Detection of Obfuscated Malicious JavaScript Code

Ammar Alazab 1,*, Ansam Khraisat 1, Moutaz Alazab 2 and Sarabjot Singh 1

1 School of Information Technology and Engineering, Melbourne Institute of Technology, Melbourne, VIC 3000, Australia; akhraisat@academic.mit.edu.au (A.K.); mit151308@stud.mit.edu.au (S.S.)
2 Faculty of Artificial Intelligence, Al-Balqa Applied University, Amman 1705, Jordan; m.alazab@bau.edu.jo
* Correspondence: aalazab@mit.edu.au

Abstract: Websites on the Internet are becoming increasingly vulnerable to malicious JavaScript code because of its strong impact and dramatic effect. Numerous recent cyberattacks use JavaScript vulnerabilities, and in some cases employ obfuscation to conceal their malice and elude detection. To secure Internet users, an adequate intrusion-detection system (IDS) for malicious JavaScript must be developed. This paper proposes an automatic IDS of obfuscated JavaScript that employs several features and machine-learning techniques that effectively distinguish malicious and benign JavaScript codes. We also present a new set of features, which can detect obfuscation in JavaScript. The features are selected based on identifying obfuscation, a popular method to bypass conventional malware detection systems. The performance of the suggested approach has been tested on JavaScript obfuscation attacks. The studies have shown that IDS based on selected features has a detection rate of 94% for malicious samples and 81% for benign samples within the dimension of the feature vector of 60.

Keywords: malware detection; intrusion detection; obfuscated malicious; machine learning; malicious JavaScript

1. Introduction

Most websites use JavaScript to enhance the usability and functionality of web applications. The JavaScript programming language, along with hypertext markup language (HTML) and cascading style sheets (CSS), is one of the three fundamental technologies for web development. Due to its ease of use and power in creating dynamic and interactive web pages, the use of JavaScript has become a standard among all web developers. According to a survey, JavaScript is used as a client-side programming language by 97.7% of all websites [1]. JavaScript code is interpreted in the user’s web browser and executed in the user’s processor instead of the web server. It allows for interacting with the document object model (DOM) of a web page and adding client-site behaviour to HTML pages. Some examples of this usage are animation of objects, validation of user input, and asynchronous communication. In addition to the web-based environment, JavaScript is also used in environments such as portable document format (PDF) documents, site-specific browsers, and desktop widgets [2].

JavaScript, not only brings versatility but also gives attackers new opportunities to exploit vulnerabilities in browsers and infect users with malicious JavaScript. Malicious JavaScript is a written program that is considered as a code that shows up as an unwanted behaviour such as by downloading and installing itself, spamming email or unwanted advertising. The main motive of obfuscated code is to fool the user to get it to install on the particular machine indirectly and exploit its execution. There are a few approaches for detecting the obfuscation of malicious JavaScript code which is like the honeypot technique or pattern-matching which fall under statistical analysis.

The creators of the malicious scripts exploited obfuscated JavaScript to conduct a variety of attacks, including cross-site request forgery (CSRF) as well as cross-site scripting.
(XSS). Existing intrusion-detection systems rely on professional expertise, yet this is a human-prone process even for specialists. To solve this problem, detecting malicious JavaScript as a defense mechanism has attracted more and more attention in cybersecurity research. The detection approaches can mainly be classified into three categories. The first category of approaches is signature-based [3]. Users create a signature for one malicious sample by generating a hash value or fingerprint, and then compare the signature to a blacklist. Although these techniques can effectively identify known harmful samples, they cannot identify variations with different hash values or fingerprints that have been updated or obscured [3,4]. The second category of approaches mainly focuses on static analysis by using machine learning techniques. These approaches extract features from the raw code of JavaScript and map each JavaScript sample to a point in the feature space, where malicious ones are separated from benign ones [5]. These approaches are promising and attractive, not only because they are scalable but also because they achieve impressive performance in simulations. However, they also have limitations. First, new characteristics are easily dodged, necessitating hundreds of thousands of data for classifiers to achieve high accuracy. Second, they cannot be utilised to categorise attack types or identify new assaults originating from malicious JavaScript. The third category of approaches tries to execute JavaScript samples and analyse their behaviours by using techniques such as honey clients or sandbox. In contrast to the static analysis on raw code, these approaches fall in the class of dynamic analysis. These approaches are normally more accurate than approaches in the first two categories, because they are able to overcome challenges resulting from attackers obfuscating malicious JavaScript [6]. But the biggest drawback is that they are not scalable and require much more time and other resources [7].

Obfuscation is the primary technique used by attackers to disguise their attacks [8]. Attackers attempt to obfuscate JavaScript to evade signature-based and static analysis approaches. Based on the processes performed, four kinds of obfuscation strategies are distinguishable among attackers [9].

1. Randomization obfuscation: Without altering the logic of JavaScript codes, attackers are able to arbitrarily insert or modify certain components. Typical methods include randomising whitespace, variable, comments, and functions names.
2. Data obfuscation: One or more variables and constants are transformed into their computational outputs by this method. String splitting and keyword substitution are both extensively used methods.
3. Encoding obfuscation: There are normally three ways adopted by attackers to encode original code: converting the code into escape ASCII characters, Unicode or hexadecimal representations, and equipping it with customized encoding and decoding functions, and employing encryption and decryption methods.
4. Logic structure obfuscation: This includes changing the execution flow by inserting redundant instructions or modifying some conditional branches.

The study [10] demonstrates that all popular antivirus software may be easily circumvented by using a variety of obfuscation methods. However, it is not true that a JavaScript code is malicious if it is obfuscated. Obfuscation is also regularly utilized by web developers to protect code privacy and intellectual property or improve efficiency. Most notably, heavy usage of JavaScript obfuscation is seen among online advertising vendors. However, people have realized that obfuscation is not equivalent to malignancy [9]. This is an obvious simplification of the malicious JavaScript detection problem, which limits these approaches’ performance in real-world applications and impairs people’s confidence in these approaches. The paper by Al-Taharwa et al. [11] is the first work that faces the non-equivalence between obfuscation and malignancy, and the detection problem is split into two subproblems: distinguishing obfuscated from unobfuscated, and distinguishing obfuscated malicious from obfuscated benign.

If we acknowledge the fact that not all obfuscated JavaScript codes are malicious, it is natural to treat the detection problem as a classification problem of two hierarchies. On the
higher level, we only consider whether a JavaScript sample is obfuscated or not. This is the main focus of existing intrusion-detection systems. Then, we have two branches leading to the lower level, and the two corresponding subproblems are classifying an unobfuscated code as malicious or benign and classifying an obfuscated code as malicious or benign. Which subproblem should be solved depends on the results from the higher level. We believe that splitting the problem into subproblems could not only improve detection performance but also reduce computing resources.

In this paper, we demonstrate the planning and implementation of an intrusion-detection system that distinguishes malicious from benign JavaScript code swiftly. We use statistical methods to analyse features of JavaScript code and use machine learning techniques to build a classification model. JavaScript code that is found to be malicious can then raise alarms to the user or be further analysed by experts. Our techniques automatically extract feature attributes, as opposed to previous methods that hand-crafted feature attributes. In addition, the dimensions of the learned features are small, resulting in a quicker detection.

This paper is structured as follows. In Section 2, we address similar work. In Section 3, extracted characteristics and selection techniques are explored. The experimental setup and findings are presented in Section 4. In Section 5, concluding remarks on future work are provided.

2. Related Work

The process of deriving useful information from vast amounts of data is referred to as machine learning. Models of machine learning consist of a set of rules, methods, or sophisticated “transfer functions” that can be utilised to locate relevant patterns in data or to recognise or anticipate behaviour. These models can be implemented to either find or create new data patterns [12]. In the field of anomaly intrusion-detection systems, machine learning approaches have seen substantial application in recent years. A variety of algorithms and approaches, including clustering and neural networks, rules for association and decision trees, as well as genetic algorithms and closest neighbour methods, are used to extract information from intrusion datasets.

There is some historical study that has investigated the usage of a variety of methods to construct anomaly-based intrusion detection systems (AIDS). Chebrolu et al. studied the performance of two feature selection procedures involving Bayesian networks (BN) and classification regression trees (CRC), and merged these methods for improved accuracy. The results of their research were published in the journal Computers in Biology and Medicine [13].

Information gain (IG) and correlation attribute evaluation were two of the feature selection methods that were combined in Bajaj et al.‘s suggested method for feature selection, which uses a combination of the aforementioned algorithms. They evaluated the functionality of the chosen characteristics by using a variety of classification approaches, including C4.5, naïve Bayes, NB-Tree, and multi-layer perceptron, among others [14,15]. In order to determine the relative relevance of IDS traits, a genetic-fuzzy rule mining technique was utilised [16]. The random tree model was utilised by Thaseen et al. in order to improve accuracy and reduce the rate of false alarms in their NIDS proposal [17]. It was recommended by Subramanian et al. to classify the NSL-KDD dataset by utilising decision tree algorithms to develop a model with respect to their metric data, as well as evaluate the performance of tree-based techniques [18].

The principles of machine learning have been applied to the development of a variety of anti-AIDS drugs. The primary goal of developing IDS through the application of machine learning approaches is to reduce the amount of human expertise that is required while simultaneously improving accuracy. Over the past few years, there has been a discernible rise in the quantity of AIDS applications that make use of machine learning strategies. The primary goal of IDS research that is based on machine learning is to identify patterns and construct an intrusion-detection system for a given dataset. In the realm of machine learning, there are often two sorts of approaches: supervised and unsupervised.
The Zarathustra research software provides a facility to read the DOM memory of a web browser [19]. A copy of the DOM for a specific website is taken from a clean virtual machine (VM) and a second copy is taken after the VM has been infected with information-stealing malware. The Zarathustra software examines the differences between the infected and uninfected DOM to develop web inject signatures related to the malware family being tested. The Zarathustra software is written in Java and makes use of the Selenium WebDriver for Firefox. The Zarathustra software was written in 2014. The Zarathustra software was built to encounter the problems communicating with the Firefox web driver. This is due to changes in the web driver protocol which occurred after the completion of the Zarathustra research. It was decided to look for other methods for reading the DOM rather than spend time recoding the Zarathustra software.

Through the use of static analysis, Peiser et al. identified malicious JavaScript code by feeding locality-sensitive hashes into a feed-forward neural network as input features [20].

There have been suggestions made for techniques that make use of machine learning in order to identify malicious JavaScript programs [21]. One example of this would be monitoring its execution upon a JavaScript code at run time by using a sequence of events to collect vectors for categorisation. Learning to recognise dangerous patterns inside the structure and operation of JavaScript code is another strategy that can be utilised [22].

Feature clustering can also be accomplished with the assistance of a wrapper technique and a classifier [23]. This strategy results in the generation of a feature subset via feature selection. The method employs a feature set that is not comprehensive, and there is a high probability that the wrapper method will experience overfitting as well as a protracted processing time.

Attackers with malicious intentions use JavaScript to carry out attacks such as drive-by download attempts, XSS, and CSRF. Due to the number of such attacks, manually detecting malicious scripts by using a professional’s specific knowledge is error-prone and difficult. Deep learning and a neural network called the bidirectional long short-term memory (BLSTM) are used in Song et al.’s [24] innovative method for identifying malicious JavaScript code. This method is based on deep learning, and it uses the BLSTM neural network. Additionally, they constructed a program-dependency graph to extract JavaScript’s semantic meaning. The model achieved an accuracy of approximately 97.7 percent.

Martin et al. [25] proposed an efficient machine learning strategy for detecting network intrusion. They included network addresses in the IDS dataset because they were helpful features. An innovative method for translating (encoding) source and destination network addresses, which are high-dimensional categorical variables, into a more manageable set of scalar values that express the likelihood of sharing a network connection at various granularities within the network address hierarchy has been proposed.

Feature matching or static word embeddings cannot spot the difference between obfuscated and unobfuscated JavaScript code. Huang et al. [26] introduced JSContana to address this issue by combining flexible context analysis with efficient key feature extraction. They used dynamic word embeddings to retrieve the real contextual representation of JavaScript code during the translation process.

Conventional procedures mainly depend on signature as well as heuristic-based methods, both of which are vulnerable to zero-day attacks. As a consequence, conventional methodology produce a substantial number of false negatives and/or positives. To address this issue, Ndichu et al. [27] uses a machine learning method dubbed Doc2Vec, which is a neural network model capable of learning text context information. The collected features are fed into a classifier model (for example, SVMs and neural networks), which determines the maliciousness of JavaScript code.

Rozi et al. [28] created a deep neural network for assessing the bytecode sequences of malicious JavaScript code and recognizing harmful JavaScript code to protect consumers from JavaScript-related cyberattacks. They generated a bytecode sequence by making use of the V8 JavaScript compiler. A bytecode sequence is an abstract idea of machine code. In addition to this, they combined a deep pyramid convolutional neural network, also
known as a DPCNN, with recurrent neural network models that were capable of handling long-range interactions in a bytecode sequence. This was done in order to discern the malicious intentions of the attacker.

Martin et al. [29] made significant contributions by extending the gaNet architecture to incorporate categorization, analyzing future extensions, and introducing the correct classifier (gaNet-C) to two difficult traffic forecasting problems: active and elephant connections.

Radanliev et al. [30] presented a novel epistemological equation developed and evaluated the use of comparative and empirical analysis. Following the comparative examination of national digital initiatives, an empirical analysis of cyberrisk-assessment methodologies was completed. Additionally [31] investigates how AI algorithms can work on low memory/limited computing IoT devices and also how AI can be developed and created to generate and compose its own algorithms.

There are several research works for detecting malicious JavaScript code in web applications. In the measurement study of Wei Xu et al. [32], they illustrate the influence of obfuscation methods in malware JavaScript code. By examining the detection efficiency of the 20 greatest common antivirus vendors to detect obfuscation malicious JavaScript, they provide the evidence of the detail that most prevalent antivirus vendors use the signature intrusion detection system (SIDS), for which cause most anti-virus vendors couldn’t identify obfuscated malicious JavaScript code precisely.

Many machine learning techniques have been used to identify JavaScript malware and assess the accuracy and performance of detecting various classes of JavaScript malware. Ndichu et al. [10] collect a dataset of obfuscated and non-obfuscated JS codes and selects and extracts a set of 45 features from the dataset. The features employed include frequency of given keywords, number of lines, characters per line, number of functions, and entropy, among others. They are unable to identify obfuscated JavaScript not existing in the training set.

Using machine learning classification to detect malicious scripts does have a disadvantage. Specifically, machine learning classification techniques are expected to classify a small subset of normal scripts as possible JavaScript malware. One example of normal and obfuscated JavaScript is packed JavaScript. Some web applications select to compress JavaScript before communicating it to users to decrease the data transmitted or avoid the theft of their source code. With packed JavaScript, it is possible to create a false positive and it may stop users from accessing these websites. Therefore, to improve the detection performance of machine learning, we extract the feature that could detect obfuscated JavaScript malware.

Likarish et al. [33] use the controlled frequency of each JavaScript keyword as a feature and build the detection model with four supervised machine learning techniques: NaiveBayes, ADtree, SVM and RIPPER. The limitation of this technique is that it is involved only with the normalized frequency of each JavaScript keyword and disregards further important features in the code.

Fraiwan et al. [34] examine the behavior of JavaScript code to create the intrusion-detection system. Their methods extracted four sets of features for the detection JavaScript malicious code: URL attributes, JavaScript code results, JavaScript code activities, and JavaScript code content. However, given that this technique is based on static analysis, they have limitations in analyzing dynamic features of JavaScript code and detecting obfuscated JavaScript code.

3. Feature Extraction

Our purpose is to design a classifier with feature selection, which could produce the best accuracy for each class of malicious JavaScript patterns. The first step is to construct the different connection models to achieve the best simplification performance for classifiers. Each feature will be rated as “very important”, “important”, or “unimportant” according to the following rules:

1. If accuracy high and training time high, then the feature is important.
2. If accuracy high and training time low, then the feature is very important.
3. If accuracy low and training time high, then the feature is unimportant.
4. If accuracy low and training time low, then the feature is unimportant.
5. If accuracy unchanged and false alarm decreased, then the feature is important.

These principles of selection were used by means of information gain. Information gain, initially applied to calculate splitting criteria for decision trees, is frequently used to discover how well each single attribute splits the given dataset. The general entropy \( I \) of a given dataset \( S \) is defined \([5]\) as

\[
I(S) = -\sum_{i=1}^{c} p_i \log_2 p_i
\]

where \( c \) denotes the total number of classes and \( p_i \) the portion of instances that belong to class \( i \). The reduction in entropy or the information gain is computed for each attribute according to

\[
IG(S, A) = I(S) - \sum_{a \in A} \frac{|S_{A=a}|}{|S|} I(S_{A=a})
\]

where \( va \) value of is \( A \) and \( S_{A,v} \) the set of instances where \( A \) value has \( v \). We applied information gain into 71 features as the quality of the feature selection is one of the most important factors that affect the effectiveness of IDS. The stages of the experiment are shown in Figure 1.

![Figure 1. Experiment methodology.](image)

1. Feature selection stage: In this stage, an information theoretical feature selection approach is used to normalize the training and test dataset for generating reduced feature set selection.
2. Classification stage: This comprises two phases, specifically the training phase and the testing phase.
3. Analysis of the result: After the testing phase, we calculate the accuracy rate, false alarm rate, and the time to build the model.

The static analysis of JavaScript files produces characteristics that can be used in JavaScript. The features of JavaScript can be broken down into two categories: statistical and lexical. To extract features out of each section of JavaScript code, a total of 170 characteristics are used. Table 1 outlines the characteristics along with brief explanations of each one. Figure 2 shows the correlation coefficients of different features.

In practical implementations of machine learning, the number of characteristics that result is typically quite enormous, yet many of those do not contribute to accuracy and may even reduce it. In this study, a decreasing drop in the number of attributes is an important factor, and it is imperative that this process be carried out while preserving a high degree of accuracy. This is because the detection process on client computers should not impede the browsing experience of customers.

We first extract the above 170 features and run an analysis on the effectiveness of these features. Then feature selection methods are used to determine the effectiveness. We plot a bar chart for each feature to visualize the difference in values between malicious and benign samples. We calculate the correlation coefficient to measure the strength of the relationship between a feature and a group of samples. Based on the correlation coefficient, we can only select the top features to decrease the dimension of the feature vector as shown in Figure 2. Figure 3 shows the visualization of differences between malicious and benign samples for each feature.
Table 1. Feature extraction.

<table>
<thead>
<tr>
<th>ID</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Length of the script</td>
<td>The number of chars in the script, without comments</td>
</tr>
<tr>
<td>2</td>
<td># of lines</td>
<td>The number of lines</td>
</tr>
<tr>
<td>3</td>
<td>Script entropy</td>
<td>The Shannon entropy of the script as a whole, without comments</td>
</tr>
<tr>
<td>4</td>
<td>Avg. line length</td>
<td>The average length of line</td>
</tr>
<tr>
<td>5</td>
<td>% of strings</td>
<td>The percentage of chars belonging to strings</td>
</tr>
<tr>
<td>6</td>
<td>% of whitespace</td>
<td>The percentage of whitespace</td>
</tr>
<tr>
<td>7</td>
<td>Ratio of comments to script</td>
<td>The ratio of chars in comments to chars in the script</td>
</tr>
<tr>
<td>8</td>
<td># of comments</td>
<td>The number of comments, including inline and multiline comments</td>
</tr>
<tr>
<td>9</td>
<td>Avg. comments per line</td>
<td>The average number of chars in comments per script line</td>
</tr>
<tr>
<td>10</td>
<td># of strings</td>
<td>The number of strings *</td>
</tr>
<tr>
<td>11</td>
<td>Avg. string entropy</td>
<td>The average Shannon entropy of strings</td>
</tr>
<tr>
<td>12</td>
<td>Maximum string entropy</td>
<td>The maximum Shannon entropy of strings</td>
</tr>
<tr>
<td>13</td>
<td>Avg. string length</td>
<td>The average number of chars per string</td>
</tr>
<tr>
<td>14</td>
<td>Maximum string length</td>
<td>The maximum length of strings</td>
</tr>
<tr>
<td>15</td>
<td># of long strings</td>
<td>The number of long strings that have more than 40 chars</td>
</tr>
<tr>
<td>16</td>
<td># of “iframe” strings</td>
<td>The number of strings that contain “iframe”</td>
</tr>
<tr>
<td>17</td>
<td># of suspicious tag strings</td>
<td>The number of strings that contain tags that can be utilized for malicious purposes, such as “script”, “object”, “embed”, and “frame”</td>
</tr>
<tr>
<td>18</td>
<td># of suspicious strings</td>
<td>The number of suspicious strings that contain “evil”, “shell”, “spray”, and “crypt”</td>
</tr>
<tr>
<td>19</td>
<td># of long variable or function names</td>
<td>The number of long variables or function names that have more than 15 chars</td>
</tr>
<tr>
<td>20</td>
<td>Avg. # of arguments per function</td>
<td>The average number of arguments per function</td>
</tr>
<tr>
<td>21</td>
<td>Avg. argument length</td>
<td>The average length of function arguments</td>
</tr>
<tr>
<td>22</td>
<td>Avg. length of the function body</td>
<td>The average number of chars per function body</td>
</tr>
<tr>
<td>23</td>
<td>% of function body</td>
<td>The percentage of chars belonging to function bodies</td>
</tr>
<tr>
<td>24</td>
<td>Ratio of # of functions to the script length</td>
<td>The ratio of the number of function definitions to the script length</td>
</tr>
<tr>
<td>25</td>
<td># of calls to set event handlers</td>
<td>The number of function calls to set event handlers, such as document.addEventListener on events: on error, on load, on before unload, on unload</td>
</tr>
<tr>
<td>26</td>
<td># of encoded chars</td>
<td>The number of encoded chars, including unicode chars and hex numbers</td>
</tr>
<tr>
<td>27</td>
<td># of backslash chars</td>
<td>The number of backslash chars</td>
</tr>
<tr>
<td>28</td>
<td># of pipe chars</td>
<td>The number of pipe chars</td>
</tr>
</tbody>
</table>
Table 1. Cont.

<table>
<thead>
<tr>
<th>ID</th>
<th>Feature Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>29–39</td>
<td>The number of occurrence of each built-in function: eval, setTimeout, setInterval, unescape, replace, document.write, charAt, substring, String.fromCharCode, String.charCodeAt, navigator.userAgent</td>
</tr>
<tr>
<td>40–50</td>
<td>The proportion of the total length of the script that is comprised of instances of a built-in function.</td>
</tr>
<tr>
<td>51–110</td>
<td>The number of occurrences of each JavaScript keyword, totally 60 keywords</td>
</tr>
<tr>
<td>111–170</td>
<td>The amount of times a keyword appears in relation to the total number of words in a screenplay.</td>
</tr>
</tbody>
</table>

* defined as datatype.

Figure 2. Correlation coefficients of different features. They are in decreasing order.
4. Experiments

The machine learning approach adopted here consists of data collection, feature extraction, training, and testing. We collected a dataset containing several JavaScript for both malicious and benign groups. We retrieved a collection of attributes for every one of the samples within the dataset, which were determined by feature analysis. The retrieved features are then utilised to generate fixed-length feature vectors for training and testing.

4.1. JavaScript Collection

The dataset contains data from two distinct sources.

1. The Alexa Top 500 websites: Downloading the JavaScript discovered on the Alexa Top 500 homepages provided a more understandable picture of actual scripts available on websites. To retrieve the scripts from such websites, BeautifulSoup was used to parse them and extract all inlined scripts. (e.g., `<script>alert("foo");</script>`). For our evaluation, we assume samples in this dataset are non-malicious and non-obfuscated. There are 4342 samples.

2. A set of malicious JavaScript tests from the VX Heaven (vxheaven.org). There are only malicious samples included in the VX Heaven repository. The majority of the malicious samples contained in the dataset are either JavaScript downloaders that are utilised in malspam operations or Exploit Kits resources that are utilised for the purpose of exploiting vulnerabilities in browser plugins. Almost all of the samples are, to some extent, obfuscated, and it appears that several obfuscation methods and tools were used. There are total of 119 malicious samples.

4.2. Model Configuration

In this study, we make use of a support vector machine, often known as an SVM. The following are some of SVM’s benefits: effective in large dimensional spaces; employs a subset of training examples in the decision function, which means it also is memory efficient; alternative kernel functions can be chosen for the decision function in order to meet a variety of circumstances [8]. We use Scikit-learn, a machine learning package for Python, to implement SVM. The parameters are: C = 3, kernel = ‘linear’, and gamma = ‘auto’. 
Because the quantity of benign samples is much greater than that of malicious samples, our data is highly imbalanced. In order to address the issue of class imbalance, we adopt a classifier-independent approach to make sure the training data is class-balanced. We use 60% of malicious samples as training data and the remaining 40% as testing data. Then we arbitrarily select the identical number of benign samples as training data and the left as testing data. The cross-validation is applied 10 times, and thus 10 datasets are generated. The results displayed below are averages of the results of the 10 rounds.

Based on correlation coefficients in the feature analysis, we select the top 30, 60, and 100 features. We will compare how this setting will affect the performance of the classifier.

### 4.3. Experiment Results

Figure 4 displays the accuracy of the categorisation when it was examined by using the malicious and benign JavaScript samples discussed previously. The level of accuracy can be determined by taking the total number of samples and dividing it by the number of successful classifications. The findings are presented in the figure with a breakdown according to the amount of features that were included in the classifier.

The values show that the classifier has the best performance when the dimension of the feature vector is 60, with the accuracy of 94% for malicious samples and 81% for benign samples. The dimension of 30 makes the classifier have a little better performance on malicious samples but not on benign samples. One thing we should mention here is that the case of 30 features needs significantly less time to train the classifier than the other two cases.

![Figure 4. The classification accuracies for the three different dimensions of the feature vector.](image)

Figure 5 provides further information regarding the findings presented above by illustrating the false positive rate (FPR) and the false negative rate (FNR) for each set of characteristics. The ratios are calculated as one fraction of samples that are malicious and samples that are benign, respectively.

According to the figure, the rate of false positives is quite high for all configurations, although the rate of false negatives is comparatively low. This is in contrast to the fact that the rate of false positives is relatively high. In the best case scenario, which makes use of 60 characteristics, only 6% of harmful samples are misclassified.

However, the large false-positive rate will cause many false alarms and may compromise clients' user experience. We will further look into this issue from several different aspects, including optimizing parameters for classifier and feature extraction. Our purpose is to have high overall precision and a low false-positive ratio.

An IDS is typically evaluated based on the following traditional performance measures:

- **True positive (TP):** Number of accurately identified malicious codes.
- **True negative (TN):** Number of accurately identified benign codes.
• False positive (FP): Number of incorrectly identified benign code, when an indicator identifies the benign file as malware.
• False negative (FN): Number of incorrectly identified malicious code, when an indicator fails to identify the malware because the virus is new and no signature is still available.
• Total Accuracy: Proportion of entirely precise classified instances, either one positive or negative.

Figure 5. The false positive rate (FPR) and false negative rate (FNR) for three different dimensions of feature vector.

The confusion matrix for a two-class classifier, which is the kind that is typically utilised in an IDS, is presented in Table 2. The examples that belong to each anticipated class are represented along the columns of the matrix, whereas the instances that belong to each actual class are represented along the rows.

Table 2. Confusion matrix of an IDS for evaluation purpose.

<table>
<thead>
<tr>
<th>Actual Class</th>
<th>Predicted Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
</tr>
<tr>
<td>Normal</td>
<td>True negative (TN)</td>
</tr>
<tr>
<td>Attack</td>
<td>False Negative (FN)</td>
</tr>
</tbody>
</table>

The detailed analysis of the accuracy of SVM classification on dataset shown in Table 3.

Table 3. Detailed accuracy of using SVM classifier.

<table>
<thead>
<tr>
<th>TP</th>
<th>Rate</th>
<th>FP</th>
<th>Rate</th>
<th>F-Measure</th>
<th>ROC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.989</td>
<td>0.272</td>
<td>0.961</td>
<td>0.989</td>
<td>0.858</td>
<td>malware</td>
</tr>
<tr>
<td>0.728</td>
<td>0.011</td>
<td>0.907</td>
<td>0.728</td>
<td>0.858</td>
<td>normal</td>
</tr>
<tr>
<td>0.955</td>
<td>0.239</td>
<td>0.954</td>
<td>0.955</td>
<td>0.858</td>
<td></td>
</tr>
</tbody>
</table>

Confusion matrix results for the SVM classifier is shown in Table 4.
Table 4. Confusion matrix results.

<table>
<thead>
<tr>
<th>Malware</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>4296</td>
<td>48</td>
</tr>
<tr>
<td>176</td>
<td>470</td>
</tr>
</tbody>
</table>

Figure 6 provides the evaluation of accuracy of our methodology with the state-of-the-art works. Figure 5 shows that the SVM produces slightly better accuracy than other existing malicious JavaScript detection methods.

The detailed analysis of the accuracy of the naive Bayes classification for the dataset shown in Table 5.

Table 5. Detailed accuracy of using naive Bayes classifier.

<table>
<thead>
<tr>
<th>TP Rate</th>
<th>FP Rate</th>
<th>F-Measure</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.344</td>
<td>0.074</td>
<td>0.508</td>
<td>Malware</td>
</tr>
<tr>
<td>0.926</td>
<td>0.656</td>
<td>0.292</td>
<td>Normal</td>
</tr>
<tr>
<td>0.419</td>
<td>0.15</td>
<td>0.48</td>
<td>Weighted Avg</td>
</tr>
</tbody>
</table>

The detailed analysis of the accuracy of the sequential minimal optimization (SMO) classification on dataset shown in Table 6.

Table 6. Detailed accuracy of using SMO classifier.

<table>
<thead>
<tr>
<th>TP Rate</th>
<th>FP Rate</th>
<th>F-Measure</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.989</td>
<td>0.272</td>
<td>0.975</td>
<td>malware</td>
</tr>
<tr>
<td>0.728</td>
<td>0.011</td>
<td>0.808</td>
<td>normal</td>
</tr>
<tr>
<td>0.955</td>
<td>0.239</td>
<td>0.953</td>
<td>Weighted Avg</td>
</tr>
</tbody>
</table>

Figure 7 provides the evaluation of accuracy of different classification methods. Figure 7 provides the evaluation of the accuracy of our methodology with the machine learning techniques. Figure 7 shows that the SVM produces better accuracy than other machine learning techniques. Figure 8 provides the evaluation of the time taken to build a model between the SVM and different classification methods. Naïve Bayes produces
less time to build the model but does not provide good accuracy. SVM gives a good time for building the model and best accuracy result. Therefore, SVM is selected for detecting malicious JavaScript.

![Figure 7](image1.png)

**Figure 7.** The comparison of accuracy between the SVM and different classification methods.

![Figure 8](image2.png)

**Figure 8.** The comparison of time taken to build model between the SVM and different classification methods.

5. Conclusions and Future Work

Many malicious JavaScripts that are used both on the client-side and on the server-side are obfuscated to evade the detection of signature-based detection systems. To mitigate this, in this paper we proposed a novel technique for the prevention and detection of malicious JavaScript codes that uses anomaly-detection techniques. A total of 170 features are extracted and we ran an analysis of the effectiveness of these features. Then machine learning was used to develop an intrusion-detection system. Our techniques automatically extracted feature attributes contrasted to other previous approaches which use manually created feature attributes. The data for the analysis was compiled by doing the analysis on a sample of 10,000 websites, 5000 of which were trusted and 5000 of which were not trusted.
This method has been tested on a substantial corpus of actual JavaScript code from the real world and is now available to the general public online. The findings of the evaluation indicate that it is possible to detect malicious code in a reliable manner by employing emulation to exert the (possibly hidden) behaviour of the script and trying to compare this actions with a (learned) model of regular JavaScript code execution. This process was carried out in order to determine whether or not it is possible to accurately detect malicious code. Experimental results indicated that our approach could detect JavaScript malware with a high detection accuracy of 98% by using SVM.

Author Contributions: Data curation, A.A. and A.K.; Formal analysis, M.A.; Funding acquisition, A.A.; Investigation, A.A.; Methodology, A.K. and S.S.; Project administration, A.K.; Validation, S.S.; Writing—review & editing, S.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Melbourne Institute Technology and The APC was funded by Melbourne Institute Technology.

Data Availability Statement: Not Applicable, the study does not report any data.

Acknowledgments: The research was supported by Melbourne Institute of Technology. The authors are grateful to Melbourne Institute of Technology for their support.

Conflicts of Interest: The authors declare that they have no conflict of interest.

Abbreviations
The following abbreviations are used in this paper:

AIDS Anomaly-based Intrusion Detection System
SVM Support Vector Machine
IDS Intrusion Detection System
CRT Classification Regression Trees

References


