FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automatic Identification of Obfuscated JavaScript using Machine Learning

Susana Maria de Sousa Lima

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. Ricardo Morla

Second Supervisor: João Routar

July 23, 2021

Automatic Identification of Obfuscated JavaScript using Machine Learning

Susana Maria de Sousa Lima

Approved in oral examination by the committee:

Chair: Prof. José Magalhães Cruz

External Examiner: Prof. Marco Vieira

Supervisor: Prof. Ricardo Morla

July 23, 2021

Abstract

JavaScript is widely used as a client-side language mainly due to its dynamic and flexible nature. However, these characteristics, combined with its popularity, make it a common vector for malicious attacks, which are often concealed by the use of obfuscation.

In this work, we propose a solution to detect obfuscated JavaScript and identify the obfuscator used in the code, based on machine learning algorithms and static code analysis. We start by comparing two different approaches to understand if using contextual information benefits the detection of obfuscated code. The first approach is a Multinomial Naive Bayes classifier with features computed from paths extracted from the code's Abstract Syntax Tree, which retain contextual information of specific nodes. The second approach is a Random Forest classifier with features defined based on standard obfuscation practices, with no context associated with them. To train and test our models, we use a collection of 100k regular (and minified) files and 170k obfuscated files, transformed with various obfuscators.

Our results show that the model that uses features without contextual information misclassifies significantly fewer files than the one that uses this type of information. By using features based on standard obfuscation practices, we can successfully detect obfuscation and identify the obfuscator used in the code, with an F1-score of 99.99% in both these tasks. We conduct additional experiments to validate assumptions made while creating the dataset and evaluate the feature-based model in various scenarios. The results obtained reinforce the importance of prioritizing the diversity of the dataset over its size when implementing an obfuscation detector. We then take a look at how our solution generalizes to code transformed with new obfuscators. We obtain mixed results, concluding that it can generalize to code transformed with some tools but not with others. Finally, we validate our solution in the context of partial obfuscation, obtaining positive results when assuming that the obfuscated code appears at the beginning of the program to mimic a malicious attack.

Keywords: Obfuscated JavaScript, Malware Mitigation, Machine Learning, Classification Algorithms

Resumo

JavaScript é uma linguagem amplamente usada no desenvolvimento de código *client-side* principalmente devido à sua natureza dinâmica e flexível. No entanto, estas características, combinadas com a popularidade da linguagem, tornam-na um vetor comum para ataques maliciosos, que são frequentemente ocultados pelo uso de ofuscação.

Neste trabalho, propomos uma solução para detectar JavaScript ofuscado e reconhecer o ofuscador utilizado para transformar o código, com base em algoritmos de *machine learning* e análise estática de código. Começamos por comparar duas abordagens distintas de modo a perceber se o uso de informação contextual beneficia a detecção de código ofuscado. A primeira abordagem é um classificador *Multinomial Naive Bayes* baseado em *features* computadas através de caminhos extraídos da Árvore de Sintaxe Abstrata do código, que retêm informaçal contextual de nós específicos da árvore. A segunda abordagem é um classificador *Random Forest* que recebe como *input* um conjunto de features baseadas em práticas comuns de ofuscação, não retendo qualquer tipo de informação contextual. Para treinar e testar os modelos, usamos uma coleção de cerca de 100 mil ficheiros regulares (e minificados) e 173 mil ficheiros ofuscados, transformados com várias ferramentas.

Os resultados obtidos mostram que o modelo que usa *features* sem informação contextual classifica incorretamente significativamente menos ficheiros que o modelo que usa este tipo de informação. Através do uso de *features* baseadas em práticas comuns de ofuscação, a nossa solução é capaz de detectar com sucesso ofuscação e identificar o ofuscador usado, com uma *F1-score* de 99,99% em ambas as tarefas. Adicionalmente, elaboramos um conjunto de experiências para validar o *dataset* criado e avaliar o modelo baseado em *features* em diferentes cenários. Os resultados obtidos reforçam a importância de priorizar a diversidade do *dataset* em relação ao seu tamanho, ao implementar um detetor de ofuscação. De seguida validamos a nossa solução na presença de código ofuscado por ferramentas desconhecidas, de modo a perceber se a solução é capaz de generalizar neste tipo de situação. Com esta experiência obtemos resultados mistos, concluindo que a solução é capaz de generalizar para alguns ofuscadores mas não para outros. Finalmente, validamos a solução na presença de código parcialmente ofuscado, obtendo resultados positivos assumindo que o código ofuscado se encontra no início do programa, de modo a replicar um ataque malicioso.

Keywords: JavaScript Ofuscado, Mitigação de Malware, *Machine Learning*, Algoritmos de Classificação

Acknowledgements

First of all, I would like to thank my supervisors, Professor Ricardo Morla and João Routar, for all the guidance and support provided during this dissertation, without whom this would not have been possible. Thank you, Professor Ricardo Morla, for all your fun and enthusiastic ideas that led to interesting and exciting debates, always balancing work and fun. Thank you, João Routar, for all your honest and fair feedback, for being the voice of reason when necessary (you had fun ideas, too!), and for making me laugh at my awkward moments. I hope I have the opportunity to work with you two again!

I also thank everyone at *Jscrambler* for all the kindness shown during this process and for allowing me to use your resources to make the best possible work. See you all soon!

To all my friends, thank you for the unconditional love, support, and patience. In particular to the ones I met (or work with) at FEUP, thank you for accompanying me in this amazing journey, for all the good and not so good moments we spent together, for listening to my endless stories and nonsense rants about random stuff, and finally, for helping me become a better person. I will miss getting mad when you do not do your part of the projects on time, but I sincerely hope you all stop doing that.

Thank you to all my family, including those not here anymore, but especially my parents and siblings, for all the love, guidance, and shaping me into the person I am today. To my parents, thank you for all the opportunities you made possible for me. To my siblings, thank you for always supporting my decisions and believing in me. Talk to you at home!

Finally, a big thank you to Professor Raúl Vidal, who surely does not remember me, for his wisdom, encouragement, and for seeing my potential in this area when I was not so sure about it.

Susana Lima

“All things truly wicked start from innocence.”

Ernest Hemingway

Contents

1 Introduction **1**
1.1 Context and Motivation . . . 1
1.2 Objectives . . . 2
1.3 Document Overview . . . 3

2 JavaScript Obfuscation **5**
2.1 JavaScript . . . 5
2.1.1 Dynamic Code Generation . . . 6
2.1.2 Challenges and Vulnerabilities . . . 7
2.2 Code Transformations . . . 7
2.2.1 Techniques . . . 8
2.2.2 Tools . . . 13
2.3 Summary . . . 14

3 Obfuscated Code Detection **15**
3.1 Background . . . 15
3.1.1 Code Analysis . . . 15
3.1.2 Abstract Syntax Tree . . . 16
3.1.3 Contextual Information . . . 16
3.1.4 Machine Learning . . . 16
3.2 Previous Work . . . 19
3.2.1 Malicious Obfuscated Code . . . 19
3.2.2 Obfuscated Code . . . 21
3.2.3 Multiple Obfuscators . . . 23
3.3 Challenges . . . 26
3.3.1 Diversity of Techniques and Tools . . . 26
3.3.2 Unknown Obfuscators . . . 26
3.3.3 Minified Code . . . 26
3.3.4 Partial Obfuscation . . . 27
3.4 Summary . . . 27

4 Detector Implementation **31**
4.1 Context . . . 31
4.2 Dataset . . . 31
4.2.1 Requirements . . . 34
4.2.2 Sources of JavaScript Code . . . 34
4.2.3 Preprocessing . . . 35
4.2.4 Transformation . . . 38

4.3 Parser . . . 41
4.3.1 Path-contexts . . . 42
4.3.2 Features . . . 43
4.4 Classifiers . . . 50
4.5 Summary . . . 51

5 Context Versus No Context 53
5.1 Experimental Setup . . . 53
5.2 How to select the paths? . . . 55
5.3 Is there any benefit in using context-based features? . . . 57
5.4 Summary . . . 63

6 Additional Questions 65
6.1 Experimental Setup . . . 65
6.2 What is the impact of training the model with fewer files? . . . 66
6.3 What is the impact of training the model with smaller files? . . . 68
6.4 Is the model biased to the sources of code used in training? . . . 70
6.5 Is the model able to detect obfuscated code transformed by unknown tools? . . . 74
6.6 What is the impact of training the model without minified code? . . . 77
6.7 Is the model able to detect obfuscation in partially obfuscated code? . . . 78
6.7.1 Classifying the whole file . . . 78
6.7.2 Classifying only a portion of the file . . . 80
6.8 Summary . . . 83

7 Conclusions 85

A Dataset Preparation 89
A.1 Data Collection . . . 89
A.2 Data Transformation . . . 90
A.2.1 *Jscrambler* . . . 91
A.2.2 *javascript-obfuscator* . . . 96
A.2.3 *defendjs* . . . 100
A.2.4 *js-obfuscator* . . . 101
A.2.5 *JSObfu* . . . 102
A.2.6 *JavaScript2img* . . . 103
A.2.7 *DaftLogic* . . . 104
A.2.8 *jsfuck* . . . 105
A.2.9 *node-obf* . . . 106
A.3 Data Parsing . . . 108

B Experiments 111

References 113

List of Figures

2.1 Example of a *Drive by Download* attack (source [49]). 6
2.2 Example of randomization obfuscation (source [70]). 10
2.3 Example of data obfuscation applied to numbers. 10
2.4 Example of data obfuscation applied to strings (source [41]). 11
2.5 Example of data obfuscation by keyword substitution. 11
2.6 Example of encoding obfuscation with customized functions (source [70]). . . . 12
2.7 Example of logic structure obfuscation applied to predicates (source [42]). 12
2.8 Example of logical structure obfuscation applied to loops, namely loop fissing (source: [62]). 13

3.1 Methodology for the automatic detection of obfuscated JavaScript strings in malicious web pages (source: [32]). 20
3.2 Example of the features used in Kaplan et al. [48] (source: [33]). 22
3.3 Example of JSOD's features (source: [28]). 23

4.1 Dataset creation pipeline. 33
4.2 Example of scripts found in HTML code. 35
4.3 Number of files kept and filtered per data source. 36
4.4 Number of files filtered per motive. 36
4.5 Example of code transformed by *JSObfu* with a size smaller than 30 bytes. 40
4.6 String related features' distribution in regular and obfuscated code. 45
4.7 Average frequency of encoding per terminal type, for regular and obfuscated code. 49
4.8 Distribution of the frequency of indentation characters in regular and obfuscated code. 49

6.1 Percentage of files, transformed with the obfuscator removed from the training set, correctly classified per obfuscator removed. 76
6.2 Percentage of misclassified partially obfuscated files, per percentage of obfuscation in the code. 79
6.3 Percentage of misclassified partially obfuscated files, per percentage of statements selected. 81
6.4 Example of partially obfuscated code where the regular code is an IIFE and the obfuscated code is a *CallExpression*, and its corresponding AST. The example only contains the essential part of the program - the comments are placeholders for the rest of the code - and of the AST. 82

A.1 Examples of the requests' endpoints issued to *GitHub*'s REST API. These requests retrieve 100 repositories from the first results page, but can be altered to retrieve from other pages as well. 89

A.2 Regular JavaScript code. 90
A.3 *Jscrambler-config1*. First configuration used for the obfuscator *Jscrambler*. . . . 91
A.4 *Jscrambler-2*. Second configuration used for the obfuscator *Jscrambler*. 92
A.5 *Jscrambler-3*. Third configuration used for the obfuscator *Jscrambler*. 93
A.6 *Jscrambler-4*. Fourth configuration used for the obfuscator *Jscrambler*. 94
A.7 Code in Figure A.2 obfuscated with *Jscrambler-config1*. 95
A.8 *javascript-obfuscator-config1*. First configuration used for the obfuscator *javascript-obfuscator*. 96
A.9 *javascript-obfuscator-2*. Second configuration used for the obfuscator *javascript-obfuscator*. 97
A.10 *javascript-obfuscator-3*. Third configuration used for the obfuscator *javascript-obfuscator*. 98
A.11 *javascript-obfuscator-4*. Fourth configuration used for the obfuscator *javascript-obfuscator*. 99
A.12 Code in Figure A.2 obfuscated with *javascript-obfuscator-config1*. 100
A.13 *defendjs-config1*. First configuration used for the obfuscator *defendjs*. 100
A.14 *defendjs-2*. Second configuration used for the obfuscator *defendjs*. 100
A.15 Code in Figure A.2 obfuscated with *defendjs-config1*. 101
A.16 *js-obfuscator-config1*. First configuration used for the obfuscator *js-obfuscator*. . 101
A.17 *js-obfuscator-2*. Second configuration used for the obfuscator *js-obfuscator*. . . . 101
A.18 *js-obfuscator-3*. Third configuration used for the obfuscator *js-obfuscator*. 102
A.19 Code in Figure A.2 obfuscated with *js-obfuscator-config1*. 102
A.20 Code in Figure A.2 obfuscated with *JSObfu*(*-config1*). 102
A.21 Code in Figure A.2 obfuscated with *JavaScript2img*(*-config1*). To be able to showcase the entire program some strings were altered, therefore the code will not run correctly. 103
A.22 Code in Figure A.2 obfuscated with *DaftLogic*(*-config1*). 104
A.23 Code in Figure A.2 obfuscated with *jsfuck*(*-config1*). 105
A.24 Code in Figure A.2 obfuscated with *node-obf*(*-config1*). 106

B.1 Example of the partially obfuscated code generated (some comments were removed to be able to showcase the entire program). 111

List of Tables

2.1 Dependencies between obfuscation evaluation metrics (source [69]). 8
2.2 Evaluation of obfuscation techniques. 9
2.3 Obfuscation tools and obfuscation techniques matrix. 14

3.1 Previous work qualitative comparison. 28
3.2 Previous work quantitative comparison. 28

4.1 Distribution of files per type of code - Regular and tool used - and configuration used, after parsing the code. All obfuscators have a common configuration, *config1*, and some have other configurations which are tool-dependent, and are represented by the tool's name and a number. 42
4.2 Average value for the length, frequency of letters, frequency of uppercase letters, frequency of words, and frequency of numbers, per type of identifier, across all regular and obfuscated files. 46
4.3 Average frequency of specific JavaScript keywords in regular and obfuscated code (top 10). 47
4.4 Average frequency of specific nodes (and groups of nodes) in regular and obfuscated code. 47

5.1 Context types and their correspondent JavaScript AST nodes and assigned weights. 55
5.2 Results obtained by the binary model, by using random or selected paths with a variety of number of paths. 56
5.3 Results obtained by the multiclass model, by using random or selected paths with a variety of number of paths. 56
5.4 Cross-validation and testing results for the binary version of the *Code2FeatureVector* and *Code2BagOfPaths* models. 58
5.5 Confusion matrix for the binary version of the *Code2BagOfPaths* model, extended by tools and configurations. In bold are the total values per class. 59
5.6 Confusion matrix for the binary version of the *Code2FeatureVector* model, extended by tools and configurations. In bold are the total values per class. 60
5.7 Cross-validation and testing results for the multiclass version of the *Code2FeatureVector* and *Code2BagOfPaths* models. 60
5.8 Confusion matrix for the multiclass version of the *Code2BagOfPaths* model, extended by tools and configurations. In bold are the total values per class. 61
5.9 Confusion matrix for the multiclass version of the *Code2FeatureVector* model, extended by tools and configurations. In bold are the total values per class. 62

6.1 Results obtained for training with 5%, 15%, 25%, and 50% of training files. . . . 67

6.2 Number of files misclassified per code type, for each percentage of the training data used. 68
6.3 Number of files misclassified per code type, for each training set. 69
6.4 Baseline model's confusion matrix extended by source. In bold are the total values per class. 71
6.5 *Code2FV_wo_Web* model's confusion matrix extended by source. In bold are the total values per class. 72
6.6 *Code2FV_wo_GitHub* model's confusion matrix extended by source. In bold are the total values per class. 73
6.7 *Code2FV_wo_NPM* model's confusion matrix extended by source. In bold are the total values per class. 74
6.8 Confusion matrix extended to represent both regular and minified code and tools. In bold are the total values per class. 77
6.9 Confusion matrix for the partially obfuscated files, when classifying the whole file. No regular files are classified in this experiment. 79
6.10 Confusion matrix for the regular and partially obfuscated files, when classifying only a portion of the file. 80

A.1 Features extracted from the code. 108
A.2 Average frequency of specific JavaScript keywords in regular and obfuscated code. 109

B.1 Size distribution per tool/class. 112
B.2 Results obtained by using training sets with smaller files to train the model, comparing with the baseline model. 112
B.3 Results obtained by using fewer code sources to train the model, comparing with the baseline model. 112
B.4 Results obtained by removing minified code from training, comparing with the baseline model. 112

Abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
CNN	Convolutional Neural Network
DOM	Document Object Model
DTIA	Decision Tree Induction Algorithm
FEUP	Faculdade de Engenharia da Universidade do Porto
FN	False Negative
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
GB	GigaByte
GHz	GigaHertz
GPU	Graphics Processing Unit
HTML	HyperText Markup Language
IIFE	Immediately-Invoked Function Expression
JS	JavaScript
JSOD	JavaScript Obfuscation detector
JSON	JavaScript Object Notation
KB	KiloByte
NPM	Node Package Manager
OMA	Obfuscated Malicious Argument
RAM	Random Access Memory
REST	Representational State Transfer
RSIA	Rule Set Induction Algorithm
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
VCLFE	Variable Context-Level Feature Extractor
XSS	Cross-site Scripting

Chapter 1

Introduction

JavaScript is widely used as a client-side programming language, being supported by most web browsers due to its overall dynamism and flexibility. However, such characteristics potentiate new vulnerabilities that can be explored with malicious intents. A practical method to obscure the purpose of a JavaScript program is to apply transformations that maintain the code's original behavior but compromise its overall understandability. The deliberate act of modifying code to be hardly readable by humans and resilient to reverse engineering tools is known as obfuscation. Although it is used as a security measure for protecting intellectual property and preventing plagiarism, obfuscation is commonly employed to conceal malicious scripts to circumvent detection by manual inspections and pattern-based security systems.

While obfuscation relies on various established techniques, automatically recognizing if a JavaScript program is obfuscated is not a simplistic task. This is mainly due to the similarity between standard programming practices and common obfuscating techniques. Additionally, the variety of obfuscators and obfuscation techniques available can compromise the development of a generic solution. Successfully distinguishing between regular and obfuscated code requires a thorough analysis of the JavaScript language, along with commonly used obfuscation techniques and tools. In this work we consider as *regular* JavaScript, any code that is not obfuscated, as well as minified code.

1.1 Context and Motivation

This dissertation was proposed by *Jscrambler*, a high growth Tech Company that creates novel products for protecting the integrity of web and mobile applications [40]. It currently has the leading JavaScript protection product, the tool *Jscrambler* (that gives name to the company), with tens of thousands of clients worldwide.

The efficient and automatic classification of JavaScript code has a lot of interest and potential. It can be used to identify possible threats, namely malicious scripts that are obfuscated. Once

an obfuscated script is detected, taking into account that attackers use these techniques to conceal their code's behavior, security measures can be applied to verify the code's intents and, if required, mitigate the attack. Another use for this type of classification is related to the tools used. There are available various obfuscators with different complexities. A detector that identifies which one was used facilitates the deobfuscation process. This distinction can also be applied to further assess the code's intents since a small set of obfuscators is commonly used to hide malware. This means that identifying code transformed by a particular obfuscator is a stronger indicator that the code might be malware compared to code transformed with other obfuscators.

The detector can be applied in a variety of contexts, such as scenarios where JavaScript code is being distributed for widespread use (galleries of browser extensions, software hosting platforms, among others); to disable obfuscated scripts from running in a browser, in order to mitigate potential threats; and to assist in the automatic collection of JavaScript malware (by incorporating it into *honeyclients* or *honeypots*).

1.2 Objectives

The main objective of this dissertation is to develop a robust model that can effectively distinguish between obfuscated and non-obfuscated JavaScript code. The model should receive a snippet of JavaScript as input and automatically identify if it is obfuscated or not. It should also take into consideration various obfuscation tools and recognize which tool was used in the code.

Other objectives can be acknowledged as well. The first is the creation of a large and diverse dataset that incorporates JavaScript code from varied sources and code transformed with numerous obfuscators and obfuscation techniques. This dataset should undergo preprocessing to ensure its quality. Additionally, the dataset should be as balanced as possible, incorporating similar amounts of regular and obfuscated code, conversely to other projects in the area. The diversity of the dataset enables evaluating the solution in different conditions, validating or not its robustness in several scenarios. The underlying goal is to implement a model that is not biased to a particular type of code, obfuscator, or obfuscation technique. In addition, the created dataset can be adapted and used in other projects that require the use of a large number of JavaScript files.

Finally, another important goal is to identify and implement an efficient and practical approach to distinguish regular code from obfuscated code based on the static analysis of the program. To accomplish this, different approaches should be compared, against the same data, to assess and weigh their benefits and limitations. Based on previous works, we aim at understanding if the use of information that retains some level of its surrounding context improves the detection of obfuscation.

1.3 Document Overview

The subsequent chapters are structured as follows:

- **Chapter 2, JavaScript Obfuscation,** presents a brief background on JavaScript obfuscation, starting with an overview of the language followed by an introduction to code transformations. This chapter provides the required background to understand the context of this dissertation.
- **Chapter 3, Obfuscated Code Detection,** presents a preliminary literature review on the main topics addressed by this dissertation, mainly work done on obfuscated and malicious code detection, and concisely describes the challenges identified. It also introduces related concepts, such as code analysis and machine learning.
- **Chapter 4, Detector Implementation,** briefly explains the problem at hand and the proposed solution to address it. It describes in detail all the steps of the solution's implementation, namely the dataset creation, the parser used, and the classifiers implementation.
- **Chapter 5, Context Versus No Context,** tries to answer the main question of this dissertation, regarding the use of contextual information in obfuscation detection. This chapter describes a set of experiments, analysing their results.
- **Chapter 6, Additional Questions,** presents and tries to answer six additional questions, through different experiments and analysis of their results.
- **Chapter 7, Conclusions,** provides an overview of the main conclusions reached, briefly comparing them to the previous work on the area. In addition, it also presents a description of the limitations of the solution and future work that can be done in the area.

Chapter 2

JavaScript Obfuscation

In this chapter, we present additional background on JavaScript obfuscation, starting with an overview of the language, followed by an introduction to code transformations. This chapter includes a summary of the JavaScript language, its dynamic code generation abilities, vulnerabilities, and a description of different transformations, techniques, and tools that can be applied to the code.

2.1 JavaScript

JavaScript is widely used and has a very strong presence on the web, being supported by all major browsers. The vast majority of all web applications use it as a client-side programming language [1] due to its dynamic, event-driven, and asynchronous nature to enhance interactivity and functionality. There are numerous frameworks and libraries to build JavaScript applications, which have also contributed to the language's popularity.

JavaScript is a highly expressive language and much of the code that is executed can be loaded and generated at runtime since it does not need a browser to compile. Its code is also used to modify the DOM at runtime. Due to its event-driven nature, JavaScript allows the registration of various event listeners to the DOM, which can be triggered by the user, timing, or asynchronous actions [29].

This language was developed in 1995 by the Netscape developer Brendan Eich. It was intended to be simple to use by developers and non-developers, but sufficiently robust to implement real web applications. It employs a prototype/instance-based inheritance system, instead of implementing classes like other object-oriented languages do (such as Java). Almost all JavaScript objects inherit properties from a chain of prototype objects, and these properties define each object's behavior [68]. Since fields, methods, and contents of any prototype can be altered at any time, the behavior exhibited by an object can vary during the program's execution. JavaScript is

[1] https://w3techs.com/technologies/overview/client$_{s}ide_{l}anguage/$

also a dynamically typed language, meaning that at different moments of the program a variable can be used with distinct types [57].

2.1.1 Dynamic Code Generation

JavaScript offers simple ways to convert text into executable code, allowing code to be generated at runtime [56]. This is appealing to developers since it enables interactive development and gives them the ability to easily extend and customize their applications' behavior at any given time.

The *eval* function is used for this purpose. It receives a string as an argument and parses it as source code, which is then instantly executed, returning the invocation result. This function is used for various reasons, such as *JSON* parsing and asynchronous loading of libraries [56]. The code executed via *eval* has access to local and global variables [56] since it is executed with the same privileges as the caller function [72]. This represents a serious security risk, allowing attackers to inject malicious code into the argument string, which is then executed.

Code can also be generated at runtime by using *setInterval* and *setTimeout*. These functions receive a string or a function as an argument. When invoked with a string, *eval* is called to parse it into invocable code. The safest approach to use them is with a named or anonymous function to avoid the unnecessary call to *eval*. Other indirect means of dynamic generation such as the *document.write* method and the *innerHtml* property can be used by directly adding script nodes to the DOM [73].

Although these methods are not malicious and can be used with completely benign code, they are considered insecure practices, therefore their use should be avoided if possible and replaced by safer alternatives [73]. Figure 2.1 depicts a scenario of a *Drive By Download* [2] attack. The *document.write* method is used, with resource to the *iframe* tag, to run the malicious script and redirect the user. Additionally the code is obfuscated and the *eval* function is used to generate the code at runtime, in an attempt to conceal it from detectors.

```
document.write( <iframe src="http
    ://sedpoo.com/?338375" width=1
    height=1></iframe> );
```

(a) Original code.

```
eval(unescape( %64%6F%63%75%6D%65%6
    E%74%2E%77%72%69%......(some
    bytes skipped) .......3E%3C%2F
    %69%66%72%61%6D%65%3E%27%29 ));
```

(b) Obfuscated code.

Figure 2.1: Example of a *Drive by Download* attack (source [49]).

[2] https://www.kaspersky.com/resource-center/definitions/drive-by-download

2.1.2 Challenges and Vulnerabilities

JavaScript's overall dynamism and flexibility, although very appealing for developers, do not come without its software engineering challenges, such as program understanding, optimization, and security [68]. A JavaScript program can be extremely complex and difficult to understand using static analysis techniques since its ability to alter and generate code at runtime can lead to unpredictable behavior. This constitutes a problem for the overall comprehension and analysis of the code, and consequently, for implementing code optimizations.

JavaScript's main vulnerability is security, since its popularity makes it a very appealing target for different types of attacks. Due to developers' inexperience, lack of security knowledge, or system complexity this component is often dismissed which leads to lower quality software and causes security breaches [72]. A variety of attacks, such as Cross-site Scripting (XSS), exploit the language's ability to inject code via different techniques, and to modify and access shared objects [57]. XSS is a malicious attack vector that occurs when malicious scripts are injected into a web page. This gives attackers privileged control of the browser and allows the execution of code into the unsuspecting user's browser [1] which can lead to session hijacking, cookie theft, or unwanted and malicious redirects [9].

2.2 Code Transformations

An effective way to conceal the purpose of a JavaScript program is to apply transformations that maintain the code's behavior but compromise its understandability. Two types of code transformations can be distinguished: minification and obfuscation. The first modifies the code to reduce its size, and the latter applies more complex transformations to affect the code's comprehensibility and interfere with its analysis [63]. The major difference between the two techniques is that, while minification also reduces the code readability, it does not make an attempt to hide the original content, contrary to obfuscation transformations [48].

More compact versions of JavaScript code can be obtained through minification. Its main purpose is to reduce load time and bandwidth usage when loading JavaScript.

Obfuscation is the intentional act of creating code that is hardly readable by humans and also resilient to reverse engineering tools [47]. There are various purposes and intentions for applying this technique. It can be used as a security measure, for protecting intellectual property, and preventing plagiarism. Firstly, it mitigates malicious attacks, such as code injections and tampering, since the attacker needs knowledge of the code in order to modify it. Secondly, it can be used by developers to conceal vulnerabilities and flaws in the software, making it harder to be exploited. Finally, it is an effective measure against intellectual property theft as it compromises the code's readability [60]. However, obfuscation can also be used to conceal malicious scripts in order to evade detection by manual code inspection and pattern-based security systems. Therefore, obfuscated malware is often not detected which allows security breaches and attacks on unsuspecting users.

While neither of these techniques are malicious per se, they can be used with malevolent intentions (mainly obfuscation techniques). One of the most common is to conceal malicious code, making it hard to detect by both manual and automatic analysis.

2.2.1 Techniques

To achieve minification of the code, different techniques can be used, such as removing whitespaces, comments, line breaks, and other redundant data. Other optimizations can be performed, such as renaming variables and functions and using implicit conditionals. These transformations do not impact the program flow since indentation characters and specific names are only used to make the code more readable by humans [65, 55].

There are numerous obfuscation techniques with different complexity levels. These techniques' effectiveness can be evaluated based on three complex metrics: potency, resilience, and cost. Potency measures the difficulty of a human to understand the obfuscated code compared to the original non-obfuscated version. On the other hand, resilience evaluates how strong the code is to automatic reverting tools. Finally, the cost is related to how many additional resources the transformed code uses during execution time, and the impact on the originated file size [43, 60].

Ideally, a technique has high potency and resiliency, but low cost. However, this scenario is not realistic. Techniques with high potency and resilience are more complex to implement, as they tamper, for example, with the code's structure and flow, increasing the cost associated. The reverse is also valid. Low-cost techniques tend to be simpler, which leads to lower potency and resilience. A code transformed to be more resistant to reverse engineering tools is also less readable by humans. Notwithstanding, the inverse is not true. A high potency technique does not directly imply high resilience. The dependencies between evaluation metrics are displayed in Table 2.1.

Growth	**Potency**	**Resilience**	**Cost**
Potency		variously	grows
Resilience	grows		grows
Cost	grows	variously	

Table 2.1: Dependencies between obfuscation evaluation metrics (source [69]).

Four categories of obfuscation can be distinguished: *Randomization*, *Data*, *Encoding*, and *Logic Structure* [70]. These are explained in more detail in sections 2.2.1.1, 2.2.1.2, 2.2.1.3, and 2.2.1.4, respectively. In Table 2.2 each technique is assessed in regards to its potency, resilience and cost, distinguishing four levels for each: none, low, medium, and high. This evaluation is based on different sources [44, 70, 62, 34] and analysis of each technique.

Category	Technique	Potency	Resilience	Cost
Randomization	Whitespace injection/removal	low	low/medium	none
	Comments injection/removal	low	low/medium	none
	Variable and function names randomization	medium	high	none
Data	Converting static data to procedures	medium	medium	medium
	Keyword substitution	medium	medium	medium
	Variables reordering	low	medium	none
Encoding	Standard encoding	low	low	low
	Customized encoding functions	high	medium	medium
	Standard encryption and decryption	medium	medium	medium
Logic Structure	Inserting irrelevant code	medium	medium	low
	Function inlining/outlining	medium	high	high
	Function cloning/fusion	high	high	high
	Function reordering	low	medium	none
	Loop transformations	medium/high	high	high
	Control flow flattening	high	high	high

Table 2.2: Evaluation of obfuscation techniques.

2.2.1.1 Randomization Obfuscation

Randomization obfuscation consists of changing elements of the code without altering its semantics. This is achieved with the resource to multiple methods, such as adding or removing indentation characters, as line breaks and whitespaces, and comments. Another simple approach is to rename variables and functions with randomly created names, resulting in names with non-obvious meanings [70]. These types of transformations aim at confusing manual analysis, having little impact in automated analysis systems. Figure 2.2 shows an example of this type of obfuscation, where random comments and indentation characters are added and the identifiers are renamed.

```
function myfunction(txt){
    alert(txt);
}
var mystring = "Hello World!'"
myfunction(mystring);
```

(a) Original code.

```
function
_i23fdfcnj          (_fdij230fdj)
//_32akfaj0ufa
{// _dafaljfamfdn
alert(  _fdij230fdj) ; //
    _dkfahajklaI3
}
var     _dfeakia1f92  //_gcvdseaepk
=   "Hello World!"; //_gpqkjk3424pkl
    _i23fdfcnj(_dfeakia1f92);
```

(b) Obfuscated code.

Figure 2.2: Example of randomization obfuscation (source [70]).

2.2.1.2 Data Obfuscation

Data obfuscation aims at converting variables and constants into the computational result of several operations [70]. Common techniques include converting static data to procedures, keyword substitution, and reordering variables.

Static data can be replaced with statements, or function calls that calculate its value at runtime [58]. This technique can be applied to different data types: booleans, numbers, strings, and arrays. Boolean literals can be transformed into expressions that return the same value but are more complex and harder to understand [45]. Similarly, numbers can be obfuscated by replacing them with equivalent mathematical operations or converting them into strings, as shown in Figure 2.3.

```
let i = 10;
```

(a) Original code.

```
let i = 4511 * (-1) - 8523 + 13044;
```

(b) Obfuscated code.

Figure 2.3: Example of data obfuscation applied to numbers.

Strings can be transformed by splitting them into multiple substrings and then concatenating them, as shown in Figure 2.4. The substrings can be reordered to make the code harder to understand. String splitting is usually used along with dynamic code generation functions (such as *eval*, *document.write*) to execute the concatenated strings [70].

```
let a = "Hello";
```

(a) Original code.

```
let a = "He";
a += "ll";
a += "o";
```

(b) Obfuscated code.

Figure 2.4: Example of data obfuscation applied to strings (source [41]).

Arrays can be obfuscated in regards to their structure or the data they store. One array can be split into various subarrays, and multiple arrays can be merged into one. On the other side, folding and flattening techniques, increasing and decreasing the number of dimensions, respectively, can be used to obfuscate the data stored in an array [60].

Another approach is to substitute JavaScript keywords that are deemed as insecure practices - *eval*, *document.write* -, with variables, as shown in Figure 2.5. This transformation's main goal is to conceal the recurrent use of unsafe keywords from automatic detection systems, although it also reduces the code's comprehension [70].

```
const s = "Hello World"
document.write(s);
```

(a) Original code.

```
const s = "Hello World"
const fo = document;
fo.write(s);
```

(b) Obfuscated code.

Figure 2.5: Example of data obfuscation by keyword substitution.

A final technique aims to reduce the code's readability by changing the variables' order. Although a simple approach, combining it with other more complex techniques, can make the code resistant to analysis.

2.2.1.3 Encoding Obfuscation

Encoding obfuscation consists of obfuscating code through encoding. Three strategies are normally applied to encode original code. The first is to use a standard encoding mechanism such as ASCII, Unicode, or hexadecimal representations. Another more complex approach is to use customized encoding/decoding functions, where the encoding function is used to create the obfuscated code, and the decoding function decodes it in runtime (see Figure 2.6). Finally, the code can also be encrypted and then decrypted during execution time [27].

```
function encode(mystring){
    var c="";
    var i =0;
    for(var i=0;i<mystring.length;i
        ++){
        c=c+
        String.fromCharCode(mystring.
        charCodeAt(i)+ I);
    }
    return c
}
var c = encode("document.write('Hello
     world!')");
document.write(c);
```

(a) Encoding function.

```
function decode(c){
    var mystring="";
    var i =0;
    for(var i=0;i <c.length;i++){
        my string = mystring +
        String.fromCharCode(c.
            charCodeAt(i)-I );
    }
    return mystring;
}
var c ="epdvnfou/xsjuf)(lfmmp! xpsme
    (*";
var mystring = decode(c);
eval(decode(c));
```

(b) Obfuscated code.

Figure 2.6: Example of encoding obfuscation with customized functions (source [70]).

2.2.1.4 Logic Structure Obfuscation

Logic structure obfuscation aims at manipulating the code's execution paths by modifying its logic structure without interfering with its original semantics [37]. This can be achieved by inserting irrelevant code, function inlining and outlining, function cloning and fusion, function reordering, loop transformations, and control flow flattening approaches.

Irrelevant code injection makes the program's structure more complex without changing its initial behavior [46]. Three types of irrelevant code can be distinguished: dead, void, and duplicated. Dead code is code that is never executed. Void code is code whose execution has no impact on the final output. Duplicated code is code that is duplicated from other code. A type of irrelevant code injection is predicate extension - predicates can be extended by adding opaque predicates, increasing the obfuscation potency without altering the condition's result [62], as shown in Figure 2.7.

```
let i = 0;
while (i < 10) {
  i++;
}
```

(a) Original code.

```
let i = 0;
while (i < 10 && 5II.m()[8][1] ==
    h5II.X()[7][2]) {
  i++;
}
```

(b) Obfuscated code.

Figure 2.7: Example of logic structure obfuscation applied to predicates (source [42]).

Another common practice is the inlining and outlining of functions. Inlining consists of replacing a call to a function with its code. Conversely, outlining creates new functions from sequential groups of statements [62]. Functions can also be obfuscated by cloning and fusion. Cloning consists of creating copies of the methods. Fusion joins two or more functions into a single one (fuses the functions' bodies) and creates an additional argument that chooses the one to execute from the joined functions bodies [62]. Functions can also be reordered to compromise the code's understanding.

Loops can be obfuscated with three techniques: tilling, unrolling, and fissing. Loop tilling is the division of the loop's iteration space in smaller blocks (inner loops). Loop unrolling consists of the replication of the loop's body to reduce the number of iterations. Finally, loop fissing breaks the loop into two or more loops with equal iteration spaces and splits the original loop's body over these extra loops [62, 58]. Figure 2.8 shows an example of loop fissing.

```
for(i=1;i<n;i++){
    a[i] += c;
    x[i+1] = d+x[i+1]*a[i];
}
```

(a) Original code.

```
for(i=1;i<n;i++){
    a[i] += c;
}
for(i=1;i<n;i++){
    x[i+1] = d+x[i+1]*a[i];
}
```

(b) Obfuscated code.

Figure 2.8: Example of logical structure obfuscation applied to loops, namely loop fissing (source: [62]).

Finally, the code's structure can be obfuscated with control flow flattening approaches, aiming to conceal the program's control flow graph by obscuring the relations between blocks. One common method to accomplish this is to split all the code's basic blocks and put them inside a single infinite loop with a switch statement that controls the program's flow [17].

2.2.2 Tools

A variety of minification tools are available on the web, such as *Google Closure Compiler* [8], and *UglifyJS* [19]. *Google Closure Compiler* is a minification tool provided by Google. It parses and analyses JavaScript, removing dead code, reducing white spaces, and shortening identifiers. It also performs other optimizations such as inlining, and constant folding [8, 63]. *Uglifyjs* is a JavaScript minifier/compressor, with better compression times, which is able to remove unnecessary brackets, conditions always valued at true/false, unused variables, and arguments [53].

Different obfuscation tools apply distinct techniques or different signatures of the same technique. Several obfuscators are available, from where two can be distinguished due to the complexity and number of techniques they apply: *javascript-obfuscator* [38] and *Jscrambler* [47]. The first is an open-source and customizable tool to obfuscate JavaScript. It implements a vast

range of techniques, such as variable renaming, string extraction and encryption, dead code injection, control flow flattening, and other transformations [38]. The second tool, *Jscrambler*, is an obfuscator provided by *Jscrambler* which offers a vast amount of transformations. It is deemed the most complex obfuscation tool currently available on the market due to the advanced obfuscation techniques it applies [47]. Additionally, in Table 2.3 are presented other 9 tools. This table illustrates a matrix relating the different tools to the obfuscation techniques detailed in Section 2.2.1.

	Jscrambler [47]	***javascript-obfuscator*** [38]	***defendjs*** [4]	***js-obfuscator*** [7]	***JSObfu*** [23]	***JavaScript2img*** [21]	***DaftLogic*** [22]	***jsfuck*** [2]	***node-obf*** [67]	***jfogs*** [25]	***gnirts*** [5]
Whitespace injection/removal	x	x	x	x	x	x	x	x	x		
Comments injection/removal	x	x	x	x	x	x	x	x	x		
Variable and function names randomization	x	x	x	x	x	x	x	x	x	x	
Converting static data to procedures	x	x	x	x	x	x	x	x	x		x
Variables reordering	x									x	
Keyword substitution											
Standard encoding	x	x		x	x						
Customized encoding functions	x										
Standard encryption and decryption											
Inserting irrelevant code	x	x	x								
Function inlining/outlining	x	x	x	x	x	x	x	x	x		
Function cloning/fusion	x		x								
Function reordering	x										
Loop transformations											
Control flow flattening	x	x	x								

Table 2.3: Obfuscation tools and obfuscation techniques matrix.

2.3 Summary

JavaScript has become a prevalent programming language due to its dynamic and expressive nature. However, such characteristics potentiate new vulnerabilities that can be disguised through code transformations, such as obfuscation. Although used for benign reasons, such as intellectual property protection, obfuscation is often used for concealing stealthy malware.

Obfuscation relies on multiple techniques, and there are several tools available that apply them. Although manual inspections easily identify obfuscated code, automatically making this detection is not a simple task, requiring a thorough analysis of JavaScript code features and obfuscation techniques that are usually used for this programming language.

Chapter 3

Obfuscated Code Detection

In this chapter, we present a preliminary literature review on the main topics addressed in this dissertation. The chapter is divided into three sections. The first introduces different subjects, such as code analysis and machine learning approaches to better comprehend the previous work presented in the following section, which includes early solutions for detecting obfuscated code. The third section details a set of challenges and possible solutions that arose from further analysis of the previous work.

3.1 Background

3.1.1 Code Analysis

Static code analysis and dynamic code analysis are two contrasting methods for analyzing the behavior of a program. The first analyzes the source code without executing it, inferring data based solely on source code. This can be achieved, for example, by traversing the code's AST or analysing its bytecode representation. Static analysis is often restricted by the need to presume runtime behavior. In contrast, dynamic code analysis executes the code and gathers data about the runtime value of variables [36].

Due to JavaScript's dynamic and reflective nature, statically analyzing the code is challenging as many behaviors are only defined at runtime. Another constraint to this type of analysis is the diversity of technologies available, which compromises a generic solution. These issues are overcome by dynamic analysis. However, others arise. There is the risk of relevant parts of the code being left unexplored, and, although the behavior of the code is observed, its reasons must be inferred, leading to ambiguous conclusions [61]. Additionally, by executing the code, the machine where the analysis is being performed can be compromised if the code contains malware. A way to surpass some of the issues present in these two types of analysis is to combine them in a hybrid approach benefiting from the advantages of both.

As in most work in the area, we tackle the problem of detecting obfuscated JavaScript by first statically analyzing the code's AST.

3.1.2 Abstract Syntax Tree

An Abstract Syntax Tree, AST, is a representation of source code, in a tree form, that contains its structural and content information, while omitting syntactic details such as punctuation and grouping parenthesis [39]. The nodes in the tree are operations, such as concatenations, loops, assignments, that occur in the source code. The leaves, or terminal nodes, of the tree represent values, such as the value of a string or a number. The AST for a program C can be represented as

$$C = \langle N, T, X, s, \delta, \phi \rangle \tag{3.1}$$

where N is the set of nonterminal nodes, T is the set of terminal nodes, X is the set of values, s is the root node ($s \in N$), δ is a function that maps each nonterminal node (in N) to a group of children, and finally, ϕ is a function that maps each terminal node to its associated value (in X) [30].

3.1.3 Contextual Information

In this work we consider as contextual information any information that retains some degree of its surrounding context. For example, using the sequence of nodes from the AST that connect to a certain keyword, instead of using the keyword by itself. Another way to incorporate some level of context is by constructing features by tracing specific calls. In this work, we focus on the first scenario, and use path-contexts to retain the contextual information of specific nodes. A path-context is a triplet $\langle \mathrm{x_s}, p, \mathrm{x_t} \rangle$ where $\mathrm{x_s}$ and $\mathrm{x_t}$ are the values associated with the start and end terminals of a path p. The path is a sequence of nonterminal nodes and the directions connecting them (up or down in the tree) that connect two terminal nodes. Every snippet of code C can be represented as:

$$C = \{\langle x_s, p, x_t \rangle | x_s, x_t \in X, p \in P\} \tag{3.2}$$

where X is the set of all values associated with terminal nodes and P the set of all paths [30, 54].

3.1.4 Machine Learning

Machine learning is the development of models that are capable of solving a given task based on features present in a training dataset [35]. It addresses two main tasks: predictive and descriptive. The first is learned in a supervised fashion and aims at inducing a model to assign a value to an unlabeled object based on the given predictive attributes. Predictive tasks can be divided into classification tasks when considering categorical domains or regression tasks regarding continuous domains [26]. Conversely, descriptive tasks leverage unsupervised learning to group similar items and retrieve useful insights (clustering algorithms) [51].

In this dissertation, we tackle the problems of detecting obfuscated JavaScript and identifying which obfuscator was applied to transform the code. The models are developed based on labeled training data, and the outcome is if the code is obfuscated or not and what obfuscator was used. The two problems addressed are classification tasks due to the qualitative nature of the target attributes.

3.1.4.1 Classification

Classification tasks are predictive tasks, leveraging from supervised learning, resourcing to labeled datasets. The main goal is to assign a qualitative label to a new object based on given predictive attributes [26]. To accomplish this, the data used is often split into two partitions: the first, training data, is used to induce the model, and the latter, testing data, is used to test the performance of the induced model [26]. A third partition can be made, validation data, used to tune the hyper-parameters of the model.

Two classification tasks can be distinguished based on the number of values the target attribute can have. The most common task is binary classification, where the target attribute can have one of two values. In this case, the class of particular interest is referred to as the *positive* class. When there are more than two possible values for the target attribute, the task consists of a multiclass problem [26].

3.1.4.2 Algorithms

A variety of algorithms can be applied to solve classification tasks. The algorithm's choice must be based on characteristics of the task being addressed, such as the type of predictive features being used and the dataset composition, regarding class balance and size. Based on the categories presented in Moreira et al. [26], five groups of algorithms can be roughly distinguished:

- *Distance-based Learning Algorithms*. Make predictions based on the new object's level of similarity to the other, previously labeled objects. The most known is the k-NN algorithm.

- *Probabilistic Classification Algorithms*. Use the information available to calculate the new object's probability of belonging to each possible class. Naive Bayes and Logistic Regression are examples of this type of algorithm.

- *Search-based Algorithms*. Perform local searches iteratively to induce the best predictive model (DTIA) or search for the model based on a set of predictive rules that retain the best predictive performance (RSIA).

- *Optimization-based Algorithms*. Aim at optimizing a specific function. The most popular ones are Support Vector Machines and Artificial Neural Networks.

- *Ensemble Learning Algorithms*. Create an ensemble of classifiers to combine the predictions of multiple classifiers. Random Forest and AdaBoost are examples of ensemble algorithms.

3.1.4.3 Performance Measures

Various performance measures can be considered to evaluate how well a classifier solves a task. The following measures [26] are applicable in binary classifications but can be adapted for classification tasks with multiple classes.

$$FPR = \frac{FP}{FP+TN} \tag{3.3}$$

False positive rate, FPR, also known as false alarm rate, measure the proportion of false positives among negative cases.

$$FNR = \frac{FN}{TP+FN} \tag{3.4}$$

False negative rate, FNR, also known as miss, is the fraction of incorrectly classified positive cases.

$$Recall = \frac{TP}{TP+FN} \tag{3.5}$$

Recall is the fraction of correctly classified positive cases.

$$Precision = \frac{TP}{TP+FP} \tag{3.6}$$

Precision is the fraction of true positives among all positive predictions.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{3.7}$$

Accuracy is the fraction of true predictions among all cases (positive and negative). It is suited for well balanced problems.

$$F1-score = \frac{2}{\frac{1}{Precision}+\frac{1}{Recall}} \tag{3.8}$$

F1-score is the harmonic mean between precision and recall. It is best suited for problems with imbalanced classes.

A machine learning model should be assessed according to different performance measures, and the priority given to each depends on the specifications of the task being addressed. When using a model to detect obfuscated code, it is more problematic to incorrectly classify a script as regular instead of obfuscated than the other way around. Therefore, greater importance should be given to recall, although F1-score is a good balance between precision and recall.

3.2 Previous Work

Prior projects on the area can be categorized according to the type of code analysis they use, the method they apply to make the classification, the detection techniques, if the impact of using different obfuscators is being considered, and on the detection tasks they address.

This section is divided into three subsections that present an incremental and chronological sequence of approaches. In Section 3.2.1 two projects are presented that aim at detecting malicious obfuscated JavaScript, often conflicting the two concepts. The first approach presented uses string pattern analysis and thresholds manually defined to make the detection. All the following projects apply machine learning algorithms to classify the code. From there, the concept of obfuscated JavaScript is detached from the one of malicious code, and projects aimed at detecting obfuscation independently of the code's intents are presented. The solutions presented in Section 3.2.2 obtain satisfactory results but do not consider the presence of code obfuscated with unknown tools in the testing set or code obfuscated with multiple tools. Two different solutions that address each one of these problems are presented, respectively, in Section 3.2.3.

All the presented projects use static analysis of the code as a step for making the classification. However, other works - Xu et al. [71], and Gorji et al. [37] - use hybrid approaches by complementing static analysis with dynamic analysis or completely dynamic approaches to detect obfuscated malicious code. Both approaches consider obfuscation in the scope of function invocations. The first compares data from static and dynamic analysis to make the detection, and the latter relies solely on dynamic analysis of the code. We consider that these approaches exceed the scope of this dissertation. However, they present different solutions for a variation of the problem at hand and valuable insights into obfuscation techniques and obfuscated code behavior.

3.2.1 Malicious Obfuscated Code

The first approaches that address obfuscated JavaScript detection aim to prevent malicious attacks by focusing on detecting obfuscated malware. These works are based on the fact that many attackers use obfuscation techniques to evade signature-based detectors and compromise the code's readability to surpass manual inspections. A problem with this type of detection is that it can raise false positives in the presence of benign obfuscated scripts.

Choi et al. [32] and Likarish et al. [52] propose two different methods to detect obfuscated malicious JavaScript code. While the first uses string pattern analysis with defined thresholds and is mainly focused on the strings used in calls to dangerous functions; the latter uses machine learning algorithms with features extracted based on specific keyword frequencies and common obfuscation techniques. Although this last work does not focus on strings passed as arguments of specific functions, it still considers similar string properties, such as their length and the presence of specific characters.

These approaches aim at detecting malicious obfuscated code, often conflating obfuscation with maliciousness. The work proposed in this dissertation makes a clear distinction between the concepts of malicious and obfuscated code, acknowledging the detection of obfuscation as

an independent task without disregarding that obfuscation is often used to conceal malware. Additionally, we do not focus on calls to specific functions. However, we incorporate the metrics presented by Choi et al. [32] to evaluate strings as features to one of our models. Some of our features resemble the ones presented by Likarish et al. [52], as they follow a similar rationale.

3.2.1.1 String Patterns

Choi et al. [32] propose an approach to detect malicious JavaScript attacks concealed through obfuscation, based on string pattern analysis. Figure 3.1 represents their methodology, a system with three main modules: a *String Extractor*, a *String Analyzer*, and a *String Deobfuscator*. The first is responsible for tracing and extracting strings used in calls to dangerous functions - *eval* and *document.write*. The strings are extracted using static data-flow analysis, often constructed by tracing different statements (such as string concatenations), which means that some contextual information is taken into account by this approach. The second module is used to analyze the previously selected strings and decide whether they are obfuscated or not. The final module is responsible for deobfuscating the strings.

Regular and obfuscated JavaScript are distinguished using three string properties: *n-gram*, *entropy*, and *word size*. The first measures the usage frequency of ASCII code using the distributions of byte values in strings. The second calculates the entropy to determine the distribution of characters in strings. The last is related to the number of characters in the words in the strings. They create *words* by dividing the strings by the space character. The three properties measured are based on common patterns present in obfuscated strings, which can be encoded, displaying special characters, and have large sizes.

To make the classification based on these properties, thresholds are defined, and if the script analyzed exceeds these values, it is deemed malicious. This method classifies four out of six malicious scripts correctly, but the dataset's size is insufficient to attain reliable conclusions. This approach would also misclassify incorrectly malicious code without string obfuscation. Neither the origin of the files nor the obfuscators used are disclaimed.

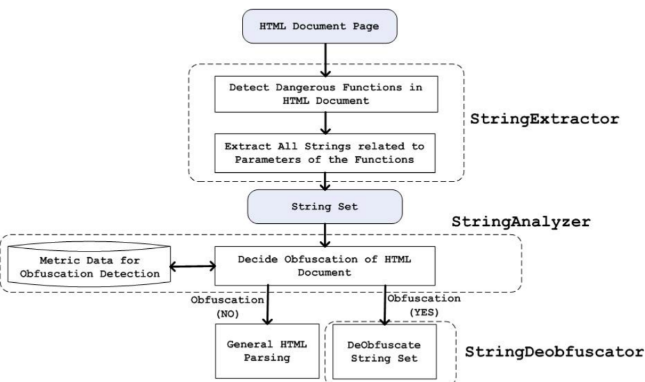


Figure 3.1: Methodology for the automatic detection of obfuscated JavaScript strings in malicious web pages (source: [32]).

3.2.1.2 Keyword Frequency

Other projects use machine learning approaches to detect obfuscation more effectively and surpass the issue of manually defining thresholds. Likarish et al. [52] approach the problem of detecting obfuscated malicious JavaScript with classification techniques.

This work aims to detect malware. Although they acknowledge that not all malware is obfuscated nor all obfuscated code is malicious, the classification of the code and features extracted are based on obfuscation practices. They observed that specific JavaScript keywords' frequency varies between malicious obfuscated code and benign code. Therefore they extract a total of 65 features, mainly consisting of the normalized frequency of each keyword. They create a dataset by crawling different websites. For the benign class, they collect a total of 50,000 scripts from the Alexa [3] 500 most popular websites. They also collect 62 obfuscated malicious scripts from various URLs blacklisted by anti-malware groups. The obfuscators and techniques used in these malicious scripts are unknown as they are collected from the web.

Then they compare the performance of four different algorithms in making the classification. Their models do not misclassify a significant number of malicious scripts as benign, nor benign as malicious. Their SVM classifier obtains a precision and recall of 92% and 74.2%, respectively. They conduct a second experiment with a different set of files, obtaining overall consistent results, validating the robustness of their approach.

3.2.2 Obfuscated Code

In previous work, the concept of obfuscated code is primarily considered in the presence of malware. Consequently, approaches that aim at detecting malicious scripts often use obfuscation as an indicator of maliciousness. This can lead to inaccurate classifications since neither all obfuscated scripts are malicious or all malicious scripts are obfuscated. In more recent works, these two concepts seem to be distinguished more clearly, acknowledging the detection of obfuscated code as an independent task.

Kaplan et al. [48], and Al-Taharwa et al. [28] propose two similar approaches for detecting obfuscated JavaScript regardless of the code's intents. Both approaches extract context-based features by traversing the code's AST and use Bayesian classifiers to make the classification.

The first experiments with different levels of context for features extracted from particular AST nodes. The context levels are related to the AST depth. For example, a 1-level context represents the node's inner surrounding context, while an n-level represents all enclosing contexts. In the end, they compromise effectiveness and efficiency by choosing a flat hierarchy without storing any context. The second project is a semantically based approach, retaining relevant contextual information from all types of AST nodes. A feature contains a chain of contexts representing the scope where it appears. It also addresses a different type of obfuscation, which includes readable patterns.

In this dissertation, we compare two different approaches for detecting obfuscation regardless of the code's intents. One is based on context-based features and another on defined features, with

no context associated with them. However, the context-based features are different from the ones presented in these two works. We do not address the detection of readable obfuscation as it would require manual transformation of the code.

3.2.2.1 AST Node's Context

To our knowledge, the first work that addresses the problem of detecting JavaScript obfuscation independently of the code's intent is conducted by Kaplan et al. [48].

They use automatic techniques to determine if a JavaScript script has been obfuscated or not, providing a confidence score. This approach implements a Bayesian classifier with features extracted from specific nodes (expressions and variable declarations) of the AST. A feature comprises two parts: the context it appears in (sequence of nodes) and the value of the AST node, considering different amounts of context, including flat, 1-level, and n-level. Figure 3.2 displays examples of the features extracted (1-level).

Feature
`try` : `unescape`
`loop` : `spray`
`loop` : `payload`
`function` : `addbehavior`
`string` : `0c`

Figure 3.2: Example of the features used in Kaplan et al. [48] (source: [33]).

To make the classification, they multiply the constituent probability of each feature extracted from the AST and then multiply the result by the prior probability of the label. The dataset used to train and test their solution comprises 563 obfuscated and 3,954 regular JavaScript samples. The obfuscated files were acquired from a team's previous project [33], and the regular samples were obtained from a collection of Firefox browser extensions. The classifier's effectiveness is evaluated by varying the feature vector's size and the levels of context considered and calculating the FNR and the FPR. From the results obtained, they concluded that the FPR increases with the number of features used, possibly due to greater ambiguity, which misleads the classifier. It is also hurt by adding more context to the features, obtaining better rates when using a flat hierarchy. Conversely, the FPR shows that the more features used, the lower its value, although this value is limited by the amount of obfuscated files in the testing set. The classifier is used with a flat hierarchy with 100 features to compromise efficiency and effectiveness in the remaining experiments.

The classifier is then tested with existing and unlabeled JavaScript code collections to understand the use of obfuscation in these collections and validate their solution's effectiveness. Some of the files identified as obfuscated were hand-checked to validate the classification. From this, they found that their approach is susceptible to false positives in the presence of files targeted for encryption. The resulting classifier has a low FPR (about 1%) and a low FNR (about 5%).

3.2.2.2 Readable Obfuscated Patterns

A similar approach is presented by Al-Taharwa et al. [28], which also uses a Naive Bayes classifier, but in contrast, extracts features from all types of AST nodes, maintaining their contextual information. They introduce the concept of readable obfuscated scripts, which was not taken into account in previous works.

Readable obfuscated code mimics practices of plain scripts, making it harder to differentiate from standard programming practices. This is accomplished by preserving the code's formatting and encoding, applying other obfuscation techniques, such as converting static data into procedures. This type of obfuscation requires manual intervention, instead of using automatic obfuscators.

The dataset used contains 1,680 files collected from various sources, mainly by crawling different websites or pre-existing datasets. The proposed system, *JSOD*, is composed of three different components: an *AST generator*, a *Variable Context-Level Feature Extractor*, and a *Bayesian-based detector*. The first is responsible for generating the AST of a given script. The second traverses the tree to extract context-based features. Each feature contains two elements: an abstract word, which includes identifiers and corresponding values, and a context chain, representing the scope of the abstract word in the raw code (see Figure 3.3). The final module, a *Bayesian-based detector*, classifies the code as obfuscated or not. This classifier receives as input a feature vector computed by first creating a vector space with all features extracted from the training and testing sets, and then, for each script applying and embedding function to map its behavior to the vector space.

```
for:+:CallExpression:^:CallExpression:MemberExpression:str
for:+:CallExpression:^:CallExpression:MemberExpression:charCodeAt
```

Figure 3.3: Example of JSOD's features (source: [28]).

They recreate (and adapt) Kaplan et al. [48] solution to be able to make a more direct comparison between the two, concluding that their approach by considering readable obfuscation patterns (which are present in the training and testing sets) outperforms the first in terms of precision and recall concerning the obfuscated class. When using a set of the ten most discriminative features, the solution has a higher precision (around 96.7%).

3.2.3 Multiple Obfuscators

The works previously described are able to distinguish between regular and obfuscated JavaScript code with satisfactory results. However, they do not consider that distinct obfuscators apply different techniques or the same techniques with different signatures. This aspect of the detection task is valuable to more accurately mimic the real use of obfuscation and increase the probability of detecting code obfuscated with unknown tools.

Both Tellenbach et al. [65], and Skolka et al. [63] address the importance of using multiple obfuscators in detecting obfuscation. However, they use different approaches. The first solely

aims at distinguishing between regular and obfuscated code but validates the developed model with known and unknown tools. The second approach also aims at identifying which tool was used to obfuscate the code but does not consider the possibility of the code being transformed with a tool never seen by the model.

This dissertation acknowledges the use of multiple obfuscators and techniques in the dataset used and the tasks addressed as we also aim to identify the obfuscator applied (from a set of known obfuscators). Additionally, we validate our solution in the presence of code transformed with unknown tools to understand possible limitations in its generalization.

3.2.3.1 Testing with Unknown Tools

Tellenbach et al. [65] acknowledge the importance of testing their solution with code obfuscated with known and unknown tools when detecting obfuscation.

To build their dataset, they collect benign and malicious code from various sources. The benign files were obtained from *jsDelivr* [14] and the Alexa [3] top 5,000 websites. The malicious samples were collected from *MELANI* [1]. While the malicious files already had some degree of obfuscation, the benign files extracted were preprocessed to ensure no transformations had been applied and remove duplicated files. Additionally, they enrich their dataset by minifying some files - with *UglifyJS* [19] - and obfuscating others - with *javascriptobfuscator.com* [20] (2 configurations, advanced and standard), *javascript-obfuscator* [38], *jfogs* [25], *JSObfu* [23], and *Google Closure Compiler* [8] [2]. The final dataset comprises 101,974 regular and obfuscated files in addition to 2,706 malicious (and obfuscated) files.

They extract a total of 45 features by traversing the code's AST and compare the performance of nine different classifiers. The features are computed based on standard obfuscation techniques, such as whitespace removal, hexadecimal or Unicode characters' encodings, and converting static data into procedures. They also use features that reflect the frequency of common JavaScript keywords.

The best of the tested classifiers, the Boosted Decision Tree, correctly classifies obfuscated code with an F1-score of around 99% when using all 45 features and around 98% when using only the 20 most descriptive features. This classifier is then used to assess how their model performs with code obfuscated with a tool not present in the training set. To accomplish this, they create two testing sets, one with the files obfuscated with a specific tool (testing set 1) and the other with 30% of the remaining scripts (testing set 2). The rest is split into training and validation sets (therefore these sets, do not contain code transformed by the specific tool). After training and validating, the model is used to classify the scripts in both testing sets. Since in testing set 2, the obfuscators used are also included in the training set, the model performs well, which is not the case in testing set 1. The results obtained by classifying the files from testing set 1 have significant variance depending on the assessed obfuscator. They conclude that some tools are more similar

[1] Swiss Reporting and Analysis Centre for Information Assurance.

[2] Although a minifier and not an obfuscator, they use it to transform smaller scripts that are usually left unchanged by other tools.

than others. Therefore, it is possible to generalize some obfuscators but not all, emphasizing the importance of using various tools and diverse techniques while training the model.

3.2.3.2 Tool Identification

In more recent work, Skolka et al. [63] compare two different machine learning approaches in solving three tasks. The first approach, via identifier frequencies, consists of creating a feature vector that summarizes identifiers' names occurrence and frequency in the code and then classifies it using an SVM. This approach is limited to a single feature of the code. Therefore, a second one, via AST convolution, is also tested. In this case, they enrich the AST with relevant information for detecting obfuscation (such as information about whitespaces and identifiers length) and adapt an existing machine learning architecture for classifying trees, a *Tree-based Convolutional Neural Network*, to make the classification. This means that the entire AST is being used to make the classification, maintaining its contextual details. In this case, no feature engineering is required since the neural network extracts the most descriptive features from the entire AST. To our knowledge, this work is the first to use this architecture for identifying transformed code. It should be noted that this classifier can be very memory-consuming if used with large ASTs and performs better with large amounts of data [66].

In regards to the classification tasks addressed, a simpler one aims at determining if a script has been transformed, either by minification or obfuscation. Another task is to distinguish regular JavaScript code, which includes minified code, from obfuscated code. A third task is to identify the obfuscation tool applied to transform the code (from a set of available tools).

The dataset used to train and test the models depends on the task being addressed. They randomly sample 10,000 files from a set of previously collected regular JavaScript files (regular files, with no obfuscation and no duplicates, collected from open-source projects) and apply transformations to copies of those files accordingly to the task being addressed. These files are restricted to sizes between 1 KB and 10 KB. They use five obfuscators - *javascript-obfuscator* [38], *javascriptobfuscator.com* [20], *DaftLogic Obfuscator* [22], *jfogs* [25], and *JSObfu* [23] - with a total of 15 different configurations, and six minifiers - *UglifyJS* [19], *babel-minify* [6], *Google Closure Compiler* [8], *javascript-minifier.com* [31], *Matthias Mullie Minify* [18], and *YUI Compressor* [24] - with a total of 31 configurations. This means that at most, each model is trained with 470,000 files, however this information is not disclosed.

Their results show that the *Tree-based CNN* outperforms the SVM model, obtaining accuracy values of 95.06%, 99.96%, and 99.68%, respectively, in each task. They apply their detector to existing collections of JavaScript and present an empirical study of code transformation techniques in the wild, reaching several relevant conclusions. They conclude that around 38% of scripts are transformed (mainly minified); that *DaftLogic Obfuscator* [22] is the most popular obfuscation tool applied; and that to load code at runtime via *eval* is the most common obfuscation technique practiced.

3.3 Challenges

After further investigation of the problem and analysis of the previous work, we identify several challenges to the successful detection of obfuscation in the wild.

3.3.1 Diversity of Techniques and Tools

There are numerous techniques and tools available for obfuscating JavaScript code. Different tools apply varying techniques or the same technique in distinctive fashions. For this reason, while detecting obfuscated code, it is essential to consider as many tools, and consequently, techniques as possible. This heterogeneity allows the development of a robust solution, able to detect diverse obfuscated code, which is what is expected to be found in the wild.

Previous projects address this challenge in the created datasets. For this purpose, two approaches are distinguished. The first is to create the entire dataset by collecting regular and obfuscated JavaScript from web pages. The second approach gathers regular JavaScript and then applies one (or more) tools to create an obfuscated set. The dataset used in this dissertation comprises code transformed by multiple obfuscators and configurations to incorporate different techniques and have a broader representation of obfuscation practices and was created following the second methodology described (see Section 4.2).

3.3.2 Unknown Obfuscators

When considering multiple obfuscators, the scenario where the testing set contains code transformed by an unknown obfuscator must be addressed. Only one of the presented projects explicitly takes this situation into account - Skolka et al. [63]. In projects where the dataset is collected from crawling different websites or using pre-existing collections, such as in Likarish et al. [52], Kaplan et al. [48], and others, since the obfuscation tools are unknown in general, this situation is possible, but the results are not analyzed under this perspective.

This dissertation addresses two tasks, the detection of obfuscated code and the identification of the tool used, and this scenario can be an issue in both of them. Two situations can occur: the training set is composed of code transformed by obfuscators similar to the unknown obfuscator; or the tools used to obfuscate the code in the training set are very distinct from the unknown obfuscator. In the first task, distinguishing obfuscated from non-obfuscated code, the model has a higher probability of detecting the obfuscation in the first scenario than the second. In the task of identifying the obfuscator, this situation is more critical since the classification will always be incorrect, as there is no class representative of this new tool. We validate our binary solution in the presence of unknown tools and analyze its results (see Section 6.5).

3.3.3 Minified Code

Previous work shows that minification is more common in the wild than obfuscation [63]. Therefore, a dataset containing minified code is a more realistic representation of JavaScript code and

programming practices. Some of the previous works presented consider the often use of minification by incorporating this type of transformation in their dataset and classifying it either as regular or minified - Tellenbach et al. [65], and Skolka et al. [63]. However, the majority of them do not (although unclear in some cases). This can be a problem when the solutions compute features based on identifiers' length or frequency of indentation characters - which could indicate the presence of minification, but not necessarily obfuscation, leading to false positives. Even when this is not the case, the solution should be tested in the presence of minification to understand the impact of this type of code in the detection of obfuscation.

Since this dissertation's main focus is to determine if a script is obfuscated, it is not of great interest to train a classifier to detect minified code. We incorporate this type of code by labeling it as regular JavaScript and then classifying it as such. This leads to more robust classifications that can distinguish obfuscation from minification and give reliable results when used to classify code with unknown origins. Additionally, we train the model with and without minified code and compare the results of predicting minified code, to better understand the importance of incorporating minification in the training set (see Section 6.6).

3.3.4 Partial Obfuscation

Code is partially obfuscated when obfuscated code appears embedded in regular code. This situation occurs when a part of a JavaScript file (e.g. a method definition) is obfuscated, and the remaining is not. If the transformed portion is considerably smaller than the regular one, a detection system can be deceived to classify it as regular code instead of obfuscated.

Neither of the previous works addresses this situation. Therefore, the behavior of these solutions in the presence of partial obfuscation is unclear. However, one can assume, since all of them consider the program as an all, if the portion of obfuscated code is considerably smaller than the regular code, they would not be able to detect the obfuscation.

This dissertation addresses this challenge by first validating the solution against partially obfuscated code and then adopting a different methodology solely used to detect obfuscation in this scenario (see Section 6.7).

3.4 Summary

Several approaches have been proposed to detect obfuscated JavaScript. The main aspects that differentiate these works are the type of code analysis implemented, the type of features extracted, and the classification techniques applied. Another relevant aspect is if the solutions are robust to using different or unknown obfuscation tools to transform the target code.

It is also possible to distinguish two main approaches to retrieve insightful information from the code. The most common is using a set of features defined based on standard obfuscation practices. The second approach is to use features that retain some contextual information by using paths of the AST or the entire tree. This dissertation presents a comparison between two variations of these approaches, aiming to understand which one is more effective in detecting obfuscation.

Table 3.1 presents a brief qualitative comparison of the previous work in regards to the topics mentioned, namely the classification approaches and tasks each work addresses. Since the approach proposed in this dissertation leverages static analysis of the code and machine learning algorithms to make the detection, the works reviewed also use similar approaches. This table also indicates if the solution retains any contextual information in the features used, and if it considers the presence of minified or partially obfuscated code in the training or testing sets.

Work	Year	Classification Approach	Retains Contextual Information	Detects Obfuscation	Considers Code's Intents	Identifies Obfuscator	Considers Unknown Tools	Considers Minification	Considers Partial Obfuscation
Choi et al. [32]	2009	thresholds	yes	yes	yes	no	no	no	no
Likarish et al. [52]	2009	machine learning	no	yes	yes	no	no	no	no
Kaplan et al. [48]	2011	machine learning	yes	yes	no	no	no	unclear	no
Al-Taharwa et al. [28]	2014	machine learning	yes	yes	no	no	no	unclear	no
Tellenbach et al. [65]	2016	machine learning	no	yes	no	no	yes	yes	no
Skolka et al. [63]	2019	machine learning	no	yes	no	yes	no	yes	no

Table 3.1: Previous work qualitative comparison.

Additionally, Table 3.2 depicts a more quantitative comparison of the projects. Besides presenting the classification tasks, it also shows the size of the datasets and the number of obfuscators (and configurations) each uses. The results obtained by the various solutions are also presented in this table. They are not directly comparable since different approaches, evaluation metrics, and datasets (regarding size and content) are used. However, the ones obtained by the more recent works are quite promising, being a good indicator that tasks of classifying obfuscated code and identifying the obfuscator used can be successfully solved.

Work	Dataset Size (number of files)	Number of Obfuscators	Classification Task	Evaluation
Choi et al. [32]	6	Unknown	Detect obfuscated malicious JS	Accuracy: 66.67%
Likarish et al. [52]	50,062	Unknown	Detect obfuscated malicious JS	Average precision:87.52% Average recall:73%
Kaplan et al. [48]	4,517	Unknown	Detect obfuscated JS	FPR: 1% FNR: 5%
Al-Taharwa et al. [28]	1,680	Unknown	Detect obfuscated JS	Precision: 96.7%
Tellenbach et al. [65]	104,680	5 (6 configurations)	Detect obfuscated JS	F1: 99%
Skolka et al. [63]	>10,000	5 (15 configurations)	1) Detect transformed JS 2) Detect obfuscated JS 3) Identify obfuscator	1) Accuracy: 95.06% 2) Accuracy: 99.96% 3) Accuracy: 99.68%

Table 3.2: Previous work quantitative comparison.

A limitation of the first four solutions presented is the datasets used. They are either small, unbalanced, use a small set of tools, or have no preprocessing to ensure the filtering of unwanted code. This is not the case in the two last solutions, as the datasets are large, incorporate various obfuscators, and were previously preprocessed to ensure their quality. Additionally, it is possible to identify several challenges for detecting obfuscation in the wild: diversity of the data, unknown tools, presence of minified code, and partial obfuscation. From the presented tables, we can conclude that no solution addresses all these challenges simultaneously. The columns *Dataset Size*

and *Number of Obfuscators* from Table 3.2, show that Tellenbach et al. [65] and Skolka et al. [63] use the most diverse datasets, considering a variety of tools and multiple configurations, incorporating several techniques. These two are also the only ones that intentionally consider minification, as shown in column *Considers Minification* from Table 3.1. Tellenbach et al. [65] approach is the only that validates the performance of their solution with unknown tools (column *Considers Unknown Tools*, Table 3.1). However, none of the presented solutions acknowledges the presence of partial obfuscation (column *Considers Partial Obfuscation*, Table 3.1). Although most of the challenges are addressed to some degree by the presented solutions, further analysis is required to develop a robust and reliable solution that can detect obfuscation in a vast range of scenarios, which what we aim to achieve in this dissertation.

Chapter 4

Detector Implementation

In this chapter, we describe in more detail the problem addressed by this dissertation, the context, and implementation of the proposed solution, including all the steps to develop the detector.

4.1 Context

The main problem we address is the automatic detection of obfuscated JavaScript code. We implement a detector whose main goal is to distinguish between regular code and obfuscated code. Additionally, the detector also identifies the obfuscator used to transform the code, from a set of known obfuscators. The detector receives as input a snippet of JavaScript code and outputs a label that identifies it as regular or obfuscated, or if required, identifies the obfuscator used to transform the code.

The detector should be able to classify JavaScript snippets with low false positive and false negative rates. However, classifying incorrectly obfuscated samples may be considered more problematic than classifying incorrectly regular samples - namely when considering the specific case of obfuscation being used to conceal malware and a next step of validating if the code is malicious or not. If the detection of the obfuscated code fails, the attack goes unnoticed, possibly hindering unsuspecting users. If a regular file is considered obfuscated, further manual investigation can determine if the file is actually malware or not and release it if is not.

4.2 Dataset

An essential component for solving a classification task is the dataset used. Our goal is to have a large and diverse dataset that incorporates different types of obfuscation - namely in the obfuscation techniques used and obfuscators applied to transform the code - to have a broader representation of obfuscation practices. Additionally, since minification is commonly used in JavaScript code and obfuscation often applies minification techniques, minified code should also be included

in the dataset, to reduce the incorrect classification of minified files. We consider minified code as regular code since it does not attempt to conceal the code's behavior, conversely to obfuscation. To the untrained eye, minification and obfuscation can be easily confused, adding to the importance of training a detector to distinguish between them.

Since we apply supervised machine learning algorithms, a labeled dataset is required. Two approaches can be used to build this dataset. The first is to gather regular JavaScript code and obfuscated code from different sources. The other is to use a non-transformed JavaScript code dataset as a basis and transform it with the desired obfuscators and transformation techniques. The first approach would require rigorous preprocessing to ensure that the code labeled as regular is regular code, and the code labeled as obfuscated is obfuscated. Additionally, it is not a viable manner for obtaining a large corpus of files obfuscated by different tools for three main reasons: obfuscation is not very common in the wild, which would require the collection and processing of a large number of files; it would require further analysis of the code to identify the obfuscator used; different obfuscators are applied more often than others which could compromise the balance of the dataset. Conversely, the second approach is a safer alternative that grants more control of the dataset. It is only required to ensure that the regular code is regular and not obfuscated or minified. This is not a trivial task but is more straightforward than detecting obfuscated code, as files with specific characteristics can be filtered from the dataset for potentially being transformed. Additionally, by using a defined set of obfuscators and minifiers to transform the code, it is easier to balance the dataset and control which tools and techniques are being used. We opt for the second approach.

The dataset creation comprises three main phases: data collection, data preprocessing, and data transformation. All the files generated are then stored and labeled appropriately:

- All regular and minified files are labeled as *Regular*.
- All obfuscated files receive two labels, the first indicating that they are obfuscated, *Obfuscated*, and the second label representing the obfuscator used to transform the code.

The workflow pipeline is presented in Figure 4.1.

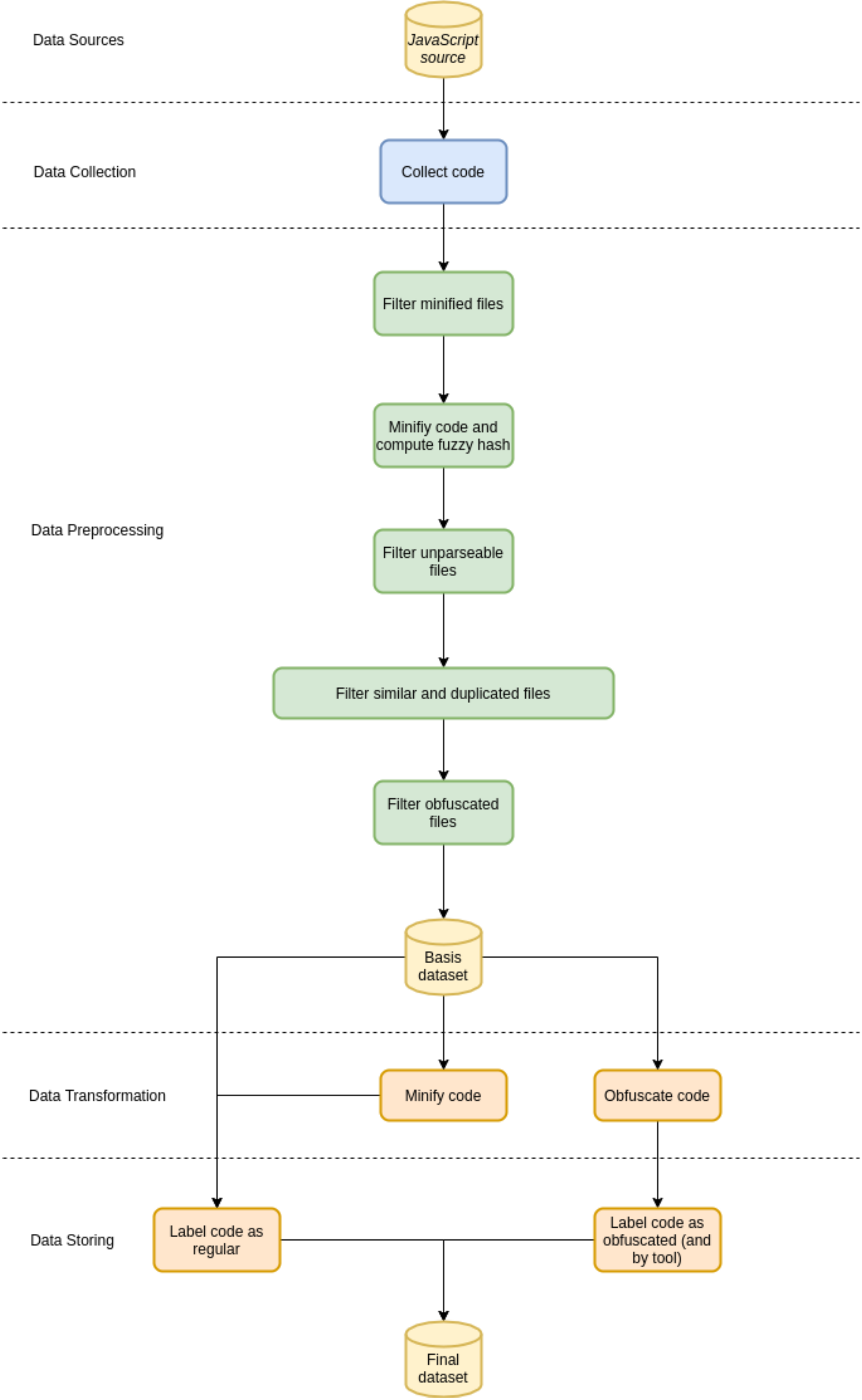


Figure 4.1: Dataset creation pipeline.

4.2.1 Requirements

The basis dataset (from where the final dataset is created, by applying the transformations) needs to verify a set of requirements:

- **The code must be representative of different types of JavaScript.** This is necessary to eliminate any possible bias that could occur if only one type of JavaScript was used, such as client-side code. Although obfuscation is primarily used on the client-side (since anyone with a browser can have access to the code), the dataset should comprise different types of JavaScript code to train a model able to detect obfuscation in different scenarios.

- **The code must not be transformed (minified or obfuscated).** The data collected will be transformed, in a next step, by a pre-defined group of minifiers and obfuscators. Although obfuscation often employs minification techniques, it is not always the case. Therefore, obfuscating minified code can lead to inaccurate representations of code transformed by specific tools, hindering the model's classification. This is also true when obfuscating previously obfuscated code, as different tools transform the code in different fashions.

- **There can not be duplicated and similar files.** This could lead to unrealistic results, better than expected, when similar files appear in the training and testing sets.

- **The code must be parseable.** Since the code will be parsed before being used in the classifier, files that fail to be parsed must be discarded to avoid issues downstream in the pipeline.

4.2.2 Sources of JavaScript Code

To create the dataset, three main sources of code were used: websites listed on *Majestic Million* [1], repositories on *GitHub* [2], and *NPM* [3] libraries. In total, 148,349 JavaScript files were collected from these sources.

Majestic Million. To have a more realistic representation of scripts found on the web, client-side JavaScript can be obtained by crawling different websites. The code can be collected by extracting inline scripts or by downloading code referenced by external files in the HTML of the website (see Figure 4.2). The list of sites visited was obtained from the *Majestic Million* service. This service offers a ranking of the top sites worldwide free of charge (contrarily to Alexa [3] that contains a paywall). We start by scrapping *Majectic Million* using Python and *BeautifulSoup* [4] to retrieve the list of sites to visit. Then we implement a scrapper in Python, using *Selenium* [5] to visit each site. After allowing the loading of dynamically generated code for at most two seconds, the scrapper parses the HTML code using *BeautifulSoup* to extract the script tags and

[1] https://de.majestic.com/reports/majestic-million
[2] https://www.github.com
[3] https://www.npmjs.com/
[4] https://pypi.org/project/beautifulsoup4/
[5] https://pypi.org/project/selenium/

their content. Inline scripts are copied to files, and external references are downloaded. In total, 74,704 JavaScript files were extracted from around 3,500 websites. In the remaining of this work, we refer to the files collected from this source as being collect from the *web*.

```
<script type="text/javascript">
// some code
</script>
```

(a) Inline script.

```
<script src="/someFile.js" type="text
    /javascript"></script>
```

(b) Code referenced by an external file.

Figure 4.2: Example of scripts found in HTML code.

GitHub. Another source of code is the set of software hosting platforms, such as *GitHub*, which contain a large variety of code for open-source websites, applications, and other programs. We collect three types of repositories from *GitHub*: browser extensions, programs written in vanilla JavaScript, and repositories containing server-side code. Browser extensions are small software units used to customize the browsing experience. They run on the browser, and therefore, their code is solely client-side. The vanilla repositories are mostly small JavaScript programs written without resourcing to external libraries. These repositories and the ones containing server-side code were included in the dataset to increase its code diversity. To accomplish this, we start by developing a JavaScript program based on the *octokit/core* [6] library that retrieves the list of repositories using *GitHub*'s REST API [15]. To retrieve the different types of repositories we make different requests, as shown in Figure A.1. Then, we implemented a Python script that clones the repositories in the list. For each type of repository, the 300 most stared repositories were cloned, obtaining a total of 25,354 JavaScript files. In the remaining of this work, we refer to the files collected from this source as being collect from *GitHub*.

NPM. Various JavaScript libraries were included in the dataset. We downloaded the top 200 most dependent-upon libraries listed in a rank [7] on a *GitHub* repository, and their dependencies, adding to the dataset a total of 48,291 JavaScript files from this source. To accomplish this, we developed a small Python script that goes through the list of packages and installs them via *NPM*. In the remaining of this work, we refer to the files collected from this source as being collect from *NPM*.

4.2.3 Preprocessing

To ensure the dataset meets the requirements defined in Section 4.2.1, we preprocess all collected files. To accomplish this, we developed a program written in JavaScript that receives as input a set of files and filters the ones that do not fit our requirements. Figure 4.3 depicts the distribution of files kept and filtered per data source. It is possible to verify that most of the collected files were

[6] https://www.npmjs.com/package/@octokit/core
[7] https://gist.github.com/anvaka/8e8fa57c7ee1350e3491

filtered. The ratio between files kept and filtered is greater for files collected from the web. After preprocessing, the dataset contains 35,687 files, around 24% of all the JavaScript files collected.

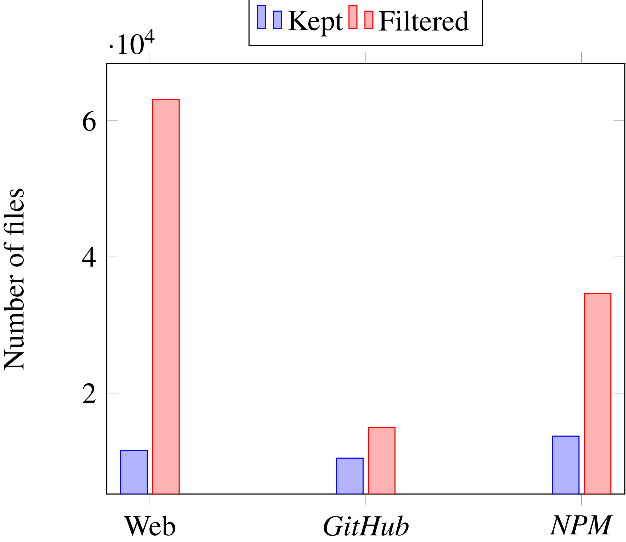


Figure 4.3: Number of files kept and filtered per data source.

The preprocessing is divided into four main steps: filter minification; filter duplicated files; filter unparseable code; and filter obfuscation. Additionally, files are also discarded if they are empty or due to timeout in the preprocessing. Figure 4.4 shows the distribution of files filtered per motive of being filtered, from where it is possible to conclude that most files are filtered because they are minified or duplicated from other files.

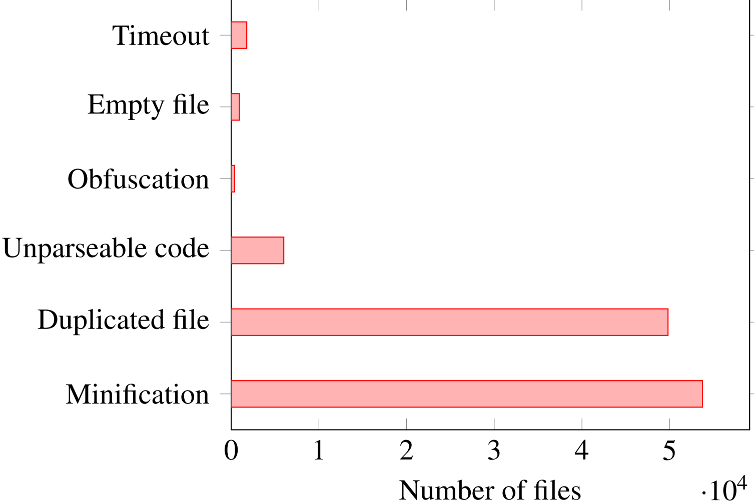


Figure 4.4: Number of files filtered per motive.

4.2.3.1 Filter Minification

A file is marked as minified if it displays at least one of the following characteristics:

- A *min.js* extension (based on the fact that this extension is commonly used to identify minified code).
- Less than 1% of indentation characters (based on [65]).
- On average, more than 100 characters per line (based on the fact that it is standard to keep the line length at 80 characters [8], and rounding it to 100 to be more tolerant to programming preferences).
- More than 10% of all lines has more than 240 characters (adapted from [65]. Instead of 1000 characters, we use 240 to capture smaller minified lines. 240 is three times the standard line size).

It should be noted that code comments are overlooked by these heuristics, as they are first removed from the code. This is done to prevent files with extensive comments from being marked as minified or minified code to be concealed by comments.

The majority of files (53,748 files) were filtered due to minification. This is also the main reason for filtering files collected from the web, where around 66% of files were marked as minified. This can be explained by the frequent use of minification to reduce load time and bandwidth usage.

4.2.3.2 Filter Duplicated Files

To filter duplicated and similar files, we compute the context-triggered piecewise hash (also known as fuzzy hash) for the minified version of each file and compare it with the ones computed for previously processed files. This type of hash is a combination of other hashing algorithms and allows the identification of similar files [50].

We start by parsing the files with *esprima* [12] and minifying them with *escodegen* [11] to remove any comments, indentation characters, and specific identifier names that could hinder the comparison. These libraries do not add any randomness to the minification; therefore, two equal files output the same minified code. Then we compute the fuzzy hash values using *ctph.js* [10]. This library allows to compute fuzzy hash values and compare their similarity scores (from 0% to 100% similar). If two files have a similarity score greater or equal to 40%, one of them is discarded. This value was obtained by trial and error and manual evaluation of multiple files. We prioritize the uniqueness of the files instead of the number of files in the dataset, therefore using a low threshold value decreases the probability of having duplicates. In this step, 49,816 files were filtered.

[8] Based on IBM's punched card, which had 80 columns [16].

4.2.3.3 Filter Unparseable Code

In the previous step, to filter duplicated files (Section 4.2.3.2), the code is minified and therefore parsed. The code is parsed with the same parser that we use in the parsing stage (see Section 4.3), and then minified. If there are any parsing errors we discard the file. In this step, 5,997 files were discarded.

4.2.3.4 Filter Obfuscation

Although obfuscation often applies minification techniques and therefore the first step of removing minified code (see Section 4.2.3.1) would filter these files, this is not always the case. Therefore an additional step to filter these files is required. In this step, 392 files were filtered. We consider a file as obfuscated if it displays three or more of the following characteristics:

- On average a string entropy greater than 1.2 (based on [32]).
- On average a string 1-gram greater than 60 (adapted from [32]).
- On average the words [9] in the strings have more than 350 characters (base on [32]).
- On average an identifier [10] length equal to 1 character (based on the fact that obfuscation often minimizes the identifiers).
- On average less than 70% of letters per identifier (based on [52]).
- On average more than 5% of encoded identifiers, strings or numbers (based on the fact that identifier, string, and number encoding are a techniques often applied by obfuscators).

4.2.3.5 Filter Others

Files with one or fewer bytes were also discarded (935 files). Additionally, files that took more than five minutes to be processed were also discarded, mainly due to performance reasons [11] (1,774 files were discarded due to timeouts).

4.2.4 Transformation

Different copies of the original files were transformed with various tools to represent all our transformations and create the final dataset. Most of the tools used are open-source and available via *NPM* packages, which means they are easily accessible and simple to use. Some of the tools can be customized with various configurations, which we take advantage of to have different levels of obfuscation. The configurations for the obfuscators were chosen based on three criteria:

[9] We consider a word any sequence of alphanumeric (and the underscore character) characters inside a string.

[10] We use the term *identifier* to represent three types of identifiers: variables, functions/methods, and parameters declared in the program. This heuristic (and similar) is applied to these three types of identifiers independently.

[11] In general, the processing time for a file is less than ten seconds.

- **Templates provided in the tool's documentation.** If the tool provides templates with configurations, some of them are used, following the rationale that they are more commonly applied to transform the code.
- **Variety of transformations.** If the tool offers a variety of transformations, choose configurations that reflect those transformations.
- **Similarity to other tools.** Choose configurations that use transformations similar to the ones applied by other tools to represent common obfuscation techniques. For this purpose we define a common obfuscation configuration, referenced as *config1*, that attempts to use a similar set of techniques for all tools: *whitespace removal*, *comments removal*, *variables and function names randomization*, *converting static data to procedures*, *function inlining/outlining*. However, depending on the tool, other techniques are also used since they may not be disabled.

The tool choice was made based on the techniques applied by the tools and their accessibility (for more details on the tools, see Section 2.3). To obfuscate the code in the dataset, nine tools were applied:

- ***Jscrambler*** [47]: it is not an open-source tool; rather it is a commercial product from the company with the same name. It uses various techniques to transform the code and is highly customizable. Four different configurations were used for this tool: the first is the *config1* configuration, and the three remaining are templates available on this tool's documentation, representing different levels of obfuscation.
- ***javascript-obfuscator*** [38]: it is an open-source tool available on *NPM*. It uses various techniques to transform the code and is highly customizable. Four different configurations were used for this tool: the first is the *config1* configuration, and the three remaining are templates available on this tool's documentation, representing different levels of obfuscation.
- ***defendjs*** [4]: it is an open-source tool available on *GitHub*. It uses various techniques to transform the code, some of which can be customized. Two configurations were used: the first is the *config1* configuration, and the latter applies stronger obfuscation techniques.
- ***js-obfuscator*** [7]: it is an open-source tool available on *NPM*. It uses various techniques to transform the code, some of which can be customized. Three configurations were used for this tool: the first is the *config1* configuration, and the two remaining apply other obfuscation techniques provided by the tool.
- ***JSObfu*** [23]: it is an open-source tool available on *RubyGems* [12]. It uses various techniques to transform the code. It is configurable by setting the number of iterations, representing the number of times the code is obfuscated. We use one iteration to obfuscate the code. In some

[12] https://rubygems.org/

cases, when the original code is not very complex, the output is very simple, i.e. we can say that it has low potency and resiliency. For this reason, obfuscated files with less than 30 bytes were discarded. Figure 4.5 shows a snippet of code transformed by this tool that is less than 30 bytes. As shown, the obfuscated code, although different from the original code, is very simple, failing to conceal its intended behavior.

```
var $ng_adcode_user_is_supporter = 1;
```

(a) Original code.

```
var y='k'.length;
```

(b) Obfuscated code.

Figure 4.5: Example of code transformed by *JSObfu* with a size smaller than 30 bytes.

- ***JavaScript2img*** [21]: it is an online tool, therefore, the obfuscation process required the implementation of a crawler to visit the website and transform the code. It is not configurable. The process of transforming the code with this tool is very time-consuming, often failing for larger files. Therefore we only transform around 10,000 files, all smaller than 20 KB (which took around a week to accomplish).
- ***DaftLogic*** [22]: it is an online tool, and like with *JavaScript2img* we encountered some limitations while transforming the code. We use the same thresholds applied to transform the code with *JavaScript2img*.
- ***jsfuck*** [2]: it is an open-source tool available on *GitHub*. It is not configurable.
- ***node-obf*** [67]: it is an open-source tool available on *NPM*. The identifiers are also randomly generated based on a given symbol, we use *$$*. The remainder of transformations are not configurable.

Other obfuscators such as *jfogs* [25] and *gnirts* [5] were also explored; however, due to the simplicity of the transformations they apply, the code transformed with these tools was not incorporated in the final dataset. The first works by creating a single function that includes all the code. Then all the terminal nodes are replaced by identifiers that are passed as arguments to the main function. In some cases, the transformed code is very similar to the original. The second tool, *gnirts*, only transforms strings, preserving the rest of the code.

The configurations we use for all configurable obfuscators are detailed in Section A.2. All obfuscators have one common configuration, *config1*. When we refer to this configuration from a specific obfuscator, we use the name of the obfuscator followed by a hyphen, and *config1*, e.g. *Jscrambler-config1*. The remaining configurations are obfuscator-dependent and do not match with configurations of other tools. These configurations are represented by the obfuscator's name followed by a hyphen and a number, e.g. *Jscrambler-2*. In Section A.2 we also present examples of code obfuscated by each obfuscator.

In order to minify the code, we used two tools: ***UglifyJS*** [19] and ***Google Closure Compiler*** [8]. Both tools are open-source and available on *NPM* and although they offer some configurations, the default versions were used. Other minifiers, such as *YUI Compressor* [24] and *babel-minify* [6], were taken into account. However, as in general they output similar code to the ones used [63], differing mainly in the names used in the variables and small optimizations applied, the two selected were considered enough.

After this process, each regular file in the dataset generates, at most, eleven additional files - since we apply nine obfuscators and two minifiers to transform the original file - that are then labeled and added to the dataset. Ideally, all files would be transformed successfully with all tools, allowing a balanced number of samples in the different classes. However, due to time, size, and transformation restrictions, this was not possible. In the following section we report how many files in the final dataset were successfully obfuscated and parsed per tool.

The program to transform the files is mainly written in JavaScript since most tools are available via *NPM*. To use *JSObfu* we develop a small Ruby script, as it is only available on *RubyGems*. Finally, to access the online tools, we use Python with a *Selenium* web driver to be able to interact and issue requests to the pages.

4.3 Parser

We leverage static code analysis to retrieve insightful information. We parse the code by computing and traversing its AST. To accomplish this, we modify an in-house, previously developed, parser that receives as input a set of JavaScript files and computes, for each file, a set of path-contexts (as defined in Section 3.1.3).

The parser is written in Python and JavaScript, and it uses *esprima* [12] to parse the code into an AST. *esprima* is a high performance and well-documented library. As JavaScript has many available features and an extensive grammar, only a subset of commonly used nodes is considered by the parser (around 60 different nodes). However, it is easily extensible to incorporate new nodes if required.

To extract the necessary information from the code, two main alterations were made to the parser. The first was to alter the paths to be more generic and representative of the code's overall structure, without storing irrelevant information, as explained in Section 4.3.1. The second was to add the ability to compute a set of features, manually defined, from the code's AST and raw source, that differentiate regular code from obfuscation practices, as described in Section 4.3.2

Not all files in the dataset were parsed correctly due to a variety of issues. Some obfuscators, such as *JavaScript2img*, originate invalid code, which is unparseable. Additionally, the transformations applied by *node-obf* and *jsfuck*, specially in larger files, originate very complex and nested AST's which the parser is incapable of handling. For this reason, we only parse files smaller than 2.7 KB for *node-obf*, and smaller than 20 KB for *jsfuck*. After parsing, the dataset has features and paths for 273,121 JavaScript files, distributed as displayed in Table 4.1.

	Configuration	Number of files
Regular	-	34,981
UglifyJS	-	35,029
Google Closure Compiler	-	30,093
Jscrambler	*Jscrambler-config1*	8,744
	Jscrambler-2	8,643
	Jscrambler-3	8,734
	Jscrambler-4	8,710
javascript-obfuscator	*javascript-obfuscator-config1*	8,592
	javascript-obfuscator-2	8,621
	javascript-obfuscator-3	8,884
	javascript-obfuscator-4	8,842
defendjs	*defendjs-config1*	14,369
	defendjs-2	14,401
js-obfuscator	*js-obfuscator-config1*	10,013
	js-obfuscator-2	10,130
	js-obfuscator-3	10,238
JSObfu	*JSObfu-config1*	21,464
JavaScript2img	*JavaScript2img-config1*	6,149
DaftLogic	*DaftLogic-config1*	10,687
jsfuck	*jsfuck-config1*	2,548
node-obf	*node-obf-config1*	3,249

Table 4.1: Distribution of files per type of code - Regular and tool used - and configuration used, after parsing the code. All obfuscators have a common configuration, *config1*, and some have other configurations which are tool-dependent, and are represented by the tool's name and a number.

4.3.1 Path-contexts

For each program, the path-contexts for the first 1,000 terminals are computed (or less if there are less than 1,000 terminals in the program). This is accomplished by going through all terminals and computing paths that connect them with other terminals. These paths do not store the directions that connect the nodes (up or down the AST). The paths have a minimum and maximum AST depth (represented by the path's length) of five and 100, respectively [13], including the terminal values. This means that paths that connect two terminals that have a depth larger than 100 are not computed.

In the original version of the parser, the paths start and end with the value of two terminal nodes, containing the sequence of nodes that connected them. For example, one of the paths extracted from the expression *x = 10;* was:

$$x|Identifier|AssignmentExpression|Literal|10 \tag{4.1}$$

[13] These values are configurable if desired.

where *x* is the value of the start node and *10* is the value of the end node, that are connected by an *AssignmentExpression* node (as the value ten is assigned to the identifier *x*). However, our focus is on the code structure and not necessarily on the values of the terminals, as obfuscators often apply some randomization while renaming variables or assigning values.

To compute more generic paths, we use a label representing the terminal nodes' values for numbers, strings, and identifiers, instead of using the actual value. This label preserves the relevant information and discards unnecessary details. However, if the value of the terminal is a common obfuscation keyword, it remains unaltered (see Section 4.3.2.3 for more details on these types of keywords).

A number is represented by the label *Number* followed by *Encoded* if the number is encoded. Similarly, a string is represented by *String* followed by *Encoded* if it contains any encoded characters. Finally, identifiers are represented by the label *Identifier* followed by two names: the first indicating if it is encoded, *Encoded*, human-readable, *Readable*, or unreadable, *Unreadable*; and the second relative to its size, *Small* for identifiers with less than three characters, *Medium* for identifiers with more than three characters and less than 15 characters, and *Large* for identifiers with more than 15 characters. For more details on our definition of encoded strings, number and identifiers, see Section 4.3.2.5. Additionally, our definition of unreadable is based on [52] and considers an identifier as unreadable if its size is greater than three characters and has a percentage of vowels lower than 5% or greater than 60%; or has a percentage of letters lower than 70%, or has more than two consecutive equal characters.

For the expression in the previous example, the altered parser computes the path:

$$IdentifierReadableSmall|Identifier|AssignmentExpression|Literal|Number \tag{4.2}$$

This new nomenclature allows similar expressions, such as *a = 1;* to be represented by the same paths. In the implementation, all the names in the paths are shortened to reduce the space required to store them.

A possible limitation of the parser is that it only stores unique paths. This means that the information of the number of times a particular path appears in the code is not taken into account. However, this reduces the space in memory required to store the paths, which allows the overall scaling of the dataset used. Additionally, as only a subset of these paths will be used in a next step, storing only the unique paths implies that this subset will be diverse and representative of different paths in the code, discarding the possibility of it containing duplicate (possibly irrelevant) information.

4.3.2 Features

To compute a feature vector, we traverse the code's AST, storing and engineering relevant data. Some features are also computed from the raw source code.

We focus on features that can represent different and standard obfuscation practices. Our features are based on previous work in the area [32, 52, 65] and manual inspection of obfuscated

code. As the detector should also identify the obfuscator used in the code, some features are tool-related. The features are mainly based on relative frequencies of specific values and statistical metrics, such as means and standard deviations.

The extracted features can be roughly divided into six categories: string related, identifier related, keyword related, node related, encoded related, and file related. We extract in total 101 different features, some of which do not fit in any of these categories. Table A.1 shows the set of features that were extracted from the code, their description, and the corresponding category they belong to.

4.3.2.1 String related

String related features are extracted from strings in the code. They aim at detecting patterns in the strings related to their length, word size, entropy, and 1-gram (byte occurrence frequency). These features are extracted from the raw version of the string, which does not decode encoded characters. We compute these features based on [32]. The word size is the average of number of characters in all words in the strings used in the code. We consider a word any sequence of alphanumeric (or the underscore character) characters inside a string. The entropy is used as a measure of the distribution of bytes in the string and the 1-gram to check the frequency of specific characters.

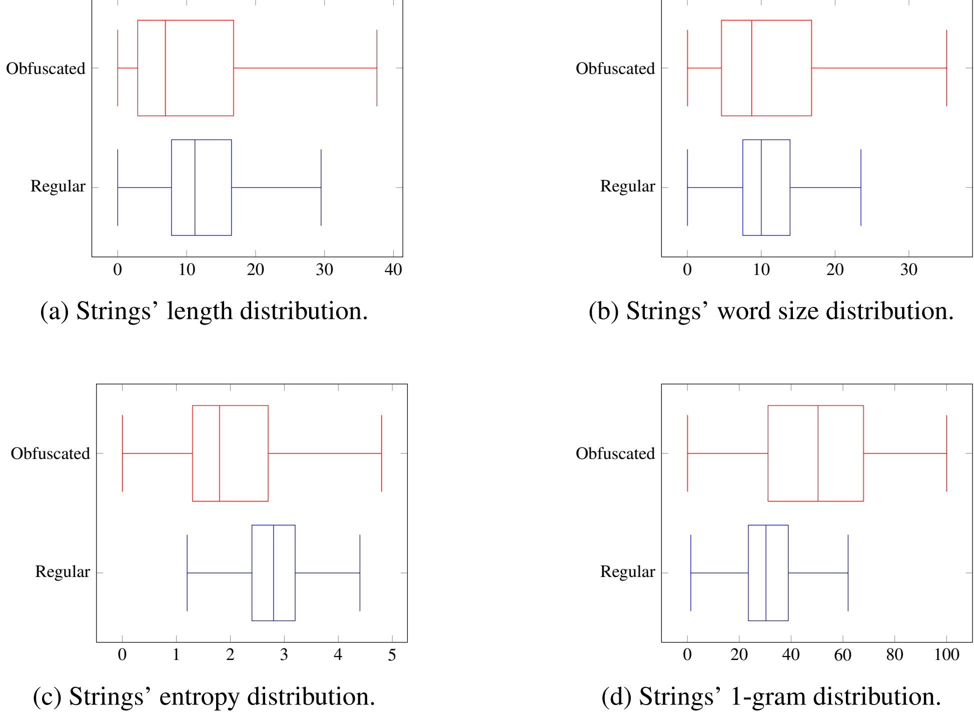
The entropy, E, of a string is calculated as follows [32]:

$$E = -\sum_{n=1}^{N} (\frac{b_n}{T} \log \frac{b_n}{T}), T = \sum_{n=1}^{N} b_n \tag{4.3}$$

where b_n is the count of each character and T is the total number of characters in the string. The 1-gram of a string is computed from the frequency of special characters, such as %, in the string. We consider special characters the ones with ASCII codes between *0x21-0x2F*, *0x3A-0x40*, *0x5B-0x5F*, or *0x7B-0x7E*.

Figure 4.6 depicts the distributions of values for the average string length, word size, entropy, and 1-gram, for strings in regular and obfuscated code. We remove the outliers from the plots to have a proper visualization of the values, as some outliers exceed significantly the values portrayed. We consider an outlier a value that is located outside the *wiskers* of the box plot.

For all the mentioned features, the distribution of values differs in regular and obfuscated code. In general, these values have greater diversity in obfuscated code, likely due the use of different obfuscators. The average string length in regular code is slightly greater than the one in obfuscated code. However, obfuscated code displays a larger range of values for the string length. On average, obfuscated code has a greater word size than regular code. The entropy values in regular code tend to by higher than in obfuscated code, however it should be noted that some files with an entropy lower than 1.2 were filtered in Section 4.2.3. Regarding the 1-gram, in obfuscated code, this value ranges from 0% to 100%. However, in regular code, this value generally does not exceed 62% - meaning that there are no strings composed solely by special characters.



(a) Strings' length distribution.

(b) Strings' word size distribution.

(c) Strings' entropy distribution.

(d) Strings' 1-gram distribution.

Figure 4.6: String related features' distribution in regular and obfuscated code.

4.3.2.2 Identifier related

Identifier related features are extracted from the identifiers used in the code. We use four categories of identifiers: declared variables, declared functions, function parameters, and other identifiers used in the code. The features are extracted independently for these categories and aim at detecting unusual identifier names based on: the length; and the frequency of letters, uppercase letters, vowels and numbers in the identifier's name. These features attempt to capture differences between the identifiers written by humans and the ones automatically generated by the tools.

Table 4.2 shows the average values for these metrics across all regular and obfuscated files. For all the categories considered, the average frequencies of numbers and uppercase letters are significantly higher in obfuscated code than in regular code. Conversely, identifiers in regular code tend to have a higher frequency of vowels than the ones in obfuscated code. The frequency of letters varies according to the type of identifier considered, however, it tends to be higher in regular code comparing to obfuscated code. The majority of the obfuscators that we use renames the variables in the code by replacing their names with ones with no obvious meaning, which may explain these differences.

	Metric	Regular	Obfuscated
Declared Variables	Length (number of characters)	5.481	3.762
	Frequency of letters (%)	80.271	58.979
	Frequency of uppercase letters (%)	5.348	18.847
	Frequency of vowels (%)	28.692	12.713
	Frequency of numbers (%)	0.441	23.892
Declared Functions	Length (number of characters)	5.104	3.322
	Frequency of letters (%)	45.208	52.454
	Frequency of uppercase letters (%)	4.528	17.256
	Frequency of vowels (%)	16.018	13.778
	Frequency of numbers (%)	0.193	16.293
Parameters	Length (number of characters)	1.751	4.513
	Frequency of letters (%)	71.714	58.098
	Frequency of uppercase letters (%)	0.765	16.446
	Frequency of vowels (%)	34.567	17.909
	Frequency of numbers (%)	0.047	25.932
Others	Length (number of characters)	6.501	4.939
	Frequency of letters (%)	95.628	67.329
	Frequency of uppercase letters (%)	6.464	17.972
	Frequency of vowels (%)	36.615	16.594
	Frequency of numbers (%)	0.314	24.924

Table 4.2: Average value for the length, frequency of letters, frequency of uppercase letters, frequency of words, and frequency of numbers, per type of identifier, across all regular and obfuscated files.

4.3.2.3 Keyword related

Keyword related features are based on the frequency of specific keywords in the code. These keywords appear often in obfuscated code, with different frequencies depending on the tool applied. They were manually defined based on previous work and assessment of obfuscated files. For each JavaScript file we compute the frequencies for each keyword by dividing the number of times the keyword appears in the code by the total number of terminal nodes in the code's AST.

Table 4.3 shows the top 10 keywords whose frequency is greater in obfuscated code than in regular code (for the complete table, see Table A.2). The presented values are the average of the frequencies for each keyword across all files. As shown, keywords such as +, *String*, and *fromCharCode* are significantly more frequent in obfuscated code than in regular code. These three (among others) are commonly used to convert static data into procedures, either by using concatenation of values, such as strings and numbers, or converting strings into results of calls to other methods.

Keyword	Average Frequency (%)	
	Regular Code	Obfuscated Code
+	0.951	3.612
String	0.036	0.570
fromCharCode	0.004	0.488
arguments	0.093	0.415
parseInt	0.025	0.157
apply	0.047	0.133
RegExp	0.023	0.102
eval	0.008	0.071
%	0.015	0.056
Array	0.074	0.113

Table 4.3: Average frequency of specific JavaScript keywords in regular and obfuscated code (top 10).

4.3.2.4 Node related

Node related features are based on the frequency of specific nodes of the code's AST. These features aim at representing patterns in the usage of certain expressions in the code. Table 4.4 depicts the frequency of specific nodes in regular and obfuscated code. We group all nodes related with control flow statements together, such as *WhileStatement*, *SwitchStatement*, among others, to capture obfuscation that applies control flow flattening techniques. These frequencies are computed by dividing the number of times a certain node (or group of nodes) appears in the code by the total number of nodes. As shown, some nodes (or groups of nodes) are more common in obfuscated code than in regular code, for example *MemberExpression* nodes. The presence of these nodes in high frequencies can be a strong indicator that the code is obfuscated. Other nodes, such as *FunctionExpression* and *FunctionDeclaration*, are more common in regular code than obfuscated code. These nodes represent the use of function expressions and function declarations in the code. The fact that in obfuscated code the frequency of these nodes is lower than in regular code may be explained by the use of function inlining techniques that replace a call to a function by its code.

Node	Average Frequency (%)	
	Regular Code	Obfuscated Code
Control Flow Nodes	1.495	3.389
MemberExpression	10.584	12.464
AssignmentExpression	2.749	3.886
FunctionExpression	1.595	1.445
FunctionDeclaration	0.419	0.230

Table 4.4: Average frequency of specific nodes (and groups of nodes) in regular and obfuscated code.

From the *MemberExpression* node, we compute the frequency of array and property accesses in the code. Array accesses are more common in obfuscated code than regular code, constituting around 75% of all *MemberExpression* nodes. Conversely, property accesses are more common in regular code, constituting around 86% of all *MemberExpression* nodes.

Finally, we also compute the frequency of terminals that have string or numerical values. To do so, we divide the number of terminal nodes with string or numerical values by the total number of terminal nodes in the code. Regular code has, on average, a higher frequency of strings than obfuscated code. Conversely, the frequency of numbers in obfuscated code is around five times higher than in regular code.

4.3.2.5 Encoded related

Encoded related features are extracted from strings, numbers, and identifiers and aim at detecting standard encoding obfuscation techniques. These features are computed by applying specific regexes to the raw version of the value of these terminals.

A string is considered encoded if at least one of its characters is encoded - written in its ASCII, octal or hexadecimal representation. For example *"\u0061\u0062\u0063"*. Similarly, a number is encoded if it is not in base 10, for example *0x1b63*. Finally, an identifier is considered encoded if it begins with an "_" followed by an encoded number, for example *_0xeb15*. This nomenclature is commonly applied by some obfuscators, such as *javascript-obfuscator* and *js-obfuscator*, to rename identifiers.

Figure 4.7 depicts the distribution of encoded strings, numbers, and identifiers - declared variables, declared functions, function parameters, and other identifiers used - in regular and obfuscated code. As shown, for all types of terminals considered, the average frequency of encoding is significantly higher in obfuscated code compared to regular code. As expected, in regular code, there are no encoded identifiers. In general, these are named to be human-readable or, in case the code is minified, short to occupy fewer memory.

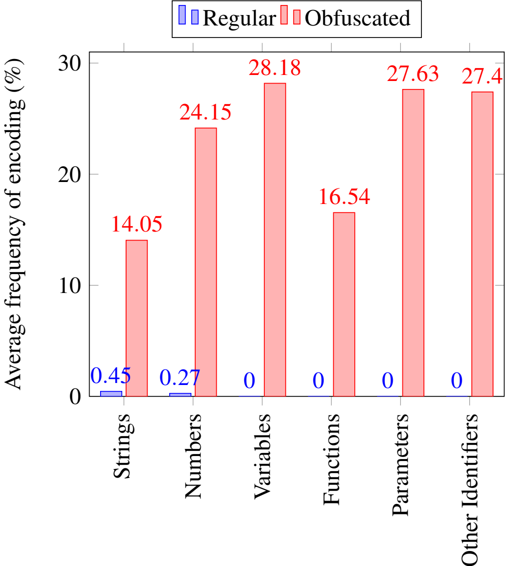


Figure 4.7: Average frequency of encoding per terminal type, for regular and obfuscated code.

4.3.2.6 File related

File related features are extracted directly from the source code, namely frequency of indentation characters and average number of characters per line.

The frequency of indentation characters is computed by dividing the number of indentation characters in the file by its total number of characters. In regular code, the average value for this feature is 7.48%, which is not very different from the one in obfuscated code, which is 8.79%. However, the obfuscated code presents fewer diversity of values, as the majority of files has less than 10% of indentation characters. This is not the case for regular code, where this value ranges from 0% to 38%, as shown in Figure 4.8.

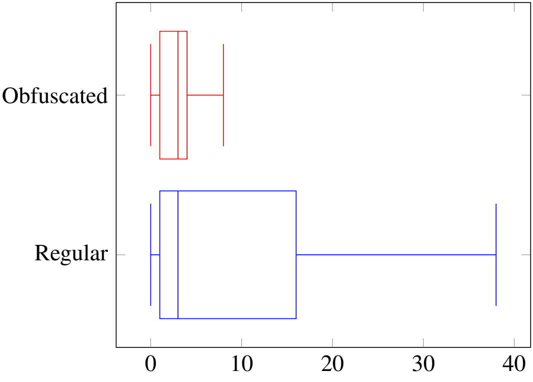


Figure 4.8: Distribution of the frequency of indentation characters in regular and obfuscated code.

The average number of characters per line in obfuscated files (25,895 characters) is around ten

times the value in regular files (2,539 characters [14]). This value is computed by dividing the total number of characters by the total number of lines in the file.

4.4 Classifiers

The features and paths computed are used to solve two classification tasks. The first task is a binary problem, to distinguish between regular and obfuscated code. The second task is to identify the obfuscator that was used to transform the code. This task is addressed as a multiclass problem, where there are as many classes as the number of obfuscators considered plus one, which represents the regular code class.

Two different models were developed to solve these tasks: *Code2BagOfPaths*, and *Code2FeatureVector*. The models are implemented in Python and with the *scikit-learn* [15] library, as it is well furnished with efficient tools for solving data analysis and machine learning tasks.

The first model, *Code2BagOfPaths*, is a *Bag Of Words* [16] implementation. Each file is represented by a set of path-contexts, and each path is considered a *word*. We create a vocabulary based on all paths in the training set. We ignore paths that appear in more than 50% of the files in the training set, as they are considered common JavaScript paths, or common obfuscated paths, and are not useful to distinguish obfuscated code from regular code, or to identify specific tools. Then we transform each code sample into a vector with the same length as the vocabulary, where each element is the *term-frequency times inverse document-frequency*, *tf-idf* [17], value of a specific path. The computed vectors are then used as input for a Multinomial Naive Bayes classifier, which was chosen due to its popularity in text classification [64]. In the remaining of this work, we also refer to this classifier as context-based model.

The *Code2FeatureVector* model is a Random Forest classifier that receives as input the features manually defined and computed from the code. In total it uses 101 features. We use the parameters provided by the default implementation of the Random Forest classifier in *scikit-learn* [18]. This includes the use of 100 trees (estimators) to make the estimation, and an unlimited maximum tree depth. We chose this classifier based on the results of preliminary experiments conducted comparing standard classification algorithms. Additionally, Random Forest models have easy interpretability, as they allow to compute the importance given to each feature, which is valuable to understand the obtained results. In the remaining of this work, we also refer to this classifier as feature-based model.

[14] We include both regular and minified code, therefore the number of characters per line in regular code is greater than standard programming practices.

[15] https://scikit-learn.org/stable/

[16] https://machinelearningmastery.com/gentle-introduction-bag-words-model/

[17] Value obtained by multiplying the term frequency by the inverted document frequency of a path [59].

[18] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

4.5 Summary

The implementation of the detector has three main steps: dataset creation, data parsing, and classification. In the first step we collect a set of JavaScript files from different websites, *GitHub* repositories and *NPM* packages. These files are preprocessed to filter minified, duplicated, unparseable, and obfuscated files. Copies of the remaining files are then transformed with two minifiers and nine obfuscators to create the final dataset. Due to time restrictions, the amount of samples transformed by some tools, mainly those that visit websites, is limited.

In the second step, each file is parsed to compute a set of path-contexts and manually defined features. A path-context is a path of the code's AST that joins any two terminal nodes. Additionally, we extract six categories of manually defined features - string related, identifier related, keyword related, node related, encoded related, and file related - that aim at distinguishing regular and obfuscated code. After parsing, as some obfuscators generate code that is unparseable due to its complexity or syntax errors, the dataset is slightly imbalanced and there are fewer samples for some obfuscators. Additionally, as size restrictions were imposed to some tools, the range of file sizes for those tools is not very broad.

Finally, we use the paths and features as input for two different classifiers. The first classifier is an implementation of *Bag Of Words* where each file is represented by a set of path-contexts, and each path is considered a *word*. A vocabulary of paths is then created from the training data. For each code sample, we compute a vector of its *term-frequency times inverse document-frequency*, *tf-idf*, in regards to the paths in the vocabulary. These vectors serve as input for a Multinomial Naive Bayes classifier. The second classifier is a Random Forest that receives as input the features defined based on standard obfuscation practices.

Chapter 5

Context Versus No Context

In this chapter we compare two approaches for detecting obfuscation and identifying obfuscators. The first approach, *Code2BagOfPaths*, uses contextual information in the form of paths of the AST, and the second, *Code2FeatureVector*, is based on defined features. We conduct a set of experiments to understand which approach is more reliable and efficient in solving the tasks addressed. In this chapter we try to answer two questions:

- **How to select the paths?** The use of randomly selected paths to represent a snippet of code for the *Code2BagOfPaths* model, may or not improve the classification when compared with the use of filtered paths.
- **Is there any benefit in using context-based features?** The use of context-based features may or not be beneficial when detecting obfuscated code.

5.1 Experimental Setup

In these experiments we use the classifiers described in Section 4.4.

We divide the dataset into two different sets: training and testing. For this, we use 70% of the files for training and 30% for testing. However, this separation is not a trivial task since there is no guarantee that the transformed versions of each file are different from their original file and between tools - minifiers often output the same minified code; some obfuscators fail to transform portions of the code, such as dictionary declarations. This means that there is the possibility that similar files appear in both the training and testing sets, which could lead to unrealistic results. To divide the dataset, we impose that:

- Each set has both original and transformed files.
- There are similar percentages of original files in the training and testing sets.
- There are similar percentages of transformed, minified or obfuscated, files in the training and testing sets.

- A transformed file, minified or obfuscated, must be in the same set as its original file.

We assign the files into the training and testing sets with the following methodology:

1. Select the tool with fewer [1] transformed files in the dataset, whose files have not been added to the training and testing sets.

2. If the sets are empty, assign the files transformed with the tool selected in step *1* according to the desired percentage, and add them to the corresponding sets - 70% for training and 30% for testing. Go to step *5*. Else go to step *3*.

3. If there are files in the sets, add the versions of the files transformed by the tool selected in step *1* that have the same original file as the ones in the sets (if they exist) to the corresponding sets.

4. Assign the remaining files transformed with the tool selected in step *1* according to the desired percentage and add them to the corresponding sets - 70% for training and 30% for testing.

5. If there is at least one tool whose files have not been added to the sets in the dataset, go to step *1*. Else go to step *6*.

6. Add the original version of the files to the corresponding sets.

This methodology guarantees that the testing set does not contain files that were obfuscated or minified from any original file that is in the training set, also ensuring a stratified division, where all classes are represented in the same proportion in both sets.

To assess the binary models, we compute their precision, recall, F1-score, and accuracy. Similarly, to assess the multiclass models, we compute the weighted average of the first three of these scores, and the accuracy for all classes. When further evaluation of the model is required - to assess if the model is overfitting -, we apply cross-validation with ten k-folds using the training data. According to the methodology above, the training data is first shuffled and then divided into ten folds. This type of splitting does not guarantee that all folds will be different, but it is unlikely that two folds are the same due to the large size of the dataset. To assess the results of the cross-validation, we compute the average of the results obtained from all folds.

All experiments were performed on a workstation with a Ryzen 7 3700x processor (3.6 GHz) with eight cores, 32 GB of RAM, and an NVIDIA GeForce GTX 1650 Super GPU.

[1] We select the tool that has the fewer files transformed in the dataset to ensure a stratified division of the code between the training and testing sets - using a random tool could result in an unbalanced distribution of the files transformed with a tool with fewer files, if their corresponding versions were not appropriately distributed between the sets.

5.2 How to select the paths?

The path choice is an essential factor while classifying the code using the *Code2BagOfPaths* model, as it determines how the code samples are represented. For this choice, two different components should be considered: the type of paths and the number of paths.

Regarding the first component, the goal is to understand if selecting the path-contexts randomly improves the classification versus filtering the paths. This is only relevant when the number of paths used is limited instead of using all the paths. Although, at first sight, filtering the paths based on the frequency of nodes commonly used in obfuscated code would seem a better approach since it focuses on standard obfuscation practices, it is likely that some of these paths are also present in regular code. Thus, by solely using these paths, the representation of the script is limited and could lead to more false positives. Our hypothesis is that randomly selecting the paths allows a more diverse representation of the entire code without focusing on specific nodes and keywords, which leads to better results.

In this experiment we consider two types of paths: randomly selected paths and filtered paths, where the paths are filtered regarding the nodes they traverse. To filter the paths, we first rank them by computing a weighted average of the frequency of certain context types. Table 5.1 shows the weights and nodes associated with each context type. The remaining nodes are given a weight of zero.

Context Type	AST nodes	Weight
Control Flow Structures	*WhileStatement*, *SwitchStatement*, *SwitchCase*, *BreakStatement*, *ContinueStatement*, *ReturnStatement*, *DoWhileStatement*, *ForStatement*, *ForInStatement*, *ForOfStatement*, *ThrowStatement*, *TryStatement*, *CatchClause*	0.3
Access Operations	*MemberExpression*, *Property*, *ArrayExpression*, *ObjectExpression*	0.2
Function Declarations and Invocations	*FunctionExpression*, *ArrowFunctionExpression*, *FunctionDeclaration*, *CallExpression*	0.2
Special Keywords	Terminal nodes with values associated with special keywords.	0.2
Logical Operations	*BinaryExpression*, *LogicalExpression*, *UnaryExpression*	0.1

Table 5.1: Context types and their correspondent JavaScript AST nodes and assigned weights.

Regarding the second component for the path choice, namely the choice of the number of paths that should be used to represent script, our goal is to understand how this choice impacts the classification. We use 25, 50, 75, and 100 paths. As some files end up having hundreds (some thousands) of paths, and as loading/transforming that amount of data would be very time and memory consuming, we do not train the model with all the extracted paths

To make the choice of paths, regarding the type of paths selected - random or filtered - and the number of paths to use, we train and test both the binary and multiclass classifiers with the training and testing sets described in Section 5.1, and all possible combinations of types of paths and number of paths mentioned - with a total of eight combinations per task. In the remaining of this work, we refer to *random-based* models as the models that use randomly selected paths, and *filtered-based* models as the models that use filtered paths.

The results for both the binary and multiclass tasks are presented in Table 5.2 and Table 5.3, respectively. The results show that the model performs better when using 100 paths, in both tasks,

when considering precision, recall and F1-score. For all configurations, regarding the types of paths and numbers of paths used, all the metrics evaluated are above 99%, which indicates a high coverage independently of the configuration used. With the increase in the number of paths, more features are computed, and the model may have more information to make the classification. Additionally, if fewer paths are used, the probability of a new, unseen path surfacing in the testing set may be higher, which may explain the results obtained.

Type of Paths	Number of Paths	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)
Random	25	99.90	99.91	99.91	99.88
	50	99.91	99.94	99.93	99.91
	75	99.90	99.96	99.93	99.92
	100	99.91	99.96	**99.93**	99.92
Filtered	25	99.00	99.77	99.38	99.21
	50	99.54	99.82	99.68	99.60
	75	99.72	99.83	99.77	99.71
	100	99.83	99.83	99.83	99.79

Table 5.2: Results obtained by the binary model, by using random or selected paths with a variety of number of paths.

Type of Paths	Number of Paths	Weighted Average Precision (%)	Weighted Average Recall (%)	Weighted Average F1-score (%)	Accuracy (%)
Random	25	99.80	99.80	99.80	99.80
	50	99.90	99.90	99.90	99.90
	75	99.90	99.90	99.90	99.90
	100	99.92	99.92	**99.92**	99.92
Filtered	25	99.29	99.28	99.27	99.28
	50	99.54	99.54	99.54	99.54
	75	99.66	99.66	99.66	99.66
	100	99.73	99.73	99.73	99.73

Table 5.3: Results obtained by the multiclass model, by using random or selected paths with a variety of number of paths.

For all the sets of paths tested, the model that uses random paths outperforms the one that uses filtered paths, resulting in higher precision and recall values, validating our hypothesis. To better understand these results, we compare the models with higher F1-score for both approaches (with 100 paths) for the binary task in the following paragraphs.

The random-based model classifies 49 regular files as obfuscated. These files comprise both regular code and minified code, some of which are the minified versions of the regular files misclassified. The majority of these files uses keywords commonly associated with obfuscation, such as *Array*, has encoded strings, or long/unreadable identifiers' names. Additionally, the majority of them are also smaller files, which means that the model uses all the paths computed, including the ones reminiscent of obfuscation, which may explain the detector's behavior. The filtered-based model misclassifies 89 regular files as obfuscated. The majority of these files uses keywords commonly associated with obfuscation, such as *fromCharCode*, *RegExp*, among others. Since the filtering gives more weight to these types of expressions, the model is likely to be using these

paths to make the classification, associating them with paths of obfuscated code, thus misclassifying them as obfuscated. Additionally, some of the files have paths similar to obfuscated files, for example using large *switch* statements inside *while* loops, which is a common practice in code obfuscated by *defendjs*, or have long identifiers or encoded strings. Finally, and for both models - filtered-based and random-based -, if they receive a set of paths never seen before, they classify the code as obfuscated [2], which also may explain the false positive values.

Regarding the false negatives, the random-based model classifies 20 obfuscated files as regular, and the filtered-based model classifies incorrectly 86 obfuscated files. The misclassification of obfuscated files seems to occur when the paths used are associated with regular code instead of obfuscated code. For example, if part of the code is not obfuscated (some tools do not obfuscate dictionaries for example) or when the code is obfuscated differently from the obfuscated code in the training set - even if the paths are similar, if they are not equal to those in the training set, the detector does not associate them with obfuscation. When the paths are selected randomly, there is a higher chance of avoiding this scenario, as the used paths are retrieved from random code blocks, covering a variety of statements and transformations. However, if the paths are filtered, and therefore focus on specific code blocks, the detection may be compromised when these blocks are transformed differently or are not transformed at all.

These conclusions can be generalized to the multiclass models, as the files are misclassified due to the same reasons. However, both models classify fewer regular files as obfuscated and more obfuscated files as regular. The lower values of false positives may indicate that the models are able to distinguish regular code from code transformed by individual obfuscators, even if the regular code contains common obfuscation practices (keywords, paths, among others). In general, the misclassified regular files are smaller, reducing the probability of using known paths, which may mislead the detector. The higher false negatives values may indicate that even if the used paths are common in obfuscated code, the models are not associating them to any specific obfuscator.

In the end, the results corroborate with our initial hypothesis that randomly selecting the paths seems to allow a better representation of the script, resulting in both lower false positives and lower false negatives. Filtering the paths seems to induce the model in error if the regular files contain specific keywords or paths. Filtering the paths also hinders the obfuscation detection in cases where the paths selected are common in regular code. The best results are obtained by using 100 random paths in both tasks addressed. Notwithstanding, the results obtained do not invalidate the use filtered paths in general, they invalidate the use of the defined methodology to filter the paths.

5.3 Is there any benefit in using context-based features?

When considering the task of obfuscation detection by statically analyzing the source code, two main approaches can be followed: extracting context-based features; and computing features based on standard obfuscation techniques, with no context associated with them.

[2] This was tested by inputting unknown paths to the classifier and predicting the outcome.

Our goal is to determine if the use of the context-based features improves the distinction of obfuscated and regular code compared to features extracted with no context associated. Our hypothesis is that, by computing features that focus on differences between regular and obfuscated code, the classifier receives all relevant information without additional noise. Although the context-based features allow the identification of patterns in the paths generated by individual tools, it is expected that the no-context features also capture these patterns.

To compare the two detectors - *Code2BagOfPaths* and *Code2FeatureVector* -, we first apply cross-validation to the training data, as described in Section 5.1. Then we test them with the testing set. For the *Code2BagOfPaths* detector, we use 100 randomly selected paths for both binary and multiclass models, based on the results obtained in Section 5.2.

Table 5.4 displays the results obtained from cross-validating and testing the binary classifier for the *Code2BagOfPaths* and *Code2FeatureVector* models. For both models, the results obtained in testing are consistent with the ones obtained by applying cross-validation to the training data. This is a strong indicator of the models' robustness and shows that they are unlikely to be overfitted to the testing data. Both models seem to be able to detect obfuscation with high precision and recall.

	Model	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)
Train	*Code2BagOfPaths*	99.887 $\pm$0.024	99.958 $\pm$0.011	99.923 $\pm$0.015	99.903 $\pm$0.017
	Code2FeatureVector	99.995 $\pm$0.005	99.979 $\pm$0.012	99.986 $\pm$0.005	99.982 $\pm$0.008
Test	*Code2BagOfPaths*	99.91	99.96	99.93	99.92
	Code2FeatureVector	100.00	99.98	**99.99**	99.99

Table 5.4: Cross-validation and testing results for the binary version of the *Code2FeatureVector* and *Code2BagOfPaths* models.

To better explore these results, we display the confusion matrices obtained for both models, in Table 5.5 and Table 5.6. As both the regular and obfuscated classes have subclasses, such as the tool and configuration used, we divide these matrices accordingly.

The *Code2BagOfPaths* model classifies 49 regular files as obfuscated and 20 obfuscated files as regular. In general, the regular files are classified incorrectly due to the presence of certain keywords and specific paths that are reminiscent of obfuscation practices. The obfuscated files classified incorrectly were transformed by one of two tools: *javascript-obfuscator*, and *js-obfuscator*. The incorrect classification of obfuscated files seems to occur when the selected paths are associated with regular code instead of obfuscated code, due to different transformations being applied in those files, or portions of code being untransformed (see Section 5.2).

	Predicted Label	
	Regular	Obfuscated
Regular	10,488	19
UglifyJS	10,508	16
Google Closure Compiler	8,987	14
Total Regular	**29,983**	**49**
Jscrambler-config1	0	2,584
Jscrambler-2	0	2,623
Jscrambler-3	0	2,556
Jscrambler-4	0	2,695
javascript-obfuscator-config1	5	2,568
javascript-obfuscator-2	0	2,600
javascript-obfuscator-3	0	2,646
javascript-obfuscator-4	0	2,671
defendjs-config1	0	4,221
defendjs-2	0	4,432
js-obfuscator-config1	5	3,008
js-obfuscator-2	4	2,997
js-obfuscator-3	6	3,101
JSObfu-config1	0	6,470
JavaScript2img-config1	0	1,849
DaftLogic-config1	0	3,195
jsfuck-config1	0	765
node-obf-config1	0	976
Total Obfuscated	**20**	**51,957**

Table 5.5: Confusion matrix for the binary version of the *Code2BagOfPaths* model, extended by tools and configurations. In bold are the total values per class.

Regarding the *Code2FeatureVector* model, it misclassifies 11 files, one as obfuscated and ten as regular. The regular file misclassified was minified with *UglifyJS*. After manually inspecting this file, we observed that the code on this file uses a set of keywords, such as *slice* and *String*, and operations, such as concatenations, commonly used in obfuscated code, which combined with the minification could explain the detector's behavior. Regarding the misclassified obfuscated files, they were all transformed by the same tool, *JSObfu*, and nine out of the ten are smaller than 488 bytes. As explained in Section 4.2.4, this obfuscator tends to generate simpler code when used with smaller files, which is a possible explanation for the incorrect classification of these files. The 10th file, that is significantly larger at around 147 KB, contains a significant portion of regular code, as the tool did not obfuscate a dictionary declaration.

	Predicted Label	
	Regular	Obfuscated
Regular	10,507	0
UglifyJS	10,523	1
Google Closure Compiler	9,001	0
Total Regular	**30,031**	**1**
Jscrambler-config1	0	2,584
Jscrambler-2	0	2,623
Jscrambler-3	0	2,556
Jscrambler-4	0	2,695
javascript-obfuscator-config1	0	2,573
javascript-obfuscator-2	0	2,600
javascript-obfuscator-3	0	2,646
javascript-obfuscator-4	0	2,671
defendjs-config1	0	4,221
defendjs-2	0	4,432
js-obfuscator-config1	0	3,013
js-obfuscator-2	0	3,001
js-obfuscator-3	0	3,107
JSObfu-config1	10	6,460
JavaScript2img-config1	0	1,849
DaftLogic-config1	0	3,195
jsfuck-config1	0	765
node-obf-config1	0	976
Total Obfuscated	**10**	**51,967**

Table 5.6: Confusion matrix for the binary version of the *Code2FeatureVector* model, extended by tools and configurations. In bold are the total values per class.

Table 5.7 presents the results obtained by the models regarding the multiclass task of identifying the tool that obfuscated the code. As in the binary task, the results obtained in the testing for both models are consistent with those obtained by cross-validating the training data. As expected, the *Code2FeatureVector* outperforms the *Code2BagOfPaths* model.

	Model	Weighted Average Precision (%)	Weighted Average Recall (%)	Weighted Average F1-score (%)	Accuracy (%)
Train	*Code2BagOfPaths*	99.917 $\pm$0.011	99.917 $\pm$0.011	99.917 $\pm$0.011	99.917 $\pm$0.011
	Code2FeatureVector	99.984 $\pm$0.005	99.984 $\pm$0.005	99.984 $\pm$0.005	99.984 $\pm$0.005
Test	*Code2BagOfPaths*	99.92	99.92	99.92	99.92
	Code2FeatureVector	99.99	99.99	**99.99**	99.99

Table 5.7: Cross-validation and testing results for the multiclass version of the *Code2FeatureVector* and *Code2BagOfPaths* models.

Table 5.8 represents the confusion matrix obtained by the *Code2BagOfPaths*, extended to represent information regarding the configurations of each tool. This model classifies incorrectly 11 regular files and 55 obfuscated files. The false positives are classified as obfuscated by one of three tools (*Jscrambler*, *defendjs* or *node-obf*), and either have keywords or operations associated with obfuscation, or are too small, which may explain the detector's behavior. The misclassified

obfuscated files were transformed either by *javascript-obfuscator*, *js-obfuscator* or *JSObfu*. As in the binary task, the obfuscated files may be classified as regular due to the use of paths associated with regular code instead of obfuscated code. However, there are two files transformed with *js-obfuscator* that the model classifies as being transformed by *javascript-obfuscator*. These tools apply similar techniques, which could result in the same paths. For example, both obfuscators start by initialising a large array with the strings used in the code, which are then accessed when required. The two misclassified files are also significantly larger than the norm for the code transformed by *js-obfuscator*. This means that there are more paths to select, which may decrease the probability of using the paths the model associates with this tool.

	Predicted Label									
	Regular	Jscrambler	javascript-obfuscator	defendjs	js-obfuscator	JSObfu	JavaScript2img	DaftLogic	jsfuck	node-obf
Regular	10,502	2	0	2	0	0	0	0	0	1
UglifyJS	10,520	2	0	2	0	0	0	0	0	0
Google Closure Compiler	8,999	2	0	0	0	0	0	0	0	0
Total Regular	**30,021**	**6**	**0**	**4**	**0**	**0**	**0**	**0**	**0**	**1**
Jscrambler-config1	0	2,584	0	0	0	0	0	0	0	0
Jscrambler-2	0	2,623	0	0	0	0	0	0	0	0
Jscrambler-3	0	2,556	0	0	0	0	0	0	0	0
Jscrambler-4	0	2,695	0	0	0	0	0	0	0	0
Total Jscrambler	**0**	**10,458**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
javascript-obfuscator-config1	17	0	2,556	0	0	0	0	0	0	0
javascript-obfuscator-2	0	0	2,600	0	0	0	0	0	0	0
javascript-obfuscator-3	0	0	2,646	0	0	0	0	0	0	0
javascript-obfuscator-4	0	0	2,671	0	0	0	0	0	0	0
Total javascript-obfuscator	**17**	**0**	**10,473**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
defendjs-config1	0	0	0	4,221	0	0	0	0	0	0
defendjs-2	0	0	0	4,432	0	0	0	0	0	0
Total defendjs	**0**	**0**	**0**	**8,653**	**0**	**0**	**0**	**0**	**0**	**0**
js-obfuscator-config1	14	0	2	0	2,997	0	0	0	0	0
js-obfuscator-2	6	0	0	0	2,995	0	0	0	0	0
js-obfuscator-3	11	0	0	0	3,096	0	0	0	0	0
Total js-obfuscator	**31**	**0**	**2**	**0**	**9,088**	**0**	**0**	**0**	**0**	**0**
JSObfu-config1 / **Total JSObfu**	**5**	**0**	**0**	**0**	**0**	**6,465**	**0**	**0**	**0**	**0**
JavaScript2img-config1 / **Total JavaScript2img**	**0**	**0**	**0**	**0**	**0**	**0**	**1,849**	**0**	**0**	**0**
DaftLogic-config1 / **Total DaftLogic**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3,195**	**0**	**0**
jsfuck-config1 / **Total jsfuck**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**765**	**0**
node-obf-config1 / **Total node-obf**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**976**

Table 5.8: Confusion matrix for the multiclass version of the *Code2BagOfPaths* model, extended by tools and configurations. In bold are the total values per class.

Considering the *Code2FeatureVector* model, it classifies correctly all regular files, regardless if they are minified or not. However, it fails to detect 11 obfuscated files, all transformed with *JSObfu*, the majority of which are also misclassified by the binary model due to their small size, and overall simplicity of code obfuscation. Table 5.9 displays the confusion matrix, extended to represent the different configurations of each obfuscator, for this model.

	Predicted Label									
	Regular	Jscrambler	javascript-obfuscator	defendjs	js-obfuscator	JSObfu	JavaScript2img	DaftLogic	jsfuck	node-obf
Regular	10,507	0	0	0	0	0	0	0	0	0
UglifyJS	10,524	0	0	0	0	0	0	0	0	0
Google Closure Compiler	9,001	0	0	0	0	0	0	0	0	0
Total Regular	**30,032**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Jscrambler-config1	0	2,584	0	0	0	0	0	0	0	0
Jscrambler-2	0	2,623	0	0	0	0	0	0	0	0
Jscrambler-3	0	2,556	0	0	0	0	0	0	0	0
Jscrambler-4	0	2,695	0	0	0	0	0	0	0	0
Total Jscrambler	**0**	**10,458**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
javascript-obfuscator-config1	0	0	2,573	0	0	0	0	0	0	0
javascript-obfuscator-2	0	0	2,600	0	0	0	0	0	0	0
javascript-obfuscator-3	0	0	2,646	0	0	0	0	0	0	0
javascript-obfuscator-4	0	0	2,671	0	0	0	0	0	0	0
Total javascript-obfuscator	**0**	**0**	**10,490**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
defendjs-config1	0	0	0	4,221	0	0	0	0	0	0
defendjs-2	0	0	0	4,432	0	0	0	0	0	0
Total defendjs	**0**	**0**	**0**	**8,653**	**0**	**0**	**0**	**0**	**0**	**0**
js-obfuscator-config1	0	0	0	0	3,013	0	0	0	0	0
js-obfuscator-2	0	0	0	0	3,001	0	0	0	0	0
js-obfuscator-3	0	0	0	0	3,107	0	0	0	0	0
Total js-obfuscator	**0**	**0**	**0**	**0**	**9,121**	**0**	**0**	**0**	**0**	**0**
JSObfu-config1 / **Total JSObfu**	**11**	**0**	**0**	**0**	**0**	**6,459**	**0**	**0**	**0**	**0**
JavaScript2img-config1 / **Total JavaScript2img**	**0**	**0**	**0**	**0**	**0**	**0**	**1,849**	**0**	**0**	**0**
DaftLogic-config1 / **Total DaftLogic**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3,195**	**0**	**0**
jsfuck-config1 / **Total jsfuck**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**765**	**0**
node-obf-config1 / **Total node-obf**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**976**

Table 5.9: Confusion matrix for the multiclass version of the *Code2FeatureVector* model, extended by tools and configurations. In bold are the total values per class.

The two models are able to detect obfuscation and identify the tool used with high precision and recall. However the *Code2FeatureVector* model outperforms the *Code2BagOfPaths* model in both of these tasks. The results obtained validate our hypothesis that using contextual features does not improve the detection of obfuscation compared to the use of features without context. Another interesting conclusion is that the *Code2FeatureVector* model only seems to fail in detecting obfuscation in code transformed by a particular tool when the files are too small, or the obfuscation does not have high resilience or potency (simpler transformations). Interestingly, the *Code2BagOfPaths* model is able to detect obfuscation in some of these files, even though they apply simpler transformations. This indicates that retaining some contextual information can be beneficial in classifying code from this particular tool. However, this model is not as reliable as the first, since it fails to detect obfuscation in code from two tools, and it is unable to detect obfuscation if the transformations generate paths similar but not equal to the ones the model was trained for. This also indicates that this second model would not be able to generalize to code transformed by different tools or unseen transformations. The *Code2BagOfPaths* model also classifies as obfuscated all files that are represented by a set of paths never seen, leading to a higher false positive values. This suggests that:

- More data is required to train the *Code2BagOfPaths* model, since there is an immense variety of code combinations and paths.
- The paths representation is not generic enough, and could be altered by, instead of using the paths to create the vocabulary, using the individual nodes - however, depending on the

implementation, some or even all contextual information may be lost with this approach.

Finally, it is also important to note that the process of traversing through the code's AST and computing features is more straightforward and takes fewer time than the one of extracting the path-contexts, which requires the computation of paths for the first terminals in the code (as described in Section 4.3). The features extracted for *Code2FeatureVector* can be directly fed into a classifier; however, the paths for *Code2BagOfPaths* must first be used as input for a *vectorizer* to extract features based on the training data, and these features are then fed into a classifier. This means additional time and memory spent in computing these context-based features.

5.4 Summary

Two different approaches were followed to implement an obfuscation detector based on the static characteristics of the code. The first approach is to traverse the code's AST, extract a set of paths that represent the code, and use those paths to automatically create features that differentiate obfuscated code from regular code. The second approach is to use a set of features, manually defined based on common obfuscation practices, extracted directly from the code's AST, with no context associated with them.

We found that choosing the paths used to represent the code is essential for the success of the first approach. We compare two selection approaches: selecting the paths randomly; and filtering them by the frequency of certain nodes and keywords (related to obfuscation). Our hypothesis was that randomly selecting the paths would allow a more diverse and realistic representation of the entire code, leading to better results. This hypothesis was confirmed by the experiments conducted, as using random paths results in higher F1-score values in both tasks addressed. Filtering the paths seems to induce the model in error if the regular files contain specific keywords or paths, or the paths selected for the obfuscated files are more common in regular code. However, these results only invalidate our filtering methodology, as others could be more useful and obtain better results.

By comparing the models for the two approaches - context versus no context -, it is possible to conclude that they are able to detect obfuscation and identify the obfuscator used to transform the code with high precision and recall. However, there is no additional benefit of using contextual features versus features manually defined, as the context-based model incorrectly classifies more regular and obfuscated files. Additionally, extracting the paths is a more costly process than computing the features, pointing to the overall efficiency and efficacy of the feature-based model.

In the end, the model that receives as input features with no context, *Code2FeatureVector*, seems to be a more reliable and less costly approach for detecting obfuscation. It is also easier to explain due to the nature of the features used.

Chapter 6

Additional Questions

In this chapter we present a set of experiments to evaluate the *Code2FeatureVector* model in different scenarios, as well as validating initial assumptions made while creating the dataset. These experiments also give additional insights on how the solution can be scaled to incorporate new obfuscators and data sources, and on how future experiments can be conducted. In this chapter we try to answer six questions:

- **What is the impact of training the model with fewer files?** Training the model with fewer files may or not have an impact in detecting obfuscation.
- **What is the impact of training the model with smaller files?** Using smaller files may or not be enough to detect obfuscation in files with various sizes.
- **Is the model biased to the sources of code used in training?** The model may or not be able to generalize to code collected from unknown sources.
- **Is the model able to detect obfuscated code transformed by unknown tools?** The model may or not be able to detect code obfuscated by unknown obfuscators.
- **What is the impact of training the model without minified code?** Training the model with minified code may or not have an impact on the detector's ability to distinguish obfuscated code from minified code, as both types of transformations apply some common techniques.
- **Is the model able to detect obfuscation in partially obfuscated code?** The model may or not be able to detect obfuscation in scenarios of partial obfuscation, where only a portion of the file is obfuscated.

6.1 Experimental Setup

To run the experiments, we use the binary version of the *Code2FeatureVector* model. The training set, testing set, the data splitting methodology, and workstation used are the ones described in Section 5.1. In general, we compare the results with the ones obtained by the *Code2FeatureVector*

model trained and tested with the original training and testing sets (see Section 5.3). In the remaining of this work, we refer to this model as *baseline model*.

Additionally, we create a new set of files to represent partial obfuscation. To accomplish this, since these files are only used for testing, we create two code collections from the original testing set: the first with every file with regular (including minified) code between 2 KB and 4,000 KB; and a second with all the obfuscated files with more than 1 KB and less than 2 KB. For each regular file in the first collection, we randomly select an obfuscated file, from the second collection, and attach it to the beginning of the regular file. The size limits were used to create a set of partially obfuscated code with at most 50% and at least 0.05% of obfuscation. This restriction was used to avoid files with more obfuscated code than regular code and percentages of obfuscation too close to zero. This is done to mimic a specific type of attack, where the obfuscated malware appears at the top of the file. Figure B.1 shows an example of the partially obfuscated code generated. After creating and parsing, this new set contains 10,727 partially obfuscated files, with obfuscation percentages between 0.05% and 50% .

To further assess the results obtained in some of the experiments, we compute the importance given by the model to each of the used features. To accomplish this we use the *feature_importances_* attribute of the Random Forest classifier implemented in *scikit-learn* [1], which stores the values for the importance given to each feature. These values are computed as the mean and standard deviation of accumulation of the impurity decrease within each tree [13].

6.2 What is the impact of training the model with fewer files?

An obfuscator, considering a specific configuration, applies the same techniques to all files. Even if these techniques are used with some random factors, such as names randomization, most transformations are captured by a set of *N* files. Our hypothesis is that after an *N* number of files, there is no need to keep increasing the size of the training set, as no new information is added.

Understanding the number of files necessary to train the detector has a significant interest, not only for performance and space-related motives, but in the scenario where code transformed with new tools is added to the dataset. Some of the tools used are open-source and freely available, which allows their use without significant restrictions. However, other tools such as *DaftLogic* and *JavaScript2img*, are online tools and have limits to the size and number of files transformed. Tools such as *jsfuck* and *node-obf* cannot be applied to some files since they create very deep ASTs that break the parser. Additionally, other tools are not freely available, such as *Jscrambler*, which can limit the number of samples available, due to associated costs - using more files leads to higher costs. Suppose the detector can deliver high precision and recall with fewer files. In that case, there is no need to use a large amount of obfuscated samples for each obfuscator, facilitating the process of transforming the files.

[1] https://scikit-learn.org/stable/

To test our hypothesis, we train the classifier with 5%, 15%, 25%, and 50% of the training data and test it with the original testing set. We then compare the results with the ones obtained by the baseline model.

Table 6.1 shows the results obtained. The model obtains a higher F1-score when using all files. However, the impact of reducing the number of files by 50% is not very significant. When using only 5% of all training files, the model is still able to detect obfuscation with high precision and recall. As the number of files in the training set increases, so does the model's training time.

Percentage of Files (%)	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	Training Time (s)
5	99.98	99.92	99.95	99.94	1.40
15	99.99	99.95	99.97	99.96	4.68
25	99.99	99.95	99.97	99.96	8.50
50	99.99	99.97	99.98	99.98	19.56
100 (baseline)	100.0	99.98	**99.99**	99.99	47.88

Table 6.1: Results obtained for training with 5%, 15%, 25%, and 50% of training files.

Table 6.2 shows the number of files misclassified by type of code - regular or transformed by a specific tool. We only represent the types where varying the size of the training data had impact in the code's classification. By assessing the results obtained by training the model with each percentage of files individually, it is possible to conclude that:

- Reducing the number of samples for each obfuscator only impacts the detection of code transformed by two tools: *js-obfuscator*, and *JSObfu*. The files transformed with the remaining tools are classified correctly independently of the percentage of files used.
- The majority of obfuscated files classified as regular were transformed by the same tool, *JSObfu*.
- The majority of obfuscated files classified as regular have smaller sizes.
- The regular files misclassified are both regular (not transformed) and minified.

	Percentage of training data used (%)				
	5	15	25	50	100 (baseline)
Regular	3	2	1	1	0
UglifyJS	3	2	2	2	1
Google Closure Compiler	3	1	1	1	0
js-obfuscator	11	4	2	1	0
JSObfu	28	24	23	15	10

Table 6.2: Number of files misclassified per code type, for each percentage of the training data used.

The results are a strong indicator that obfuscators tend to be repetitive in the transformations they apply, aiming at transforming the code without attempting to avoid detection. This validates our initial hypothesis that training the model with fewer files does not significantly hinder the detection. Since the model trained with fewer files still performs with high detection, and training with a small dataset is considerably faster, this approach can be used in future experiments. Additionally, we can use fewer files transformed by new obfuscators if we want to incorporate code obfuscated with these new tools in the dataset, which gives us insight into how to scale our solution.

6.3 What is the impact of training the model with smaller files?

Besides discarding empty files, we did not initially apply any size restrictions to the original code while creating the dataset. Note that we do apply size restrictions to code transformed by some tools, as explained in Section 4.2.4 and Section 4.3.

With this experiment we aim at understanding the impact of using a training set with smaller files versus a training set with different sized files. Our hypothesis is that using a set of smaller files to train the detector could improve performance (less time in parsing and training) without significantly hindering the model's effectiveness, as the obfuscated code would still maintain its primarily frequency-based features.

We create three additional training sets - *set1*, *set2*, and *set3* - with the following methodology:

1. Compute the 25th, 50th, and 75th percentiles of the file size for the regular (not transformed) code and for the code transformed by each tool - minifiers and obfuscators - as displayed in Table B.1.

2. Add to *set1* all files with sizes lower than the value of the 25th percentile of its corresponding type - regular or for the tool that transformed the code.

3. Add to *set2* all files with sizes lower than the value of the 50th percentile of its corresponding type. This set includes all files in *set1*.

4. Add to *set3* all files with sizes lower than the value of the 75th percentile of its corresponding type. This set includes all files in *set2*.

The detector is trained with each training set and evaluated with the original testing set. The results are then compared with the ones obtained by the baseline model.

Table 6.3 presents the number of files misclassified, for each training set, per code type - regular or transformed by a specific tool. Since the obfuscated files classified incorrectly were transformed either by *JSObfu* or *defendjs*, only these two obfuscators are represented in the table. The remaining obfuscated code was classified correctly independently of the training set used. As the size of the files decreases, the model's performance decreases as well, as it is less able to classify larger files. Therefore, the baseline model (which is trained with the original training set) performs the best. However, by using smaller files, the model is still able to detect obfuscation with high precision and recall, failing mostly in files transformed by *defendjs*. The results obtained by training with the three new sets in terms of evaluation metrics are presented in Table B.2.

	Training Set			
	set1	*set2*	*set3*	Original (baseline)
Regular	4	1	1	0
UglifyJS	263	11	2	1
Google Closure Compiler	197	1	0	0
defendjs	1,101	470	100	0
JSObfu	5	9	13	10

Table 6.3: Number of files misclassified per code type, for each training set.

The majority of files misclassified transformed by *defendjs* were transformed with the same configuration - *defendjs-2*. Further analysis of the size distribution per configuration of this tool shows that the files transformed with *defendjs-config1* tend to be smaller than the ones transformed with the *defendjs-2* configuration, as the first one applies a set of simpler techniques. Therefore, by removing the larger files, more files transformed with the second configuration are being removed from the training sets, which may explain why more files transformed with the second configuration are misclassified.

Conversely to the other obfuscated code, using smaller files in training allows for better detection of files transformed with *JSObfu*. By manually assessing the files transformed with *JSObfu* that the model is a misclassifying when trained with each training set, we observe that:

- When training with *set1* the model misclassifies five files. Four out of these five files are smaller than 98 bytes. The 5th file contains code for a large dictionary declaration that was not obfuscated.

- When training with *set2* the model misclassifies nine files. Five out of these nine files are also misclassified when training with *set1*. The remaining four files are smaller than 488 bytes.

- When training with *set3* the model misclassifies 13 files. Nine out of these 13 files are also misclassified when training with *set2*. The remaining four files are smaller than 348 bytes.

- When training with the initial training set, the model misclassifies ten files. All these files are also misclassified when training with *set3*.

Since the value for the 25th percentile of the sizes of files transformed by *JSObfu* is 880 bytes, all the files misclassified when training with the different sets, except for one, belong to this percentile. As described in Section 4.2.4, this obfuscator tends to output simpler obfuscated code that resembles minified code when applied to smaller files - which is the case in most of the misclassified files. A possible explanation for the increase in misclassified files transformed with *JSObfu* when using larger files relates to the presence of more and larger minified files in the training set. By training with more minified files, which in both *set1* and *set2* have sizes lower or equal to 1.04 KB (slightly above the 880 bytes), the model may be associating more files transformed with *JSObfu* with minification, misclassifying them.

The results also show that smaller files can generalize to larger files for the majority of obfuscators. However the size difference between percentiles, for some obfuscators, namely *JavaScript2img*, *DaftLogic*, *jsfuck*, and *node-obf*, is not very significant, which can also explain why training with smaller files is enough to classify correctly all files from these tools.

6.4 Is the model biased to the sources of code used in training?

To build the dataset, JavaScript code was collected from three different sources: websites, *GitHub* repositories, and *NPM* packages. This was done to prevent possible bias that could occur if only one source of code was used. In this experiment, we aim at understanding the impact of training the detector with code from fewer sources to validate, or not, our initial assumption. Our hypothesis is that a diverse dataset allows a better generalization of the model to different scenarios and types of code.

A strong motivation for this experiment is related to the process of collecting and processing the code. Collecting code from the web can be a very time-consuming process and requires a time-consuming preprocessing to discard minified and obfuscated files (that are more common in code collected from the web). Therefore, it would be relevant to understand if training with code from the web improves the classification as if not, this source could be dismissed in future works.

To accomplish this, we remove all the files collected from a specific source from the training set. The model is then trained with this new set and tested with the original testing set (which contains code from all sources). This process is performed for each source independently, generating three additional models: *Code2FV_wo_Web*, *Code2FV_wo_GitHub*, and *Code2FV_wo_NPM*, which are trained without files collected from the web, *GitHub*, and *NPM*, respectively.

To better understand the results obtained, we start by analyzing how the baseline model classifies files from different sources. To accomplish this, we divide each class in the confusion matrix (regular and obfuscated) by source (web, *GitHub*, *NPM*), as shown in Table 6.4. The model misclassifies more files from the web (seven files) than from any other source. However, other factors must be taken into account. All obfuscated files classified incorrectly were transformed by the same tool, *JSObfu*, which, as mentioned in Section 4.2.4, tends to originate simple obfuscated

code if the input file is too small, which is the case for all the misclassified web files. This seems to be a more probable cause for the detector's behavior.

	Predicted Label	
	Regular	Obfuscated
Regular Web	9,981	0
Regular *GitHub*	8,567	0
Regular *NPM*	11,483	1
Total Regular	**30,031**	**1**
Obfuscated Web	7	18,893
Obfuscated *GitHub*	2	13,875
Obfuscated *NPM*	1	19,199
Total Obfuscated	**10**	**51,967**

Table 6.4: Baseline model's confusion matrix extended by source. In bold are the total values per class.

The results obtained with the *Code2FV_wo_Web* model, displayed in Table 6.5, show that the model does not misclassify any regular files and it misclassifies 32 obfuscated files collected from the web. There are two possible explanations for this behavior. The first is related to the size of the files, as the ones retrieved from the web tend to be smaller. By removing them from the training set, the detector is not trained to classify similar sized files as obfuscated, which can lead to their incorrect classification. Another explanation is the overall structure and semantic of the web files. As they are solely client-side code, their overall semantics and content differ from other types of code. However, client-side code is also collected from other sources. To better understand why the detector fails to classify these files, we manually assess them, concluding that they are smaller than 512 bytes and were transformed by the same tool, *JSObfu*, seven of which the baseline model also classifies incorrectly.

	Predicted Label	
	Regular	Obfuscated
Regular Web	9,981	0
Regular *GitHub*	8,567	0
Regular *NPM*	11,484	0
Total Regular	**30,032**	**0**
Obfuscated Web	32	18,868
Obfuscated *GitHub*	0	13,877
Obfuscated *NPM*	0	19,200
Total Obfuscated	**32**	**51,945**

Table 6.5: *Code2FV_wo_Web* model's confusion matrix extended by source. In bold are the total values per class.

Considering the *Code2FV_wo_GitHub* model, all files misclassified are from the removed source, in this case *GitHub* repositories, as shown in Table 6.6. The model is able to classify all regular files correctly, but it classifies two obfuscated files as regular. These two files were transformed with *JSObfu*, and the baseline model also fails to classify them as obfuscated. The fact that all other files misclassified by the baseline model are classified correctly by this model suggests that training with code from *GitHub* repositories is slightly hindering the detector's efficacy. To better understand these results, we compute the importance of the features used by this model, as described in Section 6.1, and compare them with the importance given by the baseline model. By assessing the top 20 most important features of each model, we conclude that:

- Both models give more importance to similar sets of features - considering their 20 most important features.

- The models differ mainly in the order they prioritize these features.

- The baseline model's top 20 most important features includes the *Frequency of declared variables' names that are encoded*, which is not included in the *Code2FV_wo_GitHub* model's top 20.

- The *Code2FV_wo_GitHub* model's top 20 most important features includes the *Average frequency of vowels in declared variables' names*, which is not included in the baseline model's top 20.

The differences in importance given by the *Code2FV_wo_GitHub* model to certain features in comparison to the baseline model, may explain the obtained results. This suggests that the code collected from *GitHub* may differ slightly from the code collected from other sources.

	Predicted Label	
	Regular	Obfuscated
Regular Web	9,981	0
Regular *GitHub*	8,567	0
Regular *NPM*	11,484	0
Total Regular	**30,032**	**0**
Obfuscated Web	0	18,900
Obfuscated *GitHub*	2	13,875
Obfuscated *NPM*	0	19,200
Total Obfuscated	**2**	**51,975**

Table 6.6: *Code2FV_wo_GitHub* model's confusion matrix extended by source. In bold are the total values per class.

Finally, when considering the *Code2FV_wo_NPM* model, the results show that the detector classifies six regular files as obfuscated and one obfuscated file as regular, all collected from *NPM* packages, as shown in Table 6.7. Two of these files are also misclassified by the baseline model (one regular and one obfuscated). By manually assessing the remaining files, we observe that:

- All of them have large comments which imply a large average of number of characters per line.
- A first file has high frequencies of specific keywords related to obfuscation, such as the % symbol, in addition to encoded numbers, small identifiers and high frequency of decimal characters in the identifiers used.
- A second file has extremely long identifiers and uses uncommon characters, such as the Θ symbol, on the identifiers.
- A third file as a high frequency of variable assignments, in addition to a high frequency of numbers, and unusual characters, such as $, in the used identifiers.

To understand if these characteristics are being reflected in the features used, we compute the importance of the features used, as described in Section 6.1. We conclude that this new model gives higher importance than the baseline model to features such as *Average number of character per line*, *frequency of certain keywords*, *Frequency of encoded numbers*, *Average frequency of letters in other identifiers' names*, and *Frequency of AssignmentExpression nodes* (frequency of assignments). The importance given to these features, and others that differ from the baseline model, can also explain why the model classifies correctly files that the baseline model misclassifies. These results seem to indicate that the code collected from *NPM* packages may differ, to some degree, from the code collected from other sources.

	Predicted Label	
	Regular	Obfuscated
Regular Web	9,981	0
Regular *GitHub*	8,567	0
Regular *NPM*	11,481	3
Total Regular	**30,029**	**3**
Obfuscated Web	0	18,900
Obfuscated *GitHub*	0	13,877
Obfuscated *NPM*	1	19,199
Total Obfuscated	**1**	**51,976**

Table 6.7: *Code2FV_wo_NPM* model's confusion matrix extended by source. In bold are the total values per class.

Although the use of code collected from *GitHub* slightly hinders the model's detection, it is unknown if it helps generalize to sources of code not present in the dataset. Therefore, removing files obtained from this source is a possibility but does not necessarily imply better predictions in unknown scenarios. Overall, the results obtained validate our assumption that the dataset used must incorporate different code sources to better generalize the model. However, it raises the question of how the detector will generalize to code from other sources. A possibility is that if the semantics of the code are very different from the ones in the training set, the model will raise more false positives. Table B.3 presents results obtained for each model regarding different metrics.

6.5 Is the model able to detect obfuscated code transformed by unknown tools?

Different obfuscators apply different techniques or the same technique with different signatures, which results in a variety of obfuscated code. Ideally, a detector should be able to detect obfuscation independently of the tools used in its training, as other obfuscators are and will be available. This experiment aims to evaluate the performance of the detector in the presence of code obfuscated with unknown tools. Our hypothesis is that the detector is able to generalize to code transformed by some obfuscators but not others, depending on how similar the obfuscated code is to the code present in the training set.

As described in Section 4.2.4, all the obfuscators have one configuration in common (*config1*), which applies simpler obfuscation techniques. Thus, the files transformed with these configurations are obfuscated with the same set of techniques (in general, as some obfuscators are not configurable). The goal is to understand if, by training the model with similar techniques, the detector is able to recognize patterns in these techniques and identify them in code transformed with unknown obfuscators that apply the same set of techniques.

To evaluate the model in the presence of code obfuscated with unknown obfuscators, all files transformed with a specific obfuscator are removed from the training set. The detector is then

trained with this new set and tested with the original testing set (which contains code transformed with all the obfuscators). This procedure is applied independently to all obfuscators - nine times in total.

Figure 6.1 displays the number of files that the model classifies correctly and incorrectly considering only the files transformed by the obfuscator removed from training, per obfuscator removed. The obtained results show that:

- By removing the files transformed with *Jscrambler* from the training set, the model misclassifies six out of the 10,458 files transformed with this tool. These six files were transformed by the same configuration of the tool, *Jscrambler-config1*, and are six out of the 2,578 files transformed with this configuration. The model classifies all other obfuscated code and regular code correctly.

- By removing the files transformed by *javascript-obfuscator*, the detector is able to classify correctly all obfuscated code transformed by other tools and all regular code. However, it misclassifies 276 out of the 10,490 files transformed with this tool, which are all transformed with the *javascript-obfuscator-config1* configuration. These 276 files are 276 out of the 2,573 files transformed with this configuration of the tool.

- When considering the model trained without files transformed by *js-obfuscator*, the results show that it classifies incorrectly more than 50% of the files transformed by the different configurations of this tool (5,296 out of 9,121). The majority of the misclassified files are the smaller files transformed by this tool. The model classifies all other obfuscated and regular code correctly.

- By removing the code transformed by *JSObfu*, the model classifies 5,303 out of the 6,470 obfuscated files as regular and classifies the remaining obfuscated and regular code correctly.

- By removing the obfuscators *defendjs*, *JavaScript2img*, *DaftLogic*, and *node-obf* one by one from the training set, the model is not able to generalize and it classifies incorrectly all samples from the removed tool. However, in all cases, it classifies all remaining obfuscated and regular code correctly.

- By removing all files transformed with *jsfuck* the detector classifies all the files correctly. This means the detector has an accuracy, precision, and recall of 100%, and that by training with code transformed by this tool, the detector's efficacy is hindered.

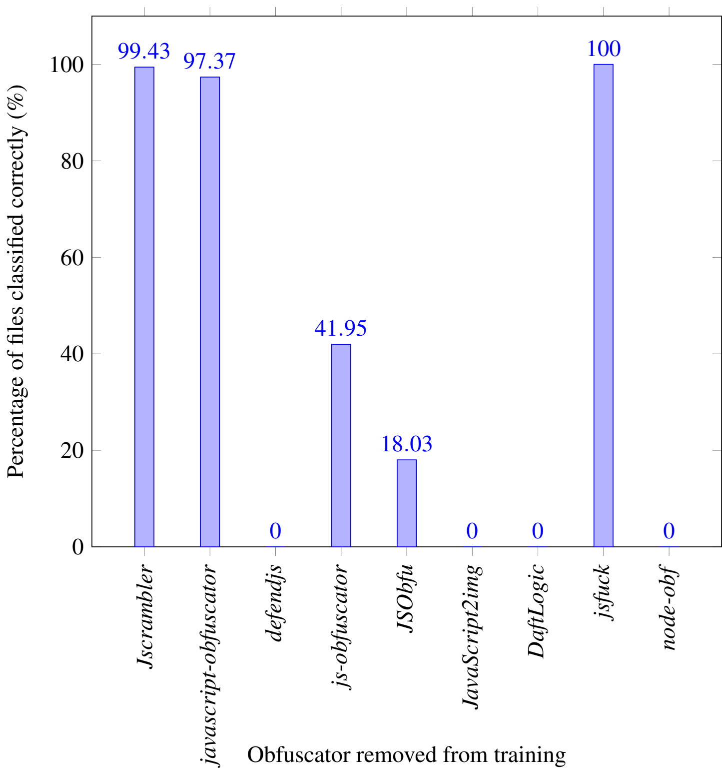


Figure 6.1: Percentage of files, transformed with the obfuscator removed from the training set, correctly classified per obfuscator removed.

From the points mentioned above, it is possible to conclude that the code transformed with some obfuscators can be generalized by the code transformed with others, but not all, validating our hypothesis. Using the *config1* configuration - common to all obfuscators - does not seem to have a direct impact on the generalization of the model to code transformed with similar techniques, which contradicts our initial assumption. This seems to allude to the relevance of training the classifier with code transformed with various obfuscators and obfuscation techniques, reiterating the importance of a diverse dataset in developing a reliable detector. However, it also proves that the model can not be generalized to all variations of obfuscation, as different obfuscators output obfuscated code with different characteristics. Notwithstanding, the model generalizes to code transformed by some obfuscators - *Jscrambler*, *javascript-obfuscator*, *jsfuck* - from the obfuscated code transformed with the other tools present in the training set. This indicates that obfuscated code transformed by some unknown tools can be detected based on the combination of characteristics of the obfuscated code present in the training set. The fact that the model is not able to detect code transformed with *DaftLogic* if not trained to do so is not very surprising since it resembles minified code and is very different from the code transformed by the other obfuscators (e.g. uses the *eval* function more frequently than the others). Both *JavaScript2img* and *node-obf* output obfuscated code very different from the code obfuscated by the other tools used, which may explain why the model can not detect obfuscation by these tools if not trained to do so. After manually assessing the code transformed by *defendjs* and *js-obfuscator*, we note that, in general, the first

resembles the code transformed by *Jscrambler*, and the second resembles the code transformed by *javascript-obfuscator*. Although these similarities can be detected by human assessment, they are not represented in the features used, which may explain why the model fails to detect obfuscation in the code transformed by *defendjs* and part of the code transformed by *js-obfuscator*.

6.6 What is the impact of training the model without minified code?

We consider minified code as regular JavaScript since it does not conceal the program's purpose. Minification techniques are mainly based on identifier name shortening and whitespace removal.

This experiment aims at understanding the impact of training the detector with minified files versus without. Our hypothesis is that, since obfuscation often includes minification of the code, if the model is not trained to classify minified code as regular, it will classify incorrectly more minified files.

We train the detector without minified code by removing minified files from the original training set. Then we use the original testing set (which includes minified files) to evaluate the model and compare the results with the ones obtained by the baseline model.

Table 6.8 presents the results obtained. In the matrix, the regular class is subdivided into three subclasses: *Regular* (not minified), *UglifyJS* (minified with *UglifyJS*), and *Google Closure Compiler* (minified with *Google Closure Compiler*). As shown, no obfuscated files were classified as regular. However, 409 regular files were labeled as obfuscated, from whom 285 were minified with *UglifyJS* and the remaining with *Google Closure Compiler*. It should be noted that neither the size of the files or their provenience seem to be related to the obtained results as they are varied and overall balanced.

	Predicted Label	
	Regular	Obfuscated
Regular	10,507	0
UglifyJS	10,239	285
Google Closure Compiler	8,877	124
Total Regular	**29,623**	**409**
Total Obfuscated	**0**	**51,977**

Table 6.8: Confusion matrix extended to represent both regular and minified code and tools. In bold are the total values per class.

By removing minified code from the training set, the model is able to classify all obfuscated files correctly, obtaining a recall of 100%, which is not the case for the baseline model, which classifies incorrectly ten obfuscated files. However, more regular files are classified as obfuscated, reducing the precision from 100.0% to 99.22%, compared with the results obtained by the baseline model - as shown in Table B.4 -, which only misclassifies one regular (minified) file.

To better understand these results, we compute the 20 features with higher importance, as described in Section 6.1, and compare them for both the new model and the baseline model.

Conversely to the baseline model, the one trained without minified code gives high importance to features such as *Frequency of indentation characters*, *Average number of characters per line*, and *Average frequency of vowels in declared variables' names*. Considering that both obfuscation and minification often remove indentation characters from the code, compressing it to a reduced number of lines, and apply name shortening techniques, the importance given to the features above may explain why the classifier misclassifies more minified code in this scenario.

Since our goal is to have a detector that can distinguish obfuscated code from regular code, regardless of whether this code is minified or not, if we evaluate the models by considering both the false positives and the false negatives, the baseline model outperforms the model tested. Therefore, the obtained results validate our hypothesis and reiterate the importance of training the model with minified code.

6.7 Is the model able to detect obfuscation in partially obfuscated code?

Partial obfuscation occurs when only a portion of the code is obfuscated. This is often the case in malicious attacks, where malware is obfuscated and then injected into regular code. In this case, the obfuscation aims at compromising the detection of malware by security systems.

To understand how to detect partial obfuscation we conduct two different experiments, as explained in Section 6.7.1 and Section 6.7.2. The obfuscated files in the training and testing sets are completely obfuscated (excluding some exceptions were the tools fail to transform portions of the code). To create a set of partially obfuscated files, we use the methodology described in Section 6.1.

6.7.1 Classifying the whole file

The goal of this experiment is to understand how the detector performs in the presence of partial obfuscation. Our hypothesis is that if a large portion of the file is obfuscated, there is a higher possibility that the detector is able to classify the file as obfuscated. However, if the code is mainly regular, the classification may be compromised, and the obfuscation may pass unnoticed. In this experiment, we use the baseline model to classify the files in the partially obfuscated set.

In this scenario, the detector classifies around 82% of partially obfuscated files as regular, and the remaining 18% as obfuscated, as shown in Table 6.9. It is worth noticing that none of the original regular files nor the obfuscated files used to create this set were previously misclassified by the baseline model. This a strong indicator that the results obtained are solely due to the presence of partially obfuscated code.

	Regular	Obfuscated
Regular	0	0
Obfuscated	8,832	1,895

Table 6.9: Confusion matrix for the partially obfuscated files, when classifying the whole file. No regular files are classified in this experiment.

Figure 6.2 shows the percentage of files misclassifed per percentage of obfuscation in the code. As expected, files with lower percentages of obfuscated code are harder to detect. As this percentage increases, the detector's ability to recognize obfuscated code improves.

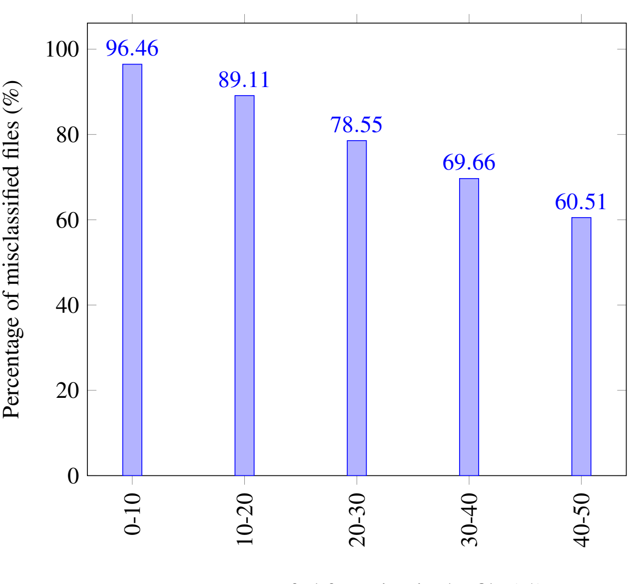


Figure 6.2: Percentage of misclassified partially obfuscated files, per percentage of obfuscation in the code.

The results can be explained by the nature of the features used. As described in Section 4.3.2, the features are computed based on the entire code and are mainly based on frequencies of specific values and statistical metrics, such as means and standard deviations. The frequency-based features are computed by dividing X, the total number of times a specific value (node, keyword) appears in the code, by N, the total number of possible outcomes of that value in the code, where $X \in N$. Suppose N is significantly larger than X. In that case, the computed result loses its significance as it will tend to zero.

The results confirm our hypothesis that there is a higher probability that the detector classifies partially obfuscated code as obfuscated if there is a high percentage of obfuscation. If there is a small portion of obfuscated code, the model has a higher probability of failing the detection.

6.7.2 Classifying only a portion of the file

In our set of partially obfuscated files, the obfuscation appears only at the beginning of the code. Our hypothesis is that, by using the N top statements of the program, instead of the entire code, the detector will be able to detect partial obfuscation in that code. The goal of this experiment is to understand if partial obfuscation (in the specific scenario we use it), can be efficiently detected by classifying only a portion of the file. Considering the results obtained in Section 6.3, we expect that the detector is able to classify correctly small portions of the code, regardless if they are obfuscated or not.

We refer to the child-nodes of the root node (which is the program node) as statements. For example, if a program as a root node with five child-nodes, we consider that this program as five statements. A statement includes the node and all its descendent-nodes.

We select the first N statements of the program. We limit the number of statements to a maximum of 200 [2] nodes. For example, if a program has 3 statements, the first with 50 nodes, the second with 25 nodes, and the third with 170 nodes, we only use the two first statements, as using all would exceed our node limit. However, if a program has a first statement with more than 200 nodes, we use that statement, exceeding the limit. We parse these statements and then apply the baseline model to classify them. This is done for both partially obfuscated files and regular files. The regular files classified are the ones used to build the partially obfuscated code set described in Section 6.1, with 10,943 files.

Table 6.10 shows the confusion matrix obtained. As shown, all regular files are classified correctly. Around 12% of the partially obfuscated files are classified as regular, an improvement over classifying the whole file.

	Regular	Obfuscated
Regular	10,943	0
Obfuscated	1,323	9,404

Table 6.10: Confusion matrix for the regular and partially obfuscated files, when classifying only a portion of the file.

With this approach, the obfuscation percentage of the entire code is not directly related to its classification, as only N statements are selected. However, the percentage of statements used can explain, to a degree, the obtained results, as using more statements can result in selecting more regular code, hindering the classification. Therefore, we divide the results by the percentage of statements selected from the program, as displayed in Figure 6.3. From here, it is possible to observe that, in general, in the files where the first 200 nodes correspond to a smaller percentage of the code's statements, the percentage of misclassification is minor.

[2] We choose 200 nodes by rounding the value of the 25th percentile of number of nodes of the obfuscated samples, which is 189.

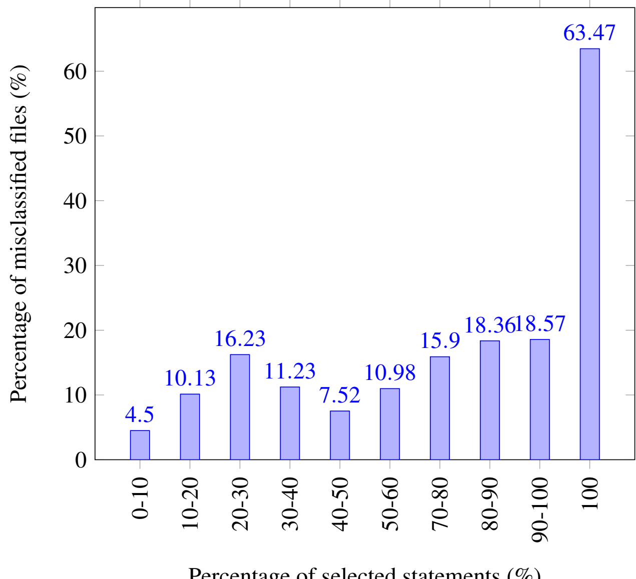


Figure 6.3: Percentage of misclassified partially obfuscated files, per percentage of statements selected.

The increase on the percentage of misclassified files when using almost all statements in the code can be explained by the results obtained from the previous experiment. As described in Section 6.7.1, the detector's ability to identify obfuscated code is affected by the percentage of obfuscation in the file. If a large percentage of a program's statements is being used to make the classification, then the results are similar to the ones obtained by classifying the entire file. Additionally, there is a specific case where in the AST the obfuscated and regular code are grouped in the same statement, so when we make the selection of the statements, we select the entire program. This occurs when the obfuscated code is a *CallExpression* node and the regular code is an immediately-invoked function expression, IIFE [3]. In the described scenario, these expressions are considered arguments of a *CallExpression* node. Figure 6.4 depicts an example of this situation.

[3]IIFEs are used to immediately execute a function as it is created.

```
eval(/* obfuscated code */)
(function () {
    /*regular code */
})
```

(a) Code.

```
{
  "type": "Program",
  "body": [
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "CallExpression",
        "callee": {
          "type": "CallExpression",
          "callee": {
            "type": "Identifier",
            "name": "eval",
          },
          "arguments": [],
        },
        "arguments": [
          {
            "type": "
                FunctionExpression",
            "params": [],
            "body": {
              "type": "BlockStatement
                  ",
              "body": [],
            },
          }
        ],
      },
    }
  ],
}
```

(b) Abstract Syntax Tree.

Figure 6.4: Example of partially obfuscated code where the regular code is an IIFE and the obfuscated code is a *CallExpression*, and its corresponding AST. The example only contains the essential part of the program - the comments are placeholders for the rest of the code - and of the AST.

The results validate our hypothesis that using only a set of the N first statements in the program is an effective approach in detecting partial obfuscation. However, this approach is only valid in the scenario described, where it is expected that the partial obfuscation appears in the beginning of the code. We expect it is possible to consider locations other than the beginning of the code by skipping a number of AST nodes before applying this technique but did not validate it.

6.8 Summary

We conducted several experiments to test and validate our approach. We use the *Code2FeatureVector* model in these experiments, namely its binary version. We expect that the overall results, can be generalized to its multiclass version.

We start by conducting a set of experiments that validate initial assumptions made while creating the dataset, giving helpful insights on how to scale our solution. In the first experiment we train the model with fewer data, concluding that the model can be trained with fewer files and still achieve high precision and recall values. The second experiment consisted of training the model with only the smallest files of each code type - regular and transformed with a specific tool. The results show that in general, if possible, it is better to use a training set with files with various sizes.

The third and fourth experiments aim to understand how the detector generalizes to different scenarios. In the third experiment we evaluate how the model generalizes to different sources of code. To accomplish this, we independently remove the code collected from each source (web, *GitHub*, *NPM*) from the training set. Then we test the model with files collected from all sources. The results show that the model only classifies incorrectly a few files, all collected from the sources removed (the number of files varies according to the source removed). The fourth experiment aims to understand how the detector behaves in the presence of code obfuscated by unknown tools. We apply a similar methodology as the one in the third experiment. The results show that the detector can detect obfuscation in code transformed by some unknown obfuscators but not by all, emphasizing the importance of using a diverse dataset that incorporates as many obfuscators and obfuscation techniques as possible.

In the fifth experiment, our goal is to understand the impact of training the model with minified files. To accomplish this we train the model without minified code. In this case, the detector does not misclassify any obfuscated code, but misclassifies significantly more regular (minified) files than the model trained with minified code. Therefore, we consider that using minified code in the training set is essential to make the distinction between obfuscated and minified JavaScript.

In the final experiment, we evaluate the detector in the context of partial obfuscation. We start by using the previously trained model to classify the set of partially obfuscated files, concluding that most files are classified as regular. In this scenario, the classification is closely related to the percentage of obfuscation in the file. In a second stage of the experiment, we select and classify only the first N statements of the program - based on the fact that we know *a priori* that the obfuscated code is at the beginning of the program. In this case, the model classifies as obfuscated most partially obfuscated files, an improvement over classifying the whole file.

Chapter 7

Conclusions

The main goal of this dissertation is to propose and develop a reliable solution for detecting obfuscated JavaScript and identifying the obfuscator used to transform the code. Our first task consisted of studying JavaScript's main features, vulnerabilities, and transformations. We study and detail both obfuscation and minification, and their techniques and tools. Additionally, we explore and review the state-of-the-art approaches that aim at resolving similar tasks, comparing and assessing each one. This initial work is required to understand the characteristics of obfuscated code and the different approaches that can be implemented to successfully detect it.

An essential component of our work, which differentiates it from most projects in this area, is the dataset used. We create our dataset to incorporate code transformed with a diverse set of obfuscators and obfuscation techniques, as well as code collected from different sources. Most projects in this area have small unbalanced datasets, with code collected from one or two sources, often with no control over the tools and obfuscation techniques applied. Another underlying goal in the creation of the dataset was its code quality. To accomplish this, we implement a preprocessing step to filter unwanted code, such as transformed and duplicated files. In the end, we obtain a large and diverse dataset that is well suited for the tasks addressed.

After further analysis of the state-of-the-art, we identify two distinct approaches for detecting obfuscation by statically analysing the code. The first is to extract paths from the code's AST, that retain the surrounding context of specific nodes or keywords. The second, and more common, is the use of defined features based on standard obfuscation practices. Based on this, we compare two different approaches to detect obfuscation: *Code2BagOfPaths*, which is a Multinomial Naive Bayes that receives as input context-based features automatically generated from a vocabulary of path-contexts; and *Code2FeatureVector*, which is a Random Forest that receives as input features defined with no context associated with them. Our results show no additional benefit in using the contextual information to make the classification. The context-based model classifies incorrectly more regular and obfuscated files than the one that uses the defined features, in both tasks addressed. After further experimenting with the *Code2FeatureVector*, we validate some assumptions

made while creating our dataset, namely the importance of the diversity of the code. Additionally, we validate our solution against partially obfuscated code, concluding that if we use the entire program to make the classification, the detector is not able to detect obfuscation in most files. However, by using only a portion of the code, we obtain very positive results, as the detection rate increases significantly.

The proposed solution has some limitations, leaving some space for improvement. The first limitation is that we can not guarantee that the solution will be able to detect code obfuscated with unknown tools, as the model is able to detect code transformed with some obfuscators even if not trained to do so, but not with others. This makes sense considering that different obfuscator apply different techniques or similar techniques in different manners. Our approach to minimize the impact of this limitation is to incorporate code transformed with a variety of obfuscators and techniques. Since the model can be trained with fewer samples and still be able to detect obfuscation with high precision and recall, it is possible to scale our solution to incorporate code transformed with more obfuscators, even if few samples are available. However, more research on how to generalize the detector should be considered.

Another limitation is the detection of partially obfuscated code. Our first approach of classifying the entire program performs poorly. A second approach that uses only a set of the first N statements obtains significantly better and promising results. However, this approach is only valid in the case where the obfuscated code appears in the beginning of the program - to mimic a malware attack -, limiting its use. Notwithstanding, this methodology can be extended to be viable in other scenarios. For example, instead of using only the first N statements, we can partition the file into chunks, by following the same splitting methodology. Then use the detector to classify these chunks individually. If at least one chunk is classified as obfuscated, then the file would be considered obfuscated (or partially obfuscated). Another possible approach, is to use the code's call graph to identify sections of the code that do not depend or are called by other sections, and classify only these independent sections - based on the assumption that obfuscated malware is not dependent of other expressions in the code, and vice-versa.

By analyzing the previous work on this area in the light of our results, it is possible to theorize that some of the solutions would not be able to generalize to obfuscated code transformed with tools and techniques different from the ones used in training (which are, in general very few), and, in some cases, to the presence of minified code. None of the state-of-the-art solutions addresses the problem of partially obfuscated code, and considering most classify the code as an all, it is not expected they would perform as well in this scenario.

In the end, all goals set for this work were achieved successfully, obtaining a solution that can successfully detect obfuscation and identify the obfuscator used to transform the code, with an F1-score (/weighted average F1-score) of 99.99% in both these tasks. Although it has some limitations, it is still a reliable solution for detecting obfuscated code transformed with a specific set of obfuscators, and it can be easily scaled to incorporate code transformed with new obfuscation tools and techniques.

The field of obfuscation in general and its detection have many different challenges that can

be addressed. The first problem would be to research the automatic deobfuscation of code. Ideally, a solution would be able to deobfuscate any obfuscated code. However, as we show in this dissertation, obfuscation presents itself in varied forms. Therefore, a starting point could be first to identify the obfuscator used by applying our detector to the code and then use that information in the deobfuscator. If the deobfuscator is implemented to deobfuscate code transformed with a specifc set of obfuscators, it can be used in this scenario. Further work should be done to reach a more generic solution. The deobfuscation of the code is essential to understand the code's intents, namely if it is malicious or not, mitigating possible attacks.

Another research topic would be the automatic obfuscation of code to be undetected by obfuscator detectors and manual assessment. In this case, the code would be transformed to resemble regular code while still concealing its intended behavior. This code would look untransformed to human assessment, yet its behavior would be hard to understand after further evaluation while also compromising its static analysis. This type of obfuscation could be used in scenarios of intellectual property protection and to prevent benign files from being flagged by security systems.

Appendix A

Dataset Preparation

A.1 Data Collection

```
https://api.github.com/search/repositories?per_page=100&page=1&q="chrome extension
    "+language:JavaScript%26sort=stars%26order=desc
```

(a) Top 100 browser (Chrome) extensions repositories.

```
https://api.github.com/search/repositories?per_page=100&page=1&q="vanilla
    javascript"+language:JavaScript%26sort=stars%26order=desc
```

(b) Top 100 vanilla JavaScript repositories.

```
https://api.github.com/search/repositories?per_page=100&page=1&q="server-side"+
    language:JavaScript%26sort=stars%26order=desc
```

(c) Top 100 repositories that contain server-side code.

Figure A.1: Examples of the requests' endpoints issued to *GitHub*'s REST API. These requests retrieve 100 repositories from the first results page, but can be altered to retrieve from other pages as well.

A.2 Data Transformation

```
function helloWorld() {
    const hello = "Hello World!"
    console.log(hello);
}
helloWorld();
```

Figure A.2: Regular JavaScript code.

A.2.1 *Jscrambler*

```
{
    "params": [
      {
        "options": {},
        "name": "whitespaceRemoval"
      },
      {
        "name": "booleanToAnything"
      },
      {
        "name": "stringSplitting",
        "options": {
          "chunks": [
            2,
            4
          ]
        }
      },
      {
        "name": "numberToString",
        "options": {}
      },
      {
        "name": "identifiersRenaming",
        "options": {
          "mode": "SAFEST"
        }
      },
      {
        "name": "variableMasking",
        "options": {
          "options": []
        }
      },
      {
        "name": "functionOutlining",
        "options": {
          "features": []
        }
      }
    ],
    "areSubscribersOrdered": false ,
    "useRecommendedOrder": true ,
    "jscramblerVersion": "7.0",
    "tolerateMinification": true ,
    "profilingDataMode": "off",
    "useAppClassification": true ,
    "browsers": {}
  }
```

Figure A.3: *Jscrambler-config1*. First configuration used for the obfuscator *Jscrambler*.

```json
{
  "params": [
    {
      "name": "objectPropertiesSparsing"
    },
    {
      "name": "variableMasking"
    },
    {
      "name": "whitespaceRemoval"
    },
    {
      "name": "dotToBracketNotation"
    },
    {
      "name": "stringConcealing"
    },
    {
      "name": "functionReordering"
    },
    {
      "name": "propertyKeysObfuscation",
      "options": {
        "encoding": [
          "hexadecimal"
        ]
      }
    },
    {
      "name": "regexObfuscation"
    },
    {
      "options": {
        "features": [
          "opaqueSteps"
        ]
      },
      "name": "controlFlowFlattening"
    },
    {
      "name": "booleanToAnything"
    },
    {
      "name": "identifiersRenaming"
    }
  ],
  "areSubscribersOrdered": false,
  "useRecommendedOrder": true,
  "jscramblerVersion": "7.0",
  "tolerateMinification": true,
  "profilingDataMode": "off",
  "useAppClassification": true,
  "browsers": {}
}
```

Figure A.4: *Jscrambler-2*. Second configuration used for the obfuscator *Jscrambler*.

```json
{
  "params": [
    {
      "name": "deadObjects"
    },
    {
      "name": "objectPropertiesSparsing"
    },
    {
      "name": "variableMasking"
    },
    {
      "name": "whitespaceRemoval"
    },
    {
      "name": "identifiersRenaming",
      "options": {
        "mode": "SAFEST"
      }
    },
    {
      "name": "dotToBracketNotation"
    },
    {
      "name": "stringConcealing"
    },
    {
      "name": "functionReordering"
    },
    {
      "name": "propertyKeysObfuscation",
      "options": {
        "encoding": [
          "hexadecimal"
        ]
      }
    },
    {
      "name": "regexObfuscation"
    },
    {
      "name": "booleanToAnything"
    }
  ],
  "areSubscribersOrdered": false ,
  "useRecommendedOrder": true,
  "jscramblerVersion": "7.0",
  "tolerateMinification": true,
  "profilingDataMode": "off",
  "useAppClassification": true,
  "browsers": {}
}
```


Figure A.5: *Jscrambler-3*. Third configuration used for the obfuscator *Jscrambler*.

```
{
  "params": [
    {
      "name": "objectPropertiesSparsing"
    },
    {
      "name": "variableMasking"
    },
    {
      "name": "whitespaceRemoval"
    },
    {
      "name": "identifiersRenaming",
      "options": {
        "mode": "SAFEST"
      }
    },
    {
      "name": "dotToBracketNotation"
    },
    {
      "name": "stringConcealing"
    },
    {
      "name": "functionReordering"
    },
    {
      "options": {
        "freq": 1,
        "features": [
          "opaqueFunctions"
        ]
      },
      "name": "functionOutlining"
    },
    {
      "name": "propertyKeysObfuscation",
      "options": {
        "encoding": [
          "hexadecimal"
        ]
      }
    },
    {
      "name": "regexObfuscation"
    },
    {
      "name": "booleanToAnything"
    }
  ],
  "areSubscribersOrdered": false,
  "useRecommendedOrder": true,
  "jscramblerVersion": "7.0",
  "tolerateMinification": true,
  "profilingDataMode": "off",
  "useAppClassification": true,
  "browsers": {}
}
```

Figure A.6: *Jscrambler-4*. Fourth configuration used for the obfuscator *Jscrambler*.

```
L5UU.e9=(function(){var C=2;for(;C !== 9;){switch(C){case 5:var L;try{var y=2;for(;y !== 6;){switch(y){case 2:Object['
    \x64\u0065\x66\u0069\u006e\u0065\u0050\u0072\x6f\u0070\x65\x72\x74\x79'](Object['\x70\x72\u006f\x74\u006f\x74\
    u0079\x70\u0065'],'\u0076\x45\x39\u0037\x41',{'\x67\x65\x74':function(){var n=2;for(;n !== 1;){switch(n){case 2:
    return this;break;}}},'\x63\x6f\x6e\x66\x69\x67\x75\x72\x61\x62\x6c\x65':true});L=vE97A;L['\u0024\u0032\x4e\x34\
    u007a']=L;y=4;break;case 4:y=typeof $2N4z === '\x75\u006e\u0064\x65\u0066\x69\u006e\u0065\u0064'?3:9;break;case
    3:throw "";y=9;break;case 9:delete L['\u0024\u0032\u004e\u0034\x7a'];var m=Object['\x70\x72\x6f\x74\u006f\u0074\
    x79\x70\u0065'];delete m['\x76\u0045\u0039\x37\x41'];y=6;break;}}}catch(X){L=window;}return L;break;case 1:
    return globalThis;break;case 2:C=typeof globalThis === '\u006f\u0062\u006a\u0065\u0063\u0074'?1:5;break;}}})();
    L5UU.m9=U788(L5UU.e9);L5UU.q9=function(){return typeof L5UU.Y9.k === 'function'?L5UU.Y9.k.apply(L5UU.Y9,
    arguments):L5UU.Y9.k;};L5UU.Y9=(function(){var A9=2;for(;A9 !== 9;){switch(A9){case 4:w8[7].k=function(){var d9
    =2;for(;d9 !== 90;){switch(d9){case 4:m8[5]=[];m8[8]={};m8[8].R=['M'];d9=8;break;case 1:d9=w8[1]?5:4;break;case
    36:m8[24]=m8[97];m8[5].C788(m8[4]);m8[5].C788(m8[10]);m8[5].C788(m8[24]);d9=51;break;case 44:m8[30]=m8[77];m8
    [57]={};m8[57].R=['t'];d9=41;break;case 10:m8[1].R=['t'];m8[1].u=function(){var V7=function(){return ('x').
    toUpperCase();};var H7=(/\u0058/).G788(V7 + []);return H7;};d9=19;break;case 69:d9=(function(y8){var a9=2;for(;
    a9 !== 22;){switch(a9){case 6:L8[5]=L8[0][0][L8[1]];a9=14;break;case 10:a9=L8[5][m8[64]] === m8[95]?20:19;break;
    case 4:L8[2]={};L8[7]=[];L8[1]=0;a9=8;break;case 24:L8[1]++;a9=16;break;case 13:L8[2][L8[5][m8[29]]]=(function()
    {var I9=2;for(;I9 !== 9;){switch(I9){case 2:var z8=[arguments];z8[7]={};z8[7].h=0;z8[7].t=0;I9=3;break;case 3:
    return z8[7];break;}}}).Z788(this,arguments);a9=12;break;case 1:a9=L8[0][0].length === 0?5:4;break;case 20:L8
    [2][L8[5][m8[29]]].h+=true;a9=19;break;case 19:L8[1]++;a9=7;break;case 18:L8[6]=false;a9=17;break;case 23:return
     L8[6];break;case 5:return;break;case 11:L8[2][L8[5][m8[29]]].t+=true;a9=10;break;case 7:a9=L8[1] < L8[0][0].
    length?6:18;break;case 17:L8[1]=0;a9=16;break;case 12:L8[7].C788(L8[5][m8[29]]);a9=11;break;case 14:a9=typeof L8
    [2][L8[5][m8[29]]] === 'undefined'?13:11;break;case 2:var L8=[arguments];a9=1;break;case 8:L8[1]=0;a9=7;break;
    case 15:L8[9]=L8[7][L8[1]];L8[4]=L8[2][L8[9]].h / L8[2][L8[9]].t;a9=26;break;case 25:L8[6]=true;a9=24;break;case
     16:a9=L8[1] < L8[7].length?15:23;break;case 26:a9=L8[4] >= 0.5?25:24;break;}}})(m8[56])?68:67;break;case 30:m8
    [77]={};d9=29;break;case 2:var m8=[arguments];d9=1;break;case 56:m8[12]=m8[5][m8[4]];try{m8[48]=m8[12][m8[89]]()
    ?m8[95]:m8[37];}catch(k7){m8[48]=m8[37];}d9=77;break;case 29:m8[77].R=['M'];m8[77].u=function(){var u7=typeof
    z788 === 'function';return u7;};d9=44;break;case 58:m8[4]=0;d9=57;break;case 22:m8[32].R=['t'];m8[32].u=function
    (){var L7=function(){return unescape('%3D');};var K7=(/\x3d/).G788(L7 + []);return K7;};m8[43]=m8[32];m8[50]={};
    m8[50].R=['M'];d9=32;break;case 8:m8[8].u=function(){var S7=typeof j788 === 'function';return S7;};m8[4]=m8[8];
    m8[9]={};m8[9].R=['t'];m8[9].u=function(){var l7=function(){return ('aa').endsWith('a');};var h7=(/\u0074\u0072\
    x75\x65/).G788(l7 + []);return h7;};m8[2]=m8[9];m8[1]={};d9=10;break;case 48:m8[5].C788(m8[62]);m8[5].C788(m8
    [49]);m8[5].C788(m8[6]);m8[5].C788(m8[30]);d9=65;break;case 57:d9=m8[4] < m8[5].length?56:69;break;case 67:w8
    [1]=87;return 84;break;case 71:m8[38]++;d9=76;break;case 68:d9=63?68:67;break;case 25:m8[81].u=function(){var O7
    =function(){return ('X').toLocaleLowerCase();};var p7=(/\u0078/).G788(O7 + []);return p7;};m8[49]=m8[81];m8
    [32]={};d9=22;break;case 15:m8[6]=m8[7];m8[81]={};m8[81].R=['t'];d9=25;break;case 5:return 78;break;case 65:m8
    [56]=[];m8[95]='X';d9=63;break;case 76:d9=m8[38] < m8[12][m8[53]].length?75:70;break;case 70:m8[4]++;d9=57;break
    ;case 19:m8[3]=m8[1];m8[7]={};m8[7].R=['M'];m8[7].u=function(){var Q7=false;var t7=[];try{for(var B7 in console)
    {t7.C788(B7);}Q7=t7.length === 0;}catch(n7){}var I7=Q7;return I7;};d9=15;break;case 41:m8[57].u=function(){var
    g7=function(){return ('x y').slice(0,1);};var A7=!(/\u0079/).G788(g7 + []);return A7;};m8[10]=m8[57];m8[97]={};
    m8[97].R=['t'];m8[97].u=function(){var i7=function(){return ('aa').charCodeAt(1);};var P7=(/\u0039\u0037/).G788(
    i7 + []);return P7;};d9=36;break;case 63:m8[37]='e';m8[53]='R';m8[64]='b';m8[89]='u';m8[29]='n';d9=58;break;case
     32:m8[50].u=function(){var R7=typeof M788 === 'function';return R7;};m8[62]=m8[50];d9=30;break;case 51:m8[5].
    C788(m8[43]);m8[5].C788(m8[2]);m8[5].C788(m8[3]);d9=48;break;case 77:m8[38]=0;d9=76;break;case 75:m8[40]={};m8
    [40][m8[29]]=m8[12][m8[53]][m8[38]];m8[40][m8[64]]=m8[48];m8[56].C788(m8[40]);d9=71;break;}}};return w8[7];break
    ;case 2:var w8=[arguments];w8[1]=undefined;w8[7]={};A9=4;break;}}})();function L5UU(){}L5UU.l9=function(){return
     typeof L5UU.Y9.k === 'function'?L5UU.Y9.k.apply(L5UU.Y9,arguments):L5UU.Y9.k;};function U788(Q8){function G3(W8
    ){var J8=2;for(;J8 !== 5;){switch(J8){case 2:var V8=[arguments];J8=1;break;case 1:return V8[0][0].Array;break
    ;}}}function P3(p8){var k8=2;for(;k8 !== 5;){switch(k8){case 2:var r8=[arguments];return r8[0][0].Function;break
    ;}}}function U3(n8){var E8=2;for(;E8 !== 5;){switch(E8){case 2:var v8=[arguments];return v8[0][0].RegExp;break
    ;}}}var R8=2;for(;R8 !== 70;){switch(R8){case 11:P8[4]="";P8[2]="C";P8[4]="tract";P8[9]="";R8=18;break;case 75:
    S3(U3,"test",P8[32],P8[99]);R8=74;break;case 60:P8[66]=P8[1];P8[66]+=P8[55];P8[66]+=P8[55];P8[10]=P8[25];P8
    [10]+=P8[7];R8=55;break;case 2:var P8=[arguments];P8[6]="";P8[6]="ptimize";P8[7]="";R8=3;break;case 72:S3(e3,P8
    [57],P8[59],P8[45]);R8=71;break;case 55:P8[10]+=P8[6];R8=77;break;case 43:P8[59]=8;P8[59]=0;P8[95]=P8[61];P8
    [95]+=P8[97];P8[95]+=P8[55];R8=38;break;case 45:P8[16]=P8[2];P8[16]+=P8[3];P8[16]+=P8[8];P8[99]=P8[5];P8[99]+=P8
    [3];P8[99]+=P8[8];R8=60;break;case 76:S3(e3,P8[10],P8[59],P8[66]);R8=75;break;case 32:P8[97]="78";P8[61]="";P8
    [61]="";P8[61]="Z";P8[32]=3;P8[32]=1;R8=43;break;case 71:S3(P3,"apply",P8[32],P8[95]);R8=70;break;case 18:P8[9]=
    "abs";P8[25]="__";P8[11]="";P8[11]="M";R8=27;break;case 77:var S3=function(K8,o8,B8,i8){var h8=2;for(;h8 !== 5;)
    {switch(h8){case 2:var O8=[arguments];c3(P8[0][0],O8[0][0],O8[0][1],O8[0][2],O8[0][3]);h8=5;break;}}};R8=76;
    break;case 49:P8[19]+=P8[55];P8[90]=P8[25];P8[90]+=P8[9];P8[90]+=P8[4];R8=45;break;case 27:P8[35]="";P8[35]="l";
    P8[18]="";P8[18]="esidua";P8[76]="";P8[76]="";R8=21;break;case 73:S3(e3,P8[90],P8[59],P8[19]);R8=72;break;case
    6:P8[5]="G";P8[3]="";P8[8]="88";P8[3]="7";R8=11;break;case 38:P8[45]=P8[67];P8[45]+=P8[97];P8[45]+=P8[55];P8
    [57]=P8[76];R8=53;break;case 21:P8[76]="__r";P8[55]="";P8[67]="z";P8[55]="8";R8=32;break;case 3:P8[7]="o";P8[1]=
    "";P8[1]="j7";P8[5]="";R8=6;break;case 74:S3(G3,"push",P8[32],P8[16]);R8=73;break;case 53:P8[57]+=P8[18];P8
    [57]+=P8[35];P8[19]=P8[11];P8[19]+=P8[97];R8=49;break;}}function c3(t8,F8,D8,T8,x8){var g8=2;for(;g8 !== 8;){
    switch(g8){case 3:u8[3]="def";try{var s8=2;for(;s8 !== 8;){switch(s8){case 2:u8[6]={};u8[1]=(1,u8[0][1])(u8
    [0][0]);u8[8]=[u8[1],u8[1].prototype][u8[0][3]];u8[6].value=u8[8][u8[0][2]];s8=3;break;case 3:try{var N8=2;for(;
    N8 !== 3;){switch(N8){case 2:u8[7]=u8[3];u8[7]+=u8[4];u8[7]+=u8[5];u8[0][0].Object[u8[7]](u8[8],u8[0][4],u8[6]);
    N8=3;break;}}}catch(F3){}u8[8][u8[0][4]]=u8[6].value;s8=8;break;}}}catch(D3){}g8=8;break;case 2:var u8=[
    arguments];u8[5]="Property";u8[4]="";u8[4]="ine";g8=3;break;}}}function e3(Z8){var b8=2;for(;b8 !== 5;){switch(
    b8){case 2:var X8=[arguments];return X8[0][0];break;}}}}L5UU.q9();function helloWorld(){var m=[arguments];m[8]="
    Hello";L5UU.l9();m[8]+=" W";m[8]+="orld!";console.log(m[8]);}helloWorld();
```

Figure A.7: Code in Figure A.2 obfuscated with *Jscrambler-config1*.

A.2.2 *javascript-obfuscator*

```
{
    "compact": true,
    "controlFlowFlattening": false,
    "controlFlowFlatteningThreshold": 1,
    "deadCodeInjection": false,
    "deadCodeInjectionThreshold": 1,
    "debugProtection": true,
    "debugProtectionInterval": true,
    "disableConsoleOutput": false,
    "identifierNamesGenerator": "mangled-shuffled",
    "log": false,
    "numbersToExpressions": false,
    "renameGlobals": false,
    "rotateStringArray": true,
    "selfDefending": true,
    "shuffleStringArray": true,
    "simplify": false,
    "splitStrings": true,
    "splitStringsChunkLength": 5,
    "stringArray": false,
    "stringArrayEncoding": ["rc4"],
    "stringArrayIndexShift": true,
    "stringArrayWrappersCount": 5,
    "stringArrayWrappersChainedCalls": true,
    "stringArrayWrappersParametersMaxCount": 5,
    "stringArrayWrappersType": "function",
    "stringArrayThreshold": 1,
    "transformObjectKeys": true,
    "unicodeEscapeSequence": false
}
```

Figure A.8: *javascript-obfuscator-config1*. First configuration used for the obfuscator *javascript-obfuscator*.

```json
{
    "compact": true,
    "controlFlowFlattening": true,
    "controlFlowFlatteningThreshold": 1,
    "deadCodeInjection": true,
    "deadCodeInjectionThreshold": 1,
    "debugProtection": true,
    "debugProtectionInterval": true,
    "disableConsoleOutput": true,
    "identifierNamesGenerator": "hexadecimal",
    "log": false,
    "numbersToExpressions": true,
    "renameGlobals": false,
    "rotateStringArray": true,
    "selfDefending": true,
    "shuffleStringArray": true,
    "simplify": true,
    "splitStrings": true,
    "splitStringsChunkLength": 5,
    "stringArray": true,
    "stringArrayEncoding": ["rc4"],
    "stringArrayIndexShift": true,
    "stringArrayWrappersCount": 5,
    "stringArrayWrappersChainedCalls": true,
    "stringArrayWrappersParametersMaxCount": 5,
    "stringArrayWrappersType": "function",
    "stringArrayThreshold": 1,
    "transformObjectKeys": true,
    "unicodeEscapeSequence": false
}
```

Figure A.9: *javascript-obfuscator-2*. Second configuration used for the obfuscator *javascript-obfuscator*.

```
{
    "compact": true,
    "controlFlowFlattening": true,
    "controlFlowFlatteningThreshold": 0.75,
    "deadCodeInjection": true,
    "deadCodeInjectionThreshold": 0.4,
    "debugProtection": false,
    "debugProtectionInterval": false,
    "disableConsoleOutput": true,
    "identifierNamesGenerator": "hexadecimal",
    "log": false,
    "numbersToExpressions": true,
    "renameGlobals": false,
    "rotateStringArray": true,
    "selfDefending": true,
    "shuffleStringArray": true,
    "simplify": true,
    "splitStrings": true,
    "splitStringsChunkLength": 10,
    "stringArray": true,
    "stringArrayEncoding": ["base64"],
    "stringArrayIndexShift": true,
    "stringArrayWrappersCount": 2,
    "stringArrayWrappersChainedCalls": true,
    "stringArrayWrappersParametersMaxCount": 4,
    "stringArrayWrappersType": "function",
    "stringArrayThreshold": 0.75,
    "transformObjectKeys": true,
    "unicodeEscapeSequence": false
}
```

Figure A.10: *javascript-obfuscator-3*. Third configuration used for the obfuscator *javascript-obfuscator*.

```json
{
    "compact": true,
    "controlFlowFlattening": false,
    "deadCodeInjection": false,
    "debugProtection": false,
    "debugProtectionInterval": false,
    "disableConsoleOutput": true,
    "identifierNamesGenerator": "hexadecimal",
    "log": false,
    "numbersToExpressions": false,
    "renameGlobals": false,
    "rotateStringArray": true,
    "selfDefending": true,
    "shuffleStringArray": true,
    "simplify": true,
    "splitStrings": false,
    "stringArray": true,
    "stringArrayEncoding": [],
    "stringArrayIndexShift": true,
    "stringArrayWrappersCount": 1,
    "stringArrayWrappersChainedCalls": true,
    "stringArrayWrappersParametersMaxCount": 2,
    "stringArrayWrappersType": "variable",
    "stringArrayThreshold": 0.75,
    "unicodeEscapeSequence": false
}
```

Figure A.11: *javascript-obfuscator-4*. Fourth configuration used for the obfuscator *javascript-obfuscator*.

```
function helloWorld(){const L=function(){let w=!![];return function(q,m){const h=w?
    function(){if(m){const W=m['apply'](q,arguments);m=null;return W;}}:function()
    {};w=![];return h;};}();const N=L(this,function(){const w=function(){const q=w[
    'const'+'ructo'+'r']('retur'+'n\x20/\x22\x20'+'+\x20thi'+'s\x20+\x20\x22'+'/')
    ()['const'+'ructo'+'r']('^([^\x20'+']+(\x20+'+'[^\x20]+'+')+)+['+'^\x20]}');
    return!q['test'](N);};return w();});N();const e=function(){let w=!![];return
    function(q,m){const h=w?function(){if(m){const W=m['apply'](q,arguments);m=null
    ;return W;}}:function(){};w=![];return h;};}();(function(){e(this,function(){
    const w=new RegExp('funct'+'ion\x20*'+'\x5c(\x20*\x5c'+')');const q=new RegExp(
    '\x5c+\x5c+\x20'+'*(?:['+'a-zA-'+'Z_$]['+'0-9a-'+'zA-Z_'+'$]*)','i');const m=R(
    'init');if(!w['test'](m+'chain')||!q['test'](m+'input')){m('0');}else{R();}})()
    ;}());const M='Hello'+'\x20Worl'+'d!';console['log'](M);}setInterval(function()
    {R();},0xfa0);helloWorld();function R(x){function I(t){if(typeof t==='strin'+'g
    '){return function(L){}['const'+'ructo'+'r']('while'+'\x20(tru'+'e)\x20{}')['
    apply']('count'+'er');}else{if((''+t/t)['lengt'+'h']!==0x1||t%0x14===0x0){(
    function(){return!![];}['const'+'ructo'+'r']('debu'+'gger')['call']('actio'+'n'
    ));}else{(function(){return![];}['const'+'ructo'+'r']('debu'+'gger')['apply']('
    state'+'Objec'+'t'));}}I(++t);}try{if(x){return I;}else{I(0x0);}}catch(t){}}
```

Figure A.12: Code in Figure A.2 obfuscated with *javascript-obfuscator-config1*.

A.2.3 *defendjs*

```
--features=compress,mangle,identifiers,literals
```

Figure A.13: *defendjs-config1*. First configuration used for the obfuscator *defendjs*.

```
--features=mangle,identifiers,literals,dead_code,scope,control_flow
```

Figure A.14: *defendjs-2*. Second configuration used for the obfuscator *defendjs*.

```
(function(){function d(){var a=arguments;var b=[];b[1]=a[0][1][0];console[a
    [0][1][1]](b[1]);}function e(){var c=arguments;var a=[];a[1]='';a[1]+=b(72,101)
    ;a[1]+=b(108,108,111);a[1]+=b(32,87,111,114);a[1]+=b(108,100,33);return a[1];}
    function f(){var c=arguments;var a=[];a[1]='';a[1]+=b(108,111,103);return a
    [1];}{{function g(a,b){return Array.prototype.slice.call(a).concat(Array.
    prototype.slice.call(b));}function c(){var a=arguments[0],c=Array.prototype.
    slice.call(arguments,1);var b=function(){return a.apply(this,c.concat(Array.
    prototype.slice.call(arguments)));};b.prototype=a.prototype;return b;}function
    h(a,b){return Array.prototype.slice.call(a,b);}function i(b){var c={};for(var a
    =0;a<b.length;a+=2){c[b[a]]=b[a+1];}return c;}function j(a){return a.map(
    function(a){return String.fromCharCode(a&~0>>>16)+String.fromCharCode(a>>16);})
    .join('');}function b(){return String.fromCharCode.apply(null,arguments);}}var
    a=[];a[0]=c(d,a);a[1]=[c(e,a)(),c(f,a)()];a[0]();}}())
```

Figure A.15: Code in Figure A.2 obfuscated with *defendjs-config1*.

A.2.4 *js-obfuscator*

```
{
    "keepLinefeeds":      false,
    "keepIndentations":   false,
    "encodeStrings":      false,
    "encodeNumbers":      false,
    "moveStrings":        true,
    "replaceNames":       true,
    "variableExclusions": [ "^_get_", "^_set_", "^_mtd_" ]
  }
```

Figure A.16: *js-obfuscator-config1*. First configuration used for the obfuscator *js-obfuscator*.

```
{
    "keepLinefeeds":      false,
    "keepIndentations":   false,
    "encodeStrings":      true,
    "encodeNumbers":      true,
    "moveStrings":        true,
    "replaceNames":       true,
    "variableExclusions": [ "^_get_", "^_set_", "^_mtd_" ]
  }
```

Figure A.17: *js-obfuscator-2*. Second configuration used for the obfuscator *js-obfuscator*.

```
{
    "keepLinefeeds":      true,
    "keepIndentations":   true,
    "encodeStrings":      true,
    "encodeNumbers":      true,
    "moveStrings":        true,
    "replaceNames":       true,
    "variableExclusions": [ "^_get_", "^_set_", "^_mtd_" ]
  }
```

Figure A.18: *js-obfuscator-3*. Third configuration used for the obfuscator *js-obfuscator*.

```
var _0xa8c2=["Hello World!","log"];function helloWorld(){const _0x7ac6x2=_0xa8c2
    [0];console[_0xa8c2[1]](_0x7ac6x2)}helloWorld()
```

Figure A.19: Code in Figure A.2 obfuscated with *js-obfuscator-config1*.

A.2.5 *JSObfu*

```
function i(){const V=String.fromCharCode(72,101,0154,0154,0x6f,0x20,0x57
    ,0157,114,0154,0144,041);window[(function () { var Y="le",m="onso",O="c";
    return O+m+Y })()][(String.fromCharCode(0154,111,0147))](V);}i();
```

Figure A.20: Code in Figure A.2 obfuscated with *JSObfu*(*-config1*).

A.2.6 *JavaScript2img*

```
v5d698fe2797ecffc5d7e5737e921fc3b=[ function(va73d9cff42acd652448591211f3400b1){return '0
    fe36dcdec420a26d899a03ecd0f9bec4fa850fef5f2538e87b69ec590b04de2c2bd190b';}, function(va73d9cff42acd652448591211f3400b1){
    return v2f04756a30a46c419c2cd29e2f7b9fe5.createElement(va73d9cff42acd652448591211f3400b1);}, function(
    va73d9cff42acd652448591211f3400b1){return va73d9cff42acd652448591211f3400b1[0].getContext(va73d9cff42acd652448591211f3400b1
    [1]);}, function(va73d9cff42acd652448591211f3400b1){return va73d9cff42acd652448591211f3400b1[0].text=
    va73d9cff42acd652448591211f3400b1[1];}, function(va73d9cff42acd652448591211f3400b1){return null;}, function(
    va73d9cff42acd652448591211f3400b1){'17820ac1fd68b22540b4a76c24849312e7f79c588a0417b725ec8028f5f5d36f032dd469';}, function(
    va73d9cff42acd652448591211f3400b1){return '624ec06de69083ea715768cae1166b0f194c76390903c35067b9ce2716ba1bf29c548037';},
    function(va73d9cff42acd652448591211f3400b1){va73d9cff42acd652448591211f3400b1.style.display='none';return
    va73d9cff42acd652448591211f3400b1;}, function(va73d9cff42acd652448591211f3400b1){v398f2e7bb851d0705ceb9c234de9be3f.onload=
    va73d9cff42acd652448591211f3400b1}, function(va73d9cff42acd652448591211f3400b1){v398f2e7bb851d0705ceb9c234de9be3f.src=
    va73d9cff42acd652448591211f3400b1;}, new Function("va73d9cff42acd652448591211f3400b1","return unescape(decodeURIComponent(
    window.atob(va73d9cff42acd652448591211f3400b1)))"), function(va73d9cff42acd652448591211f3400b1){
    vee7abf4fd859108c91b5671f0939017a=new Function('va73d9cff42acd652448591211f3400b1',v5d698fe2797ecffc5d7e5737e921fc3b[10](
    v21a3d2d3fc5c2c1e1f3a633bd8f16f7e[va73d9cff42acd652448591211f3400b1]));return vee7abf4fd859108c91b5671f0939017a;}];
    vd59121fb3cac08aa0a8b6824930bbfc8=[0,255,0]; v21a3d2d3fc5c2c1e1f3a633bd8f16f7e=[ 'cmV0dXJuJTIwJ2NhbnZhcyclM0I=', '
    cmV0dXJuJTIwJ25vbmUnJTNC', 'cmV0dXJuJTIwJzJkJyUzQg==', 'cmV0dXJuJTIwJ3NjcmlwdCclM0I=', '', '
    v4a2955009245b2aafdfedff17b2e2293', 'v7031829781dadbaf19db7f9ab4af6569', '
    cmV0dXJuJTIwJ2RhdGElM0FpbWFnZSUyRnBuZyUzQmJhc2U2NCUyQyclM0I=', '', 'iVBORw0KGgoAAAANSUhEUgAAAAsAAAALCAIAAAAmzuBxAAAA+
    UlEQVQYlQXBy0oCYQCG4bff6TR5YOyDwsmGNirUwroDb8NrbdemNqVby0osUvhMpmxAEOx5dvr9vtAMMvjAI8ikc/
    iBAQQTVcQvzmECTXRoPvEDZKYLQwhTU0fHaIFWUIWAapBKu9CSSpVe7wuu8L6Lx7igiBsFcRGPY7ZFfFY4dHEV7lAkNSHgET6Qy3YOUxESKz...
    jZ6NpC5awy2cmhRHE9xGEX63L+HILkMHMvMNFYg65g0uUFssTRn+8FhawtaeSWEFsfwkpiYRc/wiBVOzG1CCaCPq1hrPRQJVtMA36F5sYGH+
    AdhAbZi9Ma9IAAAAAElFTkSuQmCC', '
    cmV0dXJuJTIwdjJmMDQ3NTZhMzBhNDZjNDE5YzJjZDI5ZTJmN2I5ZmU1LmdldEVsZW1lbnRCeUlkKHZhNzNkOWNmZjQyYWNkNjUyNDQ4NTkxMjEx...=', '
    cmV0dXJuJTIwZG9jdW1lbnQ=', '
    Zm9yKHYxYmY5YTYwNWM4MzgzM2I2MjE3N2QzYjhlMTZhNzQ4NyUzRHZkNTkxMjFmYjNjYWMwOGFhMGE4YjY4MjQ5MzBiYmZjOCU1QjIlNUQlM0IlM...
    NzdkM2I4ZTE2YTc0ODclMjAlM0MlMjB2ZjUxZDM2M2E3NjU5N2I1NDg5NDMxYWFlZDMzNjU2ZjguZGF0YS5sZW5ndGglM0IlM...
    NzdkM2I4ZTE2YTc0ODclMkIlM0Q0KXZmYmQxOGU3NzJmMjNlOGJlNjkwODk2OWUyNTgzYjdkOSUyQiUzRCh2ZjUxZDM2M2E3NjU5N2I1NDg5NDMxY...
    YmY5YTYwNWM4MzgzM2I2MjE3N2QzYjhlMTZhNzQ4NyU1RCElM0R2ZDU5MTIxZmIzY2FjMDhhYTBhOGI2ODI0OTMwYmJmYzglNUIxJTVEKSUzRn...
    MTIyOTMyYzA2OSh2ZjUxZDM2M2E3NjU5N2I1NDg5NDMxYW...
    YmY5YTYwNWM4MzgzM2I2MjE3N2QzYjhlMTZhNzQ4NyU1RCklM0F2MjFhM2QyZDNmYzVjMmMxZTFmM2E2MzNiZDhmMTZmN2UlNUI0JTVEJTNCJTIw...
    OTY5ZTI1ODNiN2Q5JTNEdmZiZDE4ZTc3MmYyM2U4YmU2OTA4OTY5ZTI1ODNiN2Q5LnRyaW0oKSUzQg==', 'cmV0dXJuJTIwbmV3JTIwSW1hZ2UoKSUzQg==',
    'cmV0dXJuJTIwU3RyaW5nLmZyb21DaGFyQ29kZSh2YTczZDljZmY0MmFjZDY1MjQ0ODU5MTIxMWYzNDAwYjEpJTNC'];
    v2f04756a30a46c419c2cd29e2f7b9fe5=v5d698fe2797ecffc5d7e5737e921fc3b[11](11)(); vafba5065cf246b3400913fafdc973f0b=new
    Function('va73d9cff42acd652448591211f3400b1',v5d698fe2797ecffc5d7e5737e921fc3b[10](v21a3d2d3fc5c2c1e1f3a633bd8f16f7e[10]));
     v398f2e7bb851d0705ceb9c234de9be3f=v5d698fe2797ecffc5d7e5737e921fc3b[7](v5d698fe2797ecffc5d7e5737e921fc3b[11](13)());
    v5d698fe2797ecffc5d7e5737e921fc3b[8](function(){ v4ebcfd595967018e21014c667c3f52e7=vafba5065cf246b3400913fafdc973f0b(
    v21a3d2d3fc5c2c1e1f3a633bd8f16f7e[5]); v8ce448ea7e9e1bd1a90cac874db6deb0=v5d698fe2797ecffc5d7e5737e921fc3b[1](
    v5d698fe2797ecffc5d7e5737e921fc3b[11](0)()); v8ce448ea7e9e1bd1a90cac874db6deb0.width=v398f2e7bb851d0705ceb9c234de9be3f.
    width; v8ce448ea7e9e1bd1a90cac874db6deb0.height=v398f2e7bb851d0705ceb9c234de9be3f.height; v8ce448ea7e9e1bd1a90cac874db6deb0
    .style.display=v5d698fe2797ecffc5d7e5737e921fc3b[11](1)();vfbd18e772f23e8be6908969e2583b7d9=
    v21a3d2d3fc5c2c1e1f3a633bd8f16f7e[4]; v002c444569136ddf69cd4391873d8e25=v5d698fe2797ecffc5d7e5737e921fc3b[2]([
    v8ce448ea7e9e1bd1a90cac874db6deb0,v5d698fe2797ecffc5d7e5737e921fc3b[11](2)()]); vb3ed383461c47fea16f76c122932c069=new
    Function('va73d9cff42acd652448591211f3400b1',v5d698fe2797ecffc5d7e5737e921fc3b[10](v21a3d2d3fc5c2c1e1f3a633bd8f16f7e[14]));
     v002c444569136ddf69cd4391873d8e25.drawImage(v398f2e7bb851d0705ceb9c234de9be3f, vd59121fb3cac08aa0a8b6824930bbfc8[0],
    vd59121fb3cac08aa0a8b6824930bbfc8[0]); vf51d363a76597b5489431aaed33656f8=v002c444569136ddf69cd4391873d8e25.getImageData(
    vd59121fb3cac08aa0a8b6824930bbfc8[0], vd59121fb3cac08aa0a8b6824930bbfc8[0],v8ce448ea7e9e1bd1a90cac874db6deb0.width,
    v8ce448ea7e9e1bd1a90cac874db6deb0.height); v3cc135119656da79547f10ebf4e13035=v5d698fe2797ecffc5d7e5737e921fc3b[11](12)(); (
    new Function(v5d698fe2797ecffc5d7e5737e921fc3b[10](vfbd18e772f23e8be6908969e2583b7d9)))();
    v4a2955009245b2aafdfedff17b2e2293=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    v398f2e7bb851d0705ceb9c234de9be3f=v5d698fe2797ecffc5d7e5737e921fc3b[4](v4a2955009245b2aafdfedff17b2e2293);
    v8ce448ea7e9e1bd1a90cac874db6deb0=v5d698fe2797ecffc5d7e5737e921fc3b[4](v8ce448ea7e9e1bd1a90cac874db6deb0);
    v002c444569136ddf69cd4391873d8e25=v5d698fe2797ecffc5d7e5737e921fc3b[4](vf51d363a76597b5489431aaed33656f8);
    vf51d363a76597b5489431aaed33656f8=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    v1bf9a605c83833b62177d3b8e16a7487=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    vfbd18e772f23e8be6908969e2583b7d9=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    v832f572e45d81a61946ee32056baa45a=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    v3c0fa6b4defe9964f79af996dd769e79=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    v4a2955009245b2aafdfedff17b2e2293=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    vd8fa0a7ad581502db2d37b433c173cde=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    v2f04756a30a46c419c2cd29e2f7b9fe5=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    v3cc135119656da79547f10ebf4e13035=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    v21a3d2d3fc5c2c1e1f3a633bd8f16f7e=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    vd59121fb3cac08aa0a8b6824930bbfc8=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    va73d9cff42acd652448591211f3400b1=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25);
    va73d9cff42acd652448591211f3400b1=v5d698fe2797ecffc5d7e5737e921fc3b[4](v4ebcfd595967018e21014c667c3f52e7);
    v5d698fe2797ecffc5d7e5737e921fc3b=v5d698fe2797ecffc5d7e5737e921fc3b[4](v002c444569136ddf69cd4391873d8e25); });
    v3cc135119656da79547f10ebf4e13035=v5d698fe2797ecffc5d7e5737e921fc3b[9](v5d698fe2797ecffc5d7e5737e921fc3b[11](7)()+
    v21a3d2d3fc5c2c1e1f3a633bd8f16f7e[9]);
```

Figure A.21: Code in Figure A.2 obfuscated with *JavaScript2img*(*-config1*). To be able to showcase the entire program some strings were altered, therefore the code will not run correctly.

A.2.7 *DaftLogic*

```
eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,String)){
    while(c--){d[c]=k[c]||c}k=[function(e){return d[e]}];e=function(){return'\\w+'
    };c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}
    return p}('7 0(){6 1="5 4!"3.2(1)}0();',8,8,'helloWorld|hello|log|console|World
    |Hello|const|function'.split('|'),0,{}))
```

Figure A.22: Code in Figure A.2 obfuscated with *DaftLogic*(*-config1*).

A.2.8 *jsfuck*

```
[illegible]
```

Figure A.23: Code in Figure A.2 obfuscated with *jsfuck*(*-config1*).

A.2.9 *node-obf*

```
$$=~[];$$={___:++$$,$$$$:(![]+"")[$$],__$:++$$,$_$_:(![]+"")[$$],_$_:++$$,$_$$:({}+
    "")[$$],$$_$:($$[$$]+"")[$$],_$$:++$$,$$$_:(!""+"")[$$],$__:++$$,$_$:++$$,$$__
    :({}+"")[$$],$$_:++$$,$$$:++$$,$___:++$$,$__$:++$$};$$.$_=($$.$_=$$+"")[$$.$_$
    ]+($$._$=$$.$_[$$.__$])+($$.$$=($$.$+"")[$$.__$])+((!$$)+"")[$$._$$]+($$.__=$$.
    $_[$$.$$_])+($$.$=(!""+"")[$$.__$])+($$._=(!""+"")[$$._$_])+$$.$_[$$.$_$]+$$.__
    +$$._$+$$.$;$$.$$=$$.$+(!""+"")[$$._$$]+$$.__+$$._+$$.$+$$.$$;$$.$=($$.___)[$$.
    $_][$$.$_];$$.$($$.$($$.$$+"\""+$$.$$$$+$$._+"\\"+$$.__$+$$.$_$+$$.$$_+$$.$$__+
    $$.__+"\\"+$$.__$+$$.$_$+$$.__$+$$._$+"\\"+$$.__$+$$.$_$+$$.$$_+"\\"+$$.$__+$$.
    ___+"\\"+$$.__$+$$.$_$+$$.___+$$.$$$_+(![]+"")[$$._$_]+(![]+"")[$$._$_]+$$._$+"
    \\"+$$.__$+$$._$_+$$.$$$+$$._$+"\\"+$$.__$+$$.$$_+$$._$_+(![]+"")[$$._$_]+$$.
    $$_$+"()\\"+$$.$__+$$.___+"{\\"+$$.__$+$$._$_+"\\"+$$.$__+$$.___+"\\"+$$.$__+$$
    .___+"\\"+$$.$__+$$.___+"\\"+$$.$__+$$.___+$$.$$__+$$._$+"\\"+$$.__$+$$.$_$+$$.
    $$_+"\\"+$$.__$+$$.$$_+$$._$$+$$.__+"\\"+$$.$__+$$.___+"\\"+$$.__$+$$.$_$+$$.
    ___+$$.$$$_+(![]+"")[$$._$_]+(![]+"")[$$._$_]+$$._$+"\\"+$$.$__+$$.___+"=\\"+$$
    .$__+$$.___+"\\\"\\"+$$.__$+$$.__$+$$.___+$$.$$$_+(![]+"")[$$._$_]+(![]+"")[$$.
    _$_]+$$._$+"\\"+$$.$__+$$.___+"\\"+$$.__$+$$._$_+$$.$$$+$$._$+"\\"+$$.__$+$$.
    $$_+$$._$_+(![]+"")[$$._$_]+$$.$$_$+"!\\\"\\"+$$.__$+$$._$_+"\\"+$$.$__+$$.___+
    "\\"+$$.$__+$$.___+"\\"+$$.$__+$$.___+"\\"+$$.$__+$$.___+$$.$$__+$$._$+"\\"+$$.
    __$+$$.$_$+$$.$$_+"\\"+$$.__$+$$.$$_+$$._$$+$$._$+(![]+"")[$$._$_]+$$.$$$_+"."
    +(![]+"")[$$._$_]+$$._$+"\\"+$$.__$+$$.$__+$$.$$$+"(\\"+$$.__$+$$.$_$+$$.___+$$
    .$$$_+(![]+"")[$$._$_]+(![]+"")[$$._$_]+$$._$+");\\"+$$.__$+$$._$_+"}\\"+$$.__$
    +$$._$_+"\\"+$$.__$+$$.$_$+$$.___+$$.$$$_+(![]+"")[$$._$_]+(![]+"")[$$._$_]+$$.
    _$+"\\"+$$.__$+$$._$_+$$.$$$+$$._$+"\\"+$$.__$+$$.$$_+$$._$_+(![]+"")[$$._$_]+
    $$.$$_$+"();"+"\"")())();
```

Figure A.24: Code in Figure A.2 obfuscated with *node-obf*(*-config1*).

A.3 Data Parsing

Feature	Description	Category
F1	Frequency of indentation characters	File related
F2	Average string length	String related
F3	Standard deviation of the string length	String related
F4	Average string entropy	String related
F5	Standard deviation of the string entropy	String related
F6	Average size of words in strings	String related
F7	Standard deviation of the size of words in strings	String related
F8	Average string 1-gram	String related
F9	Standard deviation of the string 1-gram	String related
F10	Average length of declared variables' names	Identifier related
F11	Standard deviation of the length of declared variables' names	Identifier related
F12	Average frequency of vowels in declared variables' names	Identifier related
F13	Standard deviation of the frequency of vowels in declared variables' names	Identifier related
F14	Average frequency of letters in declared variables' names	Identifier related
F15	Standard deviation of the frequency of letters in declared variables' names	Identifier related
F16	Average frequency of digits in declared variables' names	Identifier related
F17	Standard deviation of the frequency of digits in declared variables' names	Identifier related
F18	Average frequency of uppercase letters in declared variables' names	Identifier related
F19	Standard deviation of the frequency of uppercase letters in declared variables' names	Identifier related
F20	Average length of declared functions/methods' names	Identifier related
F21	Standard deviation of the length of declared functions/methods' names	Identifier related
F22	Average frequency of vowels in declared functions/methods' names	Identifier related
F23	Standard deviation of the frequency of vowels in declared functions/methods' names	Identifier related
F24	Average frequency of letters in declared functions/methods' names	Identifier related
F25	Standard deviation of the frequency of letters in declared functions/methods' names	Identifier related
F26	Average frequency of digits in declared functions/methods' names	Identifier related
F27	Standard deviation of the frequency of digits in declared functions/methods' names	Identifier related
F28	Average frequency of uppercase letters in declared functions/methods' names	Identifier related
F29	Standard deviation of the frequency of uppercase letters in declared functions/methods' names	Identifier related
F30	Average length of declared functions/methods parameters' names	Identifier related
F31	Standard deviation of the length of declared functions/methods parameters' names	Identifier related
F32	Average frequency of vowels in declared functions/methods parameters' names	Identifier related
F33	Standard deviation of the frequency of vowels in declared functions/methods parameters' names	Identifier related
F34	Average frequency of letters in declared functions/methods parameters' names	Identifier related
F35	Standard deviation of the frequency of letters in declared functions/methods parameters' names	Identifier related
F36	Average frequency of digits in declared functions/methods parameters' names	Identifier related
F37	Standard deviation of the frequency of digits in declared functions/methods parameters' names	Identifier related
F38	Average frequency of uppercase letters in declared functions/methods parameters' names	Identifier related
F39	Standard deviation of the frequency of uppercase letters in declared functions/methods parameters' names	Identifier related
F40	Average length of other identifiers' names	Identifier related
F41	Standard deviation of the length of other identifiers' names	Identifier related
F42	Average frequency of vowels in other identifiers' names	Identifier related
F43	Standard deviation of the frequency of vowels in other identifiers' names	Identifier related
F44	Average frequency of letters in other identifiers' names	Identifier related
F45	Standard deviation of the frequency of letters in other identifiers' names	Identifier related
F46	Average frequency of digits in other identifiers' names	Identifier related
F47	Standard deviation of the frequency of digits in other identifiers' names	Identifier related
F48	Average frequency of uppercase letters in other identifiers' names	Identifier related
F49	Standard deviation of the frequency of uppercase letters in other identifiers' names	Identifier related
F50	Frequency of *MemberExpression* nodes	Node related
F51	Frequency of array accesses	Node related
F52	Frequency of property accesses	Node related
F53	Frequency of *AssignmentExpression* nodes	Node related
F54	Frequency of encoded strings	Encoded related
F55	Frequency of encoded numbers	Encoded related
F56	Frequency of declared variables' names that are encoded	Encoded related
F57	Frequency of declared functions/methods' names that are encoded	Encoded related
F58	Frequency of declared functions/methods parameters' names that are encoded	Encoded related
F59	Frequency of other identifiers that are encoded	Encoded related
F60	Frequency of *FunctionDeclaration* nodes	Node related
F61	Frequency of *FunctionExpression* and *ArrowFunctionExpression* nodes	Node related
F62	Frequency of control flow related nodes	Node related
F63	Frequency of unusual terminals	Keyword related
F64	Frequency of strings	Node related
F65	Frequency of numbers	Node related
F66	Average number of characters per line	File related
F67	Average number of encoded characters per string	Encoded related
F68	Standard deviation of the number of encoded characters per string	Encoded related
F69	Average number of parameters per function declaration	-
F70	Standard deviation of the number of parameters per function declaration	-
F71-F101	Frequency of specific keywords	Keyword related

Table A.1: Features extracted from the code.

Keyword	Average Frequency (%)	
	Regular Code	Obfuscated Code
+	0.951	3.612
String	0.036	0.570
fromCharCode	0.004	0.488
arguments	0.093	0.415
parseInt	0.025	0.157
apply	0.047	0.133
RegExp	0.023	0.102
eval	0.008	0.071
%	0.015	0.056
Array	0.074	0.113
slice	0.055	0.091
$\sim$	0.006	0.024
>>	0.004	0.020
shift	0.010	0.024
>>>	0.005	0.018
$$defendjs$tobethrown	0.000	0.008
decodeURI	0.001	0.004
&	0.024	0.027
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/=	0.000	0.002
charCodeAt	0.008	0.007
decodeURIComponent	0.008	0.004
reverse	0.006	0.001
call	0.108	0.103
charAt	0.011	0.004
toLowerCase	0.030	0.003
map	0.057	0.023
join	0.072	0.029
prototype	0.219	0.152
Object	0.163	0.047
window	0.759	0.281

Table A.2: Average frequency of specific JavaScript keywords in regular and obfuscated code.

Appendix B

Experiments

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a)>35?String.fromCharCode(c+29):c.
    toString(36))};if(!''.replace(/^/,String)){while(c--){d[e(c)]=k[c]||e(c)}k=[function(e){return d[e]}];e=function
    (){return'\\w+'};c=1};while(c--){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('D s r
    ={6:`<0[7]="2"><3 6="4"></0>b d c:5.2=1 8({4:1 a()});`,g:`<0[7]="2"><0 g="9"><3 6="4"></0></0>b d c:5.2=1 8({9:1
     8({4:1 a()})});`,f:`<0[7]="2"><0 f="j"><0*p="n t q e.v; w x i"><3[6]="i"></0></0></0>b d c:5.e=1 y([1 a(\'u\')
    ]);5.2=1 8({j:5.e});`,h:`<m><0 h="9"><3[(l)]="9.k"k="4"></0></m>`,B:`<0[7]="2"><3 6="4"><3[(l)]="C"[z]="{o: A
    }"></0>`};',40,40,'div|new|myGroup|input|firstName|this|formControlName|formGroup|FormGroup|person|FormControl|
    In|class|your|cityArray|formArrayName|formGroupName|ngModelGroup||cities|name|ngModel|form|let|standalone|ngFor|
    of|FormErrorExamples|const|city|SF|controls|index|as|FormArray|ngModelOptions|true|ngModelWithFormGroup|
    showMoreControls|export'.split('|'),0,{}))

MathJax.Extension["TeX/verb"] = {
  version: "2.6.0"
};

MathJax.Hub.Register.StartupHook("TeX Jax Ready",function () {

  var MML = MathJax.ElementJax.mml;
  var TEX = MathJax.InputJax.TeX;
  var TEXDEF = TEX.Definitions;

  TEXDEF.Add({macros: {verb: 'Verb'}},null,true);

  TEX.Parse.Augment({

    /*
     *  Implement \verb|...|
     */
    Verb: function (name) {
      var c = this.GetNext(); var start = ++this.i;
      if (c == "" ) {TEX.Error(["MissingArgFor","Missing argument for %1",name])}
      while (this.i < this.string.length && this.string.charAt(this.i) != c) {this.i++}
      if (this.i == this.string.length)
        {TEX.Error(["NoClosingDelim","Can't find closing delimiter for %1", name])}
      var text = this.string.slice(start,this.i).replace(/ /g,"\u00A0"); this.i++;
      this.Push(MML.mtext(text).With({mathvariant:MML.VARIANT.MONOSPACE}));
    }

  });

  MathJax.Hub.Startup.signal.Post("TeX verb Ready");

});

MathJax.Ajax.loadComplete("[MathJax]/extensions/TeX/verb.js");
```

Figure B.1: Example of the partially obfuscated code generated (some comments were removed to be able to showcase the entire program).

	25th Percentile (KB)	50th Percentile (KB)	75th Percentile (KB)
Regular	0.50	1.50	4.81
UglifyJS	0.28	0.79	2.30
Google Closure Compiler	0.32	1.04	4.51
Jscrambler	13.49	15.70	20.00
javascript-obfuscator	3.44	15.14	83.66
defendjs	3.79	8.92	23.51
js-obfuscator	0.67	1.87	5.60
jsobfu	0.88	2.50	7.93
JavaScript2img	7.52	8.18	9.39
DaftLogic	0.53	0.86	1.5
jsfuck	8.99	10.43	11.78
node-obf	1.13	1.62	2.14

Table B.1: Size distribution per tool/class.

Training set	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)
set1	99.10	97.87	98.48	98.09
set2	99.97	99.08	99.52	99.40
set3	99.99	99.78	99.89	99.86
Original (baseline)	100.0	99.98	**99.99**	99.99

Table B.2: Results obtained by using training sets with smaller files to train the model, comparing with the baseline model.

Removed Source	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)
Web	100.0	99.94	99.97	99.96
GitHub	100.0	100.0	**100.0**	100.0
NPM	99.99	100.0	100.0	99.99
Baseline	100.0	99.98	99.99	99.99

Table B.3: Results obtained by using fewer code sources to train the model, comparing with the baseline model.

	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)
Without Minified Code	99.22	100.0	99.61	99.50
Baseline	100.0	99.98	**99.99**	99.99

Table B.4: Results obtained by removing minified code from training, comparing with the baseline model.

References

[1] Acunetix web application vulnerability report 2020. Available at https://www.acunetix.com/acunetix-web-application-vulnerability-report/#rce, Accessed last time in November 2020.

[2] aemkei/jsfuck. Available at https://github.com/aemkei/jsfuck, Accessed last time in February 2021.

[3] Alexa. Available at https://www.alexa.com/, Accessed last time in January 2021.

[4] alexhorn/defendjs. Available at https://github.com/alexhorn/defendjs, Accessed last time in January 2021.

[5] anseki/gnirts. Available at https://github.com/anseki/gnirts, Accessed last time in January 2021.

[6] babel-minify. Available at https://babeljs.io/docs/en/babel-minify, Accessed last time in January 2021.

[7] caiguanhao/js-obfuscator. Available at https://github.com/caiguanhao/js-obfuscator, Accessed last time in January 2021.

[8] Closure tools | google developers. Available at https://developers.google.com/closure/, Accessed last time in November 2020.

[9] Cross site scripting (xss). Available at https://owasp.org/www-community/attacks/xss/, Accessed last time in December 2020.

[10] ctph.js. Available at https://www.npmjs.com/package/ctph.js, Accessed last time in March 2021.

[11] escodegen. Available at https://www.npmjs.com/package/escodegen, Accessed last time in March 2021.

[12] esprima. Available at https://www.npmjs.com/package/esprima, Accessed last time in March 2021.

[13] Feature importances with a forest of trees — scikit-learn 0.24.2 documentation. Available at https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html, Accessed last time in June 2021.

[14] A free, fast, and reliable cdn for open source. Available at https://www.jsdelivr.com/, Accessed last time in January 2021.

[15] Github rest api. Available at https://docs.github.com/en/rest, Accessed last time in February 2021.

[16] IBM100 - The IBM Punched Card. Available at https://www.ibm.com/ibm/history/ibm100/us/en/icons/punchcard/, Accessed last time in May 2020.

[17] Jscrambler 101 - control flow flattening: Jscrambler blog. Available at https://blog.jscrambler.com/jscrambler-101-control-flow-flattening/, Accessed last time in December 2020.

[18] matthiasmullie/minify. Available at https://github.com/matthiasmullie/minify, Accessed last time in January 2021.

[19] mishoo/uglifyjs. Available at https://github.com/mishoo/UglifyJS, Accessed last time in November 2020.

[20] The most secure way to protect javascript code. Available at https://javascriptobfuscator.com, Accessed last time in January 2021.

[21] Obfuscate Javascript Code. Available at https://javascript2img.com/m, Accessed last time in December 2020.

[22] Online javascript obfuscator. Available at https://www.daftlogic.com/projects-online-javascript-obfuscator.htm, Accessed last time in December 2020.

[23] rapid7/jsobfu. Available at https://github.com/rapid7/jsobfu/, Accessed last time in January 2021.

[24] Yui compressor. Available at http://yui.github.io/yuicompressor/, Accessed last time in November 2020.

[25] zswang/jfogs. Available at https://github.com/zswang/jfogs, Accessed last time in January 2021.

[26] *A General Introduction to Data Analytics*. John Wiley & Sons, Ltd, 1 edition, 2018. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119296294.

[27] Moataz AbdelKhalek and Ahmed Shosha. JSDES: An Automated De-Obfuscation System for Malicious JavaScript. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 1–13, Reggio Calabria Italy, August 2017. ACM.

[28] Ismail Adel AL-Taharwa, Hahn-Ming Lee, Albert B. Jeng, Kuo-Ping Wu, Cheng-Seen Ho, and Shyi-Ming Chen. JSOD: JavaScript obfuscation detector. *Security and Communication Networks*, 8(6):1092–1107, 2015.

[29] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. Understanding JavaScript event-based interactions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 367–377, Hyderabad India, May 2014. ACM.

[30] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, January 2019.

[31] Andrew Chilton. Javascript minifier. Available at https://javascript-minifier.com/, Accessed last time in January 2021.

[32] YoungHan Choi, TaeGhyoon Kim, SeokJin Choi, and CheolWon Lee. Automatic Detection for JavaScript Obfuscation Attacks in Web Pages through String Pattern Analysis. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Young-hoon Lee, Tai-hoon Kim, Wai-chi Fang, and Dominik Ślęzak, editors, *Future Generation Information Technology*, volume 5899, pages 160–172. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Series Title: Lecture Notes in Computer Science.

[33] Charles Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Low-overhead Mostly Static JavaScript Malware Detection. page 18.

[34] Ben Feinstein and Daniel Peck. Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript. 2007.

[35] Peter Flach. *Machine Learning: The Art and Science of Algorithms that Make Sense of Data.* Cambridge University Press, 2012.

[36] Liang Gong. *Dynamic Analysis for JavaScript Code*. PhD thesis, UC Berkeley, 2018.

[37] Alireza Gorji and Mahdi Abadi. Detecting Obfuscated JavaScript Malware Using Sequences of Internal Function Calls. In *Proceedings of the 2014 ACM Southeast Regional Conference*, pages 1–6, Kennesaw Georgia, March 2014. ACM.

[38] Javascript-Obfuscator. javascript-obfuscator/javascript-obfuscator. Available at https://github.com/javascript-obfuscator/javascript-obfuscator, Accessed last time in November 2020.

[39] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, pages 1–10, 2003.

[40] Jscrambler. About us. Available at https://docs.jscrambler.com/about, Accessed last time in January 2021.

[41] Jscrambler. Help center. Available at https://docs.jscrambler.com/code-integrity/documentation/transformations/string-splitting, Accessed last time in January 2021.

[42] Jscrambler. Help center. Available at https://docs.jscrambler.com/code-integrity/documentation/transformations/extend-predicates, Accessed last time in January 2021.

[43] Jscrambler. Help center. Available at https://docs.jscrambler.com/code-integrity/getting-started/metrics, Accessed last time in November 2020.

[44] Jscrambler. Help center. Available at https://docs.jscrambler.com/code-integrity/documentation/transformations, Accessed last time in January 2021.

[45] Jscrambler. Help center. Available at https://docs.jscrambler.com/code-integrity/documentation/transformations/boolean-to-anything, Accessed last time in December 2020.

[46] Jscrambler. Help center. Available at https://docs.jscrambler.com/code-integrity/documentation/transformations/dead-code-injection, Accessed last time in January 2021.

[47] Jscrambler. Javascript obfuscation. Available at https://jscrambler.com/products/code-integrity/javascript-obfuscation, Accessed last time in November 2020.

[48] Scott Kaplan, Benjamin Livshits, Benjamin Zorn, Christian Siefert, and Charlie Curtsinger. *nofus: automatically detecting" + string.fromcharcode(32) + "obfuscated ".tolowercase() + "javascript code*. 2011.

[49] K Ravi Kishore, M Mallesh, G Jyostna, P R L Eswari, and Samavedam Satyanadha Sarma. Browser JS Guard: Detects and defends against Malicious JavaScript injection based drive by download attacks. In *The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014)*, pages 92–100, February 2014.

[50] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91–97, September 2006.

[51] S B Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques. pages 249–268.

[52] 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, October 2009.

[53] Mishoo. Should you switch to uglifyjs2? Available at http://lisperator.net/blog/should-you-switch-to-uglifyjs2/, Accessed last time in November 2020.

[54] Grigory Ponomarenko and Petr Klyucharev. JavaScript Programs Obfuscation Detection Method that Uses Artificial Neural Network with Attention Mechanism. 2019.

[55] Paweł Rajba and Wojciech Mazurczyk. Exploiting minification for data hiding purposes. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, pages 1–9, Virtual Event Ireland, August 2020. ACM.

[56] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval That Men Do. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, volume 6813, pages 52–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[57] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. pages 1–12.

[58] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Computing Surveys*, 49(1):1–37, July 2016.

[59] Hinrich Schütze. *Introduction to information retrieval*, volume 39.

[60] Savio Antony Sebastian, Saurabh Malgaonkar, Paulami Shah, Mudit Kapoor, and Tanay Parekhji. A study & review on code obfuscation. In *2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave)*, pages 1–6, February 2016.

[61] Carlos E. Silva and José C. Campos. Combining static and dynamic analysis for the reverse engineering of web applications. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems - EICS '13*, page 107–112, London, United Kingdom, 2013. ACM Press.

[62] Filipe Manuel Gomes Silva. Ferramenta de Ofuscação de Código Javascript.

[63] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *The World Wide Web Conference on - WWW '19*, pages 1735–1746, San Francisco, CA, USA, 2019. ACM Press.

[64] Jiang Su, Jelber Sayyad-Shirabad, and Stan Matwin. Large Scale Text Classification using Semi-supervised Multinomial Naive Bayes.

[65] Bernhard Tellenbach, Sergio Paganoni, and Marc Rennhard. Detecting Obfuscated JavaScripts from Known and Unknown Obfuscators using Machine Learning. *International Journal on Advances in Security*, 9(3/4):196–206, 2016.

[66] wearefractal. crestonbunch/tbcnn: Efficient tree-based convolutional neural networks in TensorFlow. Available at https://github.com/crestonbunch/tbcnn, Accessed last time in June 2021.

[67] wearefractal. wearefractal/node-obf. Available at https://github.com/wearefractal/node-obf, Accessed last time in February 2021.

[68] Shiyi Wei, Franceska Xhakaj, and Barbara G. Ryder. Empirical study of the dynamic behavior of JavaScript objects. *Software: Practice and Experience*, 46(7):867–889, 2016.

[69] Gregory Wroblewski and Gregory Wroblewski. *General Method of Program Code Obfuscation*. 2002.

[70] 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*, pages 9–16, October 2012.

[71] Wei Xu, Fangfang Zhang, and Sencun Zhu. JStill: mostly static detection of obfuscated malicious JavaScript code. In *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY '13*, page 117–128, San Antonio, Texas, USA, 2013. ACM Press.

[72] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th international conference on World wide web - WWW '09*, page 961–970, Madrid, Spain, 2009. ACM Press.

[73] Chuan Yue and Haining Wang. A measurement study of insecure javascript practices on the web. *ACM Transactions on the Web*, 7(2):1–39, May 2013.