JavaScript Malware Detection Using Locality Sensitive Hashing

Stefan Carl Peiser, Ludwig Friborg, Riccardo Scandariato

JavaScript malware detection using locality sensitive hashing

*Stefan Carl Peiser, *Ludwig Friborg, and Riccardo Scandariato

[1] Chalmers University of Technology, Gothenburg, Sweden
stefancarlpeiser@gmail.com
[2] Chalmers University of Technology, Gothenburg, Sweden
ludwig.friborg@gmail.com
[3] Chalmers and University of Gothenburg, Gothenburg, Sweden
riccardo.scandariato@cse.gu.se

Abstract. In this paper, we explore the idea of using locality sensitive hashes as input features to a feed-forward neural network with the goal of detecting JavaScript malware through static analysis. An experiment is conducted using a dataset containing 1.5M evenly distributed benign and malicious samples provided by the anti-malware company Cyren. Four different locality sensitive hashing algorithms are tested and evaluated: Nilsimsa, ssdeep, TLSH, and SDHASH. The results show a high prediction accuracy, as well as low false positive and negative rates. These results show that LSH based neural networks are a competitive option against other state-of-the-art JavaScript malware classification solutions.

Keywords: Malware · LSH · Neural network · JavaScript

1 Introduction

JavaScript is one of the most popular scripting languages in the world as it is the 'de facto' scripting language used by internet browsers. This means that JavaScript has become a popular attack vector to infect computers of internet users as these scripts are executed automatically by browsers. In this paper we focus on static techniques to detect malicious JavaScript code, as static approaches are simpler to apply and have a performance advantage. However, detecting malicious code statically has become difficult due to code obfuscation. On top of that, in the world of JavaScript, code obfuscation is not an indicator of maliciousness as most JavaScript code on benign websites is obfuscated as a side-effect of minimizing the size of production code and preserving intellectual property.

In this paper we present an approach that works on both clear-text and obfuscated scripts. In particular, we explore the use of locality sensitive hashing

* These authors contributed equally to this work.

(LSH) as a means to extract features from the scripts. The features are fed to a neural network for the effective identification of malicious scripts. LSH is a family of dimensionality reducing algorithms, which previously has been used for document and code comparison and is used here in a novel way for malware detection.

In Section 2 we introduce background material and survey the related work. We present our approach in Section 3. In Section 4, we evaluate the approach on a large corpora of malware samples and compare the results to several alternative approaches from the state of the art (including Cujo and Zozzle). In Section 5 we discuss and investigate possible causes for false positives and false negatives during our experimentation. Finally, we present the concluding remarks in Section 6.

2 Background and related work

2.1 JavaScript malware

Almost all web pages today utilize JavaScript in some form, whether to display fancy animations or to send data to web servers. Browsers have started to run JavaScript files automatically when loading websites, which has enabled many new attack vectors. JavaScript malware have various purposes. Many try to download other malware onto the victim's computer, e.g. remote access trojans (RATs), ransomware and more, these are commonly known as drive-by-downloads malware. Other common types of malware are bitcoin miners where the malware uses the infected computer's hardware to mine cryptocurrency. Facelikers are also common, as they try to "like" various posts and pages on Facebook using infected Facebook accounts.

Often, hackers obfuscate the code of malware in an attempt to make it harder to analyse and detect. However, obfuscation is not necessarily an indicator of maliciousness as it has become the norm in JavaScript development the last few years as a way of minimizing code, hide client-side code and more.

2.2 Identification of JavaScript malware

There are several malware detection techniques that have been proposed in the state of the art. In this section we focus on the most prominent approaches, which are also used as comparison in Section 4.1. For a more complete coverage of malware identification, we refer the interested reader to the survey of Ye et al. [20].

Dynamic analysis. Ratanaworabhan et al. [13] propose a runtime heap-spraying attack detector named Nozzle. The system has been used to analyse JavaScript-based malware. Nozzle uses emulation techniques to detect executable malicious code in objects allocated within the browser heap.

A drawback with using dynamic methods is that they are often resource intensive and thus expensive to use at runtime. Thus, it is prevalent among

security vendors to use dynamic analysis methods to assess the scripts off-line and, at runtime, just compare script files with a collection of already classified samples.

Static analysis. Ndichu et al. [11] proposes using Doc2Vec to extract features from malicious JavaScript files and then feed them into a support vector machine model. The performance of the classifier is promising but the validation dataset consists of only 80 files.

Curtsinger et al. [3] propose a method named Zozzle. They evaluate both a handpicked and a automated feature extraction method to then infer the maliciousness of a JavaScript file through a naive Bayesian classifier. It is important to note that their system is only able to function on unobfuscated code.

Xu et al. [19] propose a method named *JStill*, which operates on obfuscated code. This method works by analysing code and looking for blacklisted function calls. It is important to note that the approach relies on white/black lists. Therefore, the method is limited to cover only a subset of all JavaScript malware.

Likarish et al. [9] evaluate multiple different statistical learning methods together with a tokenized feature extraction method based on different keywords. Among the methods evaluated, the models with the lowest false positive rate are ADTree [5] and RBF SVM [2]. Wang et al. [18] later provide a more refined presentation of the results presented by Likarish. They also present a deep learning approach, called SdA-LR, based on the previously mentioned feature extraction method and a deep neural network for statistical inference.

Rieck et al. [14] propose a system called Cujo, which leverages three different methods of JavaScript malware analysis. One is static, one is dynamic and one is the combination of the previous two. The static method utilizes support vector machines to learn the patterns of malicious scripts. The dynamic method uses sandboxing. The work focuses on detecting one specific type of malware, namely the drive-by-download family.

Although not related to JavaScript, the work of Raff et al. [12] is worth mentioning. They train a deep learning model that consumes entire malware executable binaries. Thus, the model learns how the malware are structured internally. However, performance is a major drawback in this approach, as it takes a month to train the model on a dataset of 2M executable binaries.

2.3 Locality Sensitive Hashing

Raff et al. [12] show that using deep learning to learn structural properties of malware seems to be a powerful way of classifying them. However, the bottleneck is represented by the time and resources it takes to learn on entire malware files. Instead of processing whole files, our idea is to find a dense representation of the file contents and to infer characteristics from said representation. Hence this paper focuses on the use of locality sensitive hashing methods to provide concise input features for a neural network.

Locality Sensitive Hashing (LSH) is a relatively new family of dimensionality-reducing algorithms, including Nilsimsa [4], TLSH [10], ssdeep [8], and SDHASH [15], which are evaluated in this work. These algorithms produce condensed

Table 1. List of the most prevalent types of malicious scripts.

Malware Type	Count	%
Redirector	166857	20.4
Trojan downloader	43505	5.3
CoinHive	6285	0.8
SEOHide	4394	0.5
IFrame	3629	0.4
FaceLiker	2285	0.3
Ramnit	1615	0.2
FakejQuery	1073	0.1
Crypted	938	0.1
Unknown type	588153	71.8
Total	818734	

representations (hashes) of the given input data. By construction, the hashes of similar files are also similar[4], hence the hashes can be used as proxies in order to compare the similarity of the original files. The benefit is that the hashes are much more concise and lend themselves to be used as features in learning algorithms.

3 Experimental setup

3.1 Dataset

The dataset contains about 1.5M scripts, of which 54% are malicious. Table 1 describes the different malware types that are present in the dataset. The data is provided by Cyren (`https://www.cyren.com`), which is a large vendor in the field of cybersecurity and supplies, among other, the scanner for email attachments used by Google and Microsoft [1]. All JavaScript files in the dataset have been collected and labeled during the first half of 2019. The files originate from various sources, e.g. from web scrapers, customers sending in files for analysis, e-mail attachments, incoming files from VirusTotal [17] and more. Each of these files goes through Cyren's malware scanners (based on dynamic analysis) and the system assigns a label to the sample indicating whether it is clean or malicious. These labels represent our ground truth.

3.2 Feature extraction

As shown in Figure 1, the locality sensitive hashes are pre-processed before being used as input to the neural network. Thus we have to take into account

[4] This contrasts to cryptographic hashing techniques, like SHA256, where the hashing algorithm minimizes the probability of collisions, i.e., two almost identical files yield two drastically different hashes.

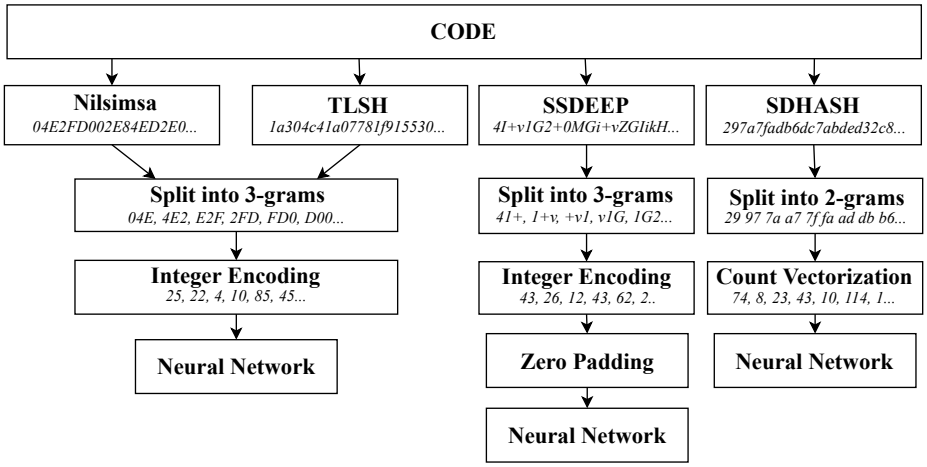

Fig. 1. Feature extraction and prediction pipeline.

the different characteristics of the hashes. Both TLSH and Nilsimsa produce a fixed-length, hex-encoded strings of 70 and 64 characters respectively. SSDEEP produces a hash that is base64 encoded and its length is variable, but has a max size of 148 characters. Finally, SDHASH produces hex encoded hashes of variable length, but with no maximum limit.

To let the neural network find patterns in the substrings of the hashes we decided to split the hashes into n-grams by using a sliding window (of size n and sliding of 1 position at a time). As detailed later, the learning algorithm (namely, the embedding layer) uses a dictionary whose size is 16^n for TLSH and Nilsimsa (hex encoding), and 64^n for ssdeep (base64 encoding). A larger dictionary has an impact on the training time and the memory consumption. Therefore, after experimentation, the trade-off decision has been made to use tri-grams. During the experimentation, we also found that using $n \geq 4$ did not yield any noticeable classification improvements but a high increase in training time.

After splitting the hash into n-grams, each hash is then encoded as a sequence of integers, i.e., each n-gram is converted to its positional value. After the encoding, we we are left with input vectors of different size for each LSH type. In the case of TLSH and Nilsimsa, the vectors are of fixed size and they are used as-is to train a neural network. In the case of SSDEEP, the vectors have variable length but, due to the nature of the output from this algorithm, there is an upper bound. In this case, we take the length of the longest vector and add zero-padding to the vectors so that they have the same length.

SDHASH produces output hashes with no definitive maximum length and no upper bound. Hence, for this hashing algorithm, the construction of the features is different. Starting from the hash, we split it into a vector of bi-grams (in place of tri-grams) and filter the vector through a count vectorizer, which returns a vector of frequencies for each unique bi-gram. We use the vector of frequencies

Table 2. Network model composition (where L is length of input vector).

Layer name	Output dimensions	
Embeddings	32xL	
Flatten	1xL	
BatchNormalization	1xL	
Dense	1x256	activation: relu
Dropout	1x256	probability: 0.125
Dense	1x64	activation: relu
Dense	1x1	activation: sigmoid

as input vector for the neural network. Note that the ordering of bi-grams gets lost in the process, which might negatively affect the performance of the neural network performance. The choice of using bi-gram is justified by the fact that, in this way, the input vector is of similar size with respect to the other algorithms.

3.3 Neural network design and implementation

A supervised learning approach with a normal deep feed-forward neural network is used to classify each locality sensitive hash. The input layer of the neural network takes the integers generated from each hash, and the output layer will return one single value, presented on a scale between 0 and 1, determining the likelihood of the input of being malicious. Table 2 provides an overview of the network structure. The embedding layer transforms positive integers into dense non-zero vectors. This was chosen to mitigate the problem of it having a high presence of sparse input-vectors in addition to some hashing methods producing hashes of an inconsistent length leading to a lot of 0-padding. Not embedding the input data resulted in worse performance and slower convergence rate for the learning model. We use both *Batch Normalization* [6] and *Dropout* [16] for regularization.

In terms of fitting the network to gain an accurate understanding of the given data *adam optimization* [7] was used together with binary cross-entropy as loss-function.

3.4 Experiments and performance indicators

By means of random sampling, we split the complete dataset into seven subsets of incrementally bigger sizes, namely 5k, 10k, 50k, 100k, 500k, 1M, 1.5M (i.e., the whole set). We use subsets of varying sizes in order to investigate the trade-off between prediction performance and training cost. Ultimately, we would like to understand how much data is necessary in order to generalize. Note that, as we use random sampling, the positive rate in the subsets is expected to be similar to the complete dataset.

For each subset and each LSH method, we run a 5-fold cross-validation experiment and measure the average performance of the prediction approach. As

Table 3. Results from the 5-fold cross-validation experiment.

LSH	ACC (%)	FPR (%)	FNR (%)
TLSH	97.79	1.01	3.25
Nilsimsa	98.05	1.09	2.69
ssdeep	97.97	0.94	2.98
SDHASH	95.06	1.83	7.63

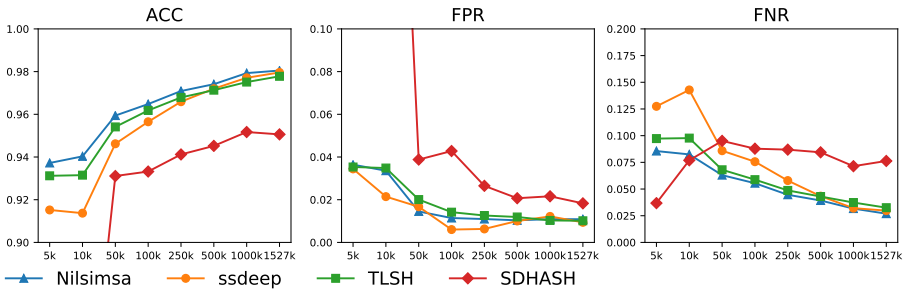

Fig. 2. Accuracy, False Positive Rate and False Negative Rate across the different experiments with increasingly larger dataset sizes.

we use 5-fold cross-validation, the results we report are averaged over the performance obtained in the individual folds.

To assess the different prediction models, we rely on three performance indicators: accuracy (ACC), false positive rate (FPR), and false negative rate (FNR). The key performance indicators are FPR and FNR. However, we also include accuracy for comparison reasons, as this indicator is often reported by the other approaches we compare to (cf. Section 4.1). The performance indicators are calculated as follows:

- $ACC = \frac{TP+TN}{TP+TN+FP+FN}$
- $FPR = \frac{FP}{FP+TN}$
- $FNR = \frac{FN}{FN+TP}$

where TP, TN, FP and FN corresponds to the number True/False Positives/Negatives.

4 Results

Table 3 shows the results from the cross-validation experiment on 1.5M samples. Figure 2 shows the results for all the experiments with different dataset sizes. It is possible to observe that Nilsimsa has a slight advantage compared to the other methods. Interestingly, and contrary to expectations, the SDHASH model,

Table 4. Comparison of the performance indicators between our models and the state of the art, where *M:C* corresponds to *Malware:Clean*, which is the amount of samples used.

Classifier	ACC (%)	FPR (%)	FNR (%)	M : C
Zozzle manual	98.2	1.50	1.20	900:8000
Zozzle auto	99.2	0.30	9.20	900:8000
JStill	97.3	17.5	0.53	30k:50k
RBF SVM	86.8	4.92	8.33	14k:12k
ADTree	82.7	2.42	14.92	14k:12k
SdA-LR	94.8	4.13	6.04	2959:2464
CUJO static	90.1	0.10	9.80	609:200k
Ours - TLSH	97.79	1.01	3.25	818k:709k
Ours - Nilsimsa	98.05	1.09	2.69	818k:709k
Ours - ssdeep	97.97	0.94	2.98	818k:709k

which used a count-vectorized style of network input, also seems to produce good results, though falling short against the other LSH methods. The results also show that the models are more prone to making false negative predictions rather than false positives, which is a beneficial trait in the world of malware detection.

Observing the graphs in Figure 2, it is possible to see that even in the smallest dataset of 5k samples, the best models (Nilsimsa, ssdeep, and TLSH) are already capable of yielding an accuracy of more than 90%. SDHASH, instead, requires a bigger dataset (50k samples and above) in order to produce stable results. In general, there is an expected trend of increased performance as the sample size grows, although with diminishing returns staring from a size of 500k samples.

4.1 Comparison to alternative approaches

In Table 4 we present a comparison between our models and other approaches that utilize static analysis. The performance values for the competing approaches are taken from the corresponding research papers.

In comparison to Zozzle, i.e., the best performing compared model we compare to, our model is quite close in performance but does not match it. However, our approach does support the classification of obfuscated JavaScript, which is not supported by Zozzle. This implies a wider range of applicability for our models. When comparing to the other models from the state of the art, our approach performs better when considering the accuracy (about 98%) and is more balanced when considering the FPR and the FNR jointly (e.g., with a threshold of about 3% for both).

In addition to this, due to the very large size of our dataset, we can reliably test the validity of our models and have confidence that a similar performance can be achieved when used in real life circumstances. Making classifiers with small datasets might lead to less generic models. Since there exists a vast diversity

Table 5. Top 10 most common false negative categories, ordered by percentage of occurrences.

	TLSH	Nilsimsa	ssdeep	SDHASH
1st	Unknown	Unknown	Unknown	Unknown
2nd	Redirect	Redirect	Redirect	Redirect
3rd	Trojan	Trojan	Trojan	Trojan
4th	CoinHive	CoinHive	CoinHive	CoinHive
5th	SEOHide	SEOHide	SEOHide	SEOHide
6th	IFrame	IFrame	IFrame	IFrame
7th	Faceliker	Faceliker	Faceliker	Faceliker
8th	Crypted	FakejQuery	FakejQuery	Crypted
9th	FakejQuery	Crypted	Ramnit	FakejQuery
10th	Ramnit	Ramnit	Crypted	Ramnit

of possible malware and clean files, a small dataset might give a skewed image of the performances of the methods, due to not being able to verify whether it works on new never-seen-before malware. In our case, we have 1.5M samples with a 54% positive rate. The only competitor that has a similarly sized dataset is Cujo, with roughly 201k samples, but very few samples of malware (0.3%). This can be further seen in Table 4, as the best performing classifiers all have very few samples of malware files compared to our dataset.

5 Discussion

In this section we discuss the possible causes of misclassifications, which might lead to false negatives and false positives.

5.1 False negatives

False negatives are misclassified malware scripts, for which we have full access to the code. This section will focus on the models trained on the entire 1.5M dataset, as these are the best performing models and also the dataset that contains all malware files, giving a better view of the shortcomings of LSH. Table 5 shows the top 10 most common types of misclassified malware, for each LSH method. The detection names come from Cyren's labelling system. The most occurring category represents the most difficult class of malware for our models to generalise. The *unknown* category contains files that got flagged for malicious behaviour but where there was not enough information to sort the files into one of the more known malware families. One very likely scenario is that the *unknown* malware belong to smaller groups of malware types which might be less prevalent in the dataset.

Observing Table 5, it is possible to see that all LSH methods lead to almost the same false types of negatives: the top 7 misclassified categories are the same for all four methods. When inspecting these files, it can be seen that they have

two common elements: either they are very similar to clean looking code, like in the case of Redirectors and FakejQuery, or the actual malicious part of the code is very small, making it easy to inject into otherwise clean code, like CoinHive.

In consequence, when these malware types are hashed, malicious information might get lost, e.g. if there is a single line of malicious code in an otherwise clean file it might result in that the hash looks more like a clean file rather than a malicious file, which is often the case with CoinHive or other cryptocurrency mining malware.

In the case of redirectors, we have malware that is not necessarily doing anything malicious, as redirecting users on websites is a very common thing, but it is the destination that is malicious. This is a similar problem with malware of the downloader type (in Table 5 are Trojan, Ramnit, FakejQuery, and Crypted) since the act of downloading is not malicious, but the file that is downloaded might be malicious. In that case, the JavaScript file itself does not actually hold any malicious code. In these cases, the destination/download URLs that are the malicious indicators might after locality sensitive hashing have a little to no difference from URLs that are benign. In other words when a locality sensitive hash is created, it might end up looking like other downloader/redirection programs that are benign.

5.2 False positives

False positives are misclassified clean files. The clean files from Cyren were not directly available to us (only the file's SHA256 signature and its LSH hashes are stored in our dataset) as these files are more likely to contain personally identifiable information. Thus we have to rely on analysing the files that come from VirusTotal, which are publicly available. In comparison to false negatives, false positives are much harder to analyse because clean files do not carry a category label that we could use as a basis for generalization. By doing a manual inspection on 50 of the publicly available false positives, the following observations have been made:

- Due to malware like FakejQuery, there is a chance that other similar benign code gets detected, e.g., code that is a fork of jQuery or a jQuery plugin.
- Shorter files give hashes that carry less information, leading to higher false positives. This is the reason why the LSH methods have a recommended minimum file/length.
- Some obfuscation techniques are less common than others, for example, encoding JavaScript statements as a string which the program then interprets using the `eval` method is highly suspicious but is not always an indicator of maliciousness. It is sometimes used to hide sensitive data that should not be able to get scraped by web crawlers.

6 Conclusion

In this paper we have shown that utilising deep learning together with locality sensitive hashing as a form of feature extraction it is possible to classify

JavaScript malware with a high accuracy and a low false positive rate. Our method works with obfuscated code and is completely static. When comparing our method to other methods of static JavaScript malware detection, our method provides competitive results without having drawbacks such as not being able to handle obfuscated code.

References

1. Cyren - Malware Attack Detection (2019), https://www.cyren.com/tl_files/downloads/resources/Cyren_Malware-Attack-Detection_Datasheet_20160915_ltr_EN_web.pdf
2. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning **20**(3) (1995). https://doi.org/10.1007/BF00994018
3. Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: Zozzle: Low-overhead mostly static javascript malware detection. Tech. Rep. MSR-TR-2010-156 (November 2010)
4. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: An open digest-based technique for spam detection. vol. 2004 (2004)
5. Freund, Y., Mason, L.: The alternating decision tree learning algorithm. In: Proceedings of the Sixteenth International Conference on Machine Learning (1999)
6. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. CoRR **abs/1502.03167** (2015), http://arxiv.org/abs/1502.03167
7. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization (2014)
8. Kornblum, J.: VirusTotal (2018), https://ssdeep-project.github.io/ssdeep/index.html
9. Likarish, P., Jung, E., Jo, I.: Obfuscated malicious javascript detection using classification techniques. In: International Conference on Malicious and Unwanted Software (MALWARE) (2009)
10. Micro, T.: TLSH (2018), https://github.com/trendmicro/tlsh
11. Ndichu, S., Ozawa, S., Misu, T., Okada, K.: A machine learning approach to malicious javascript detection using fixed length vector representation. pp. 1–8 (07 2018). https://doi.org/10.1109/IJCNN.2018.8489414
12. Raff, E., Barker, J., Sylvester, J., Brand on, R., Catanzaro, B., Nicholas, C.: Malware Detection by Eating a Whole EXE. arXiv e-prints arXiv:1710.09435 (2017)
13. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: A defense against heap-spraying code injection attacks. In: Proceedings of the Usenix Security Symposium (2009), https://www.microsoft.com/en-us/research/publication/nozzle-a-defense-against-heap-spraying-code-injection-attacks-2/
14. Rieck, K., Krueger, T., Dewald, A.: Cujo: Efficient detection and prevention of drive-by-download attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference (2010). https://doi.org/10.1145/1920261.1920267
15. sdhash@roussev.net: SDHash (2018), http://roussev.net/sdhash/sdhash.html
16. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research (2014), http://jmlr.org/papers/v15/srivastava14a.html
17. VirusTotal: VirusTotal (2018), https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works

18. Wang, Y., Cai, W.d., Wei, P.c.: A deep learning approach for detecting malicious javascript code. Security and Communication Networks (2016). https://doi.org/10.1002/sec.1441
19. Xu, W., Zhang, F., Zhu, S.: Jstill: Mostly static detection of obfuscated malicious javascript code. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy. CODASPY '13 (2013). https://doi.org/10.1145/2435349.2435364
20. Ye, Y., Li, T., Adjeroh, D., Iyengar, S.S.: A survey on malware detection using data mining techniques. ACM Comput. Surv. (2017). https://doi.org/10.1145/3073559