The Journal of Systems and Software 137 (2018) 67–77

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

New deep learning method to detect code injection attacks on hybrid applications

Ruibo Yan[a], Xi Xiao[a], Guangwu Hu[b,*], Sancheng Peng[c], Yong Jiang[a]

[a] Graduate School at Shenzhen, Tsinghua University, Shenzhen, China
[b] School of Computer Science, Shenzhen Institute of Information Technology, Shenzhen, China
[c] School of Informatics, Guangdong University of Foreign Studies, Guangzhou, China

ARTICLE INFO

Article history:
Received 25 November 2016
Revised 8 September 2017
Accepted 3 November 2017
Available online 14 November 2017

Keywords:
Code injection
Hybrid application
Abstract syntax tree
Deep learning

ABSTRACT

Mobile phones are becoming increasingly pervasive. Among them, HTML5-based hybrid applications are more and more popular because of their portability on different systems. However these applications suffer from code injection attacks. In this paper, we construct a novel deep learning network, Hybrid Deep Learning Network (HDLN), and use it to detect these attacks. At first, based on our previous work, we extract more features from Abstract Syntax Tree (AST) of JavaScript and employ three methods to select key features. Then we get the feature vectors and train HDLN to distinguish vulnerable applications from normal ones. Finally thorough experiments are done to validate our methods. The results show our detection approach with HDLN achieves 97.55% in *accuracy* and 97.60% in *AUC*, which outperforms those with other traditional classifiers and gets higher average precision than other detection methods.

1. Introduction

Mobile applications (apps) are gaining significant popularity as mobile devices are getting more and more popular. The smart phone is not only a communication tool employed by users and business, but also a means of planning and organizing people's work and private life. More and more researchers focus on mobile apps Bagheri et al. (2016). Therefore, mobile security has become increasingly important in mobile computing Xiao et al. (2012). Among all the mobile operating systems, Android is open-source and has accounted for around 85 percent of all smartphone sales worldwide in the beginning of 2016 (Global mobile os market share in sales to end users from 1st quarter 2009 to 1st quarter 2016, 2017). Android is so popular, so in this paper, we mainly focus on Android, but the idea can also be applied to other mobile operating systems.

A hybrid application is one that combines elements of both native and HTML5 applications. There are three main mobile operating systems, Android, iOS and Windows Phone. Traditionally, in order to develop mobile apps running on different systems, developers have to use different programming languages and Application Program Interfaces (APIs), which costs much effort. While using a hybrid application framework which supports cross-platform mobile app development, developers can easily build hybrid apps for different mobile operating systems with HTML, CSS and JavaScript. Hybrid HTML5 application development gains more and more momentum in mobile industry because of its convenience in developing cross-platform apps. There are many hybrid application frameworks, such as Phonegap (2017), which utilize functions of web (2017). It can display web pages and execute JavaScript functions. Since JavaScript code and data can be mixed together in web technology, once the malicious JavaScript code is injected and executed by JavaScript engine through some channels, the code injection attack (Jin et al., 2014b) happens in the hybrid apps. In this paper, our purpose is to judge whether a hybrid application has code injection vulnerabilities. The idea can be applied in mobile application markets, such as Google Play market. If a hybrid application has code injection vulnerabilities, we can tell the enterprise who owns the application to avoid financial loss and user privacy leaking.

The code injection attack also happens in PC web applications, which is called "Cross Site Scripting" (XSS). The code injection attack in hybrid apps inherits the fundamental cause of XSS, but it uses many more channels to inject malicious code than XSS. These channels are unique in mobile devices, including Contact, SMS, Barcode, MP3 metadata, etc. The code injection attack on HTML5-based apps is firstly proposed in Jin et al. (2014b) and explained in depth in Jin et al. (2014a). The work in Jin et al. (2014a) adopted static data-flow analysis to detect these attacks. It got the precision

* Corresponding author.
E-mail addresses: yrb15@mails.tsinghua.edu.cn (R. Yan), xiaox@sz.tsinghua.edu.cn (X. Xiao), hu.guangwu@sz.tsinghua.edu.cn, huguangwu@gmail.com (G. Hu), psc346@aliyun.com (S. Peng), jiangy@sz.tsinghua.edu.cn (Y. Jiang).

https://doi.org/10.1016/j.jss.2017.11.001
0164-1212/© 2017 Elsevier Inc. All rights reserved.

of 97.7%. But it costs much time. We proposed classification methods of machine learning to judge whether an application is vulnerable Xiao et al. (2015). Our previous method reduced the time but only obtained the *precision* of 95.3%.

In this paper, we employ a novel deep learning network, Hybrid Deep Learning Network (HDLN) to detect these attacks. At first, based on our previous work, we extract more features from Abstract Syntax Tree of JavaScript and use three methods to select key features. Then on the basis of feature vectors, HDLN is trained to distinguish vulnerable applications from normal ones by thorough experiments. The results show our detection approach with HDLN achieves the overall *accuracy* of 98.12%, superior to those with other traditional classifiers and gets higher average *precision* than the other detection methods (Jin et al., 2014a; Xiao et al., 2015).

Our contributions primarily lie in the following aspects. First of all, we develop a novel deep learning network, Hybrid Deep Learning Network (HDLN), and use it to detect the code injection vulnerability in hybrid applications. Our method improves both the *precision* and *accuracy* and performs far more better than the method in Xiao et al. (2015). Second, we extract new features from the AST of JavaScript in a hybrid application and use information gain, Chi-square test and document frequency to select key features. Third, we extend our dataset and conduct comprehensive experiments with different feature selection methods and different network structures and the results show the extended features and the feature selection methods can improve the detection performance.

Compared to our previous work (Xiao et al., 2015), at first we add n-grams generated from the AST of JavaScript as new features and use three methods to choose key features. Furthermore, a novel deep learning network, Hybrid Deep Learning Network (HDLN) is used to do the detection. In addition, thorough experiments are conducted to test our new network in this work.

The rest of the paper is organized as follows. In Section 2, we describe the background of code injection attacks. The detection model including HDLN is explained in detail in Section 3. In Section 4, we discuss the results of different experiments and provide an overview of related work in Section 5. At last, we conclude this paper and suggest future work in Section 6.

2. Background

It is well known that there is a dangerous characteristic in the web technology: it allows data and code to be mixed together. So hybrid apps developed by web technologies may have code injection vulnerabilities. In this section, we first describe technologies related to the code injection attack. Then we introduce Abstract Syntax Tree from which we extract features. Finally, we state deep learning neural networks.

2.1. Code injection in hybrid applications

There are three types of mobile applications: Native apps, HTML5 apps and Hybrid apps. Native apps are specific to a given mobile operating system using the specific programming language and Software Development Kit (SDK) that the respective platform supports. HTML5 apps are developed with standard web technologies-HTML5, JavaScript and CSS. This write-once-run-anywhere approach creates cross-platform mobile apps that run on multiple platforms. A hybrid application is one that combines elements of both native and HTML5 applications. Hybrid apps embed HTML5 apps inside a thin native container, combining the best of both the native and HTML apps. There are many frameworks that allow developers to create hybrid apps easily. Among them, Phonegap (2017) is the most popular one which uses the functions of WebView web (2017). There is an API addJavaScriptInterface() in WebView to allow the JavaScript code to call the native Java code. If an application has declared the required permissions, the app can access the system resources by calling the responding native code. PhoneGap does this work for developers, it provides many plugins in JavaScript language for developers to access system resources.

Since it allows data and JavaScript code to be mixed together in the web technology, code injection attacks can happen in hybrid apps. For example, when a user uses a hybrid application to scan a barcode and the content of the barcode is a piece of malicious JavaScript code to constantly send the user's locations to a specific server, the malicious code is injected and executed by JavaScript engine, and the user's location privacy is stolen by attackers. A hybrid application with code injection vulnerabilities should satisfy two conditions: firstly, the hybrid application reads unsafe data from outside or inside the device, such as scanning barcodes, Contacts and so on; secondly, the hybrid application shows the unsafe data with unsafe JavaScript APIs, such as “document.write()”.

There are many channels for code injection attacks on hybrid apps. Besides the web channel, there are some data channels unique to mobile devices, such as barcodes, RFID tags and SMS messages, metadata of multimedia files and ID channels. A more hidden code injection attack based on JavaScript coding is proposed in Xiao et al. (2015), the conclusion is that JavaScript Unicode coding, JavaScript hexadecimal coding and JavaScript octal coding are suitable for code injection.

2.2. Abstract syntax tree

In computer science, an Abstract Syntax Tree is a tree representation of the abstract syntactic structure of source code written in a programming language Abstract syntax tree (2017). Each node of the tree represents a construct in the source code. In other words, the AST is a structured description of the source code, which can be used to do static analysis on JavaScript code. Esprima (2017) is a high performance, standard-compliant JavaScript parser written in JavaScript to extract the AST from JavaScript code. Suppose we have the following piece of JavaScript code in List 1, let us see how to generate the AST.

We treat the above JavaScript code as a string and pass it to “esprima.parse”. Subsequently, the AST in JavaScript Object Notation (JSON) format is obtained. We program to parse the JSON string and transform it into an abstract syntax tree like Fig. 1. In this tree, there are six paths from the root node to leaves. The six paths are called as *path0, path1, path2, path3, path4*, and *path5*. *path0* is “Program VariableDeclaration VariableDeclarator Identifier”. The remained five paths are in the same format as *path0*. The height of the tree in Fig. 1 is 5 and the width is 3.

2.3. Deep learning neural networks

Artificial neural network is firstly proposed and consequently deep learning frameworks such as convolutional neural network and recurrent neural network are developed. Artifical Neural Network (ANN) model is proposed by McCulloch and Pitts (1943). Artificial neural neworks are typically organized in layers. The neurons of the input layer, hidden layers and the output layer are fully

```
var answer = 6 ∗ 7;
function test()
{
        alert(”HelloWorld!”);
}
```

Listing 1. JavaScript code.

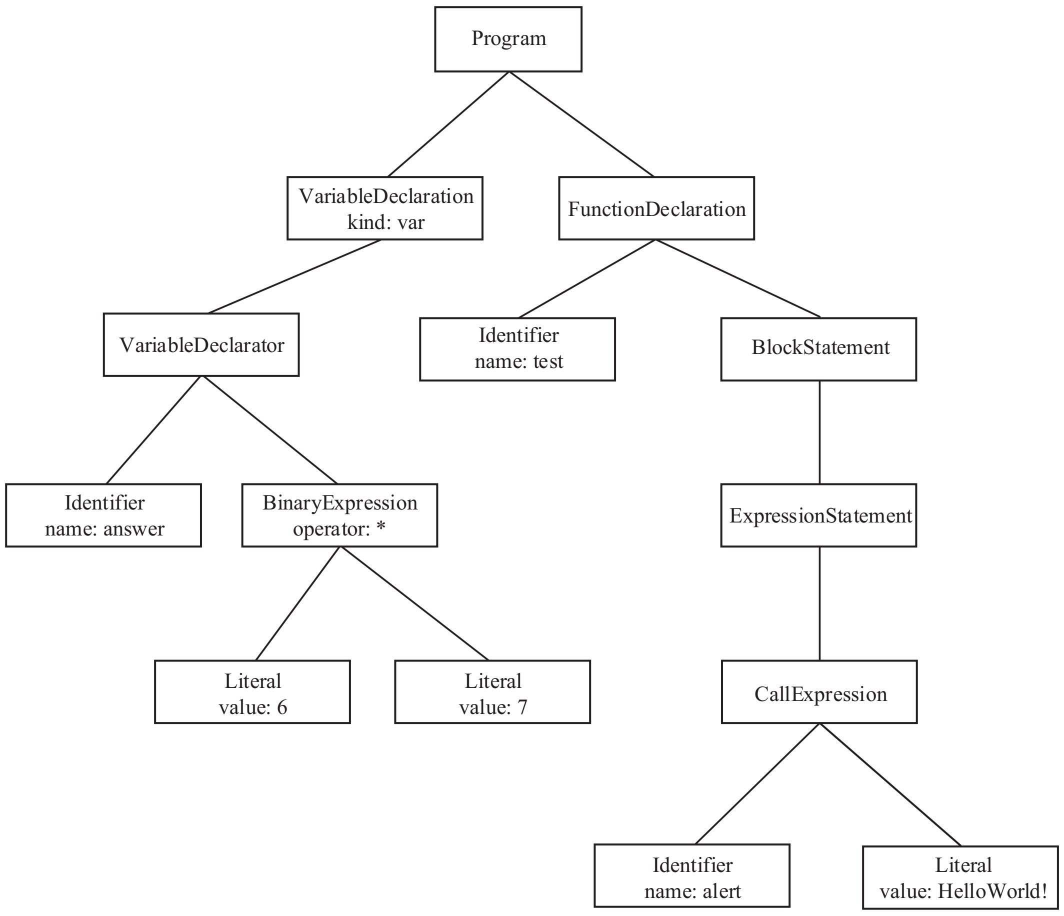


Fig. 1. Abstract Syntax Tree Example.

connected. Thus one hidden layer in ANN is called fully-connected layer (FC layer). In the training process, the network performs forward propagation then conducts backpropagation to update the weights of the network.

A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (Conv layers) following with a pooling layer (subsampling) and then connecting with one or more fully-connected layers (LeCun et al., 1998). There are three main types of layers in CNN architectures: convolutional layer, pooling layer and fully-connected layer. In the convolutional layer, each neuron is connected to a small region of the input neurons. In the pooling layer, it performs a subsampling process. CNN is mainly used for computer vision area (such as image classification) and natural language processing.

The Recurrent Neural Network (RNN) is a type of artificial neural network designed to recognize patterns in sequences of data, such as text, sentences, the spoken word, stock markets and speech. Recurrent networks take as their input not only the current input, but also what they perceived one step back in time. The decision a recurrent network reached at time step $t-1$ affects the decision it will reach one moment later at time step t. It is known that recurrent networks have memory. The purpose of adding memory to neural networks is that there is information in the sequence itself. The sequential information is kept in the hidden state of RNN.

In the mid-90s, a variation of RNN with Long Short-Term Memory (LSTM) units was proposed to solve the vanishing gradient problem (Hochreiter and Schmidhuber, 1997). LSTMs help preserve the error that can be backpropagated through time and layers. By maintaining a more constant error, they allow RNN to continue to learn over many time steps (over 1000), thereby opening a channel to link causes and effects remotely. LSTMs contain information outside the normal flow of RNN in a gated cell. Information can be stored in, written to, or read from a cell, much like data in a computers memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via the open gate, close gate and forget gate.

3. Detection model

Based on our previous work in Xiao et al. (2015), we propose a more efficient model in this paper. Besides the features in Xiao et al. (2015), we extract new features from Abstract Syntax Tree of JavaScript in hybrid applications. Then three powerful feature selection methods are used to select key features and finally a classifier of hybrid deep learning network is employed to detect vulnerable apps. Next, we will introduce our new model in three parts: feature space generation, feature selection and hybrid deep learning network. The detection model is shown in Fig. 2. In feature space generation, we state how to generate the feature space (all features) from the dataset. Three feature selection methods are adopted on the feature space to select key features in feature selection. Finally, we introduce the novel hybrid deep learning network.

3.1. Feature space generation

In our previous work Xiao et al. (2015), 14 permissions, 8 PhoneGap plugin functions and 11 unsafe JavaScript display functions are chosen as features to train a classifier. Since the code injection attack is closely related to JavaScript code in hybrid

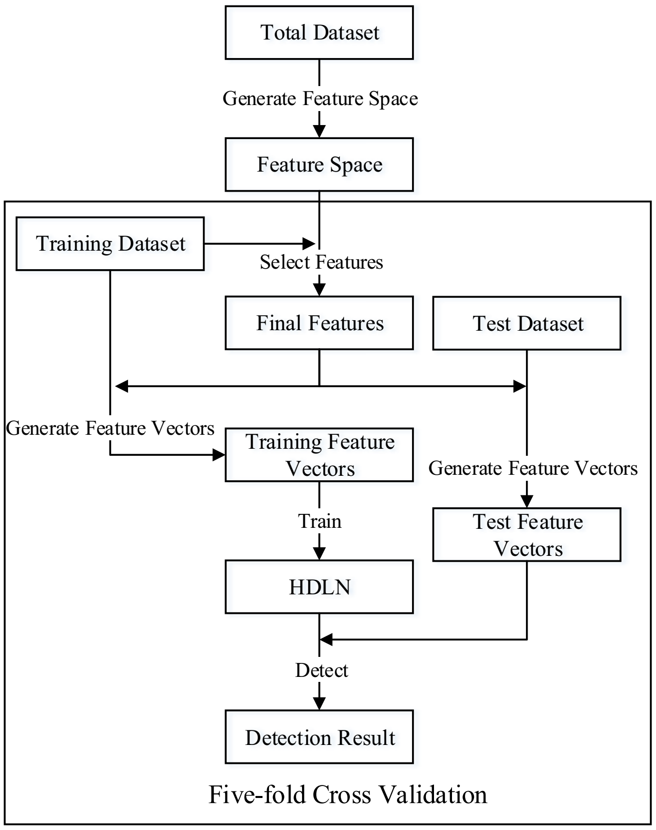


Fig. 2. Detection Model.

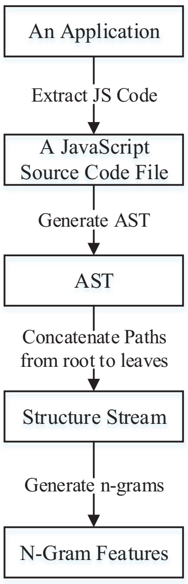


Fig. 3. Generation of N-Gram Features.

apps, we can extract more features from the AST of self-written JavaScript code. The procedure of feature space generation is shown in Fig. 3.

To use the classifier, every application must be represented by a feature vector. To achieve this goal, we do much preprocessing work. Firstly, we extract all JavaScript code written by developers (JavaScript libraries like JQuery are excluded) in an application.

Then we use Esprima to get the AST of JavaScript code in one application. Since the AST is in the format of JSON, we need to parse the JSON string to a multi-tree. We use "type" value of each node to produce a structure stream, and then get n-grams from the stream. Since one node of the AST denotes a construct in the source code, one n-gram represents n constructs and describes more parts of source code than one node of the AST (one node of the AST means 1-gram). Suppose we get the AST in Section 2.2 in Fig. 1, let us see how to generate the structure stream and *n*-grams. As we mentioned before, there are six paths from the root node to leaves of the tree in Fig. 1. We concatenate all paths with backspace to form a stream. We name the stream as "structure stream", because the content of the stream is the "type" information which describes the structure of JavaScript source code. Here the structure stream is "Program VariableDeclaration VariableDeclarator BinaryExpression Literal Program VariableDeclaration VariableDeclarator BinaryExpression Literal Program VariableDeclaration VariableDeclarator Identifier Program FunctionDeclaration BlockStatement ExpressionStatement CallExpression Identifier Program FunctionDeclaration BlockStatement ExpressionStatement CallExpression Literal Program FunctionDeclaration Identifier".

Finally, we use a sliding window to extract *n*-grams as features. The sliding window has a fixed size and moves on a structure stream from beginning to end. Fig. 4 shows the generation process of trigram features, where the sliding window length is three and the moving step is one. The sliding window length determines the length of a gram and the moving step determines how many words to move forward after generating one gram. In our experiment, the length of a gram is three and the moving step is one. All the n-grams from different applications are treated as features, which we call the n-gram features. Besides the n-gram features of the AST, we also consider the height and width of the AST as two additional features, which can be obtained through depth-first-search and breadth-first-search algorithms. In this way, we get the feature space containing 2693 trigram features, the height and width of the AST, and 33 features in our previous work (Xiao et al., 2015).

The reason why we extract new features from the AST is that code injection attacks are closely associated with JavaScript code. The AST can represent the structure of JavaScript code and can reflect function call relation to a certain degree. The trigrams can represent function call relation of depth 3.

3.2. Feature selection

Feature Selection (variable elimination) helps in understanding data, decreasing computation requirement, reducing the effect of curse of dimensionality and improving the predictor performance in machine learning applications (Chandrashekar and Sahin, 2014). As we have mentioned before, the *n*-gram features are a set of *n*-grams generated from all apps in the dataset. Finally, 2693 distinct trigrams are obtained. The number is so huge that feature selection is required.

There are many feature selection methods in text classification which can be used for *n*-gram selection, such as information gain, gain ratio, document frequency, chi-square test, and term frequency-inverse document frequency. In this paper, information gain, chi-square test and document frequency are chosen to select key features from *n*-gram features. We treat the *n*-gram set generated from an application as a document and regard each *n*-gram as a term. Next, we will introduce how the three feature selection methods work.

3.2.1. Information gain

Information Gain (IG) is a frequently used feature selection method in machine learning area. It measures the number of bits of information obtained for category prediction by knowing the

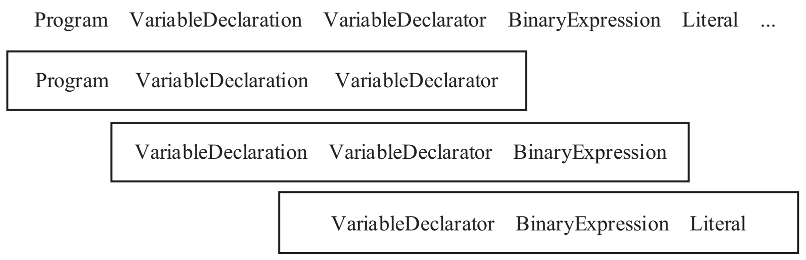

Program VariableDeclaration VariableDeclarator BinaryExpression Literal ...
Program VariableDeclaration VariableDeclarator
VariableDeclaration VariableDeclarator BinaryExpression
VariableDeclarator BinaryExpression Literal

Fig. 4. Generation of Trigram.

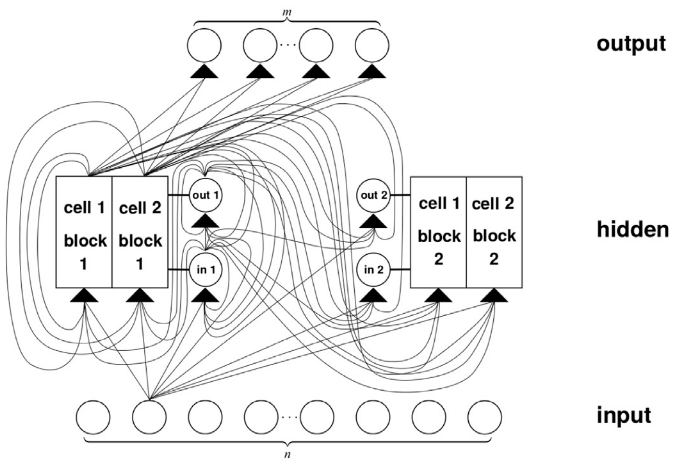


Fig. 5. LSTM Layer.

presence or absence of a term in a document (Yang and Pedersen, 1997). In detail, it is a method that measures the decrease in entropy when the feature is given rather than not given (Dasgupta et al., 2007). Let $\{c_1, c_2\}$ denote the set of classes in the target space. The information gain of a specific n-gram t is defined to be:

$$\begin{aligned} G(t) = &-\sum_{i=1}^{2} P_r(c_i)\log P_r(c_i) \\ &+P_r(t)\sum_{i=1}^{2} P_r(c_i|t)\log P_r(c_i|t) \\ &+P_r(\bar{t})\sum_{i=1}^{2} P_r(c_i|\bar{t})\log P_r(c_i|\bar{t}) \end{aligned} \tag{1}$$

In the above equation, t represents a specific n-gram. $P_r(c_i)$ represents the probability that one document (n-grams set generated from an application) belongs to class c_i (vulnerable or normal). $P_r(t)$ denotes the probability that term t exists in a document and $P_r(\bar{t})$ represents the probability that term t does not exist in a document. $P_r(c_i|t)$ denotes the probability that term t exists in class c_i and $P_r(c_i|\bar{t})$ represents the probability that term t does not exist in class c_i. If the information gain value of a term is large, the uncertainty of the class variable decreases significantly. For all the n-grams, we compute the information gain of each n-gram, sort n-grams in decreasing order of information gain and select the top 800 as key features.

3.2.2. Chi-square test

Chi-square test (CHI) measures the lack of independence between a specific n-gram t and the category c and can be compared to the χ^2 distribution with one degree of freedom to judge extremeness (Yang and Pedersen, 1997). Let t denote the n-gram to be evaluated, c represent one of the classes. And A is the number of documents t and c co-occur, B is the number of times t occurs without c, C is the number of times c occurs without t, D is the number of times both t and c do not appear and N is the total number of documents. Here, N is the total number of apps. The average chi-square value of term t is defined to be

$$\chi^2(t,c) = \frac{N \times (AD - CB)^2}{(A+C)\times(B+D)\times(A+B)\times(C+D)} \tag{2}$$

$$\chi^2_{avg}(t) = \sum_{i=1}^{2} P_r(c_i)\chi^2(t,c_i) \tag{3}$$

If the average chi-square value of a term is very large, it means the class variable is more related to the term. In this paper, we compute the average chi-square value of each n-gram and select the top 800 n-grams with larger chi-square values.

3.2.3. Document frequency

Document Frequency (DF) is the number of documents in which a term appears (Yang and Pedersen, 1997). If a term frequently appears in documents of a specific class, it means the term is more related to the class. The document frequency $DF(t, c)$ is defined as below:

$$DF(t,c) = \frac{\log(DF_t)}{\log(N_c)} \tag{4}$$

DF_t means the number of documents that term t and class c co-occur. N_c denotes the number of documents of class c. In the training dataset, there are two classes of apps, c_1 and c_2. c_1 is the normal class and c_2 is the vulnerable class. We compute the document frequencies for n-grams in each class separately and sort them in decreasing order. Then 400 top n-grams are chosen from each class and if one n-gram exists in both classes, only one is selected. In this way, it guarantees that the feature distributions are balanced in each class.

In our previous work (Xiao et al., 2015), there are 33 features including 14 permissions, 8 PhoneGap plugin functions and 11 unsafe JavaScript display functions. At last, we combine the key features from the n-gram set, the previous features and the height and width of the AST to be the final features. The number of the final features by IG, CHI and DF is 835, 835, 487, respectively. For the 33 features in Xiao et al. (2015), if the specific permission or the specific JavaScript function exists in JavaScript source code of an application, we use number 1 to denote it, otherwise 0. As for the key features, the same idea is employed. If the specific trigram exists in the n-gram set of an application, the value of the feature is set to 1, otherwise 0. And the height and width of the AST are integers. In this way, we can obtain the feature vectors of all the applications.

3.3. Novel hybrid deep learning network

The classification method consists of two phases: the training phase and the detection phase. As for the labels of the dataset, we

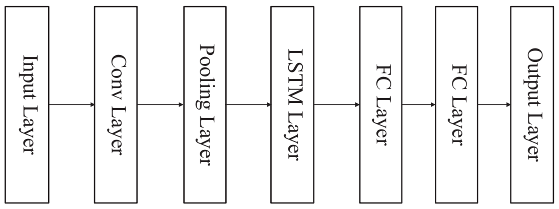


Fig. 6. Structure of HDLN.

use number 1 to denote a vulnerable application and number 0 for a normal application. In this work, we use a hybrid network to classify whether an application is vulnerable. This hybrid network combines convolutional layers, recurrent layers (Long Short Term Memory layers), and fully connected layers. The feature vectors and the corresponding labels of applications in the training dataset are fed to the hybrid network to train a model, then the trained model is used to test on feature vectors in the test dataset.

HDLN is based on types of neural network introduced in Section 2, containing the convolutional layer, the pooling layer, the LSTM layer and the fully-connected layer. The update method of the weights in HDLN is the same as CNN and LSTM. Next, we will introduce HDLN in detail, including the convolutional layer, the pooling layer, the LSTM layer and the fully-connected layer in detail.

3.3.1. Convolutional layer and pooling layer

In HDLN, the convolutional layer and the pooling layer are combined together. The convolutional layer is the core building block of a CNN. The layer's parameters include a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume (Krizhevsky et al., 2012). In the forward process, each kernel is convolved across the width and height of the input volume, then a dot product between the content of the filter and the input is computed to produce a 2-dimensional activation map of a specific filter.

Pooling is one important concept in CNN. It is a form of non-linear down sampling. There are several non-linear functions to implement pooling, among which max pooling is the most common. In the pooling process, the input image is partitioned into a set of non-overlapping rectangles and in each such sub-region, the maximum is calculated.

3.3.2. LSTM layer

LSTM is a type of recurrent neural network. Recurrent Neural Networks have loops. It considers the sequence information. The decision a recurrent network reached at time step $t-1$ affects the decision it will reach one moment later at time step t. The architecture of the LSTM layer is shown in Fig. 5. Fig. 5 is an example of a net with 8 input units, 4 output units, and 2 memory cell blocks of size 2. And in1 is the input gate, out1 represents the output gate, and cell1/block1 denotes the first memory cell of block 1. LSTMs contain information outside the normal flow of RNN in a gate cell. Information can be stored in, written to, or read from a cell, much like data in a computer's memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via the open gate, the close gate and the forget gate.

3.3.3. Fully-Connected layer

A Fully-Connected (FC) layer is a layer in which neurons have full connections to all activations in the previous layer. If the input of a fully-connected layer is a pooling layer or LSTM layer, we should flatten the input data to fully connect to neurons in the FC layer.

HDLN is composed of the above three types of layers. In HDLN, Conv layer and pooling layer are a combination as a pooling layer is often followed by a Conv layer. We can add the combination of Conv layer and pooling layer, LSTM layer, FC layer to a network randomly. The structure of HDLN is shown in Fig. 6. The HDLN is composed of a Conv layer, a pooling layer, an LSTM layer and two FC layers. Sometimes we need to change the dimension of the dataset matrix when we add a new layer after a previous layer.

The classification method consists of two phases, the training phase and the detection phase. In the training phase, we feed the feature vectors and the corresponding labels of applications in the training dataset to train HDLN. And thus the new network has the ability to recognize the vulnerable application. In the detection phase, feature vectors of applications in the test dataset are input to the trained network. Finally, a label is calculated by the trained model, which indicates the application vulnerable or not. If the label is 1, it means that the application is vulnerable to code injection attacks. Otherwise, the application is normal. The deep learning model uses its huge parameters to represent the classifying methods, so if we train the model properly and get good parameters, it can solve the classifying problem with a high performance. In addition, CNN and RNN can extract implicit features in the training phase.

4. Experiment results and discussion

In this section, the experimental data and evaluation metrics are presented firstly. Then the experimental results are shown in detail. At last, we summarize our experiments and compare our new method with traditional machine learning methods.

4.1. Dataset, environment and evaluation metrics

The experimental dataset in our previous work (Xiao et al., 2015) includes 986 apps, 578 normal apps and 408 vulnerable apps. In this paper, we extend the dataset and obtain 1788 apps including 1336 normal apps and 422 vulnerable apps.

The environment of experiments is introduced subsequently. The operating system is Ubuntu 14.04.2, 8 GB RAM, Intel(R) Core(TM) i5-3210M CPU @ 2.50 GHz. The Python Version is 2.7.6 with Keras (1.1.2). The quality of the machine we conduct our experiments is not so high, so experiments in this paper can be done with low cost.

K-fold cross validation is a common technique for estimating the performance of a classifier by partitioning the original dataset into a training set to train the model, and a test set to evaluate it. In k-fold cross-validation, the original dataset is randomly divided into k equal size subdatasets. For the k subdatasets, a single subdataset is retained as the test dataset for testing the model, and the remaining $k-1$ subdatasets are used as training dataset. Then the cross validation process is repeated k times. In this way, each of the k subdateset is used exactly once as the test dataset. Finally, the k results from the folds are averaged to produce a single estimation. The advantage of this method is that all observations are used for both training and validation, and each observation is used for validation exactly once.

We use 5-fold cross validation to evaluate the model. If we only do the training phase and test phase one time, it would lead to causal error. While using 5-fold cross validation can reduce casual error and assess the model better. As we mentioned before, we have 422 vulnerable apps. If we use 10-fold cross validation, then there are 42 vulnerable apps in the test dataset. Too few positive samples in test dataset will cause the result to have more errors, so we choose 5-fold cross validation. Evaluation metrics can be found in our previous work in Xiao et al. (2015). In this paper, we choose *recall, precision, accuracy* and *AUC*to assess the results of HDLN. The *AUC* value can describe the result of the classifier precisely since the dataset is unbalanced. The definition of evaluation metrics is

Table 1
Results of different Conv layers.

Number of hidden layers	Number of neurons	Normal apps		Vulnerable apps		Overall accuracy (%)	AUC (%)
		Precision(%)	Recall(%)	Precision(%)	Recall(%)		
5	[100c,p,50l,100f,100f]	97.15	99.40	97.97	90.75	97.33	95.78
5	[150c,p,50l,100f,100f]	97.15	99.25	97.56	90.75	97.21	96.27
5	[200c,p,50l,100f,100f]	97.01	99.48	98.22	90.27	97.27	96.03
5	[250c,p,50l,100f,100f]	97.02	99.63	98.73	90.27	97.38	96.78
5	[300c,p,50l,100f,100f]	96.79	98.95	96.67	89.56	96.70	95.82
5	[350c,p,50l,100f,100f]	97.02	99.55	98.44	90.27	97.32	95.98
5	[400c,p,50l,100f,100f]	97.23	99.63	98.74	90.99	97.55	96.02

described in detail in our previous work (Xiao et al., 2015). The dataset is unbalanced, so the AUC metric is chosen to express experimental results. The AUC value is equivalent to the probability that a randomly chosen positive example is ranked higher than a randomly chosen negative example (Fawcett, 2006). The definition of recall is

$$recall = \frac{TP}{TP + FN} \tag{5}$$

The definition of *precision* is

$$precision = \frac{TP}{TP + FP} \tag{6}$$

The definition of *accuracy* is

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{7}$$

4.2. Experimental results

Feature vectors generated with IG are called as IGfeature vectors, the others are called as CHI feature vectors and DF feature vectors. We do experiments by changing architectures of HDLN, using different feature selection methods, and comparing with traditional classifiers and other methods of other papers. We use the number of filters (convolutional kernels) to represent neurons in the convolutional layer. The number of filters is not the actual number of neurons in the convolutional layer, but the number of neurons is determined by the number of filters and the input layer. So the number of filters can represent the number of neurons in the convolutional layer. If a convolutional layer is set to 50 filters, we use “50c” to represent it. In this way, the polling layer is represented as “p”. ”50l” denotes an LSTM layer which has 50 neurons. For the same reason, a fully-connected layer with 50 neurons is called as “50f”.

4.2.1. Results of HDLN

In this part, we state the results of different HDLNs, including different number of filters in the convolutional layer, different number of neurons in the LSTM layer, different number of neurons in the FC layer and different structures.

At first, we keep the number of hidden layers fixed and change the number of filters (convolutional kernels) in the convolutional layer. The number of hidden layers is 5 and the first layer is a convolutional layer following with a pooling layer, and the rest layers are an LSTM layer with 50 neurons and two fully-connected layers with 100 neurons. The experimental result is shown in Table 1. As is shown in the table, we can see that the best *accuracy* can achieve up to 97.55% with 400 filters in the convolutional layer. At the same, the *precision* of the vulnerable apps can achieve 98.74%. The best *AUC* value can achieve 96.78%. As the number of filters increases, the result gets better generally.

Secondly, we keep the number of hidden layers unchanged and change the number of neurons in the LSTM layer. The number of hidden layers is 5 and the first layer is a convolutional layer with 200 filters following with a pooling layer, and the rest layers are an LSTM layer with different number of neurons and two fully-connected layers with 100 neurons. The experimental result is shown in Table 2. As is shown in the table, the best accuracy can achieve up to 97.55% with 300 neurons in the LSTM layer. At the same, the precision of the vulnerable apps is 98.76%. The best *AUC* value can achieve up to 97.13%. In general, as the number of filters increases, the accuracy gets higher.

Thirdly, we keep the number of hidden layers unchanged and change the number of neurons in the two fully-connected layers. The number of hidden layers is 5 and the first layer is a convolutional layer with 200 filters following with a pooling layer, and the rest layers are an LSTM layer with 550 neurons and two fully-connected layers with different number of neurons. The experimental result is shown in Table 3. It can be seen from the table that the best *accuracy* can achieve up to 97.44% with 300 neurons in the FC layer. In the meantime, the precision of the vulnerable apps is 97.74%. The *AUC* value is higher than 95.50%. In the view of *accuracy*, the effect of the number of neurons in the fully-connected layers is very slight.

Finally, we change the architectures of HDLN by adding layers. At first, we add a convolutional layer and a pooling layer to the previous structure of HDLN. Subsequently, we add an LSTM layer. Further more, another LSTM layer and another two FC layers are added in the network. Table 4 shows the experimental result. From the table, best *accuracy* is 97.44% with 8 layers. At the same time, the *precision* of the vulnerable apps is 99.24%. And the *AUC* value is 96.35%. As the number of layers increases, the result gets better then gets worse. It is because there are more parameters to tune with more layers. It is hard to train a satisfactory model with too many parameters.

4.2.2. Results of different feature selection methods

In order to evaluate feature selection methods, the structure of HDLN is kept the same and we use feature vectors generated from different feature selection methods to do the detection. In detail, the architecture of the network is fixed with a convolutional layer with 200 filters following with a pooling layer, an LSTM layer with 550 neurons and two FC layers with 100 neurons. The experimental result is shown in Table 5. From the table, we can see that the best *accuracy* is 97.44% with IG feature selection method. In the meanwhile, the *precision* of the vulnerable apps achieves up to 98.48% but the *recall* is 90.75%. The best feature selection method among the three methods in Section 3 is IG generally.

4.2.3. Comparison with other classifiers

In this part, we show the results of traditional classifiers using IG feature vectors. Bayes, Random Forest, Linear Regression and SVM are utilized to do the classification. The results are shown in Table 6. The last row is the result of HDLN with DF feature selection method. From the table, it can be seen that of the traditional classifiers, the best *accuracy* is 97.55% with the Random Forest classifier. The fact is in complete agreement with our re-

Table 2
Results of different LSTM layers.

Number of hidden layers	Number of neurons	Normal apps		Vulnerable apps		Overall accuracy (%)	AUC (%)
		Precision (%)	Recall (%)	Precision (%)	Recall (%)		
5	[200c,p,200l,100f,100f]	97.36	99.25	97.49	91.46	97.38	96.81
5	[200c,p,250l,100f,100f]	97.08	99.33	97.80	90.51	97.21	96.14
5	[200c,p,300l,100f,100f]	97.23	99.63	98.76	90.99	97.55	96.24
5	[200c,p,350l,100f,100f]	97.36	99.10	97.06	91.46	97.27	96.30
5	[200c,p,400l,100f,100f]	96.95	99.85	99.48	90.03	97.50	95.86
5	[200c,p,450l,100f,100f]	97.23	99.48	98.22	90.99	97.44	96.63
5	[200c,p,500l,100f,100f]	97.07	98.95	96.61	90.51	96.93	97.13
5	[200c,p,550l,100f,100f]	96.74	99.55	98.45	89.32	97.10	95.22
5	[200c,p,600l,100f,100f]	97.22	99.33	97.71	90.99	97.32	97.08

Table 3
Results of different FC layers.

Number of hidden layers	Number of neurons	Normal apps		Vulnerable apps		Overall accuracy (%)	AUC (%)
		Precision (%)	Recall (%)	Precision (%)	Recall (%)		
5	[200c,p,550l,100f,100f]	97.22	99.40	98.01	90.99	97.38	95.73
5	[200c,p,550l,150f,150f]	97.15	99.33	97.79	90.75	97.27	95.64
5	[200c,p,550l,200f,200f]	97.15	99.25	97.47	90.75	97.21	96.20
5	[200c,p,550l,250f,250f]	97.06	98.43	94.93	90.51	96.53	95.63
5	[200c,p,550l,300f,300f]	97.36	99.33	97.74	91.46	97.44	96.15

Table 4
Results of different net structures.

Number of hidden layers	Number of neurons	Normal apps		Vulnerable apps		Overall accuracy (%)	AUC (%)
		Precision (%)	Recall (%)	Precision (%)	Recall (%)		
5	[200c,p,550l,100f,100f]	96.95	99.70	98.97	90.03	97.38	95.58
7	[200c,p,200c,p,550l,100f,100f]	97.08	99.55	98.45	90.51	97.38	95.91
8	[200c,p,200c,p,550l,550l,100f,100f]	96.96	99.78	99.24	90.03	97.44	96.35
10	[200c,p,200c,p,550l,550l,100f,100f,100f,100f]	95.48	99.48	98.10	85.06	96.02	93.10

Table 5
Results of different feature selection methods.

Method	Number of neurons	Normal apps		Vulnerable apps		Overall accuracy (%)	AUC (%)
		Precision (%)	Recall (%)	Precision (%)	Recall (%)		
IG	[200c,p,550l,100f,100f]	97.15	99.55	98.48	90.75	97.44	96.80
CHI	[200c,p,550l,100f,100f]	97.08	99.33	97.75	90.51	97.21	95.98
DF	[200c,p,550l,100f,100f]	97.21	98.88	96.41	90.99	96.99	97.60

Table 6
Results of traditional classifiers.

Classifier	Normal apps		Vulnerable apps		Overall accuracy (%)	AUC (%)
	Precision (%)	Recall (%)	Precision (%)	Recall (%)		
Bayes	97.74	97.01	90.86	92.90	96.02	95.29
Random forest	97.02	99.85	99.49	90.28	97.55	95.13
Linear regression	97.76	98.13	94.00	92.88	96.87	95.50
SVM	97.50	99.18	97.23	91.93	97.44	95.55
BPNN	96.67	99.77	99.20	89.08	97.21	94.54
CNN	96.80	99.55	98.44	89.56	97.15	93.97
RNN	96.81	99.52	98.44	89.57	97.20	95.17
HDLN-IG	97.15	99.55	98.48	90.75	97.44	96.80

sults in Xiao et al. (2015). At the same time, the *precision* of vulnerable apps gets 99.49%. The second best traditional classifier is SVM, with an overall *accuracy* of 97.44% and the *precision* of vulnerable apps is 97.23%. As for the three kinds of DNN classifiers, BPNN (back propagation neural network), CNN and RNN, the *accuracy* values are 97.21%, 97.15%, 97.20% and the *AUC* values are 94.54%, 93.97%, 95.17%, respectively. However, the result of HDLN with IG selection method, the *accuracy* is 97.44%, higher than all the other classifiers. The *AUC* value of HDLN is also higher than all the other classifiers. HDLN also outperforms CNN and RNN. Our new deep learning methods are superior to other classifiers as shown in Table 7.

4.3. Comparison with other methods

The code injection attack on HTML5-based apps is firstly proposed in Jin et al. (2014b) and explained in depth in Jin et al. (2014a). In Jin et al. (2014a), a method was proposed to detect code injection attacks on HTML5-based hybrid apps, of which the precision is 97.7%. The data-flow analysis technique in the method leads high time complexity. The average execution time

Table 7
Results of other methods.

Methods	Methods in Jin et al. (2014a)	Methods in Xiao et al. (2015)	HDLN
Precision(%)	97.70	95.30	97.90

on an app is 15.38 s. Our previous method of machine learning in Xiao et al. (2015) reduced the time but only obtained the precision of 95.3%.

The *precision* of the detection method in Jin et al. (2014a) is 97.70%. And the *precision* of the detection method in our previous work Xiao et al. (2015) achieves 95.30%. In this paper, the precision is 97.90% in average. It shows that our HDLN classifier gets higher average precision than the other detection methods.

In the aspect of time complexity, HDLN needs much time in the training phase, but runs quickly in the test phase. The total time including training and testing of 5-fold is 14510.83 s in average. As we all know, training a neural network is time-consuming. But once we get a good training neural network model, we can save it and use it to test on the coming applications later. So if the model is not updated, it costs very little time in the test phase.

5. Related work

Since it is allowed to mix data and code together in web technology, Cross-Site Scripting (XSS) can happen. Code injection attacks on hybrid apps are like XSS, but they employ unique channels on mobile platforms. In this section, we introduce recent literature on code injection, such as XSS attacks, HTML5 vulnerabilities and attacks on hybrid apps.

5.1. XSS attacks

XSS is a hot topic and quite a lot of works are done on XSS detection. One idea is based on taint analysis, such as the automated system to detect and validate DOM-based XSS vulnerabilities in Lekies et al. (2013), the ACTARUS method in Guarnieri et al. (2011) and the Flowdroid tool in Arzt et al. (2014). There are also many other methods for XSS detection. For example, data mining methods to predict SQL injection and XSS in web applications (Shar and Tan, 2012) and prediction models using both classification and clustering to predict vulnerabilities (Shar et al., 2013) are based on machine learning. Fabien Duchene et al. proposed a way to detect web injection vulnerabilities by generating test inputs with a combination of model inference and evolutionary fuzzing (Duchene et al., 2012). An incremental learning for -Support Vector Regression called INSVR was proposed in Gu et al. (2015b). Experiments show that INSVR can avoid the infeasible updating paths as far as possible, and successfully converges to the optimal solution. An incremental Support Vector Learning for Ordinal Regression was proposed in Gu et al. (2015a) by the same team. An efficient multikeyword fuzzy ranked search scheme that is able to address the aforementioned problems was proposed in Fu et al. (2016). In addition, there are many other approaches to preventing XSS. One is through sanitization. Noxes was put forward to mitigate XSS attacks in Kirda et al. (2006). ScriptGard and XSS-GUARD were proposed to prevent XSS attacks in Saxena et al. (2011); Bisht and Venkatakrishnan (2008). SWAP (Secure Web Application Proxy) Wurzinger et al. (2009) and xJS Athanasopoulos et al. (2010) were raised to prevent code injections in web applications. Philipp Vogt et al. proposed a way to prevent XSS attacks on the client side by tracking the flow of sensitive information inside the web browser Vogt et al. (2007). A client-server based framework that alleviates the dissemination of XSS worms from the OSN was proposed in Chaudhary and Gupta (2017). The injection of sanitization primitives is processed on the client-side. The proposed framework optimized the method of auto-context-aware sanitization and incurs a low and acceptable performance overhead. Another novel framework XSS-Secure was proposed in Gupta and Gupta (2016). It detects and alleviates the propagation of Cross-Site Scripting (XSS) worms from the Online Social Network (OSN) based multimedia web applications on the cloud environment.

5.2. HTML5 vulnerabilities

Maryam et al. proposed PINlogger.js which is a JavaScript-based side channel attack revealing user PINs on an Android mobile phone (Mehrnezhad et al., 2016), PINlogger.js is able to correctly identify PINs with a high success rate which indicates another big JavaScript vulnerability on phones. Later, A new HTML5 potential security vulnerability detection process for web documents based on Hadoop was carried out (Kim et al., 2015). Further, the vulnerabilities were classified as either tag and attribute vulnerabilities or JavaScript vulnerabilities based on the characteristics of its entry. Behnaz et al. proposed a new type of code injection attack called W2AI (web-to-app injection) attacks in Hassanshahi et al. (2015). In this attack, the user is not required to install malicious apps, but merely needs to visit a malicious website in an Android browser. They implemented a tool to protect apps from this attack, which employs a novel combination of static analysis, symbolic execution and dynamic testing. Jian et al. presented a dynamic solution, called MinPerm, to infer the permissions required by hybrid mobile apps (Mao et al., 2016a). This solution can be used to test environment of an app, extract the dynamic permissions required and identify the over-claimed permissions. It is useful to remove additional permissions which attackers often utilize to inject malicious code. Patrick et al. conducted scalable analysis to discover several classes of vulnerabilities in mobile web apps and found that 28% of the studied apps had at least one vulnerability (Mutchler et al., 2015). At the same time, several changes to the Android APIs to mitigate these vulnerabilities were offered. Martin et al. designed a tool called POWERGATE, a new access-control mechanism for Web-based system applications (Georgiev et al., 2015). The tool gives different rights to “application’s own local code” and “third-party Web code” to prevent code injection attacks.

5.3. Attacks on hybrid apps

There are many research works about attacks on hybrid apps. Matthew et al. investigated security concerns in hybrid mobile apps (Hale and Hanson, 2015) and defined a five step process for discovering hybrid specific vulnerabilities. Further, they enumerated three forms of amalgamated attacks that can specifically target and exploit hybrid apps, and created a testbed platform for accessing vulnerabilities. X. Jin et al. described how HTML5-based apps can become vulnerable and how attackers can exploit their vulnerabilities through a variety of channels (Jin et al., 2014b). In Jin et al. (2014a), they transformed the detection problem into an equivalent data-flow analysis problem that seeks to identify a data flow from a code injection channel to an unsafe rendering API. The time complexity of the method is very high and the average execution time on an app is 15.38 s. This methods precision is 97.7%. Jian et al. proposed an approach to detect injected behaviors in HTML5-based Android apps Mao et al. (2016b). This new approach is based on the behavior state machine which captures an apps actions as well as the contexts where actions take place, providing sufficient information about the program context of the apps behaviors. When code injection happens, it will be detected by the deviation from the behavior state machine of the original app. Xiao et al. proposed a new type of code injection attacks based on coding and use machine learning methods to detect code injection attacks in Xiao et al. (2015).

6. Conclusion and future work

In this paper, we extend our previous work in Xiao et al. (2015). Both the experimental dataset and the feature space are extended. We employ a novel deep learning network, Hybrid Deep Learning Network (HDLN), and use it to detect code injection attacks. Besides the features in our initial work, we extract more features from the AST of JavaScript code and adopt three methods to select key features. Then three different feature vectors are obtained to train HDLN to detect whether an application is vulnerable or not. Finally, we do complete experiments to validate our classifier. The results show our detection method with HDLN achieves 97.60% in *AUC*, which outperforms those with traditional classifiers and gets higher average *precision* than the other detection methods.

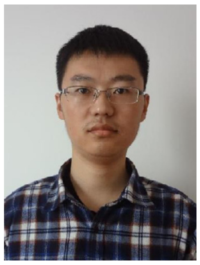
Until now, our work is focused on the static aspects, such as JavaScript code analysis. The dynamic characteristics of hybrid apps are not considered yet. In the future, we can take dynamic traits of hybrid apps into consideration.

Acknowledgment

This work is supported by the NSFC projects (61375054, 61402255, 61202358), the National High-tech R&D Program of China (2015AA016102), Guangdong Natural Science Foundation (2015A030310492, 2014A030313745) and the RD Program of Shenzhen (JCYJ20160531174259309, JCYJ20160301152145171, JCYJ20150630170146831, JSGG20150512162853495, Shenfagai [2015] 986), and Cross fund of Graduate School at Shenzhen, Tsinghua University (JC20140001).

References

Abstract syntax tree. https://wikipedia.org/wiki/Abstract_syntax_tree.
Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM SIGPLAN Not. 49 (6), 259–269.
Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E.P., Karagiannis, T., 2010. XJS: practical XSS prevention for web application development. In: Proceedings of the 2010 USENIX conference on Web application development. USENIX Association, p. 13.
Bagheri, H., Garcia, J., Sadeghi, A., Malek, S., Medvidovic, N., 2016. Software architectural principles in contemporary mobile software: from conception to practice. J. Syst. Softw. 119, 31–44.
Bisht, P., Venkatakrishnan, V., 2008. Xss-guard: precise dynamic prevention of cross-site scripting attacks. In: Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 23–43.
Chandrashekar, G., Sahin, F., 2014. A survey on feature selection methods. Comput. Electr. Eng. 40 (1), 16–28.
Chaudhary, P., Gupta, B., 2017. A novel framework to alleviate dissemination of XSS worms in online social network (OSN) using view segregation. Neural Netw. World 27 (1), 5.
Dasgupta, A., Drineas, P., Harb, B., Josifovski, V., Mahoney, M.W., 2007. Feature selection methods for text classification. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, New York, NY, USA, pp. 230–239.
Duchene, F., Groz, R., Rawat, S., Richier, J.-L., 2012. XSS vulnerability detection using model inference assisted evolutionary fuzzing. In: Proceedings of the-3rd International Workshop on Security Testing SECTEST 2012 (affiliated with ICST). IEEE Computer Society, pp. 815–817.
Esprima. http://esprima.org/.
Fawcett, T., 2006. An introduction to ROC analysis. Pattern Recogn. Lett. 27 (8), 861–874.
Fu, Z., Wu, X., Guan, C., Sun, X., Ren, K., 2016. Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. IEEE Trans. Inf. Forensics Secur. 11 (12), 2706–2716.
Georgiev, M., Jana, S., Shmatikov, V., 2015. Rethinking security of web-based system applications. In: Proceedings of the 24th International Conference on World Wide Web. ACM, pp. 366–376.
Global mobile os market share in sales to end users from 1st quarter 2009 to 1st quarter 2016. http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.
Gu, B., Sheng, V.S., Tay, K.Y., Romano, W., Li, S., 2015. Incremental support vector learning for ordinal regression. IEEE Trans. Neural Netw. Learn. Syst. 26 (7), 1403–1416.
Gu, B., Sheng, V.S., Wang, Z., Ho, D., Osman, S., Li, S., 2015. Incremental learning for ν-support vector regression. Neural Netw. 67, 140–150.
Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., Berg, R., 2011. Saving the world wide web from vulnerable javascript. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, pp. 177–187.
Gupta, S., Gupta, B., 2016. XSS-secure as a service for the platforms of online social network-based multimedia web applications in cloud. Multimed. Tools Appl. 1–33.
Hale, M.L., Hanson, S., 2015. A testbed and process for analyzing attack vectors and vulnerabilities in hybrid mobile apps connected to restful web services. In: Proceedings of the IEEE World Congress on Services. IEEE, pp. 181–188.
Hassanshahi, B., Jia, Y., Yap, R.H., Saxena, P., Liang, Z., 2015. Web-to-application injection attacks on android: characterization and detection. In: Proceedings of the European Symposium on Research in Computer Security. Springer, pp. 577–598.
Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.
Jin, X., Hu, X., Ying, K., Du, W., Yin, H., Peri, G.N., 2014. Code injection attacks on html5-based mobile apps: characterization, detection and mitigation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 66–77.
Jin, X., Luo, T., Tsui, D.G., Du, W., 2014. Code injection attacks on html5-based mobile apps. Comput. Res. Repos. abs/1410.7756.
Kim, I.-A., Cho, K.-H., Yim, H.-J., Kim, H.-K., Lee, K.-C., 2015. Hadoop-based crawling and detection of new html5 vulnerabilities on public institutions web sites. Ind. J. Sci. Technol. 8 (27).
Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N., 2006. NOXES: a client-side solution for mitigating cross-site scripting attacks. In: Proceedings of the 2006 ACM symposium on Applied computing. ACM, pp. 330–337.
Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (Eds.), Advances in Neural Information Processing Systems 25. Curran Associates, Inc., pp. 1097–1105.
LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. Proc. IEEE 86 (11), 2278–2324.
Lekies, S., Stock, B., Johns, M., 2013. 25 million flows later: large-scale detection of DOM-based XSS. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. ACM, pp. 1193–1204.
Mao, J., Ma, H., Chen, Y., Jia, Y., Liang, Z., 2016. Automatic permission inference for hybrid mobile apps. J. High Speed Netw. 22 (1), 55–64.
Mao, J., Wang, R., Chen, Y., Jia, Y., 2016. Detecting injected behaviors in html5-based android applications. J. High Speed Netw. 22 (1), 15–34.
McCulloch, W.S., Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biophys. 5 (4), 115–133.
Mehrnezhad, M., Toreini, E., Shahandashti, S.F., Hao, F., 2016. Stealing pins via mobile sensors: actual risk versus user perception. Comput. Res. Repos. abs/1605.05549.
Mutchler, P., Doupé, A., Mitchell, J., Kruegel, C., Vigna, G., 2015. A large-scale study of mobile web app security. In: Proceedings of the Mobile Security Technologies Workshop (MoST).
Phonegap. http://phonegap.com/.
Saxena, P., Molnar, D., Livshits, B., 2011. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. ACM, pp. 601–614.
Shar, L.K., Tan, H.B.K., 2012. Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, pp. 1293–1296.
Shar, L.K., Tan, H.B.K., Briand, L.C., 2013. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 642–651.
Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G., 2007. Cross site scripting prevention with dynamic data tainting and static analysis. Proceeding of the Network and Distributed System Security Symposium.
Webview. https://developer.android.com.
Wurzinger, P., Platzer, C., Ludl, C., Kirda, E., Kruegel, C., 2009. SWAP: mitigating XSS attacks using a reverse proxy. In: Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems. IEEE Computer Society, pp. 33–39.
Xiao, X., Tian, X., Zhai, Q., Xia, S., 2012. A variable-length model for masquerade detection. J. Syst. Softw. 85 (11), 2470–2478.
Xiao, X., Yan, R., Ye, R., Li, Q., Peng, S., Jiang, Y., 2015. Detection and prevention of code injection attacks on html5-based apps. In: Proceedings of the Third International Conference on Advanced Cloud and Big Data. IEEE, pp. 254–261.
Yang, Y., Pedersen, J.O., 1997. A comparative study on feature selection in text categorization. In: Proceedings of the 14th International Conference on Machine Learning, 97, pp. 412–420.



Ruibo Yan is a graduate student in Graduate School at Shenzhen, Tsinghua University. He got his B.S. degree from Northeastern University in 2015. His research interests focus on information security and machine learning.

Xi Xiao is an associate professor in Graduate School At Shenzhen, Tsinghua University. He got his Ph.D. degree in 2011 in State Key Laboratory of Information Security, Graduate University of Chinese Academy of Sciences. His research interests focus on information security and the computer network.

Guangwu Hu received the Ph.D. degree in computer science and technology from Tsinghua University in 2014. Then he became a post-doc with the Graduate School at Shenzhen, Tsinghua University. He is currently an associate professor with the Shenzhen Institute of Information Technology. His research interests include software-defined networking, next-generation Internet and Internet security.

Sancheng Peng is Professor of Guangdong University of Foreign Studies. He received his Ph.D. degree in Computer Science, from Central South University in 2010. His research interests include network and information security, social networks, trusted computing, and mobile computing. He is a member of CCF and ACM.

Yong Jiang is a professor and a Ph.D. supervisor in Graduate School at Shenzhen, Tsinghua University. He got his Ph.D. degree in Tsinghua University 2002. He mainly engaged in computer network architecture, Internet applications, and Information security. He hosted and participated in a lot of national projects such as 863 project, 973 project and et al. He has published more than 50 papers in top international journals and international conferences.