FINAL YEAR PROJECT

Detecting Malicious JavaScript Through Static Analysis

Author:
Maxsuel Wisley de Souza Trajano

Supervisor:
Dr. Lahcen OUARBYA

A thesis submitted in fulfilment of the requirements for BSc Computer Science Degree

Goldsmiths, University of London
May 6, 2022

Abstract

The Internet has become a part of every aspect of our lives. This popularity has made the Internet an attractive place for cybercriminals to distribute malware, with JavaScript frequently used as the initial attack vector. This project studies the detection of malicious JavaScript through *Static Analysis*. The author proposed a classification pipeline that used *Abstract Syntax Tree (AST)* features from collected JavaScript samples to train three machine learning models, namely *Support Vector Machine*, *Decision Tree* and *Random Forest*. The results showed that *Machine Learning* models combined with AST-based features can detect malicious JavaScript instances accurately and at a scale, and the reason behind *Random Forest* outperforming *Support Vector Machine* and *Decision Tree* models, in terms of *Precision*, *Sensitivity* and *Specificity*, lies in the inherent properties of the model.

Keywords: Malicious JavaScript, Machine Learning, Abstract Syntax Tree, Static Analysis

Acknowledgements

First and foremost, I would like to thank God for giving me strength to finish this project.

I am extremely grateful to my mother and grandmother for their infallible support all these years.

Also, I would like to express my sincere gratitude to my supervisor, Dr. Lahcen Ouarbya, for providing guidance and feedback throughout this project.

Finally, I also wish to thank Dr.Tom Cole for his valuable feedback.

Contents

List of Figures

List of Tables

List of Abbreviations

AST	**A**bstract **S**yntax **T**ree
CSS	**C**ascading **S**tyle **S**heets
DT	**D**ecision **T**ree
ECMA	**E**uropean **C**omputer **M**anufacturers **A**ssociation
HTML	**H**yperText **M**arkup **L**anguage
HTTP	**H**ypertext **T**ransfer **P**rotocol
JSON	**J**avaScript **O**bject **N**otation
MSE	**M**ean **S**quared **E**rror
NB	**N**aive **B**ayes
RF	**R**andom **F**orest
SVM	**S**upport **V**ector **M**achines

Chapter 1

Introduction

The world has gone through unprecedented changes brought about by the pandemic. At a time when social distancing laws and national lockdowns were the norms, the Internet became more than ever an essential part of our lives. Accordingly, the number of Internet users has increased rapidly: as of January 2022, there are approximately 4.95 billion users [63], or 62.5% of the world's population. That is an increase of more than 192 million users (+4.0%) compared to 2021 [62]. This tremendous growth of Internet usage makes it an attractive place for cybercriminals. As such, cybercrime has been on the rise, disrupting the lives of individuals and organizations around the world. There have been several high-profile cyberattacks since the pandemic began, affecting all sectors of our society: the Financial [67] [69], the Government [68], the Education [45], the Healthcare [48]. The monetary cost of cybercrime was estimated at around $945 billion in 2020 [73].

JavaScript was originally developed to enhance the interactivity of web pages and to improve user experience. Nevertheless, it can also be used for malicious activities. In fact, one of the most common initial attack vectors used to spread malware relies on malicious JavaScript payloads [66].To conceal the maliciousness of a JavaScript payload, attackers apply one or more different obfuscation techniques, which makes most signature-based detection fail or give an unreasonable number of false positives and false negatives.

Approximately 97.9% [41] of websites worldwide use JavaScript as their client-side programming language. Thus, considering the sheer number of JavaScript files and the ever-increasing complexity of the obfuscation techniques, the task of manually inspecting obfuscated files is infeasible. Prior research has been conducted to create tools to automatically analyze and detect malicious JavaScript. [66] [71] [51]. Typically, these tools use the abstraction of the source code with machine learning algorithms to detect recurring patterns that are predictive of malicious intent.

1.1 Aims and Objectives

The primary purpose of this project is to study the effectiveness of *Machine Learning* models trained on *AST-based* features for the task of detecting malicious obfuscated JavaScript. Specifically, the aim is to leverage the fact that benign and malicious JavaScript files have different programming constructs and use machine learning models to discriminate between benign and malicious JavaScript automatically and efficiently at a scale.

This leads to the following research question: *Can one effectively detect malicious JavaScript instances by analyzing the different syntactic structures and structural patterns found in the Abstract Syntax Tree?*

To answer the research question, the author proposed a fully static classification pipeline, which is the main contribution of this project, and can be succinctly broken down into the following components:

- Gathering benign and malicious samples of *JavaScript*: The benign samples are collected from the Internet by a web crawler written in *Python*. A subset of this data can be found at: 7.4.4. The malicious samples are freely available on Github [17].
- Creation of a novel feature set by performing *Static Analysis* on the *AST* of collected samples. This is achieved through a script written in *JavaScript*.
- Preparation of multiple datasets for future research. This is done with *Jupyter Notebook* in *Python*.
- Implementation of the following *Machine Learning* algorithms: *Support Vector Machines* (***SVM***), *Decision Tree* (***DT***), and *Random Forest* (***RF***). These models were implemented in *Python*.
- Comparison and evaluation of the performance of the models in terms of *Precision*, *Sensitivity* and *Specificity*.

1.2 Summary of the results

The result of the experiments conducted in this project shows that ***RF*** outperformed both ***DT*** and ***SVM*** with average *Precision*, *Sensitivity* and *Specificity* scores of 98.66%, 98.34% and 98.69%, whereas ***DT*** (94.49%, 93.92% and 94.30%), and ***SVM*** (84.60% 78.09% 80.94%), respectively. These findings mean that *Machine Learning* models combined with *AST-based* features are an effective method for detecting malicious obfuscated JavaScript at a scale, and the reason for ***RF*** outperforming the other models is due to the inherent nature of the model.

1.3 Outline

- **Chapter 2**: Contains the literature review in the field of detecting malicious JavaScript.
- **Chapter 3**: Describes the methodology used for the implementation. This includes the architectural choices for the classification pipeline.
- **Chapter 4**: Describes the implementation stage.
- **Chapter 5**: Consists of an in-depth evaluation of the experiments conducted in this project. Also, limitations and future works.
- **Chapter 6** The conclusion

Chapter 2

Background Research

The current chapter details the relevant technical background and literature. Firstly, a brief overview of the JavaScript language. Secondly, the different malicious code transformations is presented. Finally, the academic literature reviews the current approaches used to statically analyze and detect malicious JavaScript.

2.1 JavaScript

JavaScript was developed in 1995 by Brendan Eich at Netscape [18] to add dynamic functionality to websites. By 1996, the rising popularity of JavaScript pushed Netscape to submit a request for ECMA International to standardize the language. The following year saw the release the official standardized specification of JavaScript, which was called the ECMAScript. Nowadays, this scripting language is one of the core technologies of the Web, together with *HTML* and *CSS*, and it is used on both the client-side and server-side [54].

2.2 Code Transformation Techniques

The difference between benign and malicious JavaScript lies in its purpose. Well-intentioned but poorly designed JavaScript code can lead to vulnerabilities in web applications. On the other hand, malicious JavaScript code deliberately apply one or more obfuscation techniques to hide the maliciousness of JavaScript payloads to evade detection. However, it is important to acknowledge that code obfuscation does not imply maliciousness. Websites may choose to apply obfuscation techniques to protect intellectual property [61].The classification of the obfuscation techniques is based on the type of code transformations they perform: [76]

- **Randomization Obfuscation**: This technique consists of randomly adding or modifying elements of JavaScript code while maintaining its semantic structure intact. 2.1 shows the original code. 2.2 shows an example of inserting unnecessary whitespace, adding arbitrarily long comments, and replacing variables and function names with non-intuitive strings.

```
function greetings(name){
    alert('Hello ' + name);
}

var aName = 'Stranger';
greetings(aName);
```

Figure 2.1: Original code

```
function
// 0x7a0x650x770x6f
    vzq10x310x370//qwqx1023Zaq2
(_zeqzV10)
{ ///qwqx1 * 2
    alert(
        // ! qqxioqp1 /10
        'Hello '
        +
        // qqxioqp1 + 1
        _zeqzV10
    )}
var
            hlqjw
    = 'Stranger' // oo1pi3op1
vzq10x310x370(hlqjw)
```

Figure 2.2: Randomized Code

- **Data Obfuscation**: This technique exploits the dynamic characteristics of JavaScript to hide a plain text word to foil content matching detection. JavaScript allows the variable type to be inferred from the type of the value assigned to it [70], thus objects can be created/removed at runtime, making it difficult to limit the behavior of objects. [42].

```
alert('evil code')
```

Figure 2.3: Original code

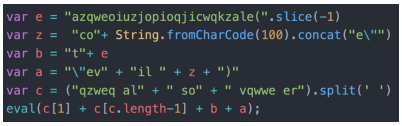


Figure 2.4: String Obfuscation Code Transformation

- **Encoding Obfuscation**: This technique changes a string by using standard encoding (ASCII, Unicode, or hexadecimal) or user-defined encoding/decoding functions.

Figure 2.5: Encoding Obfuscation Code Transformation

- **Logic Structure Obfuscation**: This technique relies on adding unnecessary conditional constructs (control-flow transformation obfuscation) to manipulate the execution paths without affecting the semantics. For example, the control-flow flattening technique [9] obfuscates loops by making the termination condition harder to reason with. Other techniques includes inserting irrelevant code (dead code injection) [10]

Figure 2.6: Logic Obfuscation Code Transformation

- **Obfuscation field reference**: JavaScript defines three syntaxes for accessing the properties of an object. The dot notation is defined as (object.identifier) , the square bracket notation is defined as (object[“identifier”]).The expression inside the square bracket notation must evaluate to a string. Strings are a data type in JavaScript, thus they can be created and manipulated at runtime, using the obfuscation strategies mentioned above.

```
// dot notation
document.write("evil code")

// bracket notation
document["write"]("evil code")

// Object  field Obfuscation Code Transformation
var zp9 = [{q: "w"}.q + {z: "r"}.z]
var piq = (1, "ma", true, "i") + (20, "qz", false , "t")
var uil = ["gold" , "val", 'laqw' , "upqoze"][(1000 - 997)].substring(-1)
document[zp9 + piq + uil]("evil code")
```

Figure 2.7: Object field Obfuscation Code Transformation

2.3 Literature Review

In 2009, Likarish et al [66] addressed the threat malicious obfuscated JavaScript posed to rule-based anti-malware by using machine learning to detect previously unseen instances of malicious scripts. Likarish's approach focused on training *Machine Learning* classifiers (*Naïve Bayes*, *Decision Tree*, *Support Vector Machine*) on the normalized frequency of each lexical token (e.g. keywords, white space, etc.) found on malicious obfuscated payloads as a feature to detect malicious patterns. The results of this experiment ranged from 60% to 90% of recall and precision, concluding that the use of deliberate feature selection with *Machine Learning* classifiers can result in highly accurate malicious JavaScript Detectors.

In 2010, Krueger et al. [71] proposed a generic method for the automatic detection of drive-by-download attacks. *Cujo* used the concept of $n-gram$, where n represented a fixed-length pattern, to extract JavaScript lexical tokens as features. An *SVM* classifier was then trained on this feature set to classify malicious JavaScript inputs, with results showing a high true positive rate of 90.2% and a low false positive rate of 0.001%. This work showed that the generic $n-gram$ feature extraction approach combined with *Machine Learning* systems could effectively detect drive-by-download attacks.

In 2011, Curtsinger et al. [51] explored how a *Bayesian* classifier could be trained on the features of the JavaScript *AST.Zozzle* relied on extracting features that were predictive of benign and malicious intent. Specifically, a feature contained two elements: the context in which it appeared (e.g. a loop, a condition) and the string of the *AST* node. The experiments showed low false positive rates between 0.01% and 4.5% and high false negative rates between 5% and 11%, thus proving that using the contextual information found at the *AST* level could be used to perform JavaScript malware detection.

More recently, Fass et al. [52] developed a *Random Forest* detection system trained on the syntactic units found in the *AST*. *JaSt* implemented the $n-gram$ technique to extract programming constructs as they originally appeared in the source code of the sample. The experiments showed a true positive rate of 99.46% and a true negative rate of 99.48%, showing that syntactical analysis outperforms lexical analysis, due to the fact that the latter lacks contextual information.

Chapter 3

Methodology

This chapter covers the methodology used in the implementation and subsequent evaluation. Each section details the chosen technologies and the reasons behind those choices. The methods adopted in this thesis are consistent with the steps of the "Universal Workflow of Machine Learning" [49]

1. Define the Problem
2. Assembling a dataset
3. Choose a metric to measure success
4. Establish an Evaluation Protocol
5. Prepare your data
6. Develop a baseline model
7. Develop a model that outperforms the baseline
8. Progressively improve the model architecture with hyperparameter tuning

3.0.1 Problem Definition

Building a static pipeline to detect malicious JavaScript is a binary classification task. Formally, given a labeled set $\mathbb{X}$ of input-output pairs x and y with N training samples $\mathbb{X} = \{(x_i, y_i)\}_{i=1}^{N}$, the goal is to learn a function $f\colon X \to Y$. Each training input x_i is a R-Dimensional vector representing the characteristics of a particular instance. Each output $y_i \in \{0, 1\}$ is either a benign or a malicious instance, respectively.

3.1 Assembling a Dataset

The experiments of this project required a dataset containing both benign and malicious JavaScript payloads. While the malicious raw data was freely available on GitHub [17], the author had to acquire the benign raw data. To achieve this, the author built a web Crawler to scrape JavaScript samples from the most popular websites in the world.

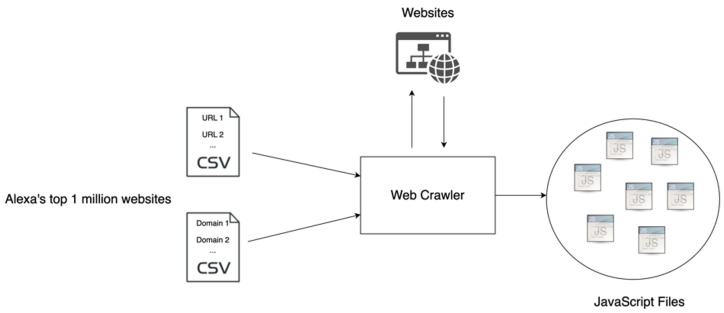


Figure 3.1: Web crawler Architecture

To meet the specifications of this component, the author used the Scrapy [33] framework. The author made this decision because Scrapy abstracts much of the complexities of retrieving and extracting data of the Internet, its ability to handle poorly formatted HTML[1], and its asynchronous nature.[2] Scrapy is a robust, high-level web crawling framework written in *Python* [30]. The architecture of Scrapy uses "spiders" to extract large amounts of data from various sources efficiently. Spiders are user-defined classes that set out the instructions for crawling and parsing the contents of a webpage.

The crawling cycle [35] starts with a HTTP Request generated by a given URL and domain. If this request gets a Response status 200, it returns the response body, which contains the HTML page.

```
2022-04-15 18:18:44 [asyncio] DEBUG: Using selector: KqueueSelector
[s] Available Scrapy objects:
[s]   scrapy     scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s]   crawler    <scrapy.crawler.Crawler object at 0x7fabda33d280>
[s]   item       {}
[s]   request    <GET http://www.google.com>
[s]   response   <200 https://www.google.com/?gws_rd=ssl>
[s]   settings   <scrapy.settings.Settings object at 0x7fabda33d430>
[s]   spider     <DefaultSpider 'default' at 0x7fabda91b8e0>
[s] Useful shortcuts:
[s]   fetch(url[, redirect=True]) Fetch URL and update local objects (by default, redirects
are followed)
[s]   fetch(req)                  Fetch a scrapy.Request and update local objects
[s]   shelp()           Shell help (print this help)
[s]   view(response)    View response in a browser
2022-04-15 18:18:45 [asyncio] DEBUG: Using selector: KqueueSelector
In [1]: response.body[:100]
Out[1]: b'<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en-GB
"><head><meta char'
```

Figure 3.2: Crawling the first 100 HTML characters of a web page

[1] commonly found at large
[2] stackoverflow

The Document Object Model (DOM) [13] is the representation of an HTML page in a tree-like data structure. The DOM provides an API for traversing and manipulating the nodes in the tree.

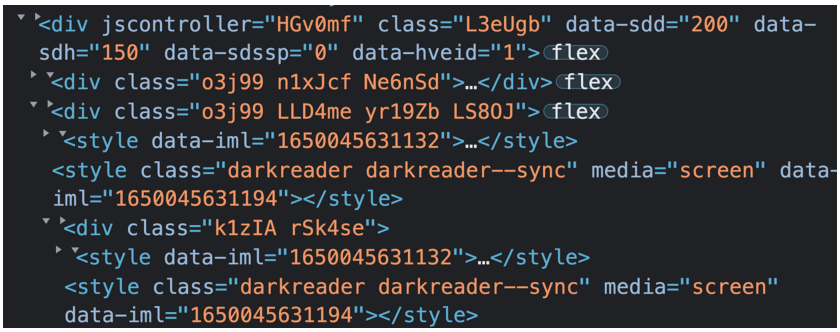


Figure 3.3: DOM representation

The final step is to extract the JavaScript code using CSS selectors [37].

```
In [17]: javascript_code = response.css('script::text').getall()[0]

In [18]: javascript_code[:400]
Out[18]: '(function(){window.google={kEI:\'86hZYpe-MNOdgQa-jqywBQ\',kEXPI:\'31\',kBL:\'Crt7\
'};google.sn=\'webhp\';google.kHL=\'en-GB\';})();(function(){\nvar f=this||self;var h,k=[];f
unction l(a){for(var b;a&&(!a.getAttribute||!(b=a.getAttribute("eid")));)a=a.parentNode;retu
rn b||h}function m(a){for(var b=null;a&&(!a.getAttribute||!(b=a.getAttribute("leid")));)a=a.
parentNode;return b}\nfunction n(a,b,c,d,g){var e="";c'
```

Figure 3.4: Extracting JavaScript code from Google

3.2 Choose a metric to measure success

Taking into the account the class imbalance problem the dataset of this project suffers, the author chose three different evaluation metrics to quantify the performance of the models: *Precision*, *Sensitivity* and *Specificity*.These metrics are robust against class imbalance since the ratio of correct classification is done in the context of each class. [58]. An analysis of the confusion matrix [21] is necessary to understand the terms used:

		Predicted Class	
		Benign	Malicious
True Class	Benign	TN	FP
	Malicious	FN	TP

Figure 3.5: Confusion Matrix

Where:

True negative (TN): The classifier correctly classified benign instances as benign.

False Positive (FP): The classifier incorrectly classified benign instances as malicious.

False-negative (FN): The classifier incorrectly classified malicious instances as benign.

True positive (TP): the classifier correctly classified malicious instances as malicious.

Precision(Positive Predictive Value): measures the ratio of positive instances correctly classified in relation to all instances classified as positive. The precision equation is:

$$Precision = \frac{TP}{TP + FP}$$

Sensitivity (True Positive Rate): measures the ratio of positive instances correctly classified as positive, i.e. the malicious samples correctly classified as malicious in relation to all malicious samples. The Sensitivity equation is:

$$Sensitivity = \frac{TP}{TP + FN}$$

Specificity (True Negative Rate): measures is the ratio of negative instances correctly classified as negative, i.e. the benign samples correctly classified as benign in relation to all benign samples. The Specificity equation is:

$$Specificity = \frac{TN}{FP + TN}$$

The *Confusion Matrix* [8] will be used to give context to the results of these metrics.This project used *scikit-learn* built-in function for *Precision* [34] and *Confusion Matrix* and implements two functions for *Sensitivity* and *Specificity*.

3.3 Establish an Evaluation Protocol

An evaluation protocol is necessary to ensure the models are neither *underfitting* nor *overfitting* and will perform well on unseen data. The author chose *Cross-validation* as the method to evaluate the performance of the models.

Cross-validation is a resampling method that randomly divides the set of instances into K subsets/folds. The first fold a held-out/validation set and the model is trained on the other $k - 1$ folds [60]. The loss function *Mean Squared error* computes the mean of the squared difference between the predicted values and the true values of each fold k times. For each k_i the model estimates the test error, $MSE_1, MSE_2, ...MSE_k$:

$$CV(_k) = \frac{1}{k}\sum_{i=1}^{k}(x_i - y_i)_i^2$$

Despite k-fold cross-validation being widely used as the standard error-estimator method, it is not appropriate to use it for imbalanced datasets. Since the dataset used for this project

is in the ratio of 55:45, the random distribution of the classes will vary in the folds, with the possibility of some folds containing more of the negative class(benign class) [58]. The solution to this problem is to use a technique called *Stratified Sampling* to ensure a balanced distribution of the classes in a k-fold every time.Literature supports k between 5 to 10, as empirical evidence shown there is a good bias-variance trade-off [57]. Hence, in this project, a *Stratified 5-fold cross-validation* $k = 5$ was implemented with *scikit-learn StratifiedKFold* [39] to check the performance of the models

3.4 Prepare the data

In this phase, the focus is on preparing the raw data for the models.

3.4.1 Feature Extraction

The purpose of this phase was to perform the *Static Analysis*. This technique examines the static characteristics of source code of without executing it. [44]. The idea was to explore the intermediate code representation (AST), to extract features (e.g. lexical tokens, syntactic units) that are predictive of benign and malicious intent. [46].

The *JavaScript* [23] language was the underpinning technology used to achieve these steps. Specifically, this thesis used *Node.js* [26] for the implementation stage. The reason for choosing this runtime environment lied on its ability to run *JavaScript* outside the web browser, its asynchronous nature, and the *NPM* libraries [25].

The *Acorn* library [1] was used to parse a *JavaScript* sample into an AST data structure. An AST is the hierarchical representation of the syntactic structure of the source code of a source code. Each node in the AST represents a programming construct. [46].

```
var greetings = 'hello'
```

Figure 3.6: Code Example

Figure 3.7: AST generation

Next, the *Estraverse* [14] library provided different depth-first search methods to traverse the *AST* nodes to extract the relevant features.

```
estraverse.traverse(ast, {
    enter: (node, parent) =>{
        if(node.type === "FunctionExpression"){
            if(parent?.type === "CallExpression"){
                newObj["funcSelfInvoking"] += 1

            }
            const body = node.body?.body
            for (const args of body){
                const value = args?.argument?.value
                if(typeof value === "string"){
                    newObj["funcReturnString"] += 1
                }
            }
        }
```

Figure 3.8: Traversing the tree to extract features

The final feature set consisted of:

- **Randomization Obfuscation**: the length of a sample
- **Data Obfuscation** & **Encoding Obfuscation**: The creation and manipulation of Strings and Objects
- **Syntactic Units**: The syntactic elements found at the *AST*.
- $1 - gram$ **Syntactic Unit**: variable with the first assigned syntactic unit.

The features were structured as arrays of objects and were serialized into *JavaScript Object Notation JSON* [43] format. Also at this stage, features were quantified by the number of occurrences of the feature (represented by 0/1, where 0 means the absence and 1 means the occurrence).

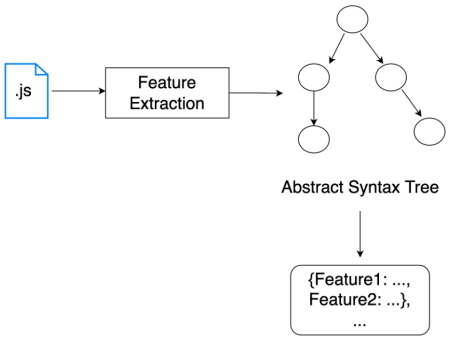


Figure 3.9: Feature Extraction Process

3.4.2 Feature Reduction

In this phase, the syntactic units that performed similar tasks or had similar abstract meanings were grouped together. This was done to reduce the complexity of the task.The same technologies used by the previous component are also used here.

3.4.3 Dataset Preparation

This phase produced the final dataset used by the models. The chosen technology to achieve this is *Python* [30], as the simplicity of the language and the wide range of data science libraries made it very appealing for prototyping. The Python's JSON module was used to read the contents of the .JSON files produced during the feature extraction [19]. The Pandas [28] and Numpy [27] libraries were used to build, clean, manipulate, and explore the dataset. Matplotlib [22] and Seaborn [36] were used to explore the data. All the experiments of this section were run on Jupyter notebook [20]. The reason for choosing this web application was that it is one of the best tools for performing data science research.

3.5 Classification - Develop a baseline model

In the next two phases, the pre-processed data from the dataset preparation was used to train the models. To achieve this, the *Scikit-learn* [32] library was chosen. This library provided the necessary mathematical and *Machine Learning* functions, permitting easily splitting and shuffling the data, hyperparameter tuning, and off-the-shelf methods to estimate the performance across the models.

3.5.1 SVM

The underlying idea of Support Vector Machines ***SVM*** is to find a computationally efficient method for splitting the hyperplanes in a high dimensional feature space [50]. Given a n-dimensional space with $X = (X_1, X_2, ..., X_n)$ inputs , a hyperplane is a flat affine subspace of dimension $n-1$(i.e. a straight line) that divides the n-dimensional space into two halves.The mathematical equation of a hyperplane in a n dimensional space is [60]:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + ..., \beta_n X_n = 0$$

Where β_0 is the intercept and $\beta_1, \beta_2 ... \beta_n$ are the slopes of the hyperplane and $X = (X_1, X_2, ..., X_n$ are the input data points in this $n-1$-dimensional space. As figure 3.10

illustrates, given the set of training observations $x_i \mapsto y_i \in \{0, 1\}$, an unknown input x_i^* either lies above:

$$\beta_0 + \beta_1 X_1^* + \beta_2 X_2^* ... \beta_n X_n^* > 0 \text{ if } y_i = 1$$

or below the line

$$\beta_0 + \beta_1 X_1^* + \beta_2 X_2^* ... \beta_n X_n * < 0 \text{ if } y_i = 0$$

More generally,

$$\beta_0 + \sum_{i=1}^{N} \beta_i x_i^*$$

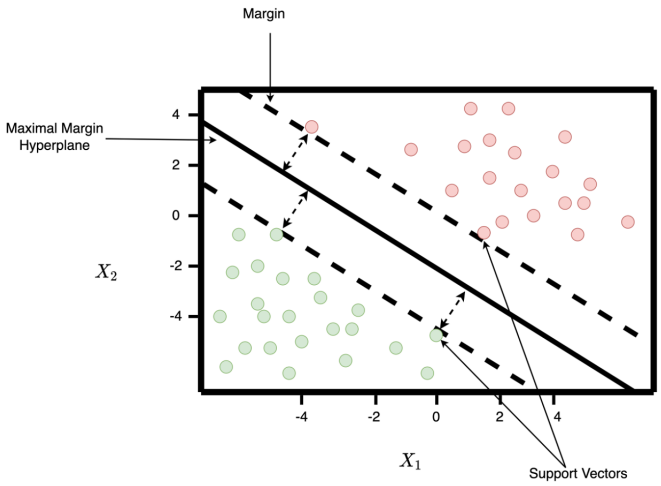


Figure 3.10: SVM - Maximal Margin Hyperplane

If x_i^* is near 0, then x_i^* is close to the hyperplane, thus making it hard to classify which class it belongs to. The intercept and slope that maximizes the distance between the hyperplane and the closest training inputs is called the *Maximum Margin Hyperplane*. [75]. The training inputs closest to the Maximum Margin Hyperplane are called *support vectors*, since they are vectors in n-dimensional space that defines the maximum margin hyperplane. An issue arises when the data is not entirely separable by a Hyperplane. As the figure bellow illustrates:

In this scenario, the option is to relax the constraint of having a Hyperplane that perfectly separates the classes. Allowing observations to violate the margin, or even the hyperplane, is called the *Soft Margin Classification* The model misclassification error may increase, but it will generalize better. Furthermore, there is a need to use a *non-linear hyperplane*. The decision boundary of non-linear classifications is defined by a *Kernel function* K commonly known as the *Kernel Trick*. This function measures the similarities between two training inputs. In this project, the function that quantified the similarities of training instances was the *Polynomial Kernel* [65]:

$$K(x_i, x_i^*) = (1 + \sum_{i=1}^{N} x_{ij} x_{ij}^*)^d)$$

Where, d is a positive integer and $d > 1$ allows non-linear decision boundaries. Combining support vectors with non-linear functions results in the technique called Support Vector

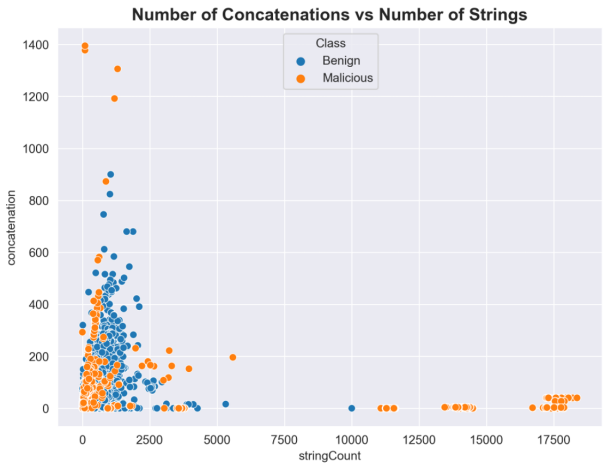


Figure 3.11: Data not linearly separable

Machines. The said attributes of ***SVM*** were the reason for choosing it as the baseline algorithm, and this project implemented the *scikit-learn Support Vector Classifier(SVC)* [7].

3.6 Develop a model that outperforms the baseline

3.6.1 Decision Trees

A Decision Tree ***DT*** comprise a series of tests, where each test compares an attribute against a decision rule. [64]. The hierarchical structure of ***DTs*** makes them highly intuitive: the topmost node is the root of the tree, non-leaf nodes along the tree are the predictors and the branches represent the outcome of the tests (CART algorithm). Leaf nodes represent the class variables $y \in \{0, 1\}$ [72]. The classification of new instances occurs by traversing down the tree from the root node to a leaf node, according to the outcome of tests. In this project, the ***DT*** created a series of tests to determine if an instance is benign or malicious.

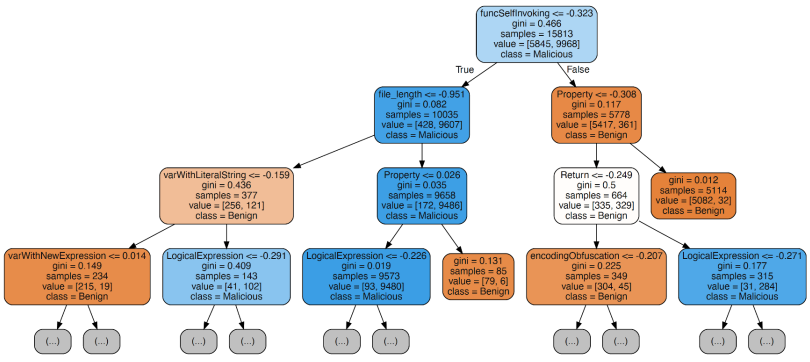


Figure 3.12: Decision Tree Graph

The Classification and Regression Tree *CART* algorithm [74] is particularly important, since this project used *scikit-learn DecisionTreeClassifier* [12] and scikit implements an optimized version of *CART* [11]. This algorithm uses a discrete splitting function, which recursively searches for the optimal feature upon which to perform the split (Gini Index). The Gini impurity measure calculates the best split at each node and is defined as [56] :

$$G_i = 1 - \sum_{i=1}^{n} R_{i,k}^2$$

Where, R is the ratio of class k instances in the i^{th} node. After that, each node further partitions the training set into smaller subsets until either all instances in a subset are homogeneous or a stopping criterion is satisfied [56].

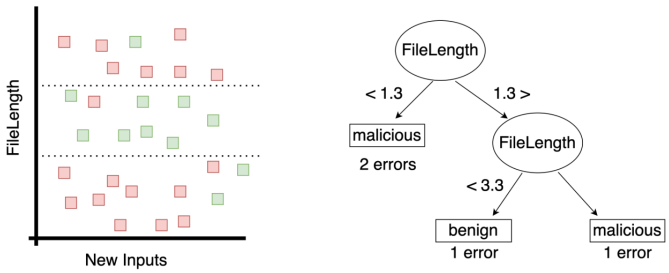


Figure 3.13: Decision Tree after two splits

The typical stopping criterion are reducing the depth of the tree, or the number of instances required in a node before a further split. Poorly chosen stopping conditions can heavily impact the ***DTs*** ability to generalize, as attempting to find the "optimal tree" only leads to *overfitting*. Rivest et al. has shown that constructing an optimal tree with$N$ training inputs is NP-Complete [59]. Thus, finding the optimal decision tree is only feasible in small problems. The characteristics of ***DT*** mentioned above were the reason for choosing it as the model that outperforms the baseline.

3.6.2 Random Forest

Random Forest ***RF*** is an ensemble learning method that aggregates a collection of randomly constructed ***DTs***. [74]. The approach of using the same classifier (***DT***) on a random subset of the training set is called *Bagging*, short for *Bootstrap Aggregation* [47]. Formally, given a set of inputs $\{(X_1, X_2, ..., X_i)\}$ and outputs $\{(Y_1, Y_2, ..., Y_i)\}$, where i is the number of equally distributed inputs, the goal is to construct a predictor $\hat{y}$: $X \rightarrow Y$ [55]. Each $\hat{y} \rightarrow (x_i, y_i) \in \{0, 1\}$.The predictor $\hat{y}_{\text{rf}}$ is constructed by an aggregation of ***DTs*** $(DT_1, DT_2, ...DT_j)$ in the following manner [57]:

1. Let J be the number of Trees
2. for $j = 1$ to J do
   (a) pick a bootstrap sample S of size T
   (b) Use S to Build a tree DT_j by recursively by repeating the following steps:
      i. Select n features at random from f features
      ii. Choose the best feature to perform the split in n
      iii. Split the node into two sub-nodes
3. Construct the RF classifier $\hat{y}_{\text{rf}}$ from the ensemble of Trees $\{DT_j\}_1^J$
4. Output $\hat{y}_{\text{rf}}$

The classification of a new unlabed instance x_j^* occurs by each individual $\hat{y}_{\mathrm{rf}j}$ model in the ensemble casting a vote in a prediction pool. The new unlabed instance is assigned to the class with the majority of votes:

$$\hat{y}_{\mathrm{rf}}(x_j^*) = \text{majority vote}\{\hat{y}_{\mathrm{rf}j}(x_j^*)\}_1^J$$

The node splitting randomness produces a greater tree variety and together with the majority voting strategy makes ***RF*** one of the most powerful *Machine Learning* algorithms. In this project, the ***RF*** was implemented with *scikit-learn* RandomForestClassifier [31].

3.7 Progressively improve the model architecture with hyperparameter tuning

All of the classification models have their own parameters and hyperparameters. Parameters are configurations that are internal to the model and are learned during the training phase. On the other hand, Hyperparameters are parameters specified outside the training phase whose configuration is used to customize the model's learning process to improve performance.

Firstly, each individual hyperparameter is plotted to address *underfitting* and *overfitting* concerns. Then, instead of manually exploring all the possible permutations of these hyperparameters, the author chose the automate this process with GridSearchCV [15]. This tuning method exhaustively examines the specified parameters for an estimator.The function takes the following parameters:

- **estimator**: instance of the model
- **para_grid**: search space: hyperparameter_name : [$value_1$, $value_2$,..., $value_n$]
- **cv**: cross-validation splitting strategy

The search space varies according to the model. The cross-validation method used StratifiedKFold $k = 5$, and the reason for choosing it is explained in 3.3.

Chapter 4

Implementation

This chapters details the chronological steps of the implementation stage. The code for the implementation can be found at:
`https://github.com/MaxsuelT/FYP-Detecting-Malicious-Javascript`

4.1 Assembling a Dataset

4.1.1 Data Collection - Benign

To overcome the initial code duplication issue 7.2, the web crawler was designed to download external JavaScript files found in the source attribute of script tags *within* HTML documents. [16].

```
rules = (
    Rule(LinkExtractor(
        tags= ('script'),
        attrs=('src')
    ),
    callback='parseScript',
    errback='errback_'
    ),
)
```

Figure 4.1: Defining the scraping rules

```
def parseScript(self, response):
    path = os.path.join(
        os.getcwd(), 'benign_scripts_1/')
    filename = path + self.get_filename(response.url)

    if (len(self.list_files()) >= 8000):
        raise CloseSpider('Stop right there')
    else:
        if not os.path.exists(filename):
            with open(filename, 'wb') as fobj:
                fobj.write(response.body)
                print(f'filename - {filename} saved!')
```

Figure 4.2: Scraping the code from external JS files

The data (URLs, Domains) used to feed the web crawler came from a dataset containing Alexa's top 1 million websites [4]. The pre-processing steps used Python's built-in module for URL handling [29], to add the missing scheme and network location for URLs.

```
def parser(url):
    url_obj = {
        'scheme': 'http://',
        'netloc': 'www.'
    }
    if url.startswith(url_obj['scheme'] + url_obj['netloc']):
        return url[len((url_obj['scheme'] + url_obj['netloc'])):]
    elif url.startswith(url_obj['scheme']):
        return url[len(url_obj['scheme']):]

    elif url.startswith(url_obj['netloc']):
        return url[len(url_obj['netloc']):]
    else:
        return url
```

Figure 4.3: Parsing the data before feeding it to the web crawler

```
df.head()
✓ 0.1s
```

	Domains	Urls
0	netflix.com	http://www.netflix.com
1	google.com	http://www.google.com
2	ftl.netflix.com	http://www.ftl.netflix.com
3	microsoft.com	http://www.microsoft.com
4	prod.ftl.netflix.com	http://www.prod.ftl.netflix.com

Figure 4.4: The data

With this data available, several crawls were made with more than 80,000 unique samples of JavaScript acquired. Finally, to create a representative benign dataset, the author manually added the source code of open-source JavaScript games [6], as well as client-side libraries and web frameworks. [5]. It felt necessary to consider the different JavaScript design patterns found at large since unusual coding patterns could lead to the misclassification of benign samples.

4.1.2 Data Collection - Malicious

The JavaScript malware samples are from a JavaScript Malware Collection by Hynek Petrak [17]. The malware samples are an assortment of trojan, loaders, phishing, etc., and were collected over four years:

- 1936 – 12 files
- 2015 – 1086 files
- 2016 – 38,485 files
- 2017 – 278 files
- 2019 – 1 file

4.2 Preparing the data

4.2.1 Feature Extraction

Initially, the author built a set of features based on randomization , data and encoding obfuscation techniques. Regarding randomization obfuscation, the features focused on characteristics of the file structure, akin to the work of *Likarish*. The concerns were whether the

payload contained sufficient JavaScript code to reveal its intent, and the percentage of white space within a payload.

```
function countChars(file, func){
    const pattern = /[\s\+]/g;
    const array = file.match(pattern);
    if(func.name === "isEmpty"){
        const number =  (
            count(array,func) / file.length
        )
        return parseFloat(Number(number).toFixed(2))
    }
```

Figure 4.5: Function to calculate the percentage of empty space in a script

The data and encoding obfuscation features were influenced by the work of *Xu* 2.2 and were developed to cover JavaScript objects that used built-in functions to create (e.g. the number of strings in a payload) and manipulated strings (e.g.string concatenation, average string length).

```
function getStrLen (obj, arr){
    let avg_str_length;
    if(obj.string_count){
        avg_str_length =  Number(
            arr.map(word => word.length)
            .reduce((acc , word) =>{
            return acc + word
         },0) / arr.length).toFixed(2)
    }else{
        avg_str_length = 0
    }
    return parseFloat(avg_str_length)
}
```

Figure 4.6: Function to calculate the average string length

After the first experiment, the author became concerned that these features were limited to the patterns and trends in the current dataset that might not occur otherwise and decided to investigate other techniques.

As a result, this research began exploiting the idea of using *Machine Learning* algorithms to find the relationships between the features automatically. As shown by *Likarish* ,*Cujo* and *Zozzle*, *AST-based* systems are an effective means to identify specific patterns found in malicious code. While the said works extracted lexical units of the samples, this project focused solely on extracting syntactical units, similar to the work of *JaSt*. Hence, at this stage, the features comprised the syntactic entities found in the *AST* of JavaScript files (e.g. Literal, IfStatement, etc.).

```
if(node.type === 'IfStatement' ||
   node.type === 'SwitchStatement' ||
   node.type === 'SwitchCase'||
   node.type === 'ConditionalExpression'){
    const flow_type = 'ControlFlow'
    if(!nodeTypes.hasOwnProperty(flow_type)){
        nodeTypes[flow_type] = 1
    }else{
        nodeTypes[flow_type]++
    }
}
```

Figure 4.7: Extracting syntactic units. Note - feature reduction already happening at this stage

Despite showing promising results, this approach was flawed, since focusing only on the programming constructs to discriminate between benign and malicious could lead to a biased model. In the current dataset, there were syntactic units that would only ever appear in benign samples (e.g. ClassDeclaration, Import and Export expressions, etc.). Thus, leading the model to misclassify unknown malicious samples that contained such syntactic units as benign.

The next problem to address was that syntactic units lacked any real context. To overcome this, the author combined the *Cujo's* $n-gram$ analysis with *JaSt's* implementation. Instead of extracting lexical units such as in *Cujo*, the focus here was on extracting the syntactic units, akin to *JaSt's* work. Furthermore,*JaSt* implemented the $n-gram$ technique where $n = 1...7$, whereas this project implemented the $n-gram$ technique where $n = 1$. Thus, the features of this work comprised of the first syntactic element assigned to a variable, e.g. varWithNewExpression, varWithCallExpression, etc.

```
estraverse.traverse(ast, {
    enter: (node) =>{
        if(node.type === "VariableDeclarator"){
            const type = node?.init?.type
            if(type){
                if(type === "NewExpression"){
                    newObj["varWithNewExpression"] += 1
                }
```

Figure 4.8: 1-gram implementation

Finally, the final feature extraction process combined a subset of all the features mentioned above. The reason for doing this was to create a more diverse and representative set of features.Therefore, the final feature set contained an assortment of randomization, data, encoding obfuscation code transformations, syntactic units, and $n-gram$ with syntactic units.

4.2.2 Dataset Preparation

This component was responsible for preparing the dataset for the models.At all stages of this project, the first step was to load the benign and the malicious JSON data. After that, the next step was to use the data to create a DataFrame and label the target variable. As previously mentioned, benign instances are labeled as 0 and malicious instances as 1.

```
def getJson(path):
    filename = os.path.join(os.getcwd(),path)
    with open(filename, 'r+') as fileobj:
        data = json.load(fileobj)
    return data

data_benign_1 = getJson(json_benign_samples_1)
df_benign_1 = pd.DataFrame(data_benign_1)
df_benign_1['Class'] = 0
```

Figure 4.9: Example: Loading - Creating Dataframe - Labeling the data

In the preprocessing steps, the first problem found was the duplicate code issue reported in the Data Collection Component. It was then acknowledged that there was a need to build a more robust crawler to avert this problem.The next issue to address was the huge discrepancy in the number of the characters in the benign and malicious scripts. The file_length feature was used to ensure all rows were between 1000 and 100.000 characters.

```
df = df.drop(df[(df.file_length > 100000)].index)
total =  len(df)
df = df.drop(df[(df.file_length < 1000)].index)
total += len(df)
total
```

✓ 0.6s

```
45835
```

Figure 4.10: Drop all rows outside the conditions. Note over 45.800 instances were dropped

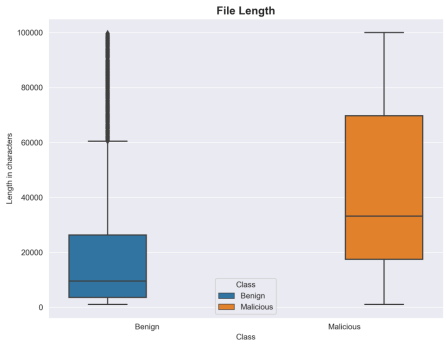


Figure 4.11: Box plot showing the distribution of the data

4.3 Classification

In this section, the pre-processed data was used to train and test the following models: ***SVM***, ***DT***, and ***RF***. The remaining of this section is divided into three parts: 4.3.1 details the initial steps common to all models. 4.3.2 details the hyperparameter tuning.4.3.3 highlights how the chosen metrics were used to evaluate the performance of the models.

4.3.1 Initial steps

Since the dataset was concatenated sequentially, the target variable was sorted by class. Thus, the first step was to load and shuffle [38] the dataset to ensure that the training, evaluating and testing sets were a representative of the original distribution of the data. Then the train_test_split [40] function was applied to split the data in 60:40 ratio.

```
filepath = os.path.join(os.getcwd(), 'data/final_experiment.csv')
df = pd.read_csv(filepath)
df['Class'] = df['Class'].apply(lambda val: 0 if val == 'Benign' else 1)
df = shuffle(df, random_state = 10)
```

Figure 4.12: Loading and shuffling the data

```
X = df.drop('Class', axis=1)
y = df.loc[:,'Class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
random_state=1, stratify=y)
```

Figure 4.13: Splitting the data into training and testing sets

4.3.2 Hyperparameter tuning

The parameters were the default values, whereas the hyperparameters were tuned by the GridSearch [15] function. The arguments and strategies used by this function are defined in 3.7:

Table 4.1: Parameter and Hyperparameters

Model	Parameters	Hyperparameters
SVM	kernel = "poly"	degree C
DT	criterion="gini" max_features = $\sqrt{f}$	max_depth max_leaf_nodes min_sample_split
RF	criterion="gini" bootstrap=True max_features $\sqrt{f}$	n_estimators max_depth max_leaf_nodes min_sample_split

SVM

Parameters:

- *Kernel*: "poly" - the type of decision boundary

Hyperparameters:

- d: the degree of the non-linear boundary
- C :the number of allowed margin violations

DT

Parameters:

- *criterion* :"Gini" - function to measure the quality of the split
- *max_features* : $\sqrt{f}$ 3.6.2 - the number of features to consider when looking for the best split

Hyperparameters:

- *max_depth* - maximum depth of the tree
- *max_leaf_nodes* - maximum number of leaf
- *min_sample_split nodes* - the minimum number of samples required to split an internal node

RF

This model shared the same parameter and hyperparameters as *DT*, with the exception of:
Parameters:

- *bootstrap*: True - bootstraps samples of the data

Hyperparameters:

- *n_estimators*: the number of trees in the forest

Hyperparameter Visualization

When visualizing the hyperparameters tuning process 5.1, the hyperparameters for:

- ***SVM***: The model is *underfitting* when loss for *degree* is increasing, whereas the model is *overfitting* for *C* when the loss is decreasing.
- ***DT*** & ***RF***: The models are *overfitting* for *max_depth* and *max_leaf_nodes* when the loss is decreasing. The *min_sample_split nodes* loss needs to increase slightly to avoid *overfitting*

4.3.3 Evaluation

The author used five important functions: *Precision* and *Confusion Matrix* are *scikit-learn* built-in functions, whereas *Sensitivity* and *Specificity* were created to calculate the said metrics. *Evaluate_model* was developed for purpose of performing cross-validation to check performance of the models on unseen data, and it took as its arguments the input(X), the output(y) variables. The *StratifiedKFold* began the 5-fold cross-validation step by shuffling the data only once to ensure the test data wont overlap, while maintaining the percentage of samples for each class. Then for each Train and Validation fold:

1. Fit the model
2. Predict the labels of the data
3. Get the precision, sensitivity, specificity, and the confusion matrix

```
def sensitivity(conf):
 return f'{(conf[1][1] / (conf[1][1] + conf[1][0]) *100):.2f}'

def specificity(conf):
 return f'{(conf[0][0] / (conf[0][0] + conf[0][1]) *100):.2f}'
```

Figure 4.14: Sensitivity and Specificity functions

```
def evaluate_model(X, y):

    sensitivity_scores = []
    specificity_scores = []
    confusion_matrix_scores = []
    precision_scores = []

    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=2)
    for train, test in  cv.split(X,y):
        X_train_strat = X.iloc[train]
        y_train_strat = y.iloc[train]
        X_test_strat = X.iloc[test]
        y_test_strat = y.iloc[test]

        svm_.fit(X_train_strat, y_train_strat)
        y_pred_strat = svm_.predict(X_test_strat)
        conf = confusion_matrix(y_test_strat,y_pred_strat)
        confusion_matrix_scores.append(conf)
        precision_scores.append(precision_score(y_test_strat,y_pred_strat))
        sensitivity_scores.append(sensitivity(conf))
        specificity_scores.append(specificity(conf))

    return (confusion_matrix_scores,
            precision_scores,sensitivity_scores,
            specificity_scores)
```

Figure 4.15: Evaluate_model function

Chapter 5

Evaluation

This is a research-oriented thesis, thus the answer to the research question and the progress made during each experiment is demonstrated through result gathering.Therefore, in this chapter, the author verifies the detection performance of the models to determine its effectiveness in relation to the research question.

Firstly, to address *underfitting* and *overfitting* concerns,the author checked the optimal hyperparameter grid, and these results are presented in the form of plots. Secondly, the optimal hyperparameters for each model are presented in the form of tables. Finally, the results for the experiments are presented in the form of tables that show the evaluation results for ***SVM***, ***DT*** and ***RF*** from the 5-fold cross-validation.

5.1 Analysis

There were three experiments conducted with ***SVM***, ***DT*** and ***RF*** models. First, the models were compared against one another in relation to their performance in the current experiment. This was done to determine whether the features captured the structure of the problem. Afterwards, each of the models were compared against their own previous performance. This was done to understand the impact the new features had in enhancing the overall detection accuracy. The performance of the models were measured in terms of *Precision*, *Sensitivity* and *Specificity* .

Table 5.1: Experiments Summary

Experiment No.	No.Instances	No.Features	Feature Description	Target Variable
1	2285 (1142 benign = 49.9% and 1143 malicious = 50.1%)	8	Randomization, Data and Encoding obfuscation 7.4.1	0 = Benign, 1 = Malicious
2	7358 (3679 benign = 50% and 3679 malicious = 50%)	64	Syntactic units, Data and Encoding Obfuscation 7.4.2	0 = Benign, 1 = Malicious
3	36797 (20023 benign = 54.4% and 16774 malicious = 45.6%) instances	34	Randomization, Data and Encoding Obfuscation, Syntactic units, 1-gram with Syntactic Units 7.4.3	0 = Benign, 1 = Malicious

5.1.1 Experiment 1

5.1.1.1 Hyperparameter Tuning

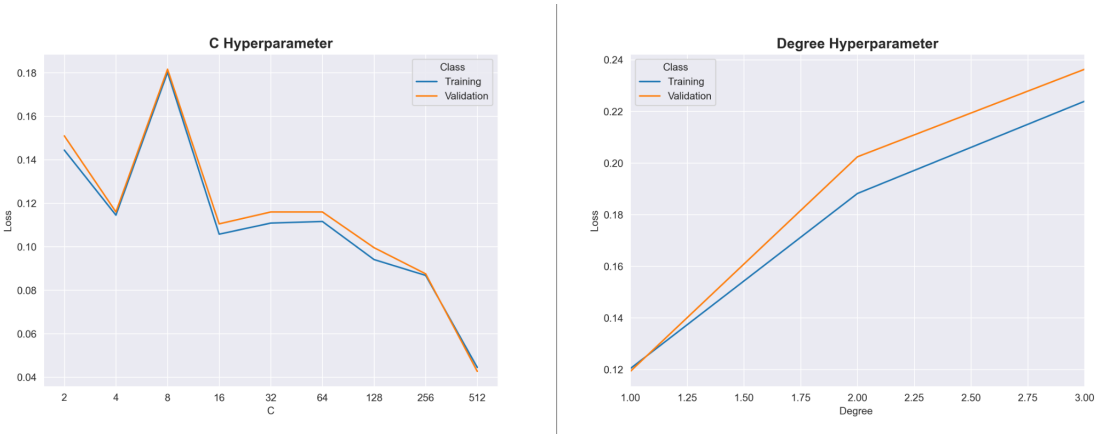


Figure 5.1: SVM Hyperparameter Tuning

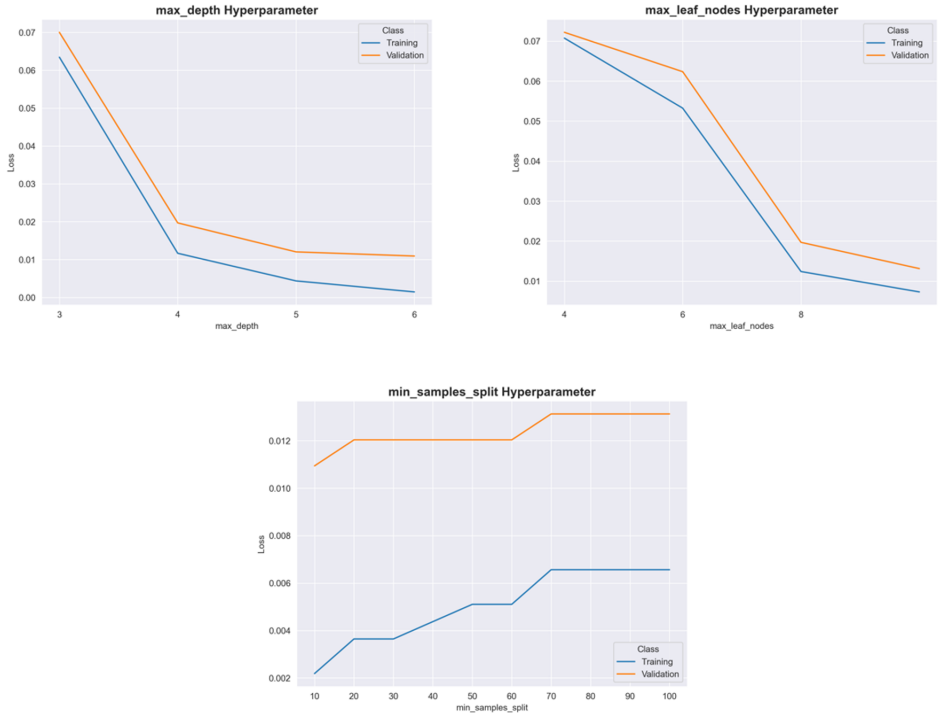


Figure 5.2: DT Hyperparameter Tuning

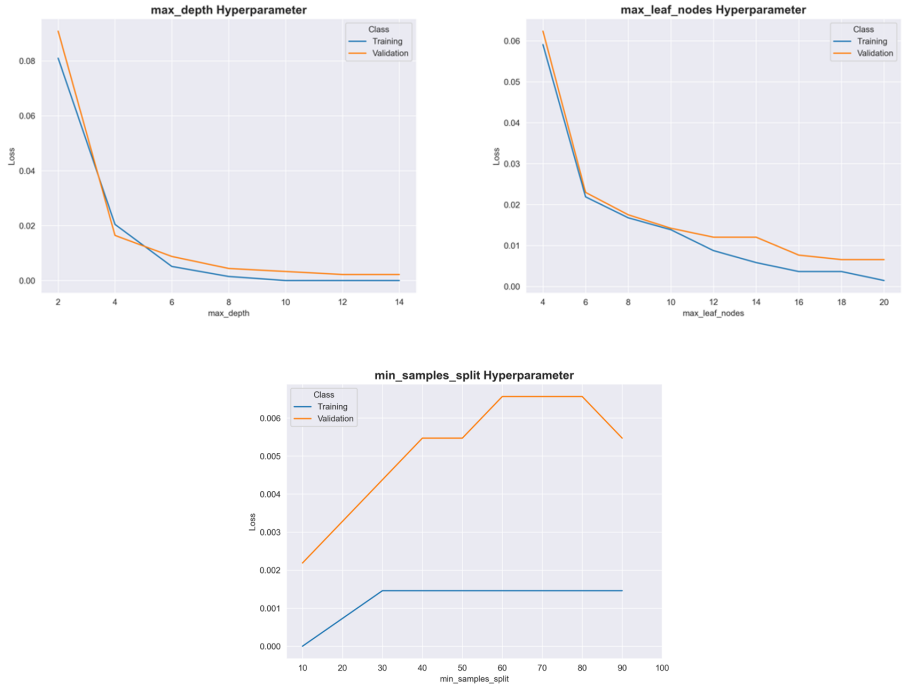


Figure 5.3: RF Hyperparameter Tuning

The optimal ranges are 4.3.2:

- ***SVM***: *C* 16 - 128. *degree*: 1 - 2
- ***DT***: *max_depth* - 4 - 6. *max_leaf_nodes* 6-8. *min_sample_split* - 20-30
- ***RF***: *max_depth* - 4 - 6. *max_leaf_nodes* 6-8. *min_sample_split* - 20-30

After tuning the hyperparameters, the models were fitted with:

Table 5.2: Parameters and Hyperparameters

Model	Parameters	Hyperparameterss
SVM	kernel = "poly"	degree=2 C=128
DT	criterion="gini" max_features = $\sqrt{f}$	max_depth = 5 max_leaf_nodes = 8 min_samples_split = 20
RF	criterion="gini" max_features = $\sqrt{f}$	n_estimators' = 75 max_depth = 4 max_leaf_nodes = 8 min_samples_split = 25

5.1.1.2 Cross-validation Results

Table 5.3: SVM

Fold No.	Precision	Sensitivity	Specificity	Confusion Matrix		
1	55.10	99.56	19.21		Benign	Malicious
				Benign	44	185
				Malicious	1	227
2	49.80	55.70	44.10		Benign	Malicious
				Benign	101	128
				Malicious	101	127
3	53.77	99.56	14.04		Benign	Malicious
				Benign	32	196
				Malicious	1	228
4	48.43	47.16	49.56		Benign	Malicious
				Benign	113	115
				Malicious	121	108
5	77.78	3.06	99.12		Benign	Malicious
				Benign	226	2
				Malicious	222	7
Average	56.98	61.01	45.21			

Table 5.4: DT

Fold No.	Precision	Sensitivity	Specificity	Confusion Matrix		
1	90.43	91.23	90.39		Benign	Malicious
				Benign	207	22
				Malicious	20	208
2	89.16	97.37	88.21		Benign	Malicious
				Benign	202	27
				Malicious	6	222
3	91.23	90.83	91.23		Benign	Malicious
				Benign	208	20
				Malicious	21	208
4	88.84	97.38	87.72		Benign	Malicious
				Benign	200	28
				Malicious	6	223
5	85.28	98.69	82.89		Benign	Malicious
				Benign	189	39
				Malicious	3	226
Average	88.99	95.10	88.09			

Table 5.5: RF

Fold No.	Precision	Sensitivity	Specificity	Confusion Matrix		
1	96.17	99.12	96.07		Benign	Malicious
				Benign	220	9
				Malicious	2	226
2	98.25	98.68	98.25		Benign	Malicious
				Benign	225	4
				Malicious	3	225
3	96.60	99.13	96.49		Benign	Malicious
				Benign	220	8
				Malicious	2	227
4	97.83	98.25	97.81		Benign	Malicious
				Benign	223	5
				Malicious	4	225
5	97.39	97.82	97.37		Benign	Malicious
				Benign	222	6
				Malicious	5	224
Average	97.25	98.60	97.20			

In this experiment, ***RF*** had the highest average scores in all metrics, outperforming ***DT*** by 8.26% in precision, 3.5% in sensitivity, and 9.11% in specificity.***SVM*** had the lowest scores across all metrics in this experiment, with sensitivity and specificity as low as 61.01% and 45.21%, respectively. The confusion matrices showed high rates of false negative and false positive in all folds, thus confirming this poor performance.

These results form the baseline for comparison in all experiments.

5.1.2 Experiment 2

5.1.2.1 Hyperparameter Tuning

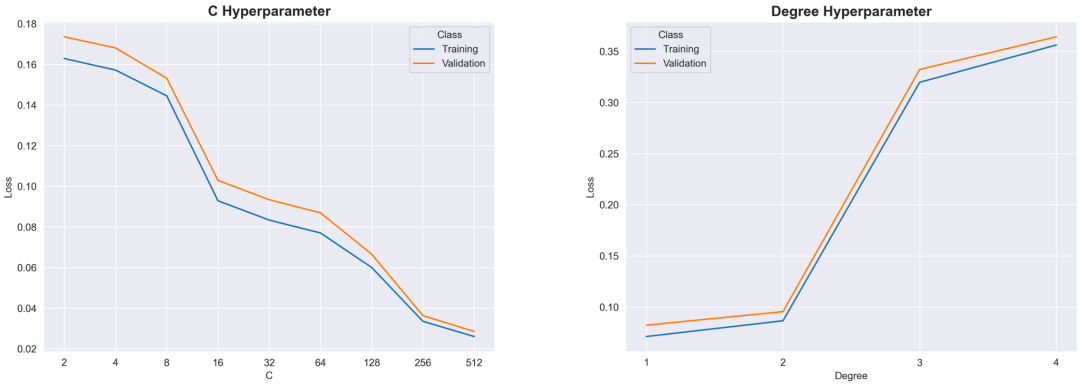


Figure 5.4: SVM Hyperparameter Tuning

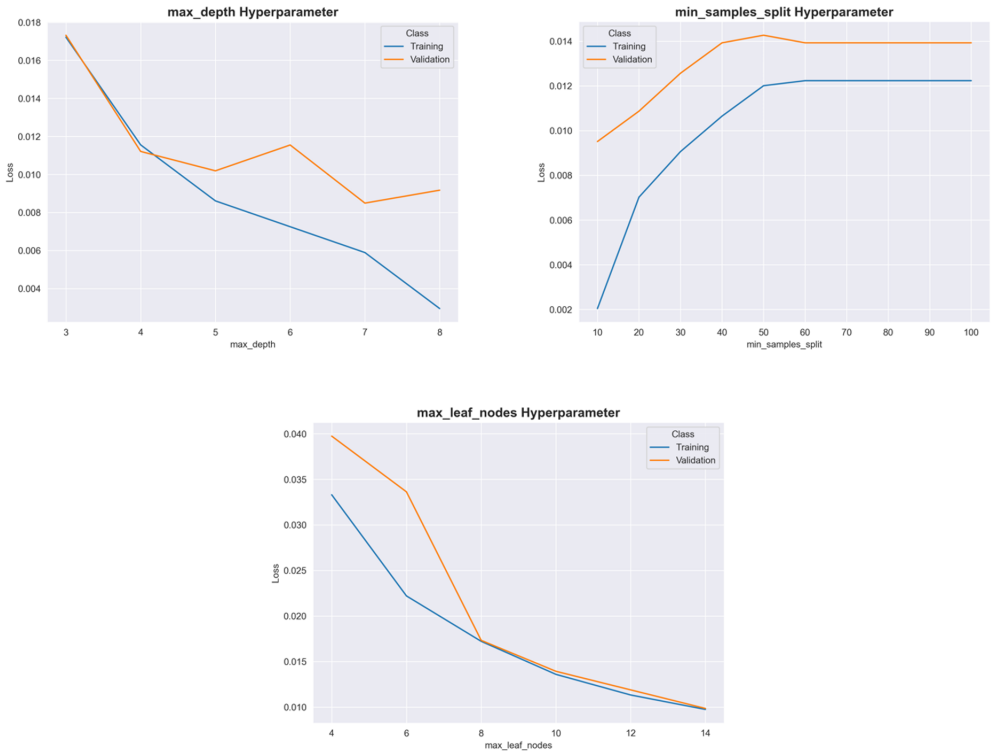


Figure 5.5: DT Hyperparameter Tuning

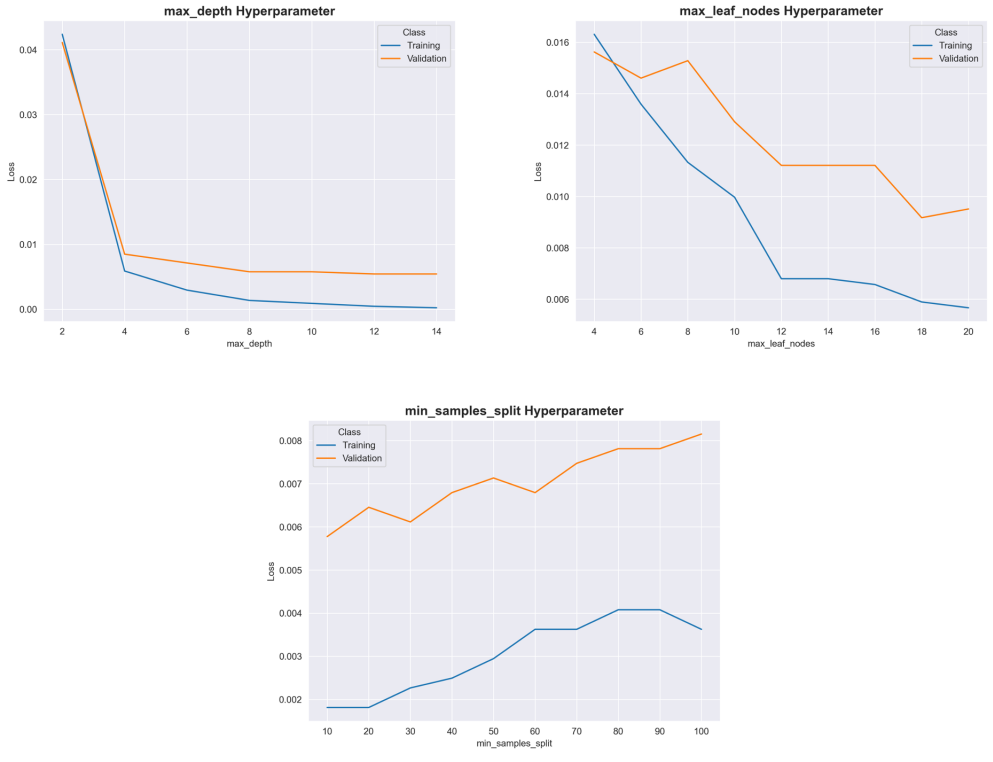


Figure 5.6: RF Hyperparameter Tuning

The optimal ranges are 4.3.2:

- ***SVM***: *C* 32 - 128. *degree*: 1 - 2
- ***DT***: *max_depth* - 4 - 6. *max_leaf_nodes* 6-8. *min_sample_split* - 20-30
- ***RF***: *max_depth* - 4 - 6. *max_leaf_nodes* 6-8. *min_sample_split* - 20-30

After tuning the hyperparameters, the models were fitted with :

Table 5.6: Parameters and Hyperparameters

Model	Parameters	Hyperparameters
SVM	kernel = "poly"	degree=1 C=128
DT	criterion="gini" max_features = $\sqrt{f}$	min_samples_split = 20 max_depth = 6, max_leaf_nodes = 8,
RF	criterion="gini" max_features = $\sqrt{f}$ bootstrap = True	n_estimators = 50 min_samples_split = 20 max_depth = 6 max_leaf_nodes = 8

5.1.2.2 Cross Validation Results

Table 5.7: SVM

Fold No.	Precision	Sensitivity	Specificity	Confusion Matrix		
1	100.00	87.23	100.00		Benign	Malicious
				Benign	736	0
				Malicious	94	642
2	99.22	86.82	99.32		Benign	Malicious
				Benign	731	5
				Malicious	97	639
3	99.39	89.27	99.46		Benign	Malicious
				Benign	732	4
				Malicious	79	657
4	99.84	86.80	99.86		Benign	Malicious
				Benign	735	1
				Malicious	97	638
5	99.84	86.55	99.86		Benign	Malicious
				Benign	734	1
				Malicious	99	637
Average	99.66	87.33	99.70			

Table 5.8: DT

Fold No.	Precision	Sensitivity	Specificity	Confusion Matrix		
1	98.98	92.26	99.05		Benign	Malicious
				Benign	729	7
				Malicious	57	679
2	99.12	91.30	99.18		Benign	Malicious
				Benign	730	6
				Malicious	64	672
3	98.71	93.75	98.78		Benign	Malicious
				Benign	729	9
				Malicious	46	690
4	89.04	94.27	88.30		Benign	Malicious
				Benign	649	86
				Malicious	37	699
5	98.16	94.42	98.23		Benign	Malicious
				Benign	723	13
				Malicious	41	694
Average	96.80	93.34	96.71			

Table 5.9: RF

Fold No.	Precision	Sensitivity	Specificity	Confusion Matrix		
1	99.72	97.42	99.73		Benign	Malicious
				Benign	734	2
				Malicious	19	717
2	99.44	96.33	99.46		Benign	Malicious
				Benign	732	4
				Malicious	27	709
3	99.31	97.28	99.32		Benign	Malicious
				Benign	731	5
				Malicious	20	716
4	99.72	98.50	99.73		Benign	Malicious
				Benign	734	2
				Malicious	11	724
5	99.86	97.55	99.86		Benign	Malicious
				Benign	734	1
				Malicious	18	718
Average	99.61	97.42	99.62			

In this experiment, ***RF*** again showed near perfect average precision, sensitivity and specificity scores of 99.61% , 97.42% and 99.62%, respectively. This indicates the model's exceptional ability to distinguish between benign and malicious samples. The confusion matrices illustrates this point: In all folds the number of false positives is no more than 5 , and the number of false negatives ranged between 11 to 27.***SVM*** performed approximately the same as ***RF*** with an average precision and specificity scores 0.05% and 0.08 higher. Furthermore, ***SVM*** outperformed ***DT*** by 2.86% and 2.99% higher in the same metrics. Nevertheless, ***SVM*** was outclassed in sensitivity by both ***RF*** and ***DT*** with scores 10.09% and 6.01% lower, respectively.

Significance of new Features

It is worth remembering that this experiment added syntactic units, Data and Encoding Obfuscation techniques as features.

SVM performance improved significantly between the first and second experiments. The average accuracy for each metric has improved by 41.16%, meaning that unlike the first experiment, SVM was able to draw a non-linear hyperplane that separated the two classes.***DT***'s performance showed an increased 4.89% in overall.***RF*** also performed well with the new features, with an average increase of by 1.19%.

5.1.3 Experiment 3

5.1.3.1 Hyperparameter Tuning

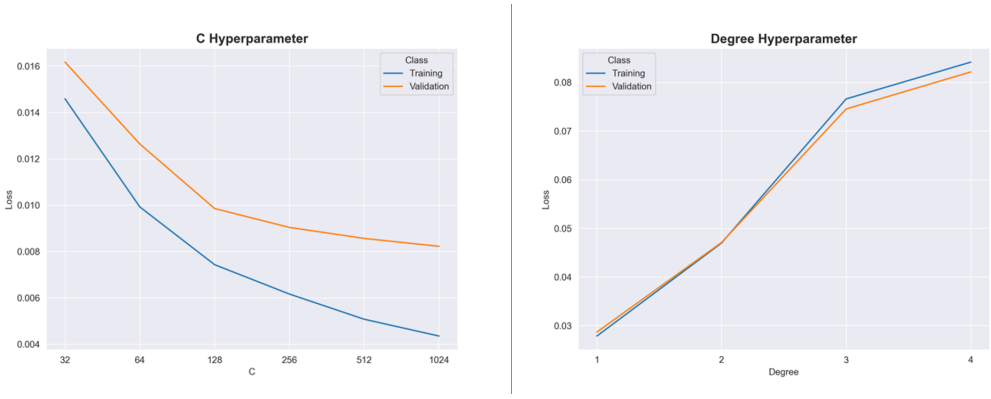


Figure 5.7: SVM Hyperparameter Tuning

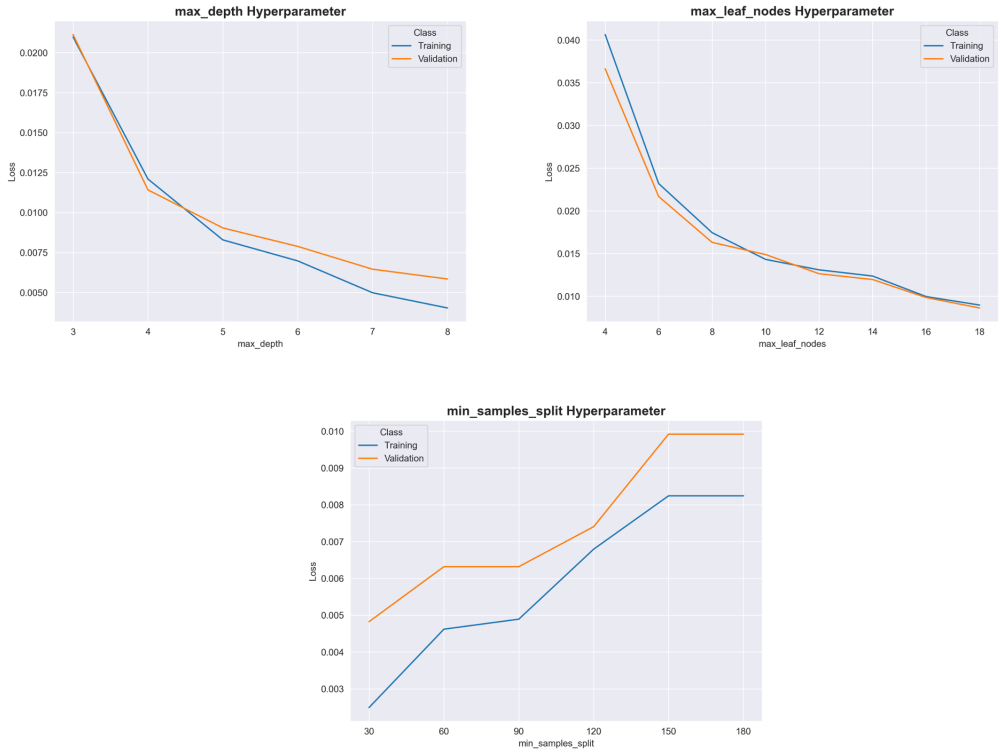


Figure 5.8: DT Hyperparameter Tuning

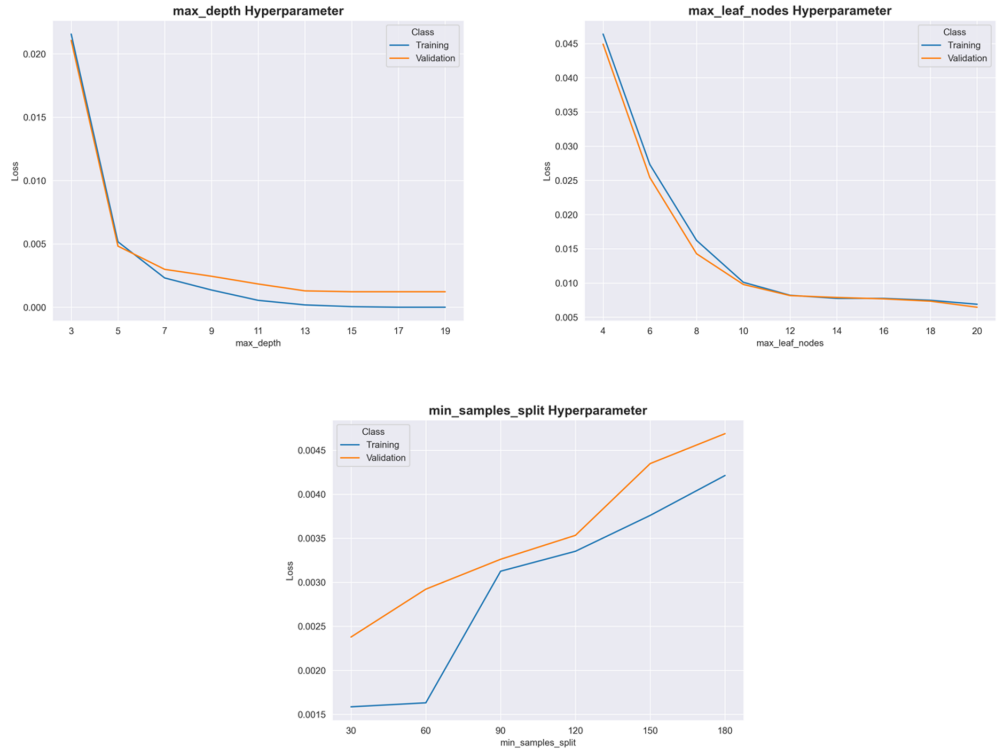


Figure 5.9: RF Hyperparameter Tuning

The optimal ranges are 4.3.2:

- ***SVM***: *C* 64 - 128. *degree*: 1 - 2
- ***DT***: *max_depth* - 4 - 6. *max_leaf_nodes* 6-10. *min_sample_split* - 60-90
- ***RF***: *max_depth* - 4 - 6. *max_leaf_nodes* 6-10. *min_sample_split* - 60-90

After tuning the hyperparameters, the models were fitted with :

Table 5.10: Parameters and Hyperparameters

Model	Parameters	Hyperparameters
SVM	kernel = "poly"	degree=1 C=128
DT	criterion="gini" max_features = $\sqrt{f}$	max_depth = 6 max_leaf_nodes = 10 min_samples_split = 90
RF	criterion="gini" max_features = $\sqrt{f}$ bootstrap = True	n_estimators = 75 max_depth = 5 max_leaf_nodes = 10 min_samples_split = 80

5.1.3.2 Cross Validation

Table 5.11: SVM

Fold No.	Precision	Sensitivity	Specificity	Confusion Matrix		
1	97.20	85.75	97.93		Benign	Malicious
				Benign	3922	83
				Malicious	478	2877
2	96.86	84.59	97.70		Benign	Malicious
				Benign	3913	92
				Malicious	517	2838
3	97.21	87.33	97.90		Benign	Malicious
				Benign	3921	84
				Malicious	425	2929
4	97.72	85.48	98.33		Benign	Malicious
				Benign	3937	67
				Malicious	487	2868
5	96.93	86.59	97.70		Benign	Malicious
				Benign	3912	92
				Malicious	450	2905
Average	97.18	85.95	97.91			

Table 5.12: DT

Fold No.	Precision	Sensitivity	Specificity	Confusion Matrix		
1	96.79	97.02	97.30		Benign	Malicious
				Benign	3897	108
				Malicious	100	3255
2	98.38	94.19	97.70		Benign	Malicious
				Benign	3953	52
				Malicious	195	3160
3	98.26	94.13	98.60		Benign	Malicious
				Benign	3949	56
				Malicious	197	3157
4	97.56	95.23	98.00		Benign	Malicious
				Benign	3294	80
				Malicious	160	3195
5	97.53	95.23	97.98		Benign	Malicious
				Benign	3923	81
				Malicious	150	3205
Average	97.70	95.22	98.12			

Table 5.13: RF

Fold No.	Precision	Sensitivity	Specificity	Confusion Matrix		
1	98.99	99.17	99.15		Benign	Malicious
				Benign	3971	34
				Malicious	28	3327
2	98.84	98.87	99.03		Benign	Malicious
				Benign	3966	39
				Malicious	38	3317
3	99.28	99.11	99.40		Benign	Malicious
				Benign	3981	24
				Malicious	30	3324
4	99.43	98.90	99.53		Benign	Malicious
				Benign	3985	19
				Malicious	37	3318
5	99.08	99.08	99.23		Benign	Malicious
				Benign	3973	31
				Malicious	31	3324
Average	99.12	99.02	99.27			

In this experiment, ***RF*** yet again displayed stellar performance with the highest average results. Precision, Sensitivity and Specificity scores were 1.42%, 3.8% and 1.15% higher than ***DT***,and 1.94%, 13.07% and 2.06% better than ***SVM*** in the same metrics. ***SVM*** confusion matrices showed a low number of false positives, but an unacceptable number of false negatives. This would pose a serious security risk, as it would allow a great number of malicious JavaScript to go undetected.

Significance of new Features

It is worth noting that in this experiment the $1-gram$ features were added to the feature set.***SVM*** yielded a much better performance than the first experiment with an average improvement of 39.28%, yet slightly worse than the second experiment, with an average of 1.88% decrease in performance in all metrics.***DT*** showed an average improvement of 6.28% and 1.4% for every metric, compared to the first and second experiments, respectively. ***RF*** performed better than the first experiment with an average improvement of 1.45% , and slightly better than the second experiment, with an average increase of 0.25%, across all metrics.

5.2 Discussion

The purpose of this project was to investigate the effectiveness of combining *Machine Learning* models with features extracted from the *AST* for the task of detecting malicious obfuscated JavaScript. Moreover, the author hypothesized that benign and malicious JavaScript are structurally different.

RF outperformed the other two models across all metrics. This model consistently captured the inherent structure of the data,regardless of the feature set used. ***DT*** saw gradual improvements with every feature set, and considering this model was a single tree, the results were surprisingly close to ***RF***. ***SVM*** performance was lacklustre and performed worse than initially thought. In the first experiment, this model was equivalent to a no-skill (random) model, unable to discriminate between the two classes.While the results greatly improved in the next experiments, its ability to correctly classify malicious inputs (*Sensitivity*) remained much lower than any other model.

At first, these scores may seem the result of *underfitting* or *overfitting*. However,the visualization of each hyperparameter guided the author during the hyperparameter optimization. The measurement criteria addressed the imbalanced class concern. Hence, the cross-validation results were a real representation of how the models would perform on unseen data.

Given the results, it is my understanding that tree-based models are better at predicting malicious JavaScript. I believe this is due to the nature of these algorithms. ***SVM*** divides the input space in a way that the data falls on either one side or the other of the hyperplane, making it very sensitive to outliers. On the other hand, ***DT*** recursively divides the input space until each path in the tree leads to a conclusion, thus finding the best representation of the problem at hand. ***RF*** is a collection of ***DTs*** whose final result originates from its majority vote mechanism, therefore making it more accurate and reliable than a single ***DT*** and consequently ***SVM***. Additionally, the results obtained by this project are in line with the results found in the literature.

Table 5.14: Performance - Literature

Author/Tool	Models	Precision	Sensitivity	Specificity
Likarish	NB DT SVM	80.00 - 92.00	65.09 - 78.07	99.7
Cujo	SVM	99.99	90.2	N/A
Zoozle	NB	95.5 - 99.9	89.00 - 95.00	N/A
Jast	RF	N/A	99.46	99.48

Table 5.15: Performance - Average results of the experiments in this work

Model	Precision	Sensitivity	Specificity
SVM	84.60	78.09	80.94
DT	94.49	93.92	94.30
RF	98.66	98.34	98.69

This project concludes that *Machine Learning* models combined with *AST*-based features is a robust method for detecting malicious obfuscated JavaScript at a scale, proving that the research question assumption of benign and malicious payloads having different syntactic structures and structural patterns is well-founded. All the above contributes to ***RF*** being the preferred *Machine Learning* model to perform this task.

5.3 Limitations

All machine learning-based approaches to the detection of malware are susceptible to misclassification. A classifier relies on the features available in the training set to learn the patterns on the data. Malicious actors may tailor new variants to avoid detection. Thus, there is a always a need to retrain the classifier and develop new features. With regards to this project, the biggest limitation was the unavailability of a malicious data. A model is only as good as the data it is trained on, consequently this lack of malicious data meant different attack patterns were not taken into account.

5.4 Feature Work

A malware detector cannot rely solely on lexical and syntactic analysis as these lack semantics. A robust approach should understand beyond the programming constructs of a file. Therefore, this project could be improved by adding a dynamic component to understand the flow of execution of the program. Another interesting feature to add would be to detect the type of attack used by the payload. Also, gathering more data is be essential.

Regarding any changes in the project, Acorn.js. was used as the JavaScript parser instead of Esprima. Acorn.js is faster and supports the latest ECMAScript.

Chapter 6

Conclusion

This project successfully explored the current applications of combining *Machine Learning* models with feature extraction from the *AST* for the task of detecting malicious JavaScript. To achieve this, the author proposed a fully static classification pipeline which included data collection, feature extraction, feature reduction, model implementation, hyperparameter tuning.

To conduct the experiments, three *Machine Learning* models were implemented, namely ***SVM***, ***DT*** and ***RF***. The metrics used to quantify the experiments were *Precision*, *Sensitivity* and *Specificity*. The results demonstrated that *Machine Learning* algorithms combined with AST-based features is an effective method for detecting malicious JavaScript instances at a scale and also concluded that the reason behind the ***RF*** model excelling at this problem lies in the underlying characteristics of the algorithm.

Bibliography

[1] Acorn package. Accessed: 05/09/2021. URL: https://www.npmjs.com/package/acorn.

[2] Alexa top 50 websites per country. Accessed: 11/11/2021. URL: https://www.alexa.com/topsites/countries.

[3] Alexa top websites. Accessed: 19/10/2021. URL: https://www.alexa.com/topsites/.

[4] Alexa's top 1 million websites. Accessed: 20/01/2022. URL: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[5] Awesome javascript. Accessed: 26/01/2022. URL: https://github.com/sorrycc/awesome-javascript.

[6] Awesome js games. Accessed: 26/01/2022. URL: https://github.com/proyecto26/awesome-jsgames.

[7] C-support vector classification. Accessed: 25/02/2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[8] Confusion matrix. Accessed: 25/02/2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html.

[9] Control flow flatenning. Accessed: 12/12/2021. URL: https://docs.jscrambler.com/code-integrity/documentation/transformations/control-flow-flattening.

[10] Dead code injection. Accessed: 12/12/2021. URL: https://docs.jscrambler.com/code-integrity/documentation/transformations/dead-code-injection.

[11] Decision trees. Accessed: 22/02/2022. URL: https://scikit-learn.org/stable/modules/tree.html#classification.

[12] Decisiontreeclassifier. Accessed: 21/02/2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html.

[13] Document object model (dom). Accessed: 20/10/2021. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

[14] Estraverse homepage. Accessed: 05/09/2021. URL: https://github.com/estools/estraverse.

[15] Gridsearch. Accessed on 20/10/2021. URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html?highlight=gridsearch#sklearn.model_selection.GridSearchCV.

[16] Html ¡script¿ src attribute. Accessed: 20/01/2022. URL: https://www.w3schools.com/tags/att_script_src.asp.

[17] Javascript malware collection on github. Accessed: 10/09/2021. URL: https://github.com/HynekPetrak/javascript-malware-collection.

[18] Javascript origins. Accessed: 20/10/2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.

[19] Json encoder and decoder. Accessed: 20/01/2022. URL: https://docs.python.org/3/library/json.html.

[20] Jupyter notebook homepage. Accessed: 02/05/2021. URL: https://jupyter.org/.

[21] Machine learning glossary. Accessed on 15/01/2022. URL: https://developers.google.com/machine-learning/glossary#confusion_matrix.

[22] Matplotlib homepage. Accessed: 16/10/2021. URL: https://matplotlib.org/.

[23] Mdn webdocks homepage. Accessed on 20/10/2021. URL: https://developer.mozilla.org/en-US/.

[24] Mdn webdocks homepage - eval. Accessed on 20/10/2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval.

[25] Node package manager homepage. Accessed: 05/09/2021. URL: https://www.npmjs.com/.

[26] Nodejs homepage. Accessed: 06/09/2021. URL: https://nodejs.org/en/.

[27] Numpy homepage. Accessed: 16/09/2021. URL: https://numpy.org/.

[28] Pandas homepage. Accessed: 16/09/2021. URL: https://pandas.pydata.org/.

[29] Parse urls into components. Accessed: 20/01/2022. URL: https://docs.python.org/3/library/urllib.parse.html.

[30] Python homepage. Accessed: 08/09/2021. URL: https://www.python.org/.

[31] Randomforestclassifier. Accessed: 28/03/2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[32] Scikit-learn homepage. Accessed: 20/10/2021. URL: https://scikit-learn.org/stable/.

[33] Scrapy homepage. Accessed: 10/10/2021. URL: https://scrapy.org/.

[34] Scrapy homepage. Accessed: 12/10/2021. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html#sklearn.metrics.average_precision_score.

[35] Scrapy homepage - requests and responses. Accessed: 10/10/2021. URL: https://docs.scrapy.org/en/latest/topics/request-response.html.

[36] Seaborn homepage. Accessed: 16/10/2021. URL: https://seaborn.pydata.org/.

[37] Selectors level 3. Accessed: 20/10/2021. URL: https://www.w3.org/TR/selectors-3/#selectors.

[38] sklearn.utils.shuffle homepage. Accessed: 03/03/2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.utils.shuffle.html.

[39] Stratifiedkfold. Accessed on 20/10/2021. URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html.

[40] train test split. Accessed on 20/10/2021. URL: https://scikit-learn.org/stable/modules/generated/sklearn.model{_}selection.train{_}test{_}split.html.

[41] W3techs - usage statistics of javascript as client-side programming language on websites. Accessed on 20/01/2022. URL: https://w3techs.com/technologies/details/cp-javascript.

[42] What is javascript. Accessed on 02/01/2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.

[43] Working with json. Accessed: 15/10/2021. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON.

[44] Static analysis of source code security: Assessment of tools against samate tests. *Information and Software Technology*, 55(8):1462–1476, 2013. URL: https://www.sciencedirect.com/science/article/pii/S0950584913000384, doi:https://doi.org/10.1016/j.infsof.2013.02.005.

[45] "alert: Further ransomware attacks on the uk education sector by cyber criminals". vol. 189, Jun 2021. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5461132/.

[46] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[47] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996. doi:10.1023/A:1018054314350.

[48] Rory. Carroll. "ransomware attack disrupts irish health services". 2021. URL: https://www.theguardian.com/world/2021/may/14/ransomware-attack-disrupts-irish-health-services/.

[49] Francois Chollet. *Deep Learning with Python*. Manning Publications Co., USA, 1st edition, 2018.

[50] Shawe Taylor J Cristianini, N. *Support Vector Machines: An Introduction to Support Vector Machines and Other Kernel based Learning Methods (ch. 6)*. Cambridge, Cambridge University Press, 1 edition, 2000.

[51] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Fast and precise in-browser javascript malware detection. USA, 2011. USENIX Association.

[52] Aurore Fass, Robert P. Krawczyk, Michael Backes, and Ben Stock. Jast: Fully syntactic detection of malicious (obfuscated) javascript. In *DIMVA, pp 303–325*, 2018.

[53] Evelyn Fix and J. L. Hodges. Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review / Revue Internationale de Statistique*, 57(3):238–247, 1989. URL: http://www.jstor.org/stable/1403797.

[54] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Associates, Inc., USA, 7th edition, 2020.

[55] Robin Genuer and Jean-Michel Poggi. Random forests. In *Random Forests with R, ch(1 and 3)*. Springer, 2020.

[56] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 2nd edition, 2019.

[57] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., 2009.

[58] Haibo He and Yunqian Ma. *Imbalanced Learning: Foundations, Algorithms, and Applications*. Wiley-IEEE Press, 1st edition, 2013.

[59] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976. URL: https://www.sciencedirect.com/science/article/pii/0020019076900958, doi:https://doi.org/10.1016/0020-0190(76)90095-8.

[60] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013.

[61] Scott F. Kaplan, Benjamin Livshits, Benjamin G. Zorn, Christian Siefert, and Charlie Cursinger. "nofus: Automatically detecting" + string.fromcharcode(32) + "obfuscated ".tolowercase() + "javascript code". 2011.

[62] Simon Kemp. Digital 2020 global overview report. Accessed: 19/10/2021. URL: https://datareportal.com/reports/digital-2020-global-digital-overview.

[63] Simon Kemp. Digital 2022 october global statshot report. Accessed: 20/03/2022. URL: https://datareportal.com/reports/digital-2022-global-overview-report.

[64] Sotiris B. Kotsiantis. Decision trees: a recent overview. *Artificial Intelligence Review*, 39:261–283, 2011.

[65] Max Kuhn and Kjell Johnson. *Applied predictive modeling, (ch. 13)*. Springer, 1st edition, 2013.

[66] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 47–54, 2009. doi:10.1109/MALWARE.2009.5403020.

[67] Kalyeena Makortoff. World's biggest meat producer jbs pays $11m cybercrime ransom. *The Guardian*, Jun 2021. URL: https://www.theguardian.com/business/2021/jun/10/worlds-biggest-meat-producer-jbs-pays-11m-cybercrime-ransom.

[68] Kari Paul and Lois Beckett. What we know – and still don't – about the worst-ever us government cyber-attack. *The Guardian*, Dec 2021. URL: https://www.theguardian.com/technology/2020/dec/18/orion-hack-solarwinds-explainer-us-government.

[69] Associated Press. Colonial pipeline confirms it paid $4.4m ransom to hacker gang after attack. *The Guardian*, May 2021. URL: https://www.theguardian.com/technology/2021/may/19/colonial-pipeline-cyber-attack-ransom.

[70] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do. In *European Conference on Object-Oriented Programming*, pages 52–78. Springer, 2011.

[71] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, page 31–39, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1920261.1920267.

[72] Lior Rokach and Oded Maimon. *Data Mining With Decision Trees: Theory and Applications*. World Scientific Publishing Co., Inc., USA, 2nd, chapter 1 edition, 2014.

[73] Z.M. Smith, E. Lostri, and McAfee (Firm). *The Hidden Costs of Cybercrime*. McAfee, 2020. URL: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-hidden-costs-of-cybercrime.pdf.

[74] Leo Breiman Statistics and Leo Breiman. Random forests. In *Machine Learning*, pages 5–32, 2001.

[75] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory, (ch. 5)*. Springer-Verlag New York, Inc., 2nd edition, 2000.

[76] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, 2012.

Chapter 7

Appendix

7.1 Algorithms

The original baseline model developed was the K-Nearest Neighbors.

The K-nearest neighbors [53] algorithm computes the distance between the training data and the test data, to find its K nearest neighbors. Given new unlabeled data, the algorithm will select the class it belongs to based on its K nearest neighbors. By default, the KNN algorithm uses the Euclidean distance, which can be calculated by the following equation:

$$d\left(p,q\right)=\sqrt{\sum_{i=1}^{n}\left(q_i-p_i\right)^2}$$

7.2 Implementation - Gathering Benign data

Initially, the author developed two web crawlers, both of which with distinct purposes. The sole purpose of the first crawler was to scrape the URLs from Alexa's top-ranked websites [3], extract the domains from the URLs, and ultimately prepare this data into a (URLs, Domains) format.

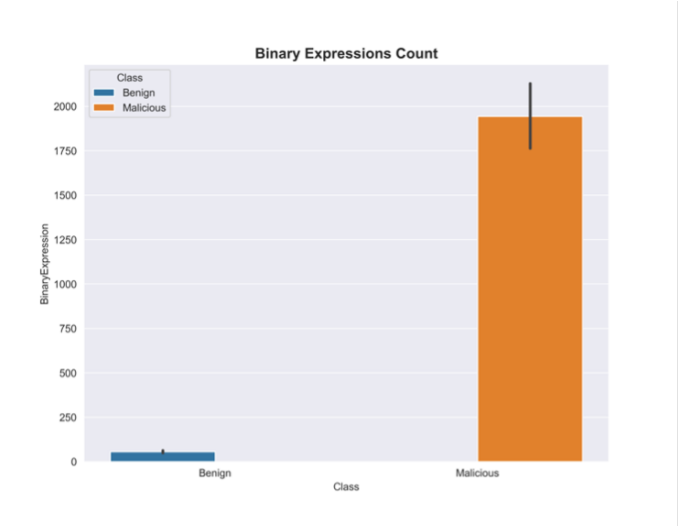
The second crawler was designed to extract JavaScript code *embedded* within HTML documents.Using the data obtained previously, this crawler conducted a crawl on Alexa's top-ranked websites, acquiring over 7600 scripts. The major drawback of this approach was the amount of duplicate code across most websites. Therefore, at that stage, it was only possible to gather 1142 unique samples of benign JavaScript.

After that, using the same methods described above, the author conducted a crawl on Alexa's top 50 websites by country. [2]. This crawl gathered over 10,000 scripts. The problem was that every country shared the same most popular websites in the world (e.g. Google, Facebook, YouTube, etc). This led to the same duplicate code issue previously reported. After removing all duplicates, there were only 3679 unique samples of benign JavaScript. Thus, given the issues encountered during this period, the methods used were deemed inappropriate.

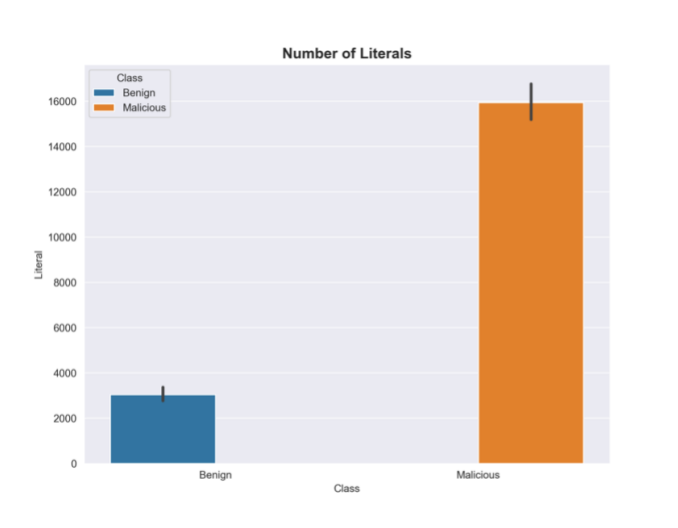
7.3 EDA

In the early stages of the project, the EDA (Exploratory Data Analysis) helped in establishing a relationship between the features, thus enabling the author to make informed decisions during the feature selection process.

Data Obfuscation Pattern: Malicious payloads uses excessive number of string concatenations. This includes string concatenation ("+" operator), arithmetic operations[1] such as addition, subtraction, multiplication, and division to extract an element of an object at a specific index.



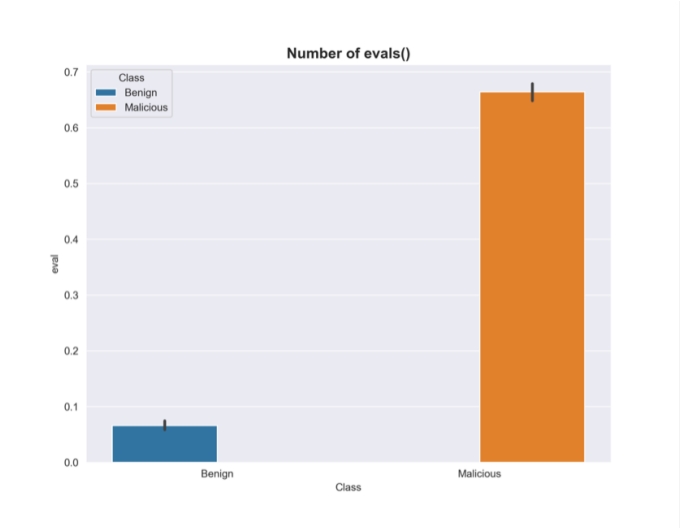
Syntactic Units: Malicious scripts rely heavily on the creation of Literals to hide their intent. JavaScript literals are constant values that can be assigned to variables.



Data: Eval [24] is one of the most infamous JavaScript built-in functions and presents a serious security risk. This function expects a string as its only argument. Malicious actors

[1] "+", "-", "*", "/" operators, respectively)

exploit the dynamic nature of JavaScript to generate data obfuscated code on the fly, and to evaluate such code at runtime using the eval function.



7.4 Datasets

7.4.1 Experiment 1

Description of the features.The number of occurrences of the following randomization and data obfuscation patterns :

1. Length in Characters : The length of a script in characters.
2. % whitespace: The percentage of the script that is whitespace.
3. Strings : The number of strings in a script.
4. Avg. string length : The average number of characters per string in a script.
5. (Number) of concatenations = The number of concatenations in a file
6. minified: check if a file is packed
7. Number of dead functions : dead code added for obfuscation
8. function returns a string

7.4.2 Experiment 2

Description of the features.The number of occurrences of the following syntactic units,data and encoding obfuscation patterns :

1. AssignmentExpression, BinaryExpression, BlockStatement, CallExpression, ControlFlow, ExpressionStatement, Functions, Identifier, Literal, LogicalExpression, MemberExpression, ReturnStatement, UnaryExpression, VariableDeclaration, VariableDeclarator, ConditionalExpression, ObjectExpressions Property, SequenceExpression, EmptyStatement, Exceptions Loops, NewExpression, ObjectPattern, TemplateElement,

TemplateLiteral, ThisExpression, UpdateExpression, ArrayPattern, BreakStatement, SwitchCase, ContinueStatement AssignmentPattern, LabeledStatement, Class, Imports ClassBody, ClassDeclaration, MethodDefinition,Super,AwaitExpression,SpreadElement, RestElement,Exports,ClassExpression, ChainExpression,TaggedTemplateExpression, ExportAllDeclaration,DebuggerStatement, YieldExpression

2. array_expr_prim_elem : the number of arrays passed as arguments
3. call_expr_str_arg: the number of function calls passed as arguments
4. obj_expr_prop_val : the number of objects passed as arguments
5. dom_operations: the number of malicious doom operations
6. eval: the number of eval calls
7. var_with_str_assign: the number of variables with string assignment
8. str_Constructor, str_charAt, str_charCodeAt, str_concat, str_fromCharCode, str_replace, str_slice,str_split, str_substring: string methods

7.4.3 Experiment 3

Description of the features.The number of occurrences of the following syntactic units,data and encoding obfuscation patterns, 1-gram :

1. ControlFlow, Exceptions, Expressions, Functions, Literal, LogicalExpression, LoopControlFlow, Loops, ObjectExpressions, Pattern, Property, Return, Statements, ThisExpression, VariableDeclarator
2. avg_str_length, concatenation, stringCount, file_length
3. arrayManipulation, dom_operations, encodingObfuscation, funcArgsCallExpression, funcArgsPrimitiveVal, funcReturnString, funcSelfInvoking , eval
4. varConcatenation, varWithArrayExpression, varWithCallExpression, varWithLiteralString, varWithNewExpression, varWithObjExpression, varWithThisExpression

7.4.4 Important links

Raw data:
`https://drive.google.com/drive/folders/1z_v8JMPTjWYaRfCUQ5ODncGoj9aRfQU_?usp=sharing`