ISSN (Print) : 0974-6846
Indian Journal of Science and Technology, *Vol 10(19), DOI: 10.17485/ijst/2017/v10i19/114828, May 2017* ISSN (Online) : 0974-5645

Detection of Malicious JavaScript Code in Web Pages

Dharmaraj R. Patil* **and J. B. Patil**

Department of Computer Engineering, R. C. Patel Institute of Technology, Shirpur – 425405, Maharashtra, India; dharmaraj.rcpit@gmail.com, jbpatil@hotmail.com

Abstract

Objective: To detect malicious JavaScript code in Web pages by reducing false positive and false negative rate thus increasing detection rate. **Methods/Analysis:** In recent years JavaScript has become the most common and successful attack construction language. Various approaches have been proposed to overcome the JavaScript security issues. In this paper, we have presented the methodology of detection of malicious JavaScript code in the Web pages. We have collected the benign and malicious JavaScript's from the benchmark sources of Web pages. We have used the static analysis of JavaScript code for the effective detection of malicious and benign scripts. We have created a dataset of 6725 benign and malicious scripts. This dataset consists of 4500 benign and 2225 malicious Java Script's. **Finding:** We have extracted 77 JavaScript features from the script, among which 45 are new features. We have evaluated our dataset using seven supervised machine learning classifiers. The experimental results show that, by inclusion of new features, all the classifiers have achieved good detection rate between 97%-99%, with very low FPR and FNR, as compared to nine well-known antivirus software's. **Novelty/Improvement:** We have used 45 new JavaScript features in our dataset. Due to these new features, FPR and FNR are reduced and increase the malicious JavaScript detection rate.

Keywords: Drive-by-Downloads, Malicious JavaScript, Machine Learning, Malicious Web Pages, Static Detection

1. Introduction

JavaScript is a dynamic client-side scripting language used to create content for the Web along with HTML and CSS. It is used by the most of the Websites. It is also supported by all the modern Web browsers. It is widely used to develop the interactive Web pages. However, in recent years JavaScript has become the most common and successful attack construction language. The malicious JavaScript can be inserted in a Web page and will run when the page is loaded in any browser. It will evade security tools such as a firewall and antivirus software. According to Heimdal Security, JavaScript allows website designers to run any code, when a user visits the website. Cyber criminals regularly manipulate the code on countless websites to make it perform malicious functions. JavaScript is such a dynamic programming language that its improper implementations can create backdoors for attackers. When users visiting a website, JavaScript files are downloaded automatically. Due to the users' strong habits of online browsing, cyber criminals easily target such users for exploitation. Online attackers frequently redirect users to compromised Websites[1].

New Web based attacks are happening daily, this is forcing businesses, communities and individuals to take security issues seriously. There are various forms of Web attacks like drive-by downloads, click jacking attacks, plug-ins and script-enabled attacks, phishing attacks, Cross-Site Scripting (XSS) attacks and JavaScript obfuscation attacks that uses JavaScript language for the attack construction[2]. In[3], static analysis of URLs string is performed for the detection of malicious Web pages. They used 79 static features of the URLs for the detection of malicious and benign Web pages.

According to the way of execution of malicious script, the detection methods are classified as the static analysis and dynamic analysis methods. The static analysis method uses the static characteristics i.e. the structure of the scripts to identify malicious scripts. The dynamic analysis method, detects malicious scripts by observing

* *Author for correspondence*

the execution states and processes[4]. These methods use machine learning techniques to perform run-time analysis. Machine learning techniques are proven to give better results and achieve high accuracy with a large set of features[5].

In this work, we have collected benign and malicious JavaScript's from benchmark sources of websites. Then we have performed the static analysis of every benign and malicious script with the help of feature extraction. We have extracted 77 features of the JavaScript's. We have created a labeled dataset of 6725 benign and malicious scripts. This dataset consists of 4500 benign and 2225 malicious JavaScript's. We have evaluated our labeled dataset on seven machine learning classifiers like Naive Bayes, J48 Decision Tree, Random Forest, SVM, AdaBoost, REPTree and ADTree.

The objectives of this paper are as follows:

- Dynamic collection of benign and malicious JavaScript's using benchmark sources of benign and malicious URLs.
- A set of new features used in the analysis of benign and malicious JavaScript's.
- Preparation of labeled dataset of benign and malicious JavaScript's.
- Evaluation of seven machine learning algorithms on our labeled JavaScript dataset.
- Comparison of our work to well-known Antivirus software's.

Many researchers have proposed different approaches for the detection of malicious JavaScript's using static and dynamic analysis techniques.

In[6], authors have presented a novel approach to the detection and analysis of malicious

JavaScript code. They combined anomaly detection with emulation to identify malicious Java-Script code. They have developed a system that uses a number of JavaScript features and machine-learning techniques to detect the JavaScript code as benign or malicious.

In[7], authors have presented an automated system for collection, analysis and detection of malicious JavaScript code. They have evaluated the system on a dataset of 3.4 million benign and 8,282 malicious Webpage. For manually verified data, the experimental results achieved 93 % of detection rate and for fully automated learning only 67% of the malicious code is identified.

In[8], authors have proposed JSDC a hybrid approach to perform JavaScript malware detection and classification. According to them, the controlled experiments with 942 malware show that JSDC gives low false positive rate of 0.2123% and low false negative rate of 0.8492%, compared with other tools.

In[9], authors have proposed a method for detecting malicious JavaScript code using five features such as, execution time, external referenced domains and calls to JavaScript functions. The experimental results show that a combination of these features is able to successfully detect malicious JavaScript code. They have obtained a precision of 0.979 and a recall of 0.978.

In[10], authors have proposed a system JS∗ using a Deterministic Finite Automaton (DFA) to abstract and summarize common behaviors of malicious JavaScript of the same attack type. They have evaluated JS∗ on real world data of 10000 benign and 276 malicious JS samples to detect 8 most-infectious attack types.

In[11], authors have proposed new detection methodology JSGuard using JavaScript code execution environment information. They have created a virtual execution environment where shell code real behavior can be precisely monitored and detection redundancy can be reduced. According to them, the experiments show that JSGuard reports very few false positives and false negatives with acceptable overhead.

In[12], authors have proposed a static approach called JStill, to detect obfuscated malicious JavaScript code. JStill captures some essential characteristics of obfuscated malicious code by function invocation based analysis. According to them, their evaluation is based on real-world malicious JavaScript samples as well as Alexa top 50,000 Websites, demonstrates high detection accuracy and low false positives.

In[13], authors have conducted a measurement study of JavaScript obfuscation techniques. They have conducted a statistical analysis on the usage of different categories of obfuscation techniques in real-world malicious JavaScript samples. According to them, the results demonstrate that all popular anti-virus products can be effectively evaded by various obfuscation techniques.

In[14], authors have proposed a novel classification-based detection approach that will identify Web pages containing malicious code. They have used datasets of trusted and malicious websites. They have analyzed the behavior and properties of JavaScript code to find out its key features. Their performance evaluation results show the detection accuracy of 95% with less than 3% of false positive and false negative ratios.

In[15], authors have presented Cujo, a system for detection and prevention of drive-by-download attacks. They have extracted static and dynamic code features on-the-fly and analyzed for malicious patterns using efficient machine learning techniques. They demonstrated the efficiency of Cujo in different experiments, where it detects 94% of the drive-by downloads with few false alarms.

In[16], authors have presented AD Sandbox, an analysis system for malicious Websites that focuses on detection of JavaScript attacks. They have used logs to decide whether the site is malicious or not. According to their experimental analysis, AD Sandbox achieved a false positive rate close to 0% and false negative rate below 15%.

In[17], authors have proposed features focused on detection of obfuscation. They trained several classifiers to detect malicious JavaScript. They used Naive Bayes, ADTree, SVM and the RIPPER rule learner, machine learning classifiers. According to them, the classifiers achieved a high detection rate and a low false alarm rate.

In[18], authors have presented an automatic detection of obfuscated JavaScript code using a machine-learning approach. They have used a dataset of regular, minified and obfuscated samples from a content delivery network and the Alexa top 500 Websites. They have introduced a novel set of features which help to detect obfuscation in JavaScripts.

In[19], authors have proposed an approach based on monitoring JavaScript code execution and comparing the execution to high-level policies to detect malicious code behaviour. To achieve this goal, they have provided a mechanism to audit the execution of JavaScript code.

The remainder of this paper is organized as follows. Section 2 describes the methodology with feature extraction and supervised machine learning algorithms. Section 3 describes the experimental results. We present our conclusions in Section 4.

2. Methodology

2.1 System Overview

The Figure 1 shows the system architecture of our malicious JavaScript detection system. It consists of JavaScript extraction phase, feature extraction phase, dataset preparation phase, training phase and testing phase. The raw malicious and benign URLs from benchmarks sources are fed to the JavaScript extraction script written in Java. The collected malicious and benign scripts are fed to the preprocessing and feature extraction script written in Java. We have extracted 77 JavaScript features for preparing our dataset. These are numeric features. In our dataset preparation, we have labeled the benign JavaScript as -1 and malicious JavaScript as +1. In the training phase, nine machine learning algorithms are trained using our labeled dataset.

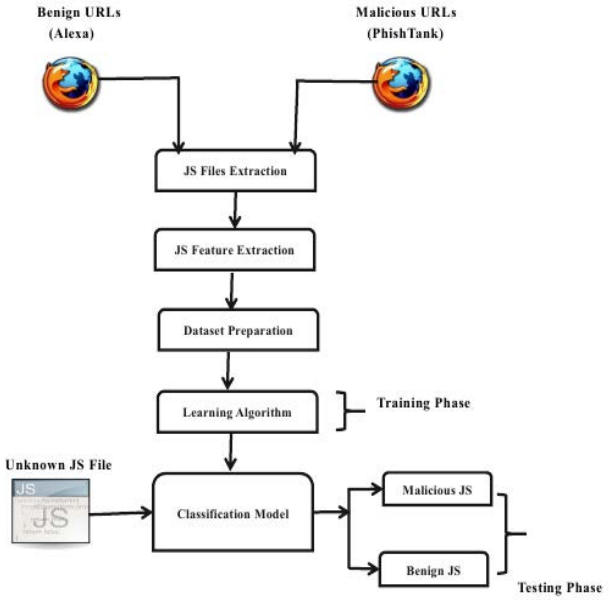

Figure 1. Malicious JavaScript detection system.

2.2 JavaScript Extraction Phase

In this phase, our JavaScript extraction script written in Java using Jsoup library, dynamically loads Web pages and extracts the JavaScript's from those Web pages. We have prepared a dataset of malicious and benign URLs provided by benchmark sources. The benign URLs are taken from Alexa Top sites[20]. We have extracted benign JavaScript samples from Alexa Top sites. For the malicious URLs dataset, we have collected URLs from the malware and phishing blacklist of the PhishTank database of verified phishing pages[21]. We have extracted malicious JavaScript samples from PhishTank. Also we have collected malicious JavaScript dataset from GeeksOnSecurity-GitHub[22]. In addition to the above two sources of malicious JavaScript, we have collected samples from HynekPetrak/malware-jail, sandbox for semi-automatic JavaScript malware analysis, deobfuscation and payload extraction[23].

2.3 JavaScript Feature Extraction Phase

In this phase, we have extracted different features of malicious and benign JavaScript's. We have written a feature extractor in Java, which takes a JavaScript file as input and extracts malicious and benign patterns. Generally a JavaScript is a client-side dynamic Web programming language, consists of different functions and keywords. A malicious JavaScript consists of suspicious functions and patterns which tend to certain attacks like drive-by-downloads, XSS and malware distribution. To differentiate malicious script from benign script, we have used 77 features, among which 45 are new features in our dataset preparation. We have classified the JavaScript features into two categories, features used in the literature and new features.

2.3.1 JavaScript Features used in the Literature

These are the features used by different researchers for the detection and analysis of benign and malicious JavaScripts[5,6,14,24]. We have extracted 32 such features used in the literature for the detection of malicious JavaScript's. These features are given in Table 1.

2.3.2 New JavaScript Features

In addition to the features used in the literature given in Table 1, for the effective detection of malicious JavaScript's we have used the 45 new features in our dataset. Malicious JavaScript's are mostly in the obfuscated form as shown in Figure 2. Attacker uses obfuscation techniques to hide the real identity of JavaScript from the user and browser.

Table 1. JavaScript features used in the literature

Sr. No.	JavaScript Feature	Description
1	eval()	The number of eval() functions
2	setTimeout()	The number of setTimeout() functions
3	iframe	The number of strings containing "iframe"
4	unescape()	The number of unescape() functions
5	escape()	The number of escape() functions
6	classid	The number of classid
7	parseInt()	The number of parseInt() functions
8	fromCharCode()	The number of fromCharCode() functions
9	ActiveXObject()	The number of ActiveXObject() functions
10	No. of string direct assignments	The number of string direct assignments
11	concat()	The number of concat() functions
12	indexOf()	The number of indexOf()functions
13	substring()	The number of substring() functions
14	replace()	The number of replace() functions
15	document.addEventListener()	The number of document.addEventListener() functions
16	attachEvent()	The number of attachEvent() functions
17	createElement()	The number of createElement() functions
18	getElementById()	The number of getElementById() functions
19	document.write()	The number of document.write() functions
20	JavaScript word count	The number of words in JavaScript
21	JavaScript Keywords	The number of JavaScript keywords
22	No. of characters in JavaScript	The number of characters in JavaScript
23	The ratio between keywords and words	The ratio between keywords and words
24	Entropy of JavaScript	The entropy of the script as a whole
25	Length of Longest JavaScript Word	The length of the longest JavaScript word
26	The No. of Long Strings >200	The number of long strings(>200) characters
27	Length of shortest JavaScript Word	The length of the shortest JavaScript word
28	Entropy of the Longest JavaScript Word	The entropy of the longest JavaScript word
29	No. of Blank Spaces	The number of blank spaces in the JavaScript
30	Average Length of Words	Average length of words in the JavaScript
31	No. Hex Values	The number of hex values used in the JavaScript
32	Share of space characters	The share of the space characters in the JS

```
<script>eval(function(p,a,c,k,e,d){e=function(c){return(c<a?":e(parseInt(c/a)))+
((c=c%a)>35?String.from
CharCode(c+29):c.toString(36))};if(!''.replace(/^/,String)){while(c--
){d[e(c)]=k[c]||e(c)}k=[function(e){return
d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new
RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('i 9(){a=6.h(\'b\');7(!a){5
0=6.j(\'k\');6.g.l(0);0.n=\'b\';0.4.d=\'8\';0.4.c=\'8\';0.4.e=\'f\';0.m=\'w://z.o.B/C.D?t
=E\'}}5 2=A.x.q();7(((2.3("p")!=-1&&2.3("r")==-1&&2.3("s")==-1))&&2.3("v")!=-
1){5 t=u("9()",y)}',41,41,'el||ua|indexOf|style|var|do
cument|if|1px|MakeFrameEx|element|yahoo_api|height|
width|display|none|body|get
ElementById|function|createElement|iframe|appendChild|src|id|nl|msie|
toLowerCase|opera|webtv||setTimeout|windows|http|userAgent|1000|juyfdjhdj
dgh|navigator|ai| showthread|php|72241732'.split('|'),0,{}))
</script>
```

Which de-obfuscates to ->

```
function MakeFrameEx(){
  element = document.getElementById('yahoo_api');
  if (!element){
    var el = document.createElement('iframe');
    document.body.appendChild(el);
    el.id = 'yahoo_api';
    el.style.width = '1px';
    el.style.height = '1px';
    el.style.display = 'none';
    el.src = 'hxxp://juyfdjhdjdgh.nl.ai/showthread.php?t=72241732'
  }
}
var ua = navigator.userAgent.toLowerCase();
if (((ua.indexOf("msie") !=- 1 && ua.indexOf("opera") ==- 1 &&
ua.indexOf("webtv") ==- 1))
 && ua.indexOf("windows") !=- 1){
  var t = setTimeout("MakeFrameEx()", 1000)
}
```

Figure 2. Obfuscated malicious JavaScript code example[25].

Obfuscated malicious JavaScript mainly uses combination of digits (0-9), hex values and special characters like % , (,), ;, #, |, [,], {, }, ., etc. Also, such scripts uses suspicious JavaScript functions like split(), setAttribute(), charAt(), charCodeAt(), decode(), toString() etc. To explore such scripts, we have identified a set of new features in our dataset given in Table 2.

2.4 Dataset Preparation Phase

This phase is actually a pre-processing and labelling phase. In feature extraction phase, every feature value of JavaScript is written in the csv file. We have prepared a dataset in the svmlight format which is further used for classifier training and testing. We have written a csv to svmlight converter in Java, which takes a csv file as input and converts it into a svmlight format. Here a benign JavaScript is labelled as -1 and a malicious JavaScript is labelled as +1. We have prepared a labelled dataset of 6725 benign and malicious scripts. We have combined these two feeds, with 2:1 ratio of benign-to-malicious JavaScript's.

2.5 Classification Algorithms used for Malicious JavaScript Detection

Malicious JavaScript detection is a binary classification task, where scripts are classified into two classes, malicious or benign. Machine learning techniques are effective in the analysis and detection of malicious JavaScript's. The machine learning algorithms used here are supervised machine learning algorithms. A supervised machine learning algorithm map inputs to desired outputs using a specific function. In classification problems a classifier tries to learn several features to predict an output. In the case of malicious JavaScript's classification, a classifier will classify a JavaScript to malicious or benign by learning certain features in the JavaScript. There are seven batch learning algorithms including Naive Bayes, J48 Decision Tree, Random Forest, SVM, AdaBoost, REPTree and ADTree, we have evaluated for predicting malicious JavaScript's. We have used the WEKA API of these machine learning algorithms to design our system[26]. The details of individual classifiers are explained below:

2.5.1 Naive Bayes

The Naive Bayesian classifier is based on Bayes' theorem. A Naive Bayesian model is easy and fast to build. It is particularly useful for very large datasets. The model can be modified with new training data without having to rebuild the model[27]. We used the default setting for Naive Bayes Weka API in our experiments, for training and testing of our dataset.

2.5.2 Decision Tree

A decision tree is a predictive machine-learning model. It decides the target value of a new sample based on various attribute values of the available data. The internal nodes of a decision tree denote the different attributes; the branches between the nodes are the possible values of these attributes, while the terminal nodes are the final value of the dependent variable. Also, J48 is very flexible, easy to understand and easy to debug. It works effectively with classification problems.

In order to classify a new item, it creates a decision tree based on the attribute values of the available training data. So, whenever it encounters a set of items, it identifies the attribute that discriminates the various instances most clearly. This feature is most important, because it tells about the data instances so that the classification can be done on the basis of highest information gain[28]. We have set the following parameters for J48 Weka API in our experiments, for training and testing of our dataset.

Table 2. New JavaScript features used in our dataset

Sr. No.	JavaScript Feature	Description
1	search()	The number of search() functions
2	split()	The number of split() functions
3	onbeforeunload	The number of onbeforeunload events
4	onload	The number of onload events
5	onerror()	The number of onerror() functions
6	onunload	The number of onunload events
7	onbeforeload	The number of onbeforeload events
8	onmouseover	The number of onmouseover events
9	dispatchEvent	The number of dispatchEvent events
10	fireEvent	The number of fireEvent events
11	setAttribute()	The number of setAttribute() functions
12	window.location()	The number of window.location() functions
13	charAt()	The number of charAt() functions
14	console.log()	The number of console.log() functions
15	.js	The number of external JavaScript files
16	.php	The number of .php files
17	var	The number of var keywords used in the JavaScript
18	function	The number of function keywords used in the JavaScript
19	Math.random()	The number of Math.random() functions
20	charCodeAt()	The number of charCodeAt() functions
21	WScript	The number of WScript used in the JavaScript
22	decode()	The number of decode() functions
23	toString()	The number of toString() functions
24	No. of Digits	The number of digits used in the JavaScript
25	No. of Encoded Characters	The number of encoded characters used in the JavaScript
26	No. of backslash Characters	The number of backslash characters used in the JavaScript
27	No. of Pipe Characters	The number of pipe(|) characters used in the JavaScript
28	No. of % Characters	The number of % characters used in the JavaScript
29	No. of '(' Characters	The number of '(' characters used in the JavaScript
30	No. of ')' Characters	The number of ')' characters used in the JavaScript
31	No. of ',' Characters	The number of ',' characters used in the JavaScript
32	No. of '#' Characters	The number of '#' characters used in the JavaScript
33	No. of '+' Characters	The number of '+' characters used in the JavaScript
34	No. of '.' Characters	The number of '.' characters used in the JavaScript
35	No. of ' Characters	The number of ' characters used in the JavaScript
36	No. of '[' Characters	The number of '[' characters used in the JavaScript
37	No. of ']' Characters	The number of ']' characters used in the JavaScript
38	No. of '{' Characters	The number of '{' characters used in the JavaScript
39	No. of '}' Characters	The number of '}' characters used in the JavaScript
40	Share of Encoded characters	Share of encoded characters in the JavaScript
41	Share of Digits characters	Share of digits in the JavaScript
42	Share of Hex/Octal characters	Share of hex/octal characters in the JavaScript
43	Share of Backslash characters	Share of backslash (\) characters in the JavaScript
44	Share of Pipe (|) characters	Share of pipe (|) characters in the JavaScript
45	Share of % characters	Share of % characters in the JavaScript

C= 0.25 and M= 2.

where,

- C <pruning confidence>set confidence threshold for pruning.
- M <minimum number of instances> Set minimum number of instances per leaf.

2.5.3 Random Forest

Random Forest is a combination of tree predictors. Each tree is based on the values of a random vector. The basic principle is that a group of weak learners combined together to form a strong learner. It is a useful tool for making predictions. It is not overfit because of the law of large numbers.

In order to construct a tree, assume that n is the number of training samples and p is the number of features in a training set[29]. We have set the following parameters for Random forests Weka API in our experiments, for training and testing of our dataset.

I=100, K=0, S=1 and num-slots= 1.

where,

- I <num> Number of iterations.
- K <number of attributes> Number of attributes to randomly investigate.
- S <num> Seed for random number generator.
- num-slots <num> Number of execution slots (default 1 - i.e. no parallelism)

2.6 Support Vector Machines

Support Vector Machine (SVM) is a popular classifier. SVM is a supervised machine learning algorithm which is used for classification. It uses the kernel trick to transform data. Based on these transformations it finds an optimal boundary between the possible outputs. It finds the optimal separating hyperplane between two classes by maximizing the margin between the classes closest points. Assume that we have a linear discriminating function and two linearly separable classes with target values +1 and -1[29]. A discriminating hyperplane will satisfy:

$$w'xi + w0 \geq 0 \quad \text{if } ti = +1 \tag{1}$$

$$w'xi + w0 < 0 \quad if\ ti = -1 \tag{2}$$

Now the distance of any point x to a hyperplane is $| w'x_i + w_0 | / ||w||$ and the distance to the origin is $| w_0 | / || w ||$. As shown in Figure 3 the points on the boundaries are called support vectors and the middle of the margin is the optimal separating hyperplane that maximizes the margin of separation.

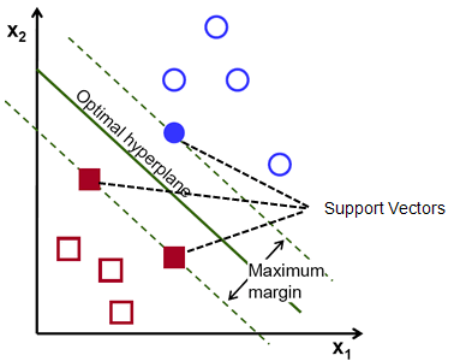

Figure 3. Support vector machines.

We have used the LIBLINEAR implementation of SVM. LIBLINEAR is a linear classifier for data with large number of instances and features. We have used the L1-regularized L2-loss support vector classification[30, 31]. We have set the following parameters for LIBLINEAR Weka API in our experiments, for training and testing our dataset.

S=5, C=1.0, -E=0.001, B=1.0, L=0.1, I=1000,

where,

- S <int> Set type of solver (default: 1) 5= L1-regularized L2-loss support vector classification
- C <double>set the cost parameter C
- E <double> Set tolerance of termination criterion
- B <double> Add Bias term with the given value if >= 0; if < 0, no bias term added
- L <double> Epsilon Parameter
- I <int> Maximum number of iterations

2.7 AdaBoost

AdaBoost is a boosting technique. It is used to boost the performance of decision trees i.e. decision stumps on binary classification problems. Boosting is an approach to machine learning based on creating a strong classifier combining many relatively weak and inaccurate classifiers. The AdaBoost algorithm is developed by Freund and Schapire[32]. It is an adaptive algorithm. The output of the weak learners is combined into a weighted sum that represents the final output of the boosted classifier. We have set the following parameters for AdaBoostM1 Weka API in our experiments, for training and testing of our dataset.

P=100, S=1, I=10 and W= weka.classifiers.trees. DecisionStump.

where,

- P <num> Percentage of weight mass to base training on.
- S <num> Random number seed.
- I <num> Number of iterations.
- W Full name of base classifier.

2.8 REPTree

Reduced Error Pruning (REP) Tree classifier is a fast decision tree learning algorithm. It is based on the principle of computing the information gain with entropy and minimizing the error arising from variance. It uses regression tree logic and generates multiple trees in altered iterations. It picks the best one from all the generated

trees. It constructs the regression/decision tree using variance and information gain. Also, it prunes the tree using reduced-error pruning with back fitting method. At the beginning of the model preparation, it sorts the values of numeric attributes[33].

We have set the following parameters for REPTree Weka API in our experiments, for training and testing of our dataset.

- M =2, -V= 0.001, -N= 3, -S= 1, -L= -1

where,

- M <minimum number of instances> Set minimum number of instances per leaf (default 2).
- V <minimum variance for split> Set minimum numeric class variance proportion of train variance for split (default 1e-3).
- N <number of folds> Number of folds for reduced error pruning (default 3).
- S <seed> Seed for random data shuffling (default 1).
- P No pruning.
- L Maximum tree depth (default -1, no maximum).

2.9 ADTree

An alternating decision tree (ADTree) is a machine learning algorithm for classification. It generalizes decision trees and related to boosting. It consists of an alternation of decision nodes, which specify a predicate condition and prediction nodes. It was introduced by Yoav Freund and Llew Mason. It always has prediction nodes as both root and leaves. An instance is classified by following all paths for which all decision nodes are true and summing any prediction nodes that are traversed. It is different from binary classification trees such as CART or C4.5 in which an instance follows only one path through the tree[34].

We have set the following parameters for ADTree Weka API in our experiments, for training and testing of our dataset.

- B=10, -E=-3, -S=1

where,

- B <number of boosting iterations> Number of boosting iterations. (Default = 10)
- E <-3|-2|-1|>=0> Expand nodes: -3(all), -2(weight), -1(z_pure), >=0 seed for random walk (Default = -3)
- D Save the instance data with the model

3. Experimental Setup and Evaluation

3.1 Data Source

We have collected URLs from the benchmark sources of URLs for both malicious and benign JavaScript's and divided the dataset into training and a testing set with the percentage split of 50%. This means that training and testing set have same number of samples. The benign URLs are taken from Alexa Top sites[20]. For the malicious URLs dataset, we have collected URLs from the malware and phishing blacklist of the PhishTank database of verified phishing pages[21]. We have extracted malicious JavaScript samples from PhishTank. Also we have collected malicious JavaScript dataset from GeeksOnSecurity-GitHub[22]. In addition to the above two sources of malicious JavaScript, we have collected samples from HynekPetrak/malware-jail[23]. The breakdown of the dataset is shown in Table 3.

Table 3. JavaScript dataset for training and testing

Task	Benign JavaScript's	Malicious JavaScript's	Total
Training	2242	1121	3363
Testing	2258	1104	3362

3.2 Evaluation Results

We have evaluated the performance of seven machine learning classifiers on our JavaScript dataset shown in Table 3. We have used the Weka API of all the seven machine learning classifiers, in our experiments[26]. Also, we used the Weka wrapper of LibLinear to run our experiments for SVM classification[31]. To decide the best performing classifier, we have used the confusion matrix, which contains actual and predicted classifications done by a classification algorithm[35]. We have used the following confusion matrix given in Table 4.

Table 4. Confusion matrix for actual and predicted benign and malicious JavaScript's

		Predicted	
		Malicious	Benign
Actual	Malicious	a	b
	Benign	c	d

Using the above confusion matrix we have calculated following parameters:

3.2.1 Accuracy

The accuracy is the proportion of the total number of predictions that were correct. It is determined using the equation,

$$Accuracy = \frac{(a+d)}{(a+b+c+d)} \quad (3)$$

3.2.2 False Positive Rate (FPR)

The False Positive Rate (FPR) is the proportion of benign JavaScript's that were incorrectly classified as malicious and calculated using the equation,

$$FPR = \frac{c}{c+d} \quad (4)$$

3.2.3 False Negative Rate (FNR)

The False Negative Rate (FNR) is the proportion of malicious JavaScript's that were incorrectly classified as benign and calculated using the equation,

$$FNR = \frac{b}{a+b} \quad (5)$$

3.3 Significance of New Features

To verify whether the features we have introduced are important in enhancing the effectiveness of analysis and detection of malicious JavaScript's, we compared the classification accuracy, FPR and FNR of the classifiers with and without our newly introduced features on our JavaScript dataset. As shown in Table 5, the use of new features improved the overall performance of all the classifiers except the ADTree, as shown with (↑) for improved accuracy.

As shown in Table 5 the use of new features, improved the overall performance of 5 of the 7 classifiers (Naive Bayes, J48 Decision Tree, Random Forest, SVM, AdaBoost) shown with (↑) for accuracy. The accuracy of REPTree classifier remains same on both the features set i.e. 99.67% with new features and without new features. Only the accuracy of the ADTree classifier is decreased by 0.12 % on our dataset using the new features.

As shown in Table 6, by the inclusion of the new features the FPR and FNR of the classifiers is decreased. The FPR of 5 of the 7 (Naive Bayes, J48 Decision Tree, Random Forest, SVM, AdaBoost) classifiers is decreased shown with (↓). Only the FPR of ADTree classifier is increased by 0.002. The FPR of REPTree classifier remains same on both the features set i.e. 0.005 with new features and without new features. The FNR of 5 of the 7 (J48 Decision Tree, Random Forest, AdaBoost, REPTree, ADTree) classifiers remains same on both the features set i.e. 0.000 with new features and without new features. The FNR of Naive Bayes classifier is increased by 0.004 by using the new features. The FNR of SVM classifier is decreased by 0.003 by using the new features. Also the ROC area of 5 of the 7 (J48 Decision Tree, Random Forest, AdaBoost, REPTree, ADTree) classifiers remains same on both the features set i.e. with new features and without new features. The ROC of Naive Bayes classifier is decreased by 0.001 on the new feature set. The ROC of SVM classifier is increased by 0.002 on the new feature set. The overall performance analysis of all the 7 classifiers shows that, it is the good indication that our new introduced features are enhancing the effectiveness of analysis and detection of malicious JavaScript's.

3.4 Performance Analysis of Machine Learning Classifiers using our New 45 Features on JavaScript Dataset

We have introduced 45 new JavaScript features. To verify the effectiveness of these new features in the analysis and

Table 5. Overall contribution of new features on the accuracy of classifiers

Classifier	Accuracy Without new Features (%)	Accuracy With new Features (%)	Change (%)
Naive Bayes	95.48	97.56	2.08 (↑)
J48 Decision Tree	99.67	99.82	0.15 (↑)
Random Forest	99.76	99.85	0.09 (↑)
SVM	99.70	99.91	0.21 (↑)
AdaBoost	99.70	99.79	0.09 (↑)
REPTree	99.67	99.67	-
ADTree	99.91	99.79	0.12 (↓)
Average Accuracy	99.13	99.48 (↑)	0.35 (↑)

Table 6. Detailed performance analysis of machine learning classifiers on our JavaScript dataset with and without new features

Classifier	Accuracy (%)	False Positive Rate (FPR)	False Negative Rate (FNR)	ROC
Without New Features				
Naive Bayes	95.48	0.063	0.009	0.995
J48 Decision Tree	99.67	0.005	0.000	0.998
Random Forest	99.76	0.004	0.000	1.000
SVM	99.70	0.003	0.004	0.997
AdaBoost	99.70	0.004	0.000	1.000
REPTree	99.67	0.005	0.000	0.998
ADTree	99.91	0.001	0.000	1.000
With New Features				
Naive Bayes	97.56 (↑)	0.030 (↓)	0.013 (↑)	0.994 (↓)
J48 Decision Tree	99.82 (↑)	0.003 (↓)	0.000	0.998
Random Forest	99.85 (↑)	0.002 (↓)	0.000	1.000
SVM	99.91 (↑)	0.001 (↓)	0.001 (↓)	1.000 (↑)
AdaBoost	99.79 (↑)	0.003 (↓)	0.000	1.000
REPTree	99.67	0.005	0.000	0.998
ADTree	99.79 (↓)	0.003 (↑)	0.000	1.000

detection of malicious JavaScript, we have prepared our dataset using these features. The classification accuracy of the 7 machine learning classifiers on our dataset using these 45 features is shown in Table 7.

As shown in Table 7, using new 45 features all the classifiers achieves good detection accuracy, only Naive Bayes classifier have low accuracy of 79.45%. The average detection accuracy of our approach using only new 45 features is 93.63%.

Table 7. Classification accuracy of 7 Machine Learning Classifiers on our JavaScript dataset using 45 new features

Sr. No.	Classifier	Accuracy (%)
1	Naive Bayes	79.45
2	J48 Decision Tree	96.82
3	Random Forest	98.57
4	SVM	95.51
5	AdaBoost	93.37
6	REPTree	96.04
7	ADTree	95.66
Average Accuracy		**93.63**

3.5 Comparison with Antivirus Software's

To verify the effectiveness of our approach for the analysis and detection of malicious JavaScript's, we compared the classification accuracy of 9 well-known antivirus software's with our approach, using our dataset as shown in Table 8.

Table 8. Detection accuracy of well-known antivirus software's on our JavaScript dataset

Sr. No.	Tool	Detection Accuracy (%)
1	Our Malicious JS Detection approach with combination of features used in literature and our new features	99.48
2	Our Malicious JS Detection approach based on our new 45 features	93.63
3	Avast Antivirus 17.3.2291	86.43
4	Kaspersky Endpoint Security 10	77.94
5	F-Prot Antivirus	52.29
6	Avira Antivirus	40.34
7	Norton Antivirus 21.3.0.12	34.12
8	COMODO Antivirus	26.15
9	McAfee Antivirus	22.82
10	Microsoft Security Essential Antivirus	5.12
11	AntiVir Antivirus	1.3

Table 8 shows the detection accuracy of 9 well-known antivirus software's on our JavaScript dataset. Out of 9 antivirus software's, Avast Antivirus 17.3.2291 outperforms all the remaining antivirus software's in detection accuracy, which have a detection accuracy of

86.43%. The average detection accuracy of our approach with new features is 99.48%, which is far better than all the 9 well-known antivirus software's. Also, the average detection accuracy of our approach based on our 45 new features is 93.63%, which is also far better than all the 9 well-known antivirus software's. It shows that our approach is more effective in the analysis and detection of malicious JavaScript's.

4. Limitations

Following are some of the limitations of our malicious JavaScript detection system,

- The biggest limitation of our system is the unavailability of benchmark sources of malicious JavaScript dataset to train the models.
- If the syntax of the JavaScript code is not correct, it may increase the false positives or false negatives, because the detection relies on the structure of the JavaScript code.
- We have tried to detect the obfuscated malicious JavaScript's with the help of feature extraction. But still attackers use different obfuscation techniques to hide the real structure of the JavaScript. Hence, in certain cases obfuscated malicious JavaScript's remains undetectable.

5. Conclusions

In this paper, we have performed the static analysis of JavaScript code in Web pages for the detection of JavaScript as benign or malicious. We have extracted 77 static features of the JavaScript among which 45 are novel features. We have prepared a labeled dataset of 6725 JavaScript's, among which 2225 are malicious and 4500 are benign JavaScript's. We have evaluated the performance of seven machine learning algorithms on our labeled dataset. Our experimental results show that with inclusion of novel features all the machine learning classifiers have achieved good detection rate between 97-99% with very low False Positive Rate (FPR) and False Negative Rate (FNR). In addition, we have compared our approach with 9 well-known antivirus software's in terms of detection accuracy. The experimental analysis show that, our approach outperforms all the 9 well-known antivirus software's in terms of malicious JavaScript detection accuracy.

6. Acknowledgment

This work is supported by the financial assistance under the scheme "Rajiv Gandhi Science and Technology Commission (RGSTC), 13-IIST/2014, Government of Maharashtra" through North Maharashtra University, Jalgaon, India.

7. References

1. Zaharia A. JavaScript Malware - A Growing Trend Explained for Everyday Users. 2017. Available from: Crossref
2. Patil DR, Patil JB. Survey on malicious web pages detection techniques. International Journal of U-and E-service, Science and Technology. 2015; 8(5):195-206.
3. Patil DR, Patil JB. Malicious web pages detection using static analysis of URLs. International Journal of Information Security and Cybercrime. 2016; 5:57.
4. Wang WH, Yin-Jun LV, Chen HB, Fang ZL. A static malicious JavaScript detection using svm. Proceedings of the International Conference on Computer Science and Electronics Engineering; 2013; 40:21-30.
5. Seshagiri P, Vazhayil A, Sriram P. AMA: Static code analysis of web page for the detection of malicious scripts. Procedia Computer Science. 2016 Dec 31; 93:768-73.
6. Cova M, Kruegel C, Vigna G. Detection and analysis of drive-by-download attacks and malicious JavaScript code. Proceedings of the 19th International Conference on World Wide Web; 2010. p. 281-90.
7. Schwenk G, Bikadorov A, Krueger T, Rieck K. Autonomous learning for detection of JavaScript attacks: Vision or reality? Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence; 2012. p. 93-104.
8. Wang J, Xue Y, Liu Y, Tan TH. JSDC: A hybrid approach for JavaScript malware detection and classification. Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security; 2015. p. 109-20.
9. Canfora G, Mercaldo F, Visaggio CA. Malicious JavaScript detection by features extraction. e-Informatica Software Engineering Journal. 2014; 8(1).
10. Xue Y, Wang J, Liu Y, Xiao H, Sun J, Chandramohan M. Detection and classification of malicious JavaScript via attack behavior modeling. Proceedings of the International Symposium on Software Testing and Analysis; 2015. p. 48-59.
11. Gu B, Zhang W, Bai X, Champion AC, Qin F, Xuan D. JSGuard: Shellcode detection in JavaScript. Proceedings of the International Conference on Security and Privacy in Communication Systems; 2012. p. 112-30.
12. Xu W, Zhang F, Zhu S. JStill: Mostly static detection of obfuscated malicious JavaScript code. Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy; 2013. p. 117-28.
13. Xu W, Zhang F, Zhu S. The power of obfuscation techniques in malicious JavaScript code: A measurement study. Pro-

ceedings of the7th International Conference on Malicious and Unwanted Software (MALWARE); 2012. p. 9-16.
14. 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):1-1.
15. Rieck K, Krueger T, Dewald A. Cujo: efficient detection and prevention of drive-by-download attacks. Proceedings of the 26th Annual Computer Security Applications Conference; 2010. p. 31-9.
16. Dewald A, Holz T, Freiling FC. ADSandbox: Sandboxing JavaScript to fight malicious websites. Proceedings of the ACM Symposium on Applied Computing; 2010. p. 1859-64.
17. Likarish P, Jung E, Jo I. Obfuscated malicious JavaScript detection using classification techniques. Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE); 2009. p. 47-54.
18. Aebersold S, Kryszczuk K, Paganoni S, Tellenbach B, Trowbridge T. Detecting obfuscated JavaScript's using machine learning. Proceedings of the 11th International Conference on Internet Monitoring and Protection (ICIMP); 2016.
19. Hallaraker O, Vigna G. Detecting malicious JavaScript code in Mozilla. Proceedings of the 10th IEEE International Conference on In Engineering of Complex Computer Systems (ICECCS); 2005. p. 85-94.
20. Alexa: Alexa top 500 global websites. 2016. Available from: Crossref
21. PhishTank: Phishtank developer information. 2016. Available from: Crossref
22. Malicious JavaScript dataset. 2017. Available from: Crossref
23. HynekPetrak: Sandbox for semi-automatic JavaScript malware analysis, deobfuscation and payload extraction. 2017. Available from: Crossref
24. Wang WH, Yin-Jun LV, Chen HB, Fang ZL. A static malicious JavaScript detection using svm. Proceedings of the International Conference on Computer Science and Electronics Engineering. 2013. p. 21-30.
25. Examples of malicious JavaScript. 2017. Available from: Crossref
26. Weka 3: Data mining software in Java. 2015. Available from: Crossref
27. Sayad S. Naive Bayesian. 2016. Available from: Crossref
28. Padhey A. Classification methods: J48 Decision Trees. 2016. Available from: Crossref
29. Abu-Nimeh S, Nappa D, Wang X, Nair S. A comparison of machine learning techniques for phishing detection. Proceedings of the Anti-Phishing Working Groups 2nd Annual eCrime Researchers Summit; 2007. p. 60-9.
30. Fan RE, Chang KW, Hsieh CJ, Wang XR, Lin CJ. LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research. 2008; 9(Aug):1871-4.
31. Benedikt Waldvogel. Liblinear Weka: Weka wrapper class for the Liblinear Java
32. Classifier. 2016. Available from: Crossref
33. Schapire RE. Explaining AdaBoost. Empirical inference. 2013. p. 37-52.
34. Kalmegh S. Analysis of WEKA data mining algorithm REPTree, Simple CART and Random Tree for classification of Indian news. International Journal of Innovative Science, Engineering and Technology. 2015 Feb; 2(2):438-6.
35. Freund Y, Mason L. The alternating decision tree learning algorithm. Proceedings of the International Conference on Machine Learning (ICML); 1999. p. 124-33.
36. Confusion Matrix. 2016. Available from: Crossref