future internet

MDPI

Article

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

Citation: Alazab, A.; Khraisat, A.; Alazab, M.; Singh, S. Detection of Obfuscated Malicious JavaScript Code. *Future Internet* **2022**, *14*, 217. https://doi.org/10.3390/fi14080217

Academic Editors: Ivan Serina and Wei Yu

Received: 19 May 2022
Accepted: 20 July 2022
Published: 22 July 2022

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, naive 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 Web Driver 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 a 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 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 $IG(S, A) = I(S) - \sum_{v\mathcal{E}A} \frac{|S_{A,v}|}{|S|} I(S_v)$ 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.

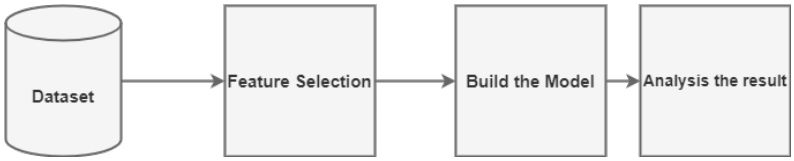

Figure 1. Experiment methodology.

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.

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

Table 1. *Cont.*

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

* defined as datatype.

ID	Descriptions	r	|r|	p-value
27	% of backslash chars	0.3446719	0.3446719	7E-123
36	# of occurences of fromCharCode	0.2981514	0.2981514	5.72E-91
47	ratio of num of occurences of fromCharCode to script length	0.2865197	0.2865197	7.442E-84
33	# of occurences of eval	0.2442366	0.2442366	9.847E-61
26	% of encoded chars	0.2103943	0.2103943	3.517E-45
40	ratio of num of occurences of charAt to script length	0.1987196	0.1987196	2.129E-40
14	maximum string length	0.1752279	0.1752279	1.177E-31
29	# of occurences of charAt	0.173474	0.173474	4.756E-31
37	# of occurences of unescape	0.1366667	0.1366667	8.827E-20
44	ratio of num of occurences of eval to script length	0.1366466	0.1366466	8.937E-20
5	% of string chars	0.1224823	0.1224823	3.634E-16
78	# of occurences of while	0.1189251	0.1189251	2.538E-15
119	ratio of num of occurences of finally to script length	0.1026608	0.1026608	8.884E-12
138	ratio of num of occurences of while to script length	0.0988079	0.0988079	5.163E-11
13	avg. string length	0.0953272	0.0953272	2.393E-10
39	# of occurences of write	0.0782793	0.0782793	2.023E-07
49	ratio of num of occurences of charCodeAt to script length	0.0731488	0.0731488	1.2E-06
12	avg. string entropy	0.0667411	0.0667411	9.455E-06
23	% of chars belonging to a function body	-0.064112	0.0641116	2.096E-05
24	ratio of # of functions to the script length	-0.062146	0.0621463	3.728E-05
123	ratio of num of occurences of function to script length	-0.062146	0.0621463	3.728E-05
6	% of whitespace	-0.061151	0.0611513	4.958E-05
38	# of occurences of charCodeAt	0.0452145	0.0452145	0.00271
59	# of occurences of finally	0.0435868	0.0435868	0.0038428
1	script length	0.042479	0.042479	0.0048437
16	# of "iframe" strings	0.0404414	0.0404414	0.0073183
20	avg num of arguments per function	0.0291864	0.0291864	0.0529643
21	avg. argument length	0.0288318	0.0288318	0.055916
4	avg chars per line	0.0284708	0.0284708	0.0590603
19	# of long variables or function names	0.0281235	0.0281235	0.0622224
85	# of occurences of do	0.0271831	0.0271831	0.0714934
121	ratio of num of occurences of var to script length	-0.026879	0.0268788	0.0747255

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

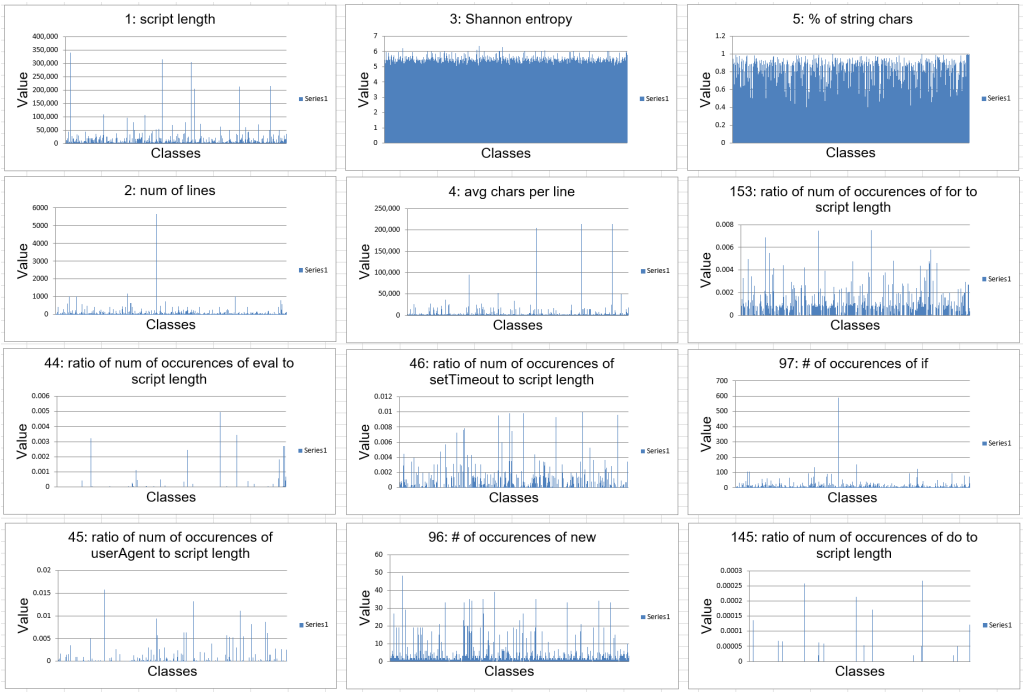

Figure 3. Visualization of differences between malicious and benign samples for each feature.

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. (eg., <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, out 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.

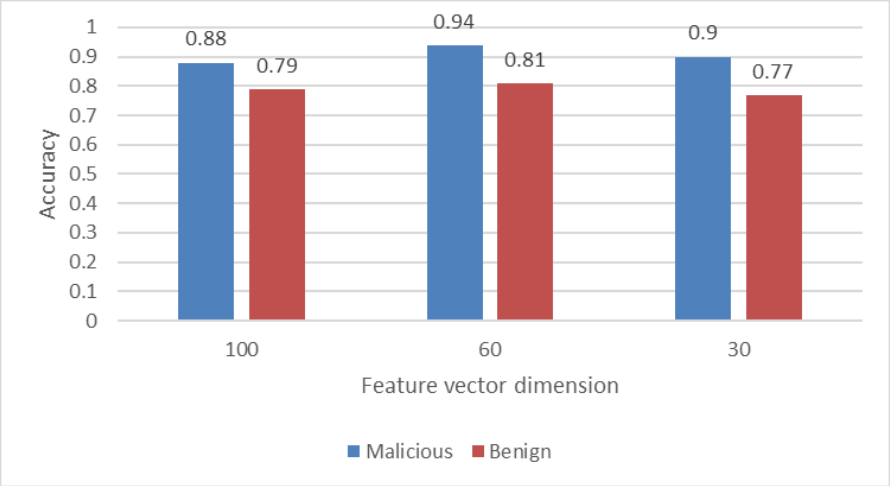

Figure 4. The classification accuracies for the three different dimensions of the feature vector.

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.

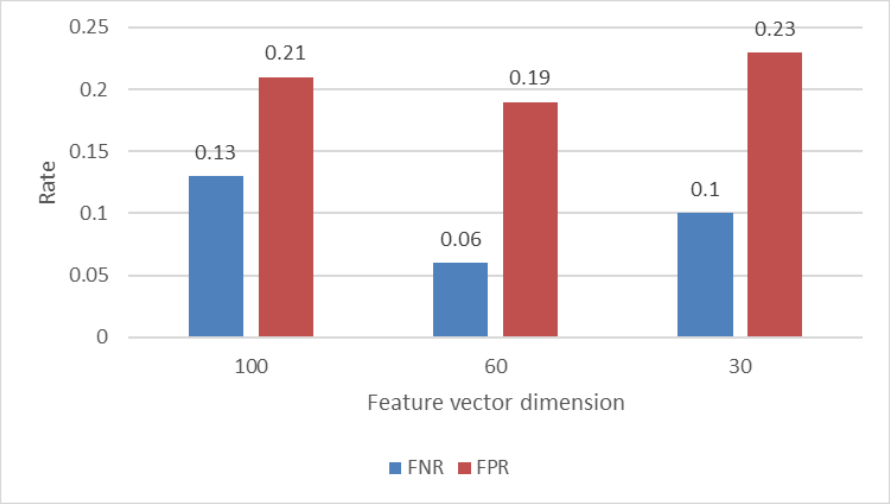

Figure 5. The false positive rate (FPR) and false negative rate (FNR) for three different dimensions of feature vector.

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.

Actual Class		**Predicted Class**	
		Normal	Attack
	Normal	True negative (TN)	False Positive (FP)
	Attack	False Negative (FN)	True positive (TP)

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

Table 3. Detailed accuracy of using SVM classifier.

TP	**Rate**	**FP**	**Rate**	**F-Measure**	**ROC**
0.989	0.272	0.961	0.989	0.858	malware
0.728	0.011	0.907	0.728	0.858	normal
0.955	0.239	0.954	0.955	0.858	

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

Table 4. Confusion matrix results.

Malware	Normal	
4296	48	Malware
176	470	Normal

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.

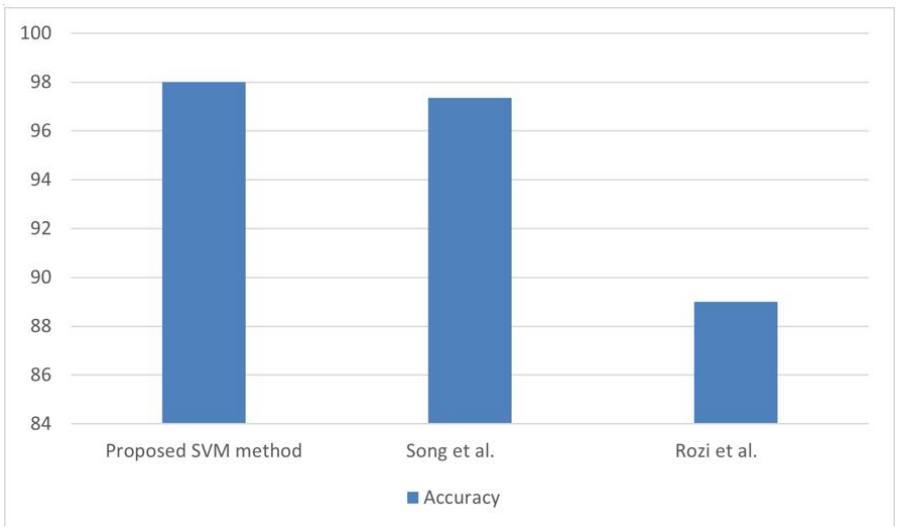

Figure 6. The comparison of accuracy between the proposed SVM based model with the existing works [24,28].

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.

TP Rate	FP Rate	F-Measure	Class
0.344	0.074	0.508	Malware
0.926	0.656	0.292	Normal
0.419	0.15	0.48	Weighted Avg

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.

TP Rate	FP Rate	F-Measure	Class
0.989	0.272	0.975	malware
0.728	0.011	0.808	normal
0.955	0.239	0.953	Weighted Avg

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.

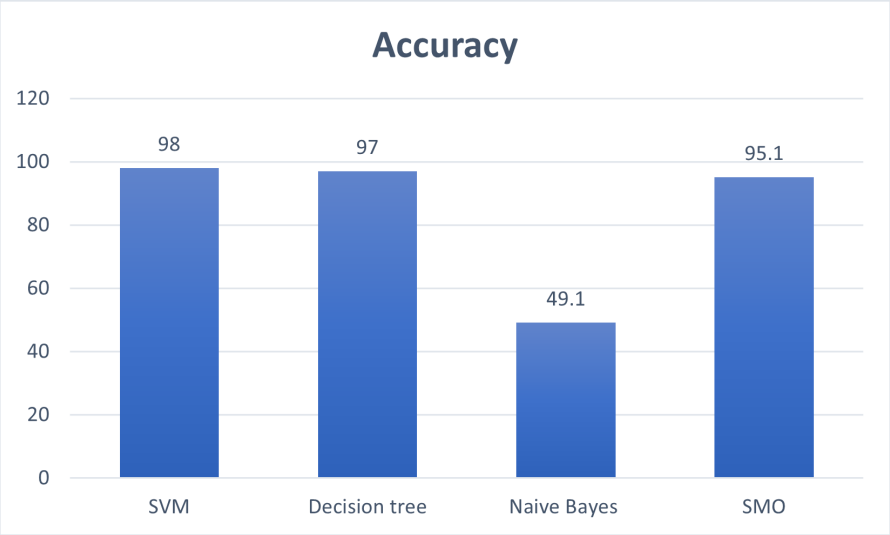

Figure 7. The comparison of accuracy between the SVM and different classification methods.

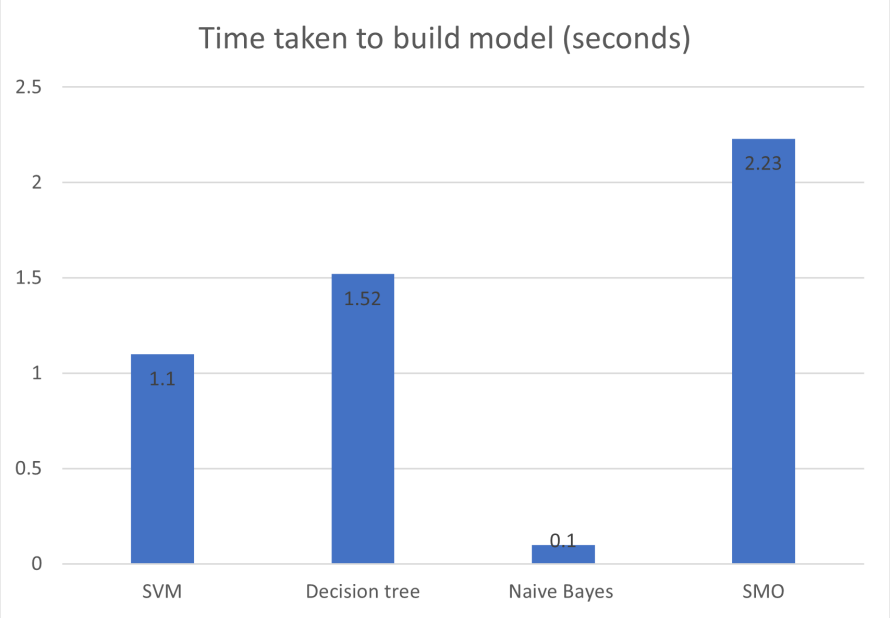

Figure 8. The comparison of time taken to build model between the SVM and different classification methods.

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

1. W3techs. Usage Statistics of Client-Side Programming Languages for Websites. 2021. Available online: https://w3techs.com/technologies/overview/client_side_language (accessed on 16 May 2022).
2. Korać, D.; Damjanović, B.; Simić, D. Information security in M-learning systems: Challenges and threats of using cookies. In Proceedings of the 2020 19th International Symposium INFOTEH-JAHORINA (INFOTEH), Sarajevo, Bosnia and Herzegovina, 18–20 March 2020; pp. 1–6.
3. Kim, G.; Lee, S.; Kim, S. A novel hybrid intrusion detection method integrating anomaly detection with misuse detection. *Expert Syst. Appl.* **2014**, *41*, 1690–1700. [CrossRef]
4. Khraisat, A.; Alazab, A. A critical review of intrusion detection systems in the internet of things: techniques, deployment strategy, validation strategy, attacks, public datasets and challenges. *Cybersecurity* **2021**, *4*, 18. [CrossRef]
5. Alazab, A.; Hobbs, M.; Abawajy, J.; Alazab, M. Using feature selection for intrusion detection system. In Proceedings of the 2012 International Symposium on Communications and Information Technologies (ISCIT), Gold Coast, QLD, Australia, 2–5 October 2012; pp. 296–301.
6. Andreasen, E.; Gong, L.; Møller, A.; Pradel, M.; Selakovic, M.; Sen, K.; Staicu, C.A. A survey of dynamic analysis and test generation for JavaScript. *ACM Comput. Surv. (CSUR)* **2017**, *50*, 1–36. [CrossRef]
7. Sihwail, R.; Omar, K.; Zainol Ariffin, K.A.; Al Afghani, S. Malware detection approach based on artifacts in memory image and dynamic analysis. *Appl. Sci.* **2019**, *9*, 3680. [CrossRef]
8. Fass, A.; Krawczyk, R.P.; Backes, M.; Stock, B. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Saclay, France, 28–29 June 2018; pp. 303–325.
9. Khraisat, A.; Gondal, I.; Vamplew, P.; Kamruzzaman, J. Survey of intrusion detection systems: Techniques, datasets and challenges. *Cybersecurity* **2019**, *2*, 20. [CrossRef]
10. Ndichu, S.; Kim, S.; Ozawa, S. Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement. *CAAI Trans. Intell. Technol.* **2020**, *5*, 184–192. [CrossRef]
11. AL-Taharwa, I.A.; Lee, H.; Jeng, A.B.; Wu, K.; Ho, C.; Chen, S. JSOD: JavaScript obfuscation detector. *Secur. Commun. Netw.* **2015**, *8*, 1092–1107. [CrossRef]
12. Dua, S.; Du, X. *Data Mining and Machine Learning in Cybersecurity*; CRC Press: Boca Raton, FL, USA, 2016.

13. Chebrolu, S.; Abraham, A.; Thomas, J.P. Feature deduction and ensemble design of intrusion detection systems. *Comput. Secur.* **2005**, *24*, 295–307. [CrossRef]
14. Bajaj, K.; Arora, A. Dimension Reduction in Intrusion Detection Features Using Discriminative Machine Learning Approach. *IJCSI Int. J. Comput. Sci. Issues* **2013**, *10*, 324–328.
15. Khraisat, A.; Gondal, I.; Vamplew, P. *An Anomaly Intrusion Detection System Using C5 Decision Tree Classifier*; Springer International Publishing: Berlin/Heidelberg, Germany, 2018; pp. 149–155.
16. Elhag, S.; Fernández, A.; Bawakid, A.; Alshomrani, S.; Herrera, F. On the combination of genetic fuzzy systems and pairwise learning for improving detection rates on Intrusion Detection Systems. *Expert Syst. Appl.* **2015**, *42*, 193–202. [CrossRef]
17. Thaseen, S.; Kumar, C.A. An analysis of supervised tree based classifiers for intrusion detection system. In Proceedings of the 2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering, Salem, India, 21–22 February 2013; pp. 294–299. [CrossRef]
18. Subramanian, S.; Srinivasan, V.B.; Ramasa, C. Study on classification algorithms for network intrusion systems. *J. Commun. Comput.* **2012**, *9*, 1242–1246.
19. Criscione, C.; Bosatelli, F.; Zanero, S.; Maggi, F. ZARATHUSTRA: Extracting Webinject signatures from banking trojans. In Proceedings of the 2014 Twelfth Annual International Conference on Privacy, Security and Trust, Toronto, ON, Canada, 23–24 July 2014; pp. 139–148. [CrossRef]
20. Peiser, S.C.; Friborg, L.; Scandariato, R. *JavaScript Malware Detection Using Locality Sensitive Hashing*; Springer International Publishing: Berlin/Heidelberg, Germany, 2020; pp. 143–154. [CrossRef]
21. Jordan, A.; Gauthier, F.; Hassanshahi, B.; Zhao, D. SAFE-PDF: Robust Detection of JavaScript PDF Malware Using Abstract Interpretation. *arXiv* **2018**, arXiv:1810.12490.
22. He, X.; Xu, L.; Cha, C. Malicious JavaScript code detection based on hybrid analysis. In Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, 4–7 December 2018; pp. 365–374.
23. Patil, D.R.; Patil, J.B. Detection of Malicious JavaScript Code in Web Pages. *Indian J. Sci. Technol.* **2017**, *10*, 1–12. [CrossRef]
24. Song, X.; Chen, C.; Cui, B.; Fu, J. Malicious JavaScript Detection Based on Bidirectional LSTM Model. *Appl. Sci.* **2020**, *10*, 3440. [CrossRef]
25. Lopez-Martin, M.; Carro, B.; Arribas, J.I.; Sanchez-Esguevillas, A. Network intrusion detection with a novel hierarchy of distances between embeddings of hash IP addresses. *Knowl.-Based Syst.* **2021**, *219*, 106887. [CrossRef]
26. Huang, Y.; Li, T.; Zhang, L.; Li, B.; Liu, X. JSContana: Malicious JavaScript detection using adaptable context analysis and key feature extraction. *Comput. Secur.* **2021**, *104*, 102218. [CrossRef]
27. Ndichu, S.; Ozawa, S.; Misu, T.; Okada, K. A Machine Learning Approach to Malicious JavaScript Detection using Fixed Length Vector Representation. In Proceedings of the 2018 International Joint Conference on Neural Networks (IJCNN), Rio de Janeiro, Brazil, 8–13 July 2018; pp. 1–8. [CrossRef]
28. Rozi, M.F.; Kim, S.; Ozawa, S. Deep Neural Networks for Malicious JavaScript Detection Using Bytecode Sequences. In Proceedings of the 2020 International Joint Conference on Neural Networks (IJCNN), Glasgow, UK, 19–24 July 2020; pp. 1–8. [CrossRef]
29. Lopez-Martin, M.; Carro, B.; Sanchez-Esguevillas, A. IoT type-of-traffic forecasting method based on gradient boosting neural networks. *Future Gener. Comput. Syst.* **2020**, *105*, 331–345. [CrossRef]
30. Radanliev, P.; De Roure, D.; Burnap, P.; Santos, O. Epistemological equation for analysing uncontrollable states in complex systems: Quantifying cyber risks from the internet of things. *Rev. Socionetwork Strateg.* **2021**, *15*, 381–411. [CrossRef] [PubMed]
31. Radanliev, P.; De Roure, D. Review of algorithms for artificial intelligence on low memory devices. *IEEE Access* **2021**, *9*, 109986–109993. [CrossRef]
32. Xu, W.; Zhang, F.; Zhu, S. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software, Fajardo, PR, USA, 16–18 October 2012; pp. 9–16.
33. Likarish, P.; Jung, E.; Jo, I. Obfuscated malicious javascript detection using classification techniques. In Proceedings of the 2009 4th International Conference on Malicious and Unwanted Software (MALWARE), Montreal, QC, Canada, 13–14 October 2009; pp. 47–54.
34. Fraiwan, M.; Al-Salman, R.; Khasawneh, N.; Conrad, S. Analysis and identification of malicious javascript code. *Inf. Secur. J. Glob. Perspect.* **2012**, *21*, 1–11. [CrossRef]