small RMSE means better model

Security Comm. Networks 2016; **9**:1520–1534 © 2016 John Wiley & Sons, Ltd.
DOI: 10.1002/sec

Table V. Performance comparison.

Classifier	Precision	Recall	*F*-measure	RMSE	Training time (s)	Testing timec
SdA-LR	**0.949**	**0.948**	**0.948**	**0.213**	1503	**0.34**
RBF SVM	0.936	0.934	0.935	0.249	334.14	42.36
ADTree	0.893	0.909	0.917	0.25	79.67	9.73
RIPPER	0.886	0.887	0.881	0.342	412.04	24.62
Naïve Bayes	0.847	0.849	0.84	0.382	**13.26**	6.83

SdA-LR, stacked denoising auto-encoder with logistic regression; SVM, support vector machine; ADTree, alternating decision tree. The best results are in bold.

performance, and large indicates bad prediction. With this in mind, the smallest RMSE value (0.213) manifests that our model is a better choice compared with other models in Table IV.

As we stated about the confusion matrix in Section 4.4, the purpose of designing a classifier based on machine learning is to operate at a high true positive rate while providing a reasonably low false positive rate. By comparing the confusion matrices in Figure 9 and the confusion matrix of our model in Table IV, we can figure out that the outstanding true positive rates from highest to lowest are in the following order: ours (93.95%), naïve Bayes (93.32%), RBF SVM (91.67%), RIPPER (90.19%), and ADTree (85.08%). While the false positive rates from the lowest to the highest are in this order: ADTree (2.42%), ours (4.13%), RBF SVM (4.92%), RIPPER (12.83%), and naïve Bayes (25.15%). Although the false positive rate of our model is not the best (i.e., our model is ranked in the second place, approximately 1.7 percentage points higher than the first ranked model, however, which has the worst true positive rate), but our model has the highest true positive rate and accuracy. In comprehensive ways, the SdA-LR model would be the optimal one.

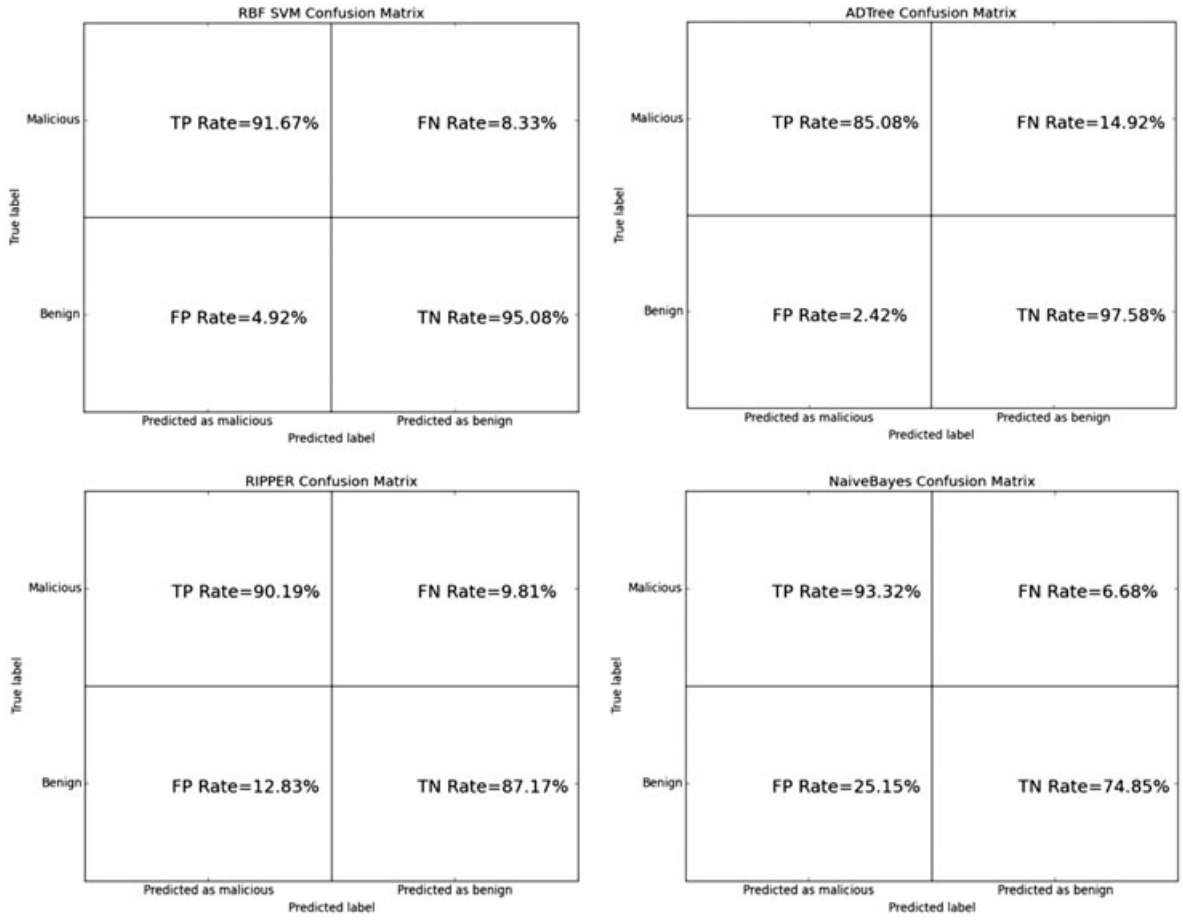


Figure 9. Confusion matrices for the detection models in the work of Likarish *et al.* [6].

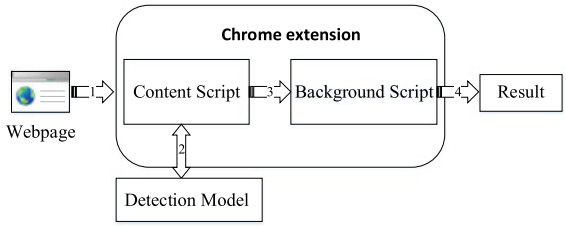


Figure 10. The Google Chrome extension for detecting malicious JavaScript code.

5. CONCLUSION AND DISCUSSION

This paper presents a method using deep features extracted by SdA for classification, to detect JavaScript on webpages as either malicious or not. Experimental results indicated that features extracted by our SdAs were useful for pattern classification, and the SdA helped to improve the accuracy of both logistic regression and the SVM classifiers. When compared with other feature extraction methods such as principle component analysis, independent component analysis, and FA, the SdA has the highest classification accuracy.

What is more, the proposed SdA-LR model was verified to have higher statistical evaluations than the existing methods. In addition, it was shown in our experimental results that building the deep architecture network with three layers of auto-encoders and 250 hidden units in each layer was the optimal choice for our JavaScript code classification. The limitation of the SdA-LR is the long training time; however, the high-speed testing time makes up for this deficiency. The classifier has some other potential drawbacks. One obvious drawback is that the classifier is likely to identify a small number of good JavaScript codes as latently bad. JavaScript codes in some websites have been packed to compress as well as to obfuscate; the reason for doing this is to reduce the size of the initial JavaScript code and to make it more difficult for one to find out what happened in the code and to steal their source code. Some benign packed JavaScript codes are the most likely to be categorized as malicious by the classifier. Namely, benign packed JavaScript code might yield a false positive to a large extent.

There are still potential possibilities where this work can be carried on further, some of which are presented in the following.

(1) Utilizing a graphical processing unit (GPU) to accelerate calculations. Using a GPU would speed up learning process of the algorithm and reduce the time to build the deep learning models. And it also would improve the prediction time and results. However, the major challenge of using a GPU would be to rewrite the algorithm so that it would make an effective utilization of hardware.
(2) Making use of the SdA architecture to build a robust prediction system for real applications. In practice, there exist many problems, such as missing data and data noise, which would make the conventional machine learning-based prediction system not practical. Thanks to the design principle of the SdA, it is of great use to utilize the robustness of the SdA for building a real system. Our model could be used as a browser extension that inspects JavaScript code on a webpage. For example, as shown in Figure 10, a Google Chrome extension could be built using our model. Specifically speaking, the main procedures of the Chrome extension can be outlined in the following:
(3) Crawl the webpage and extract the JavaScript code in the webpage.
(4) The Content Script in the Chrome extension sends the extracted JavaScript code to the detection model. The model first parses the features and then responds with whether the webpage is malicious or not.
(5) The Background Script in the Chrome extension receives the response from the Content Script.
(6) The Background Script displays the result to the user.
(7) Extending our approach in order to detect malware written in other programming languages, such as VBScript, Java, C, and C++. Because there is no need for us to manually craft features, the work flow for detecting malicious code written in other languages would be similar with that in our paper. Briefly speaking, we feed the model with training samples (e.g., VBScript code samples) and let the model itself learn the sample features and model parameters. Then, the detection model could be constructed by using the correlative features and parameters. Ultimately, the detection process could be implemented via the ready-built model.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge funding from the Shaanxi Science and Technology Industrial Research Office under the project 2015GY015.

REFERENCES

1. Symantec. Internet security threat report: trends for 2013. vol. 19, Symantec Inc., 2014.
2. Cova M, Kruegel C, Vigna G. Detection and analysis of drive-by-download attacks and malicious JavaScript

code. In Proc. of the 19th Int. Conf. on World Wide Web, New York, USA, 2010; pp. 281–290.
3. Schwenk G, Bikadorov A, Krueger T, Rieck K. Autonomous learning for detection of JavaScript attacks: vision or reality?. In *Proc. AISec.,* Raleigh, North Carolina, USA, Oct. 19, 2012; pp. 93–104.
4. Kapravelos A, Grier C, Chachra N, Kruegel C, Vigna G. Paxson V. Hulk: eliciting malicious behavior in browser extensions. 23rd USENIX Security Symposium, San Diego, CA, Aug. 2014; pp. 641–654.
5. Alazab M, Venkatraman S, Watters P, Alazab M. Information security governance: the art of detecting hidden malware. In *IT Security Governance Innovations: Theory and Research*, Mellado D, Sánchez L, FernándezMedina E, Piattini M (eds). IGI Global: headquartered in Hershey, Pennsylvania (USA), 2012; pp. 293–315.
6. Likarish P, Jung E, Jo I. Obfuscated malicious JavaScript detection using classification techniques. In 4th International Conference on Malicious and Unwanted Software (MLWARE), Oct., 2009; pp. 47–54.
7. Fraiwan M, Al-Salman R, Khasawneh N, Conrad S. Analysis and identification of malicious JavaScript code. *Information Security Journal: A Global Perspective* 2012; **21**(1):1–11.
8. Curtsinger C, Livshits B, Zorn B, Seifert C. Zozzle: fast and precise in-browser JavaScript malware detection. In Proc. of the 20th USENIX Conf. on Security, Aug. 2011.
9. Rieck K, Krueger T, Dewald A. Cujo: efficient detection and prevention of drive-by-download attacks. *In 26th Annual Computer Security Applications Conference*, Austin, Texas, USA, Dec. 2010, pp. 31–39.
10. Kim HG, Kim DJ, Cho SJ, Park MJ, Park MY. Efficient detection of malicious web pages using high-interaction client honeypots. *Journal of Information Science and Engineering* 2012; **28**(5):911–924.
11. Alosefer Y, Rana O. Honeyware: a web-based low interaction client honeypot. The Third Int. Conf. On Software Testing, Verification, and Validation, Paris, France, April 2010; pp. 410–417.
12. Egele M, Kirda E, Kruegel C. Mitigating drive-by-download attacks: challenges and open problems. IFIP WG 11.4 International Workshop, Zurich, Switzerland, April 2009; pp. 52–62.
13. Ratanaworabhan P, Livshits B, Zorn B. Nozzle: a defense against heap-spraying code injection attacks. In Proc. of USENIX Security Symposium, 2009; pp. 169–186.
14. Canali D, Cova M, Vigna G, Kruegel C. Prophiler: a fast filter for the large-scale detection of malicious web pages. In Proc. of the Int. World Wide Web, Hyderabad, India, 2011; pp. 197–206.
15. Alazab M, Venkatraman S, Watters P, Alazab M. Zero-day malware detection based on supervised learning algorithms of API call signatures. AusDM' 11 Proc. of the Ninth Australasian Data Mining Conference, vol. **121**, pp. 171–182.
16. Huda M, Abawajy J, Alazab M, Abdollalihian M, Islam R, Yearwood J. Hybrids of support vector machine wrapper and filter based framework for malware detection. *Future Generation Computer Systems* 2014. doi:10.1016/j. future.2014.06.001.
17. Alazab M. Profiling and classifying the behavior of malicious codes. *Journal of Systems and Software* 2015; **100**:91–102.
18. AL-Taharwa IA, Lee H, Jeng AB, Wu K, Ho C, Chen S. JSOD: JavaScript obfuscation detector. *Security Comm. Networks* 2015; **8**:1092–1107. doi:10.1002/sec. 1064.
19. Soska K, Christin N. Automatically detecting vulnerable websites before they turn malicious. 23rd USENIX Security Symposium, San Diego, CA, Aug. 2014; pp. 625–640.
20. Bengio Y, Lamblin P, Popovici D, Larochelle H. Greedy layer-wise training of deep networks. In Proc. Neural Inf. Process. Syst., Cambridge, MA, USA, 2007; pp. 153–160.
21. LeRoux N, Bengio Y. Deep belief networks are compact universal approximators. *Neural Computation* 2010; **22**(8):2192–2207.
22. LeCun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 1998; **86**:2278–2324.
23. Vincent P, Larochelle H, Lajoie I, Bengio Y, Manzagol P. Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research* 2010; **11**(12):3371–3408.
24. Salakhutdinov R, Hinton GE. Deep Boltzmann machines. In Proc. Int. Conf. Artif. Intell. Statist, Clearwater Beach, FL, USA, 2009; pp. 448–455.
25. Hinton GE, Osindero S, The Y. A fast learning algorithm for deep belief nets. *Neural Computation* 2006; **18**(7):1527–1554.
26. Silberer C, Lapat M. A learning grounded meaning representations with autoencoders. In Proc of the 52nd Annual Meeting of the Association for Computational Linguistics, Maryland, USA, June 2014; pp. 721–732.
27. Glorot X, Bengio Y. Understanding the difficulty of training deep feedforward neural networks. Proc. Int. Conf. Artificial Intelligence and Statistics, Chia Laguna Resort, Sardinia, Italy, 2010; pp. 249–256.
28. Softmax regression. http://deeplearning.stanford.edu/wiki/index.php/Softmax_Regression, accessed 7 April 2014.

29. Heritrix. http://crawler.archive.org/index.html.
30. Li P, Hastie TJ, Church KW. Very sparse random projections. In Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, USA, 2006; pp. 287–296.
31. Kazemian HB, Ahmed S. Comparisons of machine learning techniques for detecting malicious webpages. *Expert Systems with Applications* 2015; **42**(3):1166–1177.
32. Shyu M, Chen S, Sarinnapakom K, Chang L. A novel anomaly detection scheme based on principal component classifier. In Proc. of the IEEE Foundations and New Directions of Data Mining Workshop, Melbourne, Florida, USA, Nov. 2003; pp. 172–179.
33. Hyvarinen A, Oja E. Independent component analysis: algorithms and applications. *Neural Networks* 2000; **13**(4-5):411–430.
34. Yong AG, Pearce S. A beginner's guide to factor analysis: focusing on exploratory factor analysis. *Tutorials in Quantitative Methods for Psychology* 2013; **9**(2):79–94.