



A Machine Learning Approach to Web-based Cyber-Attack Detection

Wangari, Samuel Ndichu

(Degree)

博士 (学術)

(Date of Degree)

2021-09-25

(Date of Publication)

2022-09-01

(Resource Type)

doctoral thesis

(Report Number)

甲第8170号

(URL)

<https://hdl.handle.net/20.500.14094/D1008170>

※ 当コンテンツは神戸大学の学術成果です。無断複製・不正使用等を禁じます。著作権法で認められている範囲内で、適切にご利用ください。



Doctoral Dissertation

A Machine Learning Approach to
Web-based Cyber-Attack Detection
(ウェブ媒介型サイバー攻撃検知への
機械学習アプローチ)

July, 2021

Graduate School of Engineering,

Kobe University

WANGARI SAMUEL NDICHU

A Machine Learning Approach to Web-based Cyber-Attack Detection

(ウェブ媒介型サイバー攻撃検知への機械学習アプローチ)

WANGARI SAMUEL NDICHU

Abstract

Organizations and individuals widely establish websites and web applications as an interaction platform for services and information. These technologies have consequently been adopted across various devices and browsers. Therefore, because of their innumerable advantages and convenience, websites attract millions of visitors. This massive traffic provides exciting targets for cyber attackers. Most of these websites use JavaScript (JS) to create dynamic content. JS is a dynamic and lightweight scripting language that adds more functionality and enhances user experience. It allows a client-side script to interact with the user. However, JS code has been used to perform attacks. In general, malicious JS codes are not easy to detect because they sneakily exploit browser and plugin software vulnerabilities and attack website visitors unknowingly.

Servers and hosting services have different security layers. A minor vulnerability may be enough for a hacker to take control of a website. For example, Apache and Microsoft internet information services run a significant percentage of websites. Both have known vulnerabilities that hackers can exploit. In addition, the dynamic content in websites using JS can be floodgates to malicious JS-based attacks, for example, WordPress plugins that contain known vulnerabilities. Also, most websites are no longer designed from scratch. Developers use content management systems for publishing web content such as Joomla, WordPress, and Drupal that offer easy-

to-use platforms. There are also popular drag-and-drop website builders such as Weebly and Wix. Using these website development platforms makes it easy for anyone without in-depth technical knowledge to create a website. It also means that the website developers using these platforms overlook security most of the time. Websites created by developers with in-depth technical knowledge rely on different tools and packages to create a dynamic experience: Advertisements, animation, and other parts of a website use third-party systems. The website itself may have robust protection against malware, but the third-party tools may not. Attackers also use malicious email campaigns with attached JS files. Users are required to undergo user awareness training regularly and therefore may not click *.exe* email attachments. However, many users may not know what a JS file is and that it can be harmful. A JS file icon may look like a document, therefore confusing users unfamiliar with this file type.

The exploitation of vulnerabilities in servers, plugins, and other third-party systems enables the insertion of malicious codes into websites. These exploits use methods such as drive-by-downloads, pop-up ads, and phishing attacks on news, porn, piracy, torrent, or free software websites, among others. Many of the recent cyber-attacks exploit JS vulnerabilities, in some cases employing obfuscation to hide their maliciousness and evade detection. JS code obfuscation has advantages such as proprietary code protection, curbing reverse engineering, performance optimization, and code compression. However, JS-based attacks frequently use obfuscation to camouflage their malicious intentions, preserve the overall code behavior, and evade detection by security appliances. Therefore, this use makes obfuscation rampant in both benign and malicious JS codes. Obfuscation generates a JS code that is obscure to the human eyes and undetectable to scanners. This property hinders comprehension and analysis, significantly affecting the performance of network and information security tools, such as Intrusion Detection System (IDS) and anti-virus software.

Attacks using Uniform Resource Locators (URLs) and their JS content to per-

petrate malicious activities on the Internet are becoming rampant. Methods such as blacklisting, client honeypots, domain reputation inspection, and heuristic and signature-based systems are used to detect these malicious activities. Recently, machine learning approaches have been proposed to perform tasks such as document retrieval, sentiment analysis, spam filtering, and web search. Machine learning models learn from training data and classify test and validation data in a very short time with a high degree of accuracy. Challenges, however, still exist. First, blacklist systems are easily evaded by new or variant URLs and JS content, obfuscation, fast-flux, cloaking, and URL shortening. Second, heuristic and signature-based systems do not generalize well to zero-day attacks. Third, the domain name system (DNS) gives cybercriminals the flexibility to easily migrate their malicious servers to hide the Internet Protocol (IP) address behind domain names. Finally, crafting fully representative features for model training is a daunting task, even for domain experts. It is, therefore, primal to develop an accurate detection system for malicious JS to protect users from web-based attacks, detect JS codes that masquerade as innocuous scripts, and identify the manner and degree in which each feature contributes to the prediction. This study, therefore, develops a machine learning approach detection system for malicious and obfuscated JS and URL JS content.

This dissertation contains five chapters. In Chapter 1, the common JS vulnerabilities and JS-based attacks are introduced alongside the overview of this dissertation. In Chapter 2, Abstract Syntax Tree (AST) for code structure representation is adopted, and a machine learning approach to conduct feature learning called Doc2vec for JS-based attacks detection. The AST is obtained from the JS code using Esprima, ECMAScript parsing infrastructure for multipurpose analysis. Esprima can be used to perform lexical and syntactic analysis of a JS program. Doc2vec is a neural network model that can learn context information of texts with variable length. This model is a well-suited feature learning method for JS codes, consisting of text content ranging among single line sentences, paragraphs, and full-length documents. Besides, features learned with Doc2Vec are of low dimensions,

which ensure faster detections. AST-level manipulation and AST-level merging are conducted to enhance the feature learning process for Doc2Vec. AST-level manipulation involves randomly changing AST subtree locations, and AST-level merging involves extracting JS-AST body to create AST combinations. A classifier model judges the maliciousness of a JS code using the learned features. The performance of this approach is evaluated using the D3M dataset (Drive-by-Download Data by Marionette) for malicious JS codes and the JSUNPACK plus Alexa top 100 websites datasets for benign JS codes. The performance of Doc2Vec on plain JS codes (Plain-JS) and AST form of JS codes (AST-JS) is then compared to other feature learning methods. The experimental results show that the proposed AST features and Doc2Vec for feature learning provide better accuracy and fast classification in malicious JS code detection compared to conventional approaches and can flag malicious JS codes previously identified as hard-to-detect.

In Chapter 3, Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) of obfuscated malicious JS for machine learning models detection performance improvement is proposed. The existing deobfuscation methods assume that a specific tool can recover an original JS code entirely. For a multi-layer obfuscation, general deobfuscation tools such as Dirty Markup, Online JS Code Beautifier, and Dan’s Tools JS Code Formatter realize a formatted JS code through JS code beautification or simplification. This process involves taking an obfuscated code, analyzing it, and making it readable again to uncover its true functionality. However, some sections remain encoded. For the detection of such codes, this study performs DUD-preprocessing by function redefinition using a Virtual Machine (VM), a JS code editor, and a python `int_to_str()` function to facilitate feature learning by the FastText model. The learned feature vectors are passed to a classifier model that judges the maliciousness of a JS code. In performance evaluation, this study uses Hynek Petrak’s dataset for obfuscated malicious JS codes and SRILAB dataset, and the Majestic Million service top 10,000 websites for obfuscated benign JS codes. The performance of the FastText model is compared to the Paragraph Vector models,

the StarSpace model, the Long Short-Term Memory (LSTM) model, and the Term Frequency–Inverse Document Frequency (TF-IDF) model on the detection of DUD-preprocessed obfuscated malicious JS codes. The experimental results show that the proposed approach enhances feature learning and provides improved accuracy in the detection of obfuscated malicious JS codes.

In Chapter 4, this study proposes a feature selection and classification approach for malicious JS content using SHapley Additive exPlanations (SHAP) and tree ensemble methods. SHAP is a game-theoretic approach that explains any machine learning model’s output and connects optimal credit allocation with local explanations using the classical Shapley values and their related extensions. The SHAP values show the extent to which a feature is responsible for a change in the model output. One of the premises is that the ability to interpret and identify the most impactful features may enable the derivation of other features with additional information to improve the detection performance for JS-based attacks. SHAP values may also provide necessary insights into the distribution of specific benign and malicious JS features. While other methods learn a set of manually extracted or engineered features, the SHAP-based approach adopts features that are automatically selected based on their contributions to the model output. Such features are obtained from the distribution of continuously evolving malicious JS content, ensuring that the global and local features that contribute to the model performance are captured. AST-JS also represents adequate details of a JS code structure without executing the code. Other studies limit their scope by studying specific malicious URLs and JS attacks, such as those containing phishing attacks, social engineering, malvertisement attacks, and Denial-of-service (DoS) attacks, among others. For this, the studies use manually engineered features unique to the type of attack they intended to identify. Such approaches have challenges in that they cannot be generalized to other attack types. Another premise is that features selected using SHAP values can overcome such challenges, as the security analyst can easily and quickly identify which features significantly impact the overall model performance. The JS features

are obtained from the AST-JS, sample JS attack codes, and association rule mining. The malicious and benign JS datasets obtained from Hynek Petrak and the Majestic Million Service are used for performance evaluation. The performance of the developed model is compared to that of other models on the task of malicious JS content detection. The experimental results show that the proposed approach is a better prediction model. Using SHAP interaction values, a security analyst can easily and quickly identify AST-JS feature combinations or interactions that contribute more toward the model’s detection performance.

Finally, in Chapter 5, this study’s conclusions and future work are discussed: JS code structure representation and feature learning, obfuscated malicious JS DUD-preprocessing, and malicious JS content feature selection and classification.

Acknowledgements

First, I want to thank God the almighty for keeping my family and i safe despite the distance.

Second, I am forever grateful to Professor Seiichi Ozawa, my supervisor, for his immense support, guidance, and advice throughout the Doctoral program.

Third, I extend my appreciation to Professor Toshiaki Omori, Professor Sangwook Kim, and my papers co-authors: Takeshi Misu, Kouichirou Okada, Kazuo Makishima, Dr. Tao Ban, Dr. Takeshi Takahashi, and Dr. Daisuke Inoue, for their comments and positive criticism.

Forth, my gratitude goes to the; National Institute of Information and Communications Technology (NICT) and the Web-based Attack Response with Practical and Deployable Research Initiative (WARPDRIVE) for funding my research; Secure Brain Japan for research collaboration; and members of ES5 laboratory for their assistance during my study.

Finally, my sincere gratitude goes to The Ministry of Education, Culture, Sports, Science, and Technology (MEXT) for the scholarship award.

Contents

Abstract	i
Acknowledgements	vii
1 Introduction	1
1.1 Background	1
1.2 JavaScript Attacks and Vulnerabilities	8
1.3 JavaScript Attacks Trends	11
1.4 Previous Studies: Machine learning approaches to detect malicious, obfuscated, and URL JS content	14
1.4.1 Malicious JS codes detection	14
1.4.2 Obfuscated malicious JS codes detection	16
1.4.3 URL and URL JS content detection	21
1.5 Detection of JavaScript-based Attacks	23
1.6 Deobfuscation, Unpacking, and Decoding of Obfuscated Malicious JavaScript	24
1.7 Web-based Attack Detection by Feature Selection	25
1.8 Performance Evaluations	27
1.9 Thesis Outlines and Research Objectives	28
2 A Machine Learning Approach to Detection of JavaScript-based Attacks using AST Features and Paragraph Vectors	30
2.1 Introduction	30
2.2 Proposed method	31

2.2.1	AST form of JavaScript code	31
2.2.2	Preprocessing	33
2.2.3	Feature learning by Doc2Vec	35
2.3	Experiments	44
2.3.1	Experimental setup	44
2.3.2	Performance Comparisons	46
2.4	Conclusions	48
3	Deobfuscation, Unpacking, and Decoding of Obfuscated Malicious JavaScript for Machine Learning Models Detection Performance Improvement	52
3.1	Introduction	52
3.2	Proposed method	53
3.2.1	Preprocessing	53
3.2.2	Feature learning by the FastText model	56
3.3	Experiments	60
3.3.1	Experimental setup	60
3.3.2	Performance Comparisons	62
3.4	Conclusion	67
4	Web-based Attack Detection by Feature Selection using Tree SHapley Additive exPlanations	71
4.1	Introduction	71
4.2	Proposed method	72
4.2.1	Preprocessing	72
4.2.2	SHapley Additive exPlanations (SHAP) Feature Importance	77
4.3	Experiments	80
4.3.1	Experimental Setup	80
4.3.2	Performance Comparisons	81
4.4	Conclusions	91

5	Conclusion	94
5.1	Contributions of This Dissertation	98
5.2	Future Work	99
	Bibliography	117
A	List of Publications	118
A.1	Journal Papers	118
A.2	International Conferences	118
A.3	Domestic Conferences	119

List of Figures

1.1	Web-based attack trends.	12
1.2	Web-based attack vectors.	12
2.1	A sample of a plain JS code (plain-JS) after preprocessing using a regular expression. This representation shows the resultant Plain-JS after the removal of symbols and special characters. It is used as Doc2Vec input for the first part of the experiments.	34
2.2	Distributed Bag of Words version of Paragraph Vector (PV-DBoW). Words are randomly sampled from a text window, and a trained paragraph vector is used to predict the words.	39
2.3	Distributed Memory Model of Paragraph Vectors (PV-DM). Word vectors and paragraph vectors are concatenated to create a feature vector for prediction using a classifier.	40
2.4	The layout for learning and classification of malicious JS contents AST features.	40
2.5	ROC curves and AUC representing performance of PV-DBoW and PV-DM on malicious JS codes detection with respect to false positive and true positive rates. The dashed diagonal line represents performance threshold as random guessing. The colored curves represent the performance of the specific model for each fold during cross-validation. Also included is the mean AUC score for the individual models.	49

2.6	ROC curves and AUC scores representing performance of other models on malicious JS codes detection with respect to false positive and true positive rates. The dashed diagonal line represents performance threshold as random guessing. The colored curves represent the performance of the specific model for each fold during cross-validation. Also included is the mean AUC score for the individual models	50
3.1	The layout for Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) of a JS code and the feature learning and classification for DUD-preprocessed AST-JS features.	56
3.2	The FastText model architecture which shows the n -gram features, hidden variable and the output.	57
3.3	The layout for Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) of a JS code and the feature learning and classification for DUD-preprocessed AST-JS features.	60
3.4	ROC curves and AUC that represent the performance of the FastText, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model and the TF-IDF model on obfuscated malicious JS codes detection with the true positive rate plotted against the false positive rate. The dashed diagonal line represents the performance threshold as random guessing. The colored curves represent the performance of the each model for each fold during cross-validation. Also included is the mean AUC score for each model.	68

4.1	SHAP summary plots showing the AST-JS features, before (a), and after (b) feature selection. Each feature is contributing to pushing the model output over the JS dataset from the basevalue or average predicted probability of -0.38 (a) and -0.42 (b) towards $f(x)$ of 10.26 (a) and 15.32 (b). Both JS codes have a prediction value greater than the base value. Therefore, the JS codes are likely malicious. SHAP values for 3,000 JS code samples before (c) and after (d) feature selection are also included.	84
4.2	SHAP summary plot before feature selection. SHAP values are plotted on the x-axis for each AST-JS feature on a row and sorted in descending order of their importance. Vertically piled-up points represent the AST-JS feature density. The colors show the AST-JS feature value. The primary risk indicator of maliciousness is shown as the <i>ObjectExpression</i> feature. The next most potent indicator of maliciousness is the <i>FunctionExpression</i> feature.	86
4.3	SHAP summary plot for feature selection. The SHAP values are plotted on the x-axis for each AST-JS feature on a row and sorted in descending order of their importance. The vertically piled points represent the AST-JS feature density. The colors show the AST-JS feature values. The primary risk indicator of maliciousness is the <i>ObjectExpression</i> feature. The next most potent indicator of maliciousness is the <i>ExpressionStatement_CallExpression_Identifier</i>	87
4.4	SHAP dependence plot for the top AST-JS feature before (a) and after (b) feature selection. The <i>TryStatement</i> and <i>VariableDeclaration</i> features were automatically selected for coloring based on a potential interaction in the model. In plot (b), it can be seen that low SHAP values of the <i>ObjectExpression</i> feature influence the model output more significantly for observations where the <i>VariableDeclaration</i> feature has high SHAP values.	88

4.5	SHAP interaction values plot showing the interaction effects involving the <i>ObjectExpression</i> feature. These effects capture all vertical dispersions. Plot (a) shows that high SHAP values for the <i>ObjectExpression</i> and <i>FunctionExpression</i> features significantly influence the model output. Plot (b) shows that low SHAP values for the <i>ObjectExpression</i> feature and high SHAP values for the <i>ExpressionStatement_CallExpression_Identifier</i> feature significantly influence the model output.	89
4.6	Feature importance assigned to the AST-JS features by the Boruta, ELI5, RandomForest and SelectKBest feature selection methods. . . .	92
5.1	Example of a hard-to-detect JS code by analysis of false positive and false negative cases in the dataset. These are sample codes for which detection is ineffective while defining input as a plain-JS. These codes also include numbers and special characters in addition to JS code words. Input defined as plain-JS does not adequately capture such JS code details and code structure representation.	95
5.2	SHAP interaction values showing AST-JS features pair combinations. The plot makes it easier to determine which pair contributes more to the overall model performance and in which direction.	97

List of Tables

2.1	A list of frequently used function names in malicious JS codes, function types, possible threats, and intention. This list provides evidence that malicious and benign JS codes contain different frequent words as malicious JS codes would be expected to utilize such function names compared to benign ones heavily.	31
2a	Precision, recall, and F1-score obtained by feature learning using a plain-JS and p value representing significance difference in performance for paragraph vector models compared to other approaches. LDA stands for Latent Dirichlet Allocation, LSA stands for Latent Semantic Analysis, and TF-IDF stands for term frequency-inverse document frequency.	47
2b	Precision, Recall and F1-score obtained by feature learning on AST-JS and p value representing significance difference in performance for paragraph vector models compared to other approaches.	47
2.3	Computational time, representing the training and detection time per JS code obtained by dividing the total computation time with the number of JS code samples. It includes the time for different models.	49
3.1	JS code label, original JS codes and deobfuscated JS codes showing the success rate of the deobfuscation process.	61

3.2 Precision, Recall, and F1-score representing the performance of the FastText model, the Paragraph Vector models, the StarSpace model, the LSTM model, and the TF-IDF model on a regular obfuscated JS code dataset. The PV-DBOW model stands for the Distributed Bag of Words version of Paragraph Vector. The PV-DM model stands for the Distributed Memory Model of Paragraph Vectors. The LSTM model stands for the Long Short-Term Memory model. The TF-IDF model stands for Term Frequency–Inverse Document Frequency model. 63

3.3 Precision, recall, and F1-score obtained by feature learning using a deobfuscated JS code dataset. The PV-DBOW model stands for the Distributed Bag of Words version of Paragraph Vector, the PV-DM model stands for the Distributed Memory Model of Paragraph Vectors, the LSTM model stands for the Long Short-Term Memory model and the TF-IDF model stands for Term Frequency–Inverse Document Frequency model. 65

3.4 Precision, Recall, and F1-score obtained by feature learning using an unpacked JS code dataset. The PV-DBOW model stands for the Distributed Bag of Words version of Paragraph Vector, the PV-DM model stands for the Distributed Memory Model of Paragraph Vectors, the LSTM model stands for the Long Short-Term Memory model and the TF-IDF model stands for Term Frequency–Inverse Document Frequency model. 65

3.5	Precision, Recall and F1-score obtained by feature learning using a deobfuscated, unpacked and decoded (DUD-preprocessed) JS code dataset. The PV-DBOW model stands for the Distributed Bag of Words version of Paragraph Vector, the PV-DM model stands for the Distributed Memory Model of Paragraph Vectors, the LSTM model stands for the Long Short-Term Memory model and the TF-IDF model stands for Term Frequency–Inverse Document Frequency model.	66
3.6	The gross and net performance for the FastText model while considering the success rate of deobfuscation.	69
4.1	The dataset statistics showing the number of malicious and benign JS codes before and after preprocessing.	81
4.2	The AST-JS nodes and node combination features selected using SHapley Additive ex-Planations (SHAP).	82
4.3	XGBClassifier and LGBMClassifier parameters obtained using <i>GridSearchCV</i> for the different input types.	83
4.4	Precision, recall and F1-scores obtained by the malicious JS detection model using features defined as AST-JS nodes and 10-fold cross validation.	83
4.5	Precision, recall and F1-scores obtained by the malicious JS detection model using features defined by association rule mining and 10-fold cross validation.	89
4.6	Precision, recall and F1-scores obtained by the malicious JS detection model using features selected by SHAP value and 10-fold cross validation.	90
4.7	Precision, recall and F1-scores obtained by the malicious JS detection model using features selected by SHAP value and 10-fold cross validation. The performance is compared to the Boruta, ELI5, RandomForest, and SelectKBest feature selection methods.	91

5.1	Precision, Recall and F1-score representing the performance of para-	
	graph vector models on obfuscated JS codes.	96

Chapter 1

Introduction

1.1 Background

Websites and web applications exist widely because of the myriad services and convenience they offer. Adoption of these technologies across various devices and browsers is consequently inevitable. Likewise, because of their innumerable advantages and convenience, websites attract millions of visitors. However, even with their practical applications, the rapid development of these technologies coupled with the availability of tools and technologies to exploit vulnerabilities in servers, plugins, and other third-party systems has made them interesting targets for cyber-criminals. These exploits result in the proliferation of various types of attacks such as drive-by-downloads, pop-up ads, and phishing attacks common with cyber-criminals. Attackers insert malicious codes into news, porn, piracy, torrent, or free software websites to perform these attacks for financial gains, social engineering, or extortion [60].

Most websites use JavaScript (JS) to create dynamic content. JS is a dynamic and lightweight scripting language that adds more functionality and enhances user experience. It allows a client-side script to interact with the user. However, JS code has been used to perform attacks. In general, malicious JS codes are not easy to detect [115] because they sneakily exploit vulnerabilities of browsers, plugins, and other third-party applications and attack visitors of a website unknowingly. Therefore, it is important to accurately detect malicious JS codes to protect users

from such attacks proactively. When browsing a website, JS files are downloaded on the client-side and executed via a browser. There are differences in possible damage and method for these attacks. However, these attacks generally use JS. For example, a Cross-site scripting (XSS) vulnerability facilitating a Drive-by-Download attack when a user opens a Uniform Resource Locator (URL). Malicious JS codes with various attack types will have the same code properties at an abstract level. These codes are, therefore, detectable with the proposed method.

Servers and hosting services have different security layers. A minor vulnerability may be enough for a hacker to take control of a website. For example, Apache and Microsoft internet information services run a significant percentage of websites. Both have known vulnerabilities that hackers can exploit. In addition, the dynamic content in websites using JS can be floodgates to malicious JS-based attacks, for example, WordPress plugins that contain known vulnerabilities. Also, most websites are no longer designed from scratch. Developers use Content Management Systems (CMS) platforms for publishing web content such as Joomla, WordPress, and Drupal that offer easy-to-use platforms. There are also popular drag-and-drop website builders such as Weebly and Wix. Using these website development platforms makes it easy for anyone without in-depth technical knowledge to create a website. It also means that the website developers using these platforms overlook security most of the time. Websites created by developers with in-depth technical knowledge rely on different tools and packages to create a dynamic experience: Advertisements, animation, and other parts of a website use third-party systems. The website itself may have robust protection against malware, but the third-party tools may not. Attackers also use malicious email campaigns with attached JS files. Users are required to undergo user awareness training regularly and therefore may not click *.exe* email attachments. However, many users may not know what a JS file is and that it can be harmful. A JS file icon may look like a document, therefore confusing users unfamiliar with this file type.

Malicious JS codes perform, among others, keystroke recording, browser cookies

theft, hacking, website defacement, and Trojan horses [43]. Besides, it is possible to create a botnet by tricking users into downloading malware by social engineering [21]. Conventional security solutions such as anti-virus and Intrusion Detection Systems (IDS) employ signature and heuristic-based approaches to detect attacks. Heuristic-based methods such as file analysis, file emulation, and generic signature detection [60] use a set of expert decision rules. These can detect previously unknown and variants of malware. However, these approaches have many shortcomings. First, knowledge of an emerging attack is essential to update the program for effective detection, which creates a vulnerability window that attackers can leverage to launch zero-day and other types of attacks. Second, they suffer from a slow detection rate due to lengthy analysis and scanning time. Lastly, they are prone to suffer from large numbers of false negatives, which is a drawback to performance. Other conventional approaches for detecting malicious JS are the use of pattern matching [18], low and high interaction client honeypots, and maintenance of a URL blacklist [7, 61]. Recently, many intelligent intrusion detection systems have been proposed, which mainly focus on anomaly detection.

Recently, application fields of machine learning have been extending enormously, including applications such as text classification and clustering [66, 85, 83, 24, 14, 118, 135, 27, 52] where a machine learning model carries out tasks such as document retrieval, sentiment analysis, spam filtering, and web search. Text documents are used to train a machine learning model, which in turn is used to predict document classes or categories. A machine learning model learns from data and can accurately predict a given input after correctly conducting learning. Usually, raw data need to be transformed into a proper feature representation for inputs to attain high-performance prediction. One such representation is a fixed-length vector for text documents. These models have achieved state-of-the-art results for text classification and clustering tasks on various datasets. As such, since a JS code also contains textual data, it is envisaged that machine learning models can classify JS-based attacks. Therefore, in this dissertation, a machine learning approach to malicious

JS content detection based on AST form of JS code (AST-JS) and Paragraph Vectors is introduced.

Many of the recent cyber-attacks exploit JS vulnerabilities, in some cases employing obfuscation to hide their maliciousness and evade detection. JS obfuscation is a transformation aimed at generating a JS code that is obscure to the human eyes and undetectable to scanners. This property hinders comprehension and analysis of a JS code, significantly affecting the performance of network and information security tools. Attackers commonly use obfuscation to circumvent detection and identification by network and information security tools such as IDS and anti-virus software. Obfuscation in software and programs, such as JS codes and HyperText Markup Language (HTML), has advantages such as proprietary code protection, curbing reverse engineering, performance optimization, and code compression [116, 114]. However, JS-based attacks frequently use obfuscation to camouflage their malicious intentions, preserve the overall code behavior, and evade detection by security appliances. Therefore, this use makes obfuscation rampant in both benign and malicious JS codes. Therefore, it would be wrong to naively render a JS code as malicious if it contains some form of obfuscation [22, 48, 58].

The use of data and encoding-based obfuscation techniques can reduce the detection rate for signature and heuristics-based tools by approximately 40% and 100%, respectively [137]. Other languages, such as HTML, have a finite number of methods of obfuscation [48, 72] for example, an HTML element obfuscation by transforming an *iframe* to perform evasion attacks. On the other hand, JS code is dynamic. Hence, it is resilient to a variety of obfuscation techniques that are widely used in JS codes, such as variable and function names randomization, string splitting and concatenation, keyword substitution, dead code insertion, packing, and encoding, such as Base64 [97, 117]. It is also possible to implement customized encoding to obfuscate a JS code [137] where a special function is inserted and used to decode the JS code at runtime. Listing 1.1 is the encoding function where each character's Unicode is increased by one. Listing 1.2 is the resultant obfuscated JS code which

is decoded before execution [137]. The function "document.write('Hello world!')" in the encoding function is encoded into "epdvnfou/xsjuf)(Ifmmp!xpsme(*"; in the resultant obfuscated JS code.

```
1 function encode(mystring)
2 {
3   var c="";
4   var i =0;
5   for(var i=0;i
6     <mystring.length;i++){
7     c = c +
8     String.fromCharCode(mystring.charCodeAt
9       At(i)+1);
10  }
11  return c
12 }
13
14 var c =
15 encode("document.write('Hello world!')");
16 document.write(c);
```

Listing 1.1: The encoding function.

```
1 function decode(c)
2 {
3   var mystring="";
4   var i =0;
5   for(var i=0;i <c.length;i++){
6     mystring = mystring +
7     String.fromCharCode(c.charCodeAt(i)-1);
8   }
9   return mystring;
10 }
11
12 var c ="epdvnfou/xsjuf)(Ifmmp!xpsme(*";
13 var mystring = decode(c);
14
```

```
15 eval(decode(c));
```

Listing 1.2: The obfuscated JS code.

Therefore, it is vital to accurately detect malicious JS codes that use obfuscation to masquerade as innocuous scripts. There exist several methods to analyze such obfuscated JS codes. These approaches perform either static or dynamic analysis of obfuscated JS codes. Some of the JS code deobfuscation approaches assume that it is possible to entirely recover an original JS code using a specific deobfuscation tool. Such a strategy to analyze obfuscated malicious JS codes is applied using general deobfuscation tools such as Dirty Markup [28], Online JS Code Beautifier [69], and Dan’s Tools JS Code Formatter [26]. Applying the deobfuscation tools helps realize a readable and pretty JS code that is easier to edit and analyze. However, as part of the findings of this study, there still exists some sections of a JS code that proves hard to deobfuscate, for example, a JS code with multiple layers of obfuscation.

In Chapter 2, feature learning for regular JS code is conducted using Plain JS codes (plain-JS) features, Abstract Syntax Tree (AST) features, and Paragraph Vectors. AST-JS gives a robust property against some perturbation in JS codes. Using AST-JS, detection of obfuscated JS code with obfuscation techniques that do not affect the original semantics of JS code content would be possible. However, it is not appropriate to represent a JS code that contains packing and or encoding obfuscation techniques that convert JS code content into ASCII, Unicode, hexadecimal, and encrypted values. Only about 50% of the obfuscated JS code preserves its original semantics [119]. Therefore, another of this dissertation’s novelty is the development of a system for Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) of obfuscated malicious JS code.

By using obfuscation, cyber-criminals are finding web-based platforms to be perfect tools for launching their attacks. Attackers compromise URLs and their JS content to perform malicious activities on the Internet. Such activities include phishing, URL redirection, spamming, social engineering, botnets, and drive-by-download exploits [80, 11, 10]. The attacks are delivered through emails, malware

advertisements, texts, pop-ups, malicious scripts, and search results. The domain name system (DNS) [80, 11, 10, 37, 82] provides cyber-criminals the flexibility to easily migrate their malicious servers as they hide the IP address behind domain names [11, 10]. Securing websites is vital for maintaining confidentiality, integrity, and availability.

Methods such as maintaining a blacklist [71], client honeypots, domain reputation inspection [37], and domain and web metrics analysis [82, 126, 6, 90] can detect malicious URLs and their JS content [37, 113, 34]. However, new URLs are registered every day, and a blacklist can be evaded through URL obfuscation, fast flux, cloaking, and URL shortening.

Recently, machine learning approaches that employ feature extraction and representation learning for malicious URLs and JS content detection have been proposed [11, 10, 78, 76, 77]. Machine learning algorithms learn a prediction function based on features such as lexical, host-based, URL lifetime, and content-based features that include HTML and JS. A classifier is then used to predict whether given content is malicious or benign using a test dataset. Models such as these can be generalized to new data, unlike the blacklist approach. However, there are challenges in crafting fully representative features for model training. Therefore, in addition to the developed system for detecting malicious and obfuscated JS using fixed-length vectors and AST-JS [94, 92, 91], a system for detection of malicious JS content through feature selection is also developed in this study.

This chapter is organized as follows, in Section 1.2, a brief explanation of JS attacks and vulnerabilities is given. Previous studies on approaches that use machine learning to detect malicious and obfuscated JS codes and methods and techniques for URL and JS content detection are discussed in Section 1.3. The general overview of the proposed methods in Chapters 2, 3, and 4 are discussed in Section 1.4, Section 1.5, and Section 1.6, respectively. Section 1.7 presents the evaluation methods used in this dissertation. Finally, the thesis outlines and objectives are highlighted in Section 1.8.

1.2 JavaScript Attacks and Vulnerabilities

JS is a popular scripting language used extensively to build everything across the web from backend servers to interactive front-facing web applications, which make up 95% of the web [130]. JS is used to implement complex and advanced features on web pages, CMS platforms, and other major applications. It is used as a client-side programming language by 97.2% [130] of all the websites to build dynamic web applications. It is a ubiquitous technology that allows a client-side script to interact with a user and makes websites dynamic and interactive [89], leading to less server-client interaction, immediate feedback, and rich interfaces. It is a robust scripting language executed by the browser.

Despite its various advantages, JS has been used to perform web-based attacks as it presents new and unique vulnerabilities exploited by attackers to gain access or compromise systems. Among other methods, attackers typically inject malicious code into a legitimate website, tricking a browser into executing malware whenever the website is loaded. Attackers can easily modify the functionality of a web page using malicious JS injections. Another attack method is the use of a redirection attack. JS can be used to redirect visitors to third-party websites, stealthily install malware using drive-by-downloads, display unwanted advertisements, or mine cryptocurrencies. Another attack type is form jacking or web skimming attack, where an attacker injects malicious code into a website and extracts user data. An example of form jacking or web skimming attack is the Magecart attack [112], also called supply chain attack, that steals payment card information. Some of these JS-based attacks are explained in details here:

- Drive-by-Download attacks refer to automatically downloading malware on a computer stealthily, without the user's awareness. Malware downloads and automatically installs itself. Infected websites may contain an exploit kit. An exploit kit is a toolkit used to attack vulnerabilities and distribute malware on compromised websites. If any weakness occurs, malware is downloaded to exploit the vulnerability without the user's intention. Listing 1.3 shows

a JS code used to conduct a drive-by-download attack against the Internet Explorer browser [49]. Line 1 shows the use of fingerprinting to attack the Microsoft Internet Explorer browser. Lines 5-7 shows evasion attacks to bypass emulation-based security controls. In line 6, an evasion attack attempts to load a non-existent ActiveX Object named UM0QS4dD. Loading of the ActiveX Object fails when executed within a regular browser. Failing triggers the execution of the exploitation code in the catch block shown in lines 7-14. However, an emulation-based detection system must emulate the ActiveX Application Programming Interface (API). Emulation is done by simulating the loading and presence of any ActiveX Object. Loading of the ActiveX Object will not raise this exception. Consequently, the attack will not be successful as the exploit will not be executed. If the fingerprinting or evasion stage fails, the user is redirected to another page, as shown in line 16.

```
1  If ((navigator.appName.indexOf("MicrosoftInte" + "rnetExplorer"
    ) == 1) && (navigator.userAgent.indexOf("Windows N" + "T
    5.1") == 1) && (navigator.userAgent.indexOf("MSI" + "E8.0")
    == 1)){
2  att = btt + 1;
3  }
4  if (att == 0) {
5  try {
6  new ActiveXObject("UM0QS4dD");
7  } catch (e) {
8  var t1Mo0ul8 = '\x25' + '\u9' + '\x30' + '\x39' + YYGR16;
9  t1Mo0ul8 += t1Mo0ul8;
10 var CBmH8 = "%u";
11 var vBYG0 = unescape;
12 var EuhV2 = "BODY";
13 . . .
14 }
15 }
```

```
16 setTimeout("redir()", 3000);
```

Listing 1.3: The obfuscated JS code.

- Cross-site scripting (XSS) attack is common in web applications without proper user input validation and sanitization, for example, form data and comments. It allows the injection of malicious scripts into a website. Insertion of malicious links into a legitimate website is one of the ways for the attack perpetration. Stealing user-sensitive information such as passwords, cookies, and account details is common when a user visits such a website. Some JS-based attacks inject scripts targeting authenticated website administrators. When executed in the context and with the permission of such privileged users, the malicious script can stealthily create new rogue admin users or loosen website security settings enabling other attacks.
- Cross-Site Request Forgery (XSRF) attack is where a user authenticated to a web application is tricked into performing unwarranted actions. This malicious JS code can exploit cookies, browsers, and security permissions to perform actions on another website.
- Heap spraying attacks constitute writing of a sequence of bytes at a predetermined process memory location which is then exploited to execute arbitrary malicious code.
- Hidden iframes load JS malware from compromised websites, resulting in stealthily loading of exploits.
- Malvertising or malicious advertising takes advantage of advertisements' pervasive and rife nature, which are billions on the Internet. Most of these do not adhere to ethical standards and are illegitimate. For example, Google took down close to two billion advertisements that violated its policies in 2016.
- Structured Query Language (SQL) injection attack consists of inserting a SQL query via the input data from the client to the application. A successful

SQL injection can read sensitive data from the database, perform tasks such as insert, update or delete that modifies a database data, perform privilege operations such shutdown of a database, issuing commands to the operating system, and file recovery operations. SQL injection attacks are a type of injection attack in which SQL commands are injected into data-plane input to facilitate the execution of predefined SQL commands.

1.3 JavaScript Attacks Trends

According to ENISA [30], there were a 52% increase in the number of web-based and web application attacks in 2019 compared to 2018, as shown in Fig. 1.1. Accenture 2019 survey [1] found a 67% increase in security breaches over the last five years with a +13% increase in web-based attacks between 2017 and 2018. Web-based attacks comprise the highest increase according to this survey compared to malware (+11%), Denial of Service (+10%), phishing and social engineering (+8%), malicious code (+9%), stolen devices (+12%), and botnets (+12%). SonicWall [121] and Akamai [2] suggest that XSS, SQLi, directory traversal, broken authentication, and session management are among the top attack vectors used in Web application attacks [30] as shown in Fig. 1.2. The following are the top cyber-attack trends facing websites and web applications:

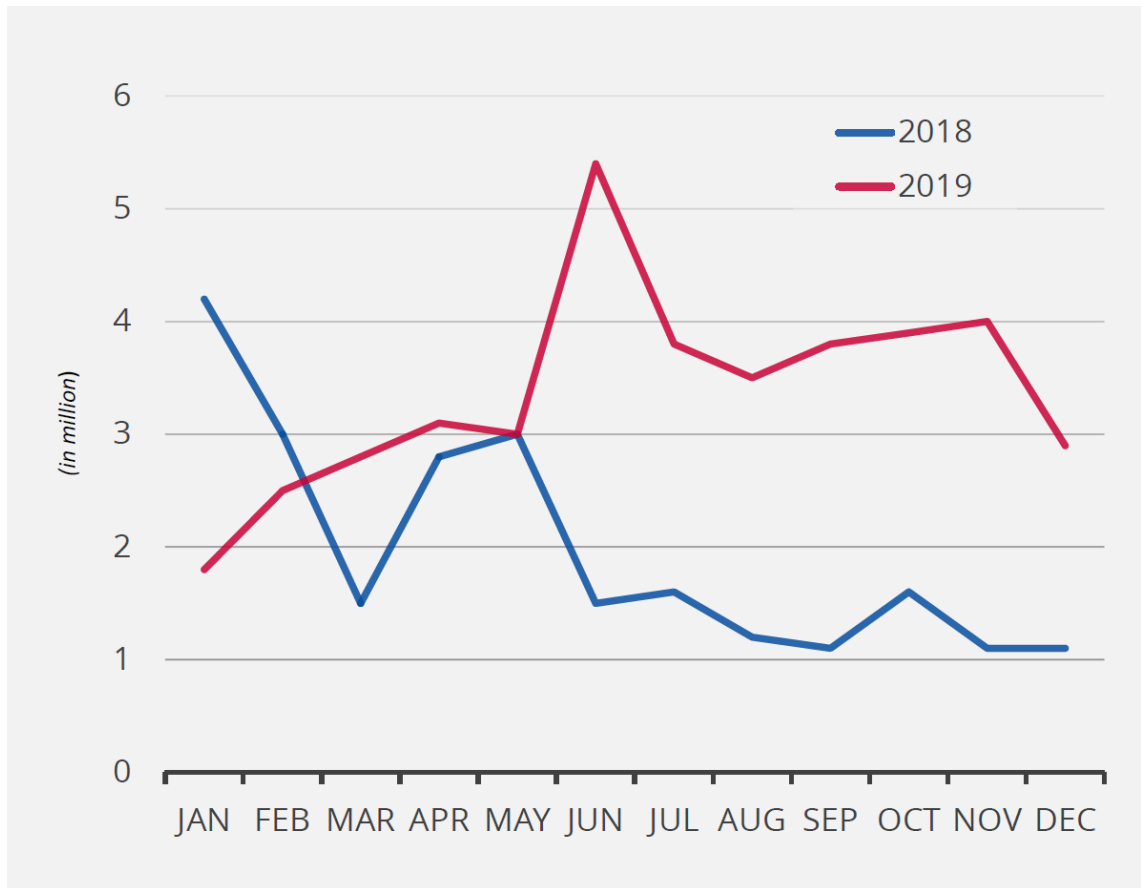


Figure 1.1: Web-based attack trends.

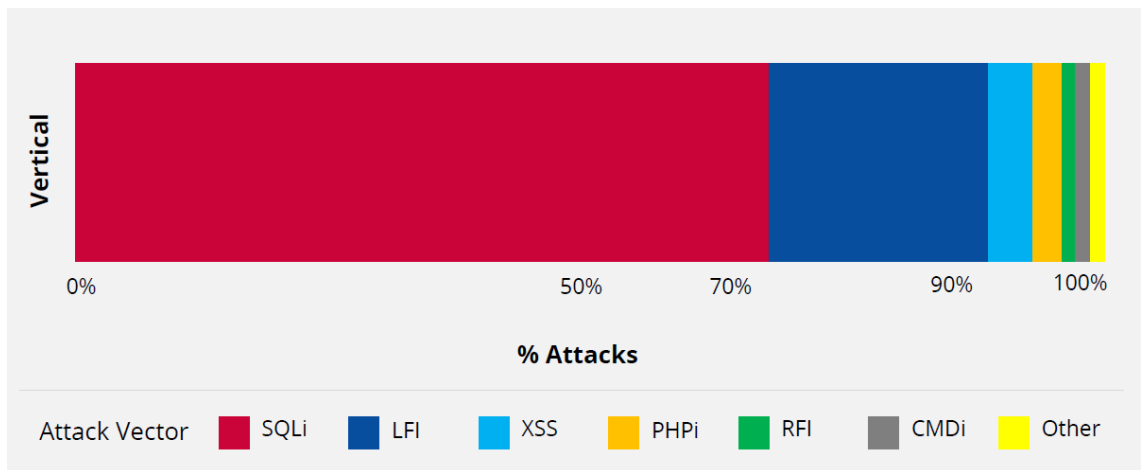


Figure 1.2: Web-based attack vectors.

- JS injection, which includes attacks such as XSS and SQL injection. XSS is one of the common JS injection attacks. 77% of websites are vulnerable to XSS attacks [122]. This vulnerability leads to the widespread use of XSS by attackers to infiltrate organizations. SQL injection is another type of JS

injection attack. 42% of websites are vulnerable to XSS attacks [29].

- Digital skimming or form jacking involves using digital Payment Card Skimming (DPCS) or form jacking for code injection to steal data. For example, attackers infiltrate credit card information and other user details by injecting malicious JS code into payment forms on checkout pages of eCommerce websites.

Payment Card Industry (PCI) compliance prevents customers from storing their three-digit credit card security numbers on a website’s servers. Therefore, attackers focus on the client-side of the website to capture user details as they are entered in real-time. Form jacking made up 71% of web breaches [31] in 2019. The persistence of attacks such as Magecart aggravates these attacks.

- Third-party vulnerabilities are pervasive. The average retail website uses between 40-60 third-party plugins and technologies [50] for personalization, tag management, advertisement, and customer reviews. Attackers exploit third-party plugins to gain access to customers’ personally identifiable information (PII) or payment data from a website. Additionally, 61% percent of companies have experienced a data breach caused by one of their vendors or third parties [16, 100], such as WordPress vulnerabilities. WordPress is used by over 40% of all websites. 98% of WordPress vulnerabilities are due to plugins [100]. Despite fewer new plugins in the ecosystem, WordPress vulnerabilities have increased with a 30% increase from 2017 to 2018 [100]. The most prevalent vulnerability types in WordPress plugins are Cross-site Scripting and SQL Injection.
- Magecart, a notorious attacker group, is synonymous with digital skimming and exploiting third-party vulnerabilities to steal payment data from websites [112]. A series of high-profile web inject attacks hitting the headlines were conducted by the Magecart group in 2019. An estimated 20,000 online stores have been breached by the group, such as Macy’s and gun manufacturer and Smith & Wesson in 2019. In addition, the Magecart group is evolving its

attack methods to avoid detection by conducting evasion attacks by swapping out skimming domains on compromised websites.

- Ad injections attacks inject adware on the client-side. Therefore, the website owner often does not know that users are being presented with unsolicited ads while accessing the website. This attack is typical on most websites. While this attack is annoying to users, it can lead to brand damage, loss of customer loyalty, and revenues to the website owner. Additionally, the attack employs other attack methods such as redirection attacks.

1.4 Previous Studies: Machine learning approaches to detect malicious, obfuscated, and URL JS content

1.4.1 Malicious JS codes detection

Cyber-attacks that employ malicious JS codes are ever-increasing, and conventional security approaches such as signature, heuristic-based approaches, and pattern matching [18] have shortcomings in terms of zero-day attacks, leading to many false negatives coupled with slow detections. It is therefore important to accurately and timely detect these codes to curb this trend. Several other approaches for detecting malicious JS, such as using low and high interaction client honeypots and maintenance of a URL blacklist [7, 61] exist. However, these are prone to suffer from the risk of infiltration, which is a shortcoming for honeypot implementations, and new URLs easily circumvent the blacklist.

Approaches that use machine learning to detect malicious JS codes have been proposed, such as monitoring the execution of a JS code at run time using a sequence of events to obtain vectors for classification [115]. Another approach is to learn and detect malicious patterns in the structure and behavior of a JS code [62]. In another example, [133] has proposed a static detection approach to extract and

analyze JS features, where the assumption is that the number of some special functions is limited in benign cases and vice versa. Finally, [134] has also proposed the use of features extracted from a JS code using a three-layer Stacked denoising Autoencoders (SdA), which is a typical neural network and pattern classification. They classify binary features obtained from a JS code. However, compared to other models such as ADTree and RBF Support Vector Machine (SVM), SdA had the second-best false positive rate. Besides, they use sparse random projection for dimensionality reduction resulting in 480 features, which leads to a long training time compared to other methods.

Another approach is to use a wrapper method and a classifier for feature clustering [60] which generates a feature subset through feature selection. However, the technique employs a limited set of features, and the wrapper method is prone to suffer from overfitting and long processing time. Misuse (signature-based detection) and anomaly detection to detect malicious websites by integrating both supervised and unsupervised learning have also been proposed [141]. The assumption is that the misuse detection method can identify well-known attack types, but it cannot detect new ones. They use this method to detect malicious websites using the C4.5 decision tree algorithm. They then use anomaly detection to counter the weakness of the misuse detection method. Finally, they use SVM to identify anomalies in the websites determined as benign by the misuse detection method. This approach achieved a high detection rate. However, it is also prone to suffer from a high false-negative rate. [132] combines dynamic and static analysis to automatically classify a malicious JS and achieve both scalability and a high true positive rate. They employ pattern analysis and build a malicious JS attack model using only JS function calls and textual content. [139] infer behavior model or typical behavior of malicious JS by active automata learning using Deterministic Finite Automaton (DFA). To identify the malicious traces of the same attack types, they use a JS replay mechanism to analyze data dependency and defense rules.

These conventional approaches have achieved a certain degree of accuracy in

the detection of malicious JS content to some extent. However, they have several limitations; these approaches identify specific attack types using constructed feature sets, subsets, and functions such as *navigator.userAgent.toLowerCase()*, *unescape()*, *eval()*, *setTimeout()*, etc. for detection of malicious JS content. These would only go as far as the predefined features. Any variant beyond the defined set of features could result in a false negative. These limitations explain why most of these methods are prone to suffer from a high false-negative rate. Another drawback to these conventional approaches is that they generate high-dimensional representations, translating to a slow detection rate. These approaches are also known to lose the order of words in the resulting representations. Use of Doc2Vec for plain-JS feature learning has previously been proposed [95]. This approach automatically learns fixed-length vector representation for a variable length of plain-JS content with better accuracy and fast detection rate compared to other feature learning methods. However, when performing feature learning for a JS code that contains elements other than JS code words, such as an obfuscated JS code, the model is prone to suffer from false-positive and false-negative cases because an obfuscated JS content does not accurately represent a JS code structure. Even though these misclassified JS cases are a few in paragraph vector models compared to other models, there is a need to improve further the performance for better detection of malicious JS codes. We, therefore, use AST-JS for code structure representation and Doc2Vec for feature learning of AST-JS. This approach is expected to provide even better accuracy as AST-JS gives a robust property against some perturbation in codes.

1.4.2 Obfuscated malicious JS codes detection

There exist several approaches for the detection of obfuscated malicious JS codes. The most basic and widely used approach is the use of JS code beautifier tools and formatters, such as DirtyMarkup JS code beautifier [28], Online JS code beautifier [69] and Dan’s Tools JS code viewer, beautifier, and formatter [26] among others. These tools make an obfuscated JS code look pretty, readable, editable, and analyz-

able. The tools are intended to reverse the effects of obfuscation for JS code analysis. However, these JS code deobfuscation tools are no match to an obfuscated JS code with multiple layers of obfuscation as some sections of the code remain obfuscated even after a JS code deobfuscation and formatting.

Other studies attempt to perform JS code deobfuscation for the discovery of the original and malicious JS codes by converting the obfuscated JS codes to regular ones for analysis [72, 140, 40]. In other approaches, a set of heuristics is created by analyzing specific obfuscator features. However, the heuristics-based approach needs security personnel intervention during code analysis, and it is limited to specific obfuscator tools [119, 123]. Other methods use a set of features such as JS code functions known to be prevalent in obfuscated malicious JS codes such as *eval()*, *unescape()* and *document.write()*. These syntactic-based features are prone to a high false positive rate [22, 48, 58]. Recently, approaches that use machine learning and deep learning to detect obfuscated JS codes are also increasing [119, 40, 123, 32, 99].

Skolka et al. [119] analyze the extent of application of minification and obfuscation in the web with an emphasis on understanding the most popular tools used for code transformation. The study point towards the development of targeted defense techniques that focus on particular obfuscation tools. They use a Tree-based Convolutional Neural Network to classify AST for three classification tasks, namely, TRANS, which determines whether a given piece of JS code has been minified or obfuscated using a tool, OBFUS, which checks for the presence of obfuscation, and TOOL-X, which determines the specific tool used to obfuscate a JS code. They, however, do not evaluate obfuscation methods applied in benign, and those in malicious JS code and study of specific obfuscation tools could lead to biased features that cannot be easily generalized to other datasets.

Some deobfuscation techniques have been developed to conduct reverse engineering of an obfuscated JS code. For example, [140] developed a generic deobfuscation method to extract a simplified code. The paper combined a dynamic analysis with

concolic (Symbolic) executions to collect JS code traces. These traces are simplified to reconstruct a control-flow graph (CFG) and data-flow of the target program using taint propagation. Obfuscation adds unnecessary transformation instructions and constructs. These details are discarded, and only the parts essential for functionality are maintained. Some automatic attacks are often specific to particular implementations of obfuscation transformations or individual attacker goals, such as CFG simplification. Some JS code analysis technique requires execution traces that cover all the code of the program being analyzed. This process ensures that the CFG is complete with no missing statements, basic blocks, or arcs [42]. However, the proposed system [140] only emphasizes the simplification of JS code execution traces without performing code coverage. This paper does not make any assumptions about the obfuscation technique. Moreover, a dynamic unpacker needs to unpack the code in memory at runtime to obtain the unpacked code.

Lu et al. [72] proposed a semantics-based approach for automatic deobfuscation of a JS code using dynamic analysis. They collect bytecode execution traces from the target program and use dynamic slicing and semantics-preserving code transformations to simplify the trace automatically, then reconstruct deobfuscated JS code from the simplified trace. The resulting code becomes observationally equivalent to the original program for the execution path considered. Thus, obfuscations are simplified away, which exposes the core logic of the computation it performs. This method achieves an excellent detection performance. However, the deobfuscation and simplification methods do not identify a JS code maliciousness, as they are designed to detect code-unfolding and string-based obfuscations.

Attackers use JS code functions, such as *eval()*, *unescape()* and *escape()*, to obfuscate malicious JS codes. These functions can be easily detected using signature-based tools. Hence the functions are mostly hidden. Gorji et al. [40] use internal function debugging and map the behavior of a webpage to a sequence of internal function calls, generating a behavioral signature for each JS code malware family. To identify these hidden functions, they intercept the suspicious function invocations,

which can be triggered at runtime using a browser. They use reverse engineering to identify internal function calls. However, obfuscation tools can result in a JS code with behavioral characteristics that are different from the original JS code [119]. Also, they select a limited set of 15 internal function calls of *jscript.dll* and *mshtml.dll* from *jscript* and *mshtml* libraries, which they assume are fully representative of the JS code malware family.

Multiple classifiers have been proposed to detect obfuscation and evaluate the feasibility and accuracy of distinguishing between different classes of a JS code. Tellenbach et al. [123] collect a data set of obfuscated and non-obfuscated JS codes and selects and extract a set of 45 features from the data set. The features used include frequency of specific keywords, number of lines, characters per line, number of functions, and entropy, among others. This paper assumes that there are a few JS codes that are both benign and obfuscated, and hence, they mainly focus on classifying obfuscated and non-obfuscated JS codes. They also fail to detect JS codes obfuscated with tools not present in the training set, and detection precision depends on the manually specified limited set of features.

For detection of obfuscated malicious JS code instances, [32] leverages the use of syntactic units using n-grams features from an AST traversal and a random forest classifier. They use specific constructs for accurate detection distinguishing obfuscation from maliciousness. However, they grouped units with the same abstract syntactic meaning, such as *FunctionDeclaration* and *VariableDeclaration* as Declaration node, and *ForStatement* and *WhileStatement* as a Statement node losing context information. They also use specific features, namely extracted syntactic units, which translates to limited learning components. In contrast, this study uses all JS AST information so that the FastText model can learn the most detailed features.

Algorithms such as j48 decision tree, Naive Bayes, Logistic Regression, and linear SVM have been proposed [99] to develop a machine-learning-based approach to detect obfuscated malicious JS code. The approach dynamically executes the trace of

a JS code and extracts none-ordered and non-consecutive sequence pattern features. In addition, a semantics-based deobfuscation technique [72] is adopted to simplify the trace using deobfuscation slicing. However, this approach assumes that the deobfuscated trace will be useful for feature extraction that represents the internal characteristics of a malicious JS code. Also, obfuscation tools have been shown to result in structurally different codes from the original ones.

These existing approaches adopt either a static or dynamic approach to deobfuscation and detection of obfuscated malicious JS codes. The main issue in static analysis is coping with obfuscation, specifically packing and encoding obfuscation. Static techniques have also been employed to assess if a malicious JS code is similar to a previously seen variant without actually performing the costly task of unpacking and decoding. However, with attacks such as zero-day JS code malware using obfuscation, a static analysis technique generates many false positives. Dynamic analysis techniques have been shown to suffer from limited code coverage and the complexity of the JS code runtime environment [49]. Limited code coverage results from cloaking, a technique employed by attackers to fingerprint the victim’s web browser. The malicious content can only be visible if certain conditions are met, such as using a specific browser version and a vulnerable plugin. The complexity of the JS code runtime environment results because JS code is in various applications utilizing the functionalities of plugin extensions they support. A dynamic analysis set-up will mainly include known browsers and plugins, ignoring less popular plugins and their vulnerabilities.

Most of the JS code dynamic analysis techniques use unfolded code context logs and assume that the original code can be retrieved from the deobfuscated one. JS code event callback feature, such as a mouse click event, also makes it challenging for dynamic analysis to trigger a JS code automatically. For example, the deobfuscator by [72] is a fully dynamic solution. However, the JS code obfuscated part requires localization as preprocessing the unfold loops during the abstraction stage. Some of these JS code deobfuscation and detection approaches assume that the transfor-

mation applied to a JS code preserves the overall semantics of the code. Benign JS codes may apply simple obfuscation techniques such as string splitting. However, in the case of malicious JS codes, attackers apply sophisticated obfuscation techniques such as packing and encoding in an attempt to perform evasion attacks. In some instances, a single code contains different obfuscation techniques, for example, double packing or packing coupled with encoding. These multi-layered obfuscation techniques dramatically affect the structure of the resultant JS code. In such a case, the resultant AST-JS would not accurately capture the structure of the original JS code. Therefore, to improve the detection performance of a machine learning model, this study proposes combining the DUD-preprocessing approach to enhance feature learning.

1.4.3 URL and URL JS content detection

Researchers have put forward many prevention techniques to keep up with the increase in attacks and new attack methods utilizing URL and JS content. Blacklists are repositories of known malicious URLs and have long been employed to detect malicious URLs and JS content. Heuristic and signature-based systems search for signs of standard attack patterns. The weakness of these approaches is that new or variant URLs and their JS and zero-day attacks can easily evade detection. Further, obfuscation is a method widely used by cyber-criminals to perpetrate attacks and evade detection.

Some studies [78, 76, 77] have proposed embedding the bag-of-words model in a URL or features of a URL and its JS content. For example, Ma et al. [78, 76, 77] proposed an approach to analyzing a URL to predict the maliciousness of websites using an online confidence-weighted (CW) algorithm for lexical feature learning. Individual feature vectors were manually engineered. Ma et al. [78] acknowledged that other potentially useful sources of information for features could improve classification accuracy. However, they do not examine a web page’s actual content, citing reasons such as user safety, model operation costs, classifier applicability to

the URL context, and reliability in obtaining a malicious webpage version. They also use bag-of-words features and therefore do not preserve token order. Conversely, the proposed approach uses JS content features. Since AST-JS is resistant against perturbation in JS content, it makes the model developed applicable for various JS contexts and content.

Other systems [11, 10, 82, 8, 96] have been proposed that use constructed features or feature engineering. For example, Bilge et al.’s [11, 10] EXPOSURE conducted a passive DNS analysis to identify malicious domains. It is a dynamic reputation system based on passive recursive DNS (RDNS) monitoring. They extracted 15 behavioral features to train the J48 decision tree classifier. Their study assumed that large volumes of DNS data requests should exhibit sufficient behavioral differences to distinguish between benign and malicious domains. In another study, a system to detect malicious domains using DNS records and domain name features was proposed by Al Messabi et al. [82] using the J48, a C4.5 decision tree. This approach studied prior DNS activities for each domain and the relationship between defective and legitimate domains’ physical behavior [8], combining several existing DNS-based and domain name-based features from previous work [11, 10, 96]. They assumed that eight unique behavioral features could accurately identify malicious websites and proactively detect an attack. However, feature engineering is a daunting task, even for domain experts. This study performed automatic feature selection to address these challenges.

Kuyama et al. [65, 64] sought to detect targeted attacks by monitoring the communication between the command and control (C&C) server and the computers in the Local Area Network (LAN). Their proposed method identified new C&C servers using supervised machine learning and extracts the WHOIS feature points, DNS information, and search site. The malicious domains were detected using a neural network and an SVM separately for performance comparison. Since manual feature selection is a tedious task, even for domain experts, this study proposes an autonomous feature-selection method using SHapley Additive exPlanations (SHAP)

values and tree ensemble methods. Other studies have attempted to detect specific URL attack types such as phishing [79, 33, 88, 142, 9, 51], malicious advertisements, and click fraud [80, 35, 68, 98, 136, 59, 3]. These approaches are suited for specific attack types and, therefore, do not generalize well to other attacks. Masri et al. [80] proposed a system for the automatic classification and detection of malicious advertisements. They used VirusTotal [129], URLVoid [128], and TrendMicro [127]. Their study relied heavily on other tools that employ signatures and blacklist services. Therefore, it was prone to problems inherent to signature- and blacklist-based systems [37].

In this study, an automatic feature selection method for malicious JS content using SHAP values is proposed. A more thorough analysis of JS content-based features may help detect threats [113, 17]. These features provide rich information for the feature learning process compared to URL-based ones, as much information is extracted from a page. However, safety concerns may arise if the JS content is executed. In this study, AST-JS is used for the code structure representation. Further, AST-JS enables the capturing of more details regarding a web page. The assumption is that more information would lead to a better prediction model. It is assumed that a correlation between various web-based attack features exists, and a machine learning model can identify these relationships with minimal effort.

1.5 Detection of JavaScript-based Attacks

In Chapter 2, the use of AST-JS as a code structure representation, a feature learning approach to malicious JS content using Doc2Vec, and a classifier for AST-JS features classification is introduced. AST-JS is obtained from the JS code using Esprima, ECMAScript parsing infrastructure for multipurpose analysis. Esprima can be used to perform lexical and syntactic analysis of a JS program. Doc2vec is a neural network model that can learn context information of texts with variable length. This model is a well-suited feature learning method for JS codes, consisting of text content ranging from single line sentences, paragraphs, and full-length documents.

Besides, features learned with Doc2Vec are of low dimensions, which ensure faster detections. Two types of AST-JS manipulations are performed To enhance feature learning for Doc2Vec: AST-level merging and AST-subtree realignment. AST-level manipulation involves randomly changing AST subtree locations, and AST-level merging involves extracting JS-AST body to create AST combinations.

The developed approach automatically learns feature vectors compared to other previous methods, which employ manually crafted feature sets and subsets. A classifier model judges the maliciousness of a JS code using the learned features. The performance of this approach is evaluated using the D3M dataset (Drive-by-Download Data by Marionette) for malicious JS codes and the JSUNPACK plus Alexa top 100 websites datasets for benign JS codes. The performance of Doc2Vec on plain-JS and AST-JS is compared to other feature learning methods. The experimental results show that the proposed AST features and Doc2Vec for feature learning provide better accuracy and fast classification in malicious JS code detection compared to conventional approaches and can flag malicious JS codes previously identified as hard-to-detect.

1.6 Deobfuscation, Unpacking, and Decoding of Obfuscated Malicious JavaScript

In Chapter 3, a model for DUD-preprocessing of obfuscated malicious JS codes is introduced. The existing deobfuscation methods assume that a specific tool can recover an original JS code entirely. For a multi-layer obfuscation, general deobfuscation tools such as Dirty Markup, Online JS Code Beautifier, and Dan’s Tools JS Code Formatter realize a formatted JS code through JS code beautification or simplification. This process involves taking an obfuscated code, analyzing it, and making it readable again to uncover its true functionality. However, some sections remain encoded. For the detection of such codes, this study performs DUD-preprocessing by function redefinition using a Virtual Machine (VM), a JS code editor, and a

python `int_to_str()` function to facilitate feature learning by the FastText model. Obfuscated, packed, and encoded malicious JS codes are deobfuscated, unpacked, and decoded, respectively. The FastText model, a machine learning model, performs feature learning for DUD-preprocessed JS codes. The learned feature vectors are passed to a classifier model that judges the maliciousness of an obfuscated JS code.

In performance evaluation, Hynek Petrak’s dataset is used for obfuscated malicious JS codes, and the SRILAB dataset and the Majestic Million service top 10,000 websites for obfuscated benign JS codes. The performance of the FastText model is compared to the Paragraph Vector models, the StarSpace model, the Long Short-Term Memory (LSTM) model, and the Term Frequency–Inverse Document Frequency (TF-IDF) model on the detection of DUD-preprocessed obfuscated malicious JS codes. Experimental results show that the proposed approach enhances feature learning and provides improved accuracy in the detection of obfuscated malicious JS codes. The model improves detection performance on obfuscated malicious JS codes, precisely packed and encoded JS code, including the ones which contain multi-layer obfuscation. In addition, the model automatically learns feature vectors compared to other previous methods that learn a set of manually extracted features. Besides, features learned are of low dimensions, hence ensuring faster detection.

1.7 Web-based Attack Detection by Feature Selection

In Chapter 4, a feature selection and classification approach for malicious JS content detection using SHAP and tree ensemble methods is proposed. SHAP is a game-theoretic approach that explains any machine learning model’s output and connects optimal credit allocation with local explanations using the classical Shapley values and their related extensions. The SHAP values show the extent to which a feature is responsible for a change in the model output. One of the premises is that the ability to interpret and identify the features with the most impact may enable the derivation

of other features with additional information to improve the detection performance for JS-based attacks. SHAP values may also provide necessary insights into the distribution of specific benign and malicious JS features. While other methods learn a set of manually extracted or engineered features, the SHAP-based approach adopts features that are automatically selected based on their contributions to the model output. Such features are obtained from the distribution of continuously evolving malicious JS content, ensuring that the global and local features that contribute to the model performance are captured.

Other studies limit their scope by studying specific malicious URLs and JS attacks, such as those containing phishing attacks, social engineering, malvertisement attacks, and Denial-of-service (DoS) attacks, among others. For this, the studies use manually engineered features unique to the type of attack they intended to identify. Such approaches have challenges in that they cannot be generalized to other attack types. Another premise is that features selected using SHAP values can overcome such challenges, as the security analyst can easily and quickly identify which features significantly impact the overall model performance.

The JS features are obtained from AST-JS, sample JS attack codes, and association rule mining. AST-JS represents adequate details of a JS code structure without executing the code. Using atomic features and allowing the model to determine the relationship between them is preferable instead of using composite features [11, 10]. Features based on SHAP values are used. While other methods learn a set of manually extracted or engineered features, in this study, features that are automatically selected based on their contributions to the model output are adopted. Malicious URLs and their JS content can stealthily perpetrate web-based attacks [94, 92, 91, 70]. Other studies [78, 20, 81] have shown that malicious URLs and their JS content exhibit features that are distinct from those of legitimate URLs. Therefore, this study focused on JS content-based features to detect malicious websites. The premise of this study is that malicious JS content originating from attackers exhibits different AST-based features than those of legitimate ones. A SHAP-based model

would learn such features from the distribution of continuously evolving malicious JS content. The malicious and benign JS datasets obtained from Hynek Petrak and the Majestic Million Service were used for performance evaluation. The performance of the proposed model is compared to that of other classifier models and feature selection methods on the task of malicious JS content detection. Experimental results show that the proposed approach is a better prediction model. In addition, using SHAP interaction values, a security analyst can easily and quickly identify AST-JS feature combinations or interactions that contribute more toward the model’s detection performance.

1.8 Performance Evaluations

In this study, three evaluation metrics are used: precision, recall, and F1-score. Precision is the ability of a classifier not to identify a benign JS code as malicious. Recall is the classifiers ability to identify all malicious JS codes. F1-score can be interpreted as a weighted harmonic mean of precision and recall [102]. Precision, recall, and F1-score are computed as,

$$Precision = \frac{TP}{TP + FP}, \quad (1.1)$$

$$Recall = \frac{TP}{TP + FN}, \quad (1.2)$$

$$F1 - score = 2 \frac{Precision \times Recall}{Precision + Recall}, \quad (1.3)$$

where; True Positive (TP) is the number of malicious JS codes correctly classified as malicious; True Negative (TN) is the number of benign JS codes correctly classified as benign; False Negative (FN) is the number of malicious JS codes classified as benign; False Positive (FP) is the number of benign JS codes classified as malicious.

For performance visualization, a Receiver Operating Characteristic (ROC) graph plotting the true positive rate against the false positive rate is used in this study. Area Under the Curve (AUC) score [102] is calculated for the ROC to determine

the model performance.

1.9 Thesis Outlines and Research Objectives

The main purpose of this dissertation is to detect web-based attacks. Therefore, a robust JS code structure representation method and low dimensions and efficient learning method for JS codes should be introduced. The approach will ensure highly accurate detection and fast classification for malicious JS codes.

The rest of this dissertation is organized as follows:

Chapter 2: A Machine Learning Approach to Detection of JS-based Attacks using AST Features and Paragraph Vectors

This chapter contains related works on JS-based attacks and the proposed JS-based attacks detection system. This model uses AST-JS, Doc2Vec, and a classifier and is a well-suited feature learning method for JS codes, consisting of text content ranging from single line sentences, paragraphs, and full-length documents.

Chapter 3: Deobfuscation, Unpacking, and Decoding of Obfuscated Malicious JavaScript for Machine Learning Models Detection Performance Improvement

This chapter contains related work on obfuscated malicious JS and the proposed model for DUD-preprocessing of obfuscated Malicious JS. The model enhances feature learning and provides improved accuracy in the detection of obfuscated malicious JS codes. Furthermore, it improves detection performance on obfuscated malicious JS codes, precisely packed and encoded JS code, including the ones which contain multi-layer obfuscation.

Chapter 4: Web-based Attack Detection by Feature Selection using Tree SHapley Additive exPlanations

This chapter contains related work on feature selection and the proposed web-based attack detection system by Feature Selection. Features based on SHAP values are automatically selected considering their contributions to the model output. Using SHAP interaction values, a security analyst can easily and quickly identify AST-

JS feature combinations or interactions that contribute more toward the model's detection performance.

Chapter 5: Conclusions

This chapter contains the summary of the dissertation and future works of this study.

The objectives of each chapter are highlighted below:

- Chapter 2: Development and evaluation of a machine learning approach for JS-based attacks detection.
- Chapter 3: Development and evaluation of a deobfuscation, unpacking, and decoding system for obfuscated malicious JS codes.
- Chapter 4: Development and evaluation of a web-based attack detection system by feature selection.
- Chapter 5: Summarizing Chapter 2 to Chapter 4.

Chapter 2

A Machine Learning Approach to Detection of JavaScript-based Attacks using AST Features and Paragraph Vectors

2.1 Introduction

Development objectives of benign JS codes differ from those of malicious ones attempting to trick or deceive users using soliciting words [36] such as 'free,' 'money,' and 'win.' This characteristic makes malicious JS codes different from benign ones. In such JS codes, Table 2.1 shows a list of frequently used function names, function types, possible threats, and intention in malicious JS codes. These have been selected from [60, 132].

Learning such specific words on JS would help to enhance the reliability of identifying the maliciousness [95]. In the previous work, a plain form of JS codes is provided to Doc2Vec as inputs, and a small dataset of 80 JS codes is used for feature learning. However, a JS code contains more structural details than information on its text contents, for example, function or variable, operational procedures, and

Table 2.1: A list of frequently used function names in malicious JS codes, function types, possible threats, and intention. This list provides evidence that malicious and benign JS codes contain different frequent words as malicious JS codes would be expected to utilize such function names compared to benign ones heavily.

Function name	Function type	Possible threats (Intension)
location.replace()	Change current URL	Redirect to malicious URL
document.write()	DOM operation	Embed malicious script
getUserAgent()	Check browser	Target specific browser
String.split()	String operation	Hide intent-encode/encrypt
setCookie()	Cookie access	Manipulate Cookie

stop conditions. Therefore, to effectively learn JS code contents, it is also imperative to consider the structure of a JS code. Therefore, the previous work [95] is extended in this Chapter so that essential information in a JS code can be considered to identify malicious intention embedded in JS codes. For this task, the use of Abstract Syntax Tree (AST) [145, 54, 87] for code structure representation, and Doc2Vec for feature learning is proposed. To this end, AST-level tweaks were performed to enhance feature learning for accurate malicious JS detection.

A brief explanation of the preprocessing for transforming a JS code into an AST structure (AST-JS) and the Doc2Vec feature learning is given in the following sections.

2.2 Proposed method

2.2.1 AST form of JavaScript code

Also called syntax tree, AST is a simplified parse tree or Concrete Syntax Tree (CST). AST represents the necessary and functional structure of code such as scopes, expressions, or declarations while at the same time avoiding redundancy by omitting unnecessary syntactic details [145, 54] such as whitespaces, punctuation marks, or comments which do not affect the original function of code even after the parsing process. Many of the recent cyber-attacks exploit JS vulnerabilities, in some cases employing obfuscation to hide their maliciousness and evade detection. AST's have

been used to accurately represent such obfuscated JS codes [23, 12, 57, 53].

The compilation process starts with lexical analysis, where a compiler performs scanning and tokenization. Then a parser takes this tokenized input and generates AST through syntax analysis [87]. Listing 2.1 shows an AST generated through syntactic analysis using $5+(1*12)$ as the source text. The JS code is represented in JS Object Notation (JSON). AST-JS can then be used as input for machine learning algorithms. These algorithms require input to be represented as feature vectors for classification or clustering tasks. This representation may result in high dimensional representations depending on the size of the dataset, which translates to longer detection time [66]. For faster detection, the use of fixed-length vector representation is proposed.

```
1  {'body': [{ 'expression': { 'left': { 'raw': None,
2    'type': 'Literal',
3    'value': 5.0}
4    'operator': '+',
5    'right': { 'left': { 'raw': None, 'type': 'Literal',
6      'value': 1.0}
7      'operator': '*',
8      'right': { 'raw': None, 'type': 'Literal', 'value': 12.0},
9      'type': 'BinaryExpression'},
10     'type': 'BinaryExpression'},
11     'type': 'ExpressionStatement' } ] },
12  'type': 'Program' }
```

Listing 2.1: A sample Abstract Syntax Tree (AST) generated using esprima, a syntactical and lexical analysis tool. This is obtained from a binary expression which contains numeric values (5, 1 and 12), operators (+ and *) and parentheses (()). It is worth noting that some details such as the parentheses, which would otherwise make the AST verbose, are omitted in the resultant AST.

2.2.2 Preprocessing

Preprocessing a JS code is a crucial step to enhance the model performance during feature learning. The regular expression is used to define the first type of Doc2Vec input. Regular expression meta-characters $[a - z]^+$ are used to specify the input range which matches any lowercase letter from a to z , remove symbols and special characters such as $*$, $=$ and $!$ such that only plain-JS text elements remain as shown in Fig. 2.1 which is a resultant JS code representation obtained from Listing 2.2. $^+$ indicates the preceding element and its one or more occurrences. In this case, it is used to match one or more characters in a regular expression. Feature learning in Doc2Vec requires a large amount of data. For this reason, a *json* file with random text data is added. This text data is necessary for two primary purposes; one acts as noise for performance validation by ensuring that the model can distinguish noisy dataset from the JS code contents during classification. Two, it is used when increasing the dataset by random insertion of text data into the JS codes to create new dummy JS code instances for feature learning. This approach is comparable to the one in previous work [95].

Code structure representation is employed using a parser called *esprima* [46] to define the second type of Doc2Vec input. *Esprima*, ECMAScript scripting-language standard, allows various code analysis and operations such as editing, transformation, visualization, and validation. This tool is used to perform syntactic analysis, parsing the JS codes into AST's. *Esprima* produces an AST-JS from a JS code string input as shown in Listing 2.3 which shows the resultant snippets of an AST-JS after parsing of Listing 2.2. This tool also checks the JS codes for syntactic errors, ensuring that the input is a valid JS program. If the input contains syntax errors, it will not be parsed into AST-JS. Finally, the plain-JS and the AST-JS are given to Doc2Vec for feature learning.

```
1 function d(a) {  
2   var c = typeof a;  
3   if ("object" == c) {
```

```

4  if (a) {
5  if (a instanceof Array) {
6  return "array";
7  ...
8  var b = Object.prototype.toStrin
9  if ("object Array" == b || "number" == ... && !a.
    propertyIsEnumerable("splice")) {
10 return "array";
11 }
12 ...
13 }
14 } else {
15 if ("function" == c && "undefined" == typeof a.call) {
16 return "object";
17 }
18 }
19 return c;
20 }

```

Listing 2.2: A snippet of a sample original JS code before preprocessing.

```

TaggedDocument(['function', 'var', 'this', 'function', 'var', 'typeof', 'if', 'object', 'if',
'if', 'instanceof', 'array', 'return', 'array', 'if', 'instanceof', 'object',
'return', 'var', 'object', 'prototype', 'toString', 'call', 'if', 'object',
'window', 'return', 'object', 'if', 'object', 'array', 'number', 'typeof',
'length', 'undefined', 'typeof', 'splice', 'undefined', 'typeof', 'splice',
'return', 'array', 'if', 'object', 'function', 'undefined', 'typeof', 'call',
'undefined', 'typeof', 'call', 'return', 'function', 'else', 'return', 'null',
'else', 'if', 'function', 'undefined', 'typeof', 'call', 'return', 'object',
'return', 'function', 'var', 'array', 'prototype', 'slice', 'call',
'arguments', 'return', 'function', 'var', 'slice', 'push', 'apply',.....]

```

Figure 2.1: A sample of a plain JS code (plain-JS) after preprocessing using a regular expression. This representation shows the resultant Plain-JS after the removal of symbols and special characters. It is used as Doc2Vec input for the first part of the experiments.

```

1  {
2  "type": "Program",
3  "body": [

```

```

4 {
5   "type": "FunctionDeclaration",
6   "id": {
7     "type": "Identifier",
8     "name": "d",
9     "range": [
10    9,
11    10
12  ]
13 },
14 "body": {
15   "type": "BlockStatement",
16   "body": [
17     {
18       ...
19       "type": "VariableDeclaration",
20       "declarations": [
21         {
22           ...

```

Listing 2.3: Snippets of a sample AST-JS after preprocessing using a parser. This representation shows the resultant snippets of AST-JS when parsing is applied to a JS code. It is used as Doc2Vec input for the second part of the experiments.

2.2.3 Feature learning by Doc2Vec

The bag-of-words approach is a common fixed-length feature that represents a sentence or a document as a simplified, unordered collection of words [143, 38]. For document classification, classifier training utilizes features such as word occurrence or count. This task capitalizes on a vocabulary of known words and a measure of the presence of known words. Documents with the same words or text are assumed to be similar, and the words can give the meaning of the document. However, bag-of-words has been known to lose context information and semantic structure. Word vectors [83] represent each word with a vector, concatenated with other word vectors,

and the resulting vector predicts other words. However, this model has only been extended up to sentence-level representation while relying on parsing [120]. Paragraph vectors [66, 85, 83, 24] learns continuous distributed vector representations for text with variable length of input sequences, for example, documents, paragraphs or sentences. This property is important because each JS code generally has a different length, and the sequence of words differentiates its behavior. Paragraph vectors have the advantages of retaining a sentence’s semantic structure, considering word order, and creating a low-dimensional representation. Word vector model [83] predicts a word given other words in a context and vice versa using the continuous bag of words model or the skip-gram model. Paragraph vectors extend word vectors by adding additional input units that represent documents as additional context. Each additional input unit acts as an ID for each input document.

For a sequence of training words $\mathbf{X} = \{\mathbf{x}_{t-2}, \mathbf{x}_{t-1}, \mathbf{x}_{t+1}, \mathbf{x}_{t+2}\}$, the model predicts the center word $\mathbf{x}_t$ represented by $\mathbf{X}_t$ and given by,

$$U_i = p(\mathbf{X}_t | \mathbf{X}, \mathbf{D}_i) \quad (2.1)$$

Here, $\mathbf{D}_i$ is the paragraph or document vector which represents unique vector mapping for i -th paragraph or document. The model has three layers. Assume that a word and a document, which is the input for the network, is represented by one-hot vectors $\mathbf{x}_i$ and $\mathbf{d}_i$. Each input layer unit corresponds to $\mathbf{x}_i$ and $\mathbf{d}_i$ representing the i -th word and a document input respectively. All input layer units make the vectors $\mathbf{X}$ and $\mathbf{D}$ for all M words and N documents in the corpus, used as the model input.

The hidden layer is of size p , which represents the paragraph vector dimensionality and is chosen during training. Both the input and hidden layers are fully connected, with each connection having a weight. Small values are used for random initialization of these weights before training the network. The hidden or embedding layer for this network is very simple for faster computation. A weighted sum of the

input layer activation is used as the activation for the hidden layer given by,

$$\hat{\mathbf{D}} = \mathbf{W}\mathbf{D} \quad (2.2)$$

Here, $\mathbf{W}$ is a $p \times N$ matrix with all weights between the input and hidden layer. $\mathbf{D}$ contains the concatenation of the word and document vectors $\mathbf{X}$ and $\mathbf{D}_i$. $\hat{\mathbf{D}}$ is the hidden layer activation, this is the paragraph vector and an estimation which represents $\mathbf{D}$ in p -dimensions. The paragraph vector $\hat{\mathbf{D}}_n$ for document ID n would be given by the n^{th} column in $\hat{\mathbf{D}}$.

The output layer corresponds to V words in the vocabulary, which gives the number of the output layer units. The output layer is also fully connected to the hidden layer. This layer represents an estimation of the term or word vector and can be represented as,

$$\mathbf{U} = \mathbf{W}'\hat{\mathbf{D}} \quad (2.3)$$

Here, $\mathbf{W}'$ is a $V \times p$ matrix with all weights between the hidden and output layer. $\mathbf{W}'$ has output vectors represented as rows for each word in the vocabulary. A word at index m would be given by $\mathbf{W}'_m$. Therefore,

$$U_m = \mathbf{W}'_m \cdot \hat{\mathbf{D}} \quad (2.4)$$

After training the network, $\mathbf{W}$ will have the word and document embeddings, respectively, which will be reused for prediction. When given a document vector, the hidden layer will be used as the output layer giving fixed-size paragraph vectors of p -dimensions. A softmax function is applied To map $\mathbb{R}$ given by $\mathbf{U}$ to a probability distribution,

$$s(\mathbf{U}_i) = \frac{e^{\mathbf{U}_i}}{\sum_k e^{\mathbf{U}_k}} \quad (2.5)$$

Where $0 < s(\mathbf{U}_i) < 1$, $\sum_i s(\mathbf{U}_i) = 1$ and $s(\mathbf{U}_i)$ boosts activation of the units with the highest function where if activation is higher, $s(\mathbf{U}_i)$ will move towards 1 and 0. The softmax function will result with $\hat{\mathbf{t}}$, an approximation of the actual term to be

predicted (an estimated probability distribution). During training, $\hat{\mathbf{t}}$ is compared to $\mathbf{t}$ the target term vector using a loss function which should be low when $\hat{\mathbf{t}}$ and $\mathbf{t}$ are similar. This can be given by cross entropy as,

$$E(\hat{\mathbf{t}}, \mathbf{t}) = -\sum_i \mathbf{t}_i \log \hat{\mathbf{t}}_i \quad (2.6)$$

Where $E(\hat{\mathbf{t}}, \mathbf{t}) > 0$, if $\mathbf{t}_i = 0$ the i^{th} element of the sum will be 0 and E is low when $\hat{\mathbf{t}}_i$ is close to 1. Let y be an index where $\mathbf{t}_y = 1$ and $\mathbf{t}$ is a one-hot vector, then,

$$E(\hat{\mathbf{t}}, \mathbf{t}) = -\log \hat{\mathbf{t}}_y \quad (2.7)$$

During training, a set of training predictions are compared to corresponding actual target vectors by a loss function minimization on a set of training points. With a set of training target vectors $\mathbf{T}$ and $\mathbf{t}_j$ as the j^{th} target vector, the loss function can be given as,

$$L = \frac{1}{|\mathbf{T}|} \sum_{j=1}^{|\mathbf{T}|} E(\hat{\mathbf{t}}_j, \mathbf{t}_j), \quad (2.8a)$$

$$L = \frac{1}{|\mathbf{T}|} \sum_{j=1}^{|\mathbf{T}|} -\mathbf{W}'_{y_j} \cdot \hat{\mathbf{D}}_j + \log \sum_{m=1}^M e^{\mathbf{W}'_m \cdot \hat{\mathbf{D}}_j} \quad (2.8b)$$

Where E is averaged over a set of training examples. y_j is the index of the actual word for training example j in the target vector $\mathbf{t}_j$ and its corresponding output vector is given by $\mathbf{W}'_{y_j}$.

Doc2Vec, an implementation [108] of paragraph vectors, is a novel feature learning method for text documents that converts a document into a feature vector. It is an unsupervised algorithm that produces vectors of a fixed-length [66]. A small neural network is trained for a prediction task using labels obtained from a JS dataset, and a part of this trained network is reused to give a document or a paragraph vector. A JS code classification task is then performed using these vectors. Doc2Vec uses many hyper-parameters to learn a feature vector. These include *vector_size*, representing the dimensionality of a feature vector, *min_count*, which ignores all words with a frequency lower than a preset one, and *window*, which is the maximum

distance between the current word and the predicted one. There are two approaches to paragraph vectors:

Distributed Bag of Words version of Paragraph Vector (PV-DBoW) PV-DBoW is an unsupervised learning framework for feature extraction in text documents. As shown in Fig. 2.2, a paragraph vector is mapped to a feature vector whose mapping function is learned with a set of randomly selected words in the same paragraph. Then the feature vector is trained and used for prediction using a classifier model.

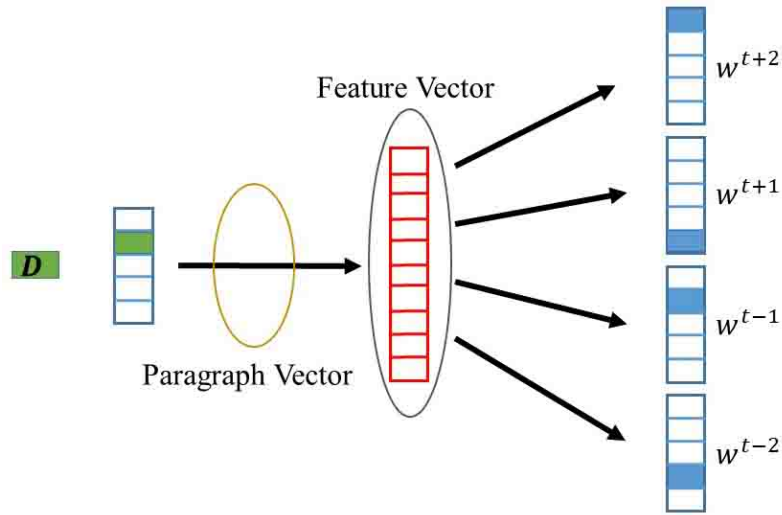


Figure 2.2: Distributed Bag of Words version of Paragraph Vector (PV-DBoW). Words are randomly sampled from a text window, and a trained paragraph vector is used to predict the words.

Distributed Memory Model of Paragraph Vectors (PV-DM) PV-DM is an unsupervised learning framework where the model learns a small neural network with a document or paragraph ID. Then the word vector and paragraph vector are concatenated or averaged to create a feature vector as shown in Fig. 2.3. This feature vector is trained and used for prediction using a classifier model.

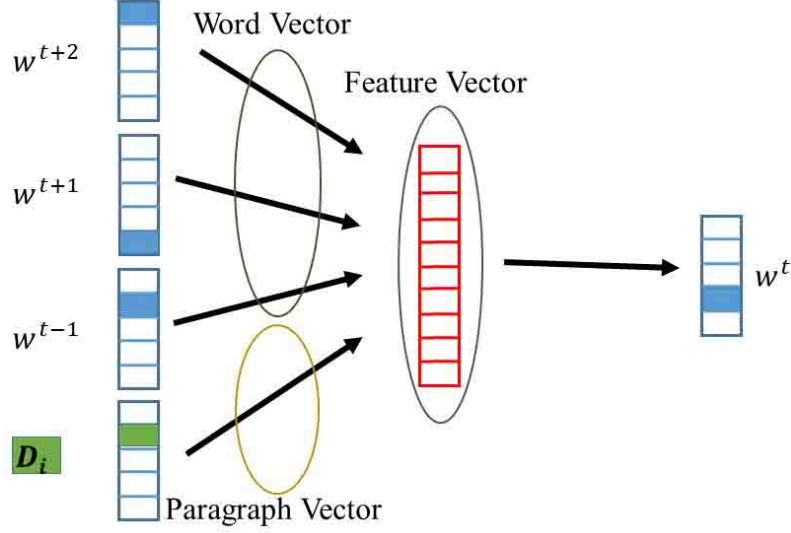


Figure 2.3: Distributed Memory Model of Paragraph Vectors (PV-DM). Word vectors and paragraph vectors are concatenated to create a feature vector for prediction using a classifier.

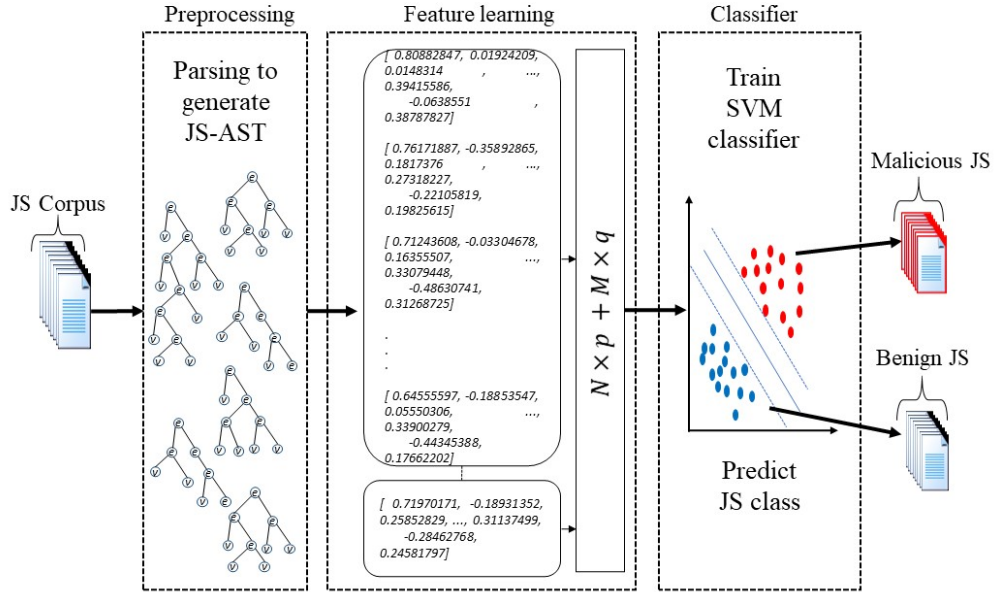


Figure 2.4: The layout for learning and classification of malicious JS contents AST features.

Fig. 2.4 shows the overview of learning and classification of malicious AST-JS features. It consists of three components, which are, preprocessing, feature learning, and classification. During preprocessing, the JS corpus is parsed into AST-JS features by syntax analysis. The AST-JS features are used for feature learning by

Doc2Vec. Training of the classifier is performed using the learned fixed-length feature vectors. A new JS code generates an AST-JS through parsing, and inference for feature vectors is conducted from the trained Doc2Vec model vector space. Lastly, the classifier uses the inferred feature vectors for prediction.

Cross-validation is performed to find the optimal hyperparameters and classifier learning with a training set of feature vectors. The performance of Doc2Vec on malicious JS contents detection is compared to other feature learning methods on the same dataset. Algorithm 1 shows the process for learning feature vectors from the JS codes in PV-DM, which consists of training on input JS codes for a given corpus. The model learns both word embeddings and document embeddings (lines 4-9), with weights initialized randomly using small values before training the network. These embeddings are of p dimensions (line 8) considering a word frequency higher than *min_count* (line 5). Algorithm 2 shows the process for learning feature vectors from the JS codes in PV-DBoW (lines 4-8), the same as in PV-DM. However, this model only learns document embeddings (line 4) while keeping the word embeddings fixed. The updating of learned embeddings for both PV-DM and PV-DBoW through negative sampling (lines 2-11), gradient descent (lines 12-18), and backpropagation (lines 21-24) as shown in algorithm three outputs the feature vectors which are then fed directly to SVM for train and test purposes.

Algorithm 1 PV-DM malicious JS contents detection

Require: JS_codes , model parameters $\mathbf{W}$ and $\mathbf{W}'$, hyper-parameters $size$, $window$, min_count .

Ensure: $neul[s]$ - Word vectors $\mathbf{U}$, Document vectors $\mathbf{D}$.

repeat until JS_codes run out.

{read a document from JS_codes }.

$Doc[] = Read_Codes(JS_codes)$;

for each $tword, tDoc \in Doc$ **do**

if $tword > min_count$ **then**

 {feed forward.}

for each $wWord, dDoc \in Context(tword, tDoc, window)$ **do**

for $s = 0$ to $size$ **do**

$neul[s] = neul[s] + syn0[wWord, dDoc][s]$;

end for

end for

end if

end for

Algorithm 2 PV-DBoW malicious JS contents detection

Require: JS_codes , model parameters $\mathbf{W}$ and $\mathbf{W}'$, hyper-parameters $size$, $window$.

Ensure: $neul[s]$ - Document vectors $\mathbf{D}$.

1: *repeat* until JS_codes run out.

2: {read a document from JS_codes }.

3: $Doc[] = Read_Codes(JS_codes)$;

4: **for each** $tDoc \in Doc$ **do**

5: {feed forward.}

6: **for each** $dDoc \in Context(tDoc, window)$ **do**

7: **for** $s = 0$ to $size$ **do**

8: $neul[s] = neul[s] + syn0[dDoc][s]$;

9: **end for**

10: **end for**

11: **end for**

Algorithm 3 Paragraph vectors training

Require: $\text{neu1}[s]$ - Word vectors $\mathbf{U}$, Document vectors $\mathbf{D}$.

Ensure: updated syn0 - Feature vectors.

```
1: {n}egative sampling.
2: for each sample  $\in$  (nsample || tword) do
3:   if sample == tword then
4:     label = true
5:   else
6:     {s}ample  $\in$  negative samples
7:     label = false
8:   end if
9:   for  $s = 0$  to size do
10:     $f \mathrel{+}= \text{neu1}[s] * \text{syn1}[\text{sample}][s];$ 
11:   end for
12:    $g = \text{getGradient}(f, \text{label});$ 
13:   for  $s = 0$  to size do
14:      $\text{neu1e}[s] \mathrel{+}= g * \text{syn1}[\text{sample}][s];$ 
15:   end for
16:   for  $s = 0$  to size do
17:      $\text{syn1}[\text{sample}][s] \mathrel{+}= g * \text{neu1}[s];$ 
18:   end for
19: end for
20: {backpropagation.}
21: for each wWord, dDoc  $\in$  Context(tword, tDoc, window) do
22:   for  $s = 0$  to size do
23:      $\text{syn0}[\text{wWord}, \text{dDoc}][s] \mathrel{+}= \text{neu1e}[s];$ 
24:   end for
25: end for
```

2.3 Experiments

2.3.1 Experimental setup

For the experiments, a dataset of 5,024 JS codes is used that consists of 2,512 malicious and 2,512 benign JS codes. These JS codes were obtained separately, all malicious codes come from the D3M dataset (Drive-by-Download Data by Marionette) [4, 45], whereas the benign JS codes are collected from a generic JS unpacker web-based portal - JSUNPACK [56] and Alexa top 100 websites [5] by depth crawling.

Performance comparison is conducted using the dataset for the two types of feature embedding: Doc2Vec for plain-JS and Doc2Vec for AST-JS. First, feature learning is conducted using input defined as a regular expression, as shown in Fig. 2.1, that is, by learning feature vectors from both benign and malicious JS codes represented as a plain-JS. For this, 40 dummy data are created for each original JS code. The resulting dataset is 200,960 JS codes. Second, feature learning is conducted using input defined as AST-JS as shown in Listing 2.3, that is, by learning code structure representation. For this, two types of manipulations are performed on the AST-JS to enhance feature learning by Doc2Vec, AST-level merging, and AST sub-tree realignment. For merging, ASTs are generated from the original JS codes by syntax analysis and then create combinations of the AST bodies to form a new AST-JS. For AST sub-tree realignment, the AST-subtrees are randomly shuffled to change their location in a particular AST-JS. Two, three, four, and five artificial AST-JS are added per each original AST-JS using these AST-level manipulations. This results in the new datasets of 15,072, 20,096, 25,120 and 30,144 AST-JS. Besides, since the actual web setting generally contains more benign JS codes than malicious JS codes, a real web setting is simulated by making ten times more benign AST-JS. This results in the new datasets of 82,896, 110,528, 138,160 and 165,792 AST-JS.

For the paragraph vector models, three hyper-parameters were found to affect the performance of the models significantly. These are the *vector_size* representing the

feature vectors' dimensions, word frequency given by *min_count*, and *window*. For *vector_size*, experiments are conducted with feature vector dimensions of the range 100-1,000. For word frequency, the range 1-10 and window size of range 1-8 are used. *M* and *N* are mapped to 200 dimensions of *p*, *min_count* = 5 for word frequency and a window size of 8. These are the hyper-parameters that were found optimal for effective feature learning and detection of malicious JS contents. Cross-validation is performed to evaluate the performance of the proposed approach with $k = 10$ by dividing the feature vectors into 10-folds, where during ten iterations, nine folds are used for classifier training and the remaining one-fold for model evaluation. SVM is used with *kernel* = 'linear' and $C = 1$ parameters as the classifier for the experiments, a classification method that constructs a hyper-plane in high dimensional space [14, 103] for regression and classification. It is mostly used as a binary classifier. For this, precision, recall, and F1-score are computed. A Receiver Operating Characteristic (ROC) graph was used for performance visualization, and Area Under the Curve (AUC) score for the ROC was calculated to determine the model performance. A *t*-test is also performed to check for significant difference in performance between the approach and other feature learning approaches: LDA (*n_components* = 100, *max_iter* = 5, *learning_method* = 'online', *learning_offset* = 50. and *random_state* = 0), LSA (*n_components* = 80, *algorithm* = 'randomized', *n_iter* = 5, *random_state* = None and *tol* = 0.0), TF-IDF (*norm* = 'l2', *use_idf* = True, *smooth_idf* = True and *sublinear_tf* = False) and Ngram (*ngram_range* = (1, 2)). A *t*-test is a statistical test to check for a significant difference in model performance by comparing the output. Using the *p*-value, which is an output of a *t*-test, one would tell whether the proposed approach truly performed better than existing ones. For the experiments, an alpha value was set to 0.05. Therefore if the *p*-value is less than the alpha value, there is a significant difference in performance.

2.3.2 Performance Comparisons

Precision, Recall, and F1-score This section presents the results of the experiments. Table 2a presents precision, recall, and F1-score for paragraph vector models and other feature learning methods trained using a plain-JS, and includes standard deviation and p -value. The overall performance of the models is generally good as each of the models achieved an average of above 50 percent in precision, recall, and F1-score. Paragraph vector models, PV-DM at 200 dimensions and PV-DBoW at 200 dimensions and *dbow_words* = 1, achieved superior results compared to the other models. PV-DBoW outperformed other models with 0.06 and 0.23 in precision, 0.09 and 0.23 in recall, and 0.09 and 0.23 in F1-score above the best and the worst of the other models.

Ngram with *ngram_size* = 1 achieved the lowest performance for the task of malicious JS content detection. Consequently, paragraph vector models resulted in fewer false positives and false negatives compared to the other models. The results of the two-tailed paired t -test to check for a significant difference in performance between paragraph vector models and the other feature learning methods further support the difference in performance for the six models on malicious JS content detection. Comparing each of the other feature learning methods with PV-DBoW shows a significant difference in performance for malicious JS contents detection where all p -values are less than an alpha value of 0.05.

The objective is to have a high recall, which translates to as few false-negative cases (misclassified malicious JS codes) as possible. A low false-negative rate is crucial because a false negative would mean a successful attack, that is, a malicious JS code that goes undetected. Assuming 10% of the dataset is taken as the test samples, the models trained with input defined as a plain-JS, shown in Table 2a, would miss 603, 804, 2,412, 2,613, 3,216, and 5,226 malicious JS codes, respectively. Even though the paragraph vector models achieve good results with fewer false negatives than these other models, there is a considerable need to improve these models' detection rates further.

Table 2a: Precision, recall, and F1-score obtained by feature learning using a plain-JS and p value representing significance difference in performance for paragraph vector models compared to other approaches. LDA stands for Latent Dirichlet Allocation, LSA stands for Latent Semantic Analysis, and TF-IDF stands for term frequency-inverse document frequency.

Model	Precision	Recall	F1-score
PV-DBoW	0.95 ($\pm$ 0.1332)	0.97 ($\pm$ 0.0429)	0.96 ($\pm$ 0.0716)
PV-DM	0.95 ($\pm$ 0.1620)	0.96 ($\pm$ 0.0473)	0.95 ($\pm$ 0.0955)
LDA	0.89 ($\pm$ 0.2456)	0.88 ($\pm$ 0.1403)	0.87 ($\pm$ 0.0866) *0.0009
LSA	0.85 ($\pm$ 0.1438)	0.87 ($\pm$ 0.1320)	0.86 ($\pm$ 0.1928) *0.0011
TF-IDF	0.83 ($\pm$ 0.3783)	0.84 ($\pm$ 0.2514)	0.84 ($\pm$ 0.2140) *0.0055
Ngram	0.72 ($\pm$ 0.0916)	0.74 ($\pm$ 0.1407)	0.73 ($\pm$ 0.1213) *0.0002

* = p value.

Table 2b presents precision, recall, and F1-score for PV-DBoW and PV-DM obtained with AST features. PV-DBoW and PV-DM achieve an improvement of over 4% and 3% in precision, 2% in recall, and 3% in F1-score compared to the same models trained using a plain-JS. The significant difference in performance is a result of JS code representation using AST-JS. AST-JS represents a JS code structure that facilitates capturing a JS code semantic and syntactic details, which would otherwise be lost when input is represented as a plain-JS.

Table 2b: Precision, Recall and F1-score obtained by feature learning on AST-JS and p value representing significance difference in performance for paragraph vector models compared to other approaches.

Model	Precision	Recall	F1-score
PV-DBoW	0.99 ($\pm$ 0.1413)	0.99 ($\pm$ 0.0819)	0.99 ($\pm$ 0.0279)
PV-DM	0.98 ($\pm$ 0.1271)	0.98 ($\pm$ 0.0766)	0.98 ($\pm$ 0.0312)
LDA	0.90 ($\pm$ 0.2200)	0.92 ($\pm$ 0.0514)	0.90 ($\pm$ 0.1388) *0.0061
LSA	0.87 ($\pm$ 0.1091)	0.90 ($\pm$ 0.1468)	0.87 ($\pm$ 0.1217) *0.0003
TF-IDF	0.85 ($\pm$ 0.1757)	0.86 ($\pm$ 0.1077)	0.87 ($\pm$ 0.3034) *0.0229
Ngram	0.76 ($\pm$ 0.1656)	0.77 ($\pm$ 0.2165)	0.76 ($\pm$ 0.1241) *0.0005

* = p value.

True positive and false positive rate The superior performance of paragraph vector-based models on feature learning for JS contents is also reflected in the ROC

graph and AUC results, as shown in Fig. 2.5 and 2.6, where paragraph vectors based models achieved high and low rates of true positives and false positives respectively compared to the other models. Ngram achieved the least AUC with a section of the curve being on the threshold. The difference in feature learning and prediction would explain the difference in performance between paragraph vector models and the other models. Ngram predicts x_i based on $x_{i-(n-1)}, \dots, x_{i-1}$ which limits its effectiveness in detection of malicious JS contents as its main focus is prediction of the next word. TF-IDF works on the word level by checking the importance of a word to a document in a corpus and produces a sparse matrix with zero elements. In the experiments, TF-IDF had the longest training time. LSA uses singular value decomposition to reduce matrix row count hence reducing the training time, but LSA has been shown to result with hard to interpret dimensions [106, 118]. LDA focuses on word and topic distribution, where each document is assumed to have a mixture of various topics that frequently use a small set of words. This approach is however more suited for information retrieval [135, 27, 52]. Using paragraph and word vectors via the Doc2Vec approach, representing words as feature vectors for a sentence, paragraph, or document, preserves word order and produces low dimensionality representations, which are not computationally expensive. As a result, malicious JS contents detection using paragraph vectors and AST’s results with a highly accurate and fast model as shown in Table 2.3 with an average detection time of 0.0019 (± 0.0008) and 0.0012 (± 0.0001) seconds per JS code for PV-DBoW and PV-DM respectively.

2.4 Conclusions

In this Chapter, a system for malicious JS detection using AST-JS for a JS code structure representation and feature learning using Doc2Vec is developed. Paragraph vector models have previously achieved good results for the task of malicious JS detection using plain-JS features. One expectation is that adopting a JS code structure representation using AST-JS would result in even better performance for

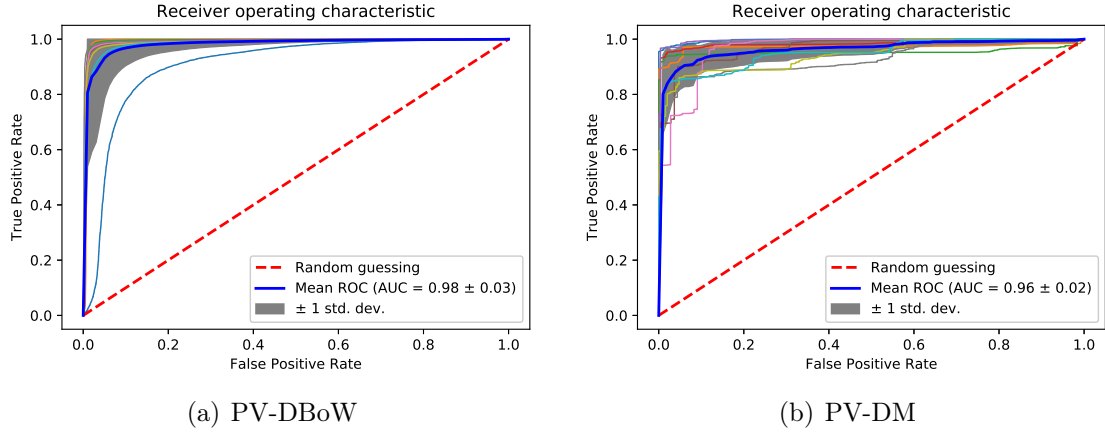


Figure 2.5: ROC curves and AUC representing performance of PV-DBoW and PV-DM on malicious JS codes detection with respect to false positive and true positive rates. The dashed diagonal line represents performance threshold as random guessing. The colored curves represent the performance of the specific model for each fold during cross-validation. Also included is the mean AUC score for the individual models.

Table 2.3: Computational time, representing the training and detection time per JS code obtained by dividing the total computation time with the number of JS code samples. It includes the time for different models.

Model	Training time	Detection time
PV-DBoW	1.4780	0.0019
PV-DM	1.5290	0.0012
LDA	0.2072	0.0190
LSA	0.0826	0.0079
TF-IDF	5.0419	0.8096
Ngram	0.9016	0.1120

the task of malicious JS code detection and enable detection of the previously identified hard-to-detect JS codes. Experiments were conducted using plain-JS, compared the performance of paragraph vector models to LDA, LSA, Ngram, and TF-IDF on detecting malicious JS code contents, and used a bigger JS dataset. Both the false negative and false positive cases were analyzed from the results obtained using a JS code input defined as plain-JS. AST-JS features are used for code structure representation to capture more details of a JS code. AST gives a robust property against some perturbation in codes. The results of the experiments provide a piece of significant evidence that the proposed approach provides good results for the task

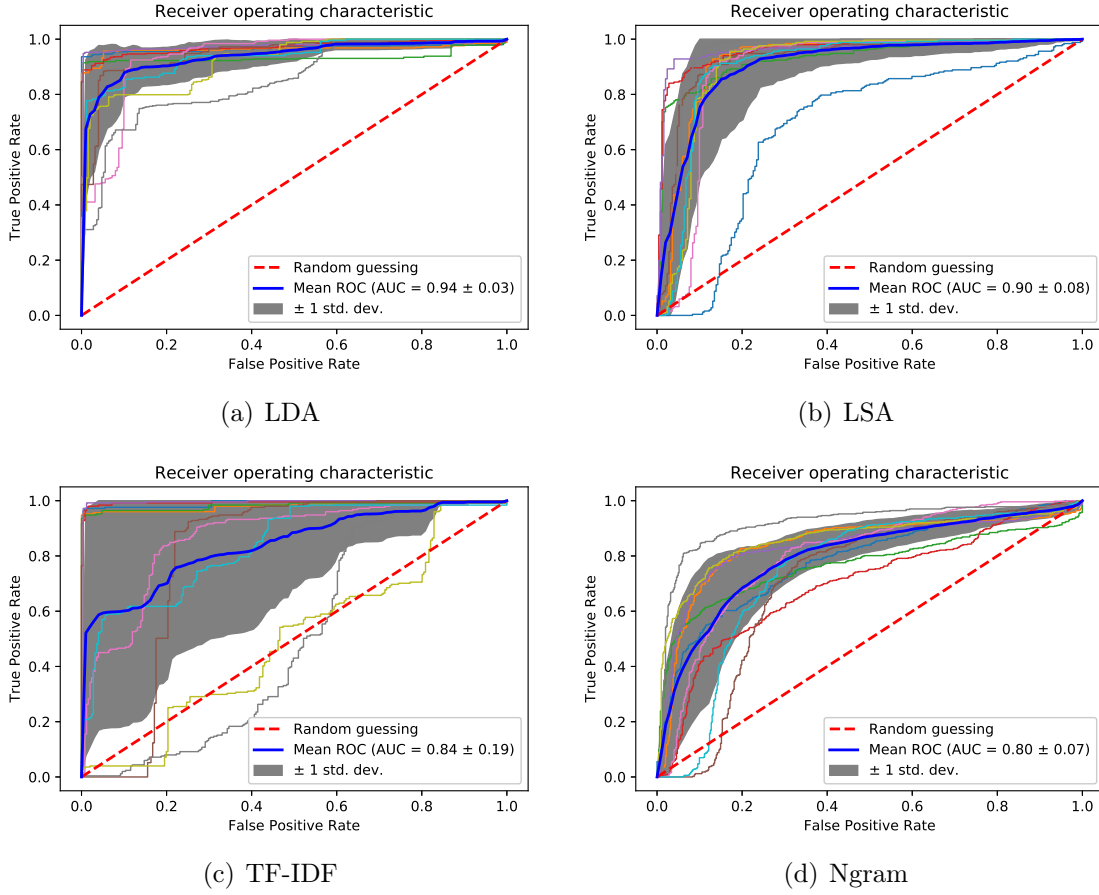


Figure 2.6: ROC curves and AUC scores representing performance of other models on malicious JS codes detection with respect to false positive and true positive rates. The dashed diagonal line represents performance threshold as random guessing. The colored curves represent the performance of the specific model for each fold during cross-validation. Also included is the mean AUC score for the individual models

of malicious JS code detection. There is a considerable performance improvement compared to the previous work, which learns a plain-JS as input for Doc2Vec. Besides, the approach eliminates the element of risk brought about by the previously proposed approaches, where a JS code content is evaluated at runtime, exposing the production environment to real-time threats. Also, Doc2Vec automatically learns the feature vectors hence ruling out the need for manually constructing and selecting functions, features (sets and subsets), and rules, which would be a tedious process requiring more resources to achieve. In addition, feature learning by Doc2Vec has the advantage of detecting new and variant malicious JS codes contents as the model has the capability of automatically learning new vector representations contrary to

conventional approaches, which would require more features, rules, and functions to be defined for effective detection of new malicious JS codes contents not previously defined.

As future work, using Generative Adversarial Network (GAN), the Doc2Vec feature learning process can incorporate adversarial machine learning for JS samples. This approach would be used to retrain the model to detect evasion and poisoning attacks performed against machine learning algorithms and, at the same time, improve the detection rate. The retraining would achieve even higher accuracy for detecting malicious JS code content using Doc2Vec for a JS code content feature learning. It is essential to have a fully representative and adequate benign and malicious JS code dataset to learn malicious JS content features successfully. Further work on the development of such a dataset is primal.

Chapter 3

Deobfuscation, Unpacking, and Decoding of Obfuscated Malicious JavaScript for Machine Learning Models Detection Performance Improvement

3.1 Introduction

Malicious JS codes attempt to trick users using items such as text, function names, function types, variables, and other contents. Obfuscation is employed to evade detection by security devices. This property can also hinder the feature learning process of a machine learning model as such JS codes contain numbers and special characters in addition to JS code words. It is essential to capture JS code details and structure by code structure representation to detect such malicious JS codes effectively. [93] developed the AST-JS approach for code structure representation and accurately detected some obfuscated malicious JS codes which contain the same semantics and syntax as the original JS codes. However, malicious JS codes employ

sophisticated obfuscation methods such as packing, encoding, and multi-layered obfuscation methods. The resulting obfuscated malicious JS codes are characterized by unnatural and unreadable syntax and corrupted code structure different from that of the original JS code. To effectively conduct feature learning on such obfuscated JS codes, it is essential to deobfuscate, unpack, and decode the obfuscated JS codes. The use of a JS code beautifier, formatter, VM, JS code editor, and a python `int_to_str()` function is proposed to obtain a Plain-JS code from an obfuscated one and the FastText model for feature learning coupled with a classifier for the prediction task.

In the following sections, the preprocessing for transforming an obfuscated JS code into a Plain-JS code and the FastText model used for feature learning are explained.

3.2 Proposed method

3.2.1 Preprocessing

Deobfuscation Deobfuscation is the process of analyzing and formatting an obfuscated JS code to make it readable again. It is also referred to as simplification or beautification. Deobfuscation uncovers the actual functionality of a JS code. A JS code beautifier is used to format an obfuscated JS code resulting in a deobfuscated JS code. However, if upon analysis of the resultant deobfuscated JS code, there is obfuscation in the dataset, there is still a need to process such an obfuscated JS code further. Such JS codes are identified as hard-to-deobfuscate JS codes as sections of the code remain obfuscated even after deobfuscation attempts.

Unpacking A JS code is packed or encrypted to hide its maliciousness, hindering its interpretation by static analysis. This concept is also used in benign JS codes for code compression and privacy purposes. JS code packers are diverse and more common with attackers attempting to evade detection. These tools wrap the entire

code using the *eval()* function that takes a string and tries to execute it as a JS code. The code is then loaded at runtime via the function.

The packed JS codes were evaluated using Oracle VM and a JS code editor. The script tags were stripped such that for each JS code; *JS_code* = `eval(function(p,a,c,k,e,d),...,obfuscated_JS_code)`, replace *eval()* with *console.log()* and parse the packed JS code to extract function arguments using *Unpacked_JS_code* = `eval('unpack' + JS_code[JS_code.find('}') + 1 : -1])` regular expression. It is important to disable the drag and drop and shared clipboard options in the VM virtual box manager and use a wifi network to avoid infecting the host Personal Computer (PC) and the Local Area Network (LAN) with malware. This process results in an unpacked JS code. However, upon analysis of the resultant unpacked JS code, there is presence of obfuscation in the dataset. The JS codes that remain obfuscated contain encoding implemented using *baseX()* functions.

Decoding JS code encoding is the process of converting all string literals of the stringArray, for example, by a JS code obfuscator tool, using either Base64 or RC4 [97, 117]. JS code packers also encode strings in a JS code to integers. The tools encode a JS code using a *baseX()* function. The resultant JS code is a shorter version that represents the long original one. It is a JS code compression method to reduce download time and represent a Uniform Resource Locator (URL) anonymously. Encoding methods such as ASCII, Unicode, and HEX affect the semantics of a JS code.

Python allows conversion of a string to an integer of a given base using *classint(x = 0)*, *classint(x, base = 10)* function, which returns an integer object constructed from a number or string given by *x*, or returns 0 if no arguments are given [107]. Encoded JS codes were evaluated by performing the inverse of this function. An *int_to_str(num, base)* function is implemented in python such that, for a number *num* and given *base*, *int(int_to_str(num, base), base) == num* which

returns a string object constructed from a number *num*, or return 0 if no arguments are given. Finally, the encoded JS codes are parsed to extract the function arguments using a JS code editor in a VM.

This process results in a Plain-JS code. The Plain-JS code is parsed to AST-JS for feature learning using the FastText model. Listing 3.1 shows a JS code with encoding obfuscation. The string "New user" in the `hello("New user")` function from Listing 3.2 had been replaced to a call to a function that retrieves its value at runtime, such as `var _0x74f5`.

```
1 var _0x74f5=["\x48\x65\x6C\x6C\x6F\x2C\x20","\x6C\x6F\x67","\x4E\x65\x77\x20\x75\x73\x65\x72"];function hello(_0xe170x2){console[_0x74f5[1]](_0x74f5[0]+_0xe170x2)}hello(_0x74f5[2])
```

Listing 3.1: An obfuscated JS code representation using hexadecimal to implement encoding.

```
1 function hello(name){
2 console.log("Hello, " + name);
3 }
4 hello("New user");
```

Listing 3.2: The original JS code.

Fig. 3.1 shows the overview of the steps to deobfuscate, unpack, and decode an obfuscated JS code. It consists of three steps, namely, deobfuscation or formatting, unpacking, and decoding. A JS code beautifier and formatter are used to deobfuscate a JS code during deobfuscation to make it pretty, readable, and easier to edit and analyze. Then, the resulting formatted JS code is analyzed for obfuscation. If there is still evidence of obfuscation, a VM and a JS code editor are used to unpack the JS code by function redefinition. The unpacked JS code is further examined for the presence of encoding, and a `int_to_str()` decoding function is applied to realize a Plain-JS code.

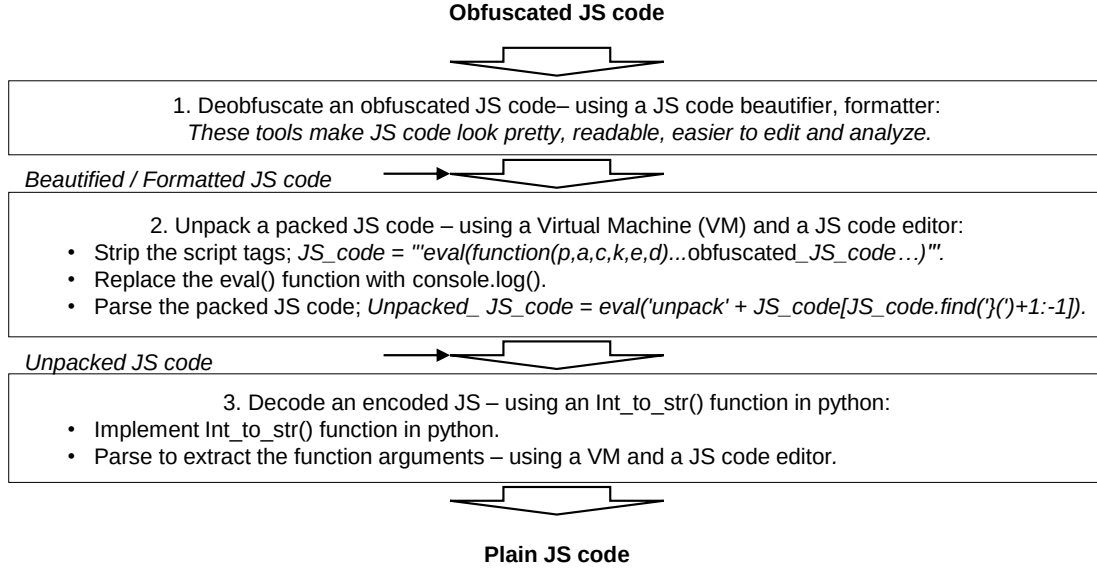


Figure 3.1: The layout for Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) of a JS code and the feature learning and classification for DUD-preprocessed AST-JS features.

3.2.2 Feature learning by the FastText model

Several models exist that learn continuous representations for a word, a sentence, and a paragraph in the Natural Language Processing (NLP) tasks. Models such as bag-of-words [144, 39], Word2Vec [84, 86], and Doc2Vec [67, 25] result in a unique vector for each word, thereby ignoring the internal structure of words. Besides, the produced vectors do not share parameters.

The FastText model [13, 55] is an approach based on the continuous skip-gram training algorithm [84]. The model represents each word as a bag of character n -grams. The resulting vector representation for a word is the sum of its character n -gram vectors considering subword information. The model is fast, therefore suitable for feature learning on big datasets, and can learn vector representation for out of vocabulary words. Fig. 3.2 shows the architecture of the FastText model for a sentence with N -gram features $\mathbf{x}_1, \dots, \mathbf{x}_N$. The features are embedded and averaged to form the hidden variable.

The Word2Vec [84] continuous skip-gram training algorithm learns a vector representation for each word x given a word vocabulary of size X , and each word

FastText Architecture

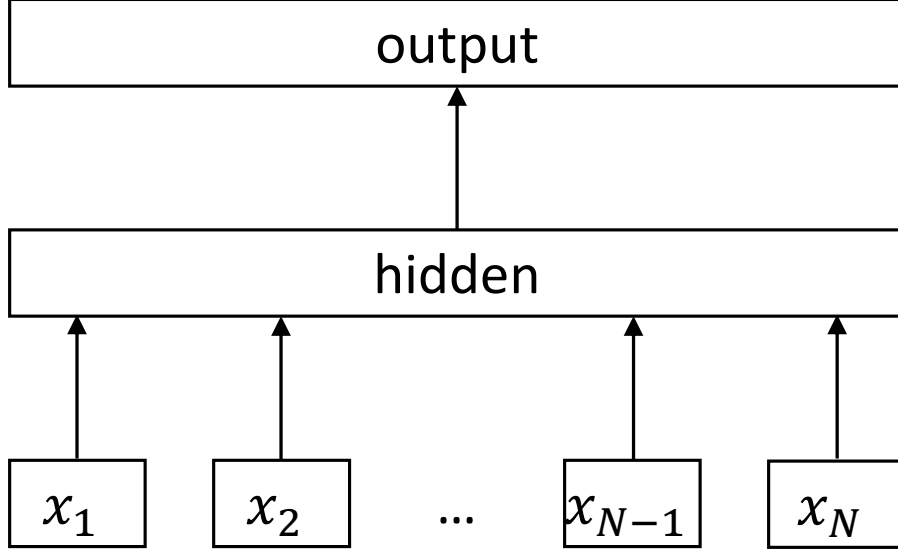


Figure 3.2: The FastText model architecture which shows the n -gram features, hidden variable and the output.

identified by index $x \in \{1, \dots, X\}$. The model predicts words in a context given a word. For a large training corpus $\mathbf{X}_c = \{\mathbf{x}_{t-2}, \mathbf{x}_{t-1}, \mathbf{x}_{t+1}, \mathbf{x}_{t+2}\}$, the model predicts the surrounding words, context $\mathbf{X}_c$ given the center word $\mathbf{x}_t$ represented by $\mathbf{X}_t$. The objective of the Word2Vec continuous skip-gram training algorithm is to maximize the log-likelihood:

$$\sum_{t=1}^T \sum_{c \in C_t} \log p(\mathbf{X}_c | \mathbf{X}_t), \quad (3.1)$$

Where C_t is the set of indices of the context $\mathbf{X}_c$. Given a scoring function f which maps pairs of $(\mathbf{X}_t, \mathbf{X}_c)$ to scores in $\mathbb{R}$, the softmax can be used to define the probability of a context word,

$$p(\mathbf{X}_c | \mathbf{X}_t) = \frac{e^{f(\mathbf{X}_t, \mathbf{X}_c)}}{\sum_{j=1}^X e^{f(\mathbf{X}_t, j)}}. \quad (3.2)$$

This type of model only predicts one context word $\mathbf{X}_c$ given the word $\mathbf{X}_t$. The task of predicting context words can be viewed as a set of independent binary classification to predict the presence or absence of $\mathbf{X}_c$. For a word $\mathbf{X}_t$, all $\mathbf{X}_c$ are the positive

examples, negative examples are sampled randomly from the dictionary. The binary logistic loss is used to obtain the negative log-likelihood for $\mathbf{X}_c$,

$$\log \left(1 + e^{-f(\mathbf{X}_t, \mathbf{X}_c)} \right) + \sum_{n \in N_{t,c}} \log \left(1 + e^{f(\mathbf{X}_t, n)} \right), \quad (3.3)$$

Where $N_{t,c}$ is a set of negative examples sampled from X . The objective function can be given by,

$$\sum_{t=1}^T \left[\sum_{c \in C_t} \ell(f(\mathbf{X}_t, \mathbf{X}_c)) + \sum_{n \in N_{t,c}} \ell(-f(\mathbf{X}_t, n)) \right], \quad (3.4)$$

by denoting the logistic loss function $\ell : x \mapsto \log(1 + e^{-x})$.

The FastText model introduces a different scoring function f where a bag of character n -gram vectors represent each word x . This scoring function takes into account the internal structure of a word. Prefixes and suffixes are distinguished using $<$ and $>$, special boundary symbols, added at the start and end of a word. For a word such as *encode* with $n = 3$, its character n -grams will be $< en, enc, nco, cod, ode, de >$ and $< encode >$ which is a special sequence that is used to learn the representation of each word. Given a dictionary of size G n -gram vectors and a word x , $G_x \subset \{1, \dots, G\}$ gives the set of n -gram vectors in x . The scoring function f is given by the sum of vector representations of a word's n -grams,

$$f(x, c) = \sum_{g \in G_x} \mathbf{Z}_g^\top \mathbf{X}_c, \quad (3.5)$$

where $\mathbf{Z}_g$ is vector representation for each n -gram g . The n -gram vector representations are shared across words. This ensures reliable vector representations are learnt for rare words [131]. A computation of the most similar word to *encode* using the trained FastText model, `model_JS.wv.most_similar("encode")`, results in `'encodedurl'`, 0.91, `'encodeuri'`, 0.89, `'htmlencode'`, 0.89, `'enc_str'`, 0.87 and `'decodeuri'`, 0.83. This function is used in malicious JS code to obfuscate the contents of a JS code. The words identified by the model are correct because each

encoding function is accompanied by a decoding function. The encoding function is used to encode a string or a URL in a JS code and the decoding function decodes it at runtime.

The FastText model implementation [109] creates n -gram vectors for words. A JS code feature vector is obtained by superimposing the n -gram word vectors. Given M as the total number of words in a JS code, the vector representation of a JS code will be given by,

$$(\mathbf{V}_{JS_code}) = \frac{\sum \mathbf{X}_n}{M}. \quad (3.6)$$

where $\mathbf{V}_{JS_code}$ is the vector representation for a JS code and $\mathbf{X}_n$ is the n -gram vector representation for a word n . A JS code vector is given by summing the average value of each component of a word n -gram vectors divided by the total number of words in a JS code.

An implementation [109] of the FastText model is used to learn the feature vectors for a JS code producing fixed-length vectors. Labels obtained from a JS code dataset are used to train the network for the prediction tasks. The vectors are then used to classify a JS code into either malicious or benign. The FastText model uses several parameters for the learning of feature vectors such as, *sg*, which represents the skip-gram training algorithm, *word_ngrams*, which enriches word vectors with subword (n -grams) information, and *min_n* and *max_n*, which control the minimum and maximum length of n -grams respectively. Other parameters include *min_count*, which ignore all words with a total frequency lower than a preset one, *size*, which represents the word vectors dimensionality, *iter* which represents the epochs over the corpus and *window*, which controls the maximum distance between the current word and the predicted one.

Fig. 3.3 shows the overview of DUD-preprocessing of a JS code. The figure also presents the feature learning process by the FastText model and the classification of the AST-JS features by a classifier model. It is a two-stage process. An obfuscated JS code is formatted, unpacked, and decoded during the first stage, and a JS code parser is used to perform syntactic analyses of the resultant Plain-JS code. Second,

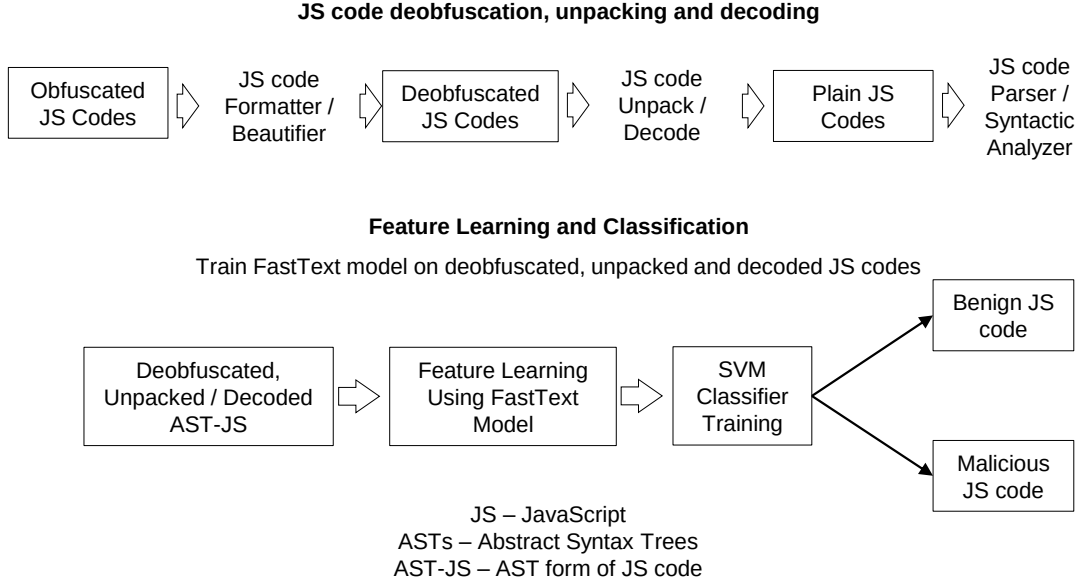


Figure 3.3: The layout for Deobfuscation, Unpacking, and Decoding (DUD-preprocessing) of a JS code and the feature learning and classification for DUD-preprocessed AST-JS features.

the AST-JS obtained from the JS code parser is passed to the FastText model for feature learning, and a classifier is used to judge the maliciousness of a JS code using the learned AST-JS feature vectors. Cross-validation and classifier learning is performed using a training set of the learned feature vectors to find the optimal hyperparameters. Then, the performance of the FastText model is compared to the Paragraph Vector models, the StarSpace model, the LSTM model, and the TF-IDF model on obfuscated malicious JS code detection on the same dataset.

3.3 Experiments

3.3.1 Experimental setup

A dataset of 40,000 malicious JS codes and 150,000 benign JS codes are used in the experimental setup. Also, 50,000 benign JS codes are added by crawling the internet. The malicious JS codes were obtained separately from the Hynek Petrak’s dataset [104] and the benign JS codes are from the SRILAB [111] and the Majestic Million service top 10,000 websites [124] datasets. These datasets are

used for both training and test purposes. As shown in Table 3.1, some malicious and benign JS codes in the dataset could not be successfully parsed because of *SyntaxError*, *UnicodeDecodeError*, *AttributeError*, *NameError*, *IndexError*, *NotImplementedError*, *KeyError* and *RecursionError*.

Table 3.1: JS code label, original JS codes and deobfuscated JS codes showing the success rate of the deobfuscation process.

JS Code Label	Original JS Codes	Deobfuscated JS Codes
Malicious	39,449	32,436
Benign	121,702	97,606

Two types of feature embedding are conducted: The FastText model for regular obfuscated JS codes and the FastText model for DUD-preprocessed JS codes. Feature learning is performed using the datasets for the FastText model and its performance compared to the Distributed Bag of Words version of the Paragraph Vector model (the PV-DBoW model), the Distributed Memory Model of the Paragraph Vector model (the PV-DM model), the StarSpace model, the LSTM model, and the TF-IDF model. First, feature learning is conducted using regular obfuscated JS codes with input defined as AST-JS, that is, by learning feature vectors from both obfuscated benign and malicious JS codes represented as an AST-JS. Second, feature learning is conducted using DUD-preprocessed JS codes with input defined as an AST-JS, that is, by performing feature learning on code structure representation for DUD-preprocessed JS codes. AST-level merging and AST sub-tree realignment [93] is also performed on the AST-JS to enhance feature learning. This AST-level manipulation facilitates simulation of an actual web setting by ensuring that the ratio of the benign AST-JS features is more compared to that of the malicious ones.

The FastText model parameters such as *sg*, *size*, *window*, *min_count*, *word_ngrams*, *min_n* and *max_n* significantly affects the performance of the FastText model. Experiments are conducted with the dimensions of the range 100-900 for *size*, 1-8 for *window*, 1-8 for *min_count* and 3-6 for *min_n* and *max_n*. Feature vectors of 200 dimensions are learned for the word vectors, with *sg* = 1

to select the skip-gram training algorithm, $window = 8$ for the number of context words, $min_count = 5$ for the word frequency, $word_ngrams = 1$ to activate the use of sub-word information and $min_n = 3$ and $max_n = 5$ for n -gram minimum and maximum length respectively. These are the parameters that were found optimal for the task of feature learning on DUD-preprocessed JS codes, for the FastText model. For performance evaluation, cross-validation is used with $k = 10$, to get 10-folds of feature vectors. The folds translates to ten iterations, divided into nine folds for classifier training and one fold for model evaluation. For feature vectors classification, SVM is used with $kernel = linear$ and $C = 1$ parameters. SVM is a classification method for regression and classification. The algorithm is mostly used as a binary classifier and it constructs a hyper-plane in high dimensional space [15, 102]. For this, precision, recall and F1-score were computed. For performance visualization, a Receiver Operating Characteristic (ROC) graph was used and calculated Area Under the Curve (AUC) score for the ROC to determine the model performance. Performance of the approach was compared to other feature learning approaches: the PV-DBoW model ($vector_size = 200$, $window = 8$, $min_count = 5$), the PV-DM model ($dm = 1$, $vector_size = 200$, $window = 8$, $min_count = 5$), the StarSpace model, the LSTM model and the TF-IDF model.

3.3.2 Performance Comparisons

Precision, Recall, and F1-score The results of the experiments are presented in this section. Table 3.2 presents the performance of the FastText model, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model when performing feature embedding for regular obfuscated JS codes with input defined as AST-JS similar to work by [93]. As shown in the table, the models can detect regular obfuscated malicious JS codes with an error rate of approximately 0.0489, 0.0717, 0.0649, 0.0892, 0.1205, and 0.1657 in recall for the FastText model, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model respectively. AST can bypass

obfuscation. Therefore this approach can detect regular obfuscated JS codes with similar syntactic and semantic characteristics as the original ones at an abstract level. However, malicious JS codes employ methods such as packing, encoding, and multi-layered obfuscation methods, which results in complex codes that hinder the detection of JS-based attacks by signature, heuristics, and machine learning approaches. The performance of Paragraph Vector models in Table 3.2 is far from perfect as this would result in many misclassified obfuscated malicious JS codes, which translates to multiple successful attacks.

Table 3.3 presents precision, recall, and F1-score for the FastText model, the Paragraph Vector models, the StarSpace model, the LSTM model, and the TF-IDF model trained using a deobfuscated JS code dataset with input defined as an AST-JS and includes the standard deviation. The models achieved an above-average performance with precision, recall, and F1-score being above 50 percent. The FastText model with $sg = 1$ and 200 dimensions of word vectors achieved superior results compared to the Paragraph Vector models. The FastText model outperformed the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model and the TF-IDF model with 0.0110, 0.0163, 0.0228, 0.0538 and 0.0963 in precision, 0.0041, 0.0099, 0.0502, 0.0769 and 0.1078 in recall and 0.0056, 0.0108, 0.0373, 0.0852

Table 3.2: Precision, Recall, and F1-score representing the performance of the FastText model, the Paragraph Vector models, the StarSpace model, the LSTM model, and the TF-IDF model on a regular obfuscated JS code dataset. The PV-DBOW model stands for the Distributed Bag of Words version of Paragraph Vector. The PV-DM model stands for the Distributed Memory Model of Paragraph Vectors. The LSTM model stands for the Long Short-Term Memory model. The TF-IDF model stands for Term Frequency–Inverse Document Frequency model.

Model	Precision	Recall	F1-score
FastText	0.9427 ($\pm$ 0.1005)	0.9511 ($\pm$ 0.0477)	0.9459 ($\pm$ 0.0528)
PV-DBoW	0.9412 ($\pm$ 0.1571)	0.9283 ($\pm$ 0.1159)	0.9331 ($\pm$ 0.0868)
PV-DM	0.9313 ($\pm$ 0.1675)	0.9351 ($\pm$ 0.1016)	0.9289 ($\pm$ 0.0873)
StarSpace	0.9160 ($\pm$ 0.0192)	0.9108 ($\pm$ 0.0193)	0.9098 ($\pm$ 0.0121)
LSTM	0.8601 ($\pm$ 0.0155)	0.8795 ($\pm$ 0.0421)	0.8552 ($\pm$ 0.1082)
TF-IDF	0.8471 ($\pm$ 0.0112)	0.8343 ($\pm$ 0.0521)	0.8317 ($\pm$ 0.1325)

and 0.1124 in F1-score, respectively. Compared to the other models, the TF-IDF model achieved the lowest performance for the task of obfuscated malicious JS code detection while applying JS code formatting. Consequently, the FastText model resulted in fewer false positives and false negatives compared to the other models.

This study aims to improve the performance of machine learning models on the detection of obfuscated malicious JS codes. A model with a high recall rate is desired to achieve this objective. Such a model would result in a few misclassified obfuscated malicious JS codes, that is, low false-negative. JS code deobfuscation results in a formatted code and can do away with only the simple forms of obfuscation. This property explains the reason why there is only a slight improvement between the models trained using regular obfuscated JS codes and those trained using a de-obfuscated JS code dataset of approximately 0.0115 for the FastText model, 0.0302 for the PV-DBoW model, 0.0176 for the PV-DM model, 0.0016 for the StarSpace model, 0.0062 for the LSTM model and 0.0205 for the TF-IDF model in the recall. The input for the models in both cases is defined as an AST-JS. The best-performing model when using a deobfuscated JS code dataset, the FastText model, has an error rate of approximately 0.0374 in the recall. Even though the FastText model is the best performing model here, many obfuscated malicious JS codes will go undetected, which translates to multiple successful attacks. The failure in detection is because sections of the obfuscated malicious JS codes that employ sophisticated obfuscation techniques cannot be easily solved by deobfuscation. Therefore, further processing of the deobfuscated malicious JS codes is needed to realize improved performance.

Table 3.4 presents precision, recall, and F1-score for the FastText model, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model obtained with AST-JS features by feature learning using an unpacked JS code dataset. The FastText model achieves an improvement of over 2.78% and 2.07% in precision, 2.09% and 0.94% in the recall, and 2.54% and 1.82% in F1-score, when compared to the same model, trained using a regular obfuscated JS code dataset and a deobfuscated JS code dataset, respectively. The resultant code

structure can explain the improvement in performance for the FastText model after deobfuscation and unpacking. Deobfuscation results in a well-formatted JS code, which facilitates the feature learning process of the model. When the unpacking is done for the deobfuscated JS code dataset, the complex code structure remaining after deobfuscation is simplified away. Therefore, the model can learn better features from a deobfuscated and unpacked JS code dataset than a regular obfuscated or a deobfuscated JS code.

Table 3.5 presents precision, recall, and F1-score for the FastText model, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the

Table 3.3: Precision, recall, and F1-score obtained by feature learning using a deobfuscated JS code dataset. The PV-DBOW model stands for the Distributed Bag of Words version of Paragraph Vector, the PV-DM model stands for the Distributed Memory Model of Paragraph Vectors, the LSTM model stands for the Long Short-Term Memory model and the TF-IDF model stands for Term Frequency–Inverse Document Frequency model.

Model	Precision	Recall	F1-score
FastText	0.9498 ($\pm$ 0.0270)	0.9626 ($\pm$ 0.0197)	0.9531 ($\pm$ 0.0288)
PV-DBoW	0.9388 ($\pm$ 0.0498)	0.9585 ($\pm$ 0.0529)	0.9475 ($\pm$ 0.0402)
PV-DM	0.9335 ($\pm$ 0.0372)	0.9527 ($\pm$ 0.0407)	0.9423 ($\pm$ 0.0272)
StarSpace	0.9270 ($\pm$ 0.0216)	0.9124 ($\pm$ 0.0537)	0.9158 ($\pm$ 0.0410)
LSTM	0.8960 ($\pm$ 0.0441)	0.8857 ($\pm$ 0.0386)	0.8679 ($\pm$ 0.0314)
TF-IDF	0.8535 ($\pm$ 0.0110)	0.8548 ($\pm$ 0.0193)	0.8407 ($\pm$ 0.0650)

Table 3.4: Precision, Recall, and F1-score obtained by feature learning using an unpacked JS code dataset. The PV-DBOW model stands for the Distributed Bag of Words version of Paragraph Vector, the PV-DM model stands for the Distributed Memory Model of Paragraph Vectors, the LSTM model stands for the Long Short-Term Memory model and the TF-IDF model stands for Term Frequency–Inverse Document Frequency model.

Model	Precision	Recall	F1-score
FastText	0.9705 ($\pm$ 0.0092)	0.9720 ($\pm$ 0.0257)	0.9713 ($\pm$ 0.0235)
PV-DBoW	0.9617 ($\pm$ 0.0087)	0.9574 ($\pm$ 0.0411)	0.9592 ($\pm$ 0.0224)
PV-DM	0.9474 ($\pm$ 0.0399)	0.9468 ($\pm$ 0.0457)	0.9460 ($\pm$ 0.0276)
StarSpace	0.9363 ($\pm$ 0.0339)	0.9226 ($\pm$ 0.0366)	0.9460 ($\pm$ 0.0542)
LSTM	0.9097 ($\pm$ 0.0270)	0.9082 ($\pm$ 0.0444)	0.9178 ($\pm$ 0.0457)
TF-IDF	0.8858 ($\pm$ 0.0237)	0.8729 ($\pm$ 0.0488)	0.8691 ($\pm$ 0.0336)

TF-IDF model obtained with AST-JS features by feature learning using a DUD-preprocessed JS code dataset. The FastText model, the best performing model, achieves an improvement of over 5.21%, 4.5%, and 2.43% in precision, 4.2%, 3.05%, and 2.11% in recall and 4.14%, 3.42%, and 1.6% in F1-score, when compared to the same model, trained using a regular obfuscated JS code dataset, a deobfuscated JS code dataset, and an unpacked JS code dataset respectively. The performance for the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model in this task is also improved. However, the FastText model achieves the highest recall rate for the task of obfuscated malicious JS code detection. This performance improvement is because deobfuscation, unpacking, and decoding results in a Plain-JS code with JS code transformations simplified away. Therefore, the resultant JS code provides better AST-JS features during feature learning.

True positive and false positive rate The improved performance of the FastText model on the task of detection of obfuscated JS codes is also shown in Fig. 3.4a, which presents the ROC graph and AUC results for the model. As shown in Fig. 3.4a, the model achieved high and low rates of true positives and false positives, respectively, compared to the PV-DBoW model, which is presented in Fig. 3.4b, the PV-DM model, which is presented in Fig. 3.4c, the StarSpace model, which

Table 3.5: Precision, Recall and F1-score obtained by feature learning using a de-obfuscated, unpacked and decoded (DUD-preprocessed) JS code dataset. The PV-DBOW model stands for the Distributed Bag of Words version of Paragraph Vector, the PV-DM model stands for the Distributed Memory Model of Paragraph Vectors, the LSTM model stands for the Long Short-Term Memory model and the TF-IDF model stands for Term Frequency–Inverse Document Frequency model.

Model	Precision	Recall	F1-score
FastText	0.9948 ($\pm$ 0.0045)	0.9931 ($\pm$ 0.0070)	0.9873 ($\pm$ 0.0152)
PV-DBoW	0.9839 ($\pm$ 0.0046)	0.9841 ($\pm$ 0.0053)	0.9801 ($\pm$ 0.0060)
PV-DM	0.9837 ($\pm$ 0.0084)	0.9802 ($\pm$ 0.0121)	0.9819 ($\pm$ 0.0132)
StarSpace	0.9571 ($\pm$ 0.0199)	0.9584 ($\pm$ 0.0204)	0.9628 ($\pm$ 0.0161)
LSTM	0.9170 ($\pm$ 0.0297)	0.9332 ($\pm$ 0.0360)	0.9234 ($\pm$ 0.0327)
TF-IDF	0.9021 ($\pm$ 0.0352)	0.9009 ($\pm$ 0.0266)	0.9042 ($\pm$ 0.0366)

is presented in Fig. 3.4d, the LSTM model, which is presented in Fig. 3.4e and the TF-IDF model, which is presented in Fig. 3.4f. The PV-DBoW model and the PV-DM model achieved the same AUC score. However, the latter model achieved the least AUC score compared to the PV-DBoW during cross-validation, as shown by sections of its ROC curve. The StarSpace model, the LSTM model, and the TF-IDF model achieved the lowest AUC scores with sections of the curves in Fig. 3.4d, Fig. 3.4e and Fig. 3.4f going below the performance threshold. The colored curves represent the performance of the model for each fold during cross-validation. The Paragraph Vector models: the PV-DBoW and the PV-DM models ignore the internal structure of words. The FastText model takes into account a word’s sub-word information. The n -gram word vectors learned by the FastText model during feature learning are shared across words. This property enables the FastText model to learn better and reliable vector representations for the DUD-preprocessed JS code dataset. Using a trained FastText model, words that do not exist in the model’s vocabulary can be inferred from the learned character n -gram vectors. The model also results in low dimensional representations for the learned n -gram word vectors.

3.4 Conclusion

In this Chapter, DUD-preprocessing of obfuscated malicious JS codes and feature learning using the FastText model is proposed. The NLP models have previously achieved good results and resulted in high-quality vector representations for words, sentences, and paragraphs. Consequently, the models have been used for malicious JS code detection using features such as Plain-JS and AST-JS. However, the models struggle to learn useful representations when faced with obfuscated JS codes as the JS codes or sections of the JS code do not maintain appropriate code structure. One expectation is that DUD-preprocessing of the obfuscated JS codes would result in improved performance for machine learning models for the task of obfuscated malicious JS code detection and enable detection of JS codes transformed using sophisticated obfuscation techniques. Experiments were conducted using feature

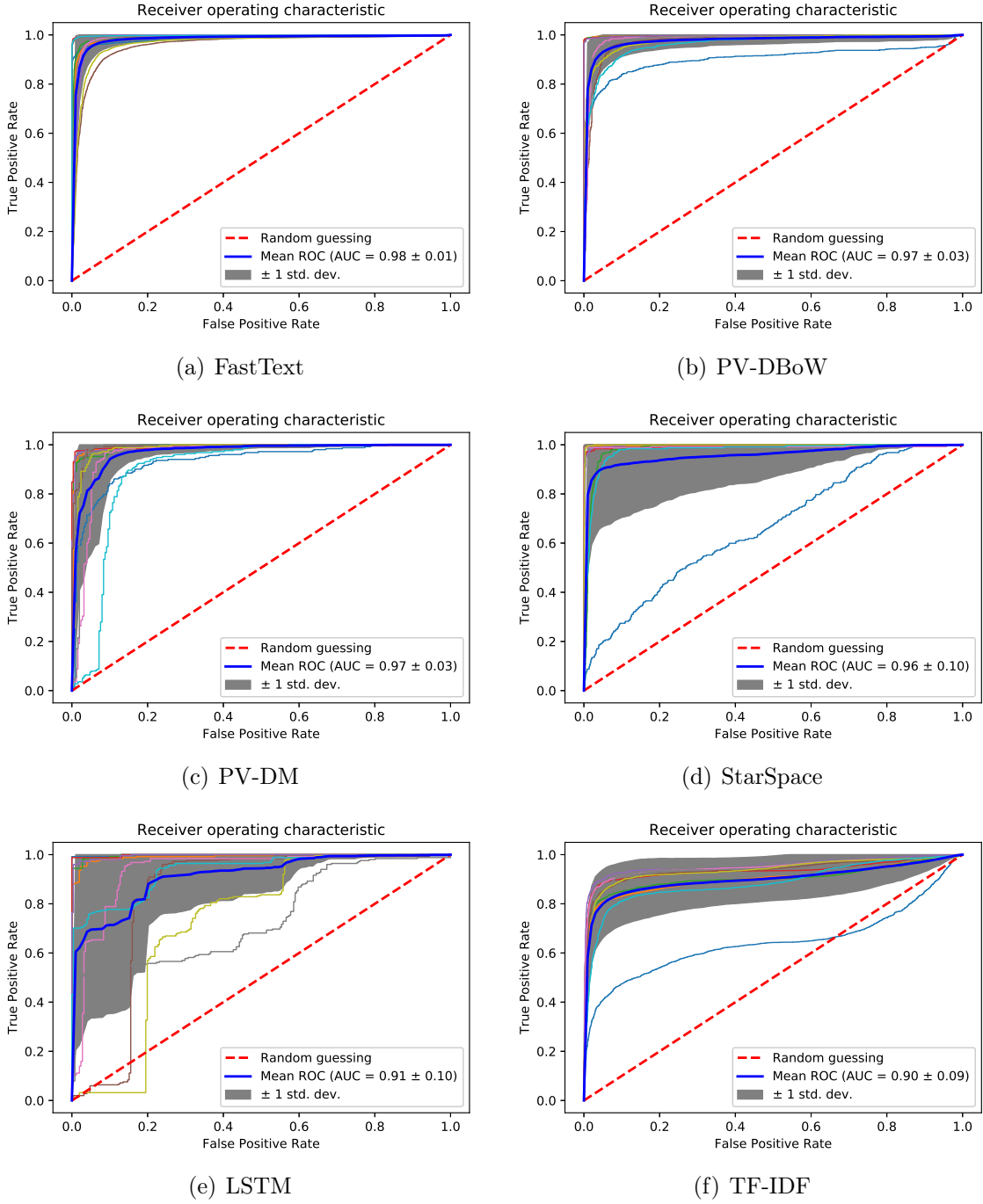


Figure 3.4: ROC curves and AUC that represent the performance of the FastText, the PV-DBoW model, the PV-DM model, the StarSpace model, the LSTM model and the TF-IDF model on obfuscated malicious JS codes detection with the true positive rate plotted against the false positive rate. The dashed diagonal line represents the performance threshold as random guessing. The colored curves represent the performance of the each model for each fold during cross-validation. Also included is the mean AUC score for each model.

embedding for regular obfuscated JS codes and DUD-preprocessed JS codes. Performance of the FastText model is compared to the Paragraph Vector models: the

PV-DBoW model and the PV-DM model, the StarSpace model, the LSTM model, and the TF-IDF model on the detection of obfuscated malicious JS codes. The resultant JS code after DUD-preprocessing is analyzed to check for the presence of obfuscation. This further processing is essential to simplify away sophisticated JS code obfuscation techniques such as JS code multi-layer obfuscations. AST-JS is used for code structure representation of the resultant Plain-JS code and realizes an improvement in performance for the models used for feature learning. AST-JS makes the structural relationship of elements within a JS code distinct while at the same time preserving all the relevant information of a JS code. The proposed approach provides improved detection performance for obfuscated malicious JS code contents detection than feature learning on regular obfuscated JS code. Besides, there is considerable performance improvement compared to the previous work [93]. The FastText model is fast and suitable for big datasets and, therefore, well suited for the JS code feature learning. The model can also learn vector representations for JS code words that are not contained in the vocabulary at the time of model training and, therefore, result in reliable vector representations for future, new and unseen JS codes.

The proposed DUD-preprocessing approach is not perfect, and the performance is only shown for the successfully deobfuscated JS in Table 3.5. That is, it is a gross performance. The success rate of deobfuscation should be considered as the NET performance. Table 3.6 presents the gross and net performance for the FastText model while considering the success rate of deobfuscation. As shown in the table, further work is necessary further to improve the detection of obfuscated malicious JS codes.

Table 3.6: The gross and net performance for the FastText model while considering the success rate of deobfuscation.

Metric	Gross performance	Net performance
Precision	0.9948	0.7015
Recall	0.9931	0.8176
F1-score	0.9873	0.7551

As future work, there is a need for a study on obfuscated URLs and their properties. URLs in JS codes are heavily obfuscated. The obfuscation is even more complicated for URLs in obfuscated malicious JS codes as attackers endeavor to hide their attack methods such as URL redirection attacks to attacker's untrusted websites. Such websites employ insecure protocols that would be effortlessly flagged down by implemented security controls such as the firewall. URL obfuscation is stealthily and deliberately applied to evade such detection. A malicious URL detection system must be developed to realize a comprehensive security system against web-based attacks.

Chapter 4

Web-based Attack Detection by Feature Selection using Tree SHapley Additive exPlanations

4.1 Introduction

When machine learning is employed to detect malicious URLs and their JS content, certain considerations have to be made, such as privacy and safety concerns, feature learning methods, and the development of a fully representative feature set.

Given the common vulnerabilities, characteristics, and features employed by malicious JS codes, it is evident that malicious websites are different from benign ones as they each have different objectives. For example, malicious JS code contains functions or a combination of functions such as *document.write*, *location.replace*, and *document.getElementById*. Among other things, these are used to perform Cross-site scripting (XSS) attacks by injecting malicious JS into the Document Object Model (DOM), direct concatenation of user input with the query string, and building of database query via string concatenation used to perform SQL injections. Therefore, developing a malicious JS detection system that can generalize well to detect different attack types is imperative. Automatic feature selection is proposed

for AST-JS features, ensuring that the global and local features that contribute to the model performance are captured. Adequate details of a JS code structure are also presented without executing the code.

The JS code preprocessing and the importance of the SHAP feature used for feature selection are explained in the following sections.

4.2 Proposed method

4.2.1 Preprocessing

AST-JS node sets and patterns Preprocessing is essential to enhance the feature learning process. To define the first input set, each JS code in the dataset was parsed to obtain the *expression*, *pattern*, *statement*, and *declaration* AST-JS nodes.

Malicious JS rarely uses raw values as attackers endeavor to evade static analyzers. Therefore, it is essential to capture features that employ covert implementations to hide their values. The feature importance of the first feature set was analyzed to capture these features using SHAP interaction values. This feature analysis exposed each AST-JS node’s contribution and their combinations to the malicious JS detection model as output. Sample JS attack codes were obtained, and association rule mining was performed using a Frequent Pattern (FP) growth mlxtend library [47, 44, 110] to identify such combinations. Six of these sample JS attack codes are explained next.

EmptyStatement_ExpressionStatement_CallExpression_Identifier

Listing 4.1 shows a JS code that lacks a definition of the index of an object in the current scope. When *foo()* is called, *x = 0* is passed, which is a local index of the function. This attack pattern is captured using the AST node *type EmptyStatement_ExpressionStatement_CallExpression_Identifier*.

```
1 function foo (x){
```

```

2 var arr = ["1", "2", "3"];
3 eval (arr [x])
4 };
5 foo (0);
6 // AST for EmptyStatement\_ExpressionStatement\_CallExpression\_
  _Identifier
7 "type": "EmptyStatement"
8 },
9 {
10 "type": "ExpressionStatement",
11 "expression": {
12 "type": "CallExpression",
13 "callee": {
14 "type": "Identifier",
15 "name": "foo"
16 },
17 "arguments": [
18 {
19 "type": "Literal",

```

Listing 4.1: A snippet of a JS code with EmptyStatement_ExpressionStatement_CallExpression_Identifier.

CallExpression_Identifier_BinaryExpression Listing 4.2 shows a JS code, where the argument of *setTimeout()* is a string concatenation with binary expression. This attack pattern is captured using the AST node *type CallExpression_Identifier_BinaryExpression*.

```

1 function startOverflow (num) {
2 }
3 if (num == 255) setTimeout ("startOverflow (" + (num + 1) + ")",
  2000);
4 // AST for CallExpression\_Identifier\_BinaryExpression
5 "type": "ExpressionStatement",
6 "expression": {
7 "type": "CallExpression",

```

```

8  "callee": {
9  "type": "Identifier",
10 "name": "setTimeout"
11 },
12 "arguments": [
13 {
14 "type": "BinaryExpression",
15 "operator": "+",
16 "left": {
17 "type": "BinaryExpression",
18 "operator": "+",

```

Listing 4.2: A snippet of a JS code with `CallExpression_Identifier_BinaryExpression`.

CallExpression_Identifier_Literal_CallExpression Listing 4.3 shows nested JS function calls, where an attacker hides the function arguments in another function call to evade static analyses. This attack pattern is captured using the AST node `-type CallExpression_Identifier_Literal_CallExpression`.

```

1  var p26 = new Array ();
2  function f85 (g55 , w27) {
3  p26 [g55] = w27;
4  };
5  function w25 (h51) {
6  return h51;
7  };
8  f85 (763 , w25 ('s'));
9  // AST for CallExpression_Identifier_Literal_CallExpression
10 "type": "ExpressionStatement",
11 "expression": {
12 "type": "CallExpression",
13 "callee": {
14 "type": "Identifier",
15 "name": "f85"
16 },

```

```

17 "arguments": [
18 {
19   "type": "Literal",
20   "value": 763,
21   "raw": "763"
22 },
23 {
24   "type": "CallExpression",

```

Listing 4.3: A snippet of a JS code with `CallExpression_Identifier_Literal_CallExpression`.

AssignmentExpression_Identifier_FunctionExpression Listing 4.4 shows variable declarations, where e in *line2* is assigned to a function c . This attack pattern is captured using the AST node *type* `AssignmentExpression_Identifier_FunctionExpression`.

```

1 eval (function (p, a, c, k, e, d) {
2   e = function (c) {
3     return (c < a ? '' : e(parseInt (c / a))) + ((c = c % a) > 35 ?
       String.fromCharCode (c + 29) : c.toString (36) ));
4   if (! ''. replace (/~/ , String )) {
5     while (c --) d[e(c)] = k[c] || e(c);
6   }
7 })
8 // AST for AssignmentExpression\_Identifier\_FunctionExpression
9 "type": "ExpressionStatement",
10 "expression": {
11   "type": "AssignmentExpression",
12   "operator": "=",
13   "left": {
14     "type": "Identifier",
15     "name": "e"
16   },
17   "right": {

```

```
18 "type": "FunctionExpression",
```

Listing 4.4: A snippet of a JS code with AssignmentExpression_Identifier_FunctionExpression.

VariableDeclarator_Identifier_LogicalExpression Listing 4.5 shows a JS code where the attacker attempts to detect the user environment, such as the browser version or operating system. This attack pattern is captured using the AST node *type VariableDeclarator_Identifier_LogicalExpression*.

```
1 var b = navigator.userAgent.match(/iPhone OS ([\ d_]+) /) ||
2 navigator.userAgent.match(/iPad OS ([\ d_]+) /) ||
3 navigator.userAgent.match(/CPU OS ([\ d_]+) /);
4 // AST for VariableDeclarator\ _Identifier\ _LogicalExpression
5 "type": "VariableDeclaration",
6 "declarations": [
7 {
8 "type": "VariableDeclarator",
9 "id": {
10 "type": "Identifier",
11 "name": "b"
12 },
13 "init": {
14 "type": "LogicalExpression",
```

Listing 4.5: A snippet of a JS code with VariableDeclarator_Identifier_LogicalExpression.

BinaryExpression_BinaryExpression Listing 4.6 shows the JS code sample string concatenation and function obfuscation codes. This attack pattern is captured using the AST node *type BinaryExpression_BinaryExpression*.

```
1 // String concatenation
2 var a = "He" + "ll" + "o";
3 var b = " World !";
4 var c = a + " " + b;
```

```

5 // Function obfuscation
6 var x = eval;
7 var y = x("do"+"cu"+"ment");
8 // AST for BinaryExpression\_BinaryExpression
9 "type": "BinaryExpression",
10 "operator": "+",
11 "left": {
12 "type": "BinaryExpression",
13 "operator": "+",
14 "left": {

```

Listing 4.6: A snippet of a JS code with BinaryExpression_BinaryExpression.

Algorithm 1 shows the association rule mining procedure for AST-JS using the FP-growth algorithm. The generated association rules for benign and malicious JS are transformed into a list (lines 14-17). The lists are then converted into individual benign and malicious data frames (lines 19-22) for AST-JS feature selection.

Finally, the feature importance of each AST-JS feature selected was analyzed using SHAP values to gain an understanding of how different features influence the malicious JS detection model’s performance. This analysis was achieved in two parts: first, using the original AST-JS features, and then using the AST-JS combination features extracted through association rule mining. Next, the SHAP feature importance is explained.

4.2.2 SHapley Additive exPlanations (SHAP) Feature Importance

SHAP is a game-theoretic approach that explains any machine learning model’s output that connects optimal credit allocation with local explanations using the classical Shapley values and their related extensions [75, 73, 74]. The values show the extent to which a feature is responsible for a change in the model output. SHAP values either increase or decrease the model prediction and balance each other out at the input’s actual prediction. The prediction starts from the base value, which

Algorithm 4 Mining frequent AST-JS node sets using the FP-growth algorithm.

Require: D - a database of benign and malicious JS codes defined as AST-JS nodes;
 $min_support$ - the minimum support count threshold.

Ensure: Benign and malicious DataFrames of AST-JS nodes and node combinations.

```
1: if  $Tree$  contains a single path  $P$  then
2:   for each combination  $\beta$  of the nodes in the path  $P$  do
3:     generate pattern  $\beta \cup \alpha$  with  $support = min\_support$  of nodes in  $\beta$ ;
4:   end for
5: else
6:   for each  $a_i$  in the Tree header do
7:     generate pattern  $\beta = a_i \cup \alpha$  with  $support = a_i.support$ ;
8:     construct  $\beta$ 's conditional pattern base and then  $\beta$ 's conditional FP_tree  $Tree_\beta$ ;
9:   end for
10:  if  $Tree_\beta \neq 0$  then
11:    call FP_growth( $Tree_\beta, \beta$ );
12:  end if
13: end if
14: for each benign 0 and malicious 1 frequent patterns do
15:   0.antecedents.apply(sorted(list(i)))+0.consequents.apply(sorted(list(i)));
16:   1.antecedents.apply(sorted(list(i)))+1.consequents.apply(sorted(list(i)));
17:   return list0, list1
18: end for
19: for each benign list0 and malicious list1 frequent patterns do
20:   TransactionEncode.fit(list0).transform(list0);
21:   TransactionEncode.fit(list1).transform(list1);
22:   pd.DataFrame(list0, list1, columns=TransactionEncode.columns_)
23: end for
```

is the mean derived from the entire prediction. The premise is that the ability to interpret the malicious JS detection model output will provide a direction for AST-JS feature selection, further improving its performance. Identifying the features with the most impact may enable derivation of other features with additional information to improve the detection performance. This knowledge will also provide necessary insights into the distribution of specific benign and malicious JS features.

A JS code input is represented as Z with a set of AST-JS features $z_1, z_2, \dots, z_n$, and its corresponding output Z' and predicted features $z'_1, z'_2, \dots, z'_n$, using the tree ensemble model g . Algorithm 2 shows the procedure for calculating tree SHAP values using tree ensemble methods that return AST-JS features (line 6) sorted in descending order of their importance. First, AST-JS features were obtained from the JS code dataset and saved as AST-JS M features. For each feature z'_i in the AST-JS M features, tree SHAP was used to obtain the SHAP values. When predicting

Algorithm 5 Calculating Tree SHAP values for AST-JS M features

Require: Z - a malicious or benign JS code to be explained, $Z_{1..j}$ - AST-JS instances that Tree SHAP uses as background examples, the g - tree ensemble model.

Ensure: $SHAPsortedM$ features - SHAP values for each AST-JS feature in JS dataset sorted in descending order of their importance.

- 1: M features $\leftarrow$ feature values based on AST-JS.
 - 2: **for each** $i \in M$ features **do**
 - 3: explainer $\leftarrow$ shap.TreeExplainer($g, Z_{1..j}$)
 - 4: $SHAPsortedM$ features[i] $\leftarrow$ explainer.shapvalues(Z, i)
 - 5: **end for**
 - 6: return $SHAPsortedM$ features
-

Algorithm 6 $g(Z, i)$ - predicting the i 'th AST-JS feature

Require: Z - a malicious or benign JS code to explain, i - the AST-JS feature to get prediction for, g - tree ensemble model.

Ensure: z'_i - the value of the i 'th AST-JS feature in Z' .

- 1: $z'_i \leftarrow g.predict(Z)[i]$
 - 2: return z'_i
-

the observed feature z'_i , the SHAP importance values for each feature, $z_1, z_2, \dots, z_n$, excluding z_i , was calculated. Tree SHAP receives g and a background set with j instances to build the local explanation model and calculate the SHAP values. Then, g takes Z and i as input and predicts Z' ; the value in the i -th feature, a feature in the AST-JS M features, is returned by Algorithm 3. This results in a two-dimensional list of the $SHAPsortedM$ features. Each row in the list represents the SHAP values one of each AST-JS feature in M features.

AST-JS features were divided into those that enhanced the performance of the malicious JS detection model and those that did not by determining whether their SHAP values moved the predicted value toward or away from the true value, respectively. Algorithm 4 shows how the SHAP values were used to select relevant AST-JS features. For each AST-JS feature (line 1), it was checked whether the true feature value given by the input JS code was greater than the predicted value (line 2); a positive SHAP value would indicate contributing features (line 3) and vice versa in (line 4). If the predicted feature value were less than the input value for a JS code (line 5), the contributing features would be indicated by a negative SHAP value and vice versa. This algorithm returns two lists, the $SHAPSelected$ and $SHAPnotSelected$ features, containing the contributing and non-contributing

Algorithm 7 Selecting AST-JS features SHAP values contributing to the malicious JS detection model.

Require: $SHAPsortedM$ features - SHAP values for each AST-JS feature in JS dataset sorted in descending order of their importance, Z - a malicious or benign JS code to be explained, Z' - the prediction for Z .

Ensure: $SHAPSelected$ features and $SHAPnotSelected$ features.

```

1: for each  $i \in SHAPsortedM$  features do
2:   if  $z_i > z'_i$  then
3:      $SHAPSelected[i] \leftarrow SHAPsortedM \text{ features}[i] > 0$ 
4:      $SHAPnotSelected[i] \leftarrow SHAPsortedM \text{ features}[i] < 0$ 
5:   else
6:      $SHAPSelected[i] \leftarrow SHAPsortedM \text{ features}[i] < 0$ 
7:      $SHAPnotSelected[i] \leftarrow SHAPsortedM \text{ features}[i] > 0$ 
8:   end if
9: end for
10: return  $SHAPSelected$  features,  $SHAPnotSelected$  features

```

AST-JS features, respectively, along with the SHAP values for each of the AST-JS M features.

The benefits of computing SHAP values are global and local interpretability, which shows the manner and degree each AST-JS feature contributes to the prediction, and each observation obtains its own set of SHAP values enabling the evaluation of the factors' impact.

4.3 Experiments

4.3.1 Experimental Setup

A dataset of 39,443 malicious and 40,000 benign JS codes was used in the experimental setup. The malicious JS codes were obtained from Hynek Petrak's dataset [105], and the benign codes were obtained from the Majestic Million Service [125]. A JS parser was used to preprocess the dataset into AST-JS nodes, resulting in 32,430 malicious and 38,891 benign JS codes as shown in Table 4.1. This resulted in 25 *expression* and *pattern* and 23 *statement* and *declaration* AST-JS features.

Next, each AST-JS feature's contribution to the malicious JS detection model's performance was visualized and investigated using SHAP values. This analysis indicated the need to define more concrete features. Using sample JS attack codes,

Table 4.1: The dataset statistics showing the number of malicious and benign JS codes before and after preprocessing.

JS Label	Before preprocessing	After preprocessing
Malicious	39,443	32,430
Benign	40,000	38,891

ten more AST-JS features were added based on combinations of the original ones. Then, association rule mining and a confidence metric were used to measure how often AST-JS nodes appear together in benign and malicious JS codes. This resulted in 33 features with parameters $min_support = 0.4$ for malicious JS codes, $min_support = 0.53$ for benign codes, and $confidence's\ min_threshold = 1$ for both benign and malicious codes. This selection reduced the original features and formed the second input set. Finally, a final set of AST-JS features is selected using the SHAP values. This selection resulted in forty-four features, comprising thirty-four AS-JS nodes and ten node combinations as shown in Table 4.2.

Performance is evaluated using the three-input feature sets, i.e., the 48 AST-JS features obtained after the initial preprocessing, the 33 features obtained using association rule mining, and the 44 features obtained by feature selection using the SHAP value.

The parameters for the two best-performing models, XGBClassifier and LGBMClassifier, using the three AST-JS feature sets, are listed in Table 4.3.

These parameters were found optimal for malicious JS content detection using the three-input feature sets. 10-fold cross-validation was performed to evaluate the performance of the approach. Precision, recall, and F1-score were computed.

4.3.2 Performance Comparisons

Precision, Recall, and F1-score The results of these experiments are presented in this section. Table 4.4 presents the precision, recall, and F1-score obtained using features defined as AST-JS nodes and 10-fold cross-validation. Each model performed well on the recall metric, especially the tree ensemble methods. The

Table 4.2: The AST-JS nodes and node combination features selected using SHapley Additive ex-Planations (SHAP).

AST-JS nodes	AST-JS node combinations
ArrayExpression	AssignmentExpression_Identifier_FunctionExpression
AssignmentExpression	AssignmentExpression_Identifier_ThisExpression
BinaryExpression	BinaryExpression_BinaryExpression
BlockStatement	CallExpression_Identifier_BinaryExpression
BreakStatement	CallExpression_Identifier_Literal
CallExpression	CallExpression_Identifier_UnaryExpression
ConditionalExpression	EmptyStatement_ExpressionStatement_CallExpression
ContinueStatement	ExpressionStatement_CallExpression_Identifier
DoWhileStatement	VariableDeclarator_Identifier_CallExpression
EmptyStatement	VariableDeclarator_Identifier_LogicalExpression
ExpressionStatement	
ForInStatement	
ForStatement	
FunctionDeclaration	
FunctionExpression	
Identifier	
IfStatement	
LabeledStatement	
Literal	
LogicalExpression	
MemberExpression	
NewExpression	
ObjectExpression	
ReturnStatement	
SequenceExpression	
SwitchStatement	
ThisExpression	
ThrowStatement	
TryStatement	
UnaryExpression	
UpdateExpression	
VariableDeclaration	
WhileStatement	
WithStatement	

best-performing model, XGBoost, could detect malicious JS content with an error rate of 0.0019 for recall, 0.0302 for precision, and 0.0162 for F1. The lower precision metric revealed that the model misclassified some benign JS codes. Even though this scenario is not harmful, it may lead to threat-alert fatigue as users and security

Table 4.3: XGBClassifier and LGBMClassifier parameters obtained using *GridSearchCV* for the different input types.

Parameter	AST-JS nodes	Association rule	SHAP value
XGBClassifier			
<i>colsample_bytree</i>	0.5	0.7	0.4
<i>gamma</i>	0.0	0.1	0.0
<i>learning_rate</i>	0.2	0.2	0.2
<i>max_depth</i>	10	8	13
<i>min_child_weight</i>	1	1	1
LGBMClassifier			
<i>lambda_l1</i>	0	0	0
<i>lambda_l2</i>	0	0	0
<i>max_depth</i>	9	6	8
<i>n_estimators</i>	520	520	520
<i>num_leaves</i>	10	10	10

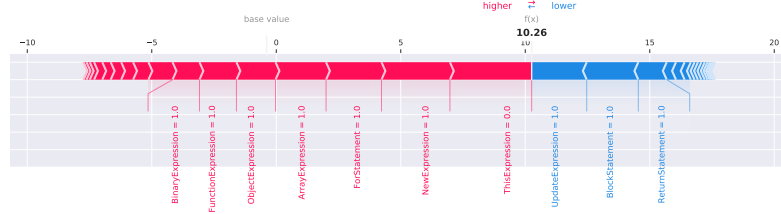
analysts will receive false alarms. Therefore, there is a need to improve this model further.

Table 4.4: Precision, recall and F1-scores obtained by the malicious JS detection model using features defined as AST-JS nodes and 10-fold cross validation.

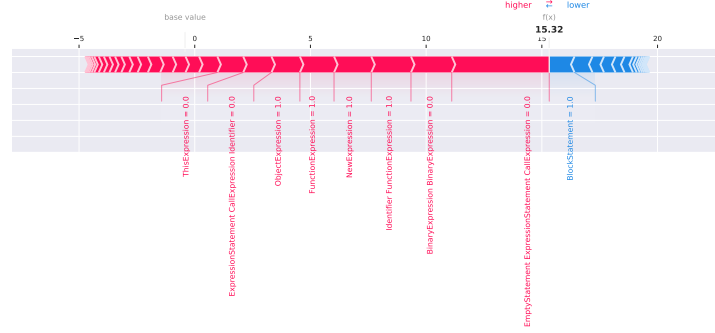
Model	Recall	Precision	F1
XGBoost	0.9981 $\pm$ 0.0008	0.9698 $\pm$ 0.0033	0.9838 $\pm$ 0.0018
LightGBM	0.9979 $\pm$ 0.0008	0.9691 $\pm$ 0.0032	0.9833 $\pm$ 0.0019
RandomForest	0.9986 $\pm$ 0.0005	0.9702 $\pm$ 0.0032	0.9842 $\pm$ 0.0018
DecisionTree	0.9983 $\pm$ 0.0005	0.9687 $\pm$ 0.0029	0.9833 $\pm$ 0.0016
LogisticRegression	0.9839 $\pm$ 0.0021	0.9414 $\pm$ 0.0039	0.9622 $\pm$ 0.0025
KNeighbors	0.8448 $\pm$ 0.0062	0.9986 $\pm$ 0.0006	0.9153 $\pm$ 0.0037
GaussianNB	0.9968 $\pm$ 0.0013	0.5493 $\pm$ 0.0029	0.7083 $\pm$ 0.0025

The AST-JS node features are analyzed using SHAP values. Fig. 4.1 shows SHAP summary plots where features pushing the prediction higher are in red and vice versa in blue. In plots (a) and (b), each SHAP value is an arrow that pushes to increase or decrease the prediction. Plot (c) and (d) show SHAP value explanations for a sample JS code dataset.

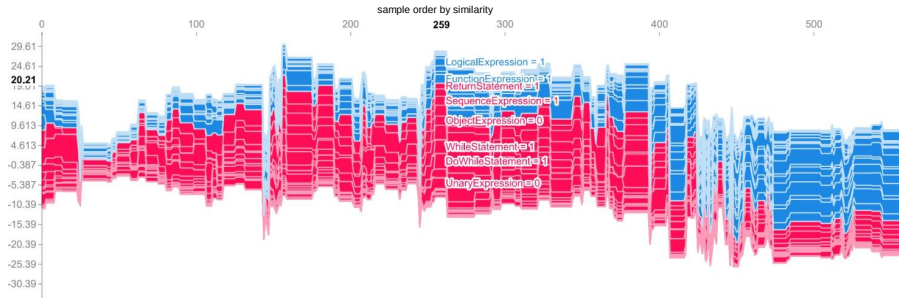
The AST-JS node features were analyzed using the SHAP values. Fig. 4.2 and Fig. 4.3 shows the relative impact of AST-JS features over the JS code dataset



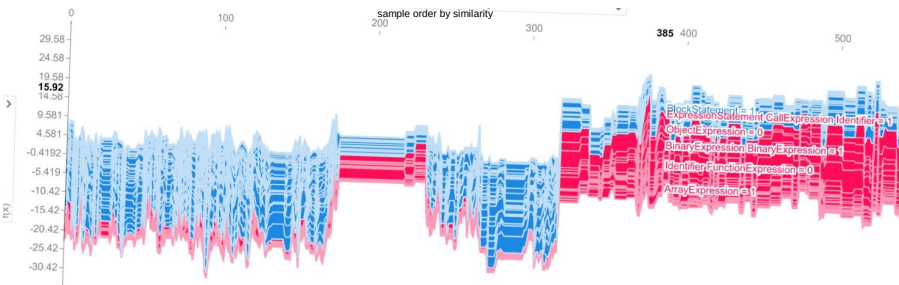
(a)



(b)



(c)



(d)

Figure 4.1: SHAP summary plots showing the AST-JS features, before (a), and after (b) feature selection. Each feature is contributing to pushing the model output over the JS dataset from the basevalue or average predicted probability of -0.38 (a) and -0.42 (b) towards $f(x)$ of 10.26 (a) and 15.32 (b). Both JS codes have a prediction value greater than the base value. Therefore, the JS codes are likely malicious. SHAP values for 3,000 JS code samples before (c) and after (d) feature selection are also included.

before and after feature selection respectively. The values were sorted by the sum of their SHAP value magnitudes. The values give the distribution of each AST-JS feature’s impact on the model output. The red and blue colors represent the high and low AST-JS feature values, respectively. The color allows visualization of how changes in the value of an AST-JS feature would affect a change in prediction; for example, high SHAP values for the *SequenceExpression* feature would indicate a JS code’s high-risk maliciousness.

It is possible to deduce that features such as the *ObjectExpression* feature would influence the model’s prediction more than the *LabeledStatement* feature using such plots. A feature such as *SequenceExpression* has a significant, positive effect on the AST-JS prediction. Therefore, high *SequenceExpression* SHAP values may indicate a higher risk for maliciousness. On the contrary, a feature such as the *FunctionExpression* has a significant, negative effect on the AST-JS class prediction, and therefore, a low *FunctionExpression* SHAP values may indicate a higher risk for maliciousness.

Additionally, interesting patterns can be observed, such as high values of the *ExpressionStatement_CallExpression_Identifier* feature clustered in a very dense region represented by the red blob. Also, low values of the *BlockStatement* feature are clustered in a very dense region, as shown by the blue blob. However, features such as the *ReturnStatement* and *FunctionExpression* have a much more uniform distribution with high and low SHAP values, respectively, that push the prediction to 1. The red and blue blobs on the left and right indicate an even distribution of that feature in the JS code dataset. Some features such as *EmptyStatement_ExpressionStatement_CallExpression* and *CallExpression_Identifier_Literal* are not crucial for most JS codes. However, these features have a significant impact on a subset of JS codes in the dataset. This scenario highlights how a globally important feature is not necessarily the most critical feature of JS codes.

Fig. 4.4 shows the SHAP dependence plot for the top AST-JS feature before (a)



Figure 4.2: SHAP summary plot before feature selection. SHAP values are plotted on the x-axis for each AST-JS feature on a row and sorted in descending order of their importance. Vertically piled-up points represent the AST-JS feature density. The colors show the AST-JS feature value. The primary risk indicator of maliciousness is shown as the ObjectExpression feature. The next most potent indicator of maliciousness is the FunctionExpression feature.

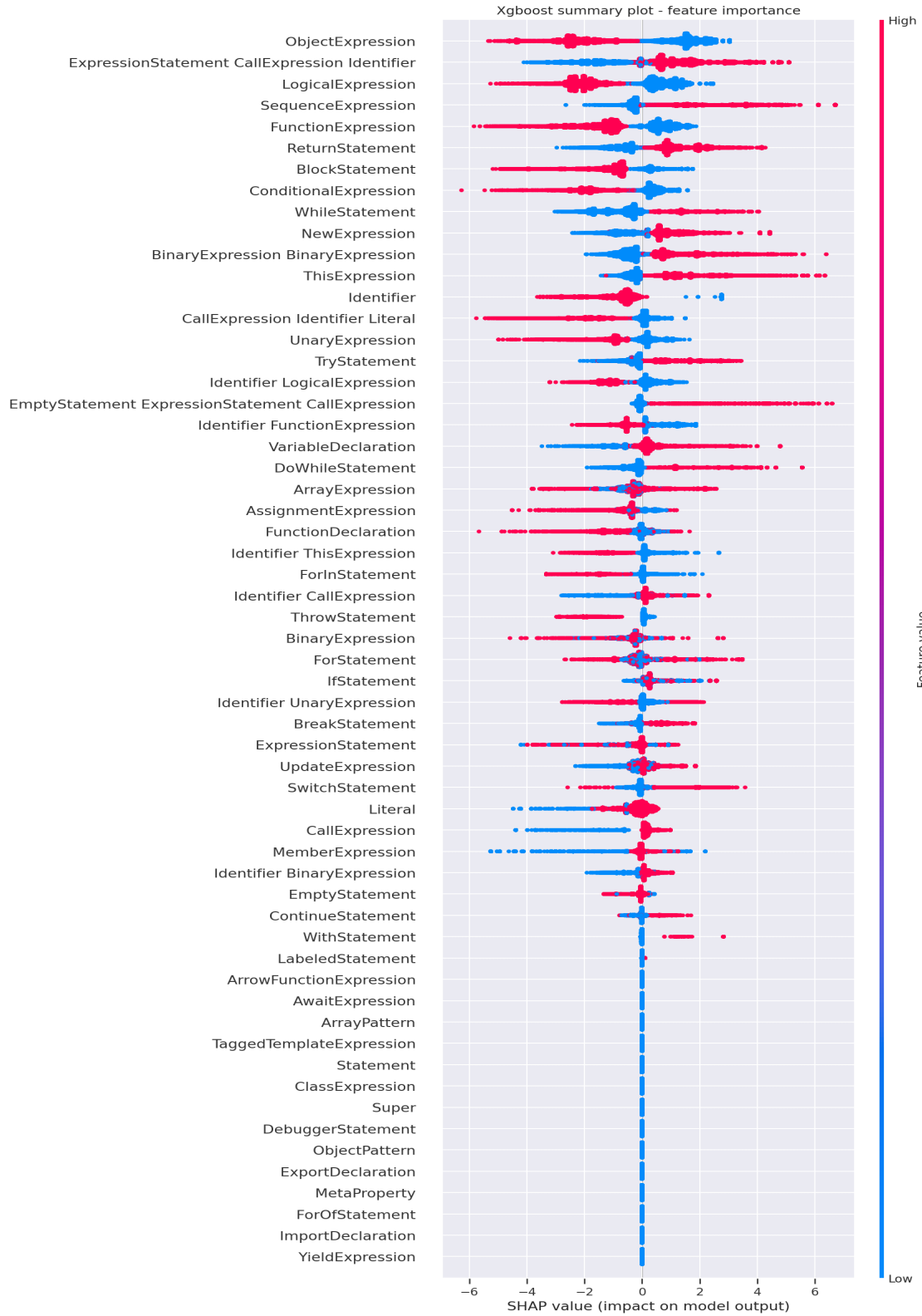


Figure 4.3: SHAP summary plot for feature selection. The SHAP values are plotted on the x-axis for each AST-JS feature on a row and sorted in descending order of their importance. The vertically piled points represent the AST-JS feature density. The colors show the AST-JS feature values. The primary risk indicator of maliciousness is the *ObjectExpression* feature. The next most potent indicator of maliciousness is the *ExpressionStatement_CallExpression_Identifier*.

and after (b) feature selection. Every dot represents a JS code. Vertical dispersion at an AST-JS feature value results from interaction effects in the model. The color highlights the high or low forces behind the interactions. The y-axis represents the SHAP values. The SHAP summary plot is obtained by projecting the SHAP dependence plot points onto the y-axis and then recoloring the value's feature.

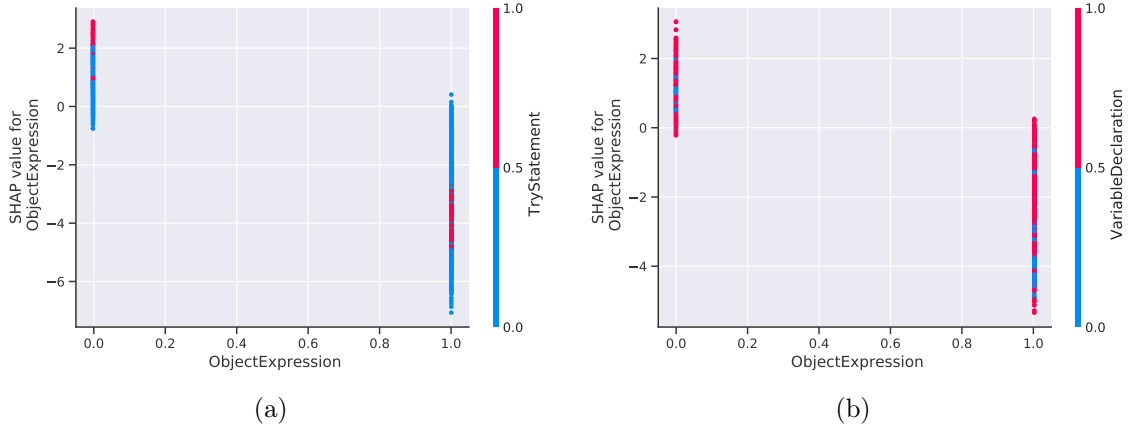


Figure 4.4: SHAP dependence plot for the top AST-JS feature before (a) and after (b) feature selection. The *TryStatement* and *VariableDeclaration* features were automatically selected for coloring based on a potential interaction in the model. In plot (b), it can be seen that low SHAP values of the *ObjectExpression* feature influence the model output more significantly for observations where the *VariableDeclaration* feature has high SHAP values.

Fig. 4.5 shows the SHAP interaction values plot for the top two AST-JS features before (a) and after (b) feature selection. It shows the main effects and some of the interaction effects for the *ObjectExpression* feature.

Table 4.5 presents precision, recall, and F1-scores obtained using features defined by association rule mining and 10-fold cross-validation. By including these features, precision was improved by 0.0127, 0.0129, 0.0131, 0.0125, 0.0188, and 0.2352 for the XGBoost, LightGBM, RandomForest, DecisionTree, LogisticRegression, and GaussianNB, respectively.

This improvement indicates a reduction in the number of misclassified benign JS codes. However, there is a reduction in the recall metric, which indicates an increase in the number of misclassified malicious JS codes. Using features from the malicious JS code samples and SHAP values, selected features based on their contribution to

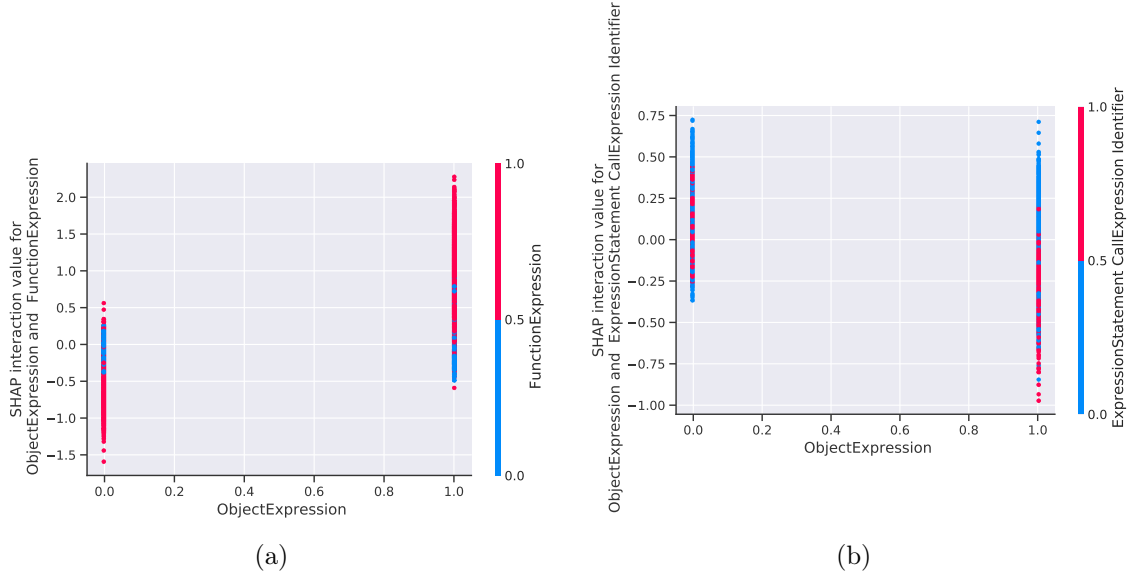


Figure 4.5: SHAP interaction values plot showing the interaction effects involving the *ObjectExpression* feature. These effects capture all vertical dispersions. Plot (a) shows that high SHAP values for the *ObjectExpression* and *FunctionExpression* features significantly influence the model output. Plot (b) shows that low SHAP values for the *ObjectExpression* feature and high SHAP values for the *ExpressionStatement_CallExpression_Identifier* feature significantly influence the model output.

the model output may improve the detection performance.

Table 4.5: Precision, recall and F1-scores obtained by the malicious JS detection model using features defined by association rule mining and 10-fold cross validation.

Model	Recall	Precision	F1
XGBoost	0.9763(± 0.0025)	0.9825(± 0.0024)	0.9794(± 0.0018)
LightGBM	0.9762(± 0.0023)	0.9820(± 0.0025)	0.9791(± 0.0018)
RandomForest	0.9757(± 0.0024)	0.9833(± 0.0025)	0.9795(± 0.0018)
DecisionTree	0.9758(± 0.0024)	0.9812(± 0.0025)	0.9785(± 0.0019)
LogisticRegression	0.9600 $\pm$ 0.0033	0.9602 $\pm$ 0.0040	0.9601 $\pm$ 0.0025
KNeighbors	0.8516 $\pm$ 0.0044	0.9991 $\pm$ 0.0007	0.9195 $\pm$ 0.0026
GaussianNB	0.8221 $\pm$ 0.0048	0.7845 $\pm$ 0.0044	0.8029 $\pm$ 0.0038

Table 4.6 presents precision, recall, and F1-scores obtained using features selected based on SHAP values and 10-fold cross-validation. Each model improved in all three metrics compared to the association rule mining features, with notable improvement for the recall and F1-score metrics. The best performing model, the XGBoost model, had an error rate of 0.0011, 0.0168, and 0.0091 for recall, precision, and F1. The

model outperformed the LightGBM and DecisionTree models with 0.03%, and the RandomForest model with 0.04% in the recall metric. A high recall rate translates to low misclassification and false-negative rate. AST-JS features selected using the SHAP values capture global and local feature importance. Therefore, these features enhance the machine learning model’s feature learning process and lead to a better prediction model. As evidenced by the experiments, high performance was achieved by all the malicious JS detection models, especially the tree ensemble methods. Consequently, these models have the least false positive and false negative rates.

Table 4.6: Precision, recall and F1-scores obtained by the malicious JS detection model using features selected by SHAP value and 10-fold cross validation.

Model	Recall	Precision	F1
XGBoost	0.9989 ± 0.0004	0.9832 ± 0.0024	0.9909 ± 0.0014
LightGBM	0.9986 ± 0.0007	0.9820 ± 0.0027	0.9902 ± 0.0015
RandomForest	0.9985 ± 0.0006	0.9840 ± 0.0024	0.9912 ± 0.0014
DecisionTree	0.9986 ± 0.0005	0.9815 ± 0.0024	0.9900 ± 0.0013
LogisticRegression	0.9895 ± 0.0013	0.9632 ± 0.0037	0.9762 ± 0.0017
KNeighbors	0.8779 ± 0.0056	0.9995 ± 0.0003	0.9347 ± 0.0031
GaussianNB	0.9373 ± 0.0056	0.6889 ± 0.0031	0.7941 ± 0.0030

Table 4.7 presents precision, recall, and F1-scores obtained using features selected based on SHAP values compared with the performance of the Boruta, ELI5, RandomForest, and SelectKBest [101, 63] feature selection methods. The proposed detection model performed better than the other methods in all three metrics, with a notable difference in precision and F1. Boruta, a wrapper around RandomForest, finds all relevant features carrying information that can be used to predict malicious JS. AST-JS features that are proven by a statistical test to be less relevant are rejected iteratively. ELI5 is used for explaining predictions. It is also referred to as permutation importance or mean decrease accuracy (MDA). The method measures how the score decreases when an AST-JS feature is eliminated. RandomForest feature importance method measures each AST-JS feature’s importance using the *entropy* function for the information gain. SelectKBest ranks AST-JS features by the k highest scores. This method measures the dependency between features us-

ing the mutual information (MI) score function. These feature selection methods have limitations in that different model iterations assign different feature values to the AST-JS features. Also, Boruta assigns a *True* or *False* value to each AST-JS feature. The permutation-based methods are computationally expensive and can have problems with highly correlated AST-JS features resulting in loss of important information. SHAP-selected AST-JS features have consistency in the feature values assigned to each AST-JS feature. Unlike Boruta, SHAP values show the degree and manner each AST-JS feature contributes to the model prediction and are model-agnostic. These features also provide interaction graphs instrumental in getting information on AST-JS feature combinations, further boosting the model detection performance.

Table 4.7: Precision, recall and F1-scores obtained by the malicious JS detection model using features selected by SHAP value and 10-fold cross validation. The performance is compared to the Boruta, ELI5, RandomForest, and SelectKBest feature selection methods.

Model	Recall	Precision	F1
SHAP	0.9989 ± 0.0004	0.9832 ± 0.0024	0.9909 ± 0.0014
Boruta	0.9983 ± 0.0008	0.9698 ± 0.0033	0.9839 ± 0.0019
ELI5	0.9982 ± 0.0008	0.9699 ± 0.0033	0.9839 ± 0.0019
RandomForest	0.9984 ± 0.0007	0.9698 ± 0.0032	0.9839 ± 0.0018
SelectKBest	0.9982 ± 0.0007	0.9687 ± 0.0034	0.9832 ± 0.0019

Fig. 4.6 shows the feature importance assigned to the AST-JS features by the Boruta, ELI5, RandomForest, and SelectKBest feature selection methods. Each method resulted in different feature values for each AST-JS feature and, therefore, a different number of features were selected for each method.

4.4 Conclusions

Web-based attacks remain a significant challenge, as evasion techniques are continuously evolving. An equally adaptive strategy is required to detect such attacks effectively. This study proposes AST-JS feature selection using SHAP values and

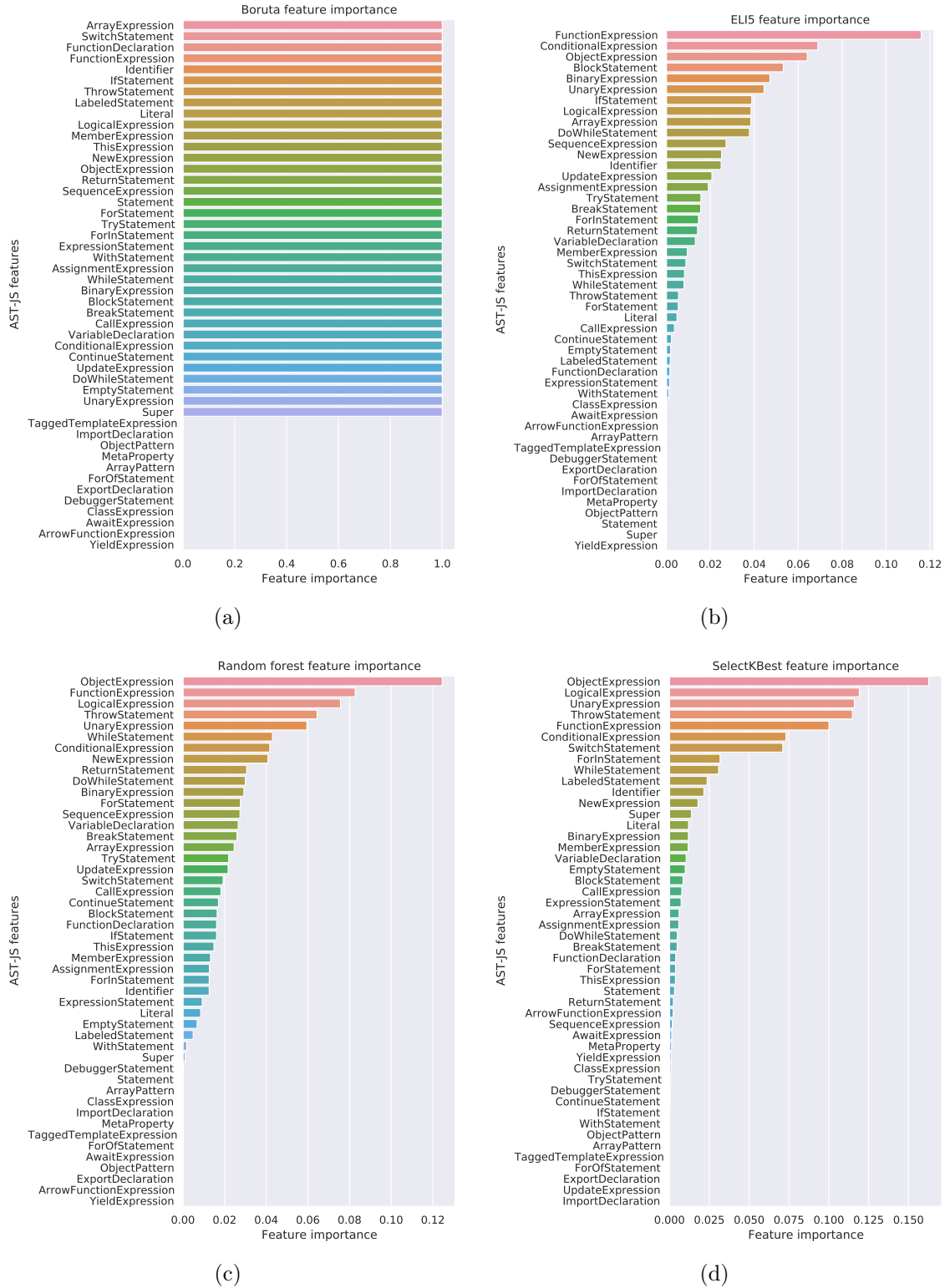


Figure 4.6: Feature importance assigned to the AST-JS features by the Boruta, ELI5, RandomForest and SelectKBest feature selection methods.

tree ensemble methods to detect these attacks. Investigating how often AST-JS nodes appear together in benign and malicious JS codes was conducted using association rule mining. One expectation is that this approach would result in fully representative features that can generalize well to other attacks. This approach results in a better malicious JS detection model.

A machine learning model can generalize for new attacks if training is performed correctly. AST-JS nodes were used to represent the JS code structure and their SHAP values for feature selection. Experiments were conducted using features selected using AST-JS node preprocessing, association rule mining, and SHAP value selected features. The performance of different models on malicious JS content detection was compared, and good detection performance for the tree ensemble methods was achieved. AST-JS nodes are resistant to perturbation in JS codes, and this ensures that the model is applicable in various JS code contexts and, therefore, to different JS attacks.

This study can be extended to investigate malicious URL features and their contribution to model performance in future work.

Chapter 5

Conclusion

In this dissertation, the study of JS-based attack detection is chosen as the main theme since this area is becoming increasingly important for researchers and organizations to prepare for the ever-evolving attacks and attack methods. As part of the findings in Chapter 2, there is a small number of JS codes identified as false positives and false negatives after performing feature learning for a plain-JS. The false positives and negatives cases are primarily attributed to obfuscation in some of the JS codes in the dataset, as shown in Fig. 5.1, which presents hard-to-detect JS codes for both benign and malicious JS codes. However, obfuscation does not necessarily indicate maliciousness of a JS code [138, 41, 19] as a few benign JS codes are often obfuscated to preserve privacy and prevent plagiarism, and to fasten code loading by removal of whitespace and comments. On the other hand, malicious JS codes are obfuscated to hide their maliciousness and evade detection and therefore use complicated or a combination of obfuscation techniques. Generally, there are four categories of obfuscation techniques [138, 41] used in JS codes,

- Randomization obfuscation by changing or inserting elements in JS codes.
- Data obfuscation by string splitting and substitution.
- Logistic structure obfuscation by manipulating a JS code execution path.
- Encoding obfuscation by converting JS codes into ASCII, Unicode, hexadeci-

```

eval(function(p,a,c,k,e,r){e=function(c){return(c<a?'':e(
y,$$(y'a$')?x(0(a)[u b,a 29(y'31y'),13 h,4h h,1j h,u c (l
'))Y(i),8 19 E X(y'1m$',{y'1h$' 17+y' 19$'))Y(m),u n E
h[0] 1r( ),t(c U(/20/g))g=y'20y',u i d 1v(),t(f)f h,u
[0]i d A 2t C) 11(),B (h[1])i d A 2t B) 11(),5k [C (h[0]i
c) y'4v$')?[0,i C] i C,y'BY' ($0(f B) y'4v$')?[0,h] h))
.I 8 10[3],13 8 10[4],1f 8 10[5],17 8 A 1k,u f 8 10[7],u
a+y' /><S P 62 T M /></G>y')N(8 G y'<G 3S 64 7A 7B
C,i B, 9 ,8 A 1a, 2j , 2K'):8 G 1,(y'2k$',y'2a$'),8 G 1,
3 A 1a),8 G 1,(y'2k$',y'2a$'),8 G 1,(y'2m$',y'M$'))N t(K i
)),u , 8,8 G 5f D()(. 15 2h(y'22$'))8 3u v j(y'1i$'),t(8
y' 2x$',y'1h$' y'3F$',y'1H$':y'1$'))Y(8 1b) 1w(y'1Z$',y'
'1d$' 1),y'36$' (y'16$' b 16+(d/2),y'1k$' 0 1a() x/2 ((0
,8 A 1d) 2r(D() (F 3g(y'4j$',a))) ,L 8),3P D() (u a 8 1y,F 8
( 1z ,a))t(b)(8 1n( j ,b))t(w)(8 1n( C ,w))t(h)(8 1n( B ,l
yy +8 18( B )+ yy 2h yy +8 18( 2h )+ yy > ,a+ <S P yy
) 0,8 2X a[1] 2p24n(a[1]) 0;8 4g a[2] 2p24n(a[2]) 0),R
Object|overflow|set|setStyle|overlay|swf|navigator|closeB
rg|none|b|show|previous|variables|parser|getElementById|i
c|d|mov|apple|t|act|vex|at|plugin|v|ideopl|v|cab|doPrepUnl
tretchToFit|zA|nocookie|cursor|styles|delete|toQueryStrin

```

(a) Malicious JS

```

3A445D725C761E757A781D497F777D7378805E721A549E445A5
5C485F445A517A481B445A489A445F477C725B789E777E501F8
5E761B713C785B777C477E481B553D357D445D445C445E445C4
1D713A749F737C713A745F477D725B785B757A713E781E737D7
1F521F689E553E357F445F445E445B445D445D445D445E445E4
3B781F517E481A553F357E445B445B445A445F445E445D445D4
1E705A497B777E473E493E777B521B481F553A357A445C445B4
5F477F737F513A513E561D561F529D481C809F357C445D445D4
9C705C749D477E481E481F553E357F445D445E445F445A789C7
7E481A445A817A481B553B357E445F445A445E445B461D477B7
1F725B773F721B777F733C473C445E489B445B473E469E733E7
9D445E473A505C733D713D705E765C781B713B733E705B505F7
3C489E473F469A473B445A489B445F473C733C713E705D765E7
3B721A777A733C473A445C489C445B473D469D733B713D705C7
9A777F521B489D473E469B473C445D489B445D473F797D561E4
1B477D481E481B501C729E721F781D653E737E753A721D477D4
7B493B445C725D705F749A777A721F481D553A357E357D445E4
5B721E501E745B721D801E445B561B561D445C525E445F813E8
1D745F721C801A501C781A761F657A765B765C721A773D585F7
3B485C473E445C813B813A445A721D501E745A721F801D445B5
3A561D473C445C813F813A445F721C5D1D745F721B8D1B445B5

```

(b) Benign JS

Figure 5.1: Example of a hard-to-detect JS code by analysis of false positive and false negative cases in the dataset. These are sample codes for which detection is ineffective while defining input as a plain-JS. These codes also include numbers and special characters in addition to JS code words. Input defined as plain-JS does not adequately capture such JS code details and code structure representation.

mal, or by encryption.

Doc2Vec maintains the semantics of words and would therefore accurately detect the first three obfuscation techniques, which do not affect the original semantics of JS code content. Therefore, using Doc2Vec and AST-JS features, most of the code details in the JS codes earlier identified as false positive and false negative cases are classified accordingly.

However, the proposed method in Chapter 2 has some limitations when faced with obfuscation in the dataset, as shown in Table 5.1, which shows the performance of paragraph vector models on obfuscated JS dataset. As shown in the table, the

proposed approach in Chapter 2 can detect obfuscated malicious JS codes with an error rate of approximately 0.08 and 0.07 in recall for PV-DBoW and PV-DM, respectively. The detection is possible because JS codes have syntactic similarities at an abstract level, and therefore, AST can bypass obfuscation.

Table 5.1: Precision, Recall and F1-score representing the performance of paragraph vector models on obfuscated JS codes.

Model	Precision	Recall	F1-score
PV-DBoW	0.94 ($\pm$ 0.1571)	0.92 ($\pm$ 0.1159)	0.93 ($\pm$ 0.0868)
PV-DM	0.93 ($\pm$ 0.1675)	0.93 ($\pm$ 0.1016)	0.92 ($\pm$ 0.0873)

One of the objectives of obfuscation is to hinder JS code reverse engineering. Therefore, after performing DUD-preprocessing for the obfuscated JS codes, the resultant Plain-JS codes may not result in the same original JS code as before JS code obfuscation was applied. However, as part of the findings in this research in Chapter 3, an obfuscated JS code that has been DUD-preprocessed results in regular Plain-JS code. The detection performance for the machine learning models is improved because the resultant JS code has the same semantic and syntactic properties as the original one. After performing deobfuscation, which is the first step in the obfuscated JS code preprocessing, the resultant JS codes have to be inspected manually for the presence of obfuscation. Some approaches have been proposed for the automatic detection of obfuscated JS codes. Some of these approaches focus on specific JS code obfuscator tools that are unique to specific obfuscation techniques. Therefore, given a dataset that contains JS codes obfuscated by a JS code obfuscator tool other than the one focused in the approach would result in misclassification of obfuscated JS codes. The existing detection methods for obfuscated JS code detection can not be easily applied to other JS code obfuscator tools. Other studies use features such as the structure of the obfuscated JS code and functions such as *eval()* and *unescape()* within a x number of bytes of each other. These are heuristic-based approaches that are limited in capability when faced with sophisticated JS code obfuscation techniques. In the case of a packed JS code, single-step unpacking is not enough as

most obfuscated malicious JS codes contain multiple layers of obfuscation.

Blacklists, heuristic- and signature-based systems are widely used to detect malicious URLs and JS codes. Current detection methods have various challenges stemming from fast-flux, cloaking, and zero-day attacks. Some services that can be used for copyright and privacy reasons in benign JS codes, such as obfuscation, can also create new or variant JS attacks. Therefore, it is biased to classify transformed URLs and JS codes as malicious without further detailed analysis. A system to analyze and accurately detect web-based attacks is needed. As part of the research findings in Chapter 4, feature selection using SHAP values results in a better prediction model. Other researchers limited their scope by studying specific malicious URLs and JS attacks, such as those containing phishing attacks, social engineering, malvertisement attacks, and Denial-of-service (DoS) attacks, among others. For this, they used manually engineered features unique to the type of attack they intended to identify. Such approaches have challenges in that they cannot be generalized to other attack types. The premise is that features selected using SHAP values can overcome such challenges, as the analyst can easily and quickly identify which features significantly impact the overall model performance. Also, as shown in Fig. 5.2, using SHAP interaction values, it is easier to tell which AST-JS feature combinations or interactions contribute more toward the model’s detection performance. These feature combinations are shown as *—. Some of these interactions are directly interpretable as attack types present in the dataset. The SHAP-based approach does not, therefore, focus on specific categories of web-based attacks or threats.

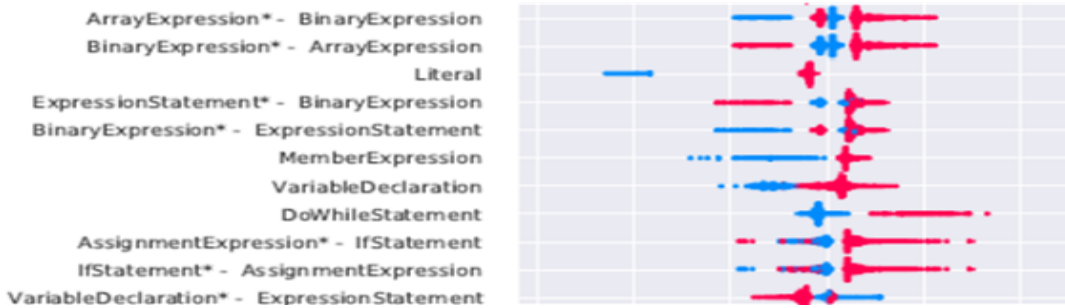


Figure 5.2: SHAP interaction values showing AST-JS features pair combinations. The plot makes it easier to determine which pair contributes more to the overall model performance and in which direction.

5.1 Contributions of This Dissertation

In Chapter 2, first, Abstract Syntax Tree (AST) for code structure representation is introduced. AST-JS gives a robust property against some perturbation in JS codes. Second, a machine learning approach called Doc2vec is used to conduct feature learning for JS-based attack detection. Features learned with Doc2Vec are of low dimensions, which ensure faster detections. In addition, Doc2Vec can learn context information of texts with variable length. Third, AST-level manipulation and AST-level merging are conducted to enhance the feature learning process for Doc2Vec. Lastly, the proposed approach automatically learns feature vectors compared to other previous methods, which employ manually crafted feature sets and subsets. The proposed AST-JS features and Doc2Vec for feature learning provide better accuracy and fast classification in malicious JS code detection compared to conventional approaches and can flag malicious JS codes previously identified as hard-to-detect.

In Chapter 3, this study proposes a hybrid-analysis approach for obfuscated malicious JS codes by integrating the use of a JS code beautifier and formatter, a Virtual Machine (VM), a JS code editor, and a python `int_to_str()` function to deobfuscate an obfuscated JS code, unpack a packed JS code and decode an encoded JS code respectively. Malicious JS codes employ sophisticated obfuscation methods such as packing, encoding, and multi-layered obfuscation methods. The resulting obfuscated malicious JS codes are characterized by unnatural and unreadable syntax and corrupted code structure different from that of the original JS code. To effectively conduct feature learning on such obfuscated JS codes, this study performs deobfuscation, unpacking, and decoding of obfuscated malicious JS codes. The FastText model automatically learns low dimensions feature hence ensuring faster detection. The FastText model is fast and suitable for big datasets and, therefore, well suited for the JS code feature learning. The model can also learn vector representations for JS code words that are not contained in the vocabulary at the time of model training and would, therefore, result in reliable vector representations for future,

new and unseen JS codes and consequently new JS-based attacks. The proposed approach enhances feature learning and provides improved accuracy in the detection of obfuscated malicious JS codes.

In Chapter 4, this study proposes an automatic feature selection and classification approach for malicious JS content using SHAP and tree ensemble methods. SHAP values provide necessary insights into the distribution of specific benign and malicious JS features. The SHAP-based approach adopts features that are automatically selected based on their contributions to the model output. SHAP-selected AST-JS features have consistency in the feature values assigned to each AST-JS feature. Unlike Boruta, SHAP values show the degree and manner each AST-JS feature contributes to the model prediction and are model-agnostic. SHAP-selected AST-JS features represent adequate details of a JS code structure without executing the code. These features also provide interaction graphs instrumental in getting information on AST-JS feature combinations, further boosting the model detection performance. The experimental results show that the proposed approach is a better prediction model.

In general, a robust JS code structure representation method of low dimensions and efficient learning method for JS codes is introduced. The approach ensures highly accurate detection and fast classification for malicious JS codes. The proposed methods can be implemented in the form of a firewall. In addition, because the model is of low dimensions, it can easily be implemented in the browser to ensure faster detection.

5.2 Future Work

As future work, there is a need to improve the performance to ensure more accurate detection of obfuscated malicious JS codes. For example, (1) by estimating the maliciousness of obfuscated features based on entropy information of string arrays and/or some known signatures of toolkits and (2) by using statistics on obfuscated parts such as a percentage of obfuscated parts in the whole code and/or which parts

(e.g., variables and functions) are obfuscated, and so on. Therefore, there is a need to identify obfuscated parts accurately, but this is also a challenging problem.

In addition, there is a need to develop a technique to check for obfuscation in the dataset automatically. This technique should remain independent of specific JS code obfuscator tools or obfuscation techniques. Recently, studies are focusing on embedding source code. Regular and obfuscated JS codes follow some code structure rules for sentences, functions, variables, operation procedures, and stop conditions. As such, a word embedding model may lose much of the information in a JS code structure. Embedding JS source code would be language-specific and share the same concept as the NLP embedding models with a few tweaks. While the existing word embedding models, such as the FastText and the Paragraph Vector models, yield high-quality vector representations on programs and source code tasks such as obfuscated malicious JS code detection, they are not well-tailored for obfuscated JS code parts. As such, parsing errors are common when trying to realize a plain-JS code for static analysis. It is envisioned that directly embedding a JS source code would facilitate capturing a JS code structure and avoiding such errors.

The objective of this study is to have a high recall, which translates to as few false-negative cases as possible. There is however a need to check the number of false positive cases as many false positives result in an annoying system that can result in alert fatigue resulting in missed or ignored alerts and delayed responses. Therefore, in future work, we endeavor to perform better training to equally improve the detection of benign JS.

The proposed method cannot detect other attacks, such as phishing attacks that employ benign JS codes to perform attacks. In future work, there is a plan to incorporate the detection of such attacks, for example, by investigating the combination of benign JS codes that result in a phishing attack. Also, phishing attack features could be incorporated for training the model and investigating other features such as inline JS.

Therefore, these remedies are left as future work.

Bibliography

- [1] AccentureSecurity. “THE COST OF CYBERCRIME, Ninth Annual Cost of Cybercrime Study, UNLOCKING THE VALUE OF IMPROVED CYBER-SECURITY PROTECTION”. In: (2019).
- [2] Akamai. “State of the Internet Security | Financial Services – Hostile Takeover Attempts”. In: (2020). URL: <https://www.akamai.com/us/en/multimedia/documents/state-of-theinternet/soti-security-financial-services-hostile-takeover-attempts-report-2020.pdf>.
- [3] M. Akiyama et al. “Analyzing the ecosystem of malicious URL redirection through longitudinal observation from honeypots”. In: *Computers & Security* 69 (2017), pp. 155–173.
- [4] M. Akiyama et al. “Design and Implementation of High Interaction Client Honeypot for Drive-by-download Attacks”. In: *IEICE Transactions on Communication, May* E93-B.5 (2010), pp. 1131–1139.
- [5] Alexa and Inc. “<https://www.alexa.com/topsites>”. In: *The top 500 sites on the web, Accessed May 2018* (2018).
- [6] Alexa and Inc. “The top 500 sites on the web”. In: *In <https://www.alexa.com/topsites>, Accessed on May* (2020).
- [7] Y. Alosefer and O. Rana. “Honeyware: A web based low interaction client honeypot”. In: *IEEE third international conference on software testing, verification and validation workshops (ICSTW)* (2010), pp. 410–417.

- [8] K. AlRoum et al. “Detecting Malware Domains: A Cyber-Threat Alarm System”. In: *In Proceeding of the International Conference on Emerging Technologies for Developing Countries*, Springer International Publishing, Cham, Switzerland (2018), pp. 181–191.
- [9] A. Anand et al. “Phishing URL detection with oversampling based on text generative adversarial networks”. In: *In proceedings of the IEEE international conference on Big Data (Big Data)*, <https://doi.org/10.1109/BigData.2018.8622547> (2018), pp. 1168–1177.
- [10] L. Bilge et al. “Exposure: A Passive DNS Analysis Service to Detect and Report Malicious Domains”. In: *Association for Computing Machinery (ACM) Transactions on Information and System Security* 16.4 (2014), pp. 1–14.
- [11] L. Bilge et al. “EXPOSURE: Finding malicious domains using passive DNS analysis”. In: *In Proceedings of the 18th Annual Network and Distributed System Security Conference (NDSS’11)* (2011), pp. 1–17.
- [12] G. Blanc et al. “Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts”. In: *Proc. 26th Advanced Information Networking and Applications Workshops (WAINA)* (2012), pp. 344–351.
- [13] P. Bojanowski et al. “Enriching word vectors with subword information”. In: *Transactions of the Association for Computational Linguistics (TACL)*, *arXiv preprint 27 arXiv:1607.04606* 5 (2017), pp. 135–146.
- [14] C.J.C. Burges. “A tutorial on Support Vector Machines for Pattern Recognition”. In: *data mining and knowledge discovery* 2.2 (1998), pp. 121–167.
- [15] C.J.C. Burges. “A tutorial on Support Vector Machines for Pattern Recognition”. In: *data mining and knowledge discovery* 2.2 (1998), pp. 121–167.
- [16] Businesswire. In: (2021). URL: <https://www.businesswire.com/news/home/20181115005665/en/Opus-Ponemon-Institute-Announce-Results-2018-Third-Party>.

- [17] D. Canali et al. “Prophiler: a fast filter for the large-scale detection of malicious web pages”. In: *In Proceedings of the 20th international conference on World Wide Web (WWW’11), ACM, Hyderabad, India, March (2011)*, pp. 197–206.
- [18] Y. Choi et al. “Automatic detection for JavaScript obfuscation attacks in web pages through string pattern analysis”. In: *International conference on future generation information technology, Springer (2009)*, pp. 160–172.
- [19] Y. Choi et al. “Automatic detection for JavaScript obfuscation attacks in web pages through string pattern analysis”. In: *In Proceedings of the 1st International Conference on Future Generation Information Technology(FGIT), Springer-Verlag Berlin Heidelberg, May (2009)*, pp. 160–172.
- [20] N. Chou et al. “Client-Side Defense against Web-Based Identity Theft”. In: *In Proceedings of The 11th Annual Network and Distributed System Security Symposium (NDSS ’04), <http://crypto.stanford.edu/SpoofGuard/webspoof.pdf> (2004)*.
- [21] M. Cova, C. Kruegel, and G. Vigna. “Detection and analysis of drive by download attacks and malicious JavaScript code”. In: *in Proceedings of the 19th International conference on World Wide Web (2010)*.
- [22] C. Curtsinger et al. “Zozzle: Fast and precise in-browser JavaScript malware detection”. In: *In Proc. of USENIX Security Symposium (2011)*, pp. 33–48.
- [23] C. Curtsinger et al. “Zozzle: Low-overhead Mostly Static JavaScript Malware Detection”. In: *In Proceedings of the Usenix Security Symposium (2011)*, pp. 1–18.
- [24] A. Dai, C. Olah, and Q. Le. “Document embedding with paragraph vectors”. In: *NIPS Deep Learning Workshop (2014)*, pp. 1–8.
- [25] A. Dai, C. Olah, and Q. Le. “Document embedding with paragraph vectors”. In: *NIPS Deep Learning Workshop (2014)*, pp. 1–8.

- [26] Dan's and Tools. "Javascript Viewer, Beautifier, Formatter and Editor". In: *In: <https://www.cleancss.com/javascript-beautify/>, Accessed on November 2019* (2019).
- [27] R. Deveaud, E. SanJuan, and P. Bellot. "Accurate and effective latent concept modeling for ad hoc information retrieval". In: *Document numérique* (2014), pp. 61–84.
- [28] DirtyMarkup. "JavaScript beautifier". In: *In: <http://www.dirtymarkup.com/>, Accessed on November 2019* (2019).
- [29] edgescan. "Fullstack Vulnerability Management, VULNERABILITY STATISTICS REPORT". In: (2020).
- [30] ENISA. "Web application attacks, ENISA Threat Landscape". In: (2020).
- [31] ENSIGHTEN. "Top Five Cyberattacks Targeting Your Website". In: (2020). URL: <https://www.ensighten.com/blog/top-five-cyberattacks-targeting-your-website-in-2020>.
- [32] A. Fass et al. "JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript". In: *in the Proceedings of the 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Saclay, France, Springer* (2018), pp. 303–325.
- [33] M.N. Feroz and S. Mengel. "Phishing URL detection using URL ranking". In: *Proceedings of the IEEE International Congress on Big Data, BigData Congress* (2015), pp. 635–638.
- [34] M. Ferreira. "Malicious URL detection using machine learning algorithms". In: *in Proceedings of the Digital Privacy Security Conference, 10.11228/dpsc.01.01* (2019), pp. 114–122.
- [35] S. Ford et al. "Analyzing and detecting malicious flash advertisements". In: *In Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09), IEEE Computer Society, Honolulu, Hawaii, USA, December* (2009), pp. 363–372.

- [36] M. Fraiwan et al. “Analysis and identification of malicious JavaScript code”. In: *information security journal: a global perspective* 21.1 (2012), pp. 1–11.
- [37] I. Ghafir and V. Prenosil. “DNS traffic analysis for malicious domains detection”. In: *In the 2nd International Conference on Signal Processing and Integrated Networks (SPIN’15), IEEE Xplore Digital Library* (2015), pp. 613–918.
- [38] Y. Goldberg. “Neural Network Methods for Natural Language Processing”. In: *Synthesis Lectures on Human Language Technologies, volume 37, Morgan and Claypool* (2017), p. 69.
- [39] Y. Goldberg. “Neural Network Methods for Natural Language Processing”. In: *Synthesis Lectures on Human Language Technologies, volume 37, Morgan and Claypool* (2017), p. 69.
- [40] A. Gorji and M. Abadi. “Detecting obfuscated JavaScript malware using sequences of internal function calls”. In: *in the Proceedings of the 52nd ACM Southeast Regional Conference (ACMSE’14), Kennesaw, GA, USA* (2014), pp. 1–6.
- [41] A. Gorji and M. Abadi. “Detecting obfuscated JavaScript malware using sequences of internal function calls”. In: *in Proc. 52nd ACM Southeast Conference (ACMSE’14), Kennesaw, GA, USA* (2014), pp. 1–6.
- [42] M. Graziano, D. Balzarotti, and A. Zidouemba. “ROPMEMU: A framework for the analysis of complex code-reuse attacks”. In: *In Proceedings of the 11th Asia Conference on Computer and Communications Security (ASIA CCS’16), ACM* (2016), pp. 47–58.
- [43] J. Grossman et al. “XSS Attacks: cross-site scripting exploits and defense”. In: *Burlington, MA: Syngress* (2007).
- [44] Jiawei Han et al. “Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach”. In: *Data Mining and Knowledge Discov-*

- ery 8.1 (Jan. 2004), pp. 53–87. DOI: 10.1023/B:DAMI.0000005258.31418.83. URL: <https://doi.org/10.1023/B:DAMI.0000005258.31418.83>.
- [45] M. Hatada et al. “Empowering Anti-malware Research in Japan by Sharing the MWS Datasets”. In: *IPSJ Journal of Information Processing, Sep* 23.5 (2015), pp. 579–588.
 - [46] A. Hidayat. “<http://esprima.org/>”. In: *Esprima Release master, April 15, Accessed May 2018* (2018).
 - [47] Li Hongtao, Maffeis Sergio, and Cheraghchi Mahdi. “Amj: An analyzer for malicious javascript”. In: *Imperial College London, Department of Computing* (2018).
 - [48] F. Howard. “Malware with your mocha: Obfuscation and anti-emulation tricks in malicious JavaScript”. In: *Sophos Lab* (2010), pp. 1–18.
 - [49] X. Hu et al. “JSForce: A Forced Execution Engine for Malicious JavaScript Detection”. In: *In Proceedings of the 13th International Conference on Security and Privacy in Communication Networks (SecureComm’17), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering* 238 (2017), pp. 704–720.
 - [50] eComm Insights. “3rd Party Impact on Performance, eCOMMERCE 3RD PARTY TECHNOLOGY INDEX”. In: (2018).
 - [51] A.K. Jain and B.B. Gupta. “Towards detection of phishing websites on client-side using machine learning based approach”. In: *Telecommunication Systems*, <https://doi.org/10.1007/s11235-017-0414-0> 68.4 (2018), pp. 687–700.
 - [52] H. Jelodar et al. “Latent Dirichlet Allocation (LDA) and Topic modeling: models applications a survey”. In: *arXiv:1711.04305 [cs.IR]*. (2017), pp. 1–40.
 - [53] T. Johnishi et al. “Classification of Hostile JavaScript based on Encoding Abstract Syntax Tree”. In: *Computer Security Symposium* (2012), pp. 232–237.

- [54] J. Jones. “Abstract Syntax Tree Implementation Idioms”. In: *Pattern Languages of Programs* (2003).
- [55] A. Joulin et al. “Bag of Tricks for Efficient Text Classification”. In: *In Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics (EACL), Short Papers. Valencia, Spain* (2017), pp. 427–431.
- [56] JSUPACK. “<http://jsunpack.jeek.org/>”. In: *A Generic JavaScript Unpacker, 2017* (2017).
- [57] M. Kamizono et al. “Categorizing Hostile JavaScript Using Abstract Syntax Tree Analysis”. In: *Information Processing Society of Japan* (2012), pp. 349–356.
- [58] S. Kaplan et al. ““NOFUS: Automatically Detecting” + String.fromCharCode(32) + “ObFuSCateD “.toLowerCase() + “JavaScript Code””. In: *Microsoft Research Technical Report MSR-TR-2011.57* (2011), pp. 1–11.
- [59] A. Zarras and A. Kapravelos et al. “The dark alleys of madison avenue: Understanding malicious advertisements”. In: *in Proceedings of the 2014 ACM Conference on Internet Measurement Conference (IMC’14), Vancouver, BC, Canada, November* (2014), pp. 373–380.
- [60] N. Khan, J. Abdullah, and A.S. Khan. “Defending malicious script attacks using machine learning classifiers”. In: *wireless communication and mobile computing* 2017.5360472 (2017), pp. 1–9.
- [61] H. Kim et al. “Efficient detection of malicious web pages using high interaction client honeypots”. In: *Journal of information science and engineering* 28.5 (2012), pp. 911–924.
- [62] T. Krueger and K. Rieck. “Intelligent defense against malicious JavaScript code”. In: *PIK - Praxis der Informationsverarbeitung und Kommunikation* 35.1 (2012), pp. 54–60.

- [63] Miron B. Kursa and Witold R. Rudnicki. “Feature Selection with the Boruta Package”. In: *Journal of Statistical Software* 36.11 (2010). DOI: 10.18637/jss.v036.i11.
- [64] M. Kuyama, Y. Kakizaki, and R. Sasaki. “Method for detecting a malicious domain by using only well known information”. In: *International Journal of Cyber-Security and Digital Forensics* 5.4 (2016), pp. 166–174.
- [65] M. Kuyama, Y. Kakizaki, and R. Sasaki. “Method for Detecting a Malicious Domain by using WHOIS and DNS features”. In: *In Proceedings of the Third International Conference on Digital Security and Forensics (DigitalSec’16), Kuala Lumpur, Malaysia* (2016), pp. 74–80.
- [66] Q. Le and T. Mikolov. “Distributed representations of sentences and documents”. In: *Proceedings of the 31st International Conference on Machine Learning (ICML-14)* (2014), pp. 1188–1196.
- [67] Q. Le and T. Mikolov. “Distributed representations of sentences and documents”. In: *Proceedings of the 31st International Conference on Machine Learning (ICML-14)* (2014), pp. 1188–1196.
- [68] Z. Li et al. “Knowing your enemy: understanding and detecting malicious web advertising”. In: *In Proceedings of the 2012 ACM conference on Computer and communications security (CCS ’12), New York, NY, USA* (2012), pp. 674–686.
- [69] E. Lielmanis and L. Newman. “Online JavaScript Beautifier (v1.10.2), Beautify, unpack or deobfuscate JavaScript and HTML, make JSON/JSONP readable”. In: *In: <https://beautifier.io/>, Accessed on November 2019* (2019).
- [70] P. Likarish and E. Jung. “A targeted web crawling for building malicious javascript collection”. In: *In Proceedings of the ACM First International Workshop on Data-Intensive Software Management and Mining (DSMM ’09), ACM, New York, USA, doi:10.1145/1651309.1651317* (2009), pp. 23–26.

- [71] LLC and OpenDNS. “PhishTank: An anti-phishing site”. In: *In: <https://www.phishtank.com>* (2016).
- [72] G. Lu and S. Debray. “Automatic simplification of obfuscated JavaScript code: A semantics-based approach”. In: *In Proceedings of the IEEE 6th International Conference on Software Security and Reliability, Gaithersburg MD USA* (2012), pp. 31–40.
- [73] Scott M Lundberg et al. “Explainable machine-learning predictions for the prevention of hypoxaemia during surgery”. In: *Nature Biomedical Engineering* 2.10 (2018), p. 749.
- [74] Scott M. Lundberg, Gabriel G. Erion, and Su-In Lee. “Consistent Individualized Feature Attribution for Tree Ensembles”. In: *CoRR* abs/1802.03888 (2018). arXiv: 1802.03888. URL: <http://arxiv.org/abs/1802.03888>.
- [75] Scott M. Lundberg et al. “From local explanations to global understanding with explainable AI for trees”. In: *Nature Machine Intelligence* 2.1 (2020), pp. 2522–5839.
- [76] J. Ma et al. “Beyond blacklists: Learning to detect malicious web sites from suspicious URLs”. In: *In Proceedings of the 15th ACM international conference on knowledge discovery and data mining (SIGKDD’09), Paris, France, ACM* (2009), pp. 1245–1254.
- [77] J. Ma et al. “Identifying suspicious URLs: An application of large-scale online learning”. In: *in Proceedings of the 26th Annual International Conference on Machine Learning (ICML’09)* (2009), pp. 681–688.
- [78] J. Ma et al. “Learning to Detect Malicious URLs”. In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 2(3).30 (2011), pp. 1–24.
- [79] S. Marchal et al. “Phishstorm: detecting phishing with streaming analytics”. In: *IEEE Transactions on Network and Service Management* 11.4 (2014), pp. 458–471.

- [80] R. Masri and M. Aldwairi. “Automated malicious advertisement detection using virustotal, urlvoid, and trendmicro”. In: *In Proceedings of the 8th International Conference on Information and Communication Systems (ICICS’17)*, doi:10.1109/IACS.2017.7921994 (2017), pp. 336–34.
- [81] D.K. McGrath and M. Gupta. “Behind phishing: An examination of phisher modi operandi”. In: *In Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET), San Francisco, CA, April* (2008).
- [82] K. Al Messabi et al. “Malware detection using dns records and domain name features”. In: *In Proceedings of the 2nd International Conference on Future Networks and Distributed Systems (ICFNDS’18), New York, NY, USA, ACM, URL: <https://doi.org/10.1145/3231053.3231082>* (2018), 29:1–29:7.
- [83] T. Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *In Proc. Advances in Neural Information Processing Systems 26* (2013), pp. 3111–3119.
- [84] T. Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *In Proc. Advances in Neural Information Processing Systems 26 (NIPS)* (2013), pp. 3111–3119.
- [85] T. Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *In ICLR Workshop Papers* (2013), pp. 1–12.
- [86] T. Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *In ICLR Workshop Papers* (2013), pp. 1–12.
- [87] J. Misek and F. Zavoral. “Mapping of Dynamic Language Constructs into Static Abstract Syntax Trees”. In: *In Proceedings of the 2010 IEEE/ACIS 9th International Conference on Computer and Information Science* (2010), pp. 625–630.
- [88] M. Moghimi and A.Y. Varjani. “New rule-based phishing detection method”. In: *Expert Systems with Application* 53 (2016), pp. 231–242.

- [89] O. Mogren. “Malicious JavaScript detection using machine learning”. In: *learning*, <http://www.cse.chalmers.se/edu/course/FDAT085> 10.11 (2017), pp. 1–7.
- [90] [n. d.]. myWOTC. “myWOT Web of Trust”. In: *In https://www.mywot.com/*, *Accessed on May* (2020).
- [91] S. Ndichu, S. Kim, and S. Ozawa. “Deobfuscation, Unpacking, and Decoding of Obfuscated Malicious JavaScript for Machine Learning Models Detection Performance Improvement”. In: *CAAI Transactions on Intelligence Technology*, DOI: 10.1049/trit.2020.0026, Online ISSN 2468-2322, 12 June (2020), pp. 1–9.
- [92] S. Ndichu et al. “A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors”. In: *Applied soft computing journal* 84.105721 (2019), pp. 1–11.
- [93] S. Ndichu et al. “A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors”. In: *Applied soft computing journal* 84.105721 (2019), pp. 1–11.
- [94] S. Ndichu et al. “A Machine Learning Approach to Malicious JavaScript Detection using Fixed Length Vector Representation”. In: *In Proceedings of 2018 International Joint Conference on Neural Networks, (IJCNN’18), Rio de Janeiro, Brazil, July 8-13* (2018), pp. 1–8.
- [95] S. Ndichu et al. “A Machine Learning Approach to Malicious JavaScript Detection using Fixed Length Vector Representation”. In: *Proc. of 2018 International Joint Conference on Neural Networks (IJCNN 2018)* (2018), pp. 1–8.
- [96] W. NWang and K. Shirley. “Breaking bad: detecting malicious domains using word segmentation”. In: *In Proceedings of the 9th IEEE Workshop on Web 2.0 Security and Privacy (W2SP’15), CoRR abs/1506.04111*, <http://arxiv.org/abs/1506.04111> (2015).

- [97] JS. Obfuscator. “Javascript Obfuscator is a powerful encoding and obfuscation technologies prevent reverse engineering, copyright infringement and unauthorized modification of your code”. In: *In: <https://javascriptobfuscator.com/Javascript-Obfuscator.aspx>, Accessed on August 2019* (2019).
- [98] R.J. Oentaryo et al. “Detecting click fraud in online advertising: a data mining approach,’ Journal of Machine Learning Research (JMLR), January”. In: *15* 1 (2014), pp. 99–140.
- [99] J. Pan and X. Mao. “Obfuscated Malicious JavaScript Detection by Machine Learning”. In: *2nd International Conference on Advances in Mechanical Engineering and Industrial Informatics (AMEII’16)* (2016), pp. 805–810.
- [100] Patchstack. “WORDPRESS VULNERABILITY DATABASE, Hand curated, verified and enriched WordPress vulnerability information”. In: (2021).
- [101] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.
- [102] F. Pedregosa et al. “Scikit-learn: Machine learning in python”. In: *Journal of Machine Learning Research - JMLR* 12 (2011), pp. 2825–2830.
- [103] F. Pedregosa et al. “Scikit-learn: Machine learning in python”. In: *Journal of Machine Learning Research - JMLR* 12 (2011), pp. 2825–2830.
- [104] H. Petrak. “Javascript Malware Collection - A collection of almost 40.000 Javascript malware samples”. In: *In: <https://github.com/HynekPetrak/javascript-malwarecollection>, Accessed on August 2019* (2019).
- [105] Hynek Petrak. “JavaScript Malware Collection - A collection of almost 40.000 JavaScript malware samples”. In: (2020). URL: <https://github.com/HynekPetrak/javascript-malwarecollection,%20Accessed%20on%20October%202020>.

- [106] B. Pincombe. “Comparison of human and latent semantic analysis (LSA) judgements of pairwise document similarities for a news corpus”. In: *Tech. rep. DSTO-RR-0278, Information Sciences Laboratory, Defence Science and Technology Organization, Department of Defense, Australian Government* (2004).
- [107] Python. “Python 2.7.17 documentation, the python standard library”. In: *In: <https://docs.python.org/2/library/functions.html>, Accessed on August 2019* (2019).
- [108] R. Radim and P. Sojka. “Software Framework for Topic Modelling with Large Corpora”. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (2010), pp. 45–50.
- [109] R. Radim and P. Sojka. “Software Framework for Topic Modelling with Large Corpora”. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (2010), pp. 45–50.
- [110] Sebastian Raschka. “MLxtend: Providing machine learning and data science utilities and extensions to Python’s scientific computing stack”. In: *The Journal of Open Source Software* 3.24 (Apr. 2018). DOI: 10.21105/joss.00638. URL: <http://joss.theoj.org/papers/10.21105/joss.00638>.
- [111] V. Raychev et al. “Learning Programs from Noisy Data”. In: *In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL’16), New York, NY, USA, ACM* (2016), pp. 761–774.
- [112] Niranjana Reddy. “Practical Cyber Forensics : an Incident-Based Approach to Forensic Investigations”. In: *Springer* (2019).
- [113] D. Sahoo, C. Liu, and S.C.H. Hoi. “Malicious URL detection using machine learning: A survey”. In: *CoRR abs/1701.07179, arXiv:1701.07179* 1.1 (2019), pp. 1–37.

- [114] S. Schrittwieser et al. “Protecting Software through obfuscation: Can it keep pace with progress in code analysis?” In: *ACM Computing Surveys (CSUR)* 49.1 (2016), 4:1–4:40.
- [115] K. Schutt et al. “Early detection of malicious behavior in JavaScript code”. In: *Proceedings of the 5th ACM Workshop on Artificial Intelligence and Security (AISec’12), Raleigh, NC, USA*, 12 (2012), pp. 15–24.
- [116] S. Sebastian et al. “A study and review on code obfuscation”. In: *World conference on futuristic trends in research and innovation for social welfare (WCFTR’16)* (2016), pp. 1–6.
- [117] T. Serafim and T. Kachalov. “JavaScript Obfuscator Tool”. In: *A free and efficient obfuscator for JavaScript (including ES2017) - A Web UI tool to the excellent (and open source) javascript-obfuscator@0.21.0*, In: <https://obfuscator.io/>, Accessed on August 2019 (2019).
- [118] Y. Shirota and B. Chakraborty. “Visual Explanation of Mathematics in Latent Semantic Analysis”. In: *2015 IIAI 4th International Congress on Advanced Applied Informatics* (2015), pp. 423–428.
- [119] P. Skolka, C. Staicu, and M. Pradel. “Anything to hide? Studying mini-fied and obfuscated code in the web”. In: *In the Proceedings of the World Wide Web Conference (WWW’19), San Francisco, CA, USA, ACM* 4 (2019), pp. 1–11.
- [120] R. Socher et al. “Parsing natural scenes and natural language with recursive neural networks”. In: *In Proceedings of the 28th International Conference on Machine Learning (ICML- 11)* (2011), pp. 129–136.
- [121] Sonicwall. “Sonicwall Cyber Threat Report”. In: (2020). URL: <https://www.sonicwall.com/resources/2020-cyber-threat-report-pdf/>.
- [122] Positive Technologies. “Web Application Vulnerabilities: Statistics for 2018”. In: (2019).

- [123] B. Tellenbach, S. Paganoni, and M. Rennhard. “Detecting Obfuscated JavaScript from known and unknown obfuscators using Machine Learning”. In: *International Journal on Advances in Security* 9.3-4 (2016), pp. 196–206.
- [124] The, Majestic, and Million. “The million domains we find with the most referring subnets”. In: *In: <https://majestic.com/reports/majestic-million>, Accessed on August 2019* (2019).
- [125] The et al. “The million domains we find with the most referring subnets”. In: *In <https://majestic.com/reports/majestic-million> , Accessed on August* (2019).
- [126] The et al. “The million domains we find with the most referring subnets”. In: *In <https://majestic.com/reports/majestic-million>, Accessed on May* (2020).
- [127] TrendMicro. “Site Safety Center,’ With one of the largest domain-reputation databases in the world, Trend Micro’s web reputation technology is a key component of Trend Micro™ Smart Protection Network™”. In: *In: <https://global.sitesafety.trendmicro.com/> , Accessed May* (2020).
- [128] URLVoid. “Website Reputation Checker,’ This service helps you detect potentially malicious websites”. In: *In: <https://www.urlvoid.com/> , Accessed May* (2020).
- [129] VirusTotal. “Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community”. In: *In: <https://www.virustotal.com/gui/home/url> , Accessed May* (2020).
- [130] W3Techs. “Web Technology Surveys, Usage statistics of JS as a client-side programming language on websites”. In: *In <https://w3techs.com/technologies/details/cp-JS>, Accessed on October 2020* (2020).
- [131] B. Wang et al. “Evaluating Word Embedding Models: Methods and Experimental Results”. In: *APSIPA Transactions on Signal and Information Processing, arXiv:1901.09785* E19.8 (2019), pp. 1–13.

- [132] J. Wang et al. “Jsdc: A hybrid approach for javascript malware detection and classification”. In: *in Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM (2015), pp. 109–120.
- [133] W.-H. Wang et al. “A static malicious JavaScript detection using SVM”. In: *In Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE)* 40 (2013), pp. 21–30.
- [134] Y. Wang, W.-d. Cai, and P.-c. Wei. “A deep learning approach for detecting malicious JavaScript code”. In: *Security and Communication Networks*, Wiley (2016), pp. 1520–1534.
- [135] Y. Wang, J. S. Lee, and I. C. Choi. “Indexing by latent dirichlet allocation and an ensemble model”. In: *Journal of the Association for Information Science and Technology* 67(7) (2016), pp. 1736–1750.
- [136] H. Xu et al. “Click fraud detection on the advertiser side”. In: *in Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS)*, Springer, September, doi: 10.1007/978-3-319-11212-1_24, Wroclaw, Poland (2014), pp. 419–438.
- [137] W. Xu, F. Zhang, and S. Zhu. “The power of obfuscation techniques in malicious JavaScript code: A measurement study”. In: *7th International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE (2012), pp. 9–16.
- [138] W. Xu, F. Zhang, and S. Zhu. “The power of obfuscation techniques in malicious JavaScript code: A measurement study”. In: *7th International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE (2012), pp. 9–16.
- [139] Y. Xue et al. “Detection and classification of malicious JavaScript via attack behavior modelling”. In: *In Proceedings of the 2015 International Symposium*

on Software Testing and Analysis, ISSTA 2015, New York, NY, USA, 2015.
ACM. (2015), pp. 48–59.

- [140] B. Yadegari et al. “A generic approach to automatic deobfuscation of executable code”. In: *In the Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE SP’15)* (2015), pp. 674–691.
- [141] S. Yoo et al. “Two-phase malicious web page detection scheme using misuse and anomaly detection”. In: *International Journal of Reliable Information and Assurance, vol. 2, no. 1* (2014), pp. 1–9.
- [142] H. Yuan et al. “Detecting Phishing Websites and Targets Based on URLs and Web page Links”. In: *In Proceedings of the 24th International Conference on Pattern Recognition (ICPR’18)*, doi: 10.1109/ICPR.2018.8546262 (2018), pp. 3669–3674.
- [143] Y. Zhang, R. Jin, and Z.-H. Zhou. “Understanding bag-of-words model: a statistical framework”. In: *International Journal of Machine Learning and Cybernetics, 1(1-4)* (2010), pp. 43–52.
- [144] Y. Zhang, R. Jin, and Z.-H. Zhou. “Understanding bag-of-words model: a statistical framework”. In: *International Journal of Machine Learning and Cybernetics, 1(1-4)* (2010), pp. 43–52.
- [145] Y. Zou and K. Kontogiannis. “Towards a portable XML-based source code representation”. In: *In ICSE 2001 Work-shops of XML Technologies and Software Engineering (XSE)* (2001).

Appendix A

List of Publications

A.1 Journal Papers

1. S. Ndichu, S. Kim, S. Ozawa, T. Misu, K. Makishima, "A Machine Learning Approach to Detection of JavaScript-based Attacks Using AST Features and Paragraph Vectors," *Applied Soft Computing Journal*, Elsevier, vol. 84, ISSN 1568-4946, 105721, 2019, pp.105721 [Chapter 2]
2. S. Ndichu, S. Kim, S. Ozawa, "Deobfuscation, Unpacking and Decoding of Obfuscated Malicious JavaScript for Machine Learning Models Detection Performance Improvement," *CAAI Transactions on Intelligence Technology Journal*, ISSN 1751-8644, 5, (3), 2020, pp. 184-192 [Chapter 3]
3. S. Ndichu, S. Kim, S. Ozawa, T. Ban, T. Takahashi, D. Inoue, "Web-based Attack Detection by Feature Selection using Tree SHapley Additive exPlanations," *Computers & Security Journal*, Elsevier, Submitted Mar 03, 2021 [Chapter 4]

A.2 International Conferences

1. S. Ndichu, S. Ozawa, T. Misu, K. Okada, "A Machine Learning Approach to Malicious JavaScript Detection using Fixed Length Vector Representation,"

IEEE/INNS International Joint Conference on Neural Networks (IJCNN), Rio de Janeiro, Brazil, July 8-13, 2018, pp. 4254–4261 [Chapter 2]

2. S. Ndichu, S. Kim, S. Ozawa, "Extended Abstract: A Machine Learning Model Performance Improvement Approach to Detection of Obfuscated JavaScript-based attacks," Network and Distributed Systems Security Symposium (NDSS'20), San Diego, CA, USA, February 23-26, 2020 [Chapter 3]

A.3 Domestic Conferences

1. S. Ndichu, S. Ozawa, T. Misu, K. Okada, "Detection of Malicious JavaScript Contents using Doc2Vec Feature Learning," 2018 Symposium on Cryptography and Information Security (SCIS'18), Niigata, Japan, January 23-26, 2018, pp. 1-7 [Chapter 2]
2. S. Ndichu, S. Kim, S. Ozawa, T. Misu, K. Okada, "Extended Abstract: Detection of JavaScript-based Attacks Using Doc2vec Feature Learning," International Workshop on Security (IWSEC'18), Sendai, Japan, September 3-5, 2018 [Chapter 2]
3. S. Ndichu, S. Kim, S. Ozawa, "Extended Abstract: Improvement in Detection of Obfuscated Malicious JavaScript Parts through Simplification and Decoding," International Workshop on Security (IWSEC'19), Tokyo Institute of Technology, Tokyo, Japan, August 28-30, 2019 [Chapter 3]

Doctoral Dissertation, Kobe University

“A Machine Learning Approach to Web-based Cyber-Attack Detection

(ウェブ媒介型サイバー攻撃検知への機械学習アプローチ)”, 119 pages

Submitted on July, 03, 2021

The date of publication is printed in cover of repository version published in Kobe University Repository Kernel.

© WANGARI SAMUEL NDICHU

All Right Reserved, 2021