Malicious Drive-By-Download Website Classification Using JavaScript Features

by

Sam Wang

B.Sc., University of Victoria, 2014

An Industrial Project Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the Department of Computer Science

Supervisory Committee

Malicious Drive-By-Download Website Classification Using
JavaScript Features

by

Sam Wang

B.Sc., University of Victoria, 2014

Supervisory Committee

Dr. Jianping Pan, Department of Computer Science
Supervisor

Dr. Sudhakar Ganti, Department of Computer Science
Departmental Member

Abstract

Supervisory Committee
Dr. Jianping Pan, Department of Computer Science
Supervisor
Dr. Sudhakar Ganti, Department of Computer Science
Departmental Member

In recent years, Drive-by-download attacks make up over 90% of web-based attacks on web users. Many web users fall victim to this type of attacks due to its simplicity and less complex requirements to be compromised. They simply need to click on a malicious URL while having some browser vulnerabilities for the malicious attackers to compromise their machine and to obtain their sensitive information. To combat these attacks, proactive blacklists are used nowadays for preventing web users from accessing these malicious web pages. This report attempts to supplement the existing proactive blacklisting framework by introducing JavaScript feature vectors for classification. These feature vectors include the functionality of JavaScript in terms of JavaScript bytecode, as well as some string analysis properties for the classification of benign and malicious web pages. A few different classifiers are tested and compared to provide insight on the different JavaScript feature vectors defined.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to acknowledge and thank the AutoBLG team (Network Security Lab) at Waseda University in Japan for providing me with the data set for this project and also for the technical support provided in processing the data. I would also like to thank the University of Victoria Parallel, Networking and Distributed Applications (PANDA) lab members for their help and support during this project and the Natural Sciences and Engineering Research Council of Canada (NSERC) for the research assistantship and equipments.

1. Background

1.1 Drive-by-Download Attacks

Drive-by-download is a type of web-based attack that involves attacking a user when the user accesses a malicious, or compromised website, via a malicious URL (linked by emails, or URL hot-linked by attackers), or during the normal exploration of the Internet. The condition of the attack is that the user fits some vulnerability profiles which the malicious site is looking for when the user accesses the site. As demonstrated in Fig. 1, when first clicking on the landing page URL, the user is redirected to an exploit URL in the background. The exploit URL looks for some browser vulnerabilities of the user, and if such vulnerabilities are found, the user is then redirected to a malware download URL where malwares are downloaded onto the user's machine without notice. These downloaded files can then steal sensitive information or computer resources from the user. In recent years, drive-by-download attacks make up over 90% of web-based attacks [1], and thus this is a hot research topic as well as a priority for researchers and security engineers alike.

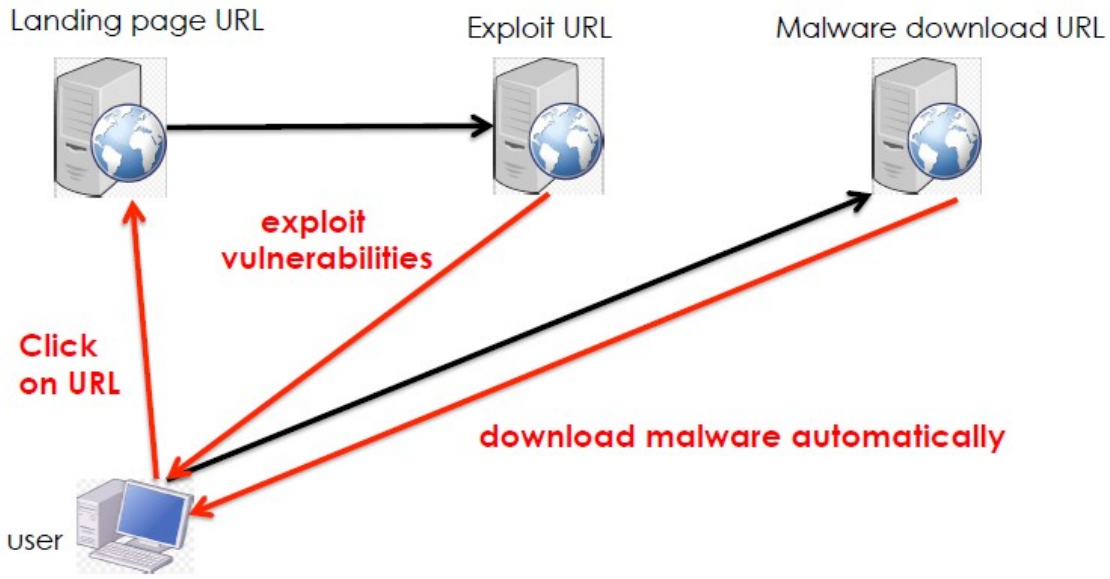


Fig. 1: Drive-by-Download attack model [1].

1.2 Current Approaches

To defend against drive-by-download attacks, two main approaches are used. The first approach is actively identifying the vulnerabilities of browsers and patching them to prevent being exploited, but in many instances, browser vulnerabilities are not identified until attacks occur, and we cannot always rely on end users to be proactive in patching. The second approach is the use of URL blacklists. Blacklists are lists of blocked URLs that ISPs, search engines, or network admins maintain, to prevent the end users from accessing these malicious URLs. Reactive blacklists have the same issue in that they rely on some users to be attacked before the blacklists can register the malicious URL. Recently, much work in the research community has focused on creating proactive blacklists that can register compromised sites and blacklist these sites before the attacks are even active.

2. Related Work

Many different angles have been explored in the detection of new malicious URLs. The International Computer Science Institute group at Berkley proposes a domain name-based approach [2], in which they check for elements such as the age of name server of domains and the relationship between the name server and the domain name. The University of California - San Diego group explores the lexical feature of a URL, and host-based features such as where the hosts are, who owns them and how they are managed [3], and then uses machine learning approaches to classify good and bad URLs. The University of Texas group analyzes both network-layer traffic and application-layer website contents in their work [4]. The Japan Advanced Institute of Science and Technology (JAIST) group creates a prediction model for predicting the downloading of malwares during a drive-by-download attack session by leveraging the Common Vulnerabilities and Exposures-ID vulnerability information as well as JavaScript opcode of websites [5]. Prophiler [6] is an all around tool that takes in HTML, JavaScript, URL and Host information to classify unknown websites as malicious or benign. The group of researchers at Yokohama National University in Japan explores ways of identifying JavaScript obfuscation [7] (more on obfuscation later in the report). Automatic Blacklist Generator (AutoBLG) is a lightweight proactive malicious URL blacklist framework [1]. This framework will be further explored in the next section.

3. AutoBLG

3.1 AutoBLG Overview

AutoBLG is a light-weight static analysis framework for discovery of new malicious URLs. The framework consists of three components: URL Expansion, URL Filtration and URL Verification (Fig. 2). The inputs of AutoBLG are currently known malicious URLs and outputs of the framework are new malicious URLs.

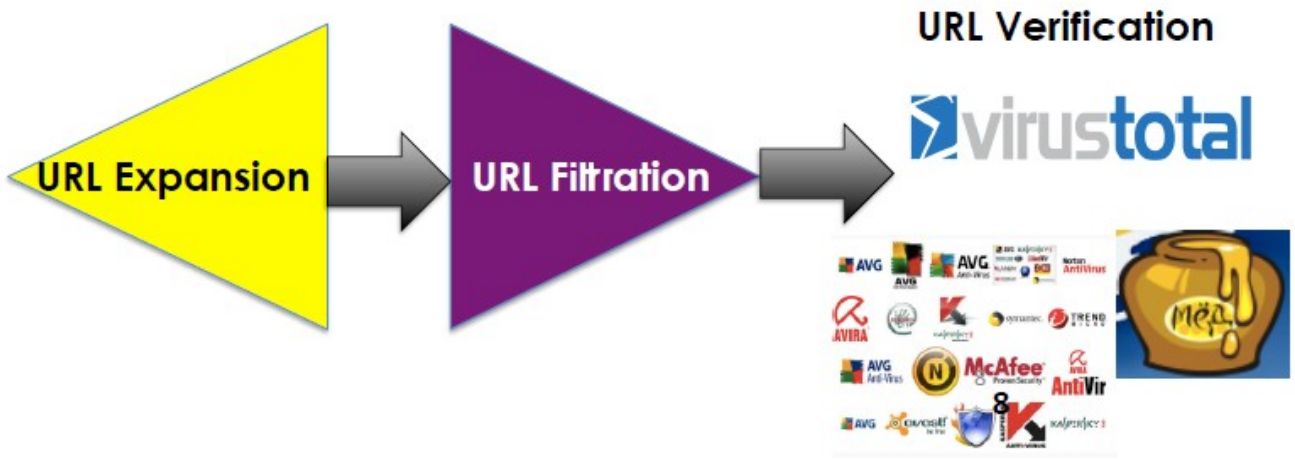


Fig. 2: High-level overview and work flow of AutoBLG [1].

3.2 URL Expansion

The URL expansion step takes in known malicious URLs and obtains the IP addresses associated with the malicious URLs. From the set of IP addresses, a passive DNS database is leveraged to return a set of Fully Qualified Domain Names (FQDN) associated with the IP addresses. This set of FQDN is considered the "neighbourhood" of existing malicious URLs in terms of IP address. The FQDN itself is not sufficient because the attacker likely places malicious web-pages deep in the directory structure of a server. The next step is leveraging search engines and web crawlers to obtain highly ranked and accessible pages inside the directories. The outputs of the URL expansion phase are a large set of URLs associated with the original malicious URLs.

3.3 URL Filtration

Due to the high number of URLs generated from the URL expansion step, a filter is used to trim them down to a much smaller set before processing them via verification techniques. The URL filtration step leverages the Bayesian Set algorithm to classify URLs in terms of maliciousness. The features used for feature extraction and classification are 10 static features from the landing page contents: the number of `iframe` and `frame` tags, the number of hidden elements with a small display area, the number of out-of-place elements, the number of embedded and object tags, the presence of `unescape` behaviour, the number of suspicious words in the script, the number of `setTimeout` functions, and the number of URLs with a different domain. Then based on the classification of the URL maliciousness the top ranked malicious URLs are passed onto the URL verification step. Overall about 1% of the inputs of URL filtration are outputted to the next step.

3.4 URL Verification

The highly malicious URLs are then finally passed onto the URL verification step for malicious verification of the URL. Three tools are leveraged in the URL verification step for this process: the Marionette web client honeypot, anti-virus software, and an online URL verification site VirusTotal. The web client honeypot can trace redirection generated by drive-by-download attacks and identify the malware distribution URL. Anti-virus software analyzes HTML and JavaScript contents statically to check for malicious contents. The VirusTotal website compares the URLs submitted to URL blacklists and cyber-attack detection systems, and then forwards the result of the comparison to users.

4. Analysis of the Current Framework and Plan

The AutoBLG framework does a good job of detecting new potential URLs while still being a light weight framework. However as the authors mentioned, there are a few things that can definitely be improved upon.

- Classification Accuracy – Right now for the 600 URLs inputted to the final verification step, 106 of them are found to be malicious or suspicious. Currently the final verification step is tedious and time consuming even for just 600 URLs, highly classification accuracy is desired for the URL filtration step either by reducing the false positive URLs and reducing the number of non-malicious URLs to be verified, or even better, by decreasing the number of false negatives and including more malicious URLs in the final verification step.
- Cloaked URL detection – Right now the set of URLs to be filtered and finally verified relies on the search engines and web crawlers to find the site in the first place. Some sites employ advanced cloaking techniques to prevent search engines and web crawlers from finding these URLs in the first place, and these cloaked URLs will never be in the set of URLs to be filtered or verified.

Currently, the AutoBLG framework leverages very little JavaScript features for the URL filtration step (only the feature "number of suspicious words in script" is observed, and only at JavaScript scripts directly inside the HTML landing pages). Even in the paper the authors mentioned that they intend to increase the use of JavaScript in the landing page in the future. By increasing the gathering and use of JavaScript, we can improve some of the above mentioned issues. In the URL filtration step, additional JavaScript based features can be added to complement the HTML based features observed right now to improve the classification accuracy. Also by scrutinizing the JavaScript, we can potentially identify hidden URLs that are missed by the web crawlers and pick up some potentially missed malicious URLs.

Unfortunately, many JavaScript codes on the Internet, especially the ones on malicious sites, have their JavaScript obfuscated, so it is hard for us to analyze and tell exactly what it is doing. JavaScript obfuscation is a technique that the JavaScript codes and data fields are scrambled and modified in a way that it is hard for the reader, or program analyzer to understand the codes, while being functionally

equivalent to the JavaScript before the obfuscation. This technique is used on many sites, not only malicious sites, but also some benign sites on the Internet.

Based on the above, my plan was to identify JavaScript classification vectors via static analysis through two different sources. The first source is through JavaScript bytecode features extracted from the JavaScript, which give us an idea of what the JavaScript is doing. The second source is taking advantage of the fact that many malicious sites contain obfuscated JavaScripts, which look different from non-obfuscated JavaScripts and have a different distribution of characters in the string. Leveraging these differences also allows us to extract some feature vectors for classification.

5. Classification via JavaScript Bytecode Features

Work for this part of the research was conducted last semester in the CSC 591 Directed Studies – Web Security course, so I will only briefly explain the thought and methodology conducted in this part of the classification. The final results of this report also include work from this part of the research, as a result this section is included.

Due to the obfuscation of JavaScripts, instead of looking directly at the JavaScript for feature vectors, I was looking to employ the technique similar to what the JAIST group did for predicting malware download [5], in that the interpreted bytecodes of the JavaScript are used instead of the JavaScript itself for feature vectors. Bytecodes has the advantage of bypassing obfuscation, as well as detecting functions and URLs hidden inside the JavaScript by partially interpreting the JavaScript.

The bytecodes used in particular are the SpiderMonkey bytecodes developed by Mozilla, and although the bytecodes is platform specific to Mozilla browser, for our feature extraction of instruction set this is sufficient.

> *"*SpiderMonkey bytecodes are the canonical form of code representation that is used in the JavaScript engine. The JavaScript front end constructs an Abstract Syntax Tree (AST) from the source text, then emits stack-based bytecodes from that AST as a part of the JSScript data structure. Bytecodes can reference atoms and objects (typically by array index) which are also contained in the JSScript data structure." - Mozilla Developer Network

The process to obtain JavaScript bytecodes involve going through the HTML of the landing pages and extracting all JavaScript blocks and downloading all JavaScript files associated. These JavaScript files are then passed onto the SpiderMonkey shell and dissembled into the bytecode forms. The feature vectors used for classification (Fig. 3) are the ratio of each of the bytecode calls to the total number of bytecode function calls.

Site_Name	strictne	string	hole	call	strict-delelem	initelem_array	enditer	length	goto	setname	
geico.com	18.5267999744	485.5299303648	0	524.819523414	0	67.3992205967	14.0548137737	28.1096275474	146.9366894525	0	
elpais.com	3.9890858611	707.9829586252	0	501.348311021	0	11.3290038455	7.8186082877	26.0088398143	180.306680921	0.3191268689	
pornerbros.com	18.4842883549	475.9704251386	0	596.1182994455	0	138.632162662	4.6210720887	23.1053604436	124.7689463956	0	
blogger.com	0	345.6998313659	0	539.629005059	0	0	8.431703204	42.1585160202	160.2023608769	0	
gazetaexpress.com	26.2764960768	478.5396416442	0	661.5967911933	0	25.2517859234	5.1235507671	35.9380489519	151.9498770348	0	
hdzog.com	25.1151109251	690.6655504395	0	671.8292172457	0	14.650481373	2.0929259104	27.2080368355	133.9472582671	0	
zdf.de	42.1348314607	764.0449438202	0	640.4494382022	0	0	8.4269662921	56.1797752809	174.1573033708	0	
instantdownloaderpro.com	31.4960629921	629.9212598425	0	834.6456692913	0	0	0	31.4960629921	188.9763779528	0	
tilestwra.com	39.8168849143	361.5458320026	0	573.7251144469	0.3193867774	34.600234217	10.8591504312	38.8587245821	181.0923027787	0.1064622591	
freepik.com	27.8960702927	356.2476416366	3.9755269258	648.6847070239	0.1347636246	45.8196323648	4.8514904857	34.229960649	155.8541318527	0.1347636246	
echoroukonline.com	32.4488183934	673.7399397989	0	663.208305759	0	23.553857211	4.4830604359	36.576080382	155.5550811576	0	
meituan.com	0	671.031096563	0	507.3649754501	0	122.749590835	16.3666121113	40.9165302782	98.1996726678	0	
wordpress.org	31.4736408258	344.5239611825	0	556.5963505564	0	24.1672599198	12.7393308104	49.0838922402	183.5962381505	0	
huffingtonpost.ca	8.2922177536	645.1345412331	0	470.583357519	0	54.3140262863	3.3168871015	37.7295907791	193.2086736598	0.4146108877	
smallpdf.com	0	821.9178082192	0	582.1917808219	0	0	0	0	205.4794520548	0	
allegro.pl	18.0956089516	415.5118308633	0	591.4286368738	0.2290583412	41.4595597499	8.0170419406	39.8561513618	157.1340220354	26.3417092333	
elmogaz.com	34.3822408252	340.458928171	0	528.1909460099	0	33.6348008072	9.8412935695	41.4829209956	204.1756982336	0.1245733363	
gmw.cn	31.4819554646	339.6467878167	0	506.2707960072	0	20.7320194523	13.5653954441	49.1425646276	183.2608139237	0	
dpstream.net	0	1100.826446281	0	1143.8016528926	0	82.6446280992	0	0	76.0330578512	0	

Fig. 3: Snapshot of JavaScript bytecode ratio distribution for each website.

6. Classification via JavaScript String Properties

6.1 JavaScript Obfuscation

The previous section can be described as classification using the functionality of the JavaScript, whereas this section includes classification using the visual difference of the two, or how the structures of the strings inside the JavaScripts differ. This is taking advantage of the fact that many malicious JavaScripts are obfuscated whereas very little benign JavaScripts are [6].

To better understand how we can leverage string properties to detect obfuscated JavaScripts and malicious web pages, we should first understand the basics of JavaScript obfuscation. Table 1 is a summary of some of the basic techniques used for obfuscation.

Table 1. Basic techniques of JavaScript obfuscation and evasion [8].

Technique	Description
String encoding	Encode literals to generate unreadable versions of them (e.g., “A” character can be presented as “%41” using URL encoding)
Integer obfuscation	Apply mathematical operation to generate numerical value as an evaluation of mathematical expression
Whitespace and comment	Remove indentations, whitespaces, and comments and write the source script into a single line randomization
Identifier reassignment	Give alias names to the defined function calls to hinder tainting analysis
Block randomization	Manipulate nested control structures to hinder analysis even by web developers
String splitting	Exploit methods of string object to dynamically generate literals

As shown above, the obfuscated JavaScript should have some character distribution that is different from non-obfuscated JavaScripts in a string, and from this we should be able to leverage this difference to differentiate the two.

6.2 String Tokenization

There have been a decent number of works conducted on the topic of detecting JavaScript obfuscation [7,8], and some attempted to deobfuscate the JavaScripts back to it's original form. Most of these involve some string analysis methods and look to entropy analysis for classification. Through an iterative test process, I ended up employing a method closely resemble the 2016 paper by the group at Yokohama National University [7]. Instead of doing the analysis on the character of the JavaScript, the JavaScript is first tokenized into 4 different categories: lower case letters are converted into the '0' characters, upper case characters are converted into the '1' characters, numbers are converted to the '2' characters, and punctuations (except white space) are converted to the '3' characters (white spaces are removed in this process). This tokenization takes into account that obfuscated JavaScripts have a different distribution compared to the normal one, while keeping only four types of characters allows for more complicated feature extraction such as n-gram analysis without having too many feature vectors.

6.3 Work Flow

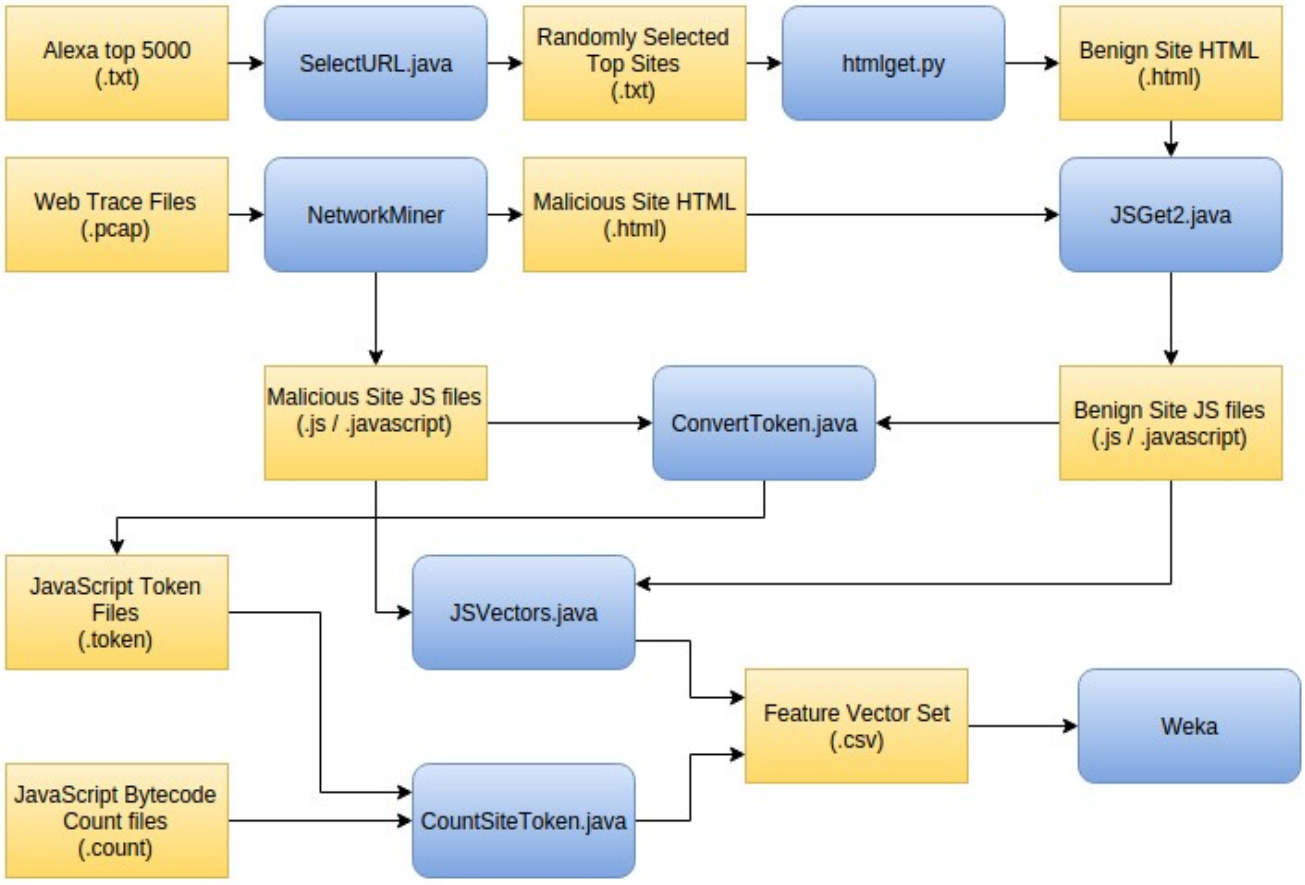


Fig. 4: Work flow of the classification process.

6.4 Data Source

Work flow to obtain the string features is shown in Fig. 4 above. For the benign inputs, the list of Alexa top 5000 sites is used as input and randomly selected 120 URLs (`SelectURL.java`) to be used as our benign sites. To obtain the HTML, the `wget` command is leveraged inside a python script to retrieve the HTML of these sites (running `htmlget.py`).

For the malicious inputs, I used the web trace data supplied by the AutoBLG team at Waseda to be the inputs. Initially I used `WireShark` to directly pull the files from HTTP export, but it turned out too time consuming for the large amount of sites for the `.pcap` file. I then looked for another tool for this task. The Network Miner tool takes in `.pcap` files and outputs all files in trace, and saves them inside folders

separated by the IP address. From these files, I am only interested in the JavaScript files and the HTML files, so all files except `.html` / `.htm` / `.js` and `.javascript` are removed first. Next, because the malicious sites also call and refer to many non-malicious sites, and this can potentially mess up our classification, I used the malicious URL list provided by Waseda to cross referenced to the IP – URL list from Network Miner, and kept only the folders with IP from the malicious URL themselves. We now have a set of folders containing JavaScript and HTML files originated from the malicious URLs themselves. This set is about 120 sites although it is reduced later slightly due to some sites having no or very little JavaScript features.

The JavaScript bytecode files are obtained from the same sources for bytecode analysis, the count files consist of purely the function calls of each of the bytecode functions, and allow us to tally up and gather the ratio and percentages for classification.

6.5 Obtaining and Tokenizing JavaScript

For the benign sites, from the HTML files, to obtain the JavaScript both inside the main body of the HTML itself and the external JavaScript files called by the HTML, the `JSGet2.java` program is used. The `JSGet2.java` file looks for the tag <script .....> and the tag </script> in the HTML file. If the starting script is found in the form <script>, the following section until </script> is assumed to be the JavaScript. If the phrase " src=" (the string "space"src=) is found in the script tag, it is assumed that the JavaScript is loaded from external file and different conditions are set to handle the URLs associated. If the tag is not a simple <script> and the " src=" phrase is not found, the phrase "javascript" is then looked for in the tag, and if it is found, the following section after the tag is then again assumed to be JavaScript and saved along with the simple <script> tag cases. A small number of JavaScripts are unable to be obtained inside the java program, which are saved in a log file and manually obtained through the browser.

For the malicious sites, the output HTML files from Network Miner are treated in the same way, and the JavaScript outputs from the Network Miner are set aside for the next step as well.

The JavaScript files are then tokenized into the 0, 1, 2, and 3 characters with whitespaces removed.

6.6 N-gram Analysis

From this reduced set of characters, n-gram analysis is used for classification. N=1 gram gives us the percentage of 0 characters, 1 characters, 2 characters and 3 characters in the JavaScripts, where N>1 gives us more sequence information. For example, for the N=3 gram, the feature vectors are the '000' sequence, '001' sequence, '002' sequence and up to the '333' sequence, where each feature is the percentage of that sequence showing up in the JavaScript. The number N = 3 is chosen from some testing results that will be explained later in the results section.

6.7 Input into Weka

For our classification, the popular machine learning tool Weka is leveraged.

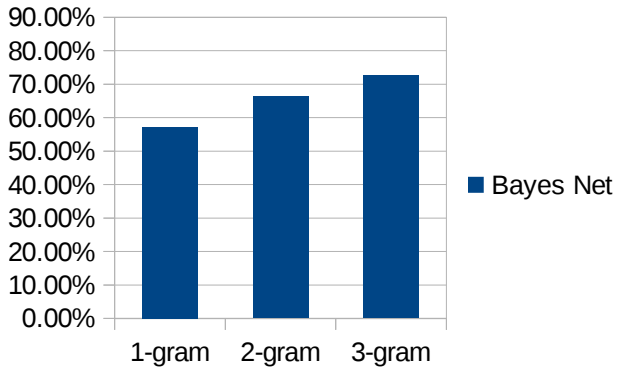
"Waikato Environment for Knowledge Analysis (Weka) is a popular suite of machine learning software written in Java, developed at the University of Waikato, New Zealand. It is free software licensed under the GNU General Public License." - Wikipedia

To classify in Weka, each row of the input `.csv` file corresponds to one of our websites, columns are the many feature vectors we are inputting, and one of the columns (usually the last) is the nominal variable on whether the website is malicious or benign. For the final classification input, 147 bytecode features (each bytecode function call to the total function call ratio), 64 N-gram features from N=3, and 5 JavaScript features (white space percentage, long string percentage, `eval()` percentage, `setTimeout()` percentage and `setInterval()` percentage) are used.

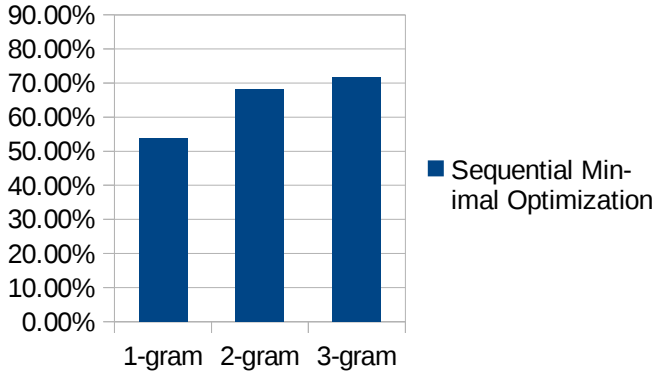
The JavaScript features are selected to round off our feature set to make up for the fact we are losing the white space information when tokenizing the JavaScripts, and the three JavaScript calls are proven by others [6] to be good indicators when classifying malicious web pages. Long string percentage here indicates ratio of strings that are longer than a certain threshold between whitespaces. The number through testing here is best at 40, which agrees with the results also presented by others [6]. Inside Weka, 10-fold cross validation is used to classify our results.

7. Results

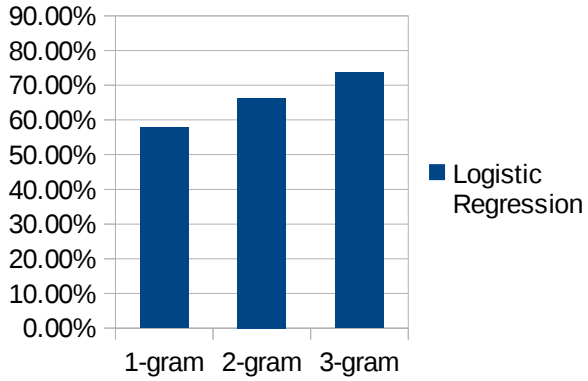
7.1 1-gram vs 2-gram vs 3-gram



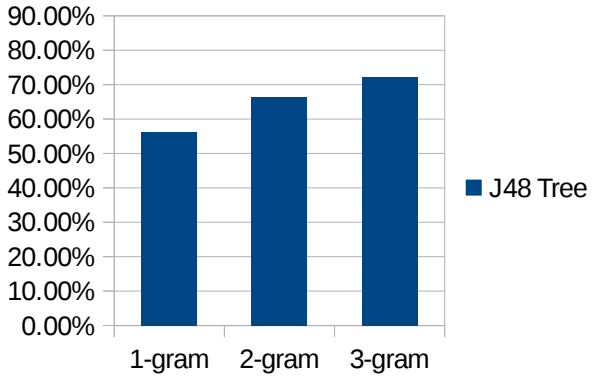
a) Classification Accuracy under Bayes Net



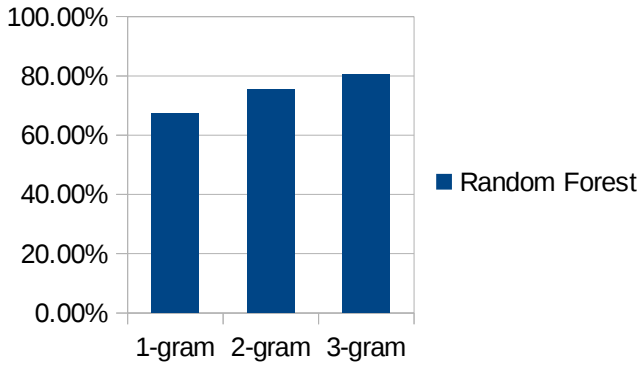
b) Classification Accuracy under Sequential Minimal Optimization



c) Classification Accuracy under Logistic Regression



d) Classification Accuracy under J48 Tree



e) Classification Accuracy under Random Forest

Fig. 5: Comparison of 1-gram vs 2-gram vs 3-gram tokenization.

Fig. 5 above is the classification accuracy of each of n-gram tokenization under five different commonly used classifiers. These classifications are under the tokenization features only and do not include any of the other features. For 1-gram, 4 features are used, for 2-gram, 16 features are used, and for 3-gram, 64 features are used. As expected, the classification accuracy increases as the number of features increases, but does slow down going from 2 to 3 gram compare to 1 to 2 gram, indicating this increase has plateaued. Due to this, and the exponential increase in the number of features when going up, 3-grams (64 features) are chosen as the stopping point for this part and will be part of our feature set for input into the final classification.

7.2 Complete Set of Features

For this complete set of features to be classified, 64 3-gram token features along with 147 bytecode features and 5 JavaScript features are used for a total of 216 features. As for number of sites, 105 benign sites and 94 malicious sites are classified. Five commonly used but different classifiers are used to observe the classification of our feature vector set.

Bayesian Net

A Bayesian network is a probabilistic graphical model that represents a set of random variables and their conditional dependencies via a directed acyclic graph.

```
Time taken to build model: 0.16 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         158               79.397  %
Incorrectly Classified Instances        41               20.603  %
Kappa statistic                          0.5896
Mean absolute error                      0.2095
Root mean squared error                  0.4473
Relative absolute error                 42.0245 %
Root relative squared error             89.58   %
Total Number of Instances              199

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC
                 0.743    0.149    0.848      0.743    0.792      0.595
                 0.851    0.257    0.748      0.851    0.796      0.595
Weighted Avg.    0.794    0.200    0.801      0.794    0.794      0.595

=== Confusion Matrix ===

  a  b   <-- classified as
 78 27 |  a = 0
 14 80 |  b = 1
```

Fig. 6: Classification result for Bayesian Network.

Fig. 6 shows the overall classification accuracy of the Bayesian Network classifier. The overall accuracy is 79.397%, true positive rate at 74.3% for the benign sites and 85.1% for the malicious sites.

Logistic Regression

Logistic regression is a regression model that measures the relationship between the categorical dependent variable (malicious 0 or 1 in our case) and one or more independent variables by estimating probabilities using a logistic function, which is the cumulative logistic distribution.

```
Time taken to build model: 0.34 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         152               76.3819 %
Incorrectly Classified Instances        47               23.6181 %
Kappa statistic                          0.5275
Mean absolute error                      0.2352
Root mean squared error                  0.4765
Relative absolute error                 47.1723 %
Root relative squared error             95.4164 %
Total Number of Instances              199

=== Detailed Accuracy By Class ===

                 TP Rate   FP Rate   Precision   Recall  F-Measure  MCC
                 0.752     0.223     0.790      0.752     0.771      0.528
                 0.777     0.248     0.737      0.777     0.756      0.528
Weighted Avg.    0.764     0.235     0.765      0.764     0.764      0.528

=== Confusion Matrix ===

  a  b   <-- classified as
 79 26 |  a = 0
 21 73 |  b = 1
```

Fig. 7: Classification result for Logistic Regression.

```
=== Classifier model (full training set) ===

Logistic Regression with ridge parameter of 1.0E-8
Coefficients...
                                        Class
Variable                                    0
=============================================
strictne                              -0.3814
string                                -0.0329
hole                                   0.5115
call                                   0.0034
strict-delelem                        78.4137
initelem_array                         0.3538
enditer                               -0.6786
length                                 0.4142
goto                                    0.277
setname                                1.0406
callee                               -26.6921
```

Fig. 7A: Snapshot of the coefficient weights for the Logistic Function.

Fig. 7 shows the overall classification accuracy of the Logistic Regression classifier. The overall accuracy is 76.3819%, true positive rate at 75.2% for benign sites and 77.7% for malicious sites. Fig. 7A is a partial snapshot (due to the high number of total features) of the coefficient weights associated with some of the feature vectors for the logistic function, some of the function calls with most weight associated include the bytecode features: `strict-delelem, strict-setgname, getrval, callee, Jssettimeout`; JavaScript features: `setTimeout, setInterval`.

SMO (Weka's method of SVM)

Support vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Sequential Minimal Optimization (SMO) is one way of solving the SVM training problem more efficiently than the standard quadratic programming solvers.

```
Time taken to build model: 0.12 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         160               80.402  %
Incorrectly Classified Instances        39               19.598  %
Kappa statistic                          0.6066
Mean absolute error                      0.196
Root mean squared error                  0.4427
Relative absolute error                 39.305  %
Root relative squared error             88.6516 %
Total Number of Instances              199

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC
                 0.819    0.213    0.811      0.819    0.815      0.607
                 0.787    0.181    0.796      0.787    0.791      0.607
Weighted Avg.    0.804    0.198    0.804      0.804    0.804      0.607

=== Confusion Matrix ===

  a  b   <-- classified as
 86 19 |  a = 0
 20 74 |  b = 1
```

Fig. 8: Classification result for SMO.

```
=== Classifier model (full training set) ===

SMO

Kernel used:
  Linear Kernel: K(x,y) = <x,y>

Classifier for classes: 0, 1

BinarySMO

Machine linear: showing attribute weights, not support vectors.

        -0.7001 * (normalized) strictne
 +       0.1042 * (normalized) string
 +      -0.2294 * (normalized) hole
 +       0.1891 * (normalized) call
 +      -0.299  * (normalized) strict-delelem
 +      -0.7891 * (normalized) initelem_array
 +       0.7363 * (normalized) enditer
 +      -0.3035 * (normalized) length
 +      -0.6055 * (normalized) goto
 +      -0.0555 * (normalized) setname
 +       0.276  * (normalized) callee
 +       0.2982 * (normalized) bindgname
 +       0.2438 * (normalized) arguments
 +      -0.0764 * (normalized) uint24
 +       0.0067 * (normalized) initlexical
```

Fig. 8A: Snapshot of the attribute weights for the support vectors.

Fig. 8 shows the overall classification accuracy of the SMO classifier. The overall accuracy is 80.402%, true positive rate at 81.9% for benign sites and 78.7% for malicious sites. Fig. 8A is a partial snapshot (due to the high number of total features) of the weights associated with the attributes for SMO. Some of the function calls with most weight associated include the bytecode features: `strictne`, `initelem_array`, `retrval`, `globalthis`, `loopentry`, `newobject`, `loophead`; 3-gram features: `001` pattern, `033` pattern, `233` pattern, `331` pattern; JavaScript features: `white_space_percentage`, `setTimeout`, `setInterval`.

Decision Tree (J48 in Weka)

Decision tree learning uses a decision tree as a predictive model which maps the observation about an item to the conclusions about the item's target value. Below is the tree generated and result.

```
Time taken to build model: 0.03 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         152               76.3819 %
Incorrectly Classified Instances        47               23.6181 %
Kappa statistic                          0.5265
Mean absolute error                      0.2372
Root mean squared error                  0.4708
Relative absolute error                 47.5657 %
Root relative squared error             94.2762 %
Total Number of Instances              199

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area
                 0.771    0.245    0.779      0.771    0.775      0.526    0.780
                 0.755    0.229    0.747      0.755    0.751      0.526    0.780
Weighted Avg.    0.764    0.237    0.764      0.764    0.764      0.526    0.780

=== Confusion Matrix ===

  a  b   <-- classified as
 81 24 |  a = 0
 23 71 |  b = 1
```

Fig. 9: Classification result for J48 tree.

```
=== Classifier model (full training set) ===

J48 pruned tree
------------------

void <= 1.154049
|   initelem <= 2.933239
|   |   initprop <= 324.74176
|   |   |   G032 <= 51.457363
|   |   |   |   330 <= 588.945482
|   |   |   |   |   222 <= 19.80198: 1 (29.0)
|   |   |   |   |   222 > 19.80198
|   |   |   |   |   |   implicitthis <= 1.160017
|   |   |   |   |   |   |   132 <= 9.036122
|   |   |   |   |   |   |   |   getelem <= 157.22301
|   |   |   |   |   |   |   |   |   bindgname <= 391.198044: 0 (14.0)
|   |   |   |   |   |   |   |   |   bindgname > 391.198044: 1 (2.0)
|   |   |   |   |   |   |   |   getelem > 157.22301: 1 (3.0)
|   |   |   |   |   |   |   132 > 9.036122: 1 (7.0)
|   |   |   |   |   |   implicitthis > 1.160017: 1 (2.0)
|   |   |   |   330 > 588.945482: 0 (11.0)
|   |   |   G032 > 51.457363: 1 (48.0/1.0)
|   |   initprop > 324.74176: 0 (7.0)
|   initelem > 2.933239: 0 (7.0)
void > 1.154049
|   getaliasedvar <= 863.507368: 0 (61.0)
|   getaliasedvar > 863.507368
|   |   rsh <= 0.20544: 0 (4.0)
|   |   rsh > 0.20544: 1 (4.0)

Number of Leaves  :     13

Size of the tree :      25
```

Fig. 9A: Pruned tree decisions for the J48 tree.

Fig. 9 shows the overall classification accuracy of the J48 tree classifier. The overall accuracy is 76.3819%, true positive rate at 77.1% for benign sites and 75.5% for malicious sites. Fig. 9A shows the tree pruning decisions for the classifier, some of the features selected to be nodes are `void`, `initelem`, `initprop`, `032` pattern etc.

Random Forest

Random forest operates by constructing a multitude of decision trees at training time and producing the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

```
RandomForest

Bagging with 100 iterations and base learner

weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 0.27 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         167               83.9196 %
Incorrectly Classified Instances        32               16.0804 %
Kappa statistic                          0.6774
Mean absolute error                      0.3093
Root mean squared error                  0.3542
Relative absolute error                 62.0415 %
Root relative squared error             70.9301 %
Total Number of Instances              199

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.848    0.170    0.848      0.848    0.848      0.677    0.927     0.937     0
                 0.830    0.152    0.830      0.830    0.830      0.677    0.927     0.921     1
Weighted Avg.    0.839    0.162    0.839      0.839    0.839      0.677    0.927     0.929

=== Confusion Matrix ===

  a  b   <-- classified as
 89 16 |  a = 0
 16 78 |  b = 1
```

Fig. 10: Classification result for Random Forest.

Fig. 10 shows the overall classification accuracy of the random forest classifier. The overall accuracy is 83.9196%, true positive rate at 84.8% for benign sites and 83.0% for malicious sites. The branching decisions are different from each iteration of tree construction and currently in Weka, the pruning decisions for Random Forest is not given as a output for this classifier.

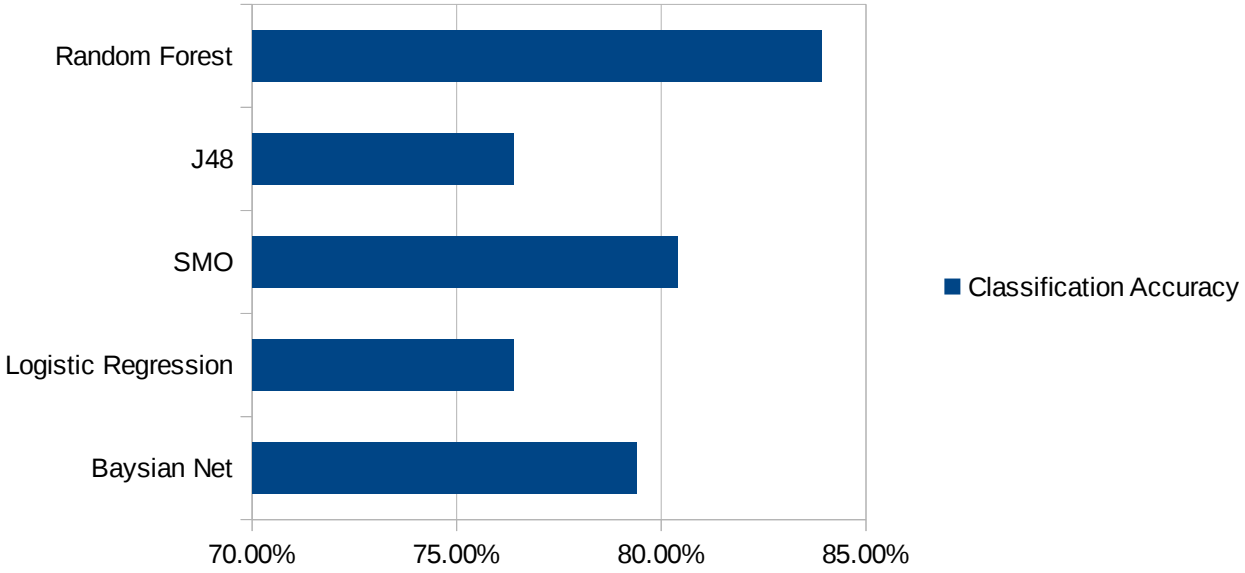


Fig. 11: Comparison of the different classifiers.

Shown by Fig. 11, the Random Forest and SVM based classifiers outperformed the other classifiers, which seem to be consistant with the results presented by others [5], perhaps indicating for this type of classification, Random Forest and SVM should be preferred. In terms of the feature vectors observed, our different classifiers give different weights for each of the features, so it is hard to pinpoint exactly which features are most telling. We can however, rank some of these features by weight associated, and the features with heavier weight overall in different classification algorithms should give us some pretty good indicators of important features for classification. Also some features are better indicators for benign sites while some features for better indicators for malicious sites. Depending on whether we are trying to optimize for true positive or true negative, the features' importance can be varied.

8. Future Improvements and Conclusion

Due to the time constraints and the scope of this project, I did not get a chance to integrate this JavaScript classification into the AutoBLG pipeline, so results currently are stand alone. If given the opportunity in the future I will try to integrate this process to observe possible classification improvements.

In terms of the source data from the malicious sites, from the traces I have extracted all JavaScript and HTML. However this does not capture the JavaScript and HTML that the trace did not see, and for the URL filtration step we would not have trace files. So for the best and consistent results, the malicious HTML and JavaScript should be obtained the same way as benign sites, which is to parse the HTML and get the JavaScript files while the site is fresh (of course most if not all of these sites from the trace are long gone now so it is not currently possible).

As for the feature vectors, not all vectors are equal in these classifications and as mentioned above, with more observations and iterations in the future we can isolate the better performing feature vectors to further improve the classification process.

Besides the current bytecode and string tokenization feature vectors, other techniques can also be considered, such as the lexical feature analysis of the URL set. There are many works in this area especially for phishing attack URLs, and whether these techniques are fitting for Drive-by-download URLs are still up for test.

9. References

[1] Sun, B., Akiyama, M., Yagi, T., Hatada, M., and Mori, T.: AutoBLG: Automatic URL Blacklist Generator Using Search Space Expansion and Filters. Proc. 20Th IEEE Symposium on Computers and Communication (ISCC 2015) pp. 205–211 (2015).

[2] Felegyhazi, M., Kreibich, C. and Paxson, V.: On the Potential of Proactive Domain Blacklisting, Proc. 3rd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More (LEET 2010), pp. 6 (2010).

[3] Ma, J., Saul, L.K., Savage, S. and Voelker, G.M.: Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs, Proc. 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2009), pp. 1245–1254 (2009).

[4] Xu, L., Zhan, Z., Xu, S., and Ye, K.: Cross-Layer Detection of Malicious Websites. Proc. *CODASPY*, pp. 141–152 (2013).

[5] Adachi, T., Omote, K.: An Approach to Predict Drive-by-Download Attacks by Vulnerability Evaluation and Opcode. 10th Asia Joint Conference on Information Security pp. 145 – 151 (2015).

[6] Canali, D., Cova, M., Vigna G., and Kruegel, C.: Prophiler: a fast filter for the large-scale detection of malicious web pages. In Proc. WWW, 2011 pp. 197-206 (2011).

[7] Su, J., Yoshioka, K., Shikata, and J., Matsumoto, T.: An Efficient Method for Detecting Obfuscated Suspicious JavaScript based on Text Pattern Analysis. International Workshop on Traffic Measurements for Cybersecurity, WTMC (2016).

[8] AL-Taharwa, I., Lee, H., Jeng, A., Wu, K., Ho, C., and Chen, S.: JSOD: JavaScript obfuscation detector. Security and Communication Networks, vol. 8: pp. 1092 – 1107 (2015).