2017

Denial-of-service attack modelling and detection for HTTP/2 services

Erwin Adi

Edith Cowan University

Denial-of-Service Attack Modelling and Detection for HTTP/2 Services

by

Erwin Adi
B.Sc. Computer Science
M.Sc. Communications Technology and Policy

Submitted to the School of Science
in fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

EDITH COWAN UNIVERSITY

2017

Denial-of-Service Attack Modelling and Detection for HTTP/2 Services

by

Erwin Adi

Submitted to the School of Science
in 2017, in fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Businesses and society alike have been heavily dependent on Internet-based services, albeit with experiences of constant and annoying disruptions caused by the adversary class. A malicious attack that can prevent establishment of Internet connections to web servers, initiated from legitimate client machines, is termed as a Denial of Service (DoS) attack; volume and intensity of which is rapidly growing thanks to the readily available attack tools and the ever-increasing network bandwidths. A majority of contemporary web servers are built on the HTTP/1.1 communication protocol. As a consequence, all literature found on DoS attack modelling and appertaining detection techniques, addresses only HTTP/1.x network traffic. This thesis presents a model of DoS attack traffic against servers employing the new communication protocol, namely HTTP/2.

The HTTP/2 protocol significantly differs from its predecessor and introduces new messaging formats and data exchange mechanisms. This creates an urgent need to understand how malicious attacks including Denial of Service, can be launched against HTTP/2 services. Moreover, the ability of attackers to vary the network traffic models to stealthy affects web services, thereby requires extensive research and modelling.

This research work not only provides a novel model for DoS attacks against HTTP/2 services, but also provides a model of stealthy variants of such attacks, that can disrupt routine web services. Specifically, HTTP/2 traffic patterns that consume computing resources of a server, such as CPU utilisation and memory consumption, were thoroughly explored and examined. The study presents four HTTP/2 attack models. The first being a flooding-based attack model, the second being a distributed model, the third and fourth are variant DoS attack models. The attack traffic analysis conducted in this study employed four machine learning techniques, namely Naïve Bayes, Decision Tree, JRip and Support Vector Machines.

The HTTP/2 normal traffic model portrays online activities of human users. The model thus formulated was employed to also generate flash-crowd traffic, i.e. a large volume of normal traffic that incapacitates a web server, similar in fashion to a DoS attack, albeit with non-malicious intent. Flash-crowd traffic generated based on the defined model was used to populate the dataset of legitimate network traffic, to fuzz the machine learning-based attack detection process. The two variants of DoS attack traffic differed in terms of the traffic intensities and the inter-packet arrival delays introduced to better analyse the type and quality of DoS attacks that can be launched against HTTP/2 services.

A detailed analysis of HTTP/2 features is also presented to rank relevant network traffic features for all four traffic models presented. These features were ranked based on legitimate as well as attack traffic observations conducted in this study. The study shows that machine learning-based analysis yields better classification performance, i.e. lower percentage of incorrectly classified instances, when the proposed HTTP/2 features are employed compared to when HTTP/1.1 features alone are used.

The study shows how HTTP/2 DoS attack can be modelled, and how future work can extend the proposed model to create variant attack traffic models that can bypass intrusion-detection systems. Likewise, as the Internet traffic and the heterogeneity of Internet-connected devices are projected to increase significantly, legitimate traffic can yield varying traffic patterns, demanding further analysis. The significance of having current legitimate traffic datasets, together with the scope to extend the DoS attack models presented herewith, suggest that research in the DoS attack analysis and detection area will benefit from the work presented in this thesis.

Keywords: HTTP/2, Denial of Service attack, traffic analysis, machine learning techniques.

Principal Supervisor: Dr. Zubair Baig
Associate Supervisor: Assoc. Prof. Philip Hingston

Declaration

I certify that this thesis does not, to the best of my knowledge and belief:

(i) incorporate without acknowledgement any material previously submitted for a degree or diploma in any institution of higher education;

(ii) contain any material previously published or written by another person except where due reference is made in the text of this thesis; or

(iii) contain any defamatory material;

Erwin Adi

2017

Acknowledgements

My Principal Supervisor, Dr. Zubair Baig has been admirably supportive through my PhD journey and the writing of this thesis. He directed me through the challenge of finding cutting-edge solutions, software tools, recent studies and research grants. Zubair is very responsive through all communication means, constantly giving advice on technical and thesis work challenges. I send my tremendous thanks to Zubair for having guided the study to the direction that contributes to its significance.

Assoc. Prof. Philip Hingston, my Associate Supervisor, played a big role in advancing my research skills. He stretched my hard work to exhaustively explore the subjects in the area and critically assessed my findings. He indicated the areas where I needed improvements and advised on relevant references. I truly appreciate Philip for shaping my academic aptitude.

I am grateful to Assoc. Prof. Chiou Peng Lam for having remarkably appraised my research methodology and thesis proposal, contributing to scholastic insights, and for co-authoring two publications. My sincere appreciation also goes to Prof. Craig Valli for his pointer to investigate the new breed of communications protocol, HTTP/2. His immense knowledge in the area has led me to delve into this novel research avenue.

The conduct of this study would not have been possible without the financial support from Edith Cowan University through the International Postgraduate Research Scholarship, which provided me with forefront access to knowledge, materials and living support in Australia. Much appreciation goes to Avaya for their Research Grant which endowed funding to the provisioning of high-end equipment employed in this study.

I will always remember the joy of being together with my lab-mates: the fellowship of attending seminars, having daily laughter, support and despair, and sharing the finesse in surviving and thriving in this beautiful country, Australia.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Purpose of the Study 5

1.3 Significance of the Study 6

1.4 Thesis Contribution 8

1.5 Thesis Structure 9

2 Literature Review **11**

2.1 HTTP/1.1 Protocol 11

2.2 DoS Attacks 16

2.2.1 Network-Based DoS Attacks 18

2.2.2 Application-Based DoS Attacks 22

2.2.3 Discussion 24

2.3 DoS Detection Techniques 25

2.3.1 Detecting Vulnerability-Based Attacks 26

2.3.2 Detecting Flooding Attacks 27

2.3.2.1 Detection Techniques for Network-Based DoS Attacks 27

2.3.2.2 Detection Techniques for Application-Based DoS Attacks 28

2.3.2.3 Anomaly Detection Techniques using Machine Learning Techniques 32

2.3.2.4 Challenges to Detecting Application-Based DoS Attacks 48
2.4 The HTTP/2 Protocol 59
2.4.1 Initiatives 59
2.4.2 Protocol Specifications 59
2.4.3 Security Considerations 62
2.4.4 Exposure to Resource Depletion 65
2.4.5 Implication to DoS Detection Technique 67
2.5 Conclusion 67

3 Research Approach and Methodology 71
3.1 Research Approach 72
3.1.1 Phase 1: Model Development 73
3.1.2 Phase 2: Traffic Generation 74
3.1.3 Phase 3: Feature Extraction and Dataset Creation 75
3.1.4 Phase 4: Feature Ranking and Traffic Classification 86
3.2 Evaluation Metrics 88
3.3 Experimental Setup 90
3.4 Ethical Issues 92
3.5 Conclusion 93

4 Legitimate Traffic Modelling and Analysis 95
4.1 Logs of Online User Browsing Behaviours 96
4.2 User Model 98
4.3 A Framework to Generate Normal Traffic 99
4.3.1 Evaluating the Framework 104
4.4 Flash-Crowd Traffic 107
4.4.1 Generating Flash-Crowd Traffic 108
4.4.2 Creating Flash-Crowd Dataset 109
4.5 Conclusion 113

5 Attack Traffic Modelling and Analysis 115

5.1 Flood Attack 116

5.1.1 Attack Model and Scenarios 117

5.1.2 Results 123

5.1.3 Discussion 126

5.1.4 Conclusion 129

5.2 Distributed Attack 129

5.2.1 Attack Model and Scenarios 130

5.2.2 Results and Discussion 136

5.2.3 Conclusion 148

6 Stealthy Attack Modelling and Analysis 151

6.1 Stealthy Attack 1 152

6.1.1 Attack Model and Scenarios 152

6.1.2 Results and Analysis 156

6.1.3 Conclusion 165

6.2 Stealthy Attack 2 165

6.2.1 Attack Model and Scenario 166

6.2.2 Results and Analysis 171

6.2.3 Conclusion 184

6.3 Analysis and Discussion 185

6.3.1 Feature Ranking Comparison with HTTP/1.1 Features . . 185

6.3.2 Performance Comparison 189

6.4 Conclusion 194

7 Conclusion 197

7.1 Contributions of the Study 198

7.2 Limitations and Future Work 202

References 205

A Machine Learning Parameter Values **217**

A.1 Naïve Bayes 217

A.2 Decision Tree 217

A.3 JRip 218

A.4 Support Vector Machines 218

A.5 Self Organizing Maps 219

List of Figures

1-1 DDoS attack 4

2-1 A client-server communication model 12
2-2 A client-server communication detailed with layers 13
2-3 Messages are encapsulated at each layer. 14
2-4 An end-to-end HTTP communication 16
2-5 DDoS attack 17
2-6 SYN flood attack 19
2-7 Smurf attack 20
2-8 A Decision Tree 40
2-9 k-NN finds the most vote from the closest k neighbours 43
2-10 (a) H3 is the maximum margin hyperplane, (b) Support vectors . 44
2-11 Frame format 60
2-12 An html response 61
2-13 Illustration on HTTP/1.1 requests/responses and HTTP/2 multiplexing 61
2-14 Illustration on HTTP/2 multiplexing from a different perspective 62
2-15 Stream states 65

3-1 A dataset consists of a set of instances, which are characterised through a set of features. 76
3-2 TCP and TLS Handshake 77
3-3 Captured packets within an observed time window 79
3-4 A snippet of captured traffic 80

3-5 The traffic generation setup 91

4-1 The framework for creating flash-crowd dataset from a defined normal user model. 95
4-2 State Transition representing a User Model 98
4-3 An example of one User Model taken from DOBBS Sample 1 . . . 99
4-4 The framework on how DOBBS log was used to generate normal traffic 100
4-5 State Transition representing a User Model 101
4-6 Generating flash-crowd traffic and creating dataset 108
4-7 The generated traffic, captured and viewed using TShark 109
4-8 Filtering traffic and extracting features 109

5-1 The flooding-based attack model 118
5-2 The window_update payload format 120
5-3 The window-size-increment tells the receiver how many more bytes the sender can transmit 120
5-4 The traffic generation setup for Investigation-1 and Investigation-3 121
5-5 The traffic generation setup for Investigation-2 123
5-6 The DDoS attack model 131
5-7 The traffic generation setup for DDoS attack 131
5-8 A higher window-size-increment value allows fewer number of subsequent window_update packets 134
5-9 Process for normal and attack traffic classification 141
5-10 Distributed Attack Performance with Naive Bayes classification . 143
5-11 Distributed Attack Performance with Decision Tree classification . 144
5-12 Distributed Attack Performance with JRip classification 145
5-13 Distributed Attack Performance with Support Vector Machine classification 146

6-1 Visualization of count_syn feature values. Left: DDoS attack. Right: Stealthy Attack-1. 156

6-2 A threshold line can split DDoS and flash-crowd traffic 157
6-3 Stealthy Attack-1 Performance with Naive Bayes classification . . 161
6-4 Stealthy Attack-1 Performance with Decision Tree classification . 162
6-5 Stealthy Attack-1 Performance with JRip classification 163
6-6 Stealthy Attack-1 Performance with Support Vector Machine classification 164
6-7 Visualization of Stealthy Attack-1 size_rstAck feature values. . . 167
6-8 A sample of HTTP/2 Request traffic produced using nghttp2 library 169
6-9 A sample of HTTP/2 Request traffic produced using curl library . 169
6-10 Visualization of the size_rstAck feature values. Left: Stealthy Attack-1. Right: Stealthy Attack-2. 171
6-11 Stealthy Attack-2 Performance with Naive Bayes classification . . 174
6-12 Stealthy Attack-2 Performance with Decision Tree classification . 175
6-13 Stealthy Attack-2 Performance with JRip classification 176
6-14 Stealthy Attack-2 Performance with Support Vector Machine classification 177
6-15 Visual inspection to find a range of X-values that yield low incorrectly classified instances values. 179
6-16 Incorrectly classified instances for all 3 attack types 181
6-17 Detection Rate performance comparison for all 3 attack types . . 181
6-18 False Alarm Rate performance comparison for all 3 attack types . 181
6-19 Cluster visualization using Self Organizing Map 183
6-20 A comparison of Stealthy Attack-1 performance when using both HTTP/1.1 features and HTTP/2 features. 192
6-21 A comparison of Stealthy Attack-2 performance when using both HTTP/1.1 features and HTTP/2 features. 193

List of Tables

1.1 The number of connected devices over the past 14 years 2

2.1 HTTP versions and years of release 13
2.2 An example of traffic samples characterised by three features . . . 34
2.3 An example of traffic samples characterised by three features . . . 36
2.4 An example of a labelled traffic dataset characterised by three features 38
2.5 The counts and probabilities of the features 39
2.6 Features used by the existing studies on DoS detection 56
2.7 The range of data to observe for the investigations in this thesis . 69

3.1 Extracted features from network traffic shown in Figure 3-3 79
3.2 Extracted features from traffic shown in Figure 3-4 81
3.3 ACK packet time lapse values, extracted from traffic shown in Figure 3-4 82
3.4 RST-ACK packet time lapse values, extracted from traffic shown in Figure 3-4 83
3.5 Features used in this study 84
3.6 Features and their names 85

4.1 Dataset sample information 97
4.2 A snippet of DOBBS Sample 1 97
4.3 Dataset sample information 105
4.4 BotMaster mimicked DOBBS Sample-1 closely 106
4.5 BotMaster mimicked DOBBS Sample-2 closely 107

4.6 Filtering traffic messages to extract features 110
4.7 Flash-crowd traffic characteristics described through its feature values 112

5.1 Computing resource consumption during attacks 124
5.2 Duration the WINDOW_UPDATE frame was sent 138
5.3 DDoS using test case 4 140
5.4 DDoS using test case 5 140
5.5 Ranked features for the distributed attack traffic 142

6.1 Stealthy Attack-1 model 155
6.2 Ranked features for Stealthy Attack-1traffic 158
6.3 Stealthy Attack-2 model 170
6.4 Ranked features for Stealthy Attack-2 traffic 173
6.5 Feature ranks using Information Gain for all 3 attack types 188
6.6 Feature ranks using Gain Ratio for all 3 attack types 188
6.7 Incorrectly classified instances (%) by machine learning techniques applied with only HTTP/1.1 features. 190

Publications Arising from the Research

Adi, E., Baig, Z., Lam, C. P., & Hingston, P. (2015). Low-rate denial-of-service attacks against HTTP/2 services. In *IT Convergence and Security (ICITCS), 2015 5th International Conference on* (pp. 1–5). IEEE.

Adi, E., Baig, Z. A., Hingston, P., & Lam, C. P. (2016). Distributed denial-of-service attacks against HTTP/2 services. *Cluster Computing*, *19* (1), 79–86.

Chapter 1

Introduction

1.1 Background

The Internet has been a significant catalyst for the world economy, with businesses and society relying on it for their communication and information needs. Ensuring the availability of these services is a challenging task due to the growing volume of Internet traffic and the various communication standards that it supports. HTTP/2 has emerged as a web communication standard over the past few years. Through this thesis, novel techniques are proposed and evaluated for detecting malicious network traffic that targets HTTP/2 services, as part of a Denial of Service (DoS) attack. In addition, various techniques are proposed to differentiate legitimate from malicious network traffic.

It is estimated that the number of Internet-connected devices in 2020 will be 50 billion ("The Internet of Things", 2014), as detailed in Table1.1. This implies that there will be more computer-connected devices per person. The ubiquity of these devices creates many convenient benefits for society. For instance, online banking speeds up purchases; mobile technology eliminates the distances between doctor-patient and teacher-students ("The Third Great Wave", 2014); and innovative applications facilitate ordering a taxi and monitoring its arrival. Consequently, an increasing number of services and devices will be connected to the Internet ("The Internet of Things", 2014), and the modern society will become more reliant

Table 1.1: The number of connected devices over the past 14 years

	# devices (billion)	world population (billion)
2003	0.5	6.3
2010	12.5	6.8
2015	25	7.2
2020	50	7.6

on the availability of Internet-based services.

Internet-connected devices have increased the demand for content. For instance, instead of displaying text as web servers were originally designed to render, contemporary web communication exchange richer content such as audio and video. Due to the multimedia content that these devices transfer and download, the current web browsing standard (HTTP/1.1) is reaching its full capacity (Grigorik, 2013b), as it was designed to exchange text. Web users often experience slow Internet speed; hence, a new standard (HTTP/2) has been designed to support communication at higher speed (Belshe, Peon, & Thomson, May 2015). HTTP/2 was published in May 2015. While the web programming software used to build websites remains unchanged, the HTTP/2 data communication techniques differ from those of HTTP/1.1. HTTP/2 architecture introduces binary framing, multiplexing, message interleaving, and application-layer flow control (Belshe et al., May 2015). As such, the traffic it sustains over the Internet media shows different patterns than what has been observed and reported in the literature, for HTTP/1.1.

The introduction of a new application-layer communication standard prompts a critical analysis of its security implications. One important security consideration is availability of web-based services. The availability of the Internet creates a healthy ecosystem: society can access the conveniences offered by businesses, and businesses make use of the growing size of the Internet to offer ever expanding services to end users/clients. Unfortunately, parasites do exist in the ecosystem. For monetary gain, adversaries threaten the availability of businesses by attempting to bring down web servers and corresponding services (Heron, 2010; Mansfield-Devine, 2011). Such a threat can translate into a Denial of Service

(DoS) attack, which is defined as, "an explicit attempt by attackers to prevent legitimate users of a service from using that service" (CERT, 1997). Other motivations for launching DoS attacks include ideological beliefs to uphold one's views while attempting to suppress the opposition's ability to publicise through websites; intellectual challenge to learn how to launch attacks; and cyber warfare, i.e. attacks supported by military or terrorist organizations (Zargar, Joshi, & Tipper, 2013).

DoS attacks can be launched either by suffocating an Internet-connected machine by sending it a large volume of traffic (*flooding* technique), or by exploiting a bug (*vulnerability*) in the target system's software that incapacitates the service (Mirkovic & Reiher, 2004). In flooding attack techniques, adversaries send more packets than the target computer can process during a frame of time. This forces the victim to either constantly use its processing capability to handle incoming packets, or to face consequences due to depletion of its memory through packet buffering. When the victim is a web server, the connected remote users experience either slow web responses, interrupted services, or no services at all. Similar service disruption symptoms can be caused through exploiting a vulnerability of the target. Web servers run software that parses and executes HTTP commands that it receives. Attackers can exploit the software vulnerability of a web server to cause the program to run undesirably. This could include running an infinite loop, causing the web server to stop responding to legitimate client requests. As illustrated in Figure 1-1, the attacks can be amplified using a group of compromised computers in order to launch a large storm of traffic (Chang, 2002). This variant is called a Distributed Denial-of-Service attack (DDoS).

DoS attacks pose a serious threat to businesses as they involve overwhelming of the communication channel with malicious traffic. In the year 2000, major websites (CNN.com, ZDNet, Yahoo) and businesses (Amazon.com, E*Trade, and eBay) were targeted as part of DoS attacks that incapacitated web servers and caused considerable monetary loss (Garber, 2000). In 2007, vital web servers in Estonia including those belonging to banks, ministries, newspapers and broadcasters were brought down through an organized DoS attack perpetrated by an

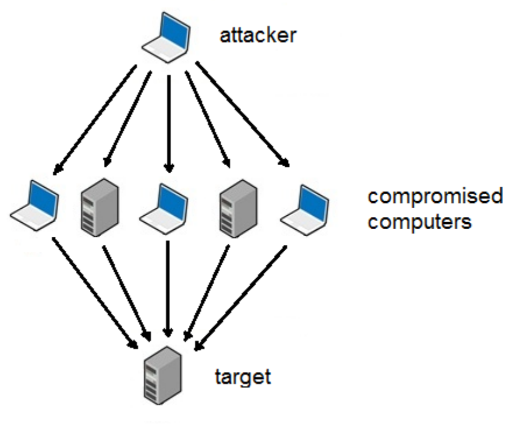


Figure 1-1: DDoS attack

adversary (*Estonian Attacks Raise Concern over Cyber 'Nuclear Winter'*, 2007). The attack crippled the Estonian government's paperless IT infrastructure and led to severe financial losses. In 2010, web servers belonging to MasterCard, Visa, and PayPal were subjected to a DDoS attack (Addley & Halliday, 2010). The attacks were launched as an act of revenge on companies that froze all payments to WikiLeaks – a website that publishes sensitive, classified information. In 2012, nine major online banking sites in the USA were made unavailable (Kitten, 2013). It was believed that the attack was launched by an adversary residing in an enemy state.

Detecting and bearing the ability to prevent DoS attacks against a web server are therefore crucial in order to provide uninterrupted services to legitimate clients. For network and web hosting operators, the ability to detect DoS traffic prevents unwanted operational costs, helps to plan future infrastructure, and allows operators to provide services that otherwise would have been disrupted by illegitimate traffic.

Unfortunately, DoS attacks are increasing in their volume, ubiquity, complexity, and use of novel techniques (Mansfield-Devine, 2011). Flood-based DoS attack volume is usually expressed in terms of bandwidth usage. In 2002, the volume of DDoS attack traffic against large carriers and content providers around the world was found to be 400 Mbps (Labovitz, 2010). In 2013, the volume was increased to 60 Gbps (Paganini, 2013). Recently the total worldwide attack traffic volume has been increasingly difficult to collect as the number has grown exponentially. To illustrate, in the second quarter of 2015 alone, DDoS attack volume touched a staggering 1,000 Gbps (Keane, 2015). In the beginning of 2016, the British Broadcasting Corporation (BBC) website was flooded with a 600 Gbps attack (Khandelwal, 2016). These threats are not to be underestimated due to the fact that all web servers can become inaccessible when flooded with more traffic than what can be processed.

Currently detection techniques for DoS attacks against web services have been based on HTTP/1.1 traffic patterns. HTTP/2 as a new protocol is vulnerable to DDoS attacks as is presented in this thesis. The new standard is no longer regarded as experimental, but rather has been formalised and deployed for public access. Although some semantics of the original protocol are preserved, the information exchange and implementation mechanism of the protocol demand memory consumption. The introduction of HTTP/2 opens up a new channel for adversaries to interrupt the availability of web servers. This means that experimentation, observation, and analysis of how HTTP/2-packet-based attacks can be launched and be detected are important new areas of research.

This research studies how HTTP/2 consumes the computing and networking resources of web servers. It develops attack models, differentiates between legitimate and malicious attack traffic, and shows how cyber attacks intended to deny access to HTTP/2 servers could be detected.

1.2 Purpose of the Study

The introduction of a new web browsing standard HTTP/2 poses a novel challenge for DoS attack detection. Hence, the objectives of this study are:

- to develop attacker models for DoS attacks against HTTP/2 servers;
- to generate traffic representing DoS attacks against HTTP/2 servers;
- to identify features and produce datasets that best represent DoS attacks against HTTP/2 servers; and

- to demonstrate how stealthy HTTP/2 DoS attacks can be analysed using machine learning techniques.

The challenge to generate HTTP/2 traffic stemmed from the unavailability of HTTP/2 implementation. Few programming libraries were available to support the development of the traffic generator. This thesis work selected the most appropriate library that could apply the model previously developed, and aimed to generate HTTP/2 attack traffic. The study also examined how the generated attack traffic consumed CPU and memory resources of an HTTP/2 web server.

In order to analyse the characteristics of the attack traffic, this study also evaluated and examined the instances of the traffic. A large number of instances that made up a dataset were required to extensively include the variations of the instances. Currently there is no publicly available dataset representing HTTP/2 traffic. Instead, the datasets used in the literature describe normal and attack traffic under HTTP/1.1 conditions. Through this work, datasets representing DoS attacks against HTTP/2 were generated to best represent live network traffic.

Each data sample was characterised by a series of features. One of the objectives of this study was to identify features that characterised HTTP/2 traffic. The study generated anomalous and legitimate HTTP/2 traffic patterns. Among detection methods discussed in the literature, machine learning techniques are suitable approaches to detect network traffic anomalies. The study explored the use of machine learning techniques for efficient and effective detection of DoS attacks. Machine learning techniques were used as a means to measure how attack traffic differs from normal traffic in this study. Building upon these investigations, the study aimed to demonstrate a stealthy HTTP/2 attack and showed how it could be modelled to bring down a web server.

1.3 Significance of the Study

There are three significant aspects of this research. These are:

- to explore the design within HTTP/2 servers that can be exploited to consume computing resources through a DDoS attack;

- to provide an extension of researchers' understanding on DoS attacks against web servers; and
- to develop knowledge on how machine learning can be used to detect DoS attacks for HTTP/2 traffic.

First, the study explored how attacks against HTTP/2 services consumed computing resources. Little was known about how much memory and CPU time were utilised for handling `HTTP/2 Requests` and `Responses`. This study set up a test bed that launched HTTP/2 packets towards an HTTP/2 server and monitored the CPU utilisation, memory consumption, network throughput, and packet loss. The results obtained from this experiment allow researchers to understand how HTTP/2 servers can become unavailable when subjected to DoS attacks.

Second, the study extended the current understanding of detecting DoS attacks against web servers. Existing studies in the area observed attacks against HTTP/1.1 servers. Only a few of the deployed servers in the Internet implement HTTP/2 and no dataset described HTTP/2 traffic. Moreover, no study was conducted for detecting DoS attacks against HTTP/2 servers. This study extends the knowledge in the area through testing and analysing different features relevant to detecting DoS attacks against HTTP/2 servers. The study initially observed whether the existing solutions (i.e. the detection methods on DoS attacks against HTTP/1.1 servers) could be used effectively. Subsequently, the study observed DoS attack traffic against an HTTP/2 server, generated a set of new features based on these observations, and evaluated the effectiveness of the new features in distinguishing DoS from normal traffic. The results provided new insights for researchers on how HTTP/2 can influence effective approaches to detecting DoS attacks.

Third, the study developed knowledge on how machine learning can be used to characterise HTTP/2 traffic. Existing techniques learned patterns and built classification rules from HTTP/1.1 data. In contrast, this research introduced new traffic patterns and features. In addition, this study exhibited different accuracy with distinguishing attacks from normal traffic. The investigation of the data involved investigations using machine learning techniques. Therefore, this study developed knowledge on how machine learning techniques can help accurately differentiate legitimate traffic from malicious.

1.4 Thesis Contribution

The main contributions of this thesis are: HTTP/2 traffic models and datasets – each with normal and attack class; HTTP/2 traffic features, and a stealthy attack model. These are detailed as follows.

- **HTTP/2 normal traffic model creation and dataset generation**. To differentiate anomalous traffic signifying DoS attacks, normal HTTP/2 traffic was generated to serve as the standard. Such synthetic traffic was crucial not only for its use in this thesis but also for research at large that could be conducted in the area. The traffic was generated as a result of simulating a model that was proposed in this study. The evaluation of the model showed how the generated normal HTTP/2 traffic can be justified by its internal validity, and that initialising the model with another sample showed how it closely represented the sample. Hence, a wider audience can repeat the model to generate different traffic patterns. In this study, the generated model was used to produce network traffic as the ground truth to create a dataset representing normal online user behaviours.

- **HTTP/2 attack traffic model creation and dataset generation**. Attack traffic consumes the CPU utilisation of the target computer. HTTP/2 introduces new message exchange mechanisms that differ to its predecessors. This study investigated the computing resources of an HTTP/2 server when subjected to different traffic parameters. It observed the CPU consumption and memory of a server, and proposed a range of HTTP/2 traffic parameter values that can incapacitate the server. These parameters served as the attack model that researchers can replicate and benefit from. Furthermore, the study created attack datasets from the generated HTTP/2 traffic that the model simulated. To our knowledge, there is currently no such publicly available dataset.

- **Examination and ranking of HTTP/2 traffic features**. Examining datasets includes selecting and identifying features when the datasets are applied to different evaluation purposes. Evaluating large data can be complex and time consuming when dealing with many features. This study proposed a set of features to characterise HTTP/2 traffic for both normal and attack class, and identified a set of the most relevant features.

- **HTTP/2 stealthy attack model definition**. Having classified attack and normal traffic, one of the important results was to present models that generate HTTP/2 traffic of both classes. This model produced stealthy attack traffic that represented similar characteristics to normal traffic. The thesis presented model parameters which give wide opportunities to researchers in the area to explore further attack traffic patterns.

1.5 Thesis Structure

HTTP/1.1 and HTTP/2 protocols are discussed in the next chapter. It details how a pair of client-server can exchange information using the protocol and shows the distinction between both protocols. Having firstly discussed how HTTP/1.1 protocol works, Chapter 2 discusses how DoS attacks can be launched in computer networks running HTTP/1.1 protocol, and how they can be detected. The chapter shows the gaps that exist in the literature on DoS detection methods when applied to HTTP/2.

To study how HTTP/2 DoS attack traffic can be detected, this thesis modelled and generated both normal and attack traffic. A computer lab was setup for the purpose of generating the traffic, extracting the features, and creating datasets for analysis. These are discussed in Chapter 3.

Methodology for normal traffic modelling and generation is discussed in Chapter 4. Similarly, the discussion on how attack traffic was modelled and generated is presented in Chapter 5. Analyses of these traffic patterns led this study to model and generate stealthy attack traffic that degraded the performance of the detection techniques previously presented. Two stealthy attack traffic models were proposed in Chapter 6. Furthermore, the chapter showed how the stealthy traffic detection analysis performed when the analysis were based on features commonly used by HTTP/1.1 DoS detection methods.

Chapter 2

Literature Review

This chapter is divided into four main sections. The first section is a technical review of HTTP/1.1 which is the current protocol for web communications. It provides details on how the HTTP/1.1 protocol facilitates communication of messages across the Internet. The second section is a technical review of DoS attack, explaining its types and variants such as DDoS. It also illustrates how the attacks can be carried out by the adversary. The third section reviews the literature on how anomalous traffic such as DoS/DDoS attacks can be detected. It further discusses machine learning techniques focusing on how they are deployed for anomaly detection and traffic analysis. The fourth section is a technical review of HTTP/2 covering: how HTTP/2 has a different information exchange mechanism from HTTP/1.1; what the relevant security implications are; how it is exposed to DoS attacks; and which techniques were proposed in this study to detect DoS attacks against HTTP/2 servers.

2.1 HTTP/1.1 Protocol

The web traffic uses the Internet to connect the user and the information resources hosted on web servers. The standard protocol for supporting web activity is the Hypertext Transfer Protocol (HTTP). This section provides a thorough review on how HTTP operates on the Internet.

The World Wide Web is a term coined by Tim Berners Lee in 1989 to denote

Client
Server

Figure 2-1: A client-server communication model

the universal world of information accessible through networked computers. It was designed with simplicity in mind for users to search and view information. A typical web communication exchange involves a search requested by a user from a *client* computer from a remote computer *server*. This is illustrated in Figure 2-1. In response to the request, the web server processes and returns data to the client machine.

In 1991, Berners-Lee initiated a high-level design of a computer communication protocol to implement the above client-server communication mechanism (Berners-Lee, Fischetti, & Foreword By-Dertouzos, 2000). This protocol was named Hypertext Transfer Protocol (HTTP) and it was unofficially named HTTP/0.9. The protocol allowed a client to request a file from a server; and allowed a server to send a response file in Hypertext Mark-up Language (HTML) format. This format can contain references to an image and links to other documents. An HTTP client, such as a web browser, can retrieve the referenced image by using HTTP: the web browser requests the image to a server, and the server responds by sending the image to the web browser.

This initiative was well received and led to the rapid growth in web-based communication. Client-side software was written to ease the use of web communication over the Internet. The first client software was Mosaic and it was renamed to Netscape when it commercialised the software in 1994. Subsequently, the HTTP Working Group introduced and revised the protocol over time (Table 2.1). HTTP has become the protocol of choice for web browsing communication, and web browsers such as Mozilla Firefox, Chrome, Internet Explorer and Opera have become popular client software for facilitating web-based information access by end-users.

Table 2.1: HTTP versions and years of release

Version	Year introduced
HTTP/0.9	1991
HTTP/1.0	1996
HTTP/1.1	1997
HTTP/2.0	2015

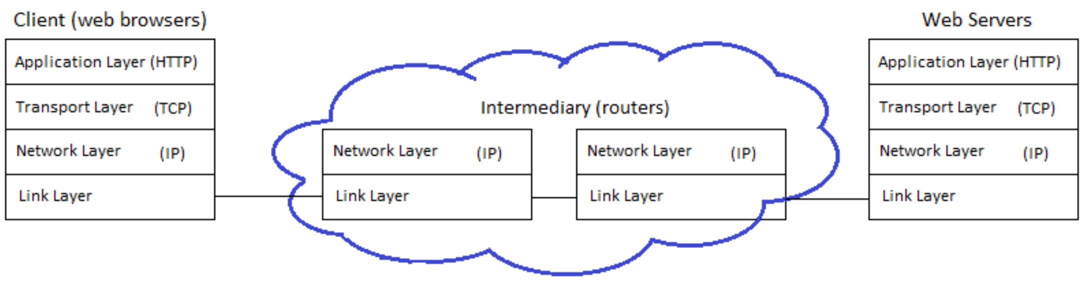


Figure 2-2: A client-server communication detailed with layers

To reach a web server, information sent by web browsers was encapsulated through different *computer communication layers* before being transmitted over the Internet. This is illustrated in Figure 2-2 which essentially depicts the Internet Protocol suite , i.e. a set of protocols used for computer communications. The figure illustrates the Internet that consists of intermediary machines such as *routers* which function to find connection paths between clients and servers. Ideally a router has many physical network interfaces to route network information from an input interface to an output interface. A series of connected routers link two end computers and allows these computers to exchange network messages. A mesh of router links creates an *Internet* communication infrastructure. The figure shows that routers are also computers that are equipped with layered computer communication architecture.

To allow communications between different computers, each message from each layer is encapsulated at the layer below, to pack the information for network transmission. A *header* is affixed at each layer, allowing the layered interconnections. This is shown in Figure 2-3. Header-affixed information in network communications is termed *packets*.

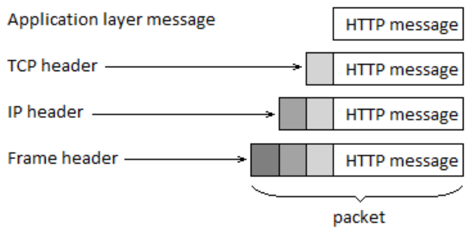


Figure 2-3: Messages are encapsulated at each layer.

The *Application Layer* describes protocols for human to interact with the network. HTTP is an example of an Application Layer protocol. Other than HTTP, examples of popular protocols are File Transfer Protocol (FTP) to transfer files, Peer-to-Peer (P2P) to seamlessly connect with another computer, and Simple Mail Transfer Protocol (SMTP) to facilitate exchange of emails. The Application Layer translates human inputs (such as a website address) for delivery to the layer below it. For example, to request a page from a remote web server, the web browser translates the website address to a destination IP address (explained shortly) and uses HTTP to send a text message such as `GET /index.html HTTP/1.1` to the destination address. To do this, the GET message is firstly encapsulated in the Transport Layer packet.

The *Transport Layer* manages a connection with a remote end, controls the flow of the packets, and informs the Application Layer service type (e.g. HTTP). The remote end in this example is the web server (which address was already typed on the web browser as described above). User Datagram Protocol (UDP) is an example of a Transport Layer protocol. UDP defines simple connections without mandating a remote end to acknowledge that a packet has arrived at the destination. On the other hand, Transmission Control Protocol (TCP) is an example of a Transport Layer protocol that mandates a remote end to acknowledge if it has received a packet. TCP is a reliable transport protocol; it initiates and terminates a connection. Controlling connections is done through sending TCP packets with some flags in the TCP header indicating the state of a connection, such as initiating a connection, ending a connection, or acknowledging a received packet. The TCP header also specifies the Application Layer service type it carries through the use of *port* numbers. For example, HTTP is defined to function on TCP port 80. To find a path to the destination address (in this example, the web server), the TCP packet is encapsulated in the Network Layer packet.

The *Network Layer* is responsible for routing of packets through the Internet. The Internet Protocol (IP) is the world standard to route packets from the source (i.e. the computer where the web browser is) to the destination address (i.e. the web server). These source and destination addresses are defined in the IP header . Routers, the intermediary devices between the source and destination machines, inspect the IP headers to find the destination address. They look up the routing table to find which interface associates with the destination address, and forward the packets to a physical interface. The *Link Layer* handles how the packets are transferred over the wire through the routers' interfaces as electric or light signals.

The communication protocols described above are repeatedly applied at each hop of a route until the packet reaches the destination address. At each router, the process is reversed from the Link Layer to the Network Layer. Routers remove the header of the Link Layer so that IP header information can be examined. Subsequently, the Network Layer to Link Layer encapsulation process as explained above is also repeated: routers find which interface is associated with the destination address, encapsulate the IP packet into a Link Layer frame, and forward the packet to a physical interface. At the destination machine, the web server removes each lower layer header until it obtains the original HTTP message.

After the web server eventually finds the `GET /index.html HTTP/1.1` message, it sends the requested HTML file back to the client. An example of an `index.html` file content is a `"<html>HelloWorld</html>"` message. The file is sent as a text message from the web server to the web browser following exactly the same procedure as previously described. The simplified end-to-end HTTP communication is illustrated in Figure 2-4. The figure illustrates how HTTP messages allow clients and servers to communicate while decoupling the protocol from network issues. In summary, HTTP allows the web browser to request files from the web server using text messages without having to define the network

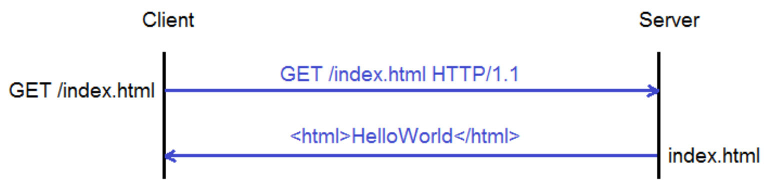


Figure 2-4: An end-to-end HTTP communication

protocols, which are standardised and implemented within HTTP supporting machines.

The web traffic sent from a client to a web server is termed as *HTTP Request*, and the response traffic is called *HTTP Response*. When a client sends a flood of HTTP Requests, the server can become too busy handling the requests and can fail to respond. An adversary can design this scenario to attack a web server. This attack is called Denial of Service (DoS) attack which is detailed in the next section.

2.2 DoS Attacks

This section reviews the underlying technical mechanism of DoS attacks. These attacks can be described as malicious attempts to inhibit the legitimate use of a computing service (CERT, 1997). There are two techniques that can be employed by an adversary to carry out such attacks: *flooding* technique, and *vulnerability*-based technique (Loukas, Gan, & Vuong, 2013; Mirkovic & Reiher, 2004).

Flood-based DoS attack

In flooding techniques, an attacker sends a large volume of traffic to a target machine (e.g. a web server) beyond its processing capabilities. This causes the target to consume its resources, comprising CPU time, memory, and network bandwidth. As such, the machine is unable to perform its services in requisite time frames, leading to incapacitation.

A flood-based DoS attack variant uses many machines to simultaneously flood a target computer (Figure 2-5). This is called a Distributed Denial of Service

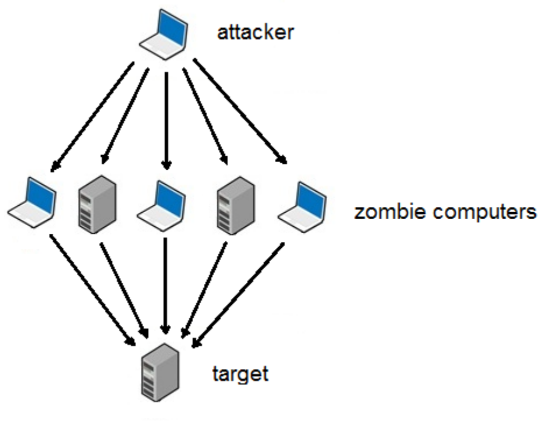


Figure 2-5: DDoS attack

(DDoS) attacks (Chang, 2002). DDoS is a flood-based attack carried out by many machines operating in tandem and generating a large volume of requests towards a target. The technique uses software agents, sometimes called DoS handlers, bots, or botnets, that are illegitimately installed on compromised computers, which are also referred to as zombies. They can launch simultaneous attack traffic towards a victim under the attacker's control, causing a much larger volume of traffic sent to the victim compared to the traditional DoS attack traffic volume.

The victim is incapacitated due to the flood of packets that it has to process, regardless of the distinction between DoS/DDoS technique. In fact, flooding attacks after year 1999 have been mostly DDoS in nature (Zargar et al., 2013). Since DDoS was derived from DoS techniques, the word DoS attacks means both DoS as well as DDoS attacks, in the context of the thesis.

Vulnerability-based DoS attack

Vulnerability is the quality of being easily hurt or attacked. In computing terminologies, 'vulnerability' refers to a weakness in the computing design that could be attacked. In vulnerability-based techniques, an attacker sends crafted information to a target machine. The information exploits the vulnerability existing in the target machine causing service disruption. An example of a vulnerability-based attack is provided in the next section.

It should be noted that a demarcation between flooding and vulnerability-based technique is not clearly outlined in the literature. Some DoS techniques succeed through flooding malicious packets that exploit a vulnerability of a system. The examples are given shortly in the following section. Furthermore, when a vulnerability of a target computer has been patched, the machine can still be subjected to flood-based DoS attacks.

Based on the communication layer it targets, DoS attacks can be divided into network-based and application-based, which is discussed in the following section.

2.2.1 Network-Based DoS Attacks

The World-wide communications are based on the Internet Protocol suite as previously discussed (page 13). In Network-based DoS attacks, adversaries leverage the features of these network protocols. Network-based DoS attacks comprise a flood of network-layer packets that causes service disruption on a target machine. The following discussion details the common and very effective attacks of this type (Northcutt & Novak, 2002; Peng, Leckie, & Ramamohanarao, 2007).

SYN flood attack

When a client machine attempts to establish communications with a server, it initiates a three-way handshake in order to establish the connection. This is a sequence of three TCP packets exchanged between the client and the server. Some TCP flags defined in the TCP header are exchanged as follows: the client sends a TCP *SYN* message to the server; the server responds with a TCP *SYN-ACK* message to the client; and the client responds with a TCP *ACK* message to the server, as is illustrated in Figure 2-6a.

After the three-way handshake, a communication link is established to facilitate bidirectional communications. This design can be abused by a malicious client, where the adversary can send a large volume of SYN packets to a target server, followed by not responding with ACK packets. This is shown in Figure 2-6b. The exploit causes the server to indefinitely wait for the ACK packet after completing the first two steps of the three-way handshake. A large amount of

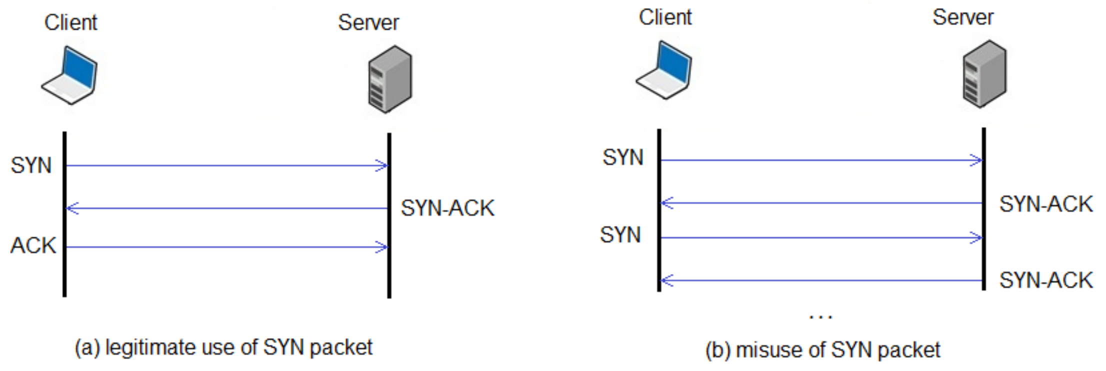


Figure 2-6: SYN flood attack

SYN packets sent, in the absence of the subsequent ACK packets can thus deplete the memory resources of the target machine.

Smurf attack

A broadcast address is an IP address to where a machine can send a packet, in order to reach all machines in the same network. For example, when a new device or computer is connected to a network, it attempts to setup an IP address. It must solicit its IP address from a remote server that supplies IP addresses. It discovers the remote server by sending its request to a broadcast address. Any computer that sends a packet to a broadcast address will effectively broadcast this request to all *hosts* in the network (Figure 2-7a), where a host is a general name for all computers or network devices such as routers and servers. In response to the request, a designated server supplies the IP address to the newly connected device.

The use of broadcast addresses can be abused when an attacker sends a spoofed *ping* packet to the broadcast address. A ping packet is a network packet sent from one machine to another to detect if the remote machine is turned on and can respond to network-based requests. A host that receives a ping packet sends a ping-response packet to the requesting host to signify that the receiving host is alive (Figure 2-7b). This is a convenient tool for legitimate network administrators to audit network connections. When a ping packet is sent from one machine to a broadcast address, all live hosts in the network send ping-response

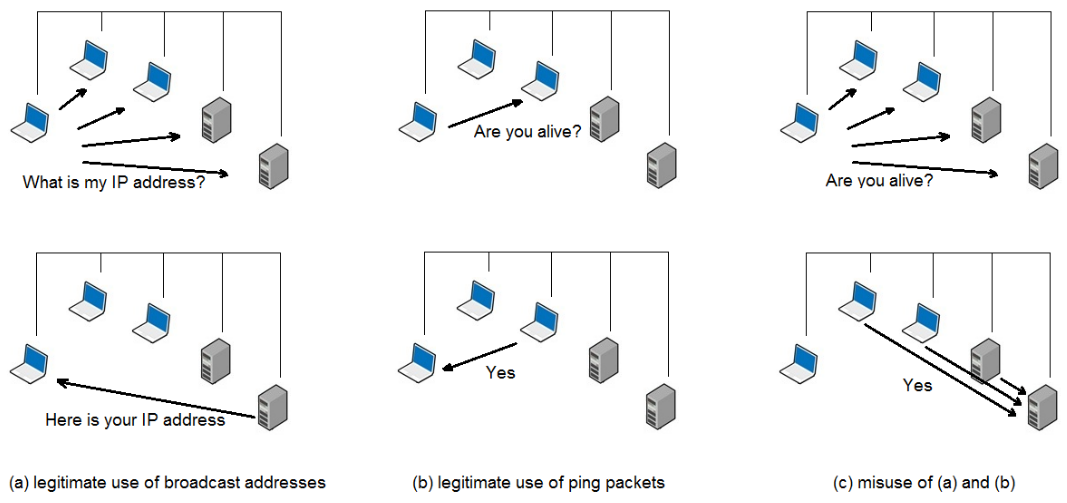

What is my IP address?
Here is your IP address
(a) legitimate use of broadcast addresses
Are you alive?
Yes
(b) legitimate use of ping packets
Are you alive?
Yes
(c) misuse of (a) and (b)

Figure 2-7: Smurf attack

packets back to the source. Smurf attack is a technique where an attacker spoofs its source IP address and sends a large volume of ping packets to a broadcast address. The spoofed source IP address is also changed by the adversary to the address of a target machine, to make the ping appear to be originating from a legitimate target machine. A spoofed ping packet sent to a broadcast address causes all hosts in the network to send ping-response packets back to the spoofed address (Figure 2-7c). A series of spoofed ping packet exploited as such can create a flood of ping-reply packets that incapacitate the legitimate target.

Tribe Flood Network

The Tribe Flood Network attack is another example of a flooding attack that exploits the use of ping packets for malicious purposes. This attack makes use of many compromised computers to generate attack traffic. Malware can be installed on the compromised machines to receive and run remote commands from the attacker. Because the machines seamlessly follow the attacker's command, they are often referred to as zombies. In Tribe Flood Network, ping-reply packets are utilised for issuing commands by an attacker to the zombie machines to initiate a DoS attack. For example, the ping-reply packet causes the malware installed on the zombie machines to simultaneously generate the previously discussed SYN flood traffic. Ping packets are not usually filtered by firewalls, rendering such attack very effective (Northcutt & Novak, 2002).

Loki

Loki extends the capability of the Tribe Flood Network by exploiting the protocol used by ping packets, i.e. the Internet Control Message Protocol (ICMP). The use of ICMP is to aid router administrations in computing the number of hops (network devices) from one machine to another, detailing the network and host status, and checking remote router parameters. The previously discussed ping packets are defined as a pair (i.e. 2^1) of ICMP messages: echo requests and replies. ICMP control header can accommodate a total of 2^{15} instructions. This ICMP header room is what Loki as an attack exploits to have greater command options, and to be successful in carrying out attacks.

Loki software acts as a client-server application, with the server agent being installed on the zombie machines, and the client program interacting with the attacker on scheduled intervals. When a compromised machine runs a Loki server, it executes commands received from a Loki client. An attacker controlling the Loki client can cause the victim machine to display stored passwords, send spam, or act as a DoS handler. As will be discussed in Section 2.3.2.4, sophisticated DoS handlers or bots are part of recent challenges in detecting DoS attacks.

WinFreeze

WinFreeze is a technique used to intoxicate a target machine and to cause failure. The target is remotely configured to flood itself with any network packets that it receives. The technique uses ICMP packets to configure a target machine to forward all network packets it receives to its own address until it is incapacitated.

Legitimately, ICMP packets can be utilised to update the network path of a remote machine in the network, so that it can efficiently reach a destination address. When a machine receives an ICMP-redirect packet, it updates its routing table according to the information that it received from the packet. In Win-

Freeze, an attacker sends spoofed ICMP-redirect packet to a target machine so that the source IP address becomes the target's IP address. In this phase, the target computer becomes a victim. The victim updates its routing table with the information from the packet and it continuously loops back any network packet to itself. A series of network packets sent this way can accumulate and consume the memory or the CPU utilisation of the victim.

2.2.2 Application-Based DoS Attacks

The Application layer provides services to Internet users. Popular examples of several application-layer services are provided above. The discussion in this subsection considers only HTTP-based DoS attacks. The attacks are grouped into two, namely vulnerability-based and flooding-based, within the scope of the thesis.

Vulnerability-based attack

Adversaries exploit the design or programming flaw of the software run on a machine. In web services, attackers can send crafted HTTP packets to attack the target. The target is any machine that runs HTTP services such as a web server, or a router that runs a web service as an interface to configure the routing parameters.

An example of crafted packets are HTTP messages that contain codes, illegal characters (such as "\"), or Null parameters. When the server is not programmed to handle these special situations, it can suffer from a number of undefined behaviours such as infinite loops, runtime errors, buffer overflows, or read/write deadlocks. These behaviours cause the computer to show signs of resource consumption such as high CPU utilisation, high memory consumption, or low network throughput.

Vulnerability-based techniques can attack the weaknesses other machines, while targeting to incapacitate a web server. Web servers are commonly connected to other networked services such as database servers. An exploit can send HTTP requests containing crafted database queries. These queries exploit vulnerabilities that exist in the web-database connector interface (Zargar et al., 2013). As a result, web users observe that the web server does not respond to their requests.

Currently, there are more than 786 known vulnerabilities that are exploited by sending HTTP packets that lead to the unavailability of a computing system (*Common Vulnerability and Exposures: The Standard for Information Security Vulnerability Names*, 2016).

Flooding-based attack

HTTP Request packets can consume the CPU utilisation of web servers. When a web server receives an HTTP Request packet, it parses the HTTP message to prepare for a response web page. The content of a response page is structured according to what the message requests. These requests can cause the web server to search and retrieve a file, connect to a database and update some parameters, or communicate with other application layer services. Afterwards, the web server returns the response web page to the remote users as a by-product of the service. These activities demand CPU utilization at the web server. A large number of HTTP Request packets will incapacitate web servers from future processing (Peng et al., 2007; Zargar et al., 2013). Subsequently, this can exhaust its CPU consumption.

HTTP Requests can also deplete a web server's memory resources. When a web server receives an HTTP Request, it maintains a session to keep information that has been communicated with the client. To illustrate, a session is commenced when a user logs-in to an e-commerce web account so that the user's shopping chart can be tracked. The session is ended when the user logs-out, which is when the shopping cart becomes empty. Without session maintenance, web users can never experience personalised logged-in accounts. In many user-friendly environments, web servers maintain sessions without mandating users to login. Session maintenance consumes a web server's memory. A large number of open sessions can deplete the memory of the web server.

Because session maintenance can incapacitate web servers, some implementations allow web servers to limit the total number of active sessions during a frame of time. However, this security measure can still be exploited. Attackers can send multiple HTTP Requests within a single session until the web server resources are exhausted. Session identification number (Session ID) is always exchanged between a client and a server. In such a scenario, the first HTTP Request causes the server to return Session ID in the HTTP Response message. Attackers can craft subsequent HTTP Request messages to always advertise the same Session ID and send a flood of such messages. This exploit can bypass the security control that limits the allowed number of sessions and incapacitate the web server.

Another web security measure includes releasing previously opened, unused sessions. Web servers close sessions where no HTTP Request is received within a predefined time. An example of this can be seen with electronic banking websites that log the users off after a period of idle time. Other than to protect users' data, this measure is considered as a technique to maintain the amount of available memory. However, a DoS attack technique can bypass this measure. Adversaries can learn the Session ID as previously described to open a session and send HTTP Requests at a rate higher than the server predefined idle period time, to maintain the opened session. When this procedure is repeated with multiple sessions, the web server eventually depletes its memory.

Some web services require CPU-intensive tasks and therefore DoS attacks can succeed by launching a series of requests (Crosby & Wallach, 2003). Web servers are commonly connected to other networked services such as database servers, mail exchange servers, voice call services, and interconnection services to other platform or software. Some services require the CPU to process computationally demanding tasks. As such, the web server is more susceptible to DoS attacks.

2.2.3 Discussion

There are two observations from the understanding of DoS techniques. First, regarding the type of DoS attacks (i.e. flooding and vulnerability), and second, regarding the DoS variants (i.e. DoS and DDoS).

It can be shown that the boundary between the two types of DoS attacks – the flooding and vulnerability-based techniques – is not always obvious. For example, the SYN flood was often illustrated as an exploit to an implementation bug (Mirkovic & Reiher, 2004; Moustis & Kotzanikolaou, 2013), while it is only effective when a flood of such packets is launched (Zargar et al., 2013).

Similarly, the boundary between DoS and DDoS seems not to be essential when analysing traffic. For example, in WinFreeze, it can be seen that an attacker does not decide on the number of zombie computers it requires to successfully flood a target. The victim is incapacitated through the network packets it floods itself. When the victim receives packets from machines across the Internet, it sends response packets that are routed back to itself. Because this process loops indefinitely, one packet sent from a legitimate user to a victim can incapacitate the victim. A web server usually serves many, rather than one user; when the victim is a web server, any number of clients can send Request packets to the victim. Hence, any number of clients can eventually flood the victim, regardless of the distinction between DoS and DDoS.

2.3 DoS Detection Techniques

This section reviews DoS detection techniques as found in the literature. It is organized into two parts according to the detection technique, i.e. vulnerability-based techniques and flooding-based techniques. Because this thesis presented HTTP/2 flooding techniques, the second part of this section meticulously discusses the research reported in the area. The review on the detection methods against flooding techniques presents both network and application-based DoS detection methods, elaborating on how research in DoS detection using machine learning techniques remains active, and analyses the challenges to detecting application-based DoS attacks.

2.3.1 Detecting Vulnerability-Based Attacks

The most effective measure to detect and prevent known vulnerability DoS attacks is through patching the vulnerability on the target system (Moustis & Kotzanikolaou, 2013). Vulnerable web servers are patched through replacing vulnerable server modules or by upgrading the entire server software to a more secure version.

Although it might be viewed as a straightforward solution, Internet-connected web servers are not always rebuilt to run on the latest software version. The activity of patching and upgrading software itself can disrupt a service. Maintaining continuous web services for this purpose requires extra hardware, knowledgeable operators and a planned-work schedule which altogether can prove to be costly. Hence, companies often adopt other means to prevent vulnerability-based DoS attacks. These include detecting known attack signatures and blocking the packets that match the signatures, configuring server session timeout, or using firewalls to limit the number of suspected connections.

Traditionally, signature-based detection was found to be administratively prohibitive. For example, when a software was vulnerable due to its inability to parse and process irrelevant characters in network traffic, such as a back-slash, the "\" signature is recorded to block the incoming packets that contain such characters. This implied that new signatures were to be updated when new exploits were understood, and therefore the server remained vulnerable when new exploits were discovered. Furthermore, good quality signatures can be complex (Paxson, 1999) as new signatures must cover the highest abstraction of an attack to represent all variations of the attack. This requires not only expert knowledge but also sufficient amount of data to analyse.

Since identifying signatures is non trivial, signature-based detection methods suffer from false-positives, i.e. some legitimate traffic is identified as an attack. This prompts research in the area remains active such that new techniques are proposed to reduce false alarm rates. In fact, signature-based detection methods were criticised to overemphasise on reducing false alarm rates, at the cost of algorithm complexity for gaining real-time performance (Hubballi & Suryanarayanan, 2014).

2.3.2 Detecting Flooding Attacks

Flooding DoS attack traffic does not necessarily rely on signatures. Attackers generate fictitious traffic packets of a large volume. Hence, many detection techniques in the literature model normal traffic patterns in order to differentiate and detect anomalous traffic patterns. Statistics have been used to prove the significance of an observation and how it differs from the defined model. The rest of this section discusses the detection techniques for flooding attacks, further divided into network-based and application-based. It provides a comprehensive background on the application-based flooding technique to review related research to the subject in this thesis, i.e. HTTP/2.

2.3.2.1 Detection Techniques for Network-Based DoS Attacks

Crafting the content of network packets (as described in Smurf attack and Win-Freeze) implies that anomalous traffic presents varying patterns, as is evident from packet header information. Hence, measuring the statistical properties of network packet headers was proposed to detect anomalies in crafted packets (Feinstein, Schnackenberg, Balupari, & Kindred, 2003). The authors used source IP, destination IP, and destination port as features, and measured the randomness/uniformity of the distribution of a network flow with reference to a specific feature (e.g. entropy of source IP). The results obtained showed that source IP and destination port are features that can effectively be used to detect DoS traffic. The method was highly accurate when applied on a large dataset that represented core Internet traffic, but degraded when applied on smaller datasets that represented smaller networks.

Many detection solutions were deployed on the routers of the network providers. These routers monitored bandwidth usage from traffic that passed through them, grouped based on destination addresses. One method was to measure normal bandwidth against anomalies (Chan et al., 2006). Hence, Smurf attacks were detected when showing statistically high number of ICMP packets when compared to TCP packets; and SYN flood attacks were detected when the number of SYN packets did not balance with the number of ACK packets.

Routers often become the subject of DoS attacks and several studies were proposed for attack detection on routers. Traffic validation was proposed to monitor anomalies in the network (Mizrak, Savage, & Marzullo, 2008). Such an approach asserted that attacks were detected when the property of ingress traffic was significantly different from traffic leaving the network. Specifically, the study monitored the flow, content, packet order and timeliness of the packets that passed through routers. The authors of the study hinted that the accuracy of the results could be undermined in actual settings where router failure occurrences were common. A similar approach that monitored incoming and outgoing packets through routers was proposed to detect anomalous traffic when the IP address was spoofed (Gonzalez, Anwar, & Joshi, 2011). The study introduced a reliable router to monitor traffic passing through other entry points to the network.

2.3.2.2 Detection Techniques for Application-Based DoS Attacks

This section explains techniques found in the literature to detect application-based DoS attacks. The broader area of study to identify flooding-based anomalous traffic falls under the study of traffic analysis. The purpose of the traffic analysis exercise is to identify traffic characteristics that portray standard application operations. Identifying the types of traffic that flows in networks is important for network operators and researchers: network operators often sought to only serve traffic that is legitimate and differentiate it from illegitimate. Anomalous traffic is generated by viruses and worms, and traffic that consumes bandwidth such as DoS/DDoS attack traffic is also malicious by class. Characterising application types within a network can help categorise traffic into legitimate and anomalous.

Traffic could be classified based on its flow, i.e. the number of packets and the size of bytes being transferred. One of the advantages of this approach is that only one direction of network flows need to be observed (Erman, Mahanti, Arlitt, & Williamson, 2007). This is useful when only one side of the communicating equipment in a network was accessible. For example, UDP traffic is unidirectional by nature, rendering traffic analysis methods that rely on bidirectional flow information ineffective. This approach could be thoroughly independent of TCP/IP information (Dyer, Coull, Ristenpart, & Shrimpton, 2012; Tang, Lin, & Luo, 2014), making it suitable for classifying applications and anomalies caused by their unintended use. For example, a resource-consuming Peer-to-Peer application that uses web-traffic port number 80 can be accurately identified through analysing the flow of one or more packets transmitted between a given pair of hosts (Auld, Moore, & Gull, 2007).

Traffic could also be analysed through its social and functional pattern, or through the behaviour of hosts. For example, a scheme was proposed to detect application types that resided within a network(Karagiannis, Papagiannaki, & Faloutsos, 2005). The study constructed the behaviour of a host from its IP address, port number, flow, and size of the packets. The work was able to identify traffic types or applications through how each host communicates with other hosts. For example, in DNS lookup activities, many hosts were involved in forwarding packets to other hosts; in web communications, fewer hosts were communicating in both directions of the communication channels; in mail services, hosts cascaded packets from one host to another; and in anomalous behaviour, many hosts communicated with one single host within a short time period. Analysing the behaviour of these hosts was effective when the observer had access to both sides of the communicating equipment (i.e. a pair of computers that communicate to each other, as in client-server communications).

Inspecting the application-payload packets could also help accurately classify their application types. Payloads are the content of any network packets. One study (Moore & Papagiannaki, 2005) aimed to address the urgency of network operators to identify legitimate customer traffic and those that misused network resources, and effectively, bandwidth. The challenge to identify legitimate traffic is that customer traffic patterns change based on emerging applications. By the same token, the challenge to identify illegitimate traffic is that attacks can use a legitimate protocol to carry their malicious instructions. The study was able to identify up to 79% of malicious traffic, where only 1 KB of each packet was examined. When the packets were parsed to classify the payload messages (e.g. FTP control), up to 98% of the traffic was identified. A near 100% traffic identification was thus achieved when the entire payload was examined.

Statistics on the traffic captured from the real networks have been reported in many studies to monitor traffic activity. In published methods, DoS attacks were defined as activities above defined thresholds of a legitimate activity. Traffic models could be constructed from the average packet rates and inter-arrival times between consecutive packets for legitimate traffic. A subsequent comparison of observed traffic with captured traffic flows facilitates anomaly detection. In order to reduce high-dimensionality of the observed traffic data, a proposed method (Jung, Krishnamurthy, & Rabinovich, 2002) grouped or clustered similar packet fields such as address, port, and protocol. Increasing levels of activity within clusters indicate DoS attacks in action.

Entropy (Shannon, 2001), defined as the degree of concentration or dispersion of random information, has often been used in statistical methods for detecting DoS attacks. A large entropy value implies large uncertainty of the random variables being studied. In detecting DoS attacks, this property of showing distributional changes was found to be superior to methods that only measure volume changes (Feinstein et al., 2003; Lakhina, Crovella, & Diot, 2005). Statistics-based detection methods use entropy to identify applications in heterogeneous network which transport large-volume applications (Petkov, Rajagopal, & Obraczka, 2013), and distinguish DoS from sudden increases in volume of legitimate traffic (Ma & Chen, 2014; Rahmani, Sahli, & Kamoun, 2012; Sachdeva & Kumar, 2014).

Wavelet analysis has been used generally for addressing optimisation problems in the signal processing domain. The technique was found in the literature to distinguish anomalous traffic (Barford, Kline, Plonka, & Ron, 2002; W. Lu & Ghorbani, 2009), and DoS attacks (Dainotti, Pescape, & Ventre, 2006; L. F. Lu, Huang, Orgun, & Zhang, 2010), with high effectiveness.

Some DoS mitigation approaches found in the literature identify malicious clients instead of differentiating traffic. Once a client is identified as malicious, its access to the server is blocked allowing legitimate traffic to pass. One approach (W. Meng, Li, & Kwok, 2014) performed identification of malicious clients through its 'IP reputation' calculation. When a client alerted an intrusion-detection system, its source IP address was recorded and monitored. For all of the client's subsequent packets, the technique adjusted the IP reputation according to the ratio of packets that alerted and passed the intrusion-detection system. Once the IP reputation surpassed a threshold, the technique blocked subsequent traffic that originated from the source IP address being monitored. Similarly, some techniques mitigated DoS through blocking illegitimate source addresses (Ferguson, 2000) and spoofed IP addresses (Chen, Das, Dhar, El-Saddik, & Nayak, 2008; Park & Lee, 2001; Savage, Wetherall, Karlin, & Anderson, 2000).

Another approach proposed in the literature identified legitimate clients through a challenge-response technique. For example, some approaches deployed a client puzzle protocol where a client machine must solve a cryptographic challenge sent by a server to identify itself as a legitimate client (Juels & Brainard, 1999; Stebila, Kuppusamy, Rangasamy, Boyd, & Nieto, 2011). In this approach, clients capable of solving a problem were granted access to a server, while denying other traffic such as DoS attacks. The drawback of this approach was that responding to cryptographic challenges could be automated and thus compromised by attackers. Other approaches require a CAPTCHA – challenge that require human end-users to type a series of letters illustrated through a distorted image – to justify that the client is human (Gligor, 2005; Kandula, Katabi, Jacob, & Berger, 2005). Client machines were tagged as legitimate after they had passed a CAPTCHA test. In this approach, a finite number of legitimate clients were served when large traffic volume was detected, thereby guaranteeing access to human users.

Machine learning techniques are popular in solving network traffic analysis problems, since they are able to classify large data spanning multi-dimensional spaces. Due to its important role in addressing the gaps shown in this thesis, this method is broadly discussed in the next section. It reviews different techniques based on machine learning techniques, and shows how they are applied for detecting DoS attacks.

2.3.2.3 Anomaly Detection Techniques using Machine Learning Techniques

This subsection discusses machine learning techniques as part of the methods used in detecting traffic anomaly. Machine learning methods are divided into two types: supervised and unsupervised. In supervised learning, training data is labelled according to class. Labelling data involves tagging the examples in the training dataset, for example into legitimate and attack class. Supervised learning techniques learn from the labelled input examples, and produce a classifier that can be used to map unseen data into one of the two previously defined classes. These classifiers can be represented in terms of a set of rules. Intrusion-detection systems that employ supervised learning techniques can be equipped to contain such rules, filtering instances according to the values of the different features, and therefore can classify new instances, for example into "legitimate" or "attack".

Unsupervised learning techniques find statistical relationships among instances of the data, and classify instances based on how strongly they correlate. The techniques do not require labelled training data for learning; rather, they learn from a probabilistic model of the data. Intrusion-detection systems that employ unsupervised learning methods are equipped with some statistical parameters, such as learning rate, and those used to calculate error measurements between a new instance and the rest of the data. Instances that are statistically different from the others are considered as belonging to another class, or anomalous.

It is not always clear to define the boundary of what constitutes machine learning methods. For example, the Naïve Bayes techniques are introduced when discussing machine learning in some techniques (Witten & Frank, 2005; Z. Yu &

Tsai, 2011), while it is referred to as Probabilistic Learning in another reference (Bhattacharyya & Kalita, 2013). Rather than redefining what constitutes machine learning techniques, this work follows Witten and Frank's definition that many statistical and probabilistic methods can be regarded as machine learning methods, since "these perspectives have converged" (Witten & Frank, 2005, p.29). Machine learning techniques involve feature selection and data visualisation which use statistics of data to construct classifier models. In selecting features, statistics tests are used to find the degree of coherence when a feature of a data sample is selected. This allows features to be ranked according to their relevance. Many machine learning techniques benefit from ranked features to reduce dimensionality and increase performance. The next subsection discusses examples of machine learning techniques.

Examples of machine learning techniques

This subsection is to introduce the most commonly used machine learning techniques in identifying attacks or traffic anomalies.

Naïve Bayes

Naïve Bayes is one of the most widely used techniques in data mining communities (X. Wu et al., 2008). The name originates from its nature to naïvely assume that the features originate from independent events. Despite this assumption, the technique works surprisingly well when tested on actual datasets (Witten & Frank, 2005, p.91).

The technique is a conditional probability model. It is the probability of an event given that another event has occurred. Stated formally, it is the probability of a data sample to belong to a class C given a new instance x is seen. In Naïve Bayes, this is commonly called the *posterior* probability:

$$p(C_k|x_1, \ldots, x_n) \tag{2.1}$$

where k is the number of classes, n is the number of features that represent

Table 2.2: An example of traffic samples characterised by three features

Class	cart	today	send	# examples
e-commerce	400	350	450	500
email	0	150	300	300
news	100	150	50	200
Total	500	650	800	1000

x. For each possible class k, the highest posterior probability is assigned to the new instance.

The technique is first applied to a training dataset to construct *prior* probabilities. The posterior probability defined in equation (2.1) is calculated by finding the *likelihood* to belong to a class, based on a *prior* observation of the *evidence*. This is formulated as follows:

$$posterior = \frac{prior \times likelihood}{evidence}$$

An example can illustrate how this technique can classify an unknown instance. The following example is adapted from Witten & Frank (Witten & Frank, 2005, p.88). Suppose that a web page must be categorised if it were an e-commerce, an email service, or a page containing sports news based on the words it contains. The category of the web page should be characterised by three words: "cart", "today", or "send". To serve as a prior evidence, suppose a researcher collected statistics from a survey. The results are summarised in Table 2.2. In this example there were 1000 web pages surveyed, of which there were 500 e-commerce pages, 300 email pages, and 200 news pages identified.

In this example, the table was the training dataset, the web-page types were the classes, and the three words were the set of features. The total number of web pages surveyed was the total number of examples or instances. The first row after the table header shows that of the 500 e-commerce pages surveyed, 400 of them contained the word "cart", 350 had the word "today", and 450 pages showed "send" words. Other rows are to be explained similarly.

If a new, unknown web page showed "cart", "today", and "send" words on its page, Naïve Bayes can be used to determine the class of this new instance. The pre-computation is done as follows.

The *prior* probabilities are the portion of the classes over all instances. Hence, in this case,
$P(ecommerce) = 500/1000 = 0.5$
$P(email = 300/1000 = 0.3$
$P(news) = 200/1000 = 0.2$

The probability of evidence values are not carried in the final calculation for two related reasons. First, they are independent of the class so their values are effectively constant. Second, the final calculation compares the posterior value of the classes, so the probability of evidence as a constant cancels out. The probabilities of *evidence* are shown here for illustration purpose:
$P(cart) = 500/1000 = 0.5$
$P(today) = 650/1000 = 0.65$
$P(send) = 800/1000 = 0.8$

The probability of *likelihood* is the joint probability of each word (i.e. "cart", "today", and "send") given a class (e.g. e-commerce), or the product of $P(x_i|C_k)$ for all $1 \leq i \leq n$. This means the number in each cell in Table 2.2 was divided by the total number of the class' examples. The likelihood probability of a class is the product of these values:
$P(cart|ecommerce) = 400/500 = 0.8$
$P(today|ecommerce) = 350/500 = 0.7$
$P(send|ecommerce) = 450/500 = 0.9$
$likelihood(ecommerce) = 0.8 \times 0.7 \times 0.9 = 0.504$

Table 2.3 shows the posterior probability on the right-most column. It shows that the e-commerce class returned the highest number. Hence, the new instance was classified as an e-commerce web page.

In classifying traffic, Naïve Bayes technique can be used to compute the probability of an instance belonging to a normal traffic class or attack class, given

Table 2.3: An example of traffic samples characterised by three features

Class	cart	today	send	likelihood	prior	posterior
e-commerce	0.80	0.70	0.90	0.504	0.5	0.252
email	0.00	0.50	1.00	0.0	0.3	0.000
news	0.50	0.75	0.25	0.094	0.2	0.019

some observed features such as packet flow and size of traffic instances. Although it is a classic classifier (Pearl, 1988), Naïve Bayes is used effectively in many recent studies in traffic analysis and DoS detection (Katkar & Kulkarni, 2013; Moore & Zuev, 2005; Mukherjee & Sharma, 2012; J. Zhang, Chen, Xiang, Zhou, & Xiang, 2013).

Traffic analysis is classifying traffic to its application service types such as mail, web, database, games, multimedia, and also attacks. Precise classification is essential for network operators to determine the policies for its quality of service. A study used Naïve Bayes to analyse and classify traffic without inspecting the payload/content of the traffic (Moore & Zuev, 2005). The dataset was obtained by extracting features from the TCP headers of the observed traffic. The features were packet flow, lapsed time between consecutive flow, port numbers, and packet size. The study showed that it achieved accuracy of 95% in classifying the traffic through application of Naïve Bayes.

One of the biggest challenges in classifying traffic is the large amount of data to analyse, given a set of features. To illustrate, Table 2.3 can become excessively large and computationally expensive with increasing number of data samples and features. A technique was proposed to rank relevant features so that only a subset of the original features were used for the purpose of classifying attack and normal traffic using Naïve Bayes (Mukherjee & Sharma, 2012). Feature selection thus reduced the dimensionality of the data and therefore offered faster computation. To compare the performance of the proposed technique, other widely used feature ranking techniques, such as Information Gain and Gain Ratio, were applied by the authors. These techniques are also used in this study and are discussed in detail in Section 3.1.4. The proposed technique in the study showed that it outperformed the widely used feature ranking techniques when selecting large volumes of data.

In contrast, another study proposed a solution when too few data sample were available (J. Zhang et al., 2013). Due to a large number of emerging applications, it was difficult to collect a large number of related data samples. The study proposed a technique to pre-process traffic before extracting features to be classified using Naïve Bayes. The technique correlated traffic flows that were generated from the same application. Overall accuracy and F-measure were used as evaluation metrics. Overall accuracy is the ratio between the number of correctly classified instances to the total number of instances to predict. F-measure is the harmonic mean of precision and recall; precision is the ratio between the number of correct positive results and the number of all instances identified as positive; recall is the ratio between the number of correct positive results and the number of positive instances. The study showed that the proposed method outperformed other machine learning techniques such as Decision Tree and k-NN in terms of overall accuracy and F-measure.

Decision Trees

Decision Tree is one of the most popular techniques applied for data classification (X. Wu et al., 2008). For classification, Decision Tree is a sequence of rules in which the current selected rule decides the subsequent rules to be selected by splitting into two or more branches forming a tree-like structure. Decision Trees learn from training samples to form a tree with its end leaf nodes signifying the classes of data such as normal or attack. A new data instance can be classified through traversing the rules from the root of the tree until an end leaf node is reached.

Decision Tree is a simple but an outstanding technique for explaining the relationship between the input instance characteristics and its target class (Rokach & Maimon, 2014). Each instance is characterised by a series of features. The root of a tree is chosen based on certain split functions of the training sample features. A feature that can produce the best split value is chosen, with its split rules determining the number of child nodes. This process is repeated on every node until only one class is found on each child node.

Table 2.4: An example of a labelled traffic dataset characterised by three features

flow	lapse	size	class
low	short	big	normal
low	short	small	normal
high	short	big	attack
medium	short	big	attack
medium	long	big	attack
medium	long	small	normal
high	long	small	attack
low	short	big	normal
low	long	big	attack
medium	long	big	attack
low	long	small	attack
high	short	small	attack
high	long	big	attack
medium	short	small	normal

An example can illustrate how a Decision Tree chooses the most relevant feature to split, and iterates the process until a class is assigned at each of the end leaf nodes. The following example is adapted from Witten & Frank (Witten & Frank, 2005, p.97). Suppose a dataset as shown in Table 2.4 describes a 15-second captured traffic. Each row represents a sample of the captured traffic at 1 second, characterised by some features. These features are the columns of the table, i.e. packet flow, the time lapse since the last packet of its type was seen, and the size of the packet. The last column labels the class of each data sample.

Table 2.4 provides the raw data for further processing such as counting the tally of the features and dividing those values with the total number of occurrences to compute the probabilities. The counts and their probabilities are shown in Table 2.5. Examining the first feature in this table, the *flow* feature showing the number of attack and normal classes are (2,3), (3,2) and (4,0) respectively. Hence, when the flow feature was to be treated as a node to split, it would generate three rules to split:

- if the flow was low, then the number of attack and normal classes was (2,3).
- if the flow was medium, then the number of attack and normal classes was (3,2).

Table 2.5: The counts and probabilities of the features

flow			lapse			size			class	
	attack	normal		attack	normal		attack	normal	attack	normal
low	2	3	short	3	4	big	6	2	9	5
med	3	2	long	6	1	small	3	3		
high	4	0								
low	2/9	3/5	short	3/9	4/5	big	6/9	2/5	9/14	5/14
med	3/9	2/5	long	6/9	1/5	small	3/9	3/5		
high	4/9	0/5								

- if the flow was high, then the number of attack and normal classes was (4,0).

In Decision Trees, a measure of split purity is applied to find the most relevant feature. This example uses Information Gain as the measure, whose formula is explained in Section 3.2. The higher the *Gain* value of a feature is, the more relevant the feature for data classification.

Decision Tree firstly finds the degree of coherence or dispersion of the *Information* at a given node. The Information represents a mix of the observed attack and normal classes through a number, in bits. The higher the Information value is, the higher the degree of dispersion. Using this measure, the information values of each node after the flow-feature split are:
$Info((2,3)) = 0.971$ bits
$Info((3,2)) = 0.971$ bits
$Info((4,0)) = 0$ bits

The Information value of the flow feature, or $Info(flow)$ is calculated as follows:
$Info((2,3),(3,2),(4,0)) = (5/14) \times 0.971 + (5/14) \times 0.971 + (4/14) \times 0) = 0.693$ bits

The examples in the training dataset had 9 attacks and 5 normal classes. Therefore, the information value of the training dataset, or $Info(training)$ is:
$Info((9,5)) = 0.940$ bits

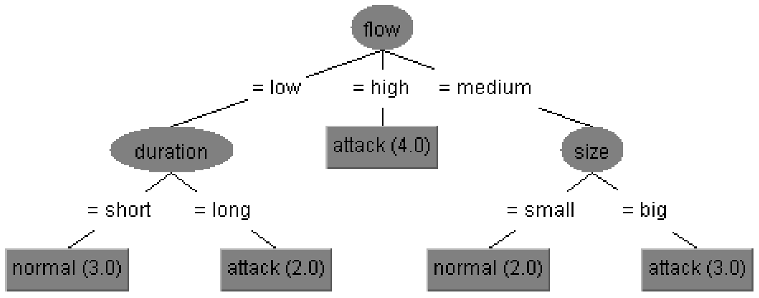


Figure 2-8: A Decision Tree

The Information Gain of the flow feature is:
$Gain(flow) = Info((9,5)) - Info((2,3),(3,2),(4,0)) = 0.940 - 0.693 = 0.247$ bits

The Information Gain is calculated against each feature resulting in:
$Gain(lapse) = 0.152$ bits
$Gain(size) = 0.048$ bits

Because the gain value for 'flow', $Gain(flow)$, yielded the highest number, the flow feature was selected as the final node by the decision tree technique. This process is iterated for each feature. The result of this example is shown in Figure 2-8. The tree can be used to analyse new, unknown instances by traversing down according to the values of the new instance features. The new instances eventually find their class when they reach a leaf node.

It can be seen from the illustration that the classification of a new instance is computationally inexpensive since it simply follows a series of pre-constructed rules, and therefore is suitable for real-time applications. This advantage prompted a study to use Decision Trees to quickly identify DoS traffic for the purpose of tracing back the attack source. (Y. C. Wu, Tseng, Yang, & Jan, 2011). The result presented show that Decision Tree yields 1.2 - 2.4% false positive rates and 2 - 10% false negative rates in detecting such attacks.

Decision Trees have also been applied in traffic analysis research for identifying botnets (Haddadi, Morgan, & Zincir-Heywood, 2014) and network anomalies

(Swamy & Lakshmi, 2012). Botnets are not only deployed to generate DDoS traffic, but also help spread spam mails and to steal passwords. A study used a Decision Tree to identify botnet behaviour from traffic patterns it generated (Haddadi et al., 2014). The scheme compared its performance analysis with Naïve Bayes and concluded that Decision Trees can produce better classification accuracies. Interestingly, higher detection rates were achieved when analysis was done on HTTP-filtered traffic, wherein only HTTP traffic was being detected. The study suggested that the bots communicated using the HTTP/1.x protocol.

Rule-Learning Techniques

Rule-learning techniques recursively seek to find a rule that can identify a class. A rule is defined as a set of feature values that can maximise an accuracy measure, e.g. incorrectly classified instances, for classification. Although rule-learning techniques can yield the same split rules as Decision Trees, the approach is different. Decision Trees seek to find a feature value that best split the classes; rule-learning techniques consider one class and seek a set of rules that covers that class. Additional rules can be sought iteratively to classify the remaining incorrectly classified instances.

The advantages of rule-learning techniques are that rule sets are relatively easy for human to understand, and are natural for programming languages to implement. Certain rules obtained from prior knowledge can therefore be programmed into rule-learning systems. However, rule-learning techniques do not scale with sample size. Therefore, a study proposed "repeated incremental pruning to produce error reduction" (RIPPER) as an extension to rule-learning techniques (Cohen, 1995). It introduced a pruning method that reduces the complexity of a tree. Pruning is a method that disregards computation to some branches of a tree. RIPPER terminates further rule computations when the last constructed rule yielded error larger than 50%.

JRip is a RIPPER implementation in Java programming language. JRip is considered faster than Decision Trees (Cohen, 1995; Mohamed, Salleh, & Omar, 2012). Because JRip is fast, it is suitable for real-time data analysis. It was used in traffic analysis studies to reduce false alarms (Gaonjur, Tarapore, Pukale, & Dhore, 2008), to select features (Yang, Tiyyagura, Chen, & Honavar, 1999) and to efficiently reduce the amount of data for an intrusion-detection system (Panda, Abraham, & Patra, 2015).

k-Nearest Neighbour

The k-Nearest Neighbour (k-NN) technique learns directly from the supplied data as the technique iteratively constructs neighbourhoods from data samples. The learning is accomplished at the same time as when classifying a new data instance. Assuming that data points represent previously observed examples, the technique computes the distance between the new instance and the rest of the data points in classification. This is illustrated in Figure 2-9. The black dot is an instance of data that is to be classified. The squares represent data points of a class, while the hexagons represent those of another class.

The distance metric used is the Euclidean function, which is defined as:

$$E(X,Y) = \sqrt{\sum_{i=1}^{m}(x_i - y_i)} \tag{2.2}$$

In equation (2.2), X and Y are two multidimensional data points characterised by the features; X is the new data instance and Y is the previously observed sample, m is the number of features, and x_i and y_i are the input values for feature i.

Euclidean distance is not the only means used as a distance measure. Other techniques include Hamming, Cosine, Chi-square, and hyper-rectangle distance measure (Randall & Martinez, 2000).

Classifying the data is done through assignment of the new data instance to the closest k neighbours of the plot. The final decision is taken by collecting the votes from all k neighbours. The new instance is classified according to the majority vote of neighbours. Therefore, the new instance in Figure 2-9 would be classified as a hexagon (class 2) when $k = 3$, or as a square (class 1) when $k = 9$.

On one hand, the technique can be computationally complex (Kotsiantis,

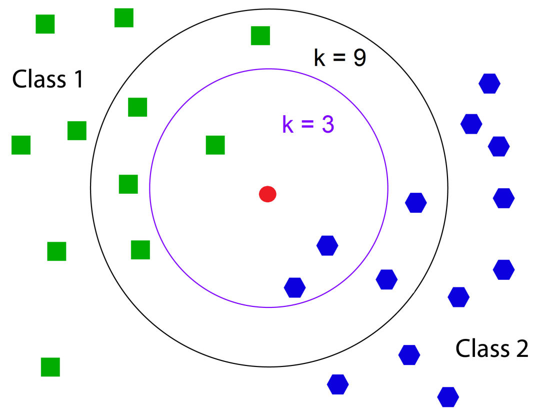


Figure 2-9: k-NN finds the most vote from the closest k neighbours

Zaharakis, & Pintelas, 2006), depending on the distance function used and the value of k. On the other hand k-NN does not require a training phase, but rather delays the computation until a new instance is to be classified. Therefore, in anomaly detection, the technique has been used to detect anomalies in real-time, allowing detection systems to proactively respond to intrusions (Tsai, Hsu, Lin, & Lin, 2009; Su, 2011; Guo, Krishnan, & Polak, 2012).

Support Vector Machines

Support Vector Machines (SVMs) (Vapnik & Vapnik, 1998) are an example of a supervised learning technique. They extend linear regression models to separate datasets whose classes are otherwise linearly separable. Suppose that a task requires separation of data points into two classes, as illustrated in Figure 2-10a, where data points are shown as the black and white dots. SVMs seek to find a line (or referred to as a 'hyperplane' when used in multidimensional spaces) that separates the two classes with the largest distance to the data points. The largest distance is called the maximum margin. H1 and H2 are possible hyperplanes, while H3 is the maximum margin hyperplane.

Data points that are closest to the maximum margin hyperplane are referred to as support vectors. This is shown in Figure 2-10b.

SVMs have been used to classify DoS traffic and normal traffic in a recent study (Sharma & Parihar, 2013). The study aimed to detect DoS in mobile Ad-

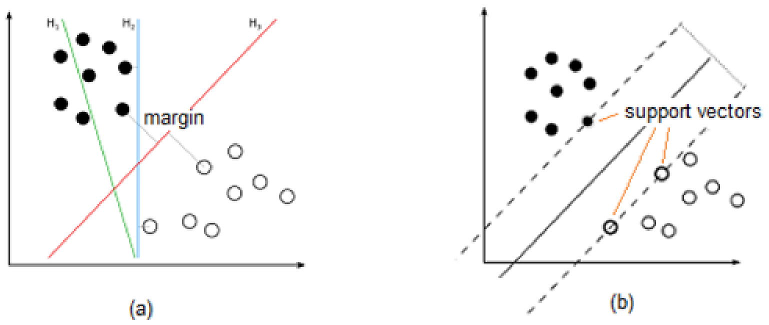


Figure 2-10: (a) H3 is the maximum margin hyperplane, (b) Support vectors

hoc networks, which is a network of mobile devices that are connected wirelessly, wherein each device acts as a router in addition to its normal intended use. Such kinds of networks are very vulnerable to DoS attacks since there is no security policy imposed on the connected devices. The study showed that SVMs classify with greater than 90% accuracy in all conducted experiments.

SVMs can also separate multiclass data. For example, a study reported in (Li, Yuan, & Guan, 2007) aimed to classify 7 application types of traffic (bulk, interactive, mail, service, www, p2p, and 'others'). Prior to this study, traffic classification were focused only on HTTP data. It yielded 96.9% accuracy in classifying the mixed traffic. The study showed that the same method was able to classify homogeneous traffic comprising 87% HTTP traffic with 99.4% accuracy. This suggests that the method was externally valid, i.e. it can achieve a desired accuracy level when applied to different sets of data.

K-means

An example of an unsupervised learning technique is k-means. This technique is initialised with k number of groups, or clusters, according to the desired number of classes (e.g. $k = 2$ for classifying traffic to 'normal' and 'attack'). Then, k random points in the data are chosen. These points are treated as the initial centre of the clusters. All other data points are assigned to their closest cluster centre and become the member of that cluster. Afterwards, the means of the data points in each cluster are calculated to choose new centre points. The process is repeated with new centre points covering different cluster members, until all centre points are stabilised as the final ones.

Since k-means does not require a training step, it could be applied to classify traffic in real time. For example, a study reported in (Bernaille, Teixeira, Akodkenou, Soule, & Salamatian, 2006) aimed to identify the categories of traffic of a large university network. The administrator of the network wished to block traffic responsible for music file sharing and gaming. These types of traffic did not use universally registered port numbers, rendering identification methods that map port-numbers-to-application ineffective. The study used k-means to identify applications such as edonkey, ftp, http, kazaa, nntp, smtp, ssh, https, and pop3s with accuracy ranging between 81.8% - 99.6%.

Another study applied the ability of k-means to detect intrusions more accurately than a previous method (Elbasiony, Sallam, Eltobely, & Fahmy, 2013). It modified the k-means technique into weighted k-means in order to assign higher weight values to more important features of the data samples. The finding showed that the detection rate was 98.3% with 1.6% false positives. This result was better than a previous study used as a comparison (Ryan, Lin, & Miikkulainen, 1998) which yielded only 94.7% detection rate with 2% false positive rate.

Artificial Neural Networks

Artificial Neural Networks (ANNs) are inspired from how human-brain neurons work (Haykin & Lippmann, 1994). They consist of 'neurons' that learn from data and recognise patterns of unseen data. ANNs assign weights to the neurons, and adjust them as it learns from each training instance. The learning process seeks to reduce the error in misclassification. ANNs can be branched into supervised and unsupervised learning (S. X. Wu & Banzhaf, 2010).

In supervised learning, the neurons learn from labelled data, i.e. instances whose classes are already known. Supervised neurons are generally modelled as shown in equation (2.3). Here Y is the output whose value can be positive or negative as indicated by the *sign* on the right hand side of the equation. Positive Y values indicate one class, while negative values indicate another. X is a set of n features. The value x_i of feature i is weighted with w_i for all n number of features. The model shows that neurons are weighted sums of inputs with a threshold value θ. The threshold value can be used to adjust decision boundaries.

$$Y = \text{sign}[\sum_{i=1}^{n}(x_i w_i) - \theta] \tag{2.3}$$

The weights are initialised with random values. Neurons learn by adjusting the weights so that the output value Y produces the correct class given input values x_i. That is, a data instance represented by its features is used to train the neurons through supplying the feature values x_i to the equation. If Y outputs a different class than the label, then the weights w_i are adjusted at its following iterations through equation (2.4).

$$\begin{gathered} e(p) = Y_d(p) - Y(p) \qquad \text{where } p = 1, 2, 3, \ldots \\ w_i(p+1) = w_i(p) + \alpha \times x_i(p) \times e(p) \end{gathered} \tag{2.4}$$

In each iteration, error values from the Y output value to the desired Y value is calculated. The error $e(p)$ at each iteration p is obtained from subtracting the output $Y(p)$ from the desired output $Y_d(p)$. This yields a positive or negative vector. Upon the following iteration (p+1), the weight w_i is increased or decreased according to the error vector with a magnitude α. Thus, the weight $w_i(p+1)$ learns (i.e. adjusts its value) from the error between the current and the desired output. This step is iterated until Y outputs the desired class after processing all instance values.

ANNs have the ability to generalise patterns that they learn from limited, noisy, and incomplete data (S. X. Wu & Banzhaf, 2010). They are therefore suitable for intrusion-detection system decision making that requires adaptation to new traffic models, such as mobile networks (Panchev, Dobrev, & Nicholson, 2014). Their ability to generalise from limited and noisy data has been used to improve intrusion-detection precision (Wang, Hao, Ma, & Huang, 2010). They have been a very effective solution in detecting DoS attacks (Bivens, Palagiri, Smith, Szymanski, & Embrechts, 2002) and illegitimate activities such as portscanning that comprises DoS attacks (Al-Jarrah & Arafat, 2014).

In unsupervised learning, there are no labelled instances to be learned by the neurons. The neural networks discover patterns from input data and adjust the weights of the neurons to classify instances into appropriate categories. The weights are adjusted through the scores obtained from the data. Processing the data is iterated until the network shows statistical regularities of the input data, thereby indicating the data classes. Self Organizing Map is an example of unsupervised neural networks, which is explained below.

Self Organizing Map

Self Organizing Map (SOM) is a class of unsupervised learning Artificial Neural Network techniques (Kohonen, 1982). Similar to ANNs, SOM output neurons adjust their weights to learn from the inputs. The output neurons are arranged in a 2 or 3-dimensional grid to facilitate visualisation. SOMs aim to adjust the weights so that adjacent output neurons represent similar values given varying input values.

To train the neurons, all weights are initialised with random values. Given an input vector, an output neuron that has the highest score is selected to become the winning node. Common score functions for this purpose include Euclidean distance and Gaussian function (Negnevitsky, 2005). The weights of other neurons within a certain radius λ from the winning node are adjusted to minimise the difference between the output values to the winning node value. This step is repeated for all input vectors within the dataset. Training is then repeated with smaller radius λ for a given number of iterations.

SOM has been used in intrusion-detection systems to reduce the map size it took to process the data (Siripanadorn, Hattagam, & Teaumroong, 2010), making it suitable for real time detection (Ramos & Abraham, 2005; Srinivasan, Vijaykumar, & Chandrasekar, 2006). Since SOM is often represented as a two-dimensional map, it has also been used to visualise network anomalies (Corchado & Herrero, 2011; Girardin, 1999; Olszewski, 2014).

Evaluation of machine learning techniques

A notable survey was done reviewing 18 significant works in traffic classification that used machine learning for traffic analysis in (Nguyen & Armitage, 2008). The survey showed that machine learning techniques have demonstrated up to 99% accuracy in classifying a large diversity of Internet traffic.

In order to assess which technique was the best for classifying what type of traffic, a study (Kim et al., 2008) analysed 7 supervised machine learning techniques: NB, Naïve Bayes Kernel Estimation, Bayesian Network, DT, k-NN, ANN, and SVM. The study assessed which technique was the best for classifying 12 types of traffic: 10 different types of applications, attack, or "none". The result consistently showed that SVM yielded the highest accuracy (more than 98%), followed by ANN and k-NN. This finding justified another study (Tsai et al., 2009) that reviewed 55 articles on intrusion detection through application of machine learning. The study concluded that SVM and k-NN were the most commonly used techniques.

This subsection discusses machine learning techniques and how they are applied in detecting traffic anomaly and DoS attacks. The techniques can learn from the data samples, adapt to new environments, produce rule sets, and predict the class of unseen data. They have been suitable for detecting anomalies by distinguishing a group of data that have significantly different properties to other groups. Applications of these techniques are to classify heterogeneous traffic, detect anomalies in large network, detect real-time anomalies, and adaptively learn from new datasets. The next subsection discusses the challenges that these solutions pose.

2.3.2.4 Challenges to Detecting Application-Based DoS Attacks

This subsection discusses challenges to detecting flood-based application-layer DoS attacks and shows what features the different approaches used. It begins by discussing two challenges: the increasing number of Internet-connected devices and the ability of adversaries to mimic normal behaviour. Furthermore, it presents examples of how research in the area is active due to the challenges, discusses solutions that the state-of-the-art studies propose and the features that have been used in the reported studies.

The increasing number of Internet-connected devices

The size of the Internet has doubled every 5 years since 2003 as a result of the growing number of Internet-connected devices and the number of deployed applications (G. Q. Zhang, Zhang, Yang, Cheng, & Zhou, 2008). As was discussed in Chapter 1, this number is projected to continue to grow exponentially. This implies more devices can be made to participate in launching DoS attacks, and researchers are challenged with analysis of bigger datasets.

The growing number of servers to support the growing Internet traffic has contributed to the innovative ways in which a DoS attack can be launched. For example, Content Delivery Networks (CDN) – a network of connected servers that are geographically located closer to the users for the purpose of providing better user experience – could be abused to become DoS amplifiers in order to launch DDoS attacks (Triukose, Al-Qudah, & Rabinovich, 2009). CDN servers could be instructed to repetitively download HTTP contents from a victim web server, which would eventually consume the network bandwidth of the web server, affecting its download speed. An attacker could compromise CDN servers and modify the cache of the CDN servers to send requests to a victim web server. At this point the attacker could terminate its connection to the CDN. In this case, the CDN servers acted as DoS amplifiers since the attacker is not involved in sending requests to the victim. When large objects (e.g. files, pictures) are requested by the CDN servers, the victim may consume excessive network bandwidth.

A later study showed that web proxies could also be utilised to launch similar kinds of attacks (Xie, Tang, Xiang, & Hu, 2013). Web proxies are machines that serve incoming/outgoing traffic on behalf of a web server in order to load balance the volume of traffic that the main server must handle. The authors used historical behaviour to observe traffic in order to identify legitimate traffic and differentiate it from attack traffic. The study identified legitimate traffic when the observed behaviour was not significantly different from historical behaviour.

This implied that the volume and dimension of the data required for traffic analysis increased. Recent datasets containing traffic generated by new devices has rendered legacy studies in detecting application types and network anomalies less accurate. Machine learning techniques have been suitable to address this challenge since they are able to group large data according to the similarities of the data attributes, reducing the high-dimensionality of data. Therefore recent studies, that have identified anomalies in network traffic, have applied machine learning techniques to classify large sized networks (Al-Jarrah et al., 2014; Y.-X. Meng, 2011). Addressing the above-mentioned problems itself has created challenges to application of machine learning techniques. The features, i.e. the attributes that play a role to determine key affecting factors, no longer accurately represent the characteristics of the instances when they are applied to classify recent data. Hence, recent studies have defined ways to effectively select the features in order to reduce the dimensionality of the data and yield high classification accuracy (Al-Jarrah et al., 2014; Baig, Sait, & Shaheen, 2013; Katkar & Kulkarni, 2013).

Larger volumes of data create challenges for research in machine learning, and consequently research in the area remains active. Early solutions became less accurate; hence, state-of-the-art studies were to increase the detection accuracies of the existing techniques (Moore & Zuev, 2005; Stroeh, Madeira, & Goldenstein, 2013). Another implication of having large data is imbalanced traffic distribution in many networks, i.e. the distribution of the traffic data generated by popular applications was found to be skewed when generated by a large number of traffic flows such as HTTP and DoS traffic (Goseva-Popstojanova, Anastasovski, Dimitrijevikj, Pantev, & Miller, 2014; Li et al., 2007). The imbalance problem was caused by machine learning classifiers that favoured large traffic classes, while poor performance was observed in classifying smaller classes. Hence, an Optimised Distance-based Nearest Neighbour technique was proposed to address the imbalance problem (D. Wu et al., 2014). The results show that the proposed technique improved the classification for small classes when tested on real traffic datasets, in terms of higher F-measure by 10% to 20%.

It has been shown that machine learning techniques have been used to address the challenge of analysing large data and heterogeneous networks in the past. They are able to group large data according to similarities of the data attributes, reducing the high-dimensionality of data. The next section shows how they were applied in learning from the data to detect novel adversarial attacks.

The ability of adversaries to mimic normal behaviour

Identifying illegitimate HTTP packets that cause denial of service have been challenged by flash crowd, i.e., legitimate, but sudden and high volume web browsing activities. This could happen when, for instance breaking news emerges, or when a popular sports match reaches a final score, causing the web servers to receive an unusually high volume of visits. Hence, the evolution of detecting illegitimate HTTP packets has been challenged with advanced adversarial behaviour that mimics flash crowds (S. Yu, Guo, & Stojmenovic, 2012; Fabian & Terzis, 2007).

Traditionally, flash crowds have been seen to originate from a large number of clients generating a sudden flash of traffic, while DoS traffic usually originates from a group of unauthenticated, and in all likelihood, illegitimate clients (Jung et al., 2002). It is observed that DoS traffic originates from a small group of clients, whilst flash crowd traffic originates from a more dispersed group of attacking nodes. Furthermore, DoS traffic originates from a group of clients that the server had not seen before, while flash crowd traffic originates from a number of clients that the server had seen before the sudden flash of traffic. Using this approach, the authors showed that the web server was able to drop DoS attack traffic, and could efficiently handle legitimate, flash crowd requests. This approach grouped the client topology; hence, access to the source addresses of all clients was required for the analysis.

Recently, new technologies have been introduced on both the client and server ends. Hence, novel studies are motivated by the evolution of the adversary to generate stealthier traffic through the use of new and enhanced techniques. An example is the use of HTTP GET requests by the adversary to generate traffic that appear to be legitimate. In response to this trend, a study observed that entropy of requests per source IP address of flash crowds is higher than those of DoS attacks (Ni, Gu, Wang, & Li, 2013). A similar study was motivated by analysing DoS traffic in the cloud, generated and guised as HTTP GET Requests, since through such action, all cloud computing resources (platform, software, and infrastructure) could be attacked simultaneously (Choi, Choi, Ko, & Kim, 2014). The study monitored the packet flow and HTTP GET requests arriving at the server end each second, and showed that their solution detected more attacks than a signature-based solution.

Another recent study was motivated through distinguishing DoS attacks from flash crowds at the backbone of a network (Zhou, Jia, Wen, Xiang, & Zhou, 2014). The proposed scheme relied on the frequency metric of web page access and was found to operate at linear complexity. In contrast to its previous study (Ni et al., 2013), their technique showed that flash crowd traffic has the least entropy and was therefore deterministic in nature, thus could be detected with convenience.

Bots, or software that automate repetitious tasks, are common tools to launch DoS attacks. The increased use of bots by the adversary class provides a new playground for launching DoS attacks. The traffic produced by such bots is similar to flash crowd traffic. Therefore, differentiating between bot and legitimate flash crowd traffic has been proposed as a solution in the past. In a study (S. Yu, Zhou, et al., 2012), the author observed that the number of concurrent users during a flash event was an order of magnitude more than the number of live bot attacks. Hence, the author defined traffic flow as a group of network packets bearing the same destination address. It was also shown that the standard deviation of the total flow of attack traffic is smaller than that of the total flow of flash crowd traffic. Future work was to investigate how attackers could manipulate the amount of traffic produced by bots to evade the proposed detection method.

One technique to detect the bots behaviour was to take into account HTTP Request packets appertaining to the browsing session of a website and corresponding user activity associated with individual page access (Ye & Zheng, 2011).

Packets that were not caused by a normal browsing behaviour showed an average frequency of HTTP Request packets that was used to browse below a certain threshold. These packets were tagged as being anomalous, possibly with intentions beyond simplistic and legitimate user browsing. Hence, the technique proposed in the paper marked the traffic as being suspect.

The concern over attackers evading known detection methods was the motivation of another work (Rahmani et al., 2012). Previous detection methods were prone to false negatives i.e., the labelling of attack packets as normal (flash crowd). It was observed that some legitimate connections had greater packet volume than the cumulative sum of other connections, making it convenient for attackers to conceal DoS traffic as a flash crowd. Therefore, it was proposed to device a measure based on the number of packets as well as the numbers of connections per browsing session. Large volume of legitimate traffic caused a higher number of connections, whilst DoS attacks caused a disparity between the number of packets and the number of connections. By analysing the coherence between the traffic features, the researchers showed that their approach was able to detect attacks that otherwise went undetected by previously proposed methods.

Machine learning techniques are suitable to address these challenges since they are able to discover relationships that exist in the data and to predict the class of unseen instances. Machine learning techniques were used in recent studies to detect the number of zombie machines that attempt to connect to a target (Agrawal, Gupta, & Jain, 2011), detect their behaviour (Garg, Singh, Sarje, & Peddoju, 2013; Liu et al., 2013), and improve the detection accuracy (Barthakur, Dahal, & Ghose, 2013). A recent study proposed a scalable solution to detect DoS attacks to large networks due to an increased use of bots (Malialis & Kudenko, 2013).

This subsection has shown the challenges in detecting DoS attacks and the state-of-the-art approaches proposed to provide various solutions. The next subsection provides a discussion summarising the solutions discussed and various features used in the approaches.

Features used to detect DoS attacks

The above discussions are summarised in Table 2.6. The table shows the features used by many of the existing studies discussed above. Many of the studies used the KDD-99 dataset (Tavallaee, Bagheri, Lu, & Ghorbani, 2009), which is a publicly available dataset to aid research in intrusion detection. This section lists the features used in the literature as a base to later compare with the HTTP/2 traffic data used in the thesis experiments (Section 2.5).

The table shows that the TCP port number is still considered as a relevant feature in many studies, although the observed traffic is not assumed to use universally registered numbers. For example, in one study (Rahmani et al., 2012), the number of connections was used as a feature that corresponded to "IP port-to-port traffic exchanged between two IP addresses during a period." Many other studies used TCP port information in combination with other information to define a connection or flow. However, TCP port number is less relevant when the study wishes to analyse traffic patterns of only one protocol that uses a predefined port number (e.g. HTTP uses port 80). An example of this was a study that sought patterns of different adversarial attacks to servers running HTTP applications in order to detect DoS attacks (Goseva-Popstojanova et al., 2014). Since the technique monitored only HTTP traffic, TCP port number was not used as a feature. Instead, flow information from the network was analysed to detect attacks.

Table 2.6 also shows that application-layer information (TCP payload) can be representative of a pattern. Earlier studies used TCP payload information to detect traffic signatures (Sen, Spatscheck, & Wang, 2004); later studies identified patterns out of the payload information. For example, when detecting a flood of HTTP packets, many studies identified patterns out of the observed HTTP information. Features identified from TCP payload were the number of HTTP Requests per observation (Ye & Zheng, 2011), the number of HTTP Requests per source IP (Ni et al., 2013), and the number of HTTP Requests per second (Choi et al., 2014). Another example was using other HTTP information such as the name of a file to be retrieved. This data was used to construct a new feature based on the number of files accessed per each 10 second interval (Chan et al., 2006).

Finally, Table 2.6 shows that the statistical properties of the traffic (such as the flow of the packets, the size of the packets, and the duration of the observation) were relevant information in detecting DoS attacks. This information was obtained without inspecting the content of the packet. Studies that aimed to detect DoS attacks used packet content as features.

While this section discussed how DoS attacks can be launched and showed current techniques in DoS attack detection and analysis, the next subsection discusses a new challenge in launching and detecting DoS attacks due to the introduction of the HTTP/2 protocol.

Table 2.6: Features used by the existing studies on DoS detection

Author (Year)	Title	Header Data				Statistical Data			Feature
		IP	TCP port	TCP flag	TCP payload	num packets	size	duration	
Jung, J., Krishnamurthy, B., & Rabinovich, M. (2002)	Flash crowds and denial of service attacks: Characterisation and implications for CDNs and web sites					✓		✓	traffic volume
		✓	✓						num distinct clients per 10s interval
					✓				num files accessed per 10s interval
Feinstein, L., Schnackenberg, D., Balupari, R., & Kindred, D. (2003)	Statistical approaches to DDoS attack detection and response	✓	✓	✓					entropy of a header parameter (e.g., entropy of source, entropy of destination port)
Sen, S., Spatscheck, O., & Wang, D. (2004)	Accurate, scalable in-network identification of P2P traffic using application signatures				✓				signature
Karagiannis, T., Papagiannaki, K., & Faloutsos, M. (2005)	BLINC: multilevel traffic classification in the dark	✓							number of hosts
		✓	✓						distribution of src ports
		✓	✓						number of packets transferred
Erman, J., Mahanti, A., Arlitt, M., & Williamson, C. (2007)	Identifying and discriminating between web and peer-to-peer traffic in the network core					✓			total num packets
					✓		✓		mean payload size (excl header)
							✓		num bytes transferred
						✓		✓	flow duration
								✓	mean inter-arrival time of packets
Auld, Moore, & Gull. (2007)	Bayesian neural networks for internet traffic classification					✓		✓	flow

continued ...

. . . continued

		Header Data				Statistical Data			
Author (Year)	Title	IP port	TCP flag	TCP payload	TCP packets	num	size	duration	Feature
Kim et al. (2008)	Internet traffic classification demystified: myths, caveats, and the best practices	✓	✓	✓		✓		✓	flow
Ye, C., & Zheng, K. (2011)	Detection of application layer distributed denial of service				✓	✓		✓	num HTTP Requests
					✓	✓		✓	frequency vector
Yu et al. (2012)	Discriminating DDoS attacks from flash crowds using flow correlation coefficient	✓				✓		✓	flow correlation coefficient
Rahmani, H., Sahli, N., & Kamoun, F. (2012)	Distributed denial-of-service attack detection scheme-based joint-entropy	✓	✓					✓	number of connections
						✓		✓	number of packets
Dyer, K. P., Coull, S. E., Ristenpart, T., & Shrimpton, T. (2012)	Peek-a-boo, I still see you. Why efficient traffic analysis countermeasures fail						✓		packet length
								✓	total trace time
						✓	✓	✓	flow direction
Malialis, K., & Kudenko, D. (2013)	Large-scale DDoS response using cooperative reinforcement learning					✓		✓	aggregate traffic arrived over the last T seconds
Ni, T., Gu, X., Wang, H., & Li, Y. (2013)	Real-time detection of application-layer DDoS attack using time series analysis	✓			✓				num HTTP Request per source IP

continued . . .

. . . continued

Author (Year)	Title	Header Data				Statistical Data			Feature
		IP port	TCP flag	TCP payload	TCP packets	num	size	duration	
Xie, Y., Tang, S., Xiang, Y., & Hu, J. (2013)	Resisting web proxy-based http attacks by temporal and spatial locality behavior				✓	✓		✓	total num requests
					✓			✓	observed behavior index
					✓			✓	historical behaviour profile
Choi, J., Choi, C., Ko, B., & Kim, P. (2014)	A method of DDoS attack detection using HTTP packet pattern and rule engine in cloud computing environment					✓			entropy num packets
			✓			✓			entropy num src port
		✓				✓			entropy num dest address
					✓	✓		✓	HTTP Request per second
Goseva-Popstojanova, K., Anastasovski, G., Dimitrijevikj, A., Pantev, R., & Miller, B. (2014)	Characterization and classification of malicious Web traffic				✓		✓		mean length of request substrings
					✓		✓		median length of request substrings
					✓		✓		max length of request substrings
					✓	✓			num requests POST
Zhou, W., Jia, W., Wen, S., Xiang, Y., & Zhou, W. (2014)	Detection and defense of application-layer DDoS attacks in backbone web traffic	✓							entropy source IP
					✓				entropy URL
						✓		✓	traffic intensity
Tang, Y., Lin, P., & Luo, Z. (2014)	Obfuscating Encrypted Web Traffic with Combined Objects					✓		✓	num packets
							✓	✓	packet length

2.4 The HTTP/2 Protocol

2.4.1 Initiatives

Hypertext Transfer Protocol (HTTP) has been the protocol of choice for web browsing communication until today. The current version, HTTP/1.1, was designed to transfer texts. As technology evolved, rich media was being increasingly transferred using the same protocol, causing the web response time to slow down. Furthermore, modern web applications that use these rich media have created a demand for more user interactions, causing the protocol to reach its limit. Consequently, web users experience slow connections to websites.

In 2009, Google introduced the SPDY (read "speedy") protocol to respond to the above problem. SPDY retained the semantics of HTTP/1.1, while adding *multiplexing* mechanisms in order to speed up web communication (Thomas, Jurdak, & Atkinson, 2012). The protocol was implemented in the Google Chrome browser, allowing users to experience faster web browsing. Google server-end services such as Search, Gmail, Maps, were able to use SPDY, leading to improved quality of Google services. In 2012, industries followed the trend. SPDY client was implemented not only in web browsers such as Firefox and Opera, but also on tablet devices such as Amazon Kindle. Major websites such as Facebook and Twitter implemented SPDY server in subsequent years.

In 2012, the HTTP Working Group proposed to adopt SPDY as a catalyst to a new HTTP version, HTTP/2 (Grigorik, 2013b). The following subsection details the anatomy of HTTP/2.

2.4.2 Protocol Specifications

HTTP/2 is designed to improve the communication speed between clients and servers. It is aimed to address the slow response rates that its predecessor suffered through introduction of message multiplexing. The following discussion is to explain the mechanism of HTTP/2 message multiplexing. The architecture was adopted from the HTTP/2 standard (Belshe et al., May 2015), authored by the

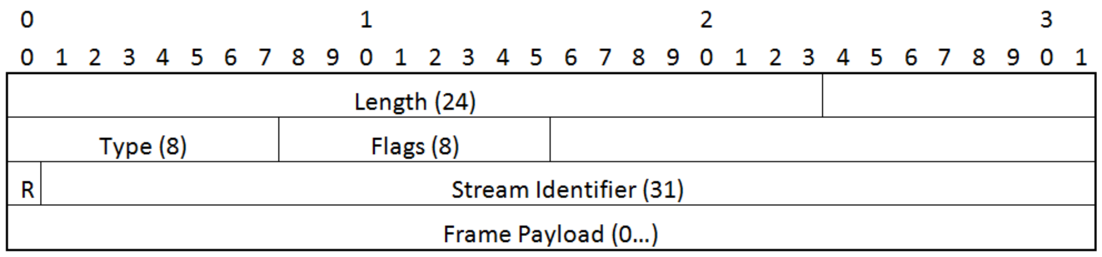


Figure 2-11: Frame format

HTTP Working Group.

The HTTP/2 protocol format is based on binary framing as opposed to the newline-delimited plain text mechanism of its predecessor. Figure 2-11 shows the layout of the frame. The first field of the row identifies the length of the frame payload. Hence, binary framing allows its parser to efficiently identify the location of the subsequent packets in the traffic flow, and quickly identify each packet's type and flags.

The binary framing also allows multiplexing, i.e., multiple requests/responses in one TCP connection per origin. First, HTTP/2 messages are broken down into independent binary frame packets according to their type. Examples of frame types are headers, data, setting and priority frames. As illustrated in Figure 2-11, the `Type` field indicates the frame type of an HTTP/2 packet. Second, each frame is assigned a stream ID through the application of the `Stream Identifier` field in the frame header. Packets with different stream IDs can be sent independently on the communication line in terms of time and direction. This technique allows multiple HTTP requests and responses to be sent within one TCP connection.

An example of the technique is illustrated as follows. Suppose that a client requested a page to a web server, and received a response page as shown in Figure 2-12. In HTTP/1.1, the client must send more that one HTTP Request message to obtain the auxiliary files mentioned in the HTML code, i.e. the example.js, Puzzle.jpg, and theme.css files. This is illustrated on the left side of Figure 2-13. The figure shows that the client sent four request messages to get the four files as coded in the HTML file.

On the other hand, HTTP/2 only requires one HTTP Request message, as

```
<html>
<body>
<p id="demo">A Paragraph.</p>
<button type="button" onclick="myFunction()">Try it</button>
<script src="example.js"></script>
<img src="Puzzle.jpg">
<link rel="stylesheet" type="text/css" href="theme.css">
</body>
</html>
```

Figure 2-12: An html response

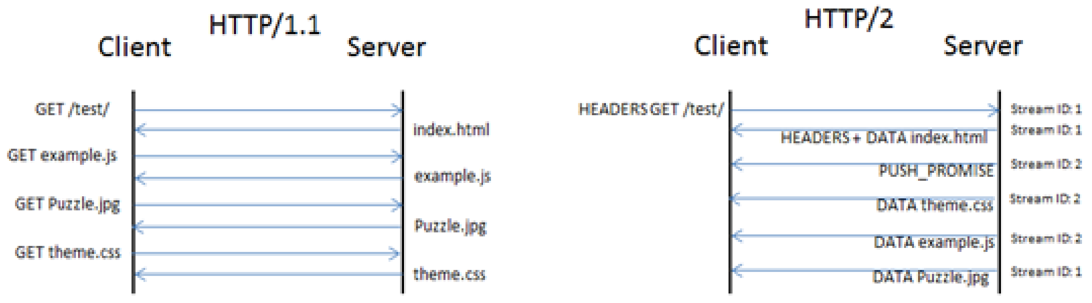


Figure 2-13: Illustration on HTTP/1.1 requests/responses and HTTP/2 multiplexing

shown on the right side of Figure 2-13. The protocol allows the server to send (push) the auxiliary files. All assets are sent to the HTTP/2 client without having to receive other client requests. In addition, the illustration shows that HTTP/2 does not necessarily have to send the files in the order of how they were coded in the HTML file (Figure 2-13). This is due to the capability of HTTP/2 to prioritise a flow of frames. As illustrated in the figure, the file theme.css is sent the last when using HTTP/1.1, but it is sent the first among the other auxiliary files when using HTTP/2.

The example shows that HTTP/2 is able to interactively send multiple requests from client browsers on one direction, and delivers assets (i.e. response pages, images, files) from the web server back to the client in different orders. In addition, the server is able to avoid waiting for subsequent client requests by pushing assets to the client. This is because the server can already identify where to find those assets. The example shows that multiplexing, i.e. interleaving and pushing frames, is one of the novel techniques that HTTP/2 introduces.

Figure 2-14 shows the same example from a different viewpoint. It illustrates

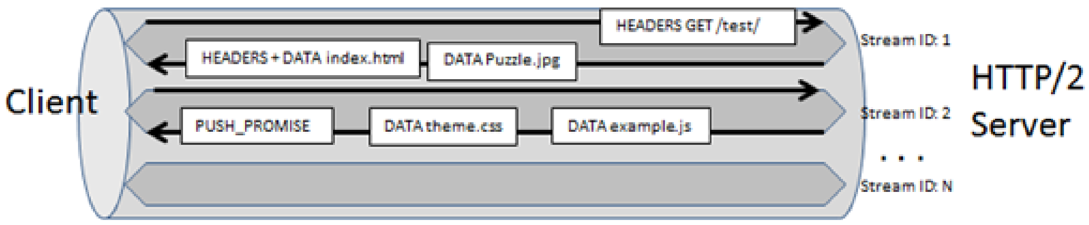


Figure 2-14: Illustration on HTTP/2 multiplexing from a different perspective

how streams (a flow of frames) are multiplexed in one connection between a client and an HTTP/2 server.

In addition to the above mechanism, what makes HTTP/2 traffic pattern different than HTTP/1.1 is that its implementation demands using Transport Layer Security (TLS) to encrypt the messages. Although HTTP/2 did not mandate the use of encryption, its implementations only support encrypted HTTP/2 services. Currently, no browser supports HTTP/2 traffic in unencrypted form. Rather than for privacy purposes, the main motivation for encryption is to ensure that the protocol can communicate without modification with intermediaries such as routers or intrusion-detection systems (Grigorik, 2013a). Such intermediary interventions could alter the client-server settings that may cause an increased communication delay, thus affecting the end-user experience (Quality of Service).

It can be seen that HTTP/2 has different message exchange mechanism to its previous version. The ability of HTTP/2 to efficiently exchange messages and transparently communicate with its predecessors implies that the implementation introduces state-of-the-art set of rules on how it manages computing/communication resources. At the same token, the new protocol suggests an array of security considerations.

2.4.3 Security Considerations

The HTTP Working Group indicated a number of potential security issues in the HTTP/2 standard (Belshe et al., May 2015). These are discussed as follows.

Server Authority. When a client fails to validate if the server was authoritative to push assets (e.g. files) to the client, then the client could receive illegitimate information. As illustrated in the right diagram in Figure 2-13, HTTP/2 servers push assets to clients, without requiring the clients to send additional messages to request for the assets. These assets can be located at different servers including those controlled by an attacker. Attacker-controlled assets can be malicious, e.g. carry virus or contain inappropriate texts. Clients that did not certify servers can thus become vulnerable to attackers.

Cross-Protocol Attack. In cross-protocol attack, illegitimate transactions in one protocol can appear as valid transactions in another protocol. An attacker can use a client to send legitimate HTTP/1.1 messages to attack an HTTP/2 server.

To illustrate, consider the Upgrade field that both HTTP/1.1 and HTTP/2 can have in their headers. In HTTP/1.1, the Upgrade field is to carry instructions that specify what additional communication protocols the client supports and would like to use. For example, a client can propose to switch to Internet Relay Chat (IRC) protocol through sending "`Upgrade: IRC/6.9`" message in the HTTP header to a server. If the server supports the protocol, it can begin communicating IRC messages with the client.

In HTTP/2, the Upgrade field provides backward compatibility, allowing clients to communicate with HTTP/1.1 if a server does not support HTTP/2. Suppose a web browser was equipped with both HTTP/2 protocol and its predecessors. Upon an initial connection to a server, the browser wished to probe if the server was HTTP/2-enabled or not. Hence, it firstly sent an HTTP/1.1 request containing an "`Upgrade: h2c`" message in the header to signal that it wished to communicate using HTTP/2. Servers that support HTTP/2 can begin communicating the client under the HTTP/2 protocol. If an HTTP/2 server implementation misinterpreted the Upgrade field message from the browser, then a legitimate message sent by a browser that uses the additional (upgrade) protocol can become a malicious instruction destined to the HTTP/2 server.

Intermediary Encapsulation Attack. HTTP/2 header fields allow values that are not valid HTTP/1.1 values. Examples of these values are carriage return, line feed, and zero characters. An adversary could take advantage of this fact to create malicious messages if an HTTP/2 parser did not correctly handle these values.

Cacheability of Pushed Responses. When an HTTP/2-enabled server allows access from multiple users, one user could cause cached assets to be sent to another user. This means attackers could reveal or forward classified information, which implies a security breach.

Use of Compression. Secret data could be recovered when compressed in the same context as data under attacker's control. For example, an attacker could generate and send messages to an HTTP/2 server. The protocol compresses the messages before they are sent through a communication channel. The attacker could observe the length and the ciphertext after the compression, and collect patterns between the cleartext and the ciphertext. The attacker could eventually infer the content of communicated secret messages.

Privacy Considerations. Settings, priorities, and flow control values could be used for fingerprinting (e.g. revealing the type of browser, machine, or activity of the remote device). Examining the values of protocol parameters has been used as a technique to fingerprint a target. The introduction frame types and flow control values in HTTP/2 can open a new landscape of how a target running HTTP/2 protocol can be fingerprinted.

Denial of Service. HTTP/2-enabled servers demand more computing resources than HTTP/1.1 machines. HTTP/2 introduces techniques previously not employed in its predecessors. These techniques can demand more CPU utilisation and memory consumption. This will be further discussed in the next subsection.

The primary elements of security are confidentiality, integrity, and availability (Pressman & Jawadekar, 1987; Tanenbaum & Van Steen, 2007). *Confidentiality* emphasises the ability of a system to guarantee that the information is only disclosed to the authorised parties. *Integrity* concerns with the accuracy and consistency of an asset against alterations. *Availability* means that information is accessible when requested by an authorised party. Out of the HTTP/2 security considerations, the Denial of Service (DoS) issue falls in this category. Detecting DoS requires pattern matching, classification, and prediction in order to separate

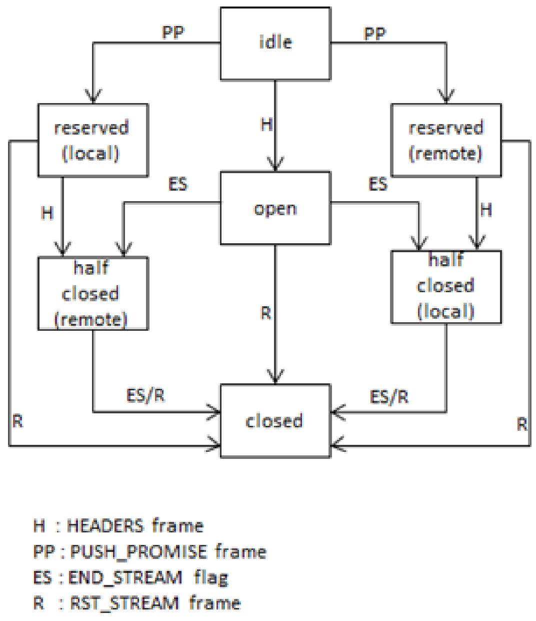


Figure 2-15: Stream states

normal from attack information. The following subsection discusses how HTTP/2 information exchange mechanism can cause a server running HTTP/2 to become unavailable.

2.4.4 Exposure to Resource Depletion

Previously it has been explained that HTTP/2 introduced multiplexing, a mechanism to interleave and prioritise web messages that was not found in HTTP/1.1. This subsection reviews how HTTP/2 multiplexing can cause resource depletion.

The review goes back to HTTP messages that are broken down to frames, as shown in Figure 2-11. Each frame in any flow direction can be grouped based on its stream ID. In order to govern the types of frames that are legal or illegal to be sent (or received) after one or a sequence of frames were sent/received, each stream ID demands maintenance of its own *state*. Figure 2-15 depicts each possible state of a given stream. This means HTTP/2-enabled equipment needs to maintain the state of each stream in its memory (e.g. stream 3 is "idle", stream 2 is "open", stream 1 is "close", etc.)

The state arrangement allows an HTTP/2 connection to have virtual concurrent streams (as shown in Figure 2-14) for the purpose of avoiding the delay of delivering higher priority frames. When too many frames have similar priority, the network can become congested. Hence, the protocol imposes a *flow control* mechanism in order for the communicating devices to advertise their susceptibility to congestion.

The `window_update` frame is an HTTP/2 message that implements flow control. The purpose of flow control is to moderate the many streams in one connection. This allows a server to lessen the packet flow on one stream, while it needs to continue processing other streams in the same connection. The window_update frame is a way that a client or a server communicates the size of data that the sender can transmit in addition to the current size. This capability did not exist in HTTP/1.1. Changing the value of window_update frame causes an HTTP/2 device to carry out processing tasks that were not seen in any HTTP/1.1 implementation. Changing this value indefinitely can consume more CPU utilisation. This study observed how changing window_update values affect the target's computing resources (Chapter 5).

The HTTP/2-standard specifies the general principle of the flow control mechanism, but states that its implementations are to select any suitable technique. Flow control at the application layer is novel; HTTP/1.1 as such did not define any flow control. Flow control implementation on an HTTP/2 server can demand great computing resources (CPU time and memory consumption) in order to monitor the condition of each connection and maintain the state of each frame. A server running HTTP/2 can consume greater CPU time and memory.

It has been shown that HTTP/2 mechanism is novel as it was not seen in the architecture of its predecessor. This implies that an HTTP/1.1-enabled web server should closely monitor its resource utilisation when the same machine was upgraded to enable HTTP/2. The HTTP/2 standard states that, "An endpoint that does not monitor this behaviour exposes itself to a risk of denial of service attack." Therefore, there is a gap in the literature to observe the network and endpoint machine parameters when running HTTP/2 services, to study how HTTP/2 services can consume computing/network resources of a web server.

2.4.5 Implication to DoS Detection Technique

This study investigated how DoS could be crafted to attack HTTP/2 services based on two gaps not explored in the literature: the introduction of multiplexing and the use of encryption.

First, HTTP/2 introduces the flow control mechanism which was not present in its predecessor, which means it adds dimensionality to traffic data. HTTP/2 traffic pattern is different from its predecessors since interleaved and multiplexed traffic that the flow-control packets manage depicts new traffic patterns. Large amount of HTTP/2 traffic requires a new technique to analyse the characteristics of the traffic with new patterns.

Second, HTTP/2 implementations are encrypted. Although secured HTTP is not novel, its implications to DoS attack detection had not been explored in the literature. Securing HTTP/1.1 with TLS or its predecessor, Secure Socket Layer (SSL), has been commonly used in the past. DoS detection methods discussed in the literature (Section 2.3) analysed unencrypted HTTP/1.1 traffic. As was shown, an avenue of the studies proposed payload inspections which required access to the unencrypted message content. In contrast, this study observes, characterises, and classifies both normal and attack TLS-encrypted HTTP/2 traffic.

While some machine learning techniques are suitable to analyse data with high dimensionality, different techniques are required to select features and analyse the performance, since one technique alone might not give the optimum result. This study uses different machine learning techniques to show comparisons of the new traffic patterns as presented by HTTP/2 traffic. The methods of generating the traffic, extracting and selecting features are discussed in the next Chapter.

2.5 Conclusion

In this chapter, various challenges and approaches for detecting DoS attacks were reviewed. Many research approaches defined features to address their specific objective. As Table 2.6 summarizes, the data collection for these features was accomplished from inspecting packet headers (e.g. the number of hosts was derived from IP headers; the number of connections from TCP headers), or the statistical properties of these packets (e.g. packet length was derived from the packet sizes).

The current challenges in detecting DoS attacks are due to the increasing size of the Internet, and the ability of adversaries to launch undetected attacks. Machine learning techniques are suitable to address the problem in the area, since they are able to learn from new environments, discover relationships that exist in data samples, adaptively create a classification rule, and provide scalable solutions. They have been used to analyse large data and heterogeneous network traffic, and detect the presence and the behaviour of zombie machines. The techniques have shown to provide high accuracy of classifying traffic and predicting the class of new instances.

A novel challenge in detecting DoS attack is due to the introduction of HTTP/2. Web servers that implement the HTTP/2 protocol can demand more computing resources than servers that implement previous HTTP versions. HTTP/2 server implementation introduces mechanisms that were not previously present such as multiplexing and flow control. This means that research in the area is challenged through the introduction of a new range of data types and features, such as window_update and TLS frames. The summary is shown in Table 2.7. The experiments in this thesis used both the existing and the new range of data to define features in order to detect DoS attacks against HTTP/2 servers.

Since this study used new range of data to recognise patterns that were not previously studied, it applied machine learning techniques to detect DoS attacks against HTTP/2 services. The next chapter describes the proposed investigation for generating the traffic data, extracting and selecting features for its analysis using machine learning in detecting DoS attacks against HTTP/2 servers.

Table 2.7: The range of data to observe for the investigations in this thesis

Observed Data	Existing Studies (HTTP/1.1)	This thesis (HTTP/2)
Statistical Data:		
flow, size, duration	✓	✓
Network-layer data:		
IP (source, destination)	✓	✓
TCP ports	✓	✓
TCP flags	✓	✓
Application-layer data:		
HTTP Request	✓	✓
HTTP/2 Frames	–	✓
TLS messages	–	✓

Chapter 3

Research Approach and Methodology

The previous chapter reviewed literature on DoS attacks and HTTP/2. Existing solutions to detect DoS attacks were presented. The topics addressed included what features of traffic were used, and how they were evaluated by the researchers. However, these existing solutions were developed for HTTP/1.1 traffic, while none explored how to detect DoS attacks against HTTP/2 servers.

In this chapter, an approach is shown to attest the distinction between HTTP/2 DoS traffic and normal traffic through observing traffic patterns. The research approach is discussed in Section 3.1. The study explored how an HTTP/2 DoS attack can be modelled and how traffic can be generated from the model. The study observed the generated traffic patterns, extracted features and created datasets to analyse the patterns. Machine learning techniques were applied to demonstrate how DoS attacks against HTTP/2 servers can be distinguished from normal traffic.

While an approach to observe traffic and generate data is shown, this chapter also discusses how the data are evaluated. Section 3.2 discusses the evaluation metrics used in the study, to provide means to analyse the generated data. Evaluation metrics allow researchers to assess incorrectly classified instances, Detection Rate, and False Alarm Rates from different results. To model, generate, and observe traffic, a lab which comprised of HTTP/2 communicating machines was designed. The lab setup is detailed in Section 3.3.

3.1 Research Approach

The proposed approach for detecting DoS attacks includes four phases which is described as follows.

Phase 1 involves traffic modelling. Traffic models allow researchers to understand how traffic can be generated. In this phase, traffic was modelled through how current equipment, tools, and methodologies can be employed to generate traffic. State-of-the-art equipment was employed, whose specifications and infrastructure are detailed in Section 3.3.

Two traffic models were developed in this study, i.e. attack and normal models. Attack traffic was modelled through how client-server machines communicates with HTTP/2 protocol, how these machines generated traffic, and how the sequence of traffic generation can be exploited to generate DoS traffic. Normal traffic was modelled through mimicking the sequence and rhythms on how human browse the Internet. Phase 1 is further detailed in Section 3.1.1.

Phase 2 provides a mechanism to generate traffic from the attack model of the previous phase. The models were simulated at the client side, and HTTP/2 traffic was generated by according to the pattern that the models describe. The generated traffic was sent by the client towards the server as illustrated in Figure 3-5. How traffic was generated and captured is discussed in Section 3.1.2. Traffic generated through the proposed model serves as input to phase 3.

In **phase 3**, network traffic features are extracted and datasets are generated to represent legitimate and malicious traffic. The study observed patterns from the generated traffic, such as the packet types and their statistical properties such as the count, size, and lapse time of each packet. These properties were used to create traffic features to characterise the different traffic patterns generated in this study. Furthermore, traffic features were used to create datasets. These datasets served as inputs to the machine learning techniques that show how attack and normal traffic were classified. Feature extraction and dataset creation procedures are discussed in detail in Section 3.1.3.

In **phase 4**, features are ranked and machine learning classifier performance is analysed. Feature ranking reduces the number of inputs for machine learning processing and analysis, through finding and ranking the most relevant features. Ranked features aid the analysis of this study to find a set of features that can describe the characteristics of the traffic models. Traffic analysers such as intrusion-detection systems depend on having a finite set of rules to efficiently detect certain traffic types. Feature ranking techniques are discussed in Section 3.1.4.

In this phase, different sets of ranked features were investigated to observe performance of classifying attack and normal traffic. Traffic classification was studied through employing four machine learning techniques, i.e. Naïve Bayes, Decision Tree, JRip, and Support Vector Machines. To analyse the performance of these machine learning techniques, several evaluation metrics were adopted. The evaluation metrics is discussed in Section 3.2.

The details of the methods in each phase are discussed in the following subsections.

3.1.1 Phase 1: Model Development

The study developed two models, i.e. normal and attack traffic model. The normal traffic model was used as the standard traffic to facilitate comparisons and analysis against attack traffic. To model normal traffic, the study used a publicly-available log file that recorded actual human activities depicting routine behaviour, such as browsing a website from one page to another, logging-in, and posting a message. The log file is described in more detail, in Section 4.1. Using the information from this log, a state transition model was used to model real human actions (Section 4.2). This model was implemented to generate flash-crowd traffic in the second phase of the study, where a number of human profiles (extracted from the log) was run using a number of virtual machines connected to an HTTP/2 server.

The attack model was developed based on several methodologies found in the literature (Igure & Williams, 2008; Loukas et al., 2013; Mirkovic & Reiher, 2004). The study observed the type of HTTP frames, the number of attacking clients, the amount of traffic generated, and the consumed resources of the target such as the CPU, memory, and network throughput. The amount of hardware required and the degree of automation to launch the attack were observed, as they alluded towards the design for a successful attack.

An attack was considered successful when the target server had consumed its entire computing resource or network bandwidth. That is, the server showed:

- a significant increase in CPU utilization, or
- a significant increase in memory utilization, or
- a significant decrease in network throughput, or
- a significant increase in packet loss.

The study explored the effect of sending various combinations of HTTP/2 frame types, or launching a large number of HTTP/2 packets, and observed whether the server showed signs of computing resource depletion. This study also described four attack models as a result of controlling different parameters of the packet generator, i.e. the number of packets, number of stream ID (Section 5.1), payload value of the HTTP/2 packets (Section 5.2), delay between packets, and delay between TCP connections (Section 6.1 to 6.2).

The normal model is discussed in Section 4.2. The attack models are discussed in Section 5.1.1, 5.2.1, 6.1.1, and 6.2.1. These models were used to generate HTTP/2 traffic.

3.1.2 Phase 2: Traffic Generation

This phase aimed to produce flash-crowd and attack traffic. Flash-crowd traffic is defined as normal traffic that consumes the computing resources of a web server.

Normal traffic was generated in this study through simulating the legitimate activities from the state-transition model described in the previous phase. When the simulation was in a specific state of the model, an HTTP/2 Request message was generated from a client to target the server. The packet generator that was discussed in Section 3.3 was implemented to send client request packets for this purpose to simulate normal-user browsing activities. The traffic it generated was captured using `TShark` (Combs, 1998–2015) at the server side. A script was developed to replicate the model and simulate different user profiles (page 103 defines 21 user profiles). This eventually created flash-crowd traffic (Section 4.4.2). Hence, the flash crowd traffic was generated out of a large volume of legitimate traffic, generated by many normal (non malicious) human users connecting to a server during a given time frame.

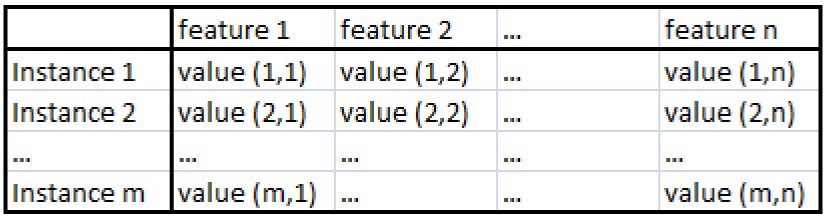
This phase also generated attack traffic, which is a flood of network packets that consume the computing resources of the server such as CPU and memory. To generate a flood of HTTP/2 packets, the packet generator was designed to constantly send HTTP/2 `window_update` packets into the outgoing network stream. From the attack models developed in the previous phase, this study generated four sets of attack traffic: HTTP/2 flood (Section 5.1), DDoS attack (Section 5.2), and two DDoS attack types that mimicked flash-crowd traffic (Section 6.1 and 6.2).

These sets of normal and attack traffic were then used to create datasets which are detailed in the subsection.

3.1.3 Phase 3: Feature Extraction and Dataset Creation

The traffic generated from the previous phase was observed by replaying TShark. The traffic was characterised by enumerating samples of the traffic. A sample is commonly termed an "example" or an "instance," used interchangeably (Witten & Frank, 2005, p.45). This study defined an instance as a 1 second time window of traffic; hence, a 10-second traffic window yielded 10 instances.

Instances are expressed through a set of name-value pairs. The names of



	feature 1	feature 2	...	feature n
Instance 1	value (1,1)	value (1,2)	...	value (1,n)
Instance 2	value (2,1)	value (2,2)	...	value (2,n)
...	...	...	...	...
Instance m	value (m,1)	...	...	value (m,n)

Figure 3-1: A dataset consists of a set of instances, which are characterised through a set of features.

the values are commonly termed as "features" or "attributes" (Witten & Frank, 2005, p.49). Stated in other words, instances consist of a set of features. Figure 3-1 illustrates m instances, where each instance is expressed through a set of n features.

A set of instances that characterise the whole traffic became a dataset. Hence, datasets can be expressed as a matrix of instances versus features as illustrated in Figure 3-1. In this study, characterizing traffic began by extracting features which is explained below.

Extract features

In order to extract features, network packets were identified according to how they differ from other packet types. Network packets serve different functions, e.g. to carry IP addresses, initiate an end-to-end connection, initiate a secure connection (i.e. through a TLS handshake), or to carry application data. Different network packets serve different purposes. This study identified features according to several defined network *packet types*.

An example of feature identification is illustrated as follows. Suppose a traffic instance was observed as shown in Figure 3-2 (adapted from (Grigorik, 2013a)). The client in the figure began initiating a connection through sending a TCP 3-way handshake, which consists of 3 message exchanges: a SYN packet is sent from an initiator (client) to its pair on the other end (server); the server replies to this with a SYN-ACK packet; and the client acknowledges this by returning an ACK packet to the server. In this case, the *packet types* are the SYN, ACK, and SYN-ACK packets, respectively.

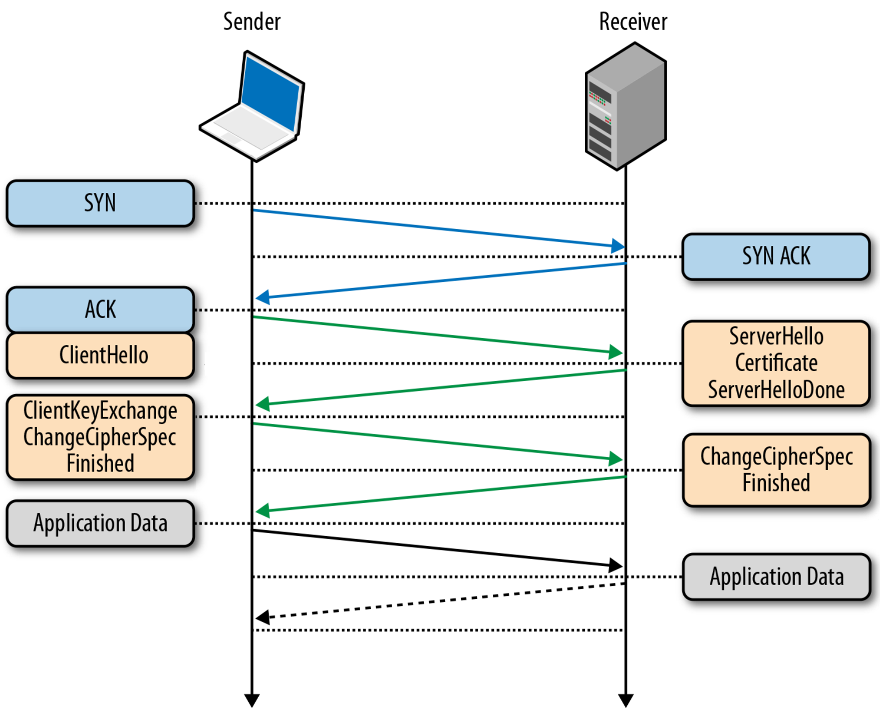


Figure 3-2: TCP and TLS Handshake

Features were extracted from the packets that the client sent to the server, because the client-to-server communication direction is the one most relevant for flooding a server. Hence, as in the above example, the packets used for analysing features in this study are pictured on the left-hand side of Figure 3-2.

The values of each packet type are characterised by three groups of features: count, size, and lapse. For each 1-second traffic instance, the values of these features were obtained as follows.

- The *count* feature is the number of packets captured, grouped by packet type.
- The *size* feature is the total number of bytes of a packet captured, grouped by packet type.
- The *lapse* feature is the time lapse between packet capture and connection initiation (i.e. the length of time between a packet and the SYN packet of a connection), grouped by packet type. For each packet type:
  - If there is more than one packet within a connection, only the lapse

value of the first packet is considered.

- Because there can be more than one connection within an instance, there are as many lapse values as the number of connections. The lapse feature considers the *minimum*, *average*, as well as *maximum* values.

A 1-second time slice was chosen instead of other interval values (e.g. 2 seconds, 5 seconds), to ease the data validation process during the simulations. For example, a 3,600-second captured traffic should produce a 3,600-row dataset.

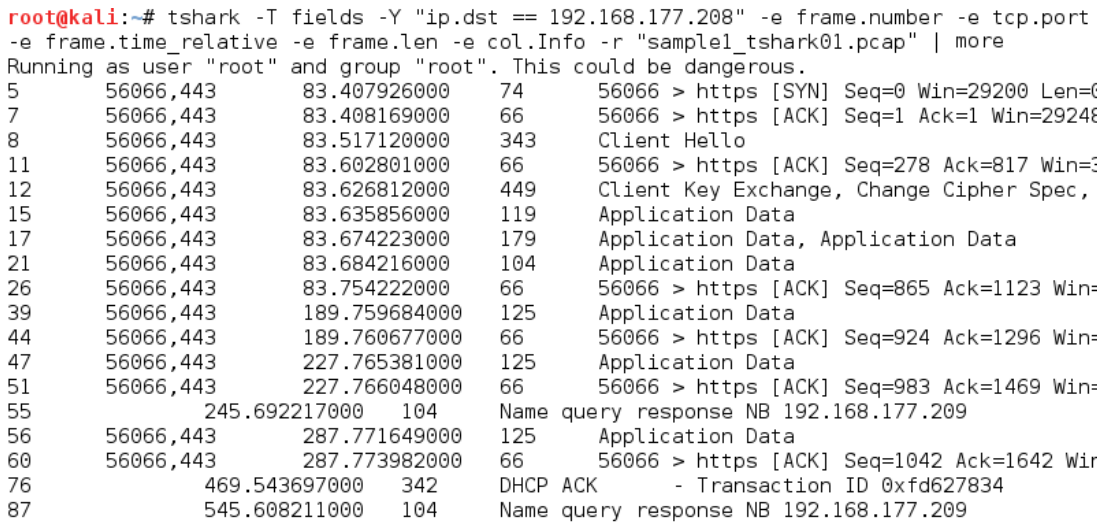
Two examples on extracting feature values are hereby presented. The first example is taken from a traffic instance shown in Figure 3-3 with the corresponding extracted feature values shown in Table 3.1. The second example is taken from a traffic instance shown in Figure 3-4 with the extracted feature values shown in Table 3.2.

The first example considers the entire packets listed in Figure 3-3 as observed within a single time window[1]. The first column of the list is packet number, the second column shows both source and destination TCP port numbers, the third column shows the time lapse relative to the time when the first packet was observed, the fourth column shows the size of the packets in bytes, and the fifth column represents the packet type. With this information, features of network traffic packets can be extracted from real traffic.

In Figure 3-3, the SYN packet marked the beginning of a connection. There was only 1 SYN packet within the observed time window. Hence, the *SYN count feature* had a value of 1; and the *SYN size feature* was equal to 74 (as shown in the 4th column of the 1st packet in the figure). These feature values are shown in the first row of Table 3.1. There was no *lapse feature* for SYN packets, because the time lapse between a SYN packet to the beginning of its connection is always zero. This is represented as "n.a." in the table.

Extracting the *count* and *size feature* values followed a similar procedure for

[1]The length of the observed time window to create datasets used for this study was 1 sec. However, the examples given in this subsection used varying observed time window values to simplify the discussions.

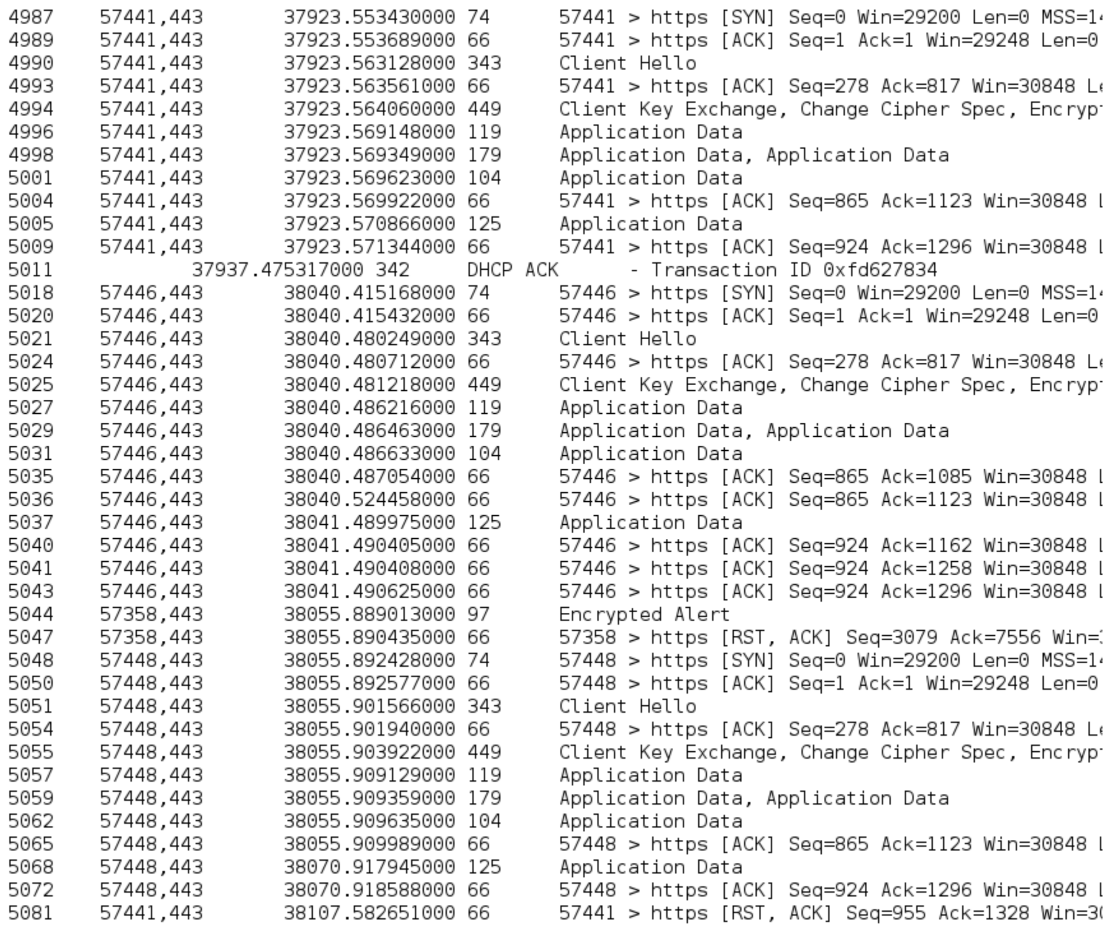


```
root@kali:~# tshark -T fields -Y "ip.dst == 192.168.177.208" -e frame.number -e tcp.port
-e frame.time_relative -e frame.len -e col.Info -r "sample1_tshark01.pcap" | more
Running as user "root" and group "root". This could be dangerous.
5       56066,443       83.407926000    74      56066 > https [SYN] Seq=0 Win=29200 Len=
7       56066,443       83.408169000    66      56066 > https [ACK] Seq=1 Ack=1 Win=2924
8       56066,443       83.517120000    343     Client Hello
11      56066,443       83.602801000    66      56066 > https [ACK] Seq=278 Ack=817 Win=
12      56066,443       83.626812000    449     Client Key Exchange, Change Cipher Spec,
15      56066,443       83.635856000    119     Application Data
17      56066,443       83.674223000    179     Application Data, Application Data
21      56066,443       83.684216000    104     Application Data
26      56066,443       83.754222000    66      56066 > https [ACK] Seq=865 Ack=1123 Win=
39      56066,443       189.759684000   125     Application Data
44      56066,443       189.760677000   66      56066 > https [ACK] Seq=924 Ack=1296 Win=
47      56066,443       227.765381000   125     Application Data
51      56066,443       227.766048000   66      56066 > https [ACK] Seq=983 Ack=1469 Win=
55              245.692217000   104     Name query response NB 192.168.177.209
56      56066,443       287.771649000   125     Application Data
60      56066,443       287.773982000   66      56066 > https [ACK] Seq=1042 Ack=1642 Wir
76              469.543697000   342     DHCP ACK      - Transaction ID 0xfd627834
87              545.608211000   104     Name query response NB 192.168.177.209
```

Figure 3-3: Captured packets within an observed time window

Table 3.1: Extracted features from network traffic shown in Figure 3-3

Packet type	Count	Size	Lapse max	Lapse ave	Lapse min
SYN	1	74	n.a.	n.a.	n.a.
ACK	6	396	0.000243	0.000243	0.000243
ClientHello	1	343	0.109194	0.109194	0.109194
ClientKeyExchange	1	449	0.220194	0.220194	0.220194
ApplicationData	6	777	0.227939	0.227939	0.227939



```
4987    57441,443       37923.553430000 74      57441 > https [SYN] Seq=0 Win=29200 Len=0 MSS=1
4989    57441,443       37923.553689000 66      57441 > https [ACK] Seq=1 Ack=1 Win=29248 Len=0
4990    57441,443       37923.563128000 343     Client Hello
4993    57441,443       37923.563561000 66      57441 > https [ACK] Seq=278 Ack=817 Win=30848 L
4994    57441,443       37923.564060000 449     Client Key Exchange, Change Cipher Spec, Encryp
4996    57441,443       37923.569148000 119     Application Data
4998    57441,443       37923.569349000 179     Application Data, Application Data
5001    57441,443       37923.569623000 104     Application Data
5004    57441,443       37923.569922000 66      57441 > https [ACK] Seq=865 Ack=1123 Win=30848 L
5005    57441,443       37923.570866000 125     Application Data
5009    57441,443       37923.571344000 66      57441 > https [ACK] Seq=924 Ack=1296 Win=30848 L
5011            37937.475317000 342     DHCP ACK      - Transaction ID 0xfd627834
5018    57446,443       38040.415168000 74      57446 > https [SYN] Seq=0 Win=29200 Len=0 MSS=1
5020    57446,443       38040.415432000 66      57446 > https [ACK] Seq=1 Ack=1 Win=29248 Len=0
5021    57446,443       38040.480249000 343     Client Hello
5024    57446,443       38040.480712000 66      57446 > https [ACK] Seq=278 Ack=817 Win=30848 L
5025    57446,443       38040.481218000 449     Client Key Exchange, Change Cipher Spec, Encryp
5027    57446,443       38040.486216000 119     Application Data
5029    57446,443       38040.486463000 179     Application Data, Application Data
5031    57446,443       38040.486633000 104     Application Data
5035    57446,443       38040.487054000 66      57446 > https [ACK] Seq=865 Ack=1085 Win=30848 L
5036    57446,443       38040.524458000 66      57446 > https [ACK] Seq=865 Ack=1123 Win=30848 L
5037    57446,443       38041.489975000 125     Application Data
5040    57446,443       38041.490405000 66      57446 > https [ACK] Seq=924 Ack=1162 Win=30848 L
5041    57446,443       38041.490408000 66      57446 > https [ACK] Seq=924 Ack=1258 Win=30848 L
5043    57446,443       38041.490625000 66      57446 > https [ACK] Seq=924 Ack=1296 Win=30848 L
5044    57358,443       38055.889013000 97      Encrypted Alert
5047    57358,443       38055.890435000 66      57358 > https [RST, ACK] Seq=3079 Ack=7556 Win=
5048    57448,443       38055.892428000 74      57448 > https [SYN] Seq=0 Win=29200 Len=0 MSS=1
5050    57448,443       38055.892577000 66      57448 > https [ACK] Seq=1 Ack=1 Win=29248 Len=0
5051    57448,443       38055.901566000 343     Client Hello
5054    57448,443       38055.901940000 66      57448 > https [ACK] Seq=278 Ack=817 Win=30848 L
5055    57448,443       38055.903922000 449     Client Key Exchange, Change Cipher Spec, Encryp
5057    57448,443       38055.909129000 119     Application Data
5059    57448,443       38055.909359000 179     Application Data, Application Data
5062    57448,443       38055.909635000 104     Application Data
5065    57448,443       38055.909989000 66      57448 > https [ACK] Seq=865 Ack=1123 Win=30848 L
5068    57448,443       38070.917945000 125     Application Data
5072    57448,443       38070.918588000 66      57448 > https [ACK] Seq=924 Ack=1296 Win=30848 L
5081    57441,443       38107.582651000 66      57441 > https [RST, ACK] Seq=955 Ack=1328 Win=3
```

Figure 3-4: A snippet of captured traffic

Table 3.2: Extracted features from traffic shown in Figure 3-4

Packet type	Count	Size	Lapse min	Lapse ave	Lapse max
SYN	3	222	n.a.	n.a.	n.a.
ACK	15	990	0.000149	0.000224	0.000264
ClientHello	3	343	0.009138	0.027972	0.065081
ClientKeyExchange	3	449	0.010630	0.029391	0.066050
ApplicationData	3	777	0.015718	0.034489	0.071048
EncryptedAlert	1	97	155.889013	155.889013	155.889013
RST-ACK	1	66	184.029221	192.0153215	200.001422

subsequent packets. Consider the second packet in Figure 3-3, i.e. the ACK packet. In this instance, 6 ACK packets were observed; their size was 66 bytes each, totalling to $6 \times 66 = 396$ bytes. Hence, the count feature value was 6, and the size feature value was 396, which are shown in the table.

To extract the lapse features from the ACK packets, only the first packet in the connection was considered. Although there were more than one ACK packets captured, only the value from the first ACK packet in the connection was extracted. Its lapse time since the connection initiation was equal to $83.408169 - 83.407926 = 0.000243$. Since there was only 1 connection in this example (with port 56066 as shown in the second column of the figure), all of the lapse features (i.e. min, average, max) of the ACK packet were identical, where the minimum, average, and maximum value of a single value is the value itself. The same explanation was valid for the rest of the packets shown in the figure – their min, average, and max lapse features were identical since there was only 1 connection within the observed time window.

Other packets in this example followed the same procedure to obtain the feature values, as shown in Table 3.1.

As for the second example, consider the packets shown in Figure 3-4. This example presents several connections within an observed time window. In addition, this example shows a case where one of the connections was initiated outside (i.e. before) the observed time window; examples obtaining the lapse feature values from this connection will be discussed below. The features extracted from this snippet of traffic are shown in Table 3.2.

Table 3.3: ACK packet time lapse values, extracted from traffic shown in Figure 3-4

TCP port	ACK packets		SYN packets		Time Lapse
	Packet #	Time Stamp	Packet #	Time Stamp	
57441	4989	37923.553689	4987	37923.553430	0.000259
57446	5020	38040.415432	5018	38040.415168	0.000264
57448	5050	38055.892577	5048	38055.892428	0.000149

The *count* and *size feature* values of the SYN packet type were obtained as follows. There were 3 SYN packets in the observed time window; hence, the *count feature* was equal to 3, and the *size feature* value was 222, equal to the cumulative size of the 3 packets. There was no SYN lapse feature as previously discussed.

The *count* and *size feature* extraction of the ACK packet type was carried out in similar manner; hence, the values were 15 and 990, respectively. The *lapse feature* values were obtained by calculating the min, average, and max lapse time from all of the first ACK packet in each connection. There were 3 connections initiated, i.e. 3 TCP ports are opened for this example. This is illustrated in Table 3.3, with the first column showing the port numbers of the initiated connections. The time lapse between the time stamp when the ACK packet and the SYN packet were observed is shown in the last column of the table. Using these time lapse values, the min, average, and max lapse features can be derived as follows:
$min(0.000259, 0.000264, 0.000149) = Lapse\ min = 0.000149$
$average(0.000259, 0.000264, 0.000149) = Lapse\ average = 0.000224$
$max(0.000259, 0.000264, 0.000149) = Lapse\ max = 0.000264$

Other *lapse feature* values of all packets in the example were extracted in similar fashion: subtract the connection initiation time stamp from the packet time stamp. In some cases, it is possible to have a TCP connection span multiple instances, where the connection is initiated in an instance before the current observed one. This was the case with the Encrypted Alert and RST-ACK packets (with TCP port number 57358) in the above example, where the connection was

Table 3.4: RST-ACK packet time lapse values, extracted from traffic shown in Figure 3-4

TCP port	RST-ACK packets		SYN packets		Time Lapse
	Packet #	Time Stamp	Packet #	Time Stamp	
57358	5047	38055.890435	(assume)	37900.000000	155.890435
57441	5081	38107.582651	4987	37923.553430	184.029221

initiated before the observed instance, as shown in Figure 3-4. In other words, the packet did not have its associated SYN packet shown in the same instance; the SYN packet must have been initiated in one of its previous instances. Extracting the time lapse values from the packets in such cases was done using the same procedure, i.e. subtract the connection initiation time stamp from the packet time stamp. Examples of the above are given as follows.

The Encrypted Alert packet (packet number 5044) in Figure 3-4 had its connection initiated before the observed traffic sample. Its TCP connection port number was 57358; there was no associated SYN packet that initiated the connection observed in the example. To simplify the discussion, assume that this connection started at time stamp 37900.000000, which happened before the observed instance. Therefore, the time lapse value for the Encrypted Alert packet was calculated as $38055.889013 - 37900.000 = 155.889013$. Since there was only 1 such packet in this example, the *lapse features* for the Encrypted Alert packet presented the same value, i.e. 155.889013. These lapse feature values are shown in Table 3.2.

Similarly, the time lapse value for the RST-ACK packet of connection 57358 could be calculated as $38055.890435 - 37900.000000 = 155.890435$. Because there were two RST-ACK packets with different connections in Figure 3-4, their lapse values were calculated as shown in Table 3.4. The lapse features could thus be derived as follows:

$min(155.890435, 184.029221) = \textit{Lapse min} = 155.890435$

$average(155.890435, 184.029221) = \textit{Lapse average} = 169.959828$

$max(155.890435, 184.029221) = \textit{Lapse max} = 184.029221$

Table 3.5: Features used in this study

network layer	packet type	count	size	lapse		
				min	ave	max
Application	Application Data	✓	✓	✓	✓	✓
	Client Hello	✓	✓	✓	✓	✓
	Client Key Exchange	✓	✓	✓	✓	✓
	Encrypted Alert	✓	✓	✓	✓	✓
TCP	SYN flag	✓	✓	(n.a.)	(n.a.)	(n.a.)
	ACK flag	✓	✓	✓	✓	✓
	RST flag	✓	✓	✓	✓	✓
	RST-ACK flag	✓	✓	✓	✓	✓
	FIN-ACK flag	✓	✓	✓	✓	✓

In this study, all lapse values could be obtained from the relevant packet time stamps in the data. There was no need to assume values. Because all time stamps were available in this study, extracting the lapse feature values from the packets was done through applying the same procedure, i.e. subtract the connection initiation time stamp from the packet time stamp.

There were a total of 42 features in this study: 5 groups (and subgroups) of features (i.e. count, size, lapse min, lapse ave, lapse max), with each group describes the values of 9 packet types, giving an initial total of $5 \times 9 = 45$ features. With no lapse features for the SYN packets, the total number of features is $45 - 3 = 42$ features. These are illustrated in Table 3.5. The features were named according to what their values described, as is shown in Table 3.6.

The table also shows features other than those discussed in the examples above, since other network messages of different layers and types were also involved in the HTTP/2 message exchange. This is detailed in Figure 3-2, a scenario of initiating an Internet connection before sending an HTTP/2 message is illustrated. An Internet connection was initiated by a TCP 3-way handshake, followed by a TLS handshake, followed by an application-layer data exchange. Here, HTTP/2 messages were shown as Application Data – the universal term used for application-layer messages. As could be seen, the packets on the left-hand side of Figure 3-2 made up the features tabulated in Table 3.5.

In addition to those extracted from the TCP and TLS handshake discussed above, the features in this study included TCP teardown packets, i.e. TCP

Table 3.6: Features and their names

packet type	count	size
Application Data	count_app	size_app
Client Hello	count_tlsHello	size_tlsHello
Client Key Exchange	count_tlsKey	size_tlsKey
Encrypted Alert	count_encAlert	size_encAlert
SYN flag	count_syn	size_syn
ACK flag	count_ack	size_ack
RST flag	count_rst	size_rst
RST-ACK flag	count_rstAck	size_rstAck
FIN-ACK flag	count_finAck	size_finAck

packet type	lapse		
	min	ave	max
Application Data	lapse_app_min	lapse_app_ave	laps_app_max
Client Hello	lapse_tlsHello_min	lapse_tlsHello_ave	lapse_tlsHello_max
Client Key Exchange	lapse_tlsKey_min	lapse_tlsKey_ave	lapse_tlsKey_max
Encrypted Alert	lapse_encAlert_min	lapse_encAlert_ave	lapse_encAlert_max
SYN flag	(not applicable)	(not applicable)	(not applicable)
ACK flag	lapse_ack_min	lapse_ack_ave	lapse_ack_max
RST flag	lapse_rst_min	lapse_rst_ave	lapse_rst_max
RST-ACK flag	lapse_rstAck_min	lapse_rstAck_ave	lapse_rstAck_max
FIN-ACK flag	lapse_finAck_min	lapse_finAck_ave	lapse_finAck_max

packets involved in terminating a connection. Since either side of the connection, i.e. client or server, could initiate a packet signifying the end of a connection, the RST-ACK and the FIN-ACK were also captured in addition to the RST packets.

Extracting features was achieved by piping TShark output to an awk program developed in this study, i.e.

```
tshark [replay instructions] | awk -f [extract feature code] > dataset.arff
```

The output of the awk program was a file that was used as the dataset.

Dataset Creation

A snapshot of traffic within an observed time window was characterised by the above features. Each snapshot represented an instance, and several instances comprised a dataset. Hence, tabular datasets were created with the rows as the instances of the traffic, and the columns as the features of each instance. One additional column, usually placed as the last one, contained nominal data that labelled the class of each instance, i.e. flash-crowd or attack.

The datasets were represented in text-formatted files. These files can be read using `Weka` (University of Waikato, 1993–2016) – a collection of machine learning technique tools. To evaluate the goodness of the dataset, the feature values were examined for irrelevant values. The procedure involved examining the correctness of the data, such as missing or inaccurate values (e.g. zero, negatives, too large values), and the correctness of the data types. Changes or inconsistencies in the procedure during data collection might result in missing values, errors, or data type inconsistencies. Therefore, these changes were also observed and studied. The datasets created in this phase served as the input data for the following phase.

3.1.4 Phase 4: Feature Ranking and Traffic Classification

This study used Weka to rank features and run a range of machine learning techniques. Ranking features is an important step before analysing machine learning performance, because some irrelevant features can produce inaccurate classifications. For example, in Decision Trees, the most relevant values should be selected to avoid or minimize errors introduced by irrelevant features. Feature ranking addresses this problem. This study used two feature ranking techniques, i.e. Information Gain and Gain Ratio.

Information Gain is a measure of purity when a feature is taken into account. Its value can be used to measure the degree of information if a new instance were classified as a certain class. The relevance value is given by equation (3.1),

$$Gain(feature) = Info(training) - Info(feature) \tag{3.1}$$

where $Info(training)$ is the amount of information when the whole set of training example is included, and $Info(feature)$ is the amount of information when a specific feature is selected. The amount of information is obtained from an entropy function that measures the degree of coherence with respect to a each class k, which is given in equation (3.2),

$$Info(x) = -\sum_{k=1}^{n} p_k log(p_k) \tag{3.2}$$

where p_k is the probability of an occurrence that an instance was classified as k when feature x is selected. In a two-class scenario as used in this study, the information of the training example set can be simplified as shown in equation (3.3).

$$Info(training) = -p_{normal} log(p_{normal}) - p_{attack} log(p_{attack}) \tag{3.3}$$

The problem with Information Gain is that features with a large range of possible values returns a near-zero entropy value. Consequently, the Gain value of the feature becomes greater than any other features causing it to become ranked higher without truly representing its relevance.

Gain Ratio is a feature ranking measure that compensates the above drawback. It normalizes the Gain value of the training dataset with the entropy value of the feature's subsets, named the $IntrinsicValue$. Gain Ratio formula is given in equation (3.4).

$$GainRatio(feature) = \frac{Gain(feature)}{IntrinsicValue} \tag{3.4}$$

The Intrinsic Value represents the information value of the feature. It disregards any information about the class of the data sample. This is given in equation (3.5).

$$IntrinsicValue = -\sum_{v \in values(ftr)} \frac{|x \in S, value(x, ftr)|}{|S|}.log_2 \frac{|x \in S, value(x, ftr)|}{|S|} \tag{3.5}$$

where S is the number of instances in the training dataset, x is a sample of the training dataset, ftr is the selected feature to measure, $values(ftr)$ is a set of all possible values of the selected feature, and $value(x, ftr)$ is the value of the selected feature in sample x.

The drawback of Gain Ratio is that it can rank a less relevant feature high, due to the feature's low intrinsic value. Therefore, this study used both Information Gain and Gain Ratio to take the advantages of each measure and to ascertain comparison of the relevance of features. This comparison is discussed in Section 6.3.1.

Ranking features was done to study the effect of machine learning performance with a selected subset of features. Four machine learning techniques were employed in this study to classify attack and normal traffic. These are Naïve Bayes, Decision Tree, JRip, and Support Vector Machines. Machine learning classification allows the investigation to understand how the generated traffic can be distinguished. For example, performance of the classification can show how one traffic model can closely mimic another. The machine learning performance was quantified using a set of evaluation metrics, as defined below.

3.2 Evaluation Metrics

Evaluation metrics were applied in order to analyse and compare the results obtained from various observations. Evaluating classification techniques was based on metrics such as incorrectly classified instances, Detection Rate, and False Alarm Rate. These metrics were further based on True Positives (TP), False Positives (FP), and False Negatives (FN) of the data, which are explained as follows.

- TP is the percentage of attacks that the machine learning technique correctly identifies as attacks.

- TN is the percentage of normal traffic that the machine learning technique correctly identifies as normal traffic.

- FP is the percentage of normal traffic that the machine learning technique incorrectly identifies as attack traffic.

- FN is the percentage of attacks that the machine learning technique incorrectly identifies as normal traffic.

The incorrectly classified instances (equation (3.6)) are self-explanatory. It shows the percentage of instances incorrectly classified out of the total number of the whole instances S.

$$\text{Incorrectly classified instances} = \frac{FP + FN}{S} \times 100\% \tag{3.6}$$

The Detection Rate DR is also the TP rate, i.e. instances correctly classified to belong to a given class. Its value lies between 0 and 1. The results shown in this study were weighted, i.e. the number of samples from each class was considered in the calculation. Hence, the weighted Detection Rate, or $DR_{weighted}$, was computed based on the values of DR_{normal} and DR_{attack}. This is given in equation (3.7)

$$\begin{aligned} DR_{normal} &= \frac{TN}{TN + FP} \\ DR_{attack} &= \frac{TP}{TP + FN} \\ DR_{weighted} &= \frac{DR_{normal} \times S_{normal} + DR_{attack} \times S_{attack}}{S} \end{aligned} \tag{3.7}$$

where S_{normal} is the number of instances in the normal class and S_{attack} is the number of instances in the attack class.

The False Alarm Rate FAR is the FP rate, i.e. the measure of instances falsely classified to belong to a given class. Similarly, its value lies between 0 and 1. The results shown in this study were also evaluated through the weighted False Alarm Rate, or $FAR_{weighted}$ metric. This is given in equation (3.8)

$$\begin{aligned} FAR_{normal} &= \frac{FN}{FN + TP} \\ FAR_{attack} &= \frac{FP}{FP + TN} \\ FAR_{weighted} &= \frac{FAR_{normal} \times S_{normal} + FAR_{attack} \times S_{attack}}{S} \end{aligned} \tag{3.8}$$

It can be seen that the chosen evaluation metrics, incorrectly classified instances, Detection Rate and False Alarm Rate facilitated comparison and assessment of the accuracy of the various DoS attack scenarios. Hence, iterative investigations were arranged by refining the way traffic was generated and measured, based on the combination of selected features. The investigations were conducted until the evaluation metrics yielded acceptable results. The evaluation metrics also allowed the study to discuss different kinds of traffic types and techniques to launch DoS attacks (Chapter 6.3).

In order to perform the investigations outlined in this chapter, the following section details the infrastructure, tools and software.

3.3 Experimental Setup

The purpose of setting up an investigatory lab is to allow exploration of different network traffic characteristics. The setup facilitated generation of network traffic and helped demonstrate how the generated traffic consumes computing resources, significant enough to cause Denial of Service. Furthermore, it facilitated application of machine learning techniques to characterise the traffic into legitimate and malicious.

The lab setup for the investigation is illustrated in Figure 3-5. All devices pictured in the figure are virtual devices run on a desktop computer that serves the host machine. The host runs Windows 10 on Intel-i7 quad core with 64 GB RAM; and the virtual machine was VMware Player 12. The host machine could run the whole virtual machines required in this study without showing any signs of RAM exhaustion or CPU utilisation.

The network connecting the clients and the servers was comprised of VMware machines. There was no bandwidth limit set on the network.

The *client* was used to generate traffic. Various traffic patterns, attacks as well as normal traffic, were generated and studied to model several client-server scenarios. Each client ran a Debian Linux Operating System, Ubuntu 15.04, on the VMware virtual machine.

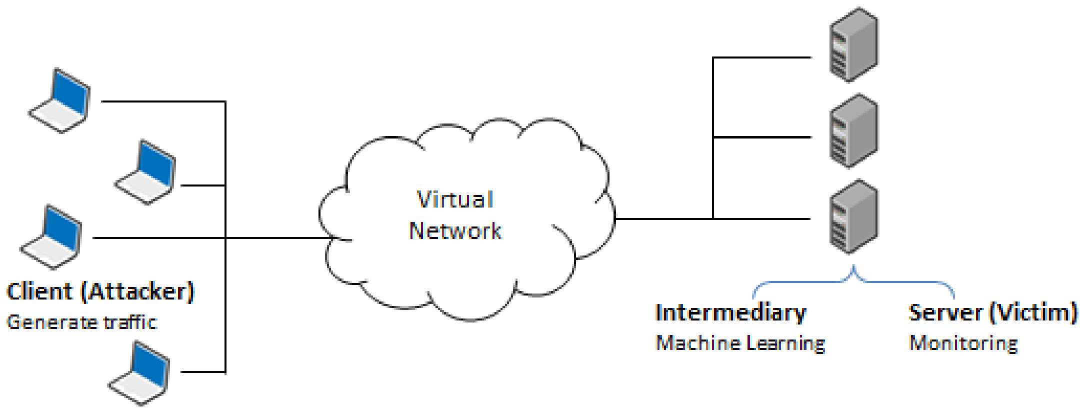


Figure 3-5: The traffic generation setup

Each client also ran a packet generator, i.e. software developed in this study to generate traffic. A packet generator is a program that allows a user to craft certain packet types to be sent at a chosen rate. The study used `nghttp2` (Tsujikawa, 2015) and `curl` (Stenberg, 1996–2016) as the library to implement the HTTP/2 protocol.

The *intermediary* intercepted traffic before it arrived at the server. Packet monitoring was installed here using publicly available tools such as `Wireshark` and its command-line version, `TShark` (Combs, 1998–2015). In this study, Wireshark was run on the host machine, capturing all traffic that passes through the virtual network (i.e. a VMware network interface). Wireshark displays graphical outputs, visualizing traffic packets that pass through a network interface. It can interactively display the content of traffic packets; hence, it was used in this study to aid in understanding HTTP/2 packet patterns sent by the clients and servers.

While Wireshark allows graphical interactions, TShark provides a command-line interface. In this study, TShark was run on the server, capturing traffic that passes through the virtual Ethernet interface on the server. The tool provides commands to select the output format of captured packets. The outputs can be saved to a file, which can be further processed for feature extraction as will be detailed in Subsection 3.1.3.

The *server* was an HTTP/2 web server. This study used a publicly available HTTP/2 server, named `libevent-server` written by the nghttp2 author

(Tsujikawa, 2015). The investigation monitored the server for effects of resource consumption. In order to detect symptoms of resource consumption, the following measurements were to be monitored (Salah, Sattar, Sqalli, & Al-Shaer, 2011): CPU usage, memory consumption, network throughput, and packet loss. When a computing resource is under load, CPU usage, memory consumption and packet loss indicators can show increasing activities, while network throughput can decrease.

Two server setups were designed to limit the above resource consumption measures to indicate activities caused by only HTTP/2 processes. First, the server only runs HTTP/2 services; it did not run web applications or back-end databases. Therefore, client requests cause the server to respond only with the Application Layer protocol it serves, the HTTP/2. Second, a simple HTML file `index.html` which contained only `"<html>HelloWorld</html>"` text message was stored on the server. Hence, upon client requests, the server responses through sending only simple text messages rather than large files.

The command `ps -ef` showed that there were 218 processes that the server executed upon startup. However, the Ubuntu System Monitoring showed that the server CPU consumption was near 0% when idle, despite having active background processes. This suggests that these background processes did not demand CPU utilisation that would interfere with the investigations in this study.

The next section provides an explanation on how the experimental setup facilitates generation of traffic and how machine learning techniques were applied for traffic classification. Evaluation metrics used to assess the result of the investigations are also discussed.

3.4 Ethical Issues

This study involved only machines. There was no human nor animal involved as a subject. In line with the Edith Cowan University procedure for PhD candidature, an Ethics Declaration approval was sought and obtained.

3.5 Conclusion

This chapter presented the methodology for data collection, evaluation and analysis adopted. It detailed the hardware equipment and software tools in a computer lab used for the investigations. The experiments were conducted to generate HTTP/2 traffic of both classes (normal and attack), following traffic models. A dataset was created from the traffic, and its evaluation metrics for analysis were detailed in this chapter. While this chapter provided overview of the methodology, the following two chapters detail the provisioning, results, and analysis of the legitimate traffic (Chapter 4) and attack traffic (Chapter 5 to 6).

Chapter 4

Legitimate Traffic Modelling and Analysis

This chapter explains how legitimate user traffic can be modelled, and how HTTP/2 traffic can be generated from the defined model. Normal traffic was subsequently built-upon to create flash-crowd traffic and a corresponding dataset. Figure 4-1 illustrates this framework.

While HTTP/2-enabled services are gaining popularity, currently most web servers still communicate using the HTTP/1.1 protocol. This implies that a sensor placed at a backbone of a computer network would not be able to tap much data on HTTP/2 traffic. Through this study, HTTP/2 traffic was subsequently modelled to mimic real user traffic and was generated as part of the experiments.

This chapter also explains how flash-crowd traffic was modelled. Flash-crowd

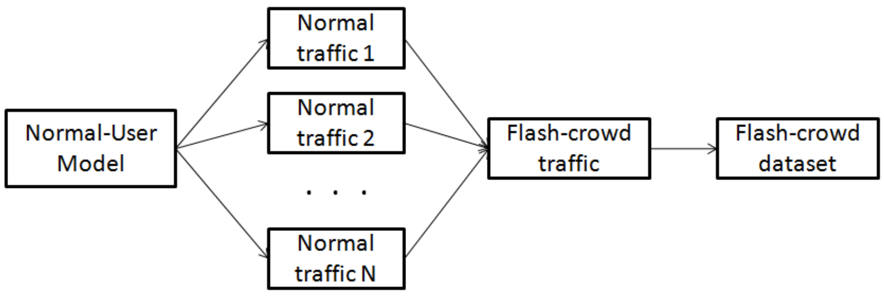


Figure 4-1: The framework for creating flash-crowd dataset from a defined normal user model.

traffic was generated from a large volume of normal traffic that best presented an HTTP/2 resource consumption on a server. Firstly, the proposed method adopted a publicly-available log that described user actions in the Internet. This was followed by a model construction task from the observed user actions. The model was simulated to generate HTTP/2 traffic, and a dataset from the traffic was created by extracting all traffic features. Secondly, flash-crowd traffic was generated from simulating the model, and the traffic was extracted through the feature extraction procedure described in Section 3.1.3 to create a normal traffic dataset.

The following section describes the publicly-available log picturing normal browsing actions.

4.1 Logs of Online User Browsing Behaviours

Behaviour of a normal Internet user was modelled from DOBBS, a publicly available log of user actions obtained from online browsing (von der Weth & Hauswirth, 2013). This log was comprised of records of user actions when a user is online, such as opening a new browser, adding a new browser tab, clicking a link or typing a web address, etc. The dataset was for a year-long activity record, and was collected from volunteers from around the world who installed a browser plugin that sent logs of these actions to a central archiving system.

In this study, a 3-day DOBBS data collection that had the most number of users and surf entries was adopted. Typically, normal users do not continuously surf the Internet; they take breaks and daily sleeps. Therefore, the 3-day sample included not only records of user actions in the Internet as described above, but also the duration of time when the users sleep overnight and other breaks. The statistics of this sample are described in Table 4.1. The table shows the number of users on each day, and the number of web surf events generated by these users. The 3-day DOBBS data is named *DOBBS Sample 1*.

There are three tables in the DOBBS log representing three types of events: first, that logged the browser's window-related data such as window mini-

Table 4.1: Dataset sample information

Sample name	Log date	# users	# distinct users	# surfs
DOBBS Sample 1	6 Aug 2013	14	21	14540
	7 Aug 2013	13		8774
	8 Aug 2013	12		9835

Table 4.2: A snippet of DOBBS Sample 1

Time	User ID	Event ID	Event description
20130826181127.900	48115555	100	New browser window opened
20130826181127.900	48115555	200	Session started
20130826181128.400	48115555	110	New browser tab opened
20130826181128.400	48115555	110	New browser tab opened
20130826181143.600	48115555	400	New web page loaded

mized/maximized, browser tab opened/closed, and whether window is focused; second, that logged session-related data such as user inactive or idle (for example due to reading the information on the browser); and third, that was related to user actions as a result of browsing activities. Each table has a *user ID* column to identify the unique user who performed the actions. For the purpose of this study, the three tables were combined and the data was sorted based on the recorded time-stamp per user ID. This log showed a story-like event illustrating how a user browsed websites during a given period of time. An example of a data sample is shown in Table 4.2.

The table illustrates that a user initialised its browser, opened two tabs, and initiated web browsing after 15 seconds. The time-stamp for each entry presented how much time a given event took.

In reality, each of the human users who took part in contributing to the DOBBS log belonged to a different time zone. They were not active and did not browse the Internet at the same time; some used the Internet at a time when others slept. Furthermore, each user showed variations on their *behaviour* such as the time it take for them to load a new page after browsing one, the amount of time they spent online, taking breaks, and sleep. Although each user behaviour differed, they all can be represented by a state transition model (described shortly), with each user having its own transition and dwell-time to represent its unique

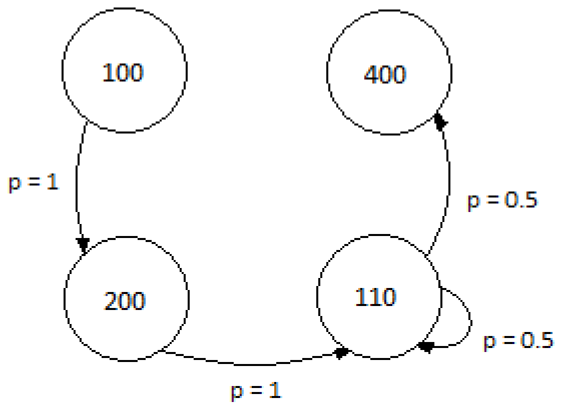


Figure 4-2: State Transition representing a User Model

behaviour. A User Model can thus be created to represent each of these user actions.

4.2 User Model

The User Model was represented in terms of states and transitions. Each recorded event was represented as a state with its own *dwell time* in that state. Specifically, the dwell time was the time an event remained in one state, before moving to another state. Therefore, each state also took into account the time spent to complete one event. Each state led to one other state or more, and the probability of a state transitioning to another state was calculated by counting its frequency of occurrence in the actual DOBBS log. This effectively modelled one sample user that browsed web sites. Figure 4-2 shows the model of traffic that was described in Table 4.2. In the figure, there was an equal chance of state 110 to transit to either state 400 or to itself, because the frequency of those transitions in the data log was equal. As for the model used in this study, the transition probability of a state was tallied from the 3-day sample, i.e. the DOBBS Sample 1, which accommodates larger data than the example given in Table 4.2. Therefore, the transition probability was more fine-grained than the results shown in the above example. The states and transition probabilities were named the *User Model.*

The User Model was constructed from a sequence of DOBBS Sample 1 entries.

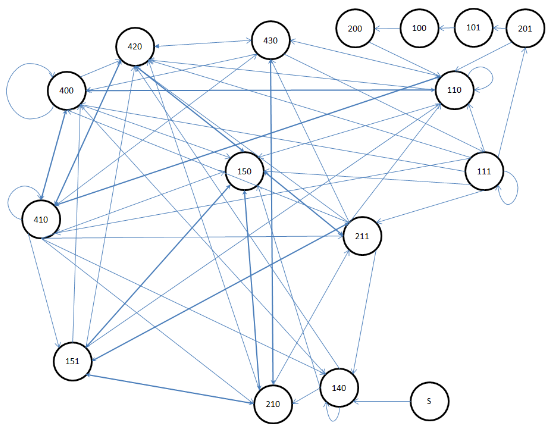


Figure 4-3: An example of one User Model taken from DOBBS Sample 1

As illustrated in Figure 4-3, the model had 16 states with many edges (transitions). The "S" state identified the first event observed in the log; therefore, it pointed to only one state in the model and acted as the starting state. The data structure of the model was coded using a two-dimensional matrix with I rows representing the current state, and J columns representing the next state. The data structure of each cell c_{ij} in the matrix had the dwell-time value for state i and the probability value to transit to the next state j. Simulating the model to transit from one state to another used a random number generator `rand()` from the Linux C library.

The defined User Model was used to generate traffic which is described as follows.

4.3 A Framework to Generate Normal Traffic

The framework describing how DOBBS log can be adopted to generate normal network traffic is depicted in Figure 4-4. Two scenarios were implemented for this research. The first one was called *Ubot*, with the "U" standing for "User". It was defined to replay one User Model and generate normal traffic that mimicked the user. Ubot takes a User Model as input, looks for a starting state and transits to other states after a given dwell-time. When Ubot is in a state that represents a user requesting a web page, Ubot generates an HTTP/2 Request packet. The

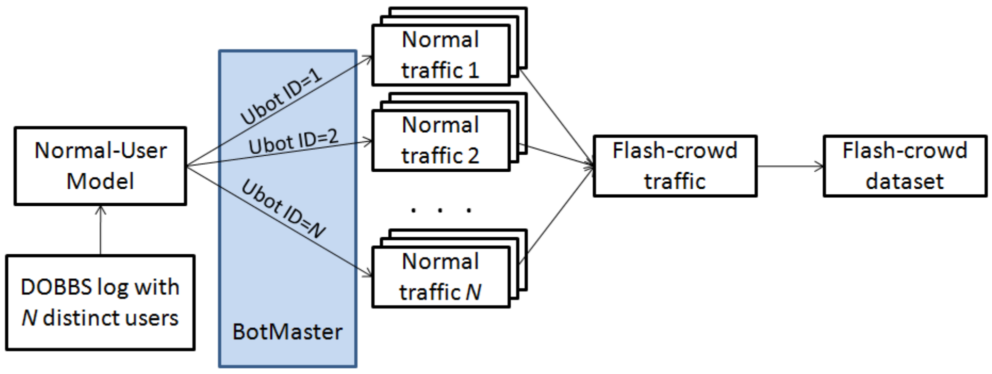


Figure 4-4: The framework on how DOBBS log was used to generate normal traffic

second scenario was named *BotMaster*, defined to run a large number of Ubot modules to generate flash-crowd traffic. Each Ubot module simulates a user. A number of Ubot modules running in tandem simulate different user behaviours, generating HTTP/2 traffic patterns that mimic normal users. BotMaster simulates flash-crowd traffic by generating a large volume of normal traffic.

Ubot simulated the User Model, with the input data acquired from the DOBBS Sample 1 (see Figure 4-4). Ubot simulated (normal) actions of 1 user, with a defined idle time on each state to represent the length of time an event took to complete, and executed another event (i.e. moved to another state) based on the probability definitions for the state transitions. Because Ubot transits to another state following a defined probability, it can show different patterns when it replays the same user. Therefore, it mimics the logged actions of a user without replaying the log verbatim. Ubot also produces variations when run repeatedly, showing different sequence of user actions, but still follows the browsing activity pattern of a specific user.

The Ubot implementation varied the pattern of user activities while maintaining the identity of that user behaviour, i.e. each user's preferred time rhythm and choice of actions. Users allocate a certain amount of time to browse and sleep; their browsing speed differs; and some launch more browser tabs than others. Although Ubot transits to another state based on a probability, its implementation maintains the identity of user behaviours. The implementation considered the phase when a user is active or idle according to its time-zone, the amount of

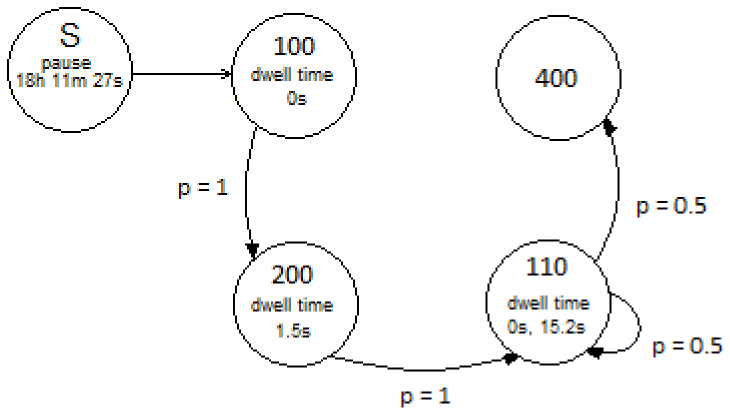


Figure 4-5: State Transition representing a User Model

break-time and sleep between activities, and the intermediate actions to complete an activity. These are described in the following procedures.

- Ubot mimicked the activity phase of the user according to its time zone. For this, a *pause* value was adopted to indicate an idle time before the first event of a user was started. When each Ubot was first started, the user that it simulated paused for the same duration as what was logged in the DOBBS log. Each user was active at a different phase; hence, each had a different pause value. For example, consider Table 4.2 as the input data. State "S" would pause for 18 hours 11 minutes 27 seconds because the first event for that user in the log occurred at that time. The pause in state "S" is illustrated in Figure 4-5. At the end of the pause time, Ubot transited to its next state indicating that the user initiated web browsing activity.

- Each state had a list of *dwell time* with each list entry signifying the amount of idle time when Ubot was in that state. Dwell time is depicted in Figure 4-5 as an element of a state. A list of dwell time in each state was collected from DOBBS Sample 1. Because a state could have more than one dwell time value, it was chosen at random for the simulation. To illustrate, using the logs from Table 4.2, the dwell time for which the user stayed in state 110 in the log was either 0 seconds or 15.2 seconds. A Ubot would randomly choose either 0 or 15.2 seconds and stayed for the chosen duration at state 110 before transitioning to another state. It then retained the dwell time it chose in memory. Suppose the Ubot firstly chose the 0-second pause, it would then pick the 15.2-second pauses when it arrived at state 110 for the second time. This rule is repeated when Ubot moves to any other state. Dwell time could also represent the user break time, a temporary withdrawal from web surfing activity, or daily sleep – which is explained next.

- Unlike robots, human sleep is usually taken overnight. Sleeping time between days could be observed within the 3-day sample (DOBBS Sample 1) as a long idle time between two consecutive days. In Ubot, the sleep time was treated the same as dwell time. As previously explained, Ubot remembered the dwell time it chose and picked another dwell time value when it reached the same state later. Once a sleep time was chosen, Ubot simulated a user until it exhausted all dwell time entries in the list. Ubot cleared its memory on the dwell time entries it chose, allowing it to regain access to the whole dwell time entries in the list including the sleep time. In other words, Ubot chooses the same sleep time value for the second time only after it has simulated all other activities. Therefore, Ubot did not sleep for more than the length the user normally sleeps, because it would not sleep twice. This arrangement allowed Ubot to mimic the duration for which a user was awake.

- When Ubot reached state 400, which according to DOBBS log means "new web page loaded," then Ubot would generate an HTTP/2 Request. The simulation code used `curl` as the programming library to generate HTTP/2 Request to a web server (as described in Section 3.3). The generated traffic was captured at the server side as described in subsection 3.1.2.

- Ubot runs indefinitely and is terminated by an operator. In this study, Ubot was terminated after a 2 GB file that captured the generated traffic was successfully collected.

Since DOBBS Sample 1 had 21 distinct user IDs in the log, this study was equipped with 21 unique user models depicting individual web-surfing behaviour.

Ubot was run to simulate any one of these users and generate traffic according to the individual patterns. Its data structure was coded using a three-dimensional matrix: a 2-dimension matrix to represent the User Model, and a one-dimension parameter designated with the user ID. Therefore, Ubot replayed any behaviour of the 21 users depending on the user ID that it was initialised with. This is depicted in Figure 4-4 with $N = 21$ users.

The second scenario implementation, BotMaster, was written to run an arbitrary number of Ubots. Its code ran *threads* – computer processes to execute multiple programming modules simultaneously. Each Ubot was assigned to a thread so that traffic could be generated simultaneously. However, a large number of threads can lead to a race condition, i.e. a situation where the behaviour of one process affected another. For example, a process can block another process from execution. Hence, the number of threads before a race condition is observed must be defined *a priori*. The challenge is that the maximum number of threads that can be launched on a machine differs depending on its computing environment. Therefore, a preliminary test was conducted in this study to find the number of threads that BotMaster could run. When tested on the virtual machine used in this study, 400 threads were found to be runnable which stopped interactively without showing any signs of blocking. However, some threads could not be stopped when 500 threads were run simultaneously, suggesting that a race condition had occurred. Therefore, for the purpose of this study, one virtual machine was set to run 200 Ubots using threads. This number was chosen to give a reasonable distance from the upper bound of 400, the number observed to avoid a race condition. It was also observed that while the BotMaster was designed to run any number of Ubots, 200 was the optimum number to run on each of the virtual machines used in this study.

To generate a large volume of HTTP/2 traffic, this study used BotMaster to mimic the behaviour of 200 users. For each of the 200 Ubots it ran, it randomly chose any one of the 21 user IDs. Because a random number was used for assigning a user ID to a Ubot, running BotMaster repeatedly would yield different compositions of user IDs assigned to the 200 Ubots. Hence, the traffic that

BotMaster generated did not show identical patterns when the simulation was replayed, or when the BotMaster was reproduced on different virtual machines. Running several BotMasters simultaneously using several virtual machines created flash-crowd traffic, which is explained in Section 4.4.2.

This section has shown a framework on how the User Model was simulated using Ubot, and how a number of Ubots could generate a larger volume of traffic using BotMaster. The next subsection discusses the evaluation of the framework.

4.3.1 Evaluating the Framework

This section shows the BotMaster simulation results when supplied with different DOBBS log samples. The expected outcome of running the simulation was that BotMaster could show that the number of visits to each state was closely related to the number of actions in the DOBBS log it corresponded to. Testing the framework included assessing all modules involved, including the User Model, Ubot, and BotMaster. Observing the simulation outputs indicate:

- The User Model internal validity. The model was expected to adapt to variations of the DOBBS log used as the input sample. For example, the list of dwell times must be able to differentiate 0 or Null values. Large dwell time values exceeding the whole duration of the sample indicated a time conversion error. The model should correctly convert overnight cases, where the clock was reset and the day count was increased. The model should also be able to adapt to different user behaviours, such as being active on one day and becoming idle on the next. Any unexpected output could question the internal validity of the model.

- The Ubot veracity to choose a next state. The next state should be randomly chosen with a probability equivalent to the tally of the corresponding event in the DOBBS log. Therefore, the number of visits to each state in the simulation should be similar to the number of logs for the event. Large deviations between the numbers shown in the simulation and that in the

Table 4.3: Dataset sample information

Sample name	Log date	# users	# distinct users	# surfs
DOBBS Sample 2	26 Aug 2013	12	18	8232
	27 Aug 2013	14		11483
	28 Aug 2013	11		11979

original log could imply that the random number generator generated unfair outputs.

- The BotMaster versatility in handling threads. BotMaster outputs depended upon the concurrency of threads to run many Ubots as though each of these were run by independent processes. When certain threads blocked other threads, some user IDs could not be correctly simulated according to their behaviours. For example one user ID could be simulated indefinitely while another could not run at all. The test done in this part of the study was able to detect such cases.

The test used two 3-day samples. One sample, the DOBBS Sample 1 had been previously discussed (Table 4.1). A second 3-day sample was extracted from DOBBS log to serve further tests and comparison. The sample, named DOBBS Sample 2, is described in Table 4.3. The second sample was chosen due to its high number of users per day, number of distinct users, and number of surfs compared to the remainder of the data in DOBBS log.

In this part of the study, BotMaster was meant to simulate a number of distinct users from a sample. Previously it was explained that a BotMaster was set to run a maximum number of 200 threads to avoid race conditions. However, for the purpose of evaluating the framework, BotMaster was set to run the same number of threads as the number of distinct user ID in the sample. Therefore, the number of threads was set to 21 when evaluating DOBBS Sample 1, and was set to 18 when evaluating DOBBS Sample 2. Each Ubot that a thread processed simulated a unique user IDs. Consequently this procedure was able to simulate different types of DOBBS samples.

Table 4.4 and 4.5 show the simulation results through running the BotMaster

Table 4.4: BotMaster mimicked DOBBS Sample-1 closely

Event/State ID	DOBBS Sample-1 count: total number of logs	BotMaster	
		average number of visits	standard deviation
100	199	190	14
101	196	190	15
110	1384	1339	73
111	1367	1333	90
140	261	253	19
150	1251	1211	47
151	1247	1201	44
155	5	5	2
200	197	188	14
201	197	189	15
210	1369	1311	28
211	1402	1347	39
215	9	8	4
400	7760	7640	348
410	10724	10567	594
420	8522	8385	520
430	6143	6034	306
total events	42233	41391	1969

using the above procedure. The table shows the number of times each state was reached during a run. For each sample, BotMaster was run 30 times to see the variations among different simulations. Each of the states in the model was visited randomly, while maintaining a similar total number of states to the total number of events within the entire simulation duration.

In simulating both samples (DOBBS Sample-1 and DOBBS Sample-2), it can be seen that the total number of logs for each event were mostly 2 standard deviations from the average number of state visits produced by BotMaster. Some of the values showed that the gap was as close as within 1 standard deviation. This meant that the events were closely mimicked by the BotMaster. The aggregate number, i.e. the total number of logs for all events were within 2 standard deviations from the simulation average. Therefore, the simulation results did not deviate significantly away from the original log.

This subsection demonstrated that the proposed framework to generate normal traffic was internally valid. The technique for mapping samples to the User

Table 4.5: BotMaster mimicked DOBBS Sample-2 closely

Event/State ID	DOBBS Sample-2 count: total number of logs	BotMaster	
		average number of visits	standard deviation
100	143	136	6
101	147	140	11
110	1587	1453	95
111	1651	1523	115
140	417	395	26
150	1224	1169	59
151	1220	1166	55
155	18	18	4
200	143	138	16
201	147	139	13
210	1703	1622	59
211	1711	1614	73
215	13	12	7
400	7537	7131	306
410	10016	9547	444
420	8025	7615	362
430	6116	5759	270
total events	41818	39576	1665

Model, which was simulated by Ubots and BotMaster, produced valid results when different samples were used. Although simulating the User Model repetitiously produced different patterns, it was confirmed that the traffic it generated mimicked normal user behaviours. In other words, the experiments in this study can confidently simulate copies of the User Model to generate a large volume of normal traffic. Normal traffic that consumed computing resources of a server can be thus labelled as flash-crowd traffic.

4.4 Flash-Crowd Traffic

While the previous section detailed how normal traffic was generated, this section explains how flash-crowd traffic was produced from the normal traffic. The generated flash-crowd traffic was subsequently captured and processed to create a dataset. This part of the framework is illustrated in Figure 4-6. The grey boxes in the figure illustrate how generating flash-crowd traffic (subsection 4.4.1) and

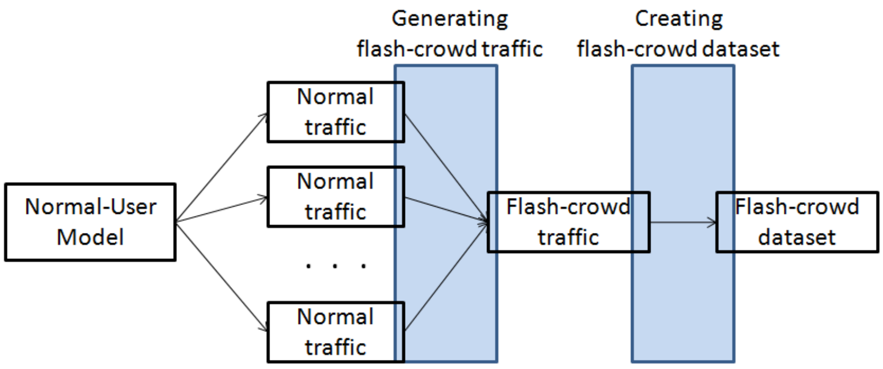


Figure 4-6: Generating flash-crowd traffic and creating dataset

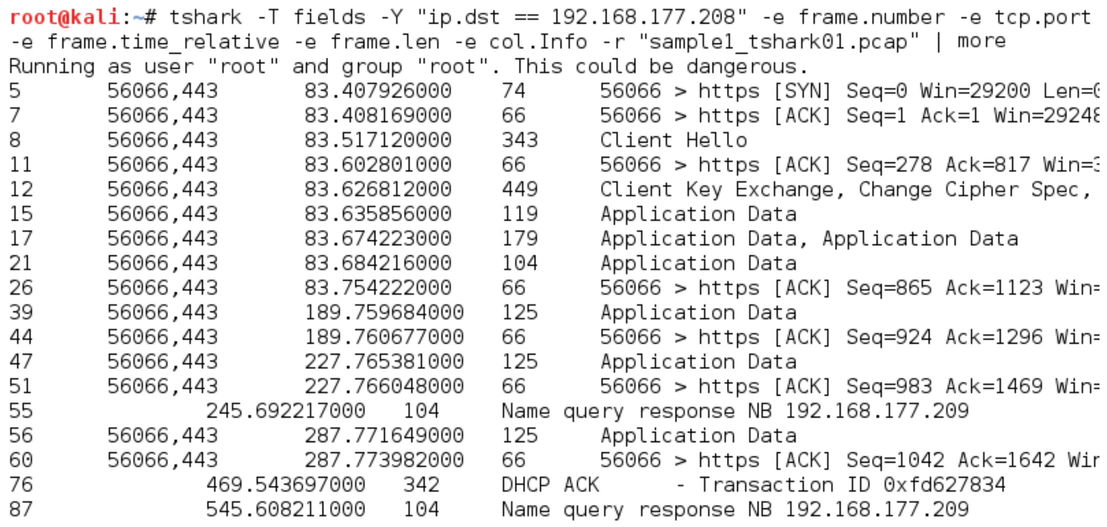
creating flash-crowd dataset (subsection 4.4.2) fit in the framework explained in this Chapter.

4.4.1 Generating Flash-Crowd Traffic

The flash-crowd traffic was generated by running several BotMasters on client machines, to target an HTTP/2 server. This is illustrated in Figure 3-5. The clients and the server were all virtual machines. To describe the left side of the figure, each client ran 1 BotMaster thread, and each BotMaster thread ran 200 Ubots. On the right side of the figure is an HTTP/2 server running on Ubuntu 15.04. Two monitoring tools were deployed on this machine: `collectl` and `TShark`. The tool `collectl` was used to monitor the resource utilisations: CPU consumption, memory usage, and network flow rate. This allowed the study to detect the condition when the server began to show signs of resource consumption. The tool `TShark` was deployed to capture the traffic generated from the clients.

A snippet of normal traffic produced by a user is illustrated in Figure 4-7. The first column represents the packet numbers; the second is the port numbers; the third is the time stamps relative to the time when the first packet was observed; the fourth is the size of the packets in bytes; and the fifth column details the packet information.

The number of clients was incrementally added to the client-server system until the server showed a sign of resource consumption. The study showed that the server reached 100% CPU consumption continually when 26 virtual machines



```
root@kali:~# tshark -T fields -Y "ip.dst == 192.168.177.208" -e frame.number -e tcp.port
-e frame.time_relative -e frame.len -e col.Info -r "sample1_tshark01.pcap" | more
Running as user "root" and group "root". This could be dangerous.
5       56066,443       83.407926000    74      56066 > https [SYN] Seq=0 Win=29200 Len=
7       56066,443       83.408169000    66      56066 > https [ACK] Seq=1 Ack=1 Win=2924
8       56066,443       83.517120000    343     Client Hello
11      56066,443       83.602801000    66      56066 > https [ACK] Seq=278 Ack=817 Win=
12      56066,443       83.626812000    449     Client Key Exchange, Change Cipher Spec,
15      56066,443       83.635856000    119     Application Data
17      56066,443       83.674223000    179     Application Data, Application Data
21      56066,443       83.684216000    104     Application Data
26      56066,443       83.754222000    66      56066 > https [ACK] Seq=865 Ack=1123 Win=
39      56066,443       189.759684000   125     Application Data
44      56066,443       189.760677000   66      56066 > https [ACK] Seq=924 Ack=1296 Win=
47      56066,443       227.765381000   125     Application Data
51      56066,443       227.766048000   66      56066 > https [ACK] Seq=983 Ack=1469 Win=
55              245.692217000   104     Name query response NB 192.168.177.209
56      56066,443       287.771649000   125     Application Data
60      56066,443       287.773982000   66      56066 > https [ACK] Seq=1042 Ack=1642 Wir
76              469.543697000   342     DHCP ACK      - Transaction ID 0xfd627834
87              545.608211000   104     Name query response NB 192.168.177.209
```

Figure 4-7: The generated traffic, captured and viewed using TShark

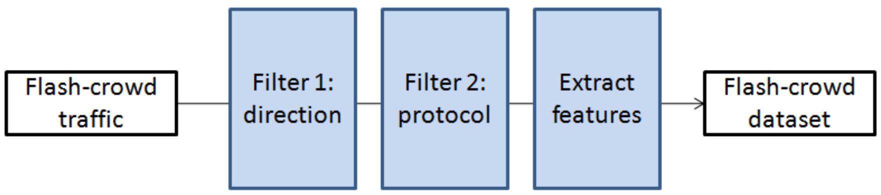


Figure 4-8: Filtering traffic and extracting features

were actively generating traffic directed towards the server. This represented $26 \times 200 = 5200$ normal users visiting a website. The captured traffic at the server side represented flash-crowd traffic, because it was generated from normal traffic pattern and it consumed the CPU utilisation of the server.

The file format was named *packet-capture* (pcap); the traffic was captured over 8706 seconds, which occupied a pcap file of size 2 GB. Currently this number is the maximum pcap-format file size, when captured using TShark. TShark was also used to replay the captured traffic for further analysis including displaying with filters, counting and searching particular events.

4.4.2 Creating Flash-Crowd Dataset

To create a dataset of flash-crowd traffic, the traffic captured was filtered and extracted. This procedure is illustrated in Figure 4-8.

The traffic captured by TShark was filtered twice, through the *direction filter*

Table 4.6: Filtering traffic messages to extract features

Network layer	Used for features	Not used
Application	HTTP/2	DNS
	SSL	DHCP
Transport	TCP	
Network	IP address	
Data Link		ARP

and the *protocol filter*. The first filter was to retain traffic only from client-to-server. This filtering procedure ascertained that the packet flow from client to server alone was captured, to represent a DoS attack. Hence, referring back to the illustration in Figure 4-7, the direction filter yielded non-contiguous packet numbers in column 1.

The second filter was a protocol filter, which operated so that only traffic involved in an end-to-end communications was considered. Messages irrelevant to a remote HTTP/2 server, such as DNS, DHCP, and ARP messages, were not considered for creating the dataset in this study. To illustrate this using Figure 4-7, traffic shown on lines 55, 76, and 87 did not pass the filter and were thus removed. Traffic that was relevant to creating the dataset was the end-to-end network data such as HTTP/2 and SSL messages (application layer), and TCP messages (transport layer). This is shown in packets 5 to 51 in Figure 4-7.

To illustrate the protocol filtering process from a common layered-network perspective, Table 4.6 groups the traffic messages into network layers that were found to be either relevant or irrelevant to the study.

After the traffic was filtered with the direction and protocol filters, it was further characterised using the feature extraction procedure. As described in Section 3.1.3, the procedure extracted the count, size, and minimum lapse, average lapse, and maximum lapse values, yielding 5 different values for each packet type. There were 9 packet types extracted as shown in Table 3.5. In this study, captured traffic was organized into 1-second instances. Therefore, each instance could be characterised by $9 \times 5 = 45$ feature values; however, there was no lapse (min, average, max) value for one of the packets (i.e. the SYN packet), yielding

42 feature values to characterise traffic in this study.

A 3,600-second flash-crowd traffic was sampled for this purpose. That is, the feature extraction procedure extracted the 42 feature values of each time frame of length 1 second of flash-crowd traffic. Therefore, the feature extraction procedure was iterated 3,600 times for obtaining 3,600 instances. The result could be arranged in a table with its columns representing the feature values, and each of its rows representing 1-second traffic instances.

This table represents the flash-crowd dataset. It describes HTTP/2 flash-crowd traffic in a 42×3600 table. Table 4.7 also summarises the values of the 42 features. Statistics were applied to represent the 3600 values of each feature through its maximum, minimum, average, and standard deviation values.

The shape of the flash-crowd traffic can be understood from the *lapse* feature values. Table 4.7 shows that some of the lapse feature values showed 0 minimum values, denoting an instance where the packet was sent immediately after its previous one. On the other hand, the maximum value of the lapse features was mostly several standard deviations away from 0, signifying that a special network condition occurred. For example, the mean values for $lapse\_app\_min = 0$, $lapse\_app\_ave = 0.032$, and $lapse\_app\_max = 1.563$. These values indicate that the Application Data packets sent by clients were received by the server 0.032 seconds on average after their connection was initiated. However, the Application Data packets can also take 1.563 seconds, or $1.563/0.032 = 48.8$ times higher than average, from connection initiation until they were received by the server. One explanation for having a very high lapse value was that the server was busy and incoming packets were queued. Hence, this observation conforms to the general understanding of flash-crowd traffic descriptions that legitimate traffic can consume server resources.

Furthermore, Table 4.7 describes the nature of the flash-crowd traffic, indicated by the gaps between the minimum and maximum values of the *count* and the *size* features. The server was not made busy during any instance of time where the volume of packets was at its minimum, but it found to approach its serving limit during a time instance when the features showed its maximum val-

Table 4.7: Flash-crowd traffic characteristics described through its feature values

no	feature name	min	max	mean	s.d.
1	count_app	18	398	201.553	34.783
2	size_app	2356	60634	30610.343	5264.759
3	lapse_app_min	0	0	0.000	0.000
4	lapse_app_ave	0	6	0.032	0.264
5	lapse_app_max	0	156	1.563	11.530
6	count_syn	33	132	76.790	13.772
7	size_syn	2442	9768	5682.460	1019.131
8	count_ack	16	464	223.777	41.804
9	size_ack	1056	30648	14778.687	2762.303
10	lapse_ack_min	0	0	0.000	0.000
11	lapse_ack_ave	0	23	0.042	0.654
12	lapse_ack_max	0	691	1.966	20.826
13	count_rst	0	231	73.048	25.134
14	size_rst	0	13860	4382.883	1508.043
15	lapse_rst_min	0	274	1.057	11.967
16	lapse_rst_ave	0	274	3.785	15.935
17	lapse_rst_max	0	873	25.631	67.556
18	count_rstAck	0	69	39.523	9.075
19	size_rstAck	0	4554	2608.503	598.944
20	lapse_rstAck_min	0	518	1.359	15.516
21	lapse_rstAck_ave	0	518	6.279	21.992
22	lapse_rstAck_max	0	979	36.71	79.556
23	count_finAck	0	112	24.902	12.201
24	size_finAck	0	7392	1643.528	805.276
25	lapse_finAck_min	0	401	1.831	18.409
26	lapse_finAck_ave	0	401	7.003	23.367
27	lapse_finAck_max	0	934	36.107	76.897
28	count_tlsHello	17	139	76.793	14.2
29	size_tlsHello	6343	51997	28695.703	5315.083
30	lapse_tlsHello_min	0	0	0.000	0.000
31	lapse_tlsHello_ave	0	0	0.000	0.000
32	lapse_tlsHello_max	0	3	0.011	0.137
33	count_tlsKey	0	19	6.154	2.475
34	size_tlsKey	0	4883	1581.549	636.142
35	lapse_tlsKey_min	0	1	0	0.017
36	lapse_tlsKey_ave	0	179	0.051	2.984
37	lapse_tlsKey_max	0	896	0.271	14.934
38	count_encAlert	1	155	76.713	15.104
39	size_encAlert	97	15035	7441.139	1465.058
40	lapse_encAlert_min	0	581	1.391	21.877
41	lapse_encAlert_ave	0	581	3.168	24.39
42	lapse_encAlert_max	0	800	23.864	68.85

ues. For example, the count_syn feature values reveal that flash-crowd traffic is made of 77 connections per second on average, which ranges from 33 to 132 connections per second.

The average number of TLS handshake initiation per second, denoted by the count_tlsHello feature, equalled to 76.793. This number is almost identical to the average number of TCP connection initiation per second, count_syn, which equalled to 76.790. This demonstrates that both the TCP SYN packet and the TLS Hello packet are always in pair in initiating a connection, as illustrated in Figure 3-2.

The count_ack mean feature value was 223.77, which is 3 times higher than the count_syn mean feature value of 76.790. This means that the server receives on average 3 ACK packets in a connection. Upon an investigation, these 3 ACK packets were used to acknowledge a SYN packet for connection initiation, a TLS Hello for TLS handshake initiation, and an HTTP Response page. Flash crowd traffic can be identified through having count_ack feature values between 16 and 464. Table 4.7 shows that other *count* and *size* feature values also show that flash-crowd traffic can be identified through feature values that lie between these low and high traffic measures.

4.5 Conclusion

This chapter showed how legitimate user behaviour while online could be modelled, and how flash-crowd traffic could be generated from the defined model. The study produced a flash-crowd dataset and showed its feature values. This dataset can be used to analyse how attack or anomalous traffic deviates from the normal traffic. The following chapter presents a scheme to model and detect attack traffic.

Chapter 5

Attack Traffic Modelling and Analysis

The earliest Denial of Service (DoS) attack reported in the literature was launched in 1974 (Heron, 2010; Dear, 2010). The attack required only one client command to cause 31 remote machines to become unresponsive. The command was designed to share the use of external devices in the network. The command launched data transmission to an external device connected to another machine in the network. However, in the absence of an external device, the machine indefinitely waited to communicate with the non-existent device and was unable to serve other input devices such as the keyboard. The attacker sent such a command to each of the machines in the network, causing all 31 of them to become unresponsive to keyboard commands. DoS attacks in the subsequent years demanded launch of more traffic from an attacking client. In 1996, a flood of ping packets (discussed in page 19) was found to cause target machines to freeze (CERT, 1996). Subsequently, the Internet has facilitated means for attackers to increase the volume of traffic to flood a target more effectively. Internet-connected machines have been used to send simultaneous traffic, creating a Distributed DoS (DDoS) attack to incapacitate a target (Chang, 2002). Henceforth, flooding-based DoS attacks after the year 1999 have been generally distributed in nature (Zargar et al., 2013).

This chapter shows how DoS attack traffic against an HTTP/2 service can be modelled. It details the impact of launching HTTP/2 packets with varying parameter settings to depict CPU utilization and memory consumption of a target machine running an HTTP/2 server. The chapter is divided into two sections. The first section presents a model of a flooding-based attack, and the second section presents a model of a distributed attack.

5.1 Flood Attack

Prior to this study, flooding-based detection techniques discussed in the literature (subsection 2.3.2.2) analysed floods of HTTP/1.1 traffic. The techniques employed HTTP/1.1 Request packets to create floods against web servers (Zargar et al., 2013). Attackers at the client side launched a big amount of packets to the server. As a result, the server was incapacitated.

In contrast, a flood of HTTP/2 Request packets observed in this study could not be used to incapacitate a target running an HTTP/2 service. A client machine was designed in this study to continuously launch HTTP/2 Request packets to an HTTP/2 server. The client used `nghttp2` library to continuously generate HTTP/2 `data` packets containing GET requests, and sent these packets towards the server. However, the server was not incapacitated. The client machine, instead of the server, reached 100% CPU utilization during the time that it launched continuous HTTP/2 Request packets. This shows that the flooding-based attack against the HTTP/2 server was unsuccessful. Therefore, flooding-based techniques previously used to incapacitate HTTP/1.1 services, such as launching a big amount of HTTP/1.1 Request packets, cannot be adapted to attack HTTP/2 services.

However, HTTP/2 packet types are not restricted to an HTTP/2 Request packet. HTTP/2 standard (Belshe et al., May 2015) defines 10 frame types, i.e. the `ping`, `data`, `settings`, `window_update`, `headers`, `priority`, `rst_stream`, `push_promise`, `go_away` and `continuation` frames. This section further explores other HTTP/2 packet types, particularly the window_update and the ping packets, to fit a model that allowed a client to launch flooding-based HTTP/2 DoS attack traffic towards a target. Such an attack model is introduced in the following subsection.

5.1.1 Attack Model and Scenarios

An attacking client was designed to generate and launch HTTP/2 attack packets towards a target server. The attack packets were generated at the client side based on a flooding-based attack model. The aim of the investigation in this section is to find HTTP/2 packet types that can be employed as attack packets, which consumes 100% CPU usage of a target server.

The flooding-based attack is modelled based on two intuitions. First, a target server can be incapacitated when processing a big amount of packets. Therefore, the proposed model allows the investigation to launch a predefined *number of attack packets* from an attacking client to a target server. Second, sending continuous attack packets could overload the CPU usage of the attacking client. Reducing the attack flow, i.e. the number of packets launched per second towards the target server, could still incapacitate the server. Hence, the model allows the investigation to add and adjust a *time delay* between successive attacking packets.

The attack model proposed in this study is shown in Figure 5-1. The model allows test cases in this study to generate a cumulative N number of packets. Hence, a variable *sent* is initialised to 0, to set that no packet has been generated. The model generates and sends one HTTP/2 packets towards a target HTTP/2 server. Hence, the variable *sent* is incremented each time after an HTTP/2 packet is generated, shown in the figure as $sent = sent + 1$. This process is iterated until a *sent* equals to the cumulative number of packets N. The terminating condition is shown in the figure as a $sent == N$ conditional statement. The process terminates when the condition is true.

Furthermore, the model provides a conditional statement to add a time delay before sending an HTTP/2 packet towards a target server. When set to *true*, the conditional statement `add time delay` allows a test case to set a delay with a 1 nanosecond granularity, and allows the model to pause for the given time

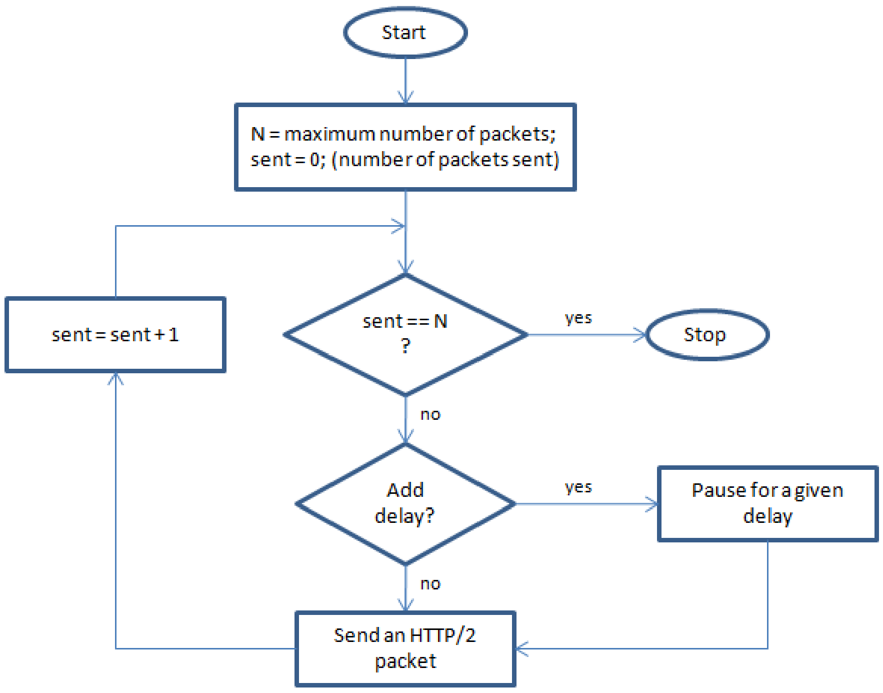


Figure 5-1: The flooding-based attack model

delay before sending an HTTP/2 packet towards a target server. To bypass this behaviour of adding a time delay, the conditional statement should be set to *false*.

In this study, the effects of varying different HTTP/2 packet types on the CPU and memory consumption of a target were observed. This is explained in Investigation-1 of the following subsection. Investigation-2 furthered the experiment through observing the number of target machines that a single client was able to flood. The effect of adding a time delay between attack packets on the computing resources of a target was observed in Investigation-3.

Investigation 1: HTTP/2 Packet Types

The first investigation was to examine how DoS attack could be launched against an HTTP/2 server. The following lab configuration was set up to facilitate the investigation. Two VMware Player virtual machines were deployed: one hosted a client (as the attacker) and the other hosted a server (as the victim). Each virtual machine was configured to have 1 processor core with 1 GB RAM, and ran Ubuntu 14.10 Linux distribution. The client and the server were connected through a 100 Mbps virtual network. The two virtual machines were run on the same host machine.

Several test cases were run to launch various packet types, i.e. `settings`, `data`, `ping`, and `window_update` packets. Some of the results were not in accord with the expectancy. For example, sending `settings` frames more than once generated an error; and transmitting `data` frames containing GET requests did not successfully cause resource depletion. It was found that the client machine (instead of the target) reached 100% CPU consumption when the `data` packet was used to flood the target. Of interest in this investigation, two packet types i.e. the `ping` and `window_update` frames could be used for further observation, as they allowed the flood-based attack model (Section 5.1.1) to iteratively send HTTP/2 packets.

The ping packet discussed in this section is an HTTP/2 frame types at the application level, as not to confuse it with ping messages at the network level, notoriously known to cause DoS attacks (Section 2.2.1). The ping frame is part of the HTTP/2 protocol.

The window_update packets are used to control the amount of bytes an HTTP/2 machine can send (discussed in Section 2.4.4). Every HTTP/2 packet is tagged with a *stream ID*, which allows packets to be grouped and transmitted asynchronously to a remote machine, regardless of the packet flow of other groups. While every HTTP/2 packet is tagged with a stream ID, the window_update packets can control the flow of different stream IDs. In other words, window_update packets allow flow control, grouped by the stream IDs of the packets.

The window_update frame format, shown in Figure 5-2, has a payload named *window-size-increment*. The figure shows that the window-size-increment occupies 31 bits of the frame. In addition, the Figure also shows a 1-bit reserved payload denoted with "R" – this reserved payload is not discussed in this study.

The window-size-increment value indicates the maximum length of a frame that the machine (that sends the frame) can transmit in addition to its previous value. This is illustrated in Figure 5-3. In the figure, a machine running HTTP/2

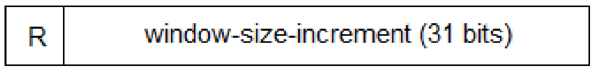


Figure 5-2: The window_update payload format

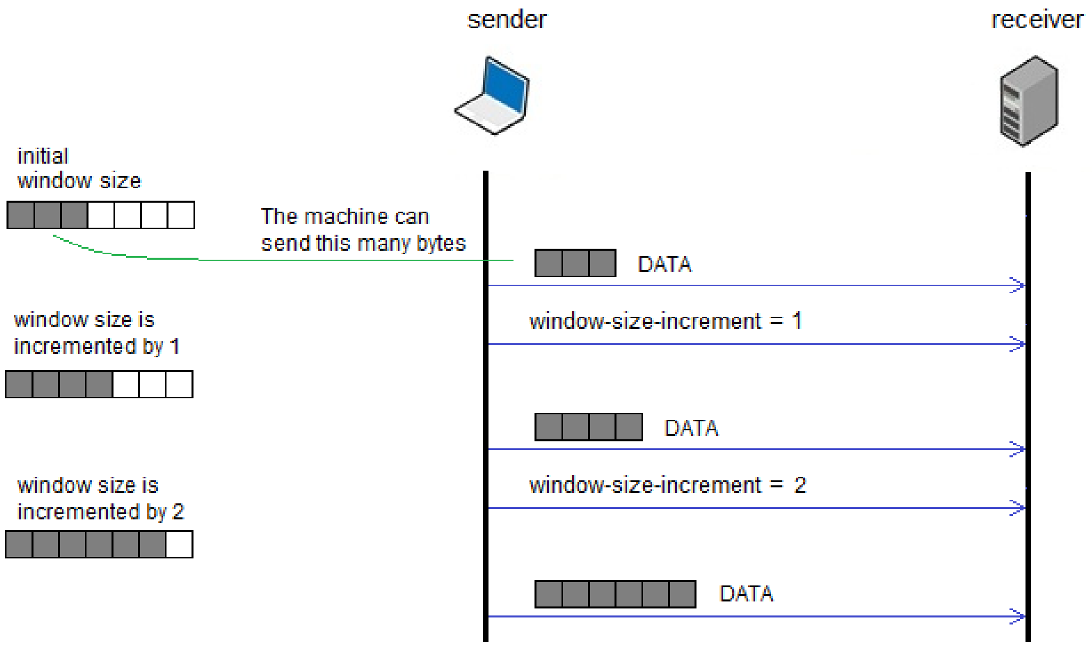

sender
receiver
initial window size
The machine can send this many bytes
DATA
window-size-increment = 1
window size is incremented by 1
DATA
window size is incremented by 2
window-size-increment = 2
DATA

Figure 5-3: The window-size-increment tells the receiver how many more bytes the sender can transmit

service initially kept a local value of window size. This value signifies how many bytes the machine can transmit. In this example, (suppose) the three shaded boxes meant that the machine could send three bytes of data. After it sent a window_update packet with window-size-increment value of 1, the machine increased its local window size to 3+1 = 4 bytes. This allowed the machine to send 1 more byte. The machine henceforth sent four-byte data to a remote machine. Likewise, a window-size-increment value of 2 indicates that future frames can send 2 more bytes. As illustrated in the figure, the machine increased its window size to $4 + 2 = 6$ bytes, and henceforth sent six-byte data.

The traffic generation setup for Investigation-1 is shown in Figure 5-4. A client machine run the flood-based attack model to generate attack traffic towards

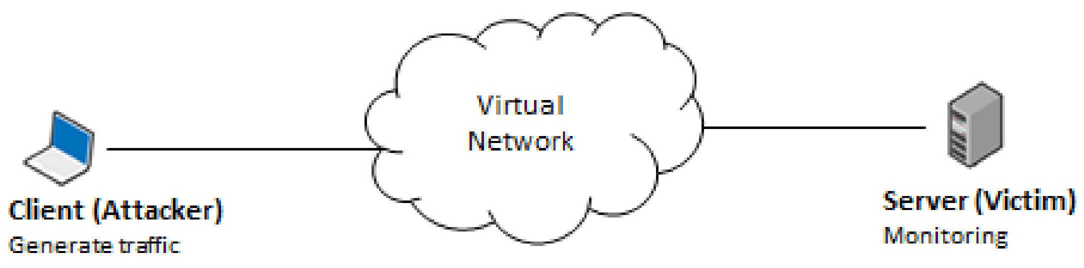


Figure 5-4: The traffic generation setup for Investigation-1 and Investigation-3

a target server. A code called *packet generator* was constructed using `nghttp2` library to generate a large number of HTTP/2 window_update packets as the attack traffic. The packet generator was equipped with some user-supplied input parameters to allow control over the total number of HTTP/2 stream IDs and the window-size-increment value used to launch an HTTP/2 packet.

In this investigation, five observations were noted when a flood of packets against an HTTP/2 server were generated. For each observation, the packet generator sent one test case of crafted HTTP/2 frame packets against the server. Hence, there were five test cases in total. These are :

- Test case 1: 2M ping packets were sent to the victim.
- Test case 2: 2M window_update packets were transmitted on stream 0, with random window-size-increment.
- Test case 3: 2M window_update packets were transmitted on stream 0, with fixed window-size-increment.
- Test case 4: 10K window_update packets with random window-size-increment were transmitted on each of 100 different stream IDs.
- Test case 5: 10K window_update packets with fix window-size-increment were transmitted on each of the 100 different stream IDs.

Each experiment was repeated 30 times to reduce the variance in the results obtained, and to improve the overall confidence in the findings. The next section presents the results of the observations. It is important to note that test case 1 required sending ping frame packets at the application level, as not to confuse it with the ping messages sent at the network level, known to cause DoS attacks (Section 2.2.1). The ping frame is part of the HTTP/2 protocol.

In order to send packets in different stream IDs as done for test cases 4 and 5, a `header` frame was sent to the receiver to open a new stream ID. Hence, there were 100 header frame packets sent to create 100 different stream IDs.

Investigation 2: The Power of the Attack

The second investigation was to observe how many servers a single client can flood. The intuition behind this was that if the client required minimal computational and storage resources to launch a DoS attack, multiple victim machines could be targeted. This is opposed to the nature of DDoS attacks, where a single server is attacked by many client machines.

In this investigation, one server was added in the network each time all servers in the network have shown 100% CPU usage. This is shown in Figure 5-5. The client was used to generate attacks according to the five test cases discussed in Investigation-1. In this investigation, one client was used to send attack traffic to all servers in the network. The investigation was concluded when any one of the servers did not yield a near 100% CPU usage. The number of servers was observed at the conclusion.

Investigation 3: Adding a Delay

The third investigation was to observe if a time-delay could make an attack to become stealthier. As shown in Figure 5-1, the attack model in this section allows a time-delay to be inserted between consecutive attack packets. The intuition behind this was that a time-delay would reduce the attack traffic rate. If the attack could be lowered gradually, the lowest possible packet rate that will translate to a successful DoS attack would be observed.

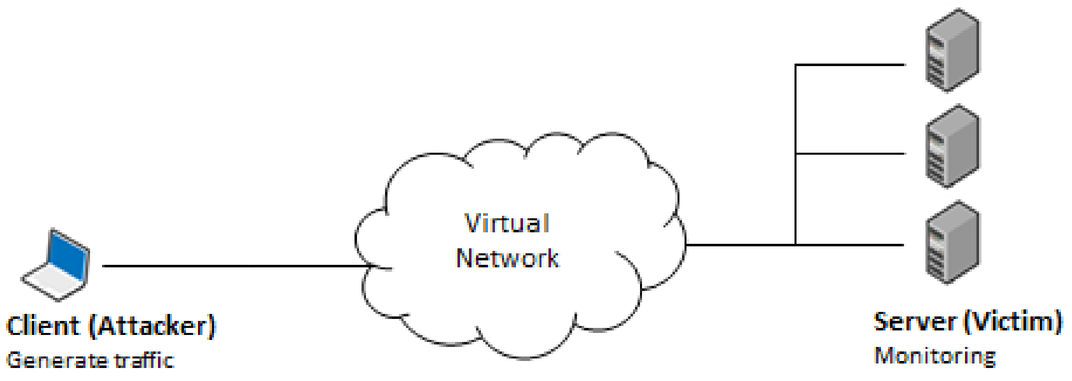


Figure 5-5: The traffic generation setup for Investigation-2

To implement a pause between attack packets, the programming interface nanosleep() was used in the code. The interface allowed the study to vary a delay up to a minimum of 1 nanosecond, before a packet was generated. The same five test cases were run again to observe the effect of adding a time-delay to the attack packets, on the behaviour of the victim machine.

The traffic generation setup is shown in Figure 5-4. A client implemented the flooding-based attack model to generate attack traffic towards a server. A delay was added before each packet was sent towards the server. This investigation varied the delay value to find a minimum value that caused the server to show 100% CPU usage.

5.1.2 Results

This section presents the results of the observed parameters of the server when subject to the attack. The effect of flooding different types of HTTP/2 packets to a server was investigated in five test cases (page 121): first, flooding with ping packets; second and third, with window_update packets on one stream with random and fixed window-size-increment respectively; fourth and fifth, with window_update packets on 100 stream IDs with random and fixed window-size-increment respectively.

Since the server resources were the parameters of interest to observe in this

Table 5.1: Computing resource consumption during attacks

Test case	CPU		size (KB/sec)		count (packets/sec)	
	ave	s.d.	ave	s.d	ave	s.d.
1	98.56	6.29	403.98	45.67	274.06	31.81
2	94.80	17.23	320.40	88.38	224.04	78.20
3	88.39	24.22	305.35	120.79	219.94	103.73
4	97.99	8.82	321.42	127.00	223.71	121.04
5	98.14	7.46	324.59	121.26	226.41	122.60

study, the host and client resource parameters were not considered for the analysis. It is useful to note that the host machine did not show any sign of resource depletion during the attack traffic generation, indicating that the observed resource parameters at the victim machine were not affected by the underlying host environment. Similarly, the client machine did not show any sign of resource depletion, suggesting that it did not require high computing resources to produce such attacks.

Result of Investigation 1

The results from each test case are shown in Table 5.1. The table shows the average (ave) and standard deviation (s.d.) of the server resource parameters: the % CPU consumed, the size of packets received per second, and the number of packets received per second.

It can be seen from the table that the average CPU consumption was near 100% for all cases. During the flooding-based attack, a simple HTTP/2 Request sent to the server was not responded to. That is, in all five observations, the server did not respond to a page request that was sent from another client terminal during the attack. The response page was received by the client as soon as the attack ended. This indicated that the DoS attack was successful in incapacitating the victim.

The server's free memory was stable at 235 MB when idle. When it was attacked based on test case 1, the memory was consumed at about 1.5 MB per second. However the available memory was then stable between 67 - 80 MB even though the attack was still ongoing. In other words, the attack consumed the server memory up to 168 MB. The test results differed when test cases 2 - 5 were run; the server memory only consumed up to 2 MB during the attack.

In all five observations, ICMP ping packets were sent from the client to the server to test network conditions such as packet loss and round-trip delay. When the network that connects the client to the server is not congested, a client receives responses for ICMP ping packets that it sends to a server. In other words, a network condition such as network congestion had occurred when a client did not receive the ICMP ping packet it sent to a server. In this study, packet loss tested through ICMP ping tests was 0%, suggesting that the network was not congested, and did not contribute to packet loss.

ICMP ping can also yield round-trip delays, which is a metric showing the time it takes for a client to receive back ICMP responses from a server. In all five observations in this study, the round-trip delay was doubled from about 500 ms when the server was idle to 1 second during an attack. This suggests that the server responded slower to ICMP ping packets during the attacks. As it was shown that the network was not congested, the slow response was due to a busy server, rather than a congested network. Hence, the doubling of round-trip-delay values shows that the server was busy.

It was difficult to sample the average and variation of the round-trip delay since different measurement tools were used: ICMP ping packets were used to measure the round-trip delay, while a collectl tool was used to monitor the other parameters. This created difficulties to align the data collected. The key observation was that there was no noted packet loss during the attack thus showing that the high CPU usage observed at the server was not due to a congested network. Because the victim machine only run an HTTP/2 server, the high CPU usage was due to a DoS attack against an HTTP/2 service.

Result of Investigation 2

After running 12 servers (as the victims) and 1 client (as the attacker), all servers were still successfully attacked. The host machine had very low remaining available memory to run a further experiment, thus the investigation concluded with the 12 servers. During the attack, all servers showed CPU usage of almost 100% and they could not respond to a given client request page. The results were consistent when attacks were launched using each of the five test cases, i.e. all test cases caused the 12 servers to show near 100% CPU usage.

Result of Investigation 3

As expected, adding a delay between attack packets yielded less CPU usage than what was shown in Table 5.1. However, when a simple page request was sent by the client in this investigation, the requested page was promptly returned by the server. This indicated that the server was still able to serve incoming requests in a timely manner. The delay value was varied during the investigation until the smallest fraction of delay that the programming interface could provide (1 nanosecond) was attempted, yet the server was not incapacitated from providing service.

It was observed that larger time delay (e.g. 100 ns) yielded lower CPU usage at the victim's end. This result confirmed the internal validity of this experiment: less frequent delays between offending packets should cause less stress on the victim. The results were also consistent when attacks were launched using each of the five test cases.

This investigation also showed that adding a time-delay did not successfully incapacitate the server, yet caused lower CPU utilization of the HTTP/2 server.

5.1.3 Discussion

Interpretation of the Results

In all five test cases sending attack traffic, the server was incapacitated during the attacks: Table 5.1 shows that the CPU utilization was near 100%. Furthermore, it was observed that the server did not send a response page during each of the five attack test scenarios, confirming that the DoS attacks were successful.

In all five scenarios tested, the network was not congested for two reasons: first, the observed packet sizes were 3 to 4 hundred KB per second (shown in

Table 5.1) which were much lower than the 100 Mbps virtual network bandwidth; and second, the ICMP ping requests yielded no packet loss. The latter signified that the network still had available capacity to receive network packets, i.e. the attack packets and the ping packets. Because the network was not congested, the inability of the server to respond to a requested page as described above was due to its CPU utilization approaching 100%.

Much of the server memory remained available during the attack, suggesting that the observed server CPU consumption was not due to its limited memory setting of 1 GB. As a comparison, the client did not show any resource consumption despite it operated at the same 1 GB memory setting. This demonstrated that the client did not require much memory or high CPU processing power to create, buffer and send attack packets to a target. Rather, it was the processing of these packets at the server that caused CPU resource depletion. This was confirmed through the second investigation which showed that one client could still pose computing power to successfully attack many machines that processed HTTP/2 packets. Hence, it can be stated with confidence that processing of HTTP/2 packets at the receiving end (server) required a lot of CPU usage.

Implication of the Observations

The study in this section indicated that it was possible to detect HTTP/2 flooding-based DoS attacks when launched through the five proposed test cases. Table 5.1 shows that attacking the server through test case 1 caused the highest CPU consumption (98.56%). However, test case 1 also yielded the highest packet size per second (403.98), and the highest numbers of packets per second (274.06). Furthermore, the rate at which the server memory was consumed at 1.5 MB per second proved to be an indication of an attack. As a result, it was possible to detect the test case-1 attack or to trigger an alarm when the memory resources depleted at higher than normal rate.

It is more economical for an attacker at the client side to incapacitate a target with less effort. Attacks demonstrated through test cases 2 - 5 proved so: lower-rate attacks could incapacitate an HTTP/2 server. In addition, these attacks did not consume noticeable memory. This implied that through certain test cases, attacks could be made more efficient, hence stealthier. Such stealthy attack traffic, when inserted with legitimate traffic, can be difficult to detect.

It was still unclear whether launching attacks through different stream IDs (as in test cases 4 - 5) were stealthier than attacking through only 1 stream ID (as in test cases 2 - 3). As shown in Table 5.1, the packet size per second and the number of packets per second did not differ significantly in these test cases (2 - 5). The investigation did not reveal if sending attacks in different stream IDs were more difficult to detect when legitimate traffic was inserted. However, since HTTP/2 traffic typically runs through multiple streams at a time, test cases 4 - 5 mimicked the legitimate traffic closely. Currently, HTTP/2 is not yet widely used; hence, legitimate HTTP/2 traffic dataset is not yet available. There is room for future study to observe if test cases 4 - 5 were stealthier than the others, which became the motivation for the study reported in the following section (i.e. Distributed Attack, Section 5.2).

External Validity

Through the second investigation, it was observed that one client machine could attack many (in this case 12) servers simultaneously. This meant that it did not need distributed machines as in a DDoS attack to successfully disrupt HTTP/2 services. Since the client still had available computing resources, it was possible for an attacker to mimic a DDoS attack through a single client.

However, this observation might not be externally valid. The investigations were done using virtual machines where there was no physical distance between the machines. When a 1 ns delay between the attack packets was inserted in the third investigation, the server was still able to serve client requests by sending a response page, indicating that the DoS attack was not successful. In reality, several millisecond delays could be noticed between two remote client-server end points. Hence, when time-delay in the communication channel was considered for launching the five attack test cases, a flood of HTTP/2 traffic was not able to incapacitate the servers as demonstrated here.

Nevertheless, the ability of one client to successfully attack many servers simultaneously that this study observed is externally valid for the following reasons. Real web applications host services that interact with databases and send big files. They are busier than the idle servers used in this investigation. Furthermore, they are visited by many clients simultaneously. When the attack traffic proposed in this study is combined with legitimate web traffic, real HTTP/2 servers in the Internet are successfully attacked.

5.1.4 Conclusion

The study demonstrated flooding-based DoS attack models targeting HTTP/2 servers. The test cases 1 - 5 showed how the attacks could be launched against a target server. However, the demonstrated attacks were still detected through observing whether the victim memory resources gradually deplete at a certain rate. It had been previously argued that it was possible to launch stealthier DoS attacks through sending packets based on traffic generation rules that exploit the way a victim processes HTTP/2 packets.

The results showed that a single malicious client flooding with window_update packets can successfully attack 12 servers at any time. This was in contrast to flooding a server with HTTP/2 Request packets, which consumed the computing resource of the client. Adding a time-delay of 1 ns between consecutive attack packets did not make a successful attack; hence, DoS attacks using the test cases herewith presented could not be made stealthier through adding time-delays higher than 1 ns between the attack packets. The demonstrated flooding-based attack model in this section was extended to construct DDoS attacks models that will be presented in the next section.

5.2 Distributed Attack

The previous section presented the effect of flooding-based attack on HTTP/2 window_update packets, and how they could incapacitate an HTTP/2 server.

While the previous section employed one attacking client, this section presents how a distributed attack, or DDoS attack, employing more than one client could be modelled. The intensity of DDoS attack traffic was analysed to see if it could bypass a hypothetical intrusion-detection system that monitored CPU consumption and increased memory usage activity.

The study aimed to generate *stealthy* attack traffic, where each attacking client caused a target server to consume 50% CPU usage, yet caused the target server to consume 100% CPU usage when a number of malicious clients were employed in attack traffic generation. Hence, further parameter values were investigated in this study. These are the window-size-increment value, the number of attack packets, and the number of malicious clients (bots) used. These parameters affected the intensity of DDoS attack traffic. Furthermore, this section describes how the attack traffic could be classified using machine learning techniques; including steps to train and test on network traffic packets.

5.2.1 Attack Model and Scenarios

A DDoS attack model is shown in Figure 5-6. This model extends the previously introduced flooding-based attack model (Figure 5-1) in setting a predefined value for the total number of packets N to be sent towards a target server, and setting a predefined window-size-increment value. The remaining procedures defined by the model remained equal as the flooding-based attack model; a packet is sent towards a target server until the number of packets sent equals to the predefined total number of packets N.

The DDoS model in Figure 5-1 allows the investigation in this section to find two parameters, i.e. the maximum number of packets N and the window-size-increment value, to generate attack traffic that caused a target server to consume 50% CPU usage. The other parameter, the number of bots required to generate attack traffic that consumes 100% CPU usage of a target server, was observed by varying the number of bots at the client side. This is illustrated in Figure 5-7.

To find the maximum number of packets, the window-size-increment value,

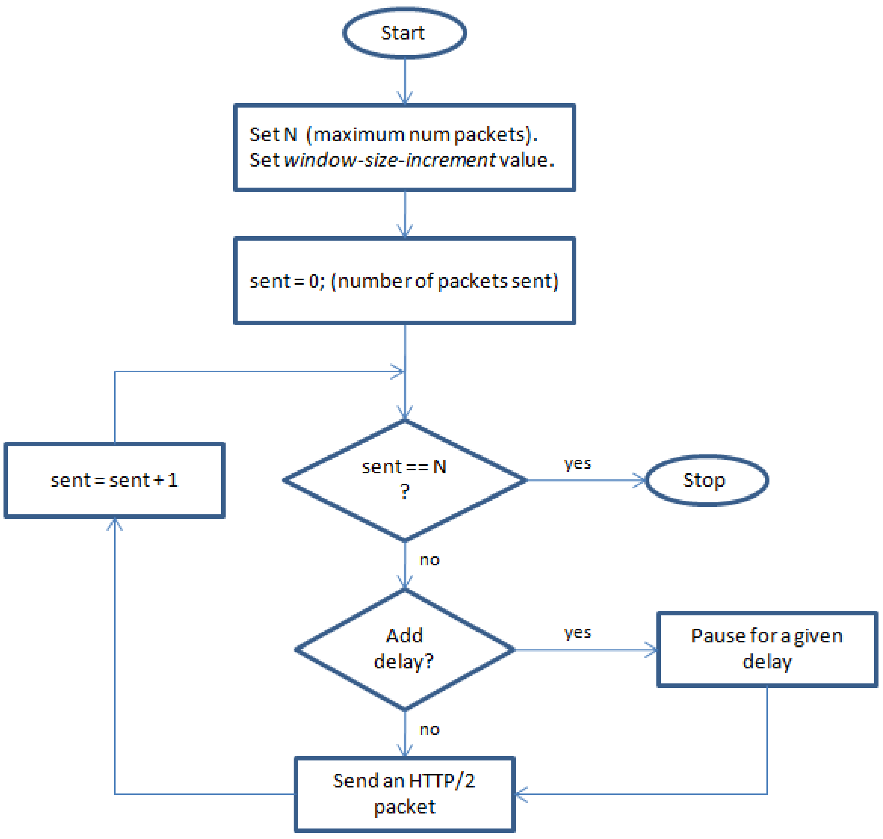

Start
Set N (maximum num packets).
Set *window-size-increment* value.
sent = 0; (number of packets sent)
sent = sent + 1
sent == N
?
yes
Stop
no
Add
delay?
yes
Pause for a given
delay
no
Send an HTTP/2
packet

Figure 5-6: The DDoS attack model

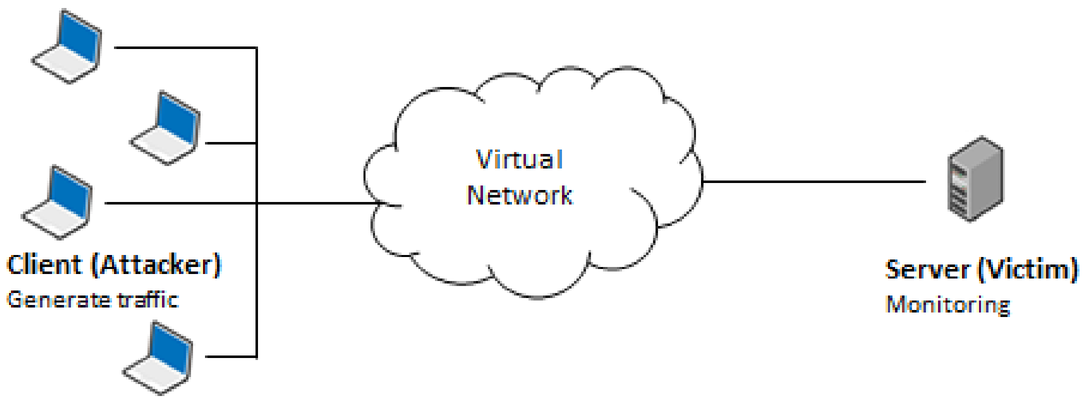


Figure 5-7: The traffic generation setup for DDoS attack

and the number of bots required in a DDoS attack, three investigations were conducted, as reported in this section. They extended the flooding-based attack model using window_update packets discussed in the previous section. Because this study is based on encrypted HTTP/2 communications, the number of window_update packets received at the server could not be directly inspected, as the window_update packets were part of the encrypted data. However, the maximum number of window_update packet communicated by a sender depends on the value of its payload, i.e. the window-size-increment. Hence, Investigation 1 aimed to find the window-size-increment value that represented 50% CPU consumption, and Investigation 2 deduced the number of window_update packets. Investigation 3 aimed to find the number of bots needed to form a successful DDoS attack that led to a 100% CPU consumption at the victim.

Investigation 1: Effective window-size-increment value

The aim of this investigation was to find the window-size-increment value that yields a 50% CPU utilization at the victim. The window-size-increment value affects the total number of window_update packets that a sender (in this case the attacking client) can send. In this investigation, a window-size-increment value d was sought until the victim machine began to show indications of excessive resource consumption through flooding by window_update packets.

The HTTP/2 standard (Belshe et al., May 2015) defines that the window-size-increment is represented as a 31-bit integer payload of a window_update frame (Figure 5-2). Hence, when the window size of a stream reaches its limit of $2^{31} - 1$, the client does not send any further window_update packet, even if the client interface (such as the packet generator used in this study) triggers a command to send a window_update packet. The implication of this behaviour is that the window-size-increment value of 1 would allow a client to send a flood of window_update packets within a much longer period of time than when window-size-increment value is set to $2^{31} - 1$. The latter would allow the client to send one or less window_update packets to a server. This is illustrated in Figure 5-8.

When a client sends a window_update packet, it updates its local window size value according to the window-size-increment value. In the example, the window size was pictured as having size equal to 3 bytes, represented by the 3 shaded boxes. If the client sent a window_update packet with window-size-value set to 1, the local window size value was incremented by 1 byte. Therefore, in the top figure, the local window size value became 4 bytes (represented by the 4 shaded boxes). A series of window_update packets sent by the client, with window-size-increment value set to 1, caused the local window size to increase by 1 byte until the local window size reached its maximum value. When the local window size reached its maximum value, no further window_update packets could be sent by the client.

Consequently, higher window-size-increment values imply fewer window_update packets to be sent. This is illustrated at the bottom of Figure 5-8. A series of window_update packets sent by the client, with window-size-increment value set to 2, caused the local window size to increase by 2 bytes until the local window size reached its maximum value. Similarly, when the local window size had reached its maximum value, no further window_update packet could be sent by the client. Comparing the top and the bottom figure, it can be seen that fewer number of window_update packets can be sent by the client when the window-size-increment is set to a higher value.

In this investigation, a client sent a flood of window_update packets to a server with window-size increment set to $d = 2^n$, where $0 \leq n \leq 31$. (The flood duration was to be observed in Investigation 2.) To implement a test framework, the test cases 2 and 3 from the previous section were employed. They are repeated here as follows:

- Test case 2: 2M window_update packets were transmitted on stream 0, with random window-size-increment.
- Test case 3: 2M window_update packets were transmitted on stream 0, with fixed window-size-increment.

In the above two investigations, test case 2 was run with random window-size-increment value that was set between 1 and d for each window_update packet, where d is a value to be sought in this investigation. Test case 3 was run with a

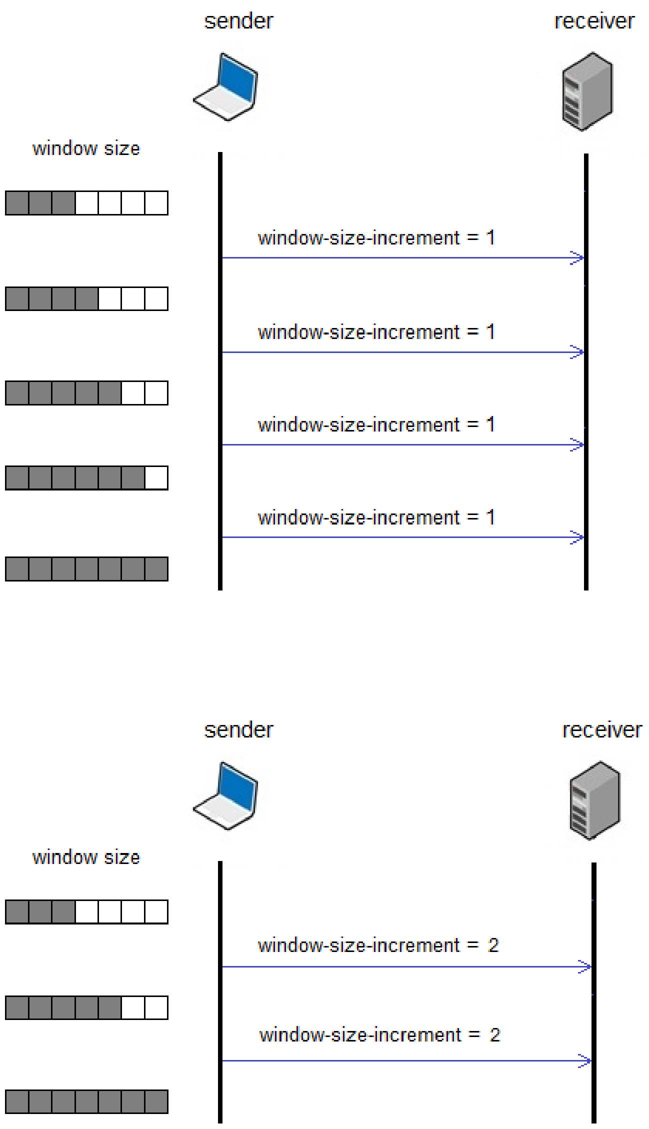

sender
receiver
window size
window-size-increment = 1
window-size-increment = 1
window-size-increment = 1
window-size-increment = 1
sender
receiver
window size
window-size-increment = 2
window-size-increment = 2

Figure 5-8: A higher window-size-increment value allows fewer number of subsequent window_update packets

fixed windows-size-increment value d. Using a monitoring tool *collectl*, the CPU consumption of the machine was monitored when flooded with varying window-size-increment values. The value found in this investigation would serve as a constant input for the next stage of investigation.

Investigation 2: Effective number of packets

The aim of this investigation was to find the number of packets required to flood an HTTP/2 service, deduced through observing time duration of attacks.

Using the window-size-increment value from the previous step, this investigation deduced the number of packets k received at the server end when the window_update packets was no longer sent by the client machine, i.e. when the local window size had reached its maximum value of $2^{31} - 1$. To find k, the client sent varying numbers of $x = 10^n$ window_update packets to the server, where $4 \leq n \leq 9$.

The study observed the *flood duration*, i.e. the time duration for which the CPU of the server showed near 50% consumption, when subjected to the varying values of x. When the local window size at the client side had not reached its maximum value, lower x values sent by the client caused a shorter flood duration observed at the server side. Consequently, higher x values caused longer flood duration.

When the local window size at the client had reached its maximum value of $2^{31} - 1$, no further window_update packets were sent by the client. Under this condition, higher x values did not show longer flood durations. Hence, x values above a certain threshold were no longer effective to flood a server. The threshold was the condition when higher x values did not yield longer flood durations. The value k was deduced to be below the value x when this condition was first observed.

The results found in this investigation were used as a parameter to launch a DDoS attack in the following investigation.

Investigation 3: DDoS attack packets

This investigation aimed to find the minimum number of attacking bots as part of a DDoS attack to indicate symptomatic depletion of resources on the victim machine represented through a delayed HTTP response from the victim. The independent variables were the *number of bots* and the *number of threads* (processes) per bot, while the variables to be observed were the *number of captured packets*, the *number of dropped packets*, and the *flood duration* (in seconds) at the victim's end.

Each bot in this investigation sent *stealthy* attack traffic, i.e. a flood of traffic that consumed 50% CPU usage of a victim. Hence, the window-size-increment value found in Investigation-1 and the maximum number of packets found in Investigation-2 was employed in this investigation. In addition, this investigation used multiple streams to simulate multiple clients under a DDoS attacks. Test cases 4 and 5 from the previous section were used as follows to send packets from an attacking client to a target server:

- Test case 4: 10K window_update packets with random window-size-increment were transmitted on each of 100 different stream IDs.
- Test case 5: 10K window_update packets with fix window-size-increment were transmitted on each of 100 different stream IDs.

5.2.2 Results and Discussion

Result of Investigation 1

As previously explained, creating a flood of window_update messages is not trivial: an open stream established between HTTP/2 clients and servers cannot be set up for a client to send an endless flood of window_update packets to a server. The client does not send the frame when the local window size value of a stream is high enough that it would exceed the maximum value, when supplied with a given window-size-increment value. Therefore, the victim's CPU would not be continuously over-utilised. Hence, the CPU consumption was studied in this investigation, when the client sent a flood of window_update packets to the server given a window-size-increment value of $d = 2^n$ with $0 \leq n \leq 31$.

The results obtained are presented as follows. When $n > 29$, there was no observable CPU consumption. A calculation can show that there were only $2^{31}/2^{29} = 2^2$, or at most 4 frame packets sent to the server assuming a very small initial window size. On the contrary, the CPU consumption was 100% at all other times when $n = 0$; and after 1 hour and 7 minutes into simulation, the CPU consumption returned back to normal with no further packets received at the server end. This finding confirmed the above explanation that the window_update packets were prevented from arrival at the server after the window size of the stream reached its maximum value of $2^{31} - 1$. This investigation sought a window-size-increment value d that consumed around 50% CPU, and through further experiments, this value was found to be $d = 2^{14}$, or 16,384.

Result of Investigation 2

Given the window-size-increment value of 16,384 from the previous step, this investigation sought to find the minimum number of packets, and to observe the time duration of a flood of window_update packets which can be sent by an attacking client to a target victim on a given stream. These were measured against the methods described for test cases 2 and 3, i.e. sending a flood of window_update packets with a fixed window-size-increment value of 16,384 (as in test case 2), and those with a random window-size-increment value between 1 and 16,384 (as in test case 3).

The results of investigation 3 are shown in Table 5.2. The results indicated that varying the number of window_update packets sent did not cause a large variation of duration for which the flood packets were observed at the victim.

The number of window_update packets, or k, was deduced to lie between 100K and 1M since the flood duration did not show significant fluctuations when the number of packets sent were above 1M. The average flood duration for test case 3, when $100K < k \leq 1M$ is (30+54+34+36)/4 = 38.5 seconds. The number of window_update packets can be confirmed with the following calculations. The HTTP/2 standard stated that the initial window size is 65,535 (Belshe et al., May 2015, p.23). Therefore, when the window-size-increment value was set to 16,384

Table 5.2: Duration the WINDOW_UPDATE frame was sent

Test case	Number of packets	Flood duration (ms)
2	10K	4
	100K	23
	1M	64
	10M	56
	100M	64
	1B	70
3	10K	5
	100K	25
	1M	30
	10M	54
	100M	34
	1B	36

or 2^{14}, the value k was found to be $(2^{31} - 1 - 65,535)/2^{14} = 131,068$. Attacking clients that sent higher number than 131,068 packets to a target server would observe similar flood duration of 38.5 seconds.

The implication of this observation was that a packet generator should be designed to send an effective number $131,068$ of window_update packets. An attacking client designed to send 1M packet should distribute the packets into several stream IDs. For example, 10K window_update into 100 stream IDs. The distribution of 1M packet into 100 stream IDs was used as a parameter for the following investigation.

Result of Investigation 3

This investigation sought to find the number of attacking bots it took for an HTTP/2 service to reveal a symptom of resource depletion when one attacking bot alone did not indicate suspect behaviour. From the previous investigations, it was found that the victim's CPU was only utilized around 50% for about 64 ms (for test case 2) or 30 ms (for test case 3). Any normal computing activities (e.g. disk writing) could show such indications. Hence, the proposed methods produced *stealthy* attack traffic which could bypass a hypothetical intrusion-detection-system that triggered an alert when a client caused a machine to reach near 100% CPU consumption. The values found in the first and second investigation served as the ground to set up a stealthy attacking bot. These are 16,384 window-size-increment as the window_update payload value, and a maximum number of 131,068 window_update packets on a stream.

To simulate hundreds of clients as seen in a flash crowd (or bots as in a DDoS attack), the methods described in test cases 4 and 5 were employed. Each of these test cases required sending of 10K window_update packets in each 100 different stream IDs successively, that in this case represented 100 clients connecting to a server in succession (and the flood duration was part of the observed variable). In addition, a delay t_d between individual stream IDs was inserted, where 10 ms $\leq t_d \leq$ 200 ms. That is, an attacking client sent 10K window_update packets on one stream without any delay t_d in between the packets, and a delay was added before the client sent other 10K window_update packets on another stream. This investigation observed different delay values inserted between stream IDs at the client side, and a delay value was noted as soon as packet drops were reported by TShark at the server side. This delay value represented the smallest delay that can be added between stream IDs until the server showed packet drops. The delay value for test case 4 was found to be $t_d = 45$ ms; values smaller than this number showed that some packets were always dropped, while values exceeding this number showed no packets dropped. The delay value for test case 5 was found to be $t_d = 100$ ms.

In this investigation, both threads and virtual machines were used to represent the number of attacking bots. As discussed above, each bot sent stealthy traffic to a target server. The traffic was designed to consume 50% server CPU for duration of 38.5 ms; and consumed no more than 2 MB server memory. The study varied the number of virtual-machine clients and the number of threads per bot to represent a DDoS attack, each thread sending 100 packets sequentially (test cases 4 and 5). The results are shown in Table 5.3 and 5.4.

It can be seen that a higher number of packets were dropped when the number of attacking bots were increased. This was to confirm that the computing resources of the victim were consumed using the stealthy traffic generated; and subsequently, the victim was compromised.

Table 5.3: DDoS using test case 4

Num bots	Num threads/bot	Captured	Dropped	Flood duration (sec)
1	1	144775	298	30.11
2	1	293879	8199	29.90
3	1	353986	22786	29.59
4	1	345743	83780	29.41
1	2	231041	2575	29.52
2	2	330567	16305	29.35
3	2	362003	76313	29.77
4	2	344580	80256	29.69

Table 5.4: DDoS using test case 5

Num bots	Num threads/bot	Captured	Dropped	Flood duration (sec)
1	1	263522	391	29.73
2	1	410097	1379	30.27
3	1	513076	3984	30.14
4	1	532612	6560	30.33
1	2	266191	132	30.21
2	2	384540	1460	29.96
3	2	531913	4625	30.27
4	2	530751	7257	30.44

To witness the significance of the observed packets dropped, the study manually sent an HTTP/2 Request every second and measured the time it took for the server to respond to a `hello.htm` page, and subsequently closed the conversation. When 4 virtual-machine clients (bots) with 2 threads/bot were employed, up to 40 ms of delay for test case 4 was observed, and up to 97 ms delay for test case 5. This delay was higher than a 0.3 ms delay that was observed when the attack traffic was not present. As real web applications serve more than just displaying a `hello.htm` message, the stealthy offending traffic proposed in this section can translate to a recipe that fully incapacitates an HTTP/2 service upon introduction of variant client behaviour types, based on diverse web page access patterns.

Comparison to flash-crowd traffic

In this part of the study, one of the above attack traffic patterns was analysed using several machine learning techniques. Specifically, the study utilized the

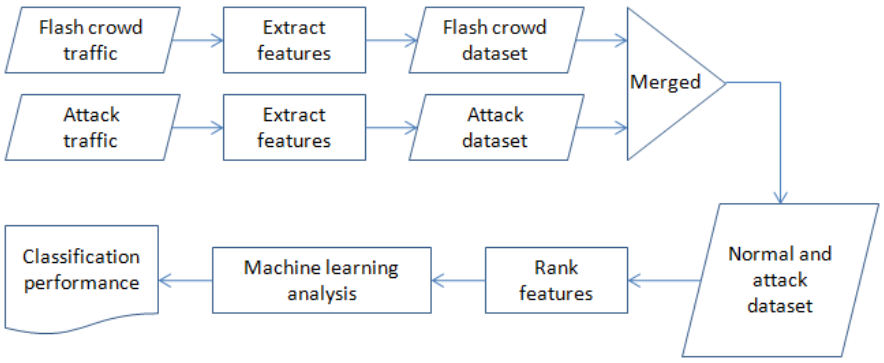


Figure 5-9: Process for normal and attack traffic classification

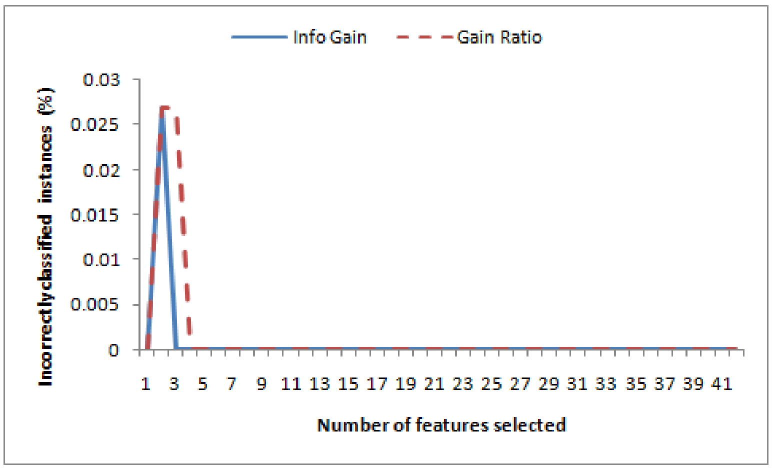
attack traffic shown in the last row of Table 5.4, i.e. using test case 5 with 4 bots and 2 threads/bot. The purpose was to understand how the attack traffic can be distinguished from flash-crowd traffic.

Figure 5-9 describes the process for classifying attack and flash-crowd traffic. A set of features from both the attack and flash-crowd traffic was extracted (as explain in Section 3.1.3). The feature extraction yielded a dataset with 124 attack traffic instances. The dataset was merged with the flash-crowd dataset which consisted of 3600 instances (explained in Section 4.4.2), yielding a dataset with a total of 3724 instances. The features from the dataset were ranked using two features selection techniques, Information Gain and Gain Ratio (as explained in Section 3.1.4). The study applied four machine learning techniques, i.e. Naïve Bayes and Decision Tree J48, JRip and Support Vector Machines, to classify the attack traffic when subjected to the flash-crowd traffic that was described in Section 4.4.2. The parameter values of these machine learning techniques are given in Appendix A. The result of the feature ranking is shown in Table 5.5 and the result of the machine learning classification is shown in Figure 5-10 to 5-13.

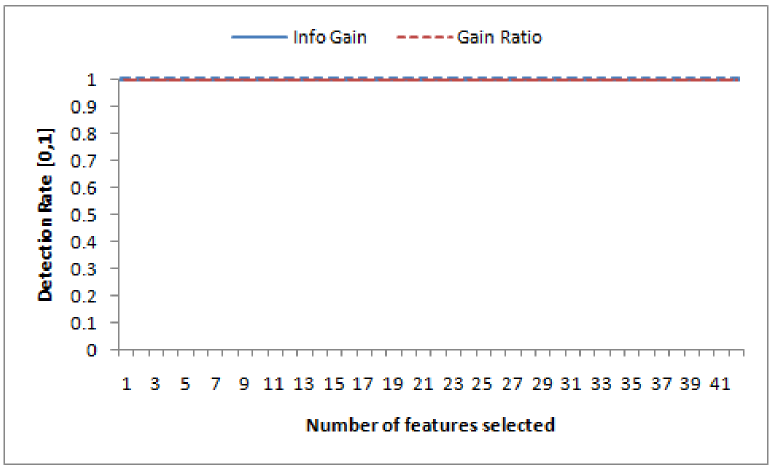
The incorrectly classified instances graphs (the (a) graphs of Figure 5-10 to 5-13) showed that the attack traffic could be distinguished from flash-crowd traffic using these machine learning techniques. Naïve Bayes and Decision Tree perfectly classified the two classes when at least 4 most relevant features were employed (Figure 5-10.a and 5-11.a). Although Naïve Bayes produced 0.027% incorrectly classified instances when 2 features were used, it yielded a perfect classification

Table 5.5: Ranked features for the distributed attack traffic

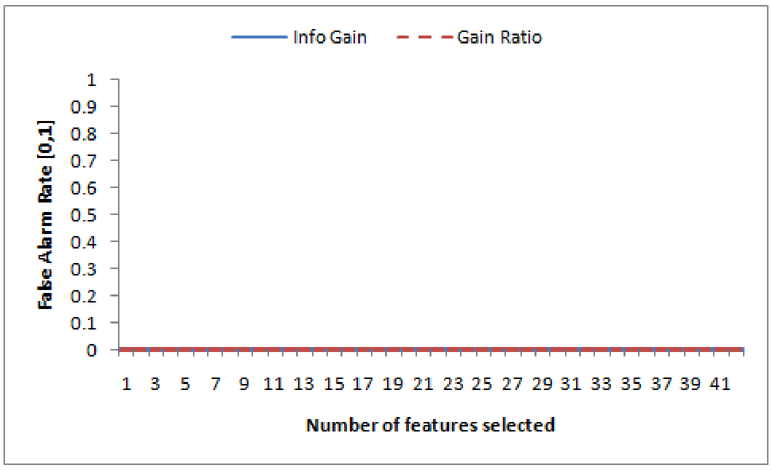
rank	Information Gain		Gain Ratio	
	feature #	feature name	feature #	feature name
1	1	count_app	1	count_app
2	9	size_ack	8	count_ack
3	7	size_syn	9	size_ack
4	6	count_syn	7	size_syn
5	29	size_tlsHello	29	size_tlsHello
6	28	count_tlsHello	6	count_syn
7	2	size_app	28	count_tlsHello
8	8	count_ack	2	size_app
9	18	count_rstAck	39	size_encAlert
10	19	size_rstAck	38	count_encAlert
11	38	count_encAlert	13	count_rst
12	39	size_encAlert	14	size_rst
13	34	size_tlsKey	19	size_rstAck
14	14	size_rst	18	count_rstAck
15	13	count_rst	23	count_finAck
16	23	count_finAck	24	size_finAck
17	24	size_finAck	10	lapse_ack_min
18	33	count_tlsKey	32	lapse_tlsHello_max
19	22	lapse_rstAck_max	31	lapse_tlsHello_ave
20	21	lapse_rstAck_ave	30	lapse_tlsHello_min
21	12	lapse_ack_max	35	lapse_tlsKey_min
22	11	lapse_ack_ave	36	lapse_tlsKey_ave
23	27	lapse_finAck_max	37	lapse_tlsKey_max
24	26	lapse_finAck_ave	4	lapse_app_ave
25	4	lapse_app_ave	3	lapse_app_min
26	37	lapse_tlsKey_max	11	lapse_ack_ave
27	36	lapse_tlsKey_ave	33	count_tlsKey
28	17	lapse_rst_max	34	size_tlsKey
29	5	lapse_app_max	12	lapse_ack_max
30	42	lapse_encAlert_max	5	lapse_app_max
31	10	lapse_ack_min	22	lapse_rstAck_max
32	31	lapse_tlsHello_ave	21	lapse_rstAck_ave
33	32	lapse_tlsHello_max	27	lapse_finAck_max
34	16	lapse_rst_ave	26	lapse_finAck_ave
35	35	lapse_tlsKey_min	42	lapse_encAlert_max
36	30	lapse_tlsHello_min	17	lapse_rst_max
37	3	lapse_app_min	16	lapse_rst_ave
38	40	lapse_encAlert_min	40	lapse_encAlert_min
39	15	lapse_rst_min	15	lapse_rst_min
40	20	lapse_rstAck_min	41	lapse_encAlert_ave
41	41	lapse_encAlert_ave	20	lapse_rstAck_min
42	25	lapse_finAck_min	25	lapse_finAck_min



(a) Incorrectly classified instances (%)

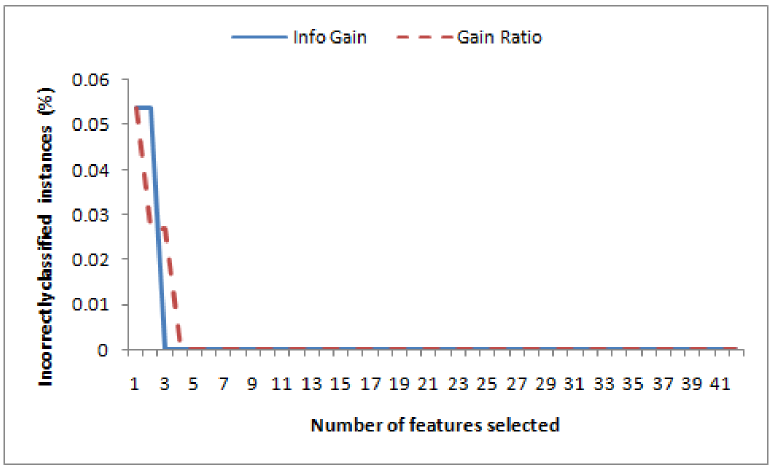


(b) Detection Rate

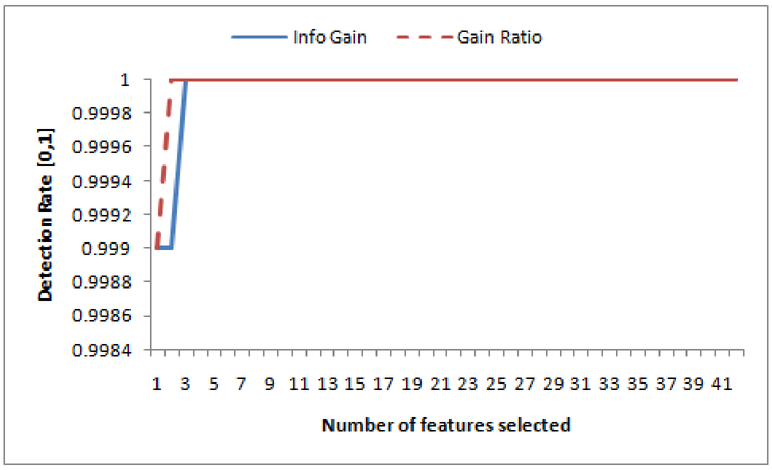


(c) False Alarm Rate

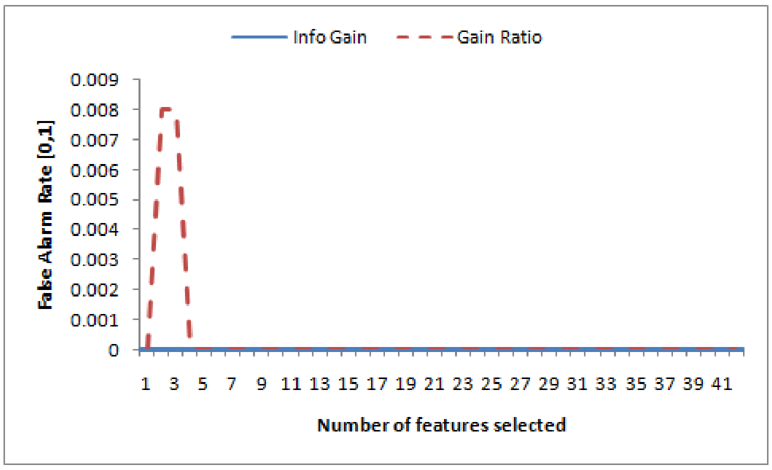
Figure 5-10: Distributed Attack Performance with Naive Bayes classification



(a) Incorrectly classified instances (%)

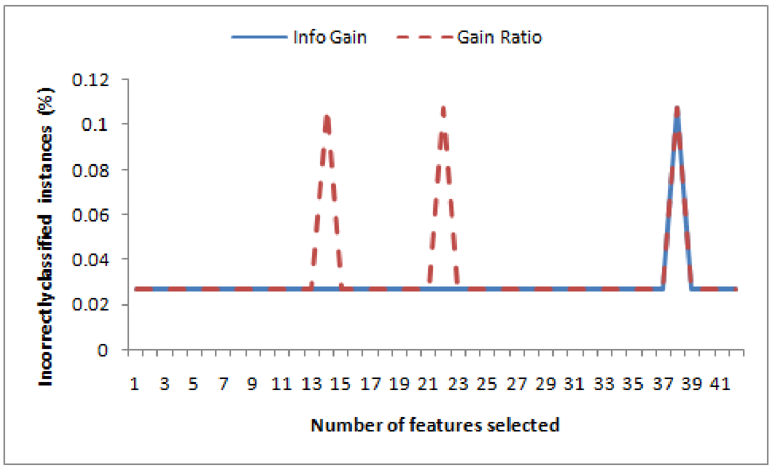


(b) Detection Rate

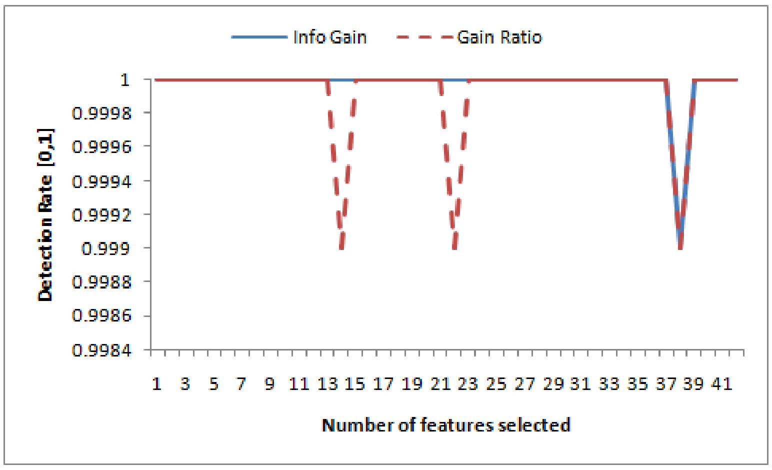


(c) False Alarm Rate

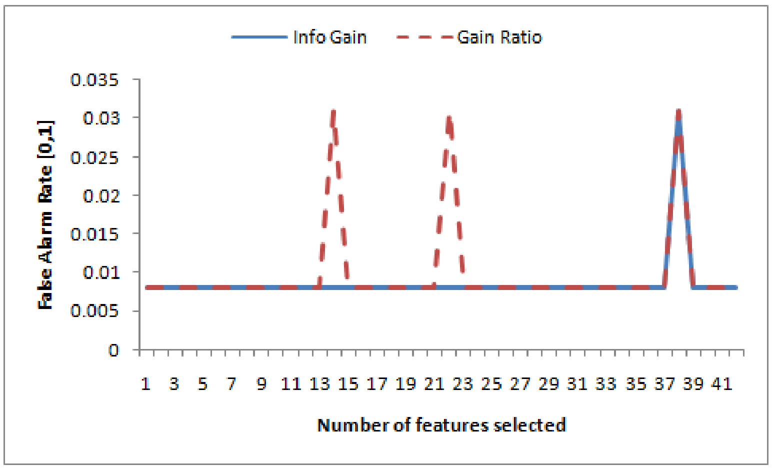
Figure 5-11: Distributed Attack Performance with Decision Tree classification



(a) Incorrectly classified instances (%)

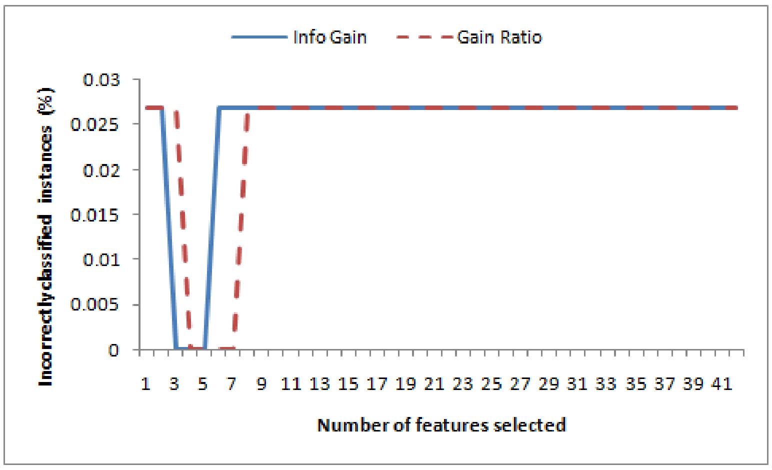


(b) Detection Rate

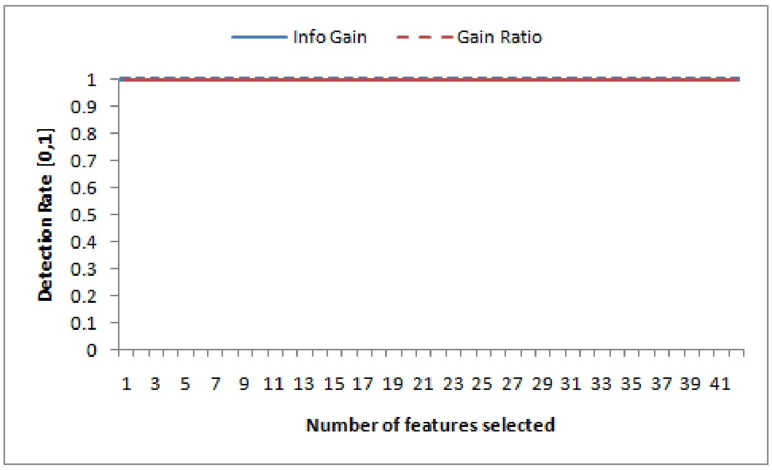


(c) False Alarm Rate

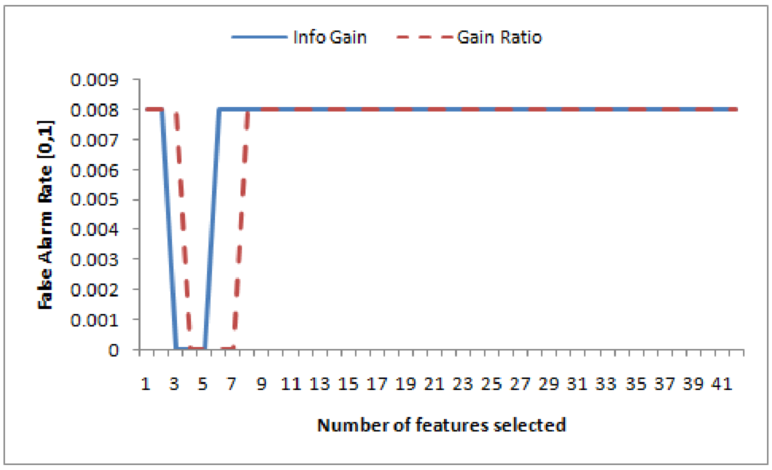
Figure 5-12: Distributed Attack Performance with JRip classification



(a) Incorrectly classified instances (%)



(b) Detection Rate



(c) False Alarm Rate

Figure 5-13: Distributed Attack Performance with Support Vector Machine classification

when only 1 feature was used, which is the count_app feature. This showed that Naïve Bayes was able to classify the DDoS traffic from flash-crowd through examining the amount of Application Data. While Decision Tree produced 0.027% incorrectly classified instances when using 2 Information Gain-ranked features or 3 Gain Ratio-ranked features, it yielded 0.054% incorrectly classified instances when only 1 feature was used, the i.e. the count_app. Both Naïve Bayes and Decision Tree yielded 0% incorrectly classified instances when more than 4 features were employed, regardless of the ranking technique used.

The attack traffic was not perfectly identified when JRip was used, as it returned some incorrectly classified instances regardless of the number of features selected (Figure 5-12.a). JRip yielded 0.027% incorrectly classified instances at best, while yielded higher incorrectly classified instances (0.107%) when using some feature sets, i.e. 38 Information Gain-ranked features; and 14, 22, and 38 Gain Ratio-ranked features. With JRip, no feature sets yielded a perfect classification.

When analysed using Support Vector Machines, the attack traffic could be perfectly identified when 3 to 5 Information Gain-ranked features were used, or when 4 to 7 Gain Ratio-ranked features were employed (Figure 5-13.a). Support Vector Machine yielded 0.027% incorrectly classified instances when feature sets outside the above range were used. That is, higher than 5 and lower than 3 Information Gain-ranked features; and higher than 7 and lower than 4 Gain Ratio-ranked features.

The Detection Rate (the (b) graphs of Figure 5-10 to 5-13) and the False Alarm Rate (the (c) graphs of Figure 5-10 to 5-13) of the machine learning classification results also supported the above analysis. All machine learning techniques were able to reach a perfect Detection Rate of 1 and a clean False Alarm Rate of 0, mostly when the number of features selected was higher than 3. An exception was shown with JRip analysis which did not reach a zero False Alarm Rate when analysed with all series of ranked features.

The detailed numbers for the Detection Rate are as follows. Naïve Bayes (Figure 5-10.b) and Support Vector Machine (Figure 5-13.b) yielded Detection

Rate of 1 in all number of features selected . Decision Tree showed similar results, but the Detection Rate (Figure 5-11.b) degraded to 0.999 when 2 Information Gain-ranked features were used, and when 1 Gain Ratio-ranked feature was used. JRip yielded Detection Rate of mostly 1 (Figure 5-12.b). It degraded to 0.999 when 38 Information Gain-ranked features were used; or when 14, 22, or 38 Gain Ratio-ranked features were used.

The detailed numbers for the False Alarm Rate are as follows. Naïve Bayes (Figure 5-11.c) yielded False Alarm Rate of 0 in all number of features selected. Decision Tree (Figure 5-11.c) showed similar result, but it degraded to 0.008 when 2 - 3 features were selected with Gain Ratio. JRip only reached 0.008 False Alarm Rate when using almost all number of features selected. It degraded to 0.031 when 38 Information Gain-ranked features were used; or when 14, 22, or 38 Gain Ratio-ranked features were used. Support Vector Machine (Figure 5-13.c) was able to reach 0 False Alarm Rate; it degraded to 0.008 when 3 to 5 Information Gain-ranked features were used, or when 4 to 7 Gain Ratio-ranked features were employed.

The analysis in this section showed that machine learning techniques were able to detect the DDoS model simulated. Naïve Bayes, Decision Tree, and Support Vector Machines were able to distinguish DDoS from normal traffic when using more than 4 features out of the 42 features proposed in this study.

5.2.3 Conclusion

This section presented a DDoS attack model where each attacking bot sent a large volume of HTTP/2 traffic to a victim machine in a fixed time frame. From the results obtained, it was noted that the protocol itself did not restrict the intensity of traffic generated, and therefore, auxiliary mechanisms ought to be deployed for identifying the volumes and patterns of network traffic communicated between a client and server machine.

Three varying investigations were conducted to analyse the behaviour of a victim machine when subject to large HTTP/2 traffic volume through an established connection stream. The first investigation found that a flood of window_update packets sent by an attacking client caused a target server to show 50% CPU consumption when the window-size-increment value was set to 16,384. The second investigation showed that the effective number of window_update packets that the client sent was 131,068; a higher number than this did not yield longer flood duration. The third investigation demonstrated that 4 attacking bots were able to cause a target server to show 100% CPU consumption. The traffic flood that each of these bots sent caused the target server to consume only 50% CPU consumption. Therefore, the proposed DDoS attack model could bypass a hypothetical intrusion-detection system that monitored its resource consumption such as CPU consumed.

However, the attack traffic could be distinguished from flash-crowd traffic using various machine learning techniques. Naïve Bayes, Decision Tree, and Support Vector Machines were able to distinguish DDoS from normal traffic when more than 4 out of 42 features proposed in this study were employed. Machine learning techniques performed well when they were used to detect the proposed HTTP/2 DDoS attack traffic.

The next chapter aims to present a stealthier attack model than the DDoS model presented here. It was observed that a stealthier model could be achieved through a better understanding of flash-crowd traffic characteristics and mimicking the feature values.

Chapter 6

Stealthy Attack Modelling and Analysis

The study carried out and reported in Chapter 4 presented the characteristics of legitimate, HTTP/2 flash-crowd traffic and introduced an HTTP/2 attack traffic model (Chapter 5). The attack traffic based on the presented model was able to bypass hypothetical intrusion-detection systems that monitored CPU consumption caused by process execution remote clients to service and also successfully incapacitate a target machine when a number of attacking clients collectively launched a DDoS attack. The attack traffic characteristics presented previously could be distinguished from flash-crowd traffic through machine learning-based analysis. This chapter extends the attack model introduced in the previous chapter to demonstrate how HTTP/2 attack traffic which is stealthy in nature can degrade the performance of machine learning analysis. It aims to show how HTTP/2 attack traffic could cause machine learning techniques to incorrectly classify traffic instances, leading to degraded Detection Rate and False Alarm Rate.

The chapter introduces two models, namely, *Stealthy Attack 1* and *Stealthy Attack 2* and shows how the generated traffic could be distinguished from flash-crowd traffic based on machine learning. The analysis applied four machine learning techniques, i.e. Naïve Bayes, Decision Tree, JRip, and Support Vector Machines. In addition, the Self Organizing Map was applied to visualize the clusters of similar traffic groups. Furthermore this chapter compared the HTTP/2 attack traffic characteristics when the analysis was built upon features traditionally employed in the literature for HTTP/1.1 DoS attack analysis.

6.1 Stealthy Attack 1

Stealthy traffic causes machine learning analysis to yield more incorrectly classified instances. This section introduces a variant of the attack model presented in Chapter 5 to generate stealthy attack traffic, as opposed to flash-crowd traffic. The study aimed to model attacks whose traffic continually consumed the victim's computing resource, yet caused machine learning techniques to incorrectly classify some traffic instances. It contrasted the traffic characteristics produced from the investigation in this section to that of the DDoS traffic produced and reported in Section 5.2.

In addition, the study tried to find the least possible number of attacking clients to successfully incapacitate a victim machine. In this chapter, attacking clients are named as 'bots'. There were two motivations for this study. First, it would aid in quantifying malicious flood traffic launched from one generic computer (e.g. from an attacker's place of residence) towards a victim. Second, it quantifies the number of remote computers that would be required to be compromised to run bots, when an attacker intends to launch a DDoS attack against a victim.

6.1.1 Attack Model and Scenarios

The study proposed to *camouflage* attack traffic with features of normal traffic. The stealthy attack model presented herewith comprises two groups of bots to camouflage attacks. One group attempts to exactly mimic the flash-crowd traffic features, and another does the generation the offending traffic derived from the current understanding (Chapter 5) (Adi, Baig, Lam, & Hingston, 2015; Adi, Baig, Hingston, & Lam, 2016), i.e.: a flood of 131,068 window_update packets with window-size-increment payload set to 16,384 sent by an attacking bot for 38.5 ms caused a target server to consume 50% CPU consumption; and 4 attacking bots, where each bot sent such a flood caused a target server to consume 100% CPU resources. The mimicking bots are labelled as the *mime group*, and the attacking ones are labelled as the *offending group*.

The proposed implementation was to have the number of TCP connections of the attack traffic mimic that of the flash-crowd traffic closely. The feature that represented this characteristic was the count_syn feature, which is defined as the volume of SYN packets observed in a one-second traffic instance. Hence, the mime group attempted to mimic the number of SYN packets/sec of flash-crowd traffic generated by the bots targeting a victim, while the offending group generated attack traffic towards the victim. The attack traffic aimed to continually consume 100% CPU usage of the victim. The study controlled the amount of traffic generated by each group, until instances of 100% CPU consumption were observed at the victim machine.

To control the amount of traffic, two independent variables were added to the packet generator that constructed both the mime and offending bot anatomy. First, instead of indefinitely transmitting a flood of window_update packets as attempted in the previous sections, the bot sent intermittent floods. A flood was defined as 131,068 (or 131K) window_update packets in 38.5 ms. Here, *stealthy factor* is defined as the outcome of rolling an x-sided dice, where a random integer was generated between 1 and x. A flood from a bot towards a target server was launched when x equalled to 1. When $x \neq 1$, the bot sent only 1 HTTP/2 Request and then disconnected the TCP connection. Higher stealthy factor numbers imply smaller chances to have an outcome value of 1 out of a given stealthy factor x; hence, the higher the stealthy factor the less frequently for launching a flood. This mechanism created intermittent floods from the bots towards the victim rather than continuous attack traffic.

Second, a *delay* variable was introduced. Instead of pausing for 100 ms between streams as previously proposed (Section 5.2.2), the bots disconnected the

TCP connection and reconnected after a given delay. This variable controlled the flow of SYN packets/sec from each bot towards the victim. As illustrated on page 19, a SYN packet initiates a client-server TCP connection. Hence, the bots created SYN packets with a fixed delay between connections. The investigation in this section searched for a delay value between connections initiated by the mime group and the offending group. Larger delay values aided the mime group to mimic the number of SYN packets/second of flash-crowd traffic. However, larger delay values caused the offending group to send less attack traffic flow, causing the CPU consumption of the victim to slide from 100%.

In bot-induced DDoS attacks, the higher the number of bots, the closer the traffic pattern that they generated is to flash-crowd traffic (S. Yu, Guo, & Stojmenovic, 2012). This is reasonable, since both flash-crowd as well as attack traffic floods are generated from Internet-connected machines. In this study, a minimum number of bots were observed when the traffic that the bots generated caused a target machine to continually show 100% CPU consumption. An attacking bot anatomy is modelled as being built upon five parameters:

- *number of threads*: the number of simultaneous processes runs on a bot machine, where each process can independently initiate a TCP connection with a remote machine, i.e. a target server.
- *number of streams*: the number of stream IDs in one TCP connection, where each stream ID generates a traffic flood towards a target machine.
- *number of window_update*: the number of window_update packets in each stream.
- *stealthy factor*: the frequency at which a TCP connection is used to send a flood of packets against a target machine, equals to $1/stealthyfactor$.
- *delay*: a time delay between successive TCP connections.

This section proposed one implementation of a stealthy attack model which is shown in Table 6.1. The table shows that the proposed stealthy attack traffic

Table 6.1: Stealthy Attack-1 model

	Bot 1	Bot 2
Number of threads	1	1
Number of streams	1	1
Number of window_update	131K	131K
Stealthy factor	50	500
Delay between connections	11 ms	11 ms

could be generated by simply two bots, with one bot representing the mime group and the other the offending group. One virtual machine was used to run each bot. Bot 1 acts as the offending group that sends 131K window_update packets as attack traffic towards the victim, the value of which was obtained from Section 5.2.2 (page 138). The attack traffic is sent periodically with a stealthy factor equals to 50. This means the attack traffic is sent when a random variable x yields 1 of 50 chances, otherwise the bot sent 1 HTTP/2 Request and disconnected its TCP connection with the victim. Bot 2 sends less frequent attack traffic, as it is assigned a stealthy factor 500. This number is ten times higher than Bot 1, to maintain the desired number of TCP connections in its attempt to mimic flash-crowd traffic.

The traffic these bots generated was extracted using the feature extraction method described in Section 3.1.3, yielding an attack traffic dataset with 549 instances. The attack traffic dataset was merged with the flash-crowd dataset which consisting of 3600 instances (explained in Section 4.4.2), yielding a dataset with 4149 instances. The features from the dataset were ranked using Information Gain and Gain Ratio techniques, upon which the dataset comprising flash-crowd and attack traffic features was classified using four machine learning techniques, i.e. Naïve Bayes, Decision Tree, JRip, and Support Vector Machines. The machine learning classifications were executed on an Intel Core-i3 machine with a 2 GB RAM. Three machine learning techniques, i.e. Naïve Bayes, Decision Tree, and JRip, took less than 1 minute to yield classification results. Support Vector Machines took more than 2 minutes to yield classification results. The parameter values of these machine learning techniques are given in Appendix A. The

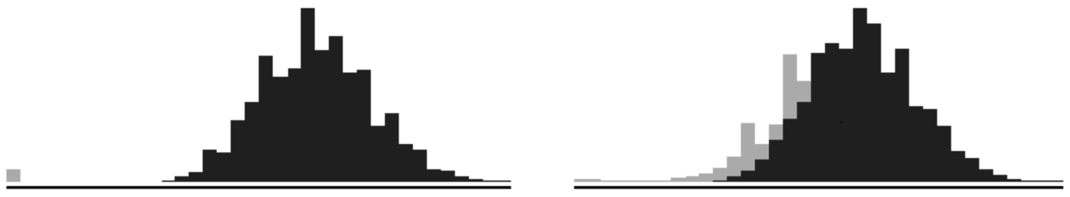


Figure 6-1: Visualization of count_syn feature values. Left: DDoS attack. Right: Stealthy Attack-1.

following subsection presents the results of this investigation.

6.1.2 Results and Analysis

This section compares the results from observing the TCP connection flow for DDoS and Stealthy Attack-1 traffic. It shows the feature ranking results as well as the results of machine learning classification of Stealthy Attack-1 traffic.

Comparison of the number of TCP connections

The proposed implementation method of camouflaging attack traffic was intended to mimic the number of TCP connections from flash-crowd traffic. The number of TCP connections in the dataset comprising flash-crowd and attack traffic was represented by the count_syn feature. Figure 6-1 presents how the count_syn feature values of DDoS and Stealthy Attack-1 compare to the count_syn feature values of flash-crowd. The figure was obtained from Weka. The X-axis shows the number of SYN packets/sec, and the Y-axis shows the number of instances, or the tally for the X values. The left figure illustrates the count_syn features of the DDoS attack, particularly from the last row of Table 5.4, i.e. using test case 5 with 4 bots and 2 threads/bot. The right figure illustrates the count_syn values for the Stealthy Attack-1 as proposed in this section. There are similarities as well as contrasting differences between the two figures.

In both left and right figures, the black graph shows the distribution of the flash-crowd count_syn feature values, and the grey one shows that of attack traffic. Hence, it could be seen that in both figures, the flash-crowd traffic produces

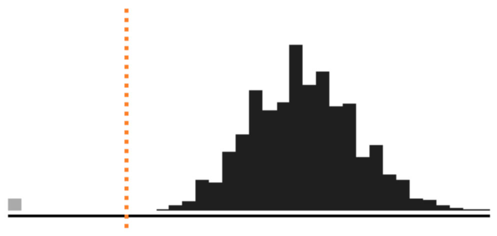


Figure 6-2: A threshold line can split DDoS and flash-crowd traffic

higher flow of SYN packets/sec than the DDoS attack traffic, as the values of both black graphs are shown on the right hand side of their respective grey graphs. However, the difference is that the DDoS traffic (the left figure) can be visually classified, since the attack traffic (grey) is shown on the far left of the flash-crowd traffic (black). It is unambiguous to choose a threshold value such as a vertical dotted-line to split the two colours as illustrated in Figure 6-2. In contrast, the Stealthy Attack-1 traffic (Figure 6-1 right) camouflaged the attack traffic as the values can be seen to overlap with those of the flash-crowd traffic.

Hence, the two proposed bots in this section (Table 6.1) were able to camouflage the number of SYN packets/sec and operate as flash crowd, remaining undetected. This traffic was stealthier than the DDoS traffic of Section 5.2.

Feature ranking and performance analysis

Table 6.2 lists the results of ranking the traffic features when subjected to Stealthy Attack-1 and the flash-crowd dataset. The more relevant features are ranked closer to the top of the list. The order of the rank number is used to select the features when the processed traffic is analysed using machine learning techniques. For example, when the number of features selected is set to 1, then the techniques used the top-ranked feature to split the two classes. When the number of features selected is n, a set of features ranked $\{1 \ldots n\}$ is employed. This is repeated until the total number of 42 features (as shown in Table 3.6 on page 85) was selected. Figure 6-3 to 6-6 present the performance analysis when the different sets of features are employed by each of the machine learning techniques.

Table 6.2: Ranked features for Stealthy Attack-1traffic

rank	Information Gain		Gain Ratio	
	feature #	feature name	feature #	feature name
1	19	size_rstAck	19	size_rstAck
2	18	count_rstAck	18	count_rstAck
3	1	count_app	2	size_app
4	34	size_tlsKey	34	size_tlsKey
5	29	size_tlsHello	1	count_app
6	33	count_tlsKey	33	count_tlsKey
7	2	size_app	6	count_syn
8	6	count_syn	7	size_syn
9	7	size_syn	28	count_tlsHello
10	28	count_tlsHello	39	size_encAlert
11	38	count_encAlert	38	count_encAlert
12	39	size_encAlert	24	size_finAck
13	24	size_finAck	23	count_finAck
14	23	count_finAck	27	lapse_finAck_max
15	8	count_ack	22	lapse_rstAck_max
16	14	size_rst	26	lapse_finAck_ave
17	13	count_rst	21	lapse_rstAck_ave
18	9	size_ack	29	size_tlsHello
19	27	lapse_finAck_max	17	lapse_rst_max
20	22	lapse_rstAck_max	16	lapse_rst_ave
21	26	lapse_finAck_ave	14	size_rst
22	21	lapse_rstAck_ave	13	count_rst
23	17	lapse_rst_max	42	lapse_encAlert_max
24	16	lapse_rst_ave	8	count_ack
25	42	lapse_encAlert_max	41	lapse_encAlert_ave
26	41	lapse_encAlert_ave	9	size_ack
27	20	lapse_rstAck_min	20	lapse_rstAck_min
28	25	lapse_finAck_min	25	lapse_finAck_min
29	5	lapse_app_max	5	lapse_app_max
30	15	lapse_rst_min	15	lapse_rst_min
31	12	lapse_ack_max	12	lapse_ack_max
32	4	lapse_app_ave	4	lapse_app_ave
33	37	lapse_tlsKey_max	37	lapse_tlsKey_max
34	11	lapse_ack_ave	11	lapse_ack_ave
35	32	lapse_tlsHello_max	32	lapse_tlsHello_max
36	3	lapse_app_min	36	lapse_tlsKey_ave
37	40	lapse_encAlert_min	3	lapse_app_min
38	35	lapse_tlsKey_min	40	lapse_encAlert_min
39	36	lapse_tlsKey_ave	31	lapse_tlsHello_ave
40	30	lapse_tlsHello_min	35	lapse_tlsKey_min
41	31	lapse_tlsHello_ave	30	lapse_tlsHello_min
42	10	lapse_ack_min	10	lapse_ack_min

The (a) graphs from Figure 6-3 to 6-6 reveal the incorrectly classified instances when Stealthy Attack-1 were analysed against flash-crowd traffic. These graphs show more sets of selected features that yield incorrectly classified instances, than the DDoS graphs previously presented (the (a) graphs from Figure 5-10 to 5-13). In other words, the graphs show more x values that yield above-zero y values, signifying that the Stealthy Attack-1 traffic is stealthier than the DDoS traffic. It could be seen that Naïve Bayes (Figure 6-3.a) did not classify the traffic unimpaired, as the incorrectly classified instances were above 0. Similarly, Decision Tree (Figure 6-4.a) could distinguish the two traffic types only in special cases when certain sets of features were selected using Gain Ratio. Contrary to the DDoS traffic, the Stealthy Attack-1 traffic led to more incorrectly classified instances when analysed using the two machine learning techniques, i.e. Naïve Bayes and Decision Tree. This can be seen where the number of features selected are 5 or more, Stealthy Attack-1 analysis with Naïve Bayes (Figure 6-3.a) and Decision Tree (Figure 6-4.a) show more incorrectly classified instances than DDoS analysis with Naïve Bayes (Figure 5-10.a) and Decision Tree (Figure 5-11.a).

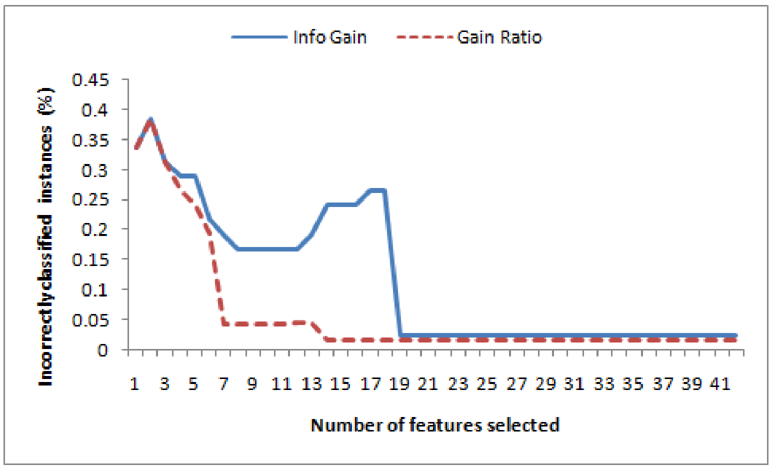
In contrast, JRip (Figure 6-5.a) showed that some y values reached 0, given certain x values, suggesting that Stealthy Attack-1 can be classified using JRip. Contrary to DDoS analysis with JRip (Figure 5-12.a) where all feature sets yielded incorrectly classified instances, the Stealthy Attack-1 analysis with JRip can have 0 incorrectly classified instances when certain sets of features were selected. That is, Information Gain-ranked features yielded 0 incorrectly classified instances when 10, 12, 15, and 17 features were selected. Gain Ratio-ranked features yielded 0 incorrectly classified instances when 5, 7 to 13, and 17 features were selected. It can be seen that it was difficult to predefine a set of features to yield 0 incorrectly classified instances, as the simulations did not show a regular pattern. Hence, while JRip was able to distinguish Stealthy Attack-1 from flash-crowd traffic, it had a drawback that a minimum set of features to yield 0 incorrectly classified instances was difficult to define.

Support Vector Machines (Figure 6-6.a) performed best as they showed 0 incorrectly classified instances when the number of features selected were 5 or above. Hence, a hypothetical intrusion-detection system that employed Support Vector Machine could adopt a set of features with at least 5 most relevant features selected, to detect attack traffic. On the other hand it showed 17.5% incorrectly classified instances when 3 most relevant Information Gain-ranked features were employed, or 11.6% when Gain Ratio-ranked features were employed. These numbers were the highest (worst) compared to the performance of other techniques (Naïve Bayes, Decision Tree, JRip). Hence, the hypothetical intrusion-detection-system could yield the worst percentage of incorrectly classified instances among the other techniques tested, when predefined to employ an incomplete set of features (such as a set of 3 features as demonstrated here).

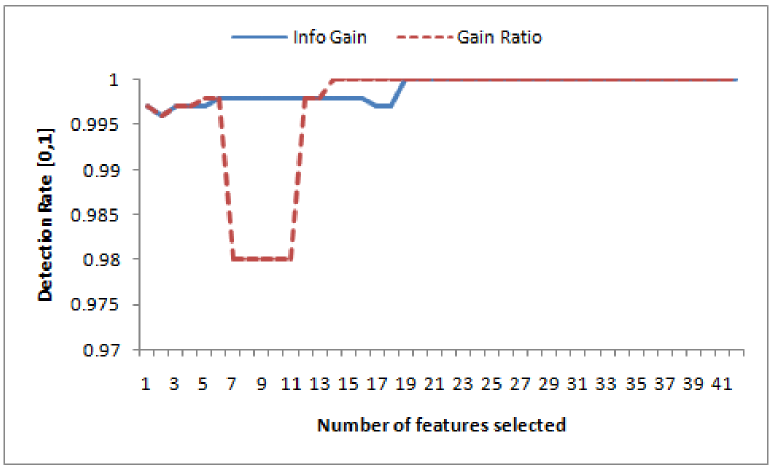
When True Positives were examined, all four machine learning techniques tested were able to produce a high rate of instances correctly classified as a given class. As shown in graphs (b) from Figure 6-3 to 6-6, Support Vector Machine yielded the highest Detection Rate with the least number of features, followed by Decision Tree and Naïve Bayes. It was difficult to define a threshold value for JRip to choose the number of features selected that resulted in a perfect Detection Rate, as the graph showed that the performance degraded when a certain feature set was selected. In particular, figure 6-5.b shows that JRip yielded Detection Rate of less than 1 when 1-4, 6-7, 19, 25-26, 30, 40 and 42 Information Gain-ranked features were selected; or when 1-4, 19, 25-26, 30, 40, and 42 Gain Ratio-ranked features were selected.

However, the high Detection Rate obtained using the four machine learning techniques tested were not without an associated cost. As shown in graphs (c) of Figure 6-3 to 6-6, Support Vector Machine yielded 18.4% normal traffic instances falsely classified as attacks when 3 features were employed. Other techniques tested showed between 0.4% to 0.5% incorrectly classified normal traffic.

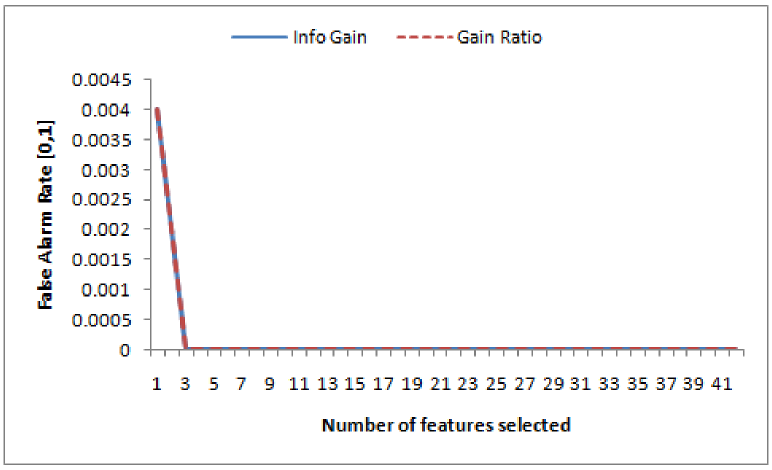
This study described the characteristics of stealthy attack traffic that made it indistinguishable when compared to flash-crowd traffic. Feature selection procedures were applied to rank the most relevant features using two techniques, Information Gain and Gain Ratio. The machine learning analysis showed the percentage of incorrectly classified instances, Detection Rate and False Alarm



(a) Incorrectly classified instances (%)

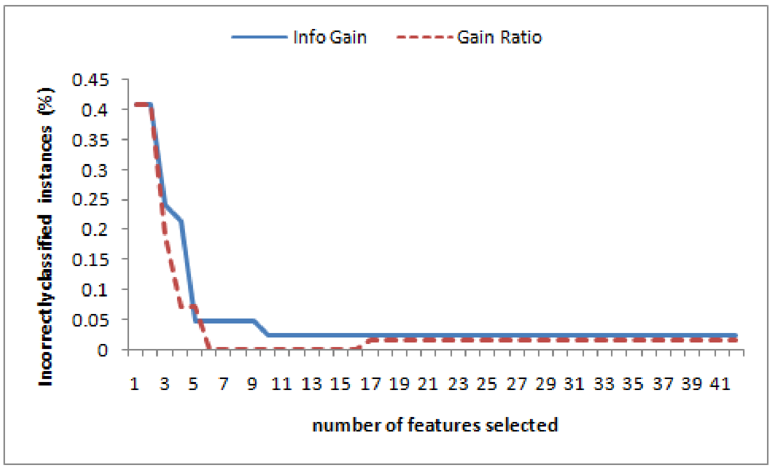


(b) Detection Rate

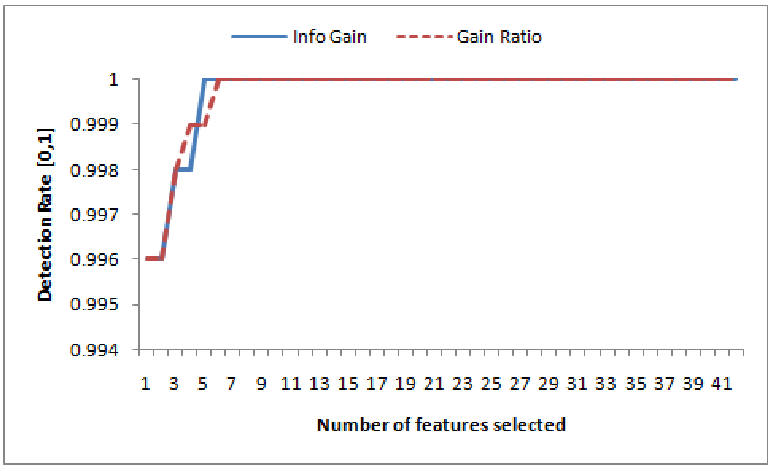


(c) False Alarm Rate

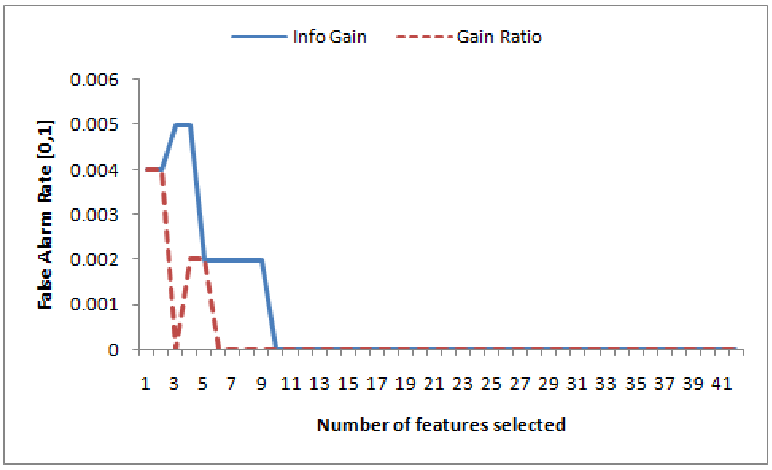
Figure 6-3: Stealthy Attack-1 Performance with Naive Bayes classification



(a) Incorrectly classified instances (%)

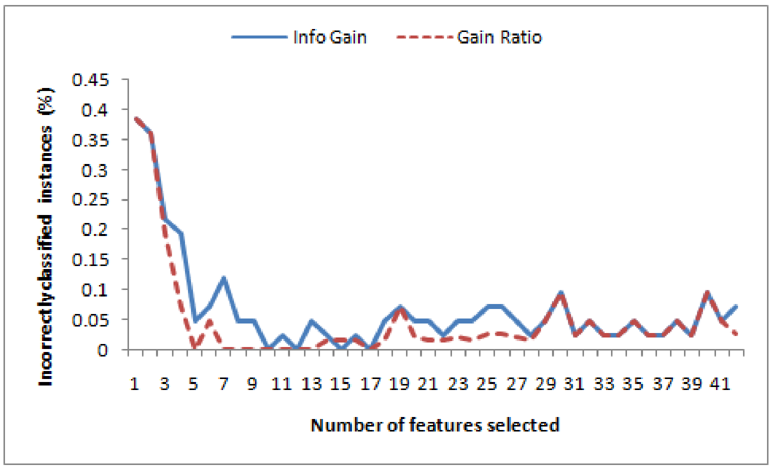


(b) Detection Rate

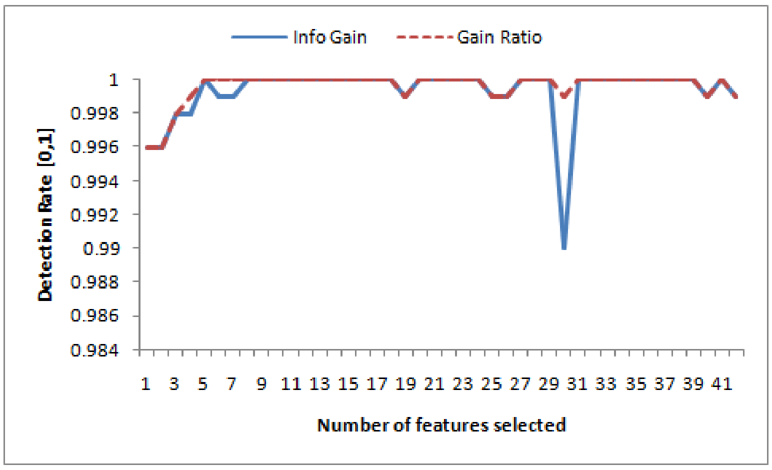


(c) False Alarm Rate

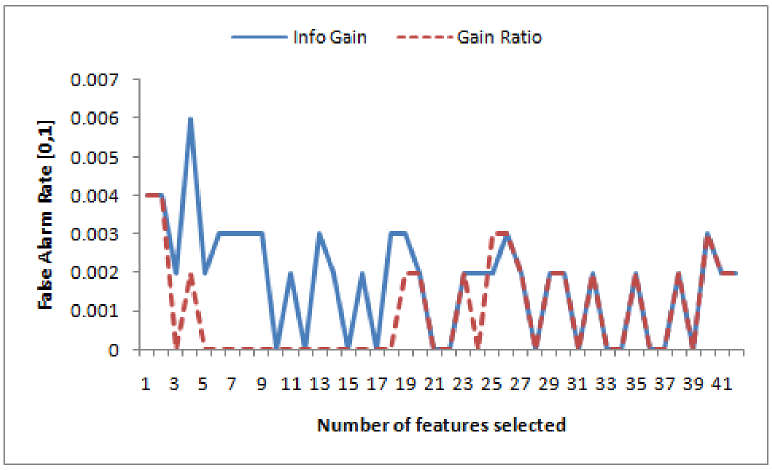
Figure 6-4: Stealthy Attack-1 Performance with Decision Tree classification



(a) Incorrectly classified instances (%)

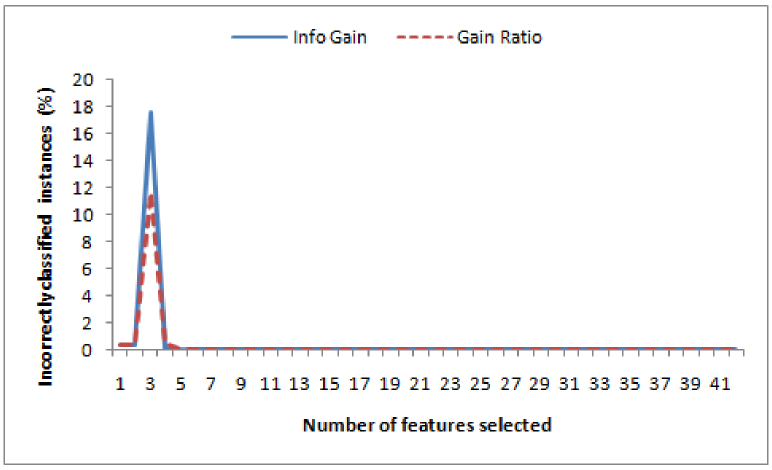


(b) Detection Rate

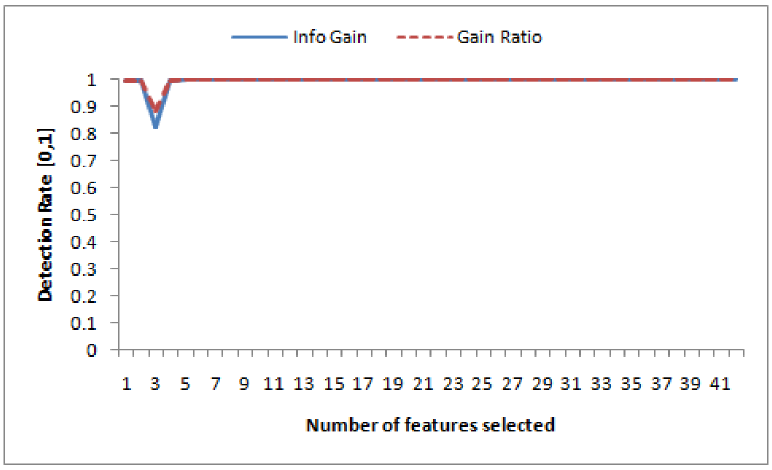


(c) False Alarm Rate

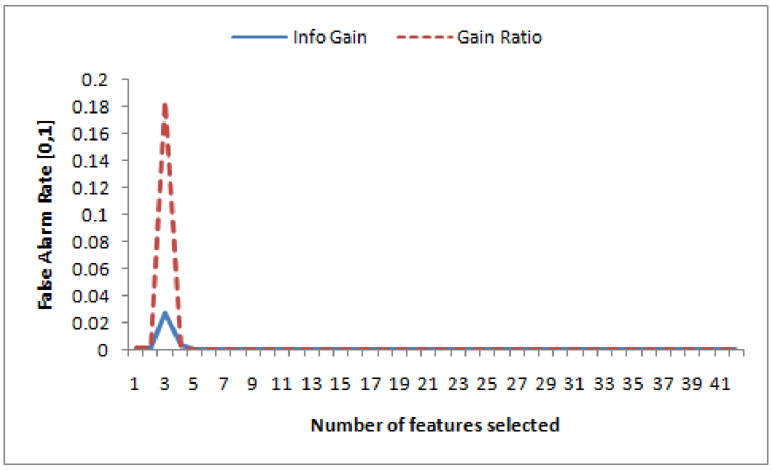
Figure 6-5: Stealthy Attack-1 Performance with JRip classification



(a) Incorrectly classified instances (%)



(b) Detection Rate



(c) False Alarm Rate

Figure 6-6: Stealthy Attack-1 Performance with Support Vector Machine classification

Rate when classifying the attack and flash-crowd traffic.

6.1.3 Conclusion

This section presented an attacking-bot model that impaired the performance of two machine learning techniques when classifying the bots-generated and flash-crowd traffic. The investigation showed that the attack model could rely on as few as 2 bots (Table 6.1) and remain undetected. Using the features proposed for this study (Table 3.6), the four machine learning techniques (Naïve Bayes, Decision Tree, JRip, Support Vector Machines) were able to yield accurate Detection Rates and low False Alarm Rates. However, Stealthy Attack-1 analysis with these four machine learning techniques showed more sets of selected features that yielded incorrectly classified instances (the (a) graphs from Figure 6-3 to 6-6), than the DDoS analysis (the (a) graphs from Figure 5-10 to 5-13). This demonstrates that the traffic generated through the proposed attack model in this section was stealthier than the Distributed Attack traffic presented earlier (Section 5.2).

The implementation method in this section attempted to mimic the flash-crowd traffic features. Particularly the method attempted to mimic the count_syn feature values, which is an HTTP/2 feature for the number of SYN packets sent from an attacking client towards a target machine. The next section examines how the attack model could resemble other flash-crowd traffic features, and therefore operate in even more stealthier manner.

6.2 Stealthy Attack 2

Previously it was demonstrated that HTTP/2 DDoS attack traffic could be generated to cause a CPU depletion of an HTTP/2 server without triggering an alarm at a hypothetical intrusion-detection system that monitored per connection CPU and memory consumption (Section 5.2). However, the traffic model it produced could be accurately classified when compared to flash-crowd traffic: a set of 4 features or more yielded 0 incorrectly classified instances when Naïve

Bayes, Decision Tree, or Support Vector Machines were employed. Therefore, Stealthy Attack-1 was proposed as a model to produce stealthier traffic less easily detectable by the detection system. It was shown that four machine learning techniques incorrectly classified the attack traffic that the model produced as flash-crowd traffic (Section 6.1).

Stealthy Attack-1 traffic was crafted to delude machine learning-based detection by camouflaging one of the traffic features (count_syn), which is representative of TCP connection flow between a device pair. This was done through having a group of bots (it was found that 1 bot would suffice) to send traffic, mimicked by count_syn feature values to represent flash-crowd traffic. The proposed attack model in the previous section led to another investigation if the same camouflage strategy could be adopted to produce stealthier HTTP/2 attack traffic. Hence, the aim of this section is to examine how two groups of bots could launch stealthier attack traffic than the previously defined Stealthy Attack-1 model. The study reported in this section is therefore named Stealthy Attack-2.

The following subsection details the proposed method to model and generate Stealthy Attack-2 traffic. It then analyses the traffic it generated using the same four machine learning techniques, and presents a comparison of the results.

6.2.1 Attack Model and Scenario

Similar to the strategy adopted to model Stealthy Attack-1 as presented in Section 6.1, this study proposes to camouflage attack traffic to mimic flash-crowd traffic. The difference is that this study aimed to have attack traffic mimics another feature value of flash-crowd traffic, i.e. size_rstAck. Hence, it was aimed that Stealthy Attack-2 traffic is stealthier than Stealthy Attack-1 traffic, indicated through yielding more incorrectly classified instances when analysed with machine learning techniques.

Two groups of bots were defined as part of the proposed model. A mime group aimed to mimic the flash-crowd traffic, and an offending group to generate attack traffic. In this study, the mime group mimicked another feature values,

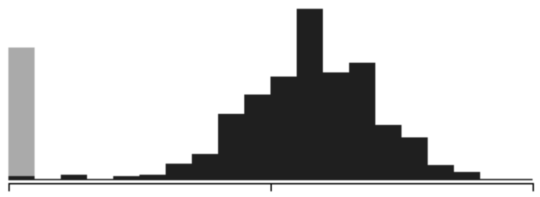


Figure 6-7: Visualization of Stealthy Attack-1 size_rstAck feature values.

i.e. size_rstAck in addition to the count_syn feature values of flash-crowd traffic. The minimum number of bots was observed when the traffic that the bots generated caused a target machine to continually show 100% CPU consumption.

The intuition behind mimicking the size_rstAck feature values of the flash-crowd traffic is that the Stealthy Attack-1 feature ranking results (Table 6.2) pointed to size_rstAck as the most relevant features. This feature represents the total size of TCP packets with RST and ACK flags set, or named *RST-ACK packets*, observed in a 1-second traffic instance on the victim machine. For connection termination, a TCP packet with the RST flag set, or named *RST packet*, is sent by one bot machine to a remote machine. RST-ACK packets sent by the bot are essentially RST packets with the ACK flag set, to also acknowledge a previous packet received by the same machine. To further mimic the flash-crowd traffic, the Stealthy Attack-2 study proposed to have the mime-bot group closely mimic the values of the size_rstAck feature.

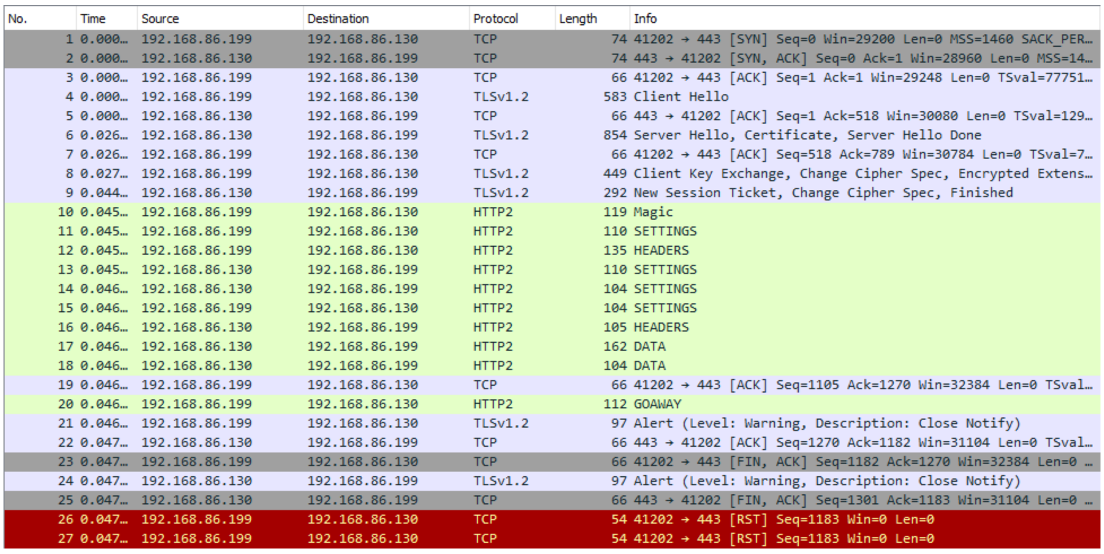
Figure 6-7 shows the size_rstAck feature values of both Stealthy Attack-1 and flash-crowd traffic dataset. The X-axis represents the feature values, and the Y-axis represents the number of traffic instances that was representative of such a feature value. The size_rstAck values of Stealthy Attack-1 traffic are shown as the tall grey bar on the left side of the figure, while the values of flash-crowd traffic is shown as the black normally-distributed graph on the right hand side of the figure. The figure shows that the size_rstAck values of both traffic datasets are different: Stealthy Attack-1 traffic shows lower size_rstAck values than those of flash-crowd traffic.

The difference between the size_rstAck values produced by the two traffic patterns was due to varying implementations of the attack and the flash-crowd traffic. The former was implemented using `nghttp2` library, while the latter was using `curl`. It is acceptable to have various implementations of a communication standard, since standards also clearly indicate the implementation requirement levels that ranged from "must" to "optional" for HTTP/2, according to RFC 2119 (Bradner, 1997). Consequently different HTTP/2 libraries, while maintaining the requirements mandated by the HTTP/2 standard (Belshe et al., May 2015), are not uniform in their implementations, and subsequently the traffic pattern they produce varies.

An example of the variations is shown in Figure 6-8 and 6-9, detailing traffic associated with one sample HTTP/2 Request traffic through implementation of `nghttp2` and `curl` libraries. The "Protocol" column specifies which data samples relate to HTTP/2 packets[1] and which ones to TCP packets. It could be seen from these figures that the traffic pattern produced by the two implementations differed. With nghttp2, the client was observed to terminate the connection by sending a RST packet. On the other hand, the curl library terminated the connection with a RST packet while at the same time acknowledging previously received packets; hence, the client was observed to finish the connection with an RST-ACK packet.

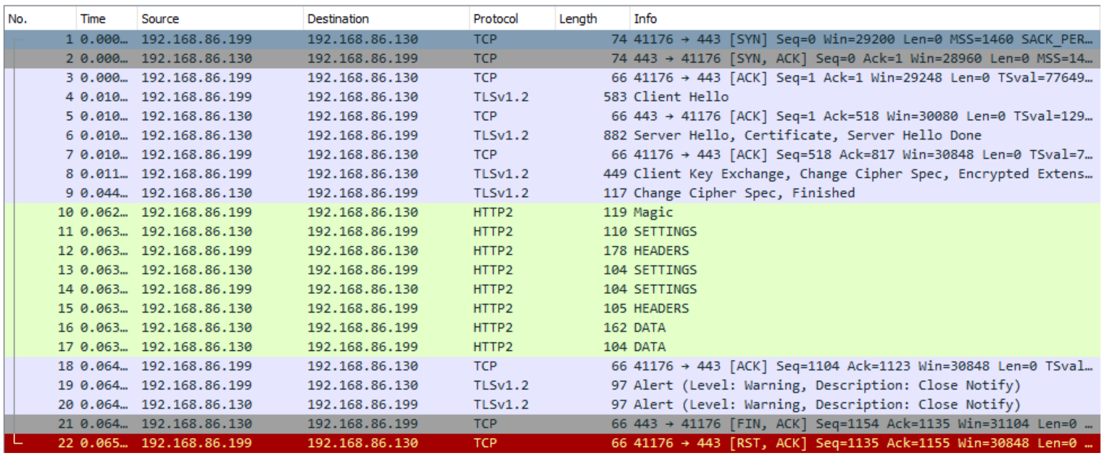
Understanding the cause of these differences, the study in this section therefore uses curl to implement the mime-bot group, while maintaining the use of nghttp2 as the engine for the offending group. The Stealthy Attack-2 implementation is presented in Table 6.3, comprising 4 bots. Bots 1 and 2 are the offending group, launching a flood of attacking traffic periodically towards a target machine. A flood was defined as 131K window_update packets sent in 38.5 ms. The stealthy factor 5 meant that a flood was sent once every 5 seconds on average by chance. Hence, there was $1/5 \times 1/5 = 1/25$ chance during each second that the two bots simultaneously launched a flood of traffic towards a victim. Bots 3 and 4 formed

[1]Encrypted messages cannot be inspected. However the content of the encrypted HTTP/2 traffic in this example could be revealed by the monitoring tool Wireshark, since the TLS Server Key was supplied to its configuration for debug and research purposes in this study. Otherwise, TLS/SSL encrypted traffic only showed "Application Data" to designate its data packets.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000…	192.168.86.199	192.168.86.130	TCP	74	41202 → 443 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PER…
2	0.000…	192.168.86.130	192.168.86.199	TCP	74	443 → 41202 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=14…
3	0.000…	192.168.86.199	192.168.86.130	TCP	66	41202 → 443 [ACK] Seq=1 Ack=1 Win=29248 Len=0 TSval=77751…
4	0.000…	192.168.86.199	192.168.86.130	TLSv1.2	583	Client Hello
5	0.000…	192.168.86.130	192.168.86.199	TCP	66	443 → 41202 [ACK] Seq=1 Ack=518 Win=30080 Len=0 TSval=129…
6	0.026…	192.168.86.130	192.168.86.199	TLSv1.2	854	Server Hello, Certificate, Server Hello Done
7	0.026…	192.168.86.199	192.168.86.130	TCP	66	41202 → 443 [ACK] Seq=518 Ack=789 Win=30784 Len=0 TSval=7…
8	0.027…	192.168.86.199	192.168.86.130	TLSv1.2	449	Client Key Exchange, Change Cipher Spec, Encrypted Extens…
9	0.044…	192.168.86.130	192.168.86.199	TLSv1.2	292	New Session Ticket, Change Cipher Spec, Finished
10	0.045…	192.168.86.199	192.168.86.130	HTTP2	119	Magic
11	0.045…	192.168.86.199	192.168.86.130	HTTP2	110	SETTINGS
12	0.045…	192.168.86.199	192.168.86.130	HTTP2	135	HEADERS
13	0.045…	192.168.86.130	192.168.86.199	HTTP2	110	SETTINGS
14	0.046…	192.168.86.130	192.168.86.199	HTTP2	104	SETTINGS
15	0.046…	192.168.86.199	192.168.86.130	HTTP2	104	SETTINGS
16	0.046…	192.168.86.130	192.168.86.199	HTTP2	105	HEADERS
17	0.046…	192.168.86.130	192.168.86.199	HTTP2	162	DATA
18	0.046…	192.168.86.130	192.168.86.199	HTTP2	104	DATA
19	0.046…	192.168.86.199	192.168.86.130	TCP	66	41202 → 443 [ACK] Seq=1105 Ack=1270 Win=32384 Len=0 TSval…
20	0.046…	192.168.86.199	192.168.86.130	HTTP2	112	GOAWAY
21	0.046…	192.168.86.199	192.168.86.130	TLSv1.2	97	Alert (Level: Warning, Description: Close Notify)
22	0.047…	192.168.86.130	192.168.86.199	TCP	66	443 → 41202 [ACK] Seq=1270 Ack=1182 Win=31104 Len=0 TSval…
23	0.047…	192.168.86.199	192.168.86.130	TCP	66	41202 → 443 [FIN, ACK] Seq=1182 Ack=1270 Win=32384 Len=0 …
24	0.047…	192.168.86.130	192.168.86.199	TLSv1.2	97	Alert (Level: Warning, Description: Close Notify)
25	0.047…	192.168.86.130	192.168.86.199	TCP	66	443 → 41202 [FIN, ACK] Seq=1301 Ack=1183 Win=31104 Len=0 …
26	0.047…	192.168.86.199	192.168.86.130	TCP	54	41202 → 443 [RST] Seq=1183 Win=0 Len=0
27	0.047…	192.168.86.199	192.168.86.130	TCP	54	41202 → 443 [RST] Seq=1183 Win=0 Len=0

Figure 6-8: A sample of HTTP/2 Request traffic produced using nghttp2 library



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000…	192.168.86.199	192.168.86.130	TCP	74	41176 → 443 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PER…
2	0.000…	192.168.86.130	192.168.86.199	TCP	74	443 → 41176 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=14…
3	0.000…	192.168.86.199	192.168.86.130	TCP	66	41176 → 443 [ACK] Seq=1 Ack=1 Win=29248 Len=0 TSval=77649…
4	0.010…	192.168.86.199	192.168.86.130	TLSv1.2	583	Client Hello
5	0.010…	192.168.86.130	192.168.86.199	TCP	66	443 → 41176 [ACK] Seq=1 Ack=518 Win=30080 Len=0 TSval=129…
6	0.010…	192.168.86.130	192.168.86.199	TLSv1.2	882	Server Hello, Certificate, Server Hello Done
7	0.010…	192.168.86.199	192.168.86.130	TCP	66	41176 → 443 [ACK] Seq=518 Ack=817 Win=30848 Len=0 TSval=7…
8	0.011…	192.168.86.199	192.168.86.130	TLSv1.2	449	Client Key Exchange, Change Cipher Spec, Encrypted Extens…
9	0.044…	192.168.86.130	192.168.86.199	TLSv1.2	117	Change Cipher Spec, Finished
10	0.062…	192.168.86.199	192.168.86.130	HTTP2	119	Magic
11	0.063…	192.168.86.130	192.168.86.199	HTTP2	110	SETTINGS
12	0.063…	192.168.86.199	192.168.86.130	HTTP2	178	HEADERS
13	0.063…	192.168.86.130	192.168.86.199	HTTP2	104	SETTINGS
14	0.063…	192.168.86.199	192.168.86.130	HTTP2	104	SETTINGS
15	0.063…	192.168.86.130	192.168.86.199	HTTP2	105	HEADERS
16	0.063…	192.168.86.130	192.168.86.199	HTTP2	162	DATA
17	0.063…	192.168.86.130	192.168.86.199	HTTP2	104	DATA
18	0.064…	192.168.86.199	192.168.86.130	TCP	66	41176 → 443 [ACK] Seq=1104 Ack=1123 Win=30848 Len=0 TSval…
19	0.064…	192.168.86.199	192.168.86.130	TLSv1.2	97	Alert (Level: Warning, Description: Close Notify)
20	0.064…	192.168.86.130	192.168.86.199	TLSv1.2	97	Alert (Level: Warning, Description: Close Notify)
21	0.064…	192.168.86.130	192.168.86.199	TCP	66	443 → 41176 [FIN, ACK] Seq=1154 Ack=1135 Win=31104 Len=0 …
22	0.065…	192.168.86.199	192.168.86.130	TCP	66	41176 → 443 [RST, ACK] Seq=1135 Ack=1155 Win=30848 Len=0 …

Figure 6-9: A sample of HTTP/2 Request traffic produced using curl library

Table 6.3: Stealthy Attack-2 model

	Bot 1	Bot 2	Bot 3	Bot 4
# threads	1	1	2	40
# streams	1	1	1	1
# window_update	131K	131K	0	0
stealthy factor	5	5	n.a.	n.a.
delay	1 sec	1 sec	0.001 ms	5 sec

the mime group that attempted to mimic the flash-crowd traffic. Because these two bots did not send any window_update traffic to the victim, any number assigned to the stealthy factor did not serve any purpose for launching attack packets. Hence, the table shows "n.a." for the stealthy factors. The number of threads indicates the number of instances of the above scenario that were run by each bot during a given time frame. These threads, although relatively small in their number (2 for Bot 3, and 40 for Bot 4), were an attempt to mimic the 5200 users (Section 4.4.1) that comprised flash-crowd traffic.

Maintaining consistency with the performance measurement previously carried out for Stealthy Attack-1, the traffic generated in this section was extracted using the feature extraction method described in Section 3.1.3 (page 75), yielding an attack traffic dataset with 254 instances. The dataset was merged with the flash-crowd dataset which consisted of 3600 instances (explained in Section 4.4.2), yielding a dataset with 3854 instances. The features from the dataset were ranked using Information Gain and Gain Ratio techniques, following which the traffic was classified using four machine learning techniques, i.e. Naïve Bayes, Decision Tree, JRip, and Support Vector Machines. The machine learning classifications were executed on an Intel Core-i3 machine with a 2 GB RAM. Three machine learning techniques, i.e. Naïve Bayes, Decision Tree, and JRip, took less than 1 minute to yield classification results. Support Vector Machines took more than 2 minutes to yield classification results. The parameter values of these machine learning techniques are given in Appendix A. The following subsection presents the results of this investigation.

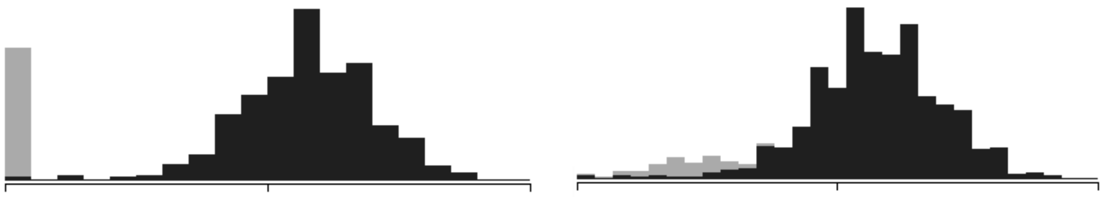


Figure 6-10: Visualization of the size_rstAck feature values. Left: Stealthy Attack-1. Right: Stealthy Attack-2.

6.2.2 Results and Analysis

Comparison of the RST-ACK flag packets size

It was aimed to have the size_rstAck feature values of the attack traffic resemble that of the flash-crowd traffic. Figure 6-10 shows the comparison of the Stealthy Attack-1 size_rstAck values (the left figure), with the results obtained through experiments reported in this section (the right figure).

In both figures, the black graph represents the flash-crowd size_rstAck values, and the grey represents the attack traffic size_rstAck values. The left figure shows that the Stealthy Attack-1 traffic (grey) could be clearly distinguished from the flash-crowd traffic (black) by designating a threshold value, such as a vertical line separating the two colours. In contrast, the right figure shows that the Stealthy Attack-2 size_rstAck values amalgamated with that of the flash-crowd traffic, causing ambiguity in differentiation of both traffic types.

The size_rstAck values extracted from the Stealthy Attack-2 traffic were generated by the mime group, i.e. bots 3 and 4 shown in Table 6.3. Hence, the right side of Figure 6-10 demonstrates that the mime group camouflaged the attack traffic with flash-crowd and caused encumbrance in the detection process.

Feature ranking and performance analysis

Table 6.4 shows the ranked values of features extracted from Stealthy Attack-2 as well as flash-crowd traffic. The size_rstAck feature was no longer ranked as the topmost. The table shows that it was listed as rank 6 when measured using

Information Gain, and ordered 7 with Gain Ratio.

The performance analysis of machine learning techniques is shown in Figure 6-11 to 6-14. The results are for Naïve Bayes, Decision Tree, JRip, and Support Vector Machines. The following discussion is to firstly discuss the incorrectly classified instances, which are shown on the (a) part of the figures, followed by Detection Rates (the b graphs) and False Alarm Rates (the c graphs).

Stealthy Attack-2 traffic yielded incorrectly classified instances when using the first three classifiers, i.e. Naïve Bayes, Decision Tree and JRip, regardless of the feature sets used. Only Support Vector Machine were able to distinguish the traffic when at least 5 Information Gain-ranked features or 13 Gain Ratio-ranked features were selected. Interestingly, the traffic led to Naïve Bayes yielding higher percentage of incorrectly classified instances with greater number of features selected. This result showed that Stealthy Attack-2 caused a degraded classification when the graphs were compared to those of Stealthy Attack-1.

Other performance measurements made included Detection Rate and False Alarm Rate. Three of the techniques, i.e. Naïve Bayes, JRip and Support Vector Machines, reached a Detection Rate of 100% when certain feature sets were selected, but no pattern was observed on the number of features required to achieve the same. Decision Tree did not yield a 100% Detection Rate for any of the feature sets analysed.

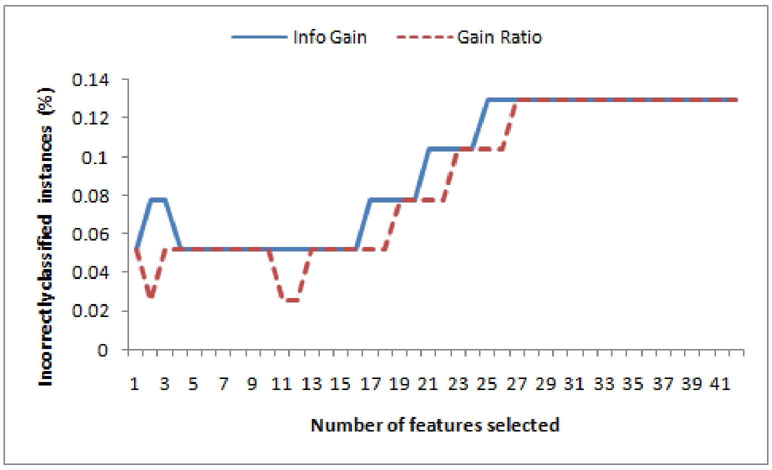
Decision Tree yielded a 0% False Alarm Rate. However other techniques did not perform as well. Naïve Bayes and JRip always yielded high False Alarm Rates with all feature sets. Support Vector Machine could reach 0% False Alarm Rate when at least 13 features were selected.

Comparison with DDoS and Stealthy Attack-1 performance

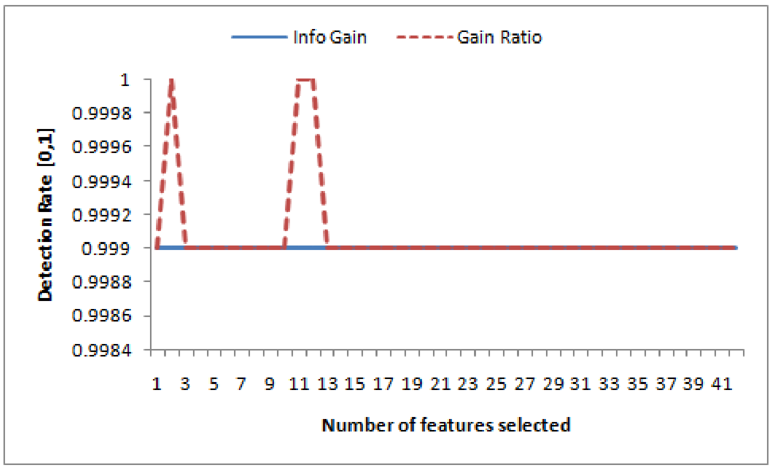
This part of the study compares the performance measurement obtained in this section with those from the previous sections. It considers the 3 attack traffic models presented in this study, DDoS (Section 5.2.1), Stealthy Attack-1 (Section 6.1.1) and Stealthy Attack-2 traffic (Section 6.2.1). Here, the best performance values of the incorrectly classified instances, Detection Rate, and False Positive

Table 6.4: Ranked features for Stealthy Attack-2 traffic

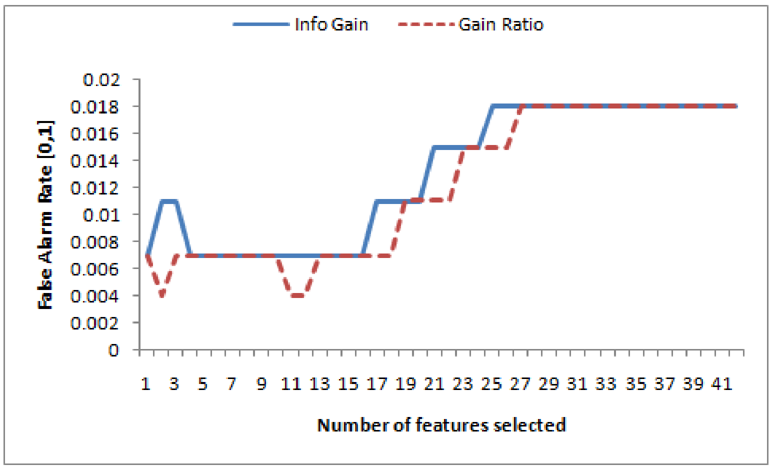
rank	Information Gain		Gain Ratio	
	feature #	feature name	feature #	feature name
1	34	size_tlsKey	19	size_tlsKey
2	33	count_tlsKey	18	size_app
3	1	count_app	2	count_tlsKey
4	2	size_app	34	count_app
5	29	size_tlsHello	1	lapse_tlsHello_max
6	19	size_rstAck	33	count_rstAck
7	18	count_rstAck	6	size_rstAck
8	42	lapse_encAlert_max	7	lapse_tlsHello_ave
9	22	lapse_rstAck_max	28	lapse_encAlert_max
10	17	lapse_rst_max	39	lapse_rstAck_max
11	27	lapse_finAck_max	38	lapse_rst_max
12	7	size_syn	24	lapse_finAck_max
13	6	count_syn	23	size_tlsHello
14	28	count_tlsHello	27	count_encAlert
15	38	count_encAlert	22	size_encAlert
16	39	size_encAlert	26	size_syn
17	21	lapse_rstAck_ave	21	count_syn
18	13	count_rst	29	count_tlsHello
19	14	size_rst	17	lapse_rstAck_ave
20	8	count_ack	16	count_rst
21	9	size_ack	14	size_rst
22	24	size_finAck	13	count_ack
23	23	count_finAck	42	size_ack
24	26	lapse_finAck_ave	8	lapse_finAck_ave
25	41	lapse_encAlert_ave	41	size_finAck
26	16	lapse_rst_ave	9	count_finAck
27	20	lapse_rstAck_min	20	lapse_rst_ave
28	25	lapse_finAck_min	25	lapse_encAlert_ave
29	15	lapse_rst_min	5	lapse_rstAck_min
30	32	lapse_tlsHello_max	15	lapse_finAck_min
31	31	lapse_tlsHello_ave	12	lapse_rst_min
32	11	lapse_ack_ave	4	lapse_app_min
33	3	lapse_app_min	37	lapse_encAlert_min
34	40	lapse_encAlert_min	11	lapse_tlsKey_max
35	5	lapse_app_max	32	lapse_app_ave
36	4	lapse_app_ave	36	lapse_app_max
37	12	lapse_ack_max	3	lapse_ack_max
38	10	lapse_ack_min	40	lapse_tlsHello_min
39	30	lapse_tlsHello_min	31	lapse_tlsKey_ave
40	37	lapse_tlsKey_max	35	lapse_ack_ave
41	36	lapse_tlsKey_ave	30	lapse_ack_min
42	35	lapse_tlsKey_min	10	lapse_tlsKey_min



(a) Incorrectly classified instances (%)

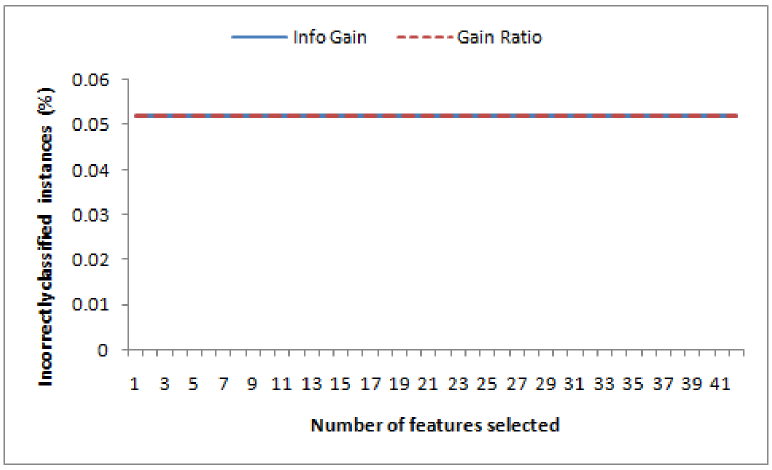


(b) Detection Rate

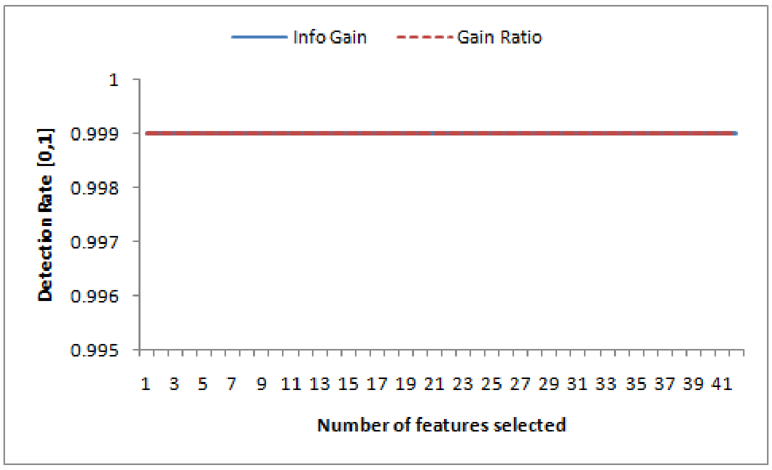


(c) False Alarm Rate

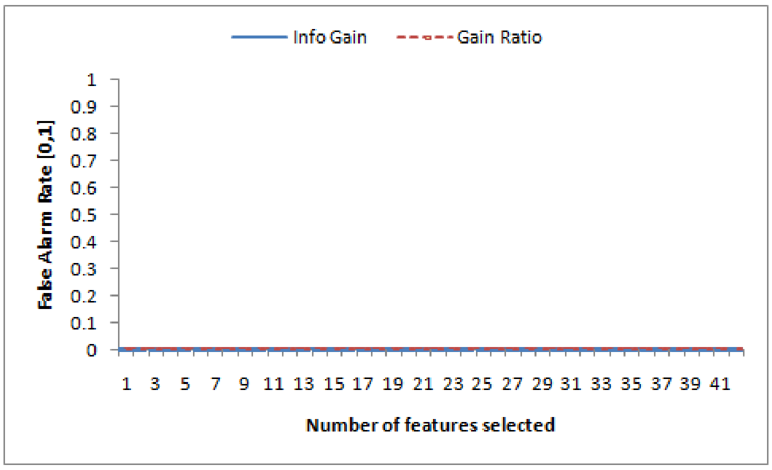
Figure 6-11: Stealthy Attack-2 Performance with Naive Bayes classification



(a) Incorrectly classified instances (%)

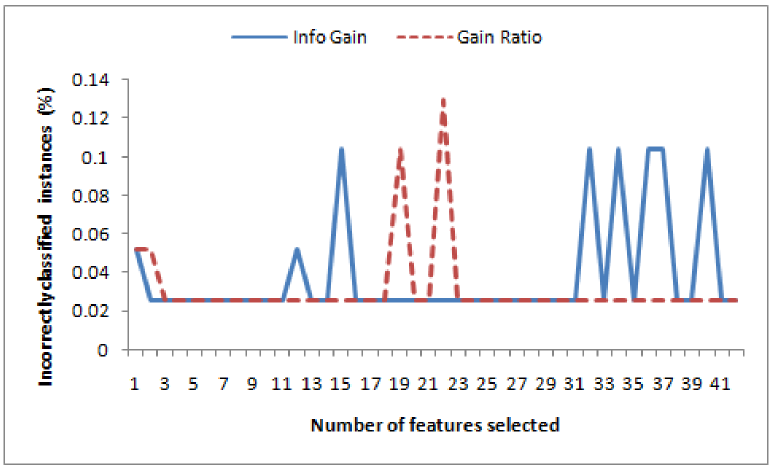


(b) Detection Rate

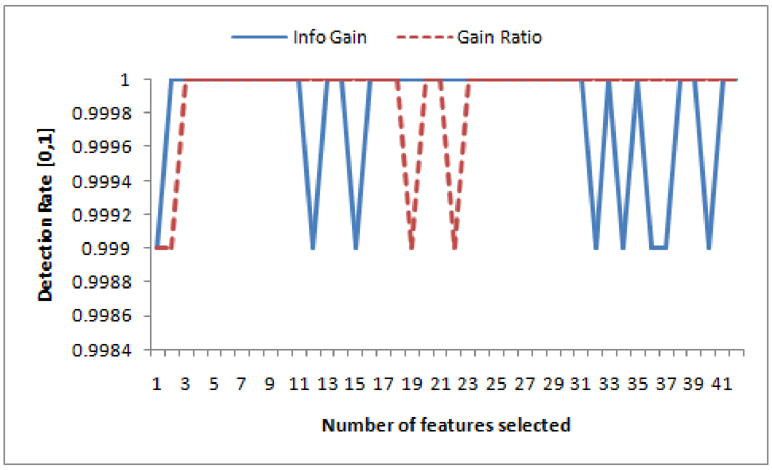


(c) False Alarm Rate

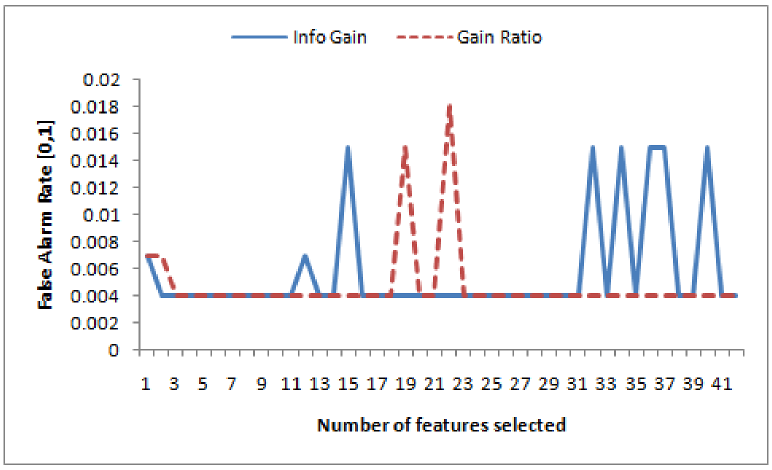
Figure 6-12: Stealthy Attack-2 Performance with Decision Tree classification



(a) Incorrectly classified instances (%)

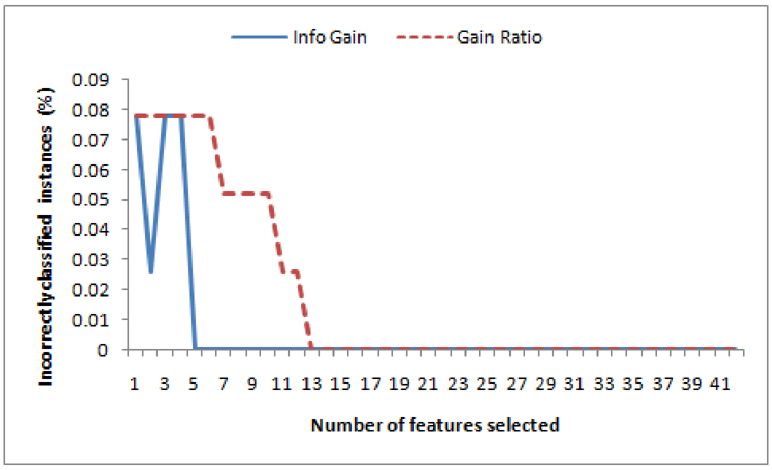


(b) Detection Rate

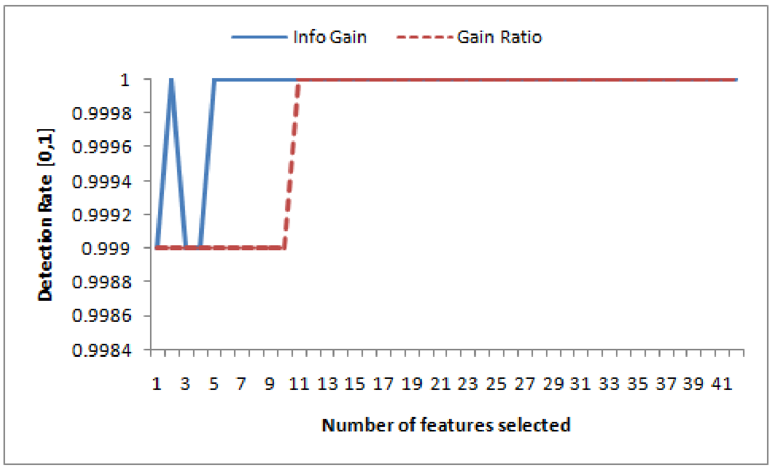


(c) False Alarm Rate

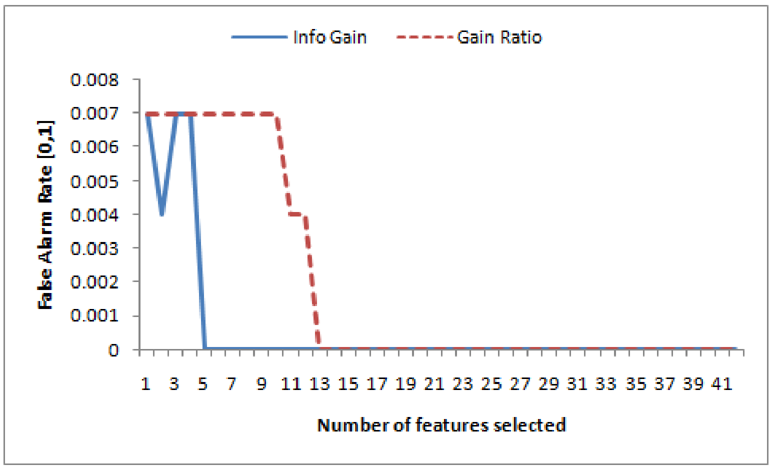
Figure 6-13: Stealthy Attack-2 Performance with JRip classification



(a) Incorrectly classified instances (%)



(b) Detection Rate



(c) False Alarm Rate

Figure 6-14: Stealthy Attack-2 Performance with Support Vector Machine classification

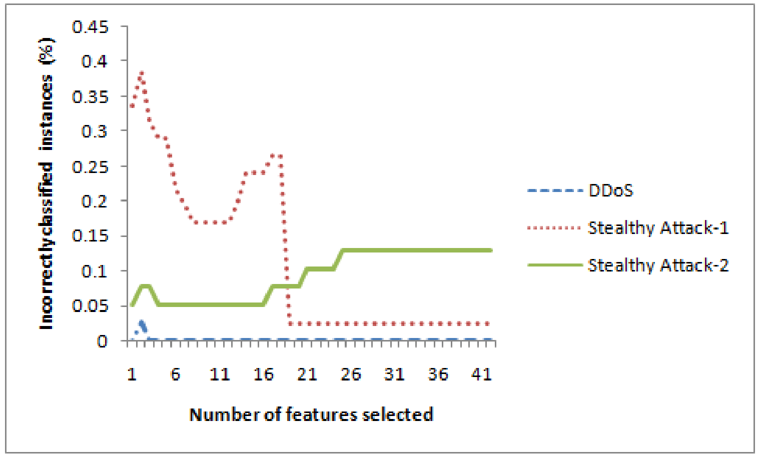
Rate were compared. Three approaches were applied: through visual inspections, through comparing the best (highest/lowest) performance values, and through Self Organizing Map-based visualization. The first two approaches, the visual inspections and comparing the best performance values, analysed Figure 5-10 to 5-13 for DDoS, Figure 6-3 to 6-6 for Stealthy Attack-1, and Figure 6-11 to 6-14 for Stealthy Attack-2 performance results.

First, visual inspections were applied to find a range of numbers of selected features, to yield a low percentage of incorrectly classified instances. The reasoning behind this is that a hypothetical intrusion-detection system would choose rules that can yield the lowest 'incorrectly classified instances' value. Figure 6-15 was used to visually choose a number of selected features. The figure combined the incorrectly classified instances (Y-axis) of the 3 attack traffic models when Information Gain-ranked features were employed. For example, Figure 6-15.a combines the graphs from Figure 5-10.a for DDoS, Figure 6-3.a for Stealthy Attack-1, and Figure 6-11.a for Stealthy Attack-2 results. The X-axis represents the number of features selected. Figure 6-15 helps visualize a range of X-axis values that yield low Y-axis values

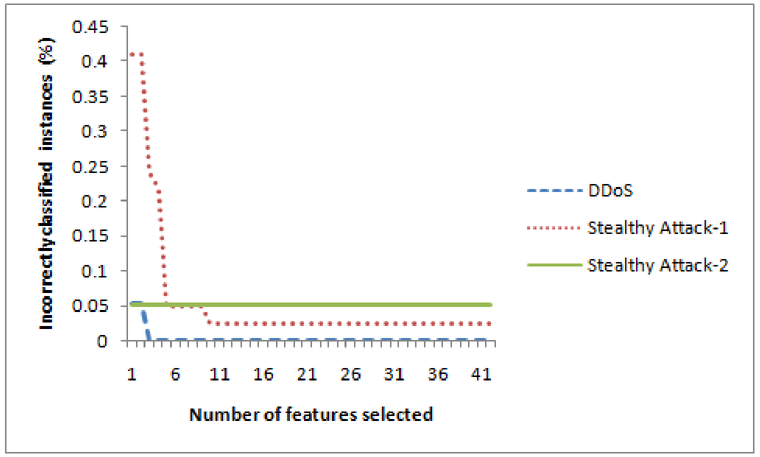
From visually inspecting Figure 6-15.a, a hypothetical intrusion-detection-system that employed Naïve Bayes would choose 19 selected features or higher to obtain low Y values. In this regard, Stealthy Attack-2 was found to be the stealthiest due to its high Y values, followed by Stealthy Attack-1 and DDoS.

A visual inspection on Decision Tree (Figure 6-15.b) also shows that Stealthy Attack-2 yielded the highest Y values, hence the stealthiest, when a hypothetical intrusion-detection system employed 5 Information Gain-ranked features or higher. The figure also shows that Stealthy Attack-1 was less stealthy, and DDoS was the least stealthy.

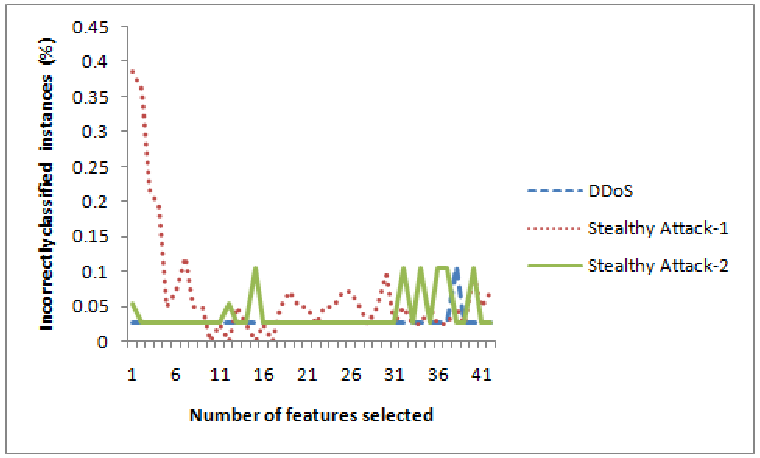
JRip (Figure 6-15.c) graphs show that between 10 and 17 features can be selected to yield low Y values. When these features are selected, Stealthy Attack-2 was the stealthiest (as it showed the highest Y value), followed by DDoS and Stealthy Attack-1.



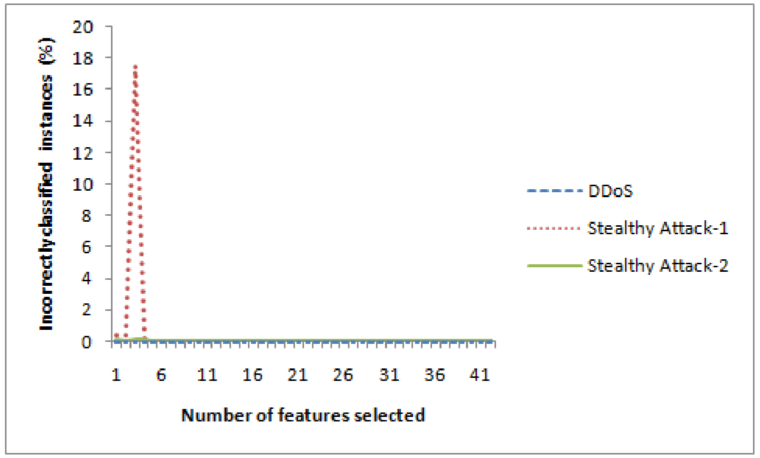
(a) Naïve Bayes



(b) Decision Tree



(c) JRip



(d) Support Vector Machine

Figure 6-15: Visual inspection to find a range of X-values that yield low incorrectly classified instances values.

Support Vector Machine (Figure 6-15.d) show otherwise. When 5 or more features were employed, DDoS was the stealthiest, as it yielded 0.024% incorrectly classified instances (Figure 5-13). A hypothetical intrusion-detection system can distinguish the other two attack traffic models, Stealthy Attack-1 and Stealthy Attack-2, from flash-crowd when 5 or more features were employed.

From visual inspection, it was discussed that Naïve Bayes, Decision Tree, and JRip show that Stealthy Attack-2 bears the stealthiest traffic among the 3 traffic models analysed. Support Vector Machines yield otherwise, with DDoS being the stealthiest.

The second investigation compared the boundary values of the performance results. The boundary performance values were indicated by the best figure a classifier measured regardless of the number of features selected, i.e. the farthest Y value regardless of X. Hence, these were the lowest values of incorrectly classified instances, the highest values of Detection Rate, and the lowest values of False Alarm Rate. The comparisons are presented in Figure 6-16 to 6-17.

The incorrectly classified instances (Figure 6-16) showed the higher bars as stealthier traffic. Hence, Stealthy Attack-2 traffic was the stealthiest when measured with Naïve Bayes and Decision Tree (Figure 6-16). However, this result was not uniform across other classifiers tested. JRip showed that DDoS traffic was stealthier than Stealthy Attack-2, as DDoS showed a higher bar than Stealthy Attack-2. Similarly, JRip showed that Stealthy Attack-2 was stealthier than Stealthy Attack-1, as the former showed a higher bar than the latter. Support Vector Machines were able to yield 0% incorrectly classified instances when classifying the 3 proposed attack traffic models. Hence, it can be concluded that Support Vector Machines can be applied to identify all three attack traffic models presented in this study.

The Detection Rate showed the lower bars. Figure 6-17 shows that Stealthy Attack-2 was the stealthiest among other traffic types as it yielded a low Detection Rate when Naïve Bayes and Decision Tree classifiers were employed. The other two classifiers, JRip and Support Vector Machines, were able to yield a 100% Detection Rate.

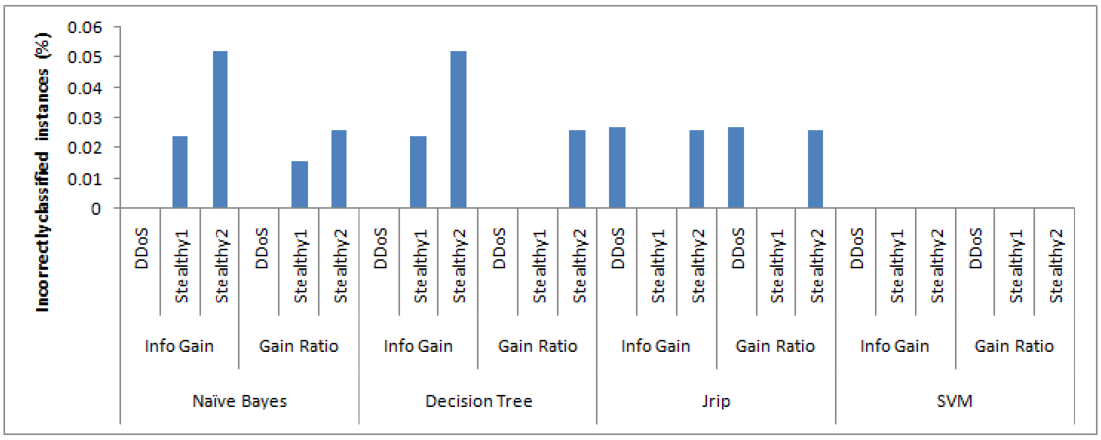


Figure 6-16: Incorrectly classified instances for all 3 attack types

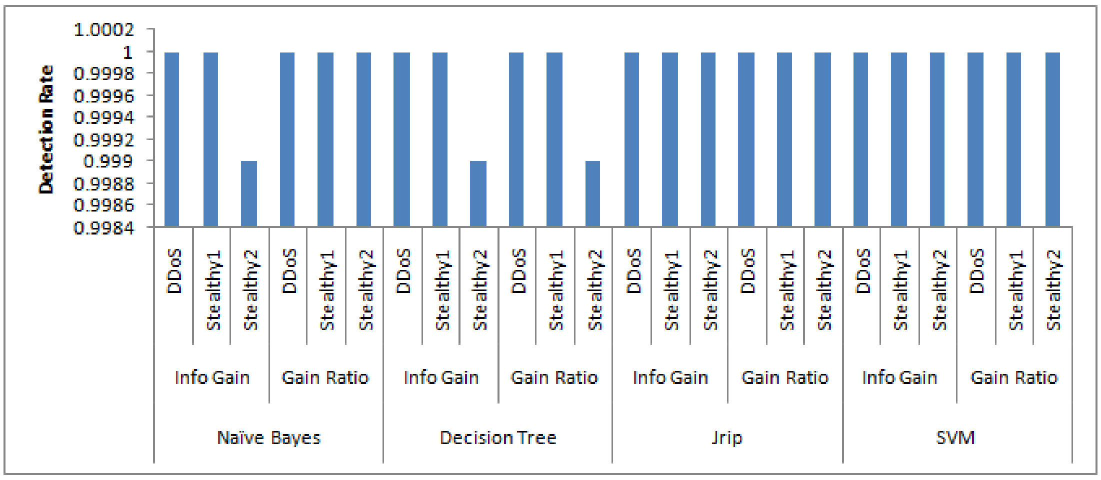


Figure 6-17: Detection Rate performance comparison for all 3 attack types

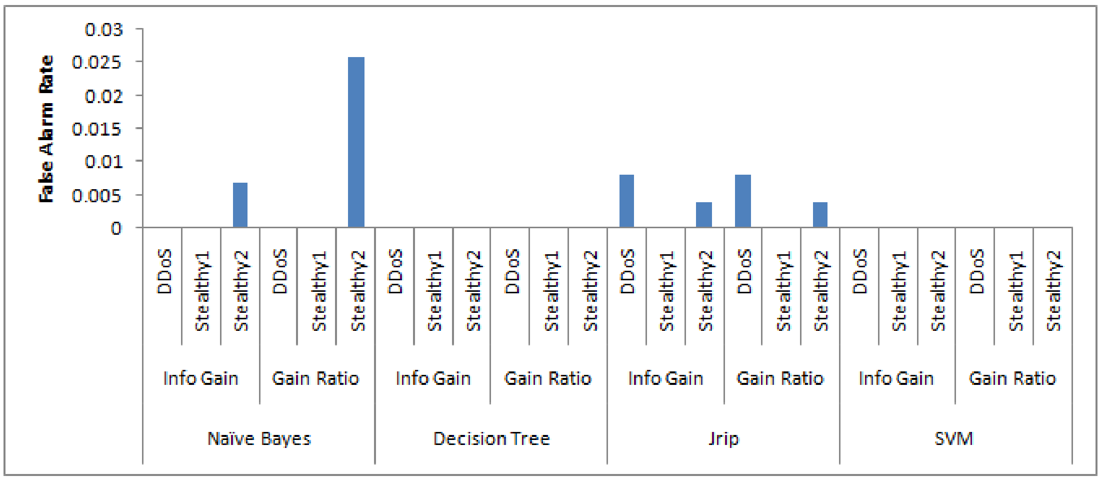


Figure 6-18: False Alarm Rate performance comparison for all 3 attack types

The False Alarm Rate showed the higher bars as the stealthier traffic (Figure 6-18). Stealthy Attack-2 was the stealthiest when classified using Naïve Bayes. However, it was second to DDoS traffic when measured through JRip. Other classifiers, namely JRip and Support Vector Machine showed a False Alarm Rate of 0% for all traffic types.

The third traffic comparison analysis was done using Self Organizing Maps, a machine learning technique that visualize data in clusters. The parameter values of the machine learning technique are given in Appendix A. For this analysis, four datasets were generated in this study, i.e. flash-crowd, DDoS, Stealthy Attack-1 and Stealthy Attack-2, were merged, yielding a dataset with 4527 instances. The Self Organizing Map analysis was run on an Intel Core-i3 machine with a 2 GB RAM.

The results are shown in Figure 6-19. The X axis shows the classes from the dataset generated in this study, which are separated by black vertical lines. Each integer value of X corresponds to an instance in the dataset. The Y axis shows how the technique assigns the instances from the dataset into one of four clusters. If the technique produces perfect classification, the graph would show only one cluster colour Y for each X traffic dataset, i.e.:

- Blue for Stealthy Attack-1,
- Red for flash-crowd traffic,
- Turquoise (greenish-blue) for DDoS, and
- Green for Stealthy Attack-2.

However, the graph assigned other colours to three of the four traffic datasets. For example, Stealthy Attack-1 was not only blue, but also green and turquoise, indicating how the Self Organizing Map technique incorrectly assigned the blue traffic in a separate cluster. Similarly, Stealthy Attack-2 traffic was also mistakenly classified as other attack traffic. Stealthy Attack-2 cluster was assigned green, but Self Organizing Map clustered the traffic as green, blue, and turquoise. On the other hand, all DDoS traffic was distinguished simply as a turquoise cluster.

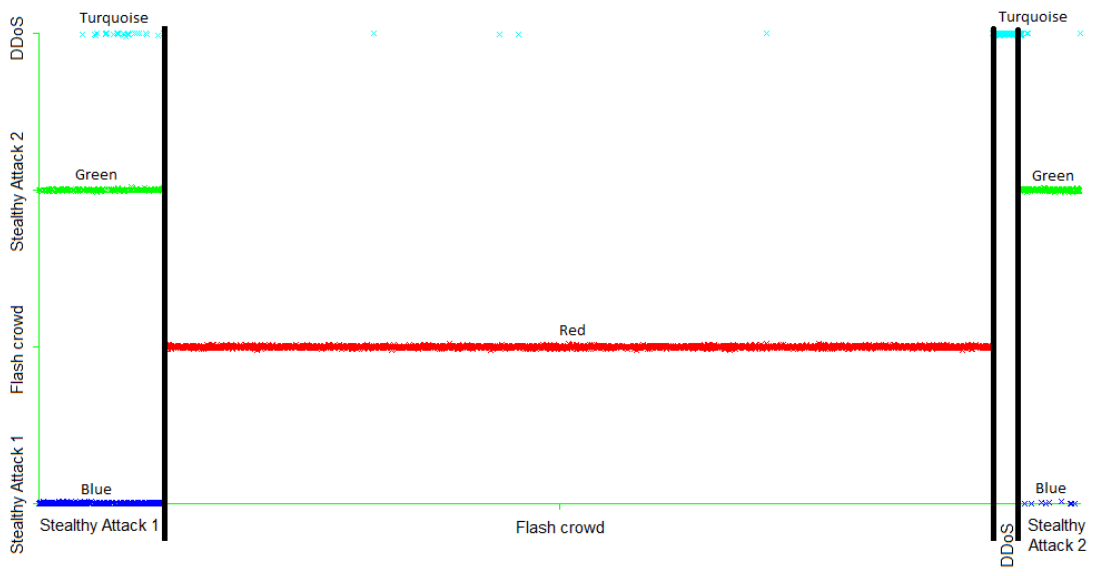


Figure 6-19: Cluster visualization using Self Organizing Map

The flash-crowd traffic was labelled as a red cluster. Although Self Organizing Map showed some flash-crowd instances labelled as DDoS, most flash-crowd instances were categorised in the red cluster. This indicates that Self Organizing Map was able to accurately distinguish flash-crowd traffic from attack traffic.

While the Self Organizing Map analysis was run on an Intel Core-i3 machine with a 2 GB RAM, it took more than 10 minutes for Weka to analyse the dataset with 4527 instances. In this regards, Self Organizing Maps were not suitable for online environment, where real-time analysis is required to classify traffic.

To conclude, this section presented three approaches to compare the attack traffic with flash crowd using several machine learning techniques. It could be seen that Stealthy Attack-2 showed many signs that it was stealthier than Stealthy Attack-1 and DDoS, particularly when the analysis applied Naïve Bayes and Decision Tree. Although JRip generated different results, it did not show a consistent trend given a range of numbers of features selected. Hence, it was difficult to predefine a set of features where JRip could yield the best performance.

Support Vector Machines can be used as a means to distinguish all of the proposed HTTP/2 attack traffic from flash-crowd. However, it had some drawbacks. First, certain sets of features selected returned adverse classification results. Second, it took a very long execution time (more than 2 minutes) to run the classifier compared to the other ones tested, which took less than 1 second to obtain the classification results. The Support Vector Machine analysis was run on an Intel Core-i3 machine with a 2 GB RAM. The analysis time length showed that Support Vector Machines were inappropriate to be used as a real-time intrusion-detection system.

6.2.3 Conclusion

This section proposed a stealthier attack model through mimicking a different traffic feature, i.e. the total size of RST-ACK packets observed in a second. Using the same attack model as previously described, the study proposed two groups of bots, where one offending group generated attack traffic and another group mimicked the flash-crowd traffic. The mime group was built using the same programming library (i.e. curl) as the one used to build the flash-crowd traffic. The results showed that the Stealthy Attack-2 RST-ACK feature values (i.e. the size_rstAck feature) overlapped with the same feature values of the flash-crowd traffic, indicating how the former traffic mimicked the latter. Furthermore, the Stealthy Attack-2 classification using three classifiers (Naïve Bayes, Decision Tree and JRip) resulted in stealthier performance measurements than the Stealthy Attack-1 and DDoS traffic models.

Support Vector Machine classifier was able to separate attack traffic from flash crowd more accurately than other classifiers. However, it was not appropriate to be used as a real-time intrusion-detection system, since it took more than 2 minutes to run the analysis on a dataset of 4527 instances. Similarly, although Self Organizing Map was able to distinguish normal traffic from attack accurately, it required more than 10 minutes to produce the results which would again be unacceptable in online environment.

6.3 Analysis and Discussion

The study has introduced HTTP/2 flash-crowd traffic (Chapter 4) and attack traffic models (Chapter 5). The flash crowd traffic was generated by having 5200 simulated normal end users simultaneously visiting an HTTP/2 server (Subsection 4.4.2); and the attack traffic was obtained by flooding the server with HTTP/2 window_update packets (Section 5.1). This flood of HTTP/2 packets could bypass a hypothetical intrusion-detection system that monitored the CPU and memory consumption of a machine (Section 5.2), and degraded classification performance to encumber differentiation between attack and flash-crowd traffic (Section 6.1 to 6.2). It was demonstrated that 2 and 4 bots were able to yield stealthier traffic than DDoS attacks. Hence, HTTP/2 flood attacks were found to be more efficient than HTTP/1.1 Request floods that required a substantially large amount of HTTP packets and bots to incapacitate the server.

This section also discusses the comparison of HTTP/2 flood attack performance with that of HTTP/1.1 from other viewpoints. It compares the importance of certain determining factors to classify traffic (Section 6.3.1) and the classification performance when using only the HTTP/1.1 features (Section 6.3.2).

6.3.1 Feature Ranking Comparison with HTTP/1.1 Features

Current research on DDoS attack classification explores methods to distinguish attack traffic from normal or flash-crowd traffic operating in an HTTP/1.1 environment. In contrast, this study operated in HTTP/2 environment to classify normal from attack traffic. This section also discusses how the HTTP/1.1 features can be ranked differently as distinguishing factors when applied to HTTP/2 traffic.

To make a comparison, the study investigated features that were used for both HTTP/1.1 and HTTP/2 traffic analysis. Listing these common features was a non-trivial task, because the traffic patterns varied significantly. HTTP/1.1 traffic analysis was reliant on unencrypted packets, while HTTP/2 analysis carried out in this study was applied to encrypted traffic. HTTP/1.1 traffic analysis required deep packet inspection, where unencrypted HTTP packets such as the content of HTTP Requests (Jung et al., 2002; Oikonomou & Mirkovic, 2009; Saleh & Abdul Manaf, 2015; Zhou et al., 2014), its distribution (Bhatia, Mohay, Tickle, & Ahmed, 2011), or flow (Sachdeva & Kumar, 2014) could be analysed. Similarly, the number of resources (i.e. files or pages) that clients requested from a server was one of the features to model flash-crowd traffic (Bhatia, Mohay, Schmidt, & Tickle, 2012). Again, these solutions relied on unencrypted HTTP data.

As a result, many of the previous methods that inspected HTTP/1.1 traffic were no longer applicable when implemented for encrypted traffic analysis of HTTP/2. This is because encrypted traffic conceals the content of the application-layer data. Hence, browsing behaviour that depended on HTTP data and packet flow measurements, and required application-data inspection could not be analysed through these techniques. Consequently, deep packet inspection methods and features that relied on certain HTTP/1.1 packet flow observations as described above could not be applied to analyse encrypted HTTP/2 traffic.

However, certain aspects were adoptable from HTTP/1.1 traffic analysis and applied to HTTP/2 traffic. Both HTTP protocols required IP headers (explained on page 15) to deliver client to server HTTP messages. The IP header was not commonly encrypted in HTTP/2 unless for tunnelling purposes (where client to server IP headers were encrypted to obscure client-server IP addresses). Hence, IP header information was valuable analysis. Traditionally, studies observed the entropy of the IP address and the port number as the distinguishing features to detect DDoS attacks against HTTP/1.1 services (Kumar, Joshi, & Singh, 2007; Lakhina et al., 2005). The disadvantage of this solution was that IP address spoofing, or forging the source IP address, has increasingly been a common practice adopted by adversaries. Attackers could mimic the statistical properties of legitimate traffic and bypass detection methods. Therefore, these features alone are insufficient for categorizing DDoS attack traffic.

A more recent approach (Rahmani et al., 2012) applied statistical components of the network headers, i.e. the number of connections and the number of packets, to define a determining factor using joint-entropy. Joint entropy is a measure of uncertainty of values from a pair of variables. The method showed high coherence between any two factors in a flash-crowd event, while attacks yielded a deviation from some values that indicated coherency. The study clarified that it only inspected the IP header field which did not require deep packet inspections. However, as previously explained, IP header values can be spoofed and the distribution of the values could be made to mimic flash-crowd properties, making it harder to distinguish.

The features used in the above HTTP/1.1 DDoS detection methods that inspected IP headers could be applied to compare the stealthiness of the HTTP/2 attack traffic proposed in this study. These HTTP/2 features were the `count_app` and the `count_syn` features, representing the two HTTP/1.1 features, i.e. the number of packets and the number of connections, respectively.

Other than these sets of features used for the study, the HTTP/2 traffic analysis in this thesis applied machine learning techniques, while the previous study (Rahmani et al., 2012) used joint-entropy. In this case, the advantage of using machine learning is that a range of features could be compared and ranked according to their relevance to detecting and differentiating attack traffic.

The results are presented in Tables 6.5 and 6.6, which summarize the work presented in from the previous sections. The two HTTP/1.1 traffic features are shown at the top because they were the most distinguishing factors for distinguishing HTTP/1.1 DDoS from flash-crowd traffic. When applied to analyse HTTP/2 traffic, it can be seen that the count_syn feature (highlighted) deviated significantly from the top ranked feature. This was true for both Information Gain (Table 6.5) and Gain Ratio-based feature ranking (Table 6.6).

Furthermore, Tables 6.5 and 6.6 showed that the count_app feature (highlighted), remained a highly relevant feature as it was still ranked near the top for all three HTTP/2 traffic types, i.e. the DDoS, Stealthy Attack-1 and Stealthy Attack-2. This confirmed that the Denial-of-Service attack demonstrated in this study was categorized accurately as a flooding-based attack, and the HTTP/2

Table 6.5: Feature ranks using Information Gain for all 3 attack types

HTTP/1.1	HTTP/2		
	DDoS	Stealthy Attack 1	Stealthy Attack 2
count_app	count_app	size_rstAck	size_tlsKey
count_syn	size_ack	count_rstAck	count_tlsKey
	size_syn	count_app	count_app
	count_syn	size_tlsKey	size_app
	size_tlsHello	size_tlsHello	size_tlsHello
	count_tlsHello	count_tlsKey	size_rstAck
	size_app	size_app	count_rstAck
	count_ack	count_syn	lapse_encAlert_max
	count_rstAck	size_syn	lapse_rstAck_max
	size_rstAck	count_tlsHello	lapse_rst_max
	count_encAlert	count_encAlert	lapse_finAck_max
	size_encAlert	size_encAlert	size_syn
	size_tlsKey	size_finAck	count_syn

Table 6.6: Feature ranks using Gain Ratio for all 3 attack types

HTTP/1.1	HTTP/2		
	DDoS	Stealthy Attack 1	Stealthy Attack 2
count_app	count_app	size_rstAck	size_tlsKey
count_syn	count_ack	count_rstAck	size_app
	size_ack	size_app	count_tlsKey
	size_syn	size_tlsKey	count_app
	size_tlsHello	count_app	lapse_tlsHello_max
	count_syn	count_tlsKey	count_rstAck
	count_tlsHello	count_syn	size_rstAck
	size_app	size_syn	lapse_tlsHello_ave
	size_encAlert	count_tlsHello	lapse_encAlert_max
	count_encAlert	size_encAlert	lapse_rstAck_max
	count_rst	count_encAlert	lapse_rst_max
	size_rst	size_finAck	lapse_finAck_max
	size_rstAck	count_finAck	size_tlsHello
	count_rstAck	lapse_finAck_max	count_encAlert
	count_finAck	lapse_rstAck_max	size_encAlert
	size_finAck	lapse_finAck_ave	size_syn
	lapse_ack_min	lapse_rstAck_ave	count_syn

server was incapacitated due to a high traffic volume from the attack.

This led to further discussions on the external validity of the outcomes achieved. Hence, a comparison with other studies was made to justify how the standard traffic model used in this study, i.e. the synthetically generated flash-crowd traffic, can be differentiated from real flash-crowd traffic. For example, the standard traffic used might be too dense because the flash-crowd traffic was generated using 5,200 clients (Section 4.4.1) while another study used only 80 clients to generate similar traffic volume (Sachdeva & Kumar, 2014). When the HTTP/2 attack traffic was compared to a network with such low traffic density, the attack traffic would yield a much higher number of packets and therefore could be easily distinguishable from flash-crowd. However, the study that used 80 clients also used simulated data; hence, the low number of clients did not mimic real traffic accurately.

Currently there is no real HTTP/2 dataset available. The closest publicly available HTTP/1.1 dataset is the World Cup 98 (*WorldCup98 dataset*, 1998). The data from this dataset was based on more than 3,000 client requests generated per second. Assuming user-browsing time was 15 seconds as in the case of the normal User Model (Section 4.2 page 97), the number of clients in this dataset was equal to 45,000. This number was almost 9 times higher than the 5,200 clients used to generate flash-crowd traffic in this study. If real HTTP/2 traffic characteristics were similar to that, the stealthy traffic proposed in this section (Section 6.1 and 6.2) would cause the count_app feature to become less relevant. Therefore, when a machine learning technique is applied to classify flash-crowd traffic, the stealthy traffic could yield more stealthy properties such as a higher number of incorrectly classified instances, i.e. false negatives.

6.3.2 Performance Comparison

While the previous section identified features used in HTTP/1.1 as determining factors for differentiating flash-crowd from DDoS attack traffic, this section presented how machine learning techniques perform when the traffic was anal-

Table 6.7: Incorrectly classified instances (%) by machine learning techniques applied with only HTTP/1.1 features.

	Stealthy Attack 1	Stealthy Attack 2
Naïve Bayes	0.2651	0.5189
Decision Tree	0.1687	0.2335
JRip	0.1205	0.2335
Support Vector Machine	0	0.3373

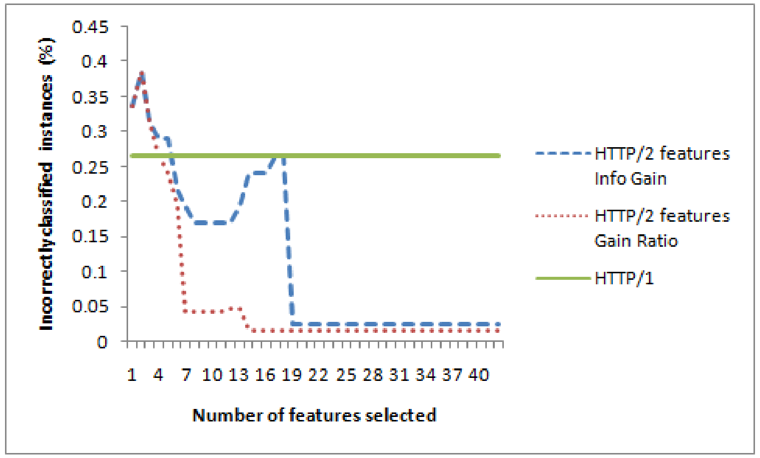
ysed using two HTTP/1.1 features, i.e. the `count_app` and `count_syn` features. Furthermore, it compared the machine learning technique performance analysis when using the two HTTP/1.1 features (the count_app and count_syn features) to the 42 HTTP/2 features proposed in this study (Table 3.6). The discussion also examined the incorrectly classified instances as an evaluation measure.

The machine learning-based analysis results, when using the HTTP/1.1 features as determining factors, is shown in Table 6.7. It can be seen that Stealthy Attack-2 yielded higher percentages of incorrectly classified instances than Stealthy Attack-1. This was another evidence that Stealthy Attack-2 proved to be stealthier than Stealthy Attack-1.

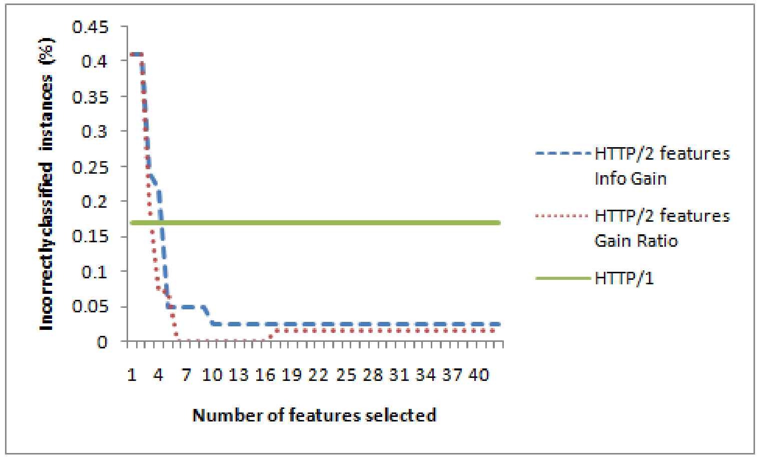
The methodologies for comparison of the results with previously presented attack models (Section 6.1 and 6.2) are shown in Figure 6-20 and 6-21. The former is a comparison to the Stealthy Attack-1 performance, while the latter is a comparison to the Stealthy Attack-2 performance. In the two figures, classification performance using HTTP/1.1-features from Table 6.7 is shown as a straight line in all graphs to serve as a baseline value. The baseline value was to visualize gaps to classification performance results using HTTP/2-features, illustrated in red and blue in the figures. It can be seen from the figures that the green lines are higher, away from the X-axis, than the red and blue graphs. This means that classifying the Stealthy Attack-1 and Stealthy Attack-2 from flash crowd yielded more incorrectly classified instances when employing HTTP/1.1 features than when employing the HTTP/2 features proposed in this study (Table 3.6). The HTTP/2 features yielded better results than the HTTP/1.1 features, when they were employed by machine learning techniques to distinguish HTTP/2 attack traffic from flash crowd traffic.

Another observation drawn from Figures 6-20 and 6-21 is that Stealthy Attack-2 is stealthier than Stealthy Attack-1. When the green lines – which represent performance results using HTTP/1.1. features – were set as a baseline, larger green-red and green-blue gaps are clearly seen on the Stealthy Attack-2 figures (Figure 6-21). That is, the figures show larger gaps between green-red graphs on the Stealthy Attack-2 performance (Figure 6-21) than the green-red graphs on the Stealthy Attack-1 performance (Figure 6-20). Similarly, larger green-blue gaps can be seen on the Stealthy Attack-2 (Figure 6-21) than the Stealthy Attack-1 (Figure 6-20) performance. Therefore, when a hypothetical intrusion-detection-system, equipped with a classifier employing HTTP/1.1 features was used to measure the incorrectly classified instances of the two traffic models proposed in this section (Section 6.1 and 6.2), Stealthy Attack-2 yielded stealthier traffic than Stealthy Attack-1.

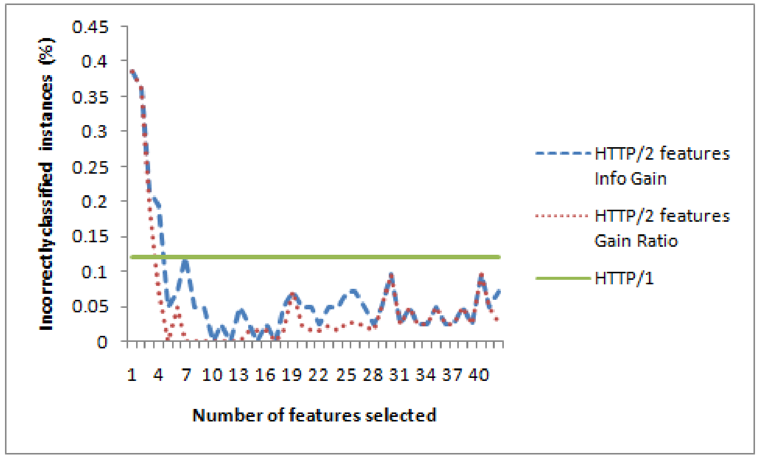
Two conclusions can be drawn from the analysis and discussions in this section. First, Stealthy Attack-2 yielded more incorrectly classified instances than Stealthy Attack-1. The former showed that the count_app and count_syn features were ranked lower than the latter (Section 6.3.1). These features were the two features employed by DDoS detection studies in the literature for HTTP/1.1 traffic. Furthermore, Stealthy Attack-2 yielded high incorrectly classified instances than Stealthy Attack-1 when analysed with HTTP/1.1 features (Section 6.3.2). This demonstrates that Stealthy Attack-2 mimicked flash-crowd traffic closer than Stealthy Attack-1.



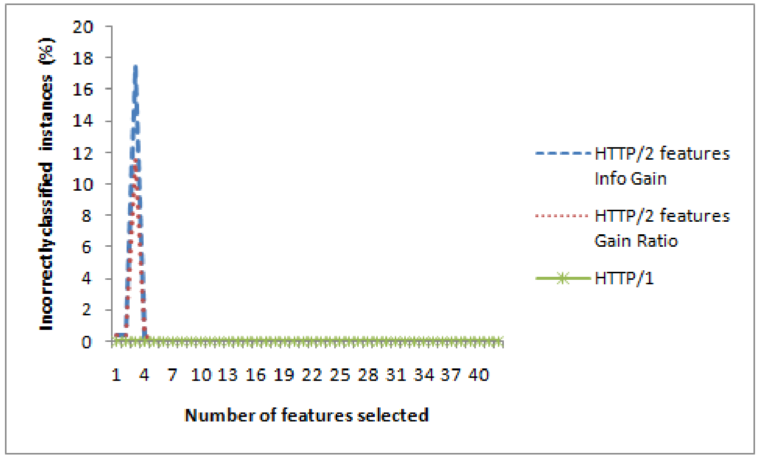
(a) Naïve Bayes



(b) Decision Tree

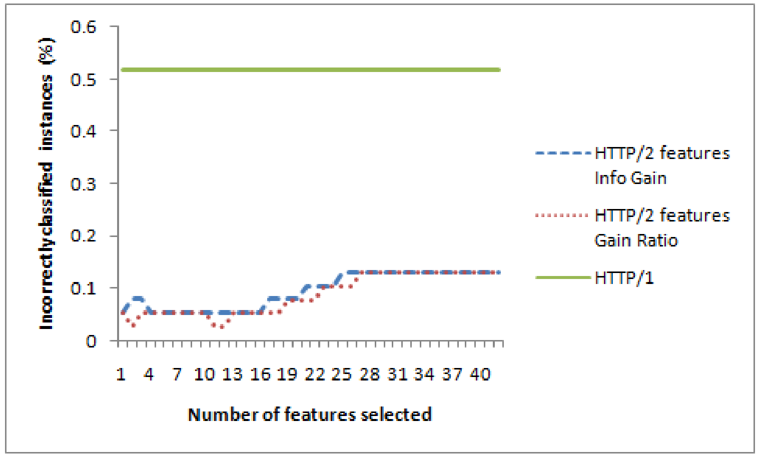


(c) JRip

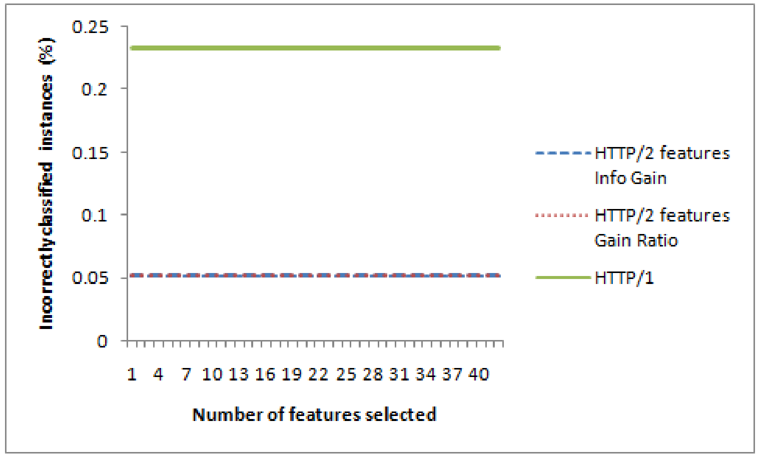


(d) Support Vector Machine

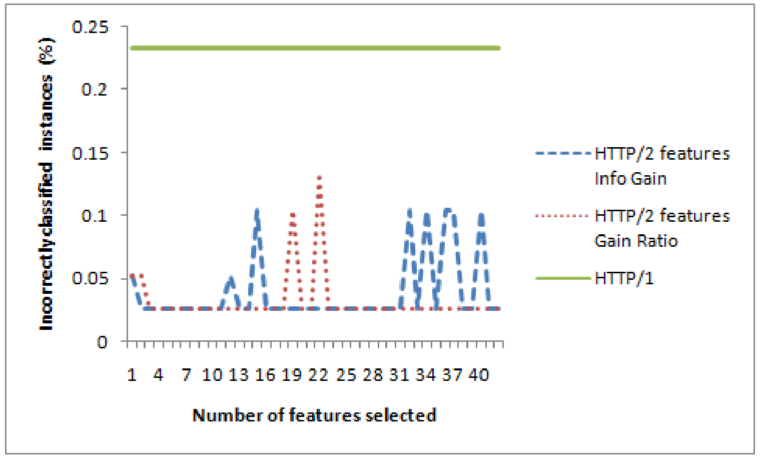
Figure 6-20: A comparison of Stealthy Attack-1 performance when using both HTTP/1.1 features and HTTP/2 features.



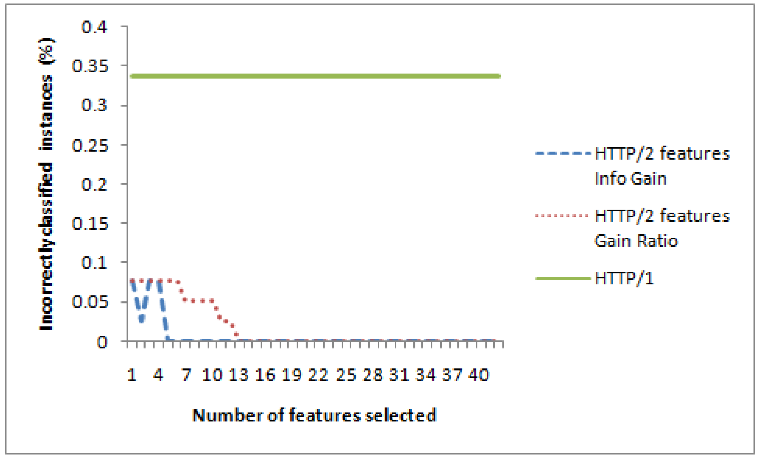
(a) Naïve Bayes



(b) Decision Tree



(c) JRip



(d) Support Vector Machine

Figure 6-21: A comparison of Stealthy Attack-2 performance when using both HTTP/1.1 features and HTTP/2 features.

Second, the 42 HTTP/2 features proposed in this study (Table 3.6) were able to yield better results than the HTTP/1.1 features proposed in the literature, in distinguishing attack from flash-crowd traffic. This is shown by the larger gaps between the graphs produced by the Stealthy Attack-2 and Stealthy Attack-1 traffic to the results produced by HTTP/1.1 features (Figure 6-20 and 6-21). This observation was true in all four cases, i.e. when the analysis used Naïve Bayes, Decision Tree, JRip, and Support Vector Machines. This demonstrates that the HTTP/2 features proposed in this study yielded better results than the HTTP/1.1 features employed in the literature.

6.4 Conclusion

This section introduced two sets of HTTP/2 attack traffic, namely, Stealthy Attack-1 and Stealthy Attack-2. It was shown that these traffic types were stealthier than the DDoS traffic presented in the previous chapter. The evaluation presented in the section showed that machine learning techniques performed poorly when deployed to distinguish attack traffic from flash crowd traffic.

Two groups of bots were introduced to generate attack traffic. A mime group was assigned to mimic flash-crowd traffic, and an offending group was designed to generate flooding traffic. Stealthy Attack-1 was modelled to consist of one bot for each group, yielding a total of two bots to generate traffic that continually consumed 100% CPU usage on a target server. Stealthy Attack-2 model consisted of two bots for each group, yielding a total of four bots. These numbers, i.e. two bots for Stealthy Attack-1 and four bots for Stealthy Attack-2, were significantly lower than 5200, the number of clients deployed to generate flash-crowd traffic. Since two to four machines can generate attack traffic, HTTP/2 DDoS attacks can become more ubiquitous in the future. These can be launched from an attacker's place of residence as opposed to traditional DDoS attack methods that required distributed computers to launch simultaneous attack floods targeting a victim.

Three analyses conducted showed that Stealthy Attack-2 was stealthier than Stealthy Attack-1. *First*, machine learning-based classifications employing

HTTP/2 features proposed in this study (Table 3.6) showed that Stealthy Attack-2 yielded higher percentages of incorrectly classified instances than Stealthy Attack-1 (Figure 6-15). A hypothetical intrusion-detection system would perform worse when Stealthy Attack-2 traffic model was chosen rather than Stealthy Attack-1. *Second*, Stealthy Attack-2 did not reach 0% incorrectly classified instances when analysed with Naïve Bayes, Decision Tree, and JRip (Figure 6-16). In contrast, Stealthy Attack-1 reached 0% incorrectly classified instances when analysed with Decision Tree and JRip. *Third*, Stealthy Attack-2 showed higher incorrectly classified instances than Stealthy Attack-1 when HTTP/1.1 features (i.e. count_app and count_syn) were used as distinguishing factors to classify attack and flash-crowd (Figure 6-20 and 6-21). Thus, a hypothetical intrusion-detection-system that was equipped with machine learning classifiers analysing HTTP/1.1 traffic would perform worse when Stealthy Attack-2 traffic model was chosen rather than Stealthy Attack-1, to generate attack traffic.

Support Vector Machines and Self Organizing Maps can be used to classify HTTP/2 attack from flash-crowd traffic. Support Vector Machine yielded 0% incorrectly classified instances (Figure 6-16), 100% Detection Rate (Figure 6-17), and 0% False Alarm Rate (Figure 6-18) when classifying all three attack models presented in this study, i.e. DDoS, Stealthy Attack-1, and Stealthy Attack-2. Similarly, a Self Organizing Map-based visualization (Figure 6-19) showed that the cluster representing flash-crowd traffic was coloured uniformly, indicating that the Self Organizing Map accurately clustered normal and attack traffic. However, both Support Vector Machine and Self Organizing Map took a very long execution time to produce analysis results. Support Vector Machines spent more than 2 minutes, while Self Organizing Map spent more than 10 minutes when analysing each of the datasets consisting of between 3,700 to 4,400 instances. The analysis was run on an Intel Core-i3 with a 2 GB RAM. Therefore, it can be seen that Support Vector Machines and Self Organizing Maps were not suitable techniques for real-time intrusion-detection systems.

The study further showed how features used in HTTP/1.1 traffic were not highly relevant when applied to analyse HTTP/2 traffic. The HTTP/1.1 features employed in the literature, i.e. the count_app and the count_syn features, ranked as less relevant features when they were employed to analyse HTTP/2 traffic (Table 6.5 and 6.6). The tables showed that the two features were not ranked at the top of the list. On the other hand, a hypothetical intrusion-detection system would perform better when it employs HTTP/2 features proposed in this study (Table 3.6). Figure 6-20 and 6-21 show that graphs representing classification performance employing HTTP/2 features were lower than the curves representing classification performance employing HTTP/1.1 features. Thus, the proposed HTTP/2 features of this study contributed to better classification performance.

Chapter 7

Conclusion

Denial of Service (DoS) attacks are known to disrupt routine Internet services that modern society benefits from. Detecting and having the ability to prevent DoS attacks against a web server have been widely studied in the literature. Currently, research on detection of DoS attacks against web servers as found in the literature is associated with HTTP/1.1 traffic. The HTTP/1.1 protocol has been the global web communication standard for nearly two decades, and the new version of this standard, namely HTTP/2, was published very recently, i.e. in May 2015. This thesis work investigated, modelled and analysed flooding-based DoS attacks against HTTP/2 services. The study presented HTTP/2 normal traffic model (Chapter 4) and various attack models (Chapter 5 and 6), to illustrate the mechanism adopted by the adversary for launching such attacks against web services. In addition, a thorough analysis was performed based on machine learning techniques so as to differentiate attack traffic from legitimate. A proposal of a stealthier version of the DoS attack was presented, to encumber the detection process. The findings reported in this thesis demonstrate how HTTP/2 attack traffic can be modelled; identify future directions of work to extend the proposed models; and allow follow-up studies in traffic analysis and identification as active research areas.

7.1 Contributions of the Study

The main contributions of this study are outlined as follows:

- An HTTP/2 legitimate traffic model was defined and analysed (Chapter 4).
- Four HTTP/2 attack traffic models were defined and analysed (Chapter 5 and 6).
- HTTP/2 traffic features (Table 3.5) were analysed and ranked based on known feature ranking techniques.
- HTTP/2 normal and attack traffic datasets (Section 4.4.2, 5.2, 6.1, and 6.2).

This study introduced four HTTP/2 DoS attack models, i.e. flooding-based attacks (Section 5.1), DDoS attacks (Section 5.2), Stealthy Attack-1 (Section 6.1) and Stealthy Attack-2 (Section 6.2). The first two models, flooding-based and DDoS, demonstrated how HTTP/2 traffic can be modelled to generate attack traffic that incapacitates a target machine through sheer traffic intensity. The traffic generated based on these four defined models caused a target machine to reach 100% CPU consumption. In addition, the two traffic models were distinguishable from legitimate traffic. Machine learning techniques were subsequently applied to classify the flooding-based and DDoS traffic from legitimate traffic. Following this, the latter two models, Stealthy Attack-1 and Stealthy Attack-2, were formulated to impair the performance of machine learning classifiers from accurately differentiating legitimate from attack traffic. These two traffic models caused machine learning classifiers to show higher percentages of incorrectly classified instances, lower Detection Rates, and higher False Alarm Rates. The four HTTP/2 attack traffic models presented in this thesis were one of the key contributions of the study.

To show how the latter two attack traffic models (i.e. Stealthy Attack-1 and Stealthy Attack-2) were classified from normal traffic, this study also presented an HTTP/2 legitimate traffic model. The legitimate model was subsequently extended to generate flash-crowd traffic, i.e. traffic generated by legitimate online users, where a high number of users causes a web server to exceed its serving capacity and becomes unresponsive to client requests. The study presented the legitimate model and the generated flash crowd model in Chapter 4.

The legitimate traffic model was built upon a publicly available log of online human actions. The log consisted of user activities, or events, with a time stamp on each event. This allowed the study to represent activities of a normal user through a state transition model (Section 4.2). Different patterns of user actions, representing different user behaviours, were represented by the state transition model. In the study conducted, 21 distinct user behaviours were modelled to generate legitimate traffic. The normal user model definition is one of the contributions of this study.

This model was applied to generate flash-crowd traffic, i.e. a volume of traffic that consumed 100% CPU resource of an HTTP/2 web server. The study found that 5,200 user models were required to operate simultaneously in order to generate flash-crowd traffic. To characterise the generated flash-crowd traffic, this study presented distinguishing factors for traffic identification through a set of network traffic features. Three feature groups were presented, i.e. the count, size, and lapse features. The count and size features identified the number and the size of packets per second, respectively. The lapse feature identified the minimum, average, and maximum time lapse of packets since a TCP connection to transport packets was established. Each feature was used to describe the characteristics of different packet types observed in the generated traffic in this study. There were a total of 42 features presented in Table 3.6. The HTTP/2-feature set is one of the contributions of this study.

This set of features was applied to characterise the flash-crowd traffic generated in this study. The characteristics of the flash-crowd traffic are shown in Table 4.7. The values of these features were employed as the legitimate traffic; values significantly different from these feature values indicated traffic anomalies. Hence, the legitimate traffic features were used to identify attack traffic presented in this study.

The traffic generated by the first attack model, the flooding-based one, caused a victim machine to become unresponsive to client requests. This signified that the attack successfully incapacitated the victim machine. The flooding-based attack traffic caused a victim machine to show 100% CPU consumption. This model employed an HTTP/2 packet type, namely, window_update to flood a target machine.

The second attack model, DDoS model, extended the flooding-based model to generate stealthy traffic, i.e. a traffic flood that bypassed a hypothetical intrusion-detection-system that monitored CPU and memory consumption of a victim machine. The study found that when the window_update payload, i.e. window-size-increment was set to 16,384, then a flood of 131K window_update packets sent within 38.5 seconds consumed 50% CPU of a target machine. This CPU load did not incapacitate a target machine, thereby allowing the traffic to bypass the hypothetical intrusion-detection system previously described. The DDoS traffic was generated by four attacking clients, where each client launched 2 attack traffic volumes, based on the above defined models of stealthy traffic. The generated traffic caused a target machine to consume 100% CPU utilisation. However, the traffic was distinguished from legitimate traffic. Hence, the study introduced a third model to investigate how stealthy attacks operate.

The third attack model, Stealthy Attack-1, extended the DDoS model to camouflage attack traffic characteristics. Two bots were employed to mimic the count_syn feature values of attack traffic. This feature was the number of TCP packets with the SYN flag set, observed per second. The traffic was analysed through employing four machine learning techniques (Naïve Bayes, Decision Tree, JRip, and Support Vector Machines). The analysis showed that Stealthy Attack-1 traffic produced more incorrectly classified instances than the DDoS traffic, suggesting that the Stealthy Attack-1 traffic is stealthier than the DDoS model.

The fourth attack model, Stealthy Attack-2, extended the Stealthy Attack-1 model to demonstrate how stealthier attack traffic can be modelled. Four bots were employed to mimic the size_rstAck feature values of attack traffic. This feature defined the size of TCP packets with RST-ACK flag set, observed per second. Three machine learning techniques, i.e. Naïve Bayes, Decision Tree and JRip, showed that Stealthy Attack-2 produced more incorrectly classified instances than Stealthy Attack-1, suggesting that the former is stealthier than the latter.

The study further showed how features used in HTTP/1.1 traffic were not highly relevant when used to analyse HTTP/2 traffic. Instead, the analysis of the two HTTP/2 stealthy attack traffic models, Stealthy Attack-1 and Stealthy Attack-2, performed better when the HTTP/2 features (Table 3.6) were used as the distinguishing factors. Machine learning classifiers (Naïve Bayes, Decision Tree, JRip, and Support Vector Machines) yielded less percentage of incorrectly classified instances when applying HTTP/2 features than HTTP/1.1 features.

The study demonstrated that two machine learning techniques, Support Vector Machines and Self Organizing Maps, were able to distinguish attack and legitimate traffic with a high degree of accuracy. Support Vector Machines can classify the two traffic types when at least 5 Information Gain-ranked features or 13 Gain Ratio-ranked features were selected, yielding 0% incorrectly classified instances. Self Organizing Maps showed flash-crowd traffic cluster in a colour code different from other traffic clusters, suggesting that the traffic is successfully differentiated between the two classes. However, it took more than 2 minutes and 10 minutes, respectively, to run the analysis of Support Vector Machines and Self Organizing Maps. Hence, these machine learning techniques were not suitable for real-time applications such as intrusion-detection systems.

The study indicated that DoS attack analysis will remain to be an active research area with the ready-adoption of the HTTP/2 standard. The study demonstrated that a small number of machines can be designed to produce HTTP/2 DoS attack traffic. It took only 2 machines to disrupt an HTTP/2 service as opposed to an estimated number of 45,000 clients required for HTTP/1.1 DDoS attack traffic generation. The use of more bots, such as 4 bots that the Stealthy Attack-2 modelled, could camouflage more attack properties with legitimate properties. Future studies that employ higher number of bots can create further similarities between attack and normal traffic.

7.2 Limitations and Future Work

Two limitations are identified in this study. First, the legitimate traffic dataset was obtained from synthetic data. Further HTTP/2 DoS traffic analysis can benefit from actual HTTP/2 flash-crowd traffic datasets when they become available. Second, the attack models were based on one of the HTTP/2 frame types, the `window_update` packet. Overall there are 10 frame types which can be combined to produce different traffic patterns. These combinations were outside the scope of this study and serve as future work.

Synthetically-generated traffic was set as the baseline standard for machine learning techniques to analyse the four attack traffic models presented in this study. The legitimate traffic was generated through the implementation of curl, a programming library that generated HTTP/2 traffic. Real web traffic is produced by heterogeneous devices and implementations, which altogether create legitimate traffic patterns. These patterns can show different feature values when compared to the ones used in this study. Future work in HTTP/2 DoS attack design and analysis can benefit from actual HTTP/2 flash-crowd traffic.

Another limitation of this study is that only one of 10 HTTP/2 frame types, i.e. `window_update`, was exploited to serve as the anatomy of the proposed attack models. The study observed that a flood of HTTP/2 packets consisted of other frame types such as `ping`, `data`, and `settings` packets did not successfully incapacitate a target server. The rest of the frame types were outside the scope of this study; these are: the `headers`, `priority`, `rst_stream`, `push_promise`, `go_away` and `continuation` frames. It can be seen that HTTP/2 attack models were endowed with different techniques than its predecessor, one of which was to examine how these frame types can be employed. Combinations of these frame types can produce a range of traffic patterns that have not been studied.

This study has contributed in creating novel, HTTP/2 legitimate and attack datasets. As Internet-connected devices and its heterogeneity are projected to increase significantly in the future, research focusing on creating, collecting and synthesizing current Internet traffic datasets will continue to extend knowledge published through this research work. On the other hand, HTTP/2 introduces new techniques capable of generating various traffic patterns, which have been shown in this thesis to produce DoS attack traffic of varying characteristics. These can further create a race amongst state-of-the art HTTP/2 DoS studies in attack design, analysis, and detection.

References

Addley, E., & Halliday, J. (2010). *Wikileaks supporters disrupt Visa and MasterCard sites in 'operation payback'* [Web Page]. Guardian News and Media Limited. Retrieved from `http://www.theguardian.com/world/2010/dec/08/wikileaks-visa-mastercard-operation-payback` (Accessed: 2016-09-12)

Adi, E., Baig, Z., Lam, C. P., & Hingston, P. (2015). Low-rate denial-of-service attacks against HTTP/2 services. In *IT Convergence and Security (ICITCS), 2015 5th International Conference on* (pp. 1–5). IEEE.

Adi, E., Baig, Z. A., Hingston, P., & Lam, C. P. (2016). Distributed denial-of-service attacks against HTTP/2 services. *Cluster Computing*, *19*(1), 79–86. doi: 10.1007/s10586-015-0528-7

Agrawal, P., Gupta, B., & Jain, S. (2011). SVM based scheme for predicting number of zombies in a DDoS attack [Conference Proceedings]. In *European Intelligence and Security Informatics Conference (EISIC)* (p. 178-182). IEEE.

Al-Jarrah, O., & Arafat, A. (2014). Network intrusion detection system using attack behavior classification [Conference Proceedings]. In *Information and Communication Systems (ICICS), 2014 5th International Conference on* (p. 1-6). IEEE.

Al-Jarrah, O., Siddiqui, A., Elsalamouny, M., Yoo, P., Muhaidat, S., & Kim, K. (2014). Machine-learning-based feature selection techniques for large-scale network intrusion detection [Conference Proceedings]. In *Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on* (p. 177-181). IEEE.

Auld, T., Moore, A. W., & Gull, S. F. (2007). Bayesian neural networks for internet traffic classification [Journal Article]. *Neural Networks, IEEE Transactions on*, *18*(1), 223-239.

Baig, Z. A., Sait, S. M., & Shaheen, A. (2013). GMDH-based networks for intelligent intrusion detection [Journal Article]. *Engineering Applications of Artificial Intelligence*, *26*(7), 1731-1740.

Barford, P., Kline, J., Plonka, D., & Ron, A. (2002). A signal analysis of network traffic anomalies [Conference Proceedings]. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment* (p. 71-82). ACM.

Barthakur, P., Dahal, M., & Ghose, M. K. (2013). An efficient machine learning based classification scheme for detecting distributed command & control traffic of P2P botnets [Journal Article]. *International Journal of Modern*

Education and Computer Science (IJMECS), *5*(10), 9.

Belshe, M., Peon, R., & Thomson, M. (May 2015). *Hypertext Transfer Protocol version 2 (HTTP/2)* (Report No. RFC 7540). Internet Engineering Task Force (IETF).

Bernaille, L., Teixeira, R., Akodkenou, I., Soule, A., & Salamatian, K. (2006). Traffic classification on the fly [Journal Article]. *ACM SIGCOMM Computer Communication Review*, *36*(2), 23-26.

Berners-Lee, T., Fischetti, M., & Foreword By-Dertouzos, M. L. (2000). *Weaving the web: The original design and ultimate destiny of the world wide web by its inventor.* Harper Information.

Bhatia, S., Mohay, G., Schmidt, D., & Tickle, A. (2012). Modelling web-server flash events. In *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on* (pp. 79–86). IEEE.

Bhatia, S., Mohay, G., Tickle, A., & Ahmed, E. (2011). Parametric differences between a real-world distributed denial-of-service attack and a flash event. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on* (pp. 210–217).

Bhattacharyya, D. K., & Kalita, J. K. (2013). *Network anomaly detection: A machine learning perspective* [Book]. CRC Press.

Bivens, A., Palagiri, C., Smith, R., Szymanski, B., & Embrechts, M. (2002). Network-based intrusion detection using neural networks [Journal Article]. *Intelligent Engineering Systems through Artificial Neural Networks*, *12*(1), 579-584.

Bradner, S. (1997). *Key words for use in RFCs to indicate requirement levels* (Report No. RFC 2119). Network Working Group.

CERT. (1996). *CERT Advisory CA-1996-26: Denial-of-Service attack via ping* [Web Page]. Carnegie Mellon University. Retrieved from `http://www.cert.org/historical/advisories/CA-1996-26.cfm` (Accessed: 2016-10-14)

CERT. (1997). *Denial of service attacks* [Web Page]. Carnegie Mellon University. Retrieved from `https://www.cert.org/information-for/denial_of_service.cfm?` (Accessed: 2016-09-06)

Chan, E. Y., Chan, H., Chan, K., Chan, P., Chanson, S. T., Cheung, M., ... Hui, L. C. K. (2006). Intrusion detection routers: Design, implementation and evaluation using an experimental testbed [Journal Article]. *Selected Areas in Communications, IEEE Journal on*, *24*(10), 1889-1900.

Chang, R. K. (2002). Defending against flooding-based Distributed Denial-of-Service attacks: a tutorial [Journal Article]. *Communications Magazine, IEEE*, *40*(10), 42-51.

Chen, Y., Das, S., Dhar, P., El-Saddik, A., & Nayak, A. (2008). Detecting and preventing IP-spoofed distributed DoS attacks [Journal Article]. *IJ Network Security*, *7*(1), 69-80.

Choi, J., Choi, C., Ko, B., & Kim, P. (2014). A method of DDoS attack detection using HTTP packet pattern and rule engine in cloud computing environment [Journal Article]. *Soft Computing*, 1-7.

Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 115–123).

Combs, G. (1998–2015). *Wireshark (version 2.0.1)* [Software]. Retrieved from `https://www.wireshark.org/`

Common vulnerability and exposures: The standard for information security vulnerability names [Online Database]. (2016). The Mitre Corporation. Retrieved from `http://cve.mitre.org/cgibin/cvekey.scgi?keyword=HTTP+denial+of+service`

Corchado, E., & Herrero, l. (2011). Neural visualization of network traffic data for intrusion detection [Journal Article]. *Applied Soft Computing*, *11*(2), 2042-2056.

Crosby, S. A., & Wallach, D. S. (2003). Denial of Service via algorithmic complexity attacks [Conference Proceedings]. In *USENIX Security* (Vol. 2).

Dainotti, A., Pescape, A., & Ventre, G. (2006). NIS04-1: Wavelet-based detection of DoS attacks [Conference Proceedings]. In *Global Telecommunications Conference, 2006 (GLOBECOM'06)* (p. 1-6). IEEE.

Dear, B. (2010). *Perhaps the first Denial-of-Service attack?* [Web Page]. PLATO History Foundation. Retrieved from `http://www.platohistory.org/blog/2010/02/perhaps-the-first-denial-of-service-attack.html` (Accessed: 2016-10-14)

Dyer, K. P., Coull, S. E., Ristenpart, T., & Shrimpton, T. (2012). Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail [Conference Proceedings]. In *Security and Privacy (SP), 2012 IEEE Symposium on* (p. 332-346). IEEE.

Elbasiony, R. M., Sallam, E. A., Eltobely, T. E., & Fahmy, M. M. (2013). A hybrid network intrusion detection framework based on random forests and weighted k-means [Journal Article]. *Ain Shams Engineering Journal*, *4*(4), 753-762.

Erman, J., Mahanti, A., Arlitt, M., & Williamson, C. (2007). Identifying and discriminating between web and Peer-to-Peer traffic in the network core [Conference Proceedings]. In *Proceedings of the 16th International Conference on World Wide Web* (p. 883-892). ACM.

Estonian attacks raise concern over cyber 'nuclear winter' [Web Page]. (2007, May 24). Information Week. Retrieved from `http://www.informationweek.com/estonian-attacks-raise-concern-over-cyber-nuclear-winter/d/d-id/1055474?` (Accessed: 2016-09-07)

Fabian, M., & Terzis, M. A. (2007). My botnet is bigger than yours (maybe, better than yours): Why size estimates remain challenging. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Understanding Botnets, Cambridge, USA.*

Feinstein, L., Schnackenberg, D., Balupari, R., & Kindred, D. (2003). Statistical approaches to DDoS attack detection and response [Conference Proceedings]. In *DARPA Information Survivability Conference and Exposition, 2003. Proceedings* (Vol. 1, p. 303-314). IEEE.

Ferguson, P. (2000). *Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing* (Report No. RFC 2267). Network

Working Group.

Gaonjur, P., Tarapore, N., Pukale, S., & Dhore, M. (2008). Using neuro-fuzzy techniques to reduce false alerts in IDS. In *2008 16th IEEE International Conference on Networks* (pp. 1–6).

Garber, L. (2000). Denial-of-Service attacks RIP the Internet [Journal Article]. *Computer*, *33*(4), 12-17.

Garg, S., Singh, A. K., Sarje, A. K., & Peddoju, S. K. (2013). Behaviour analysis of machine learning algorithms for detecting P2P botnets [Conference Proceedings]. In *Advanced Computing Technologies (ICACT), 15th International Conference on* (p. 1-4). IEEE.

Girardin, L. (1999). An eye on network intruder-administrator shootouts [Conference Proceedings]. In *Workshop on Intrusion Detection and Network Monitoring* (p. 19-28).

Gligor, V. D. (2005). Guaranteeing access in spite of distributed service-flooding attacks [Conference Proceedings]. In *Security Protocols* (p. 80-96). Springer.

Gonzalez, J. M., Anwar, M., & Joshi, J. B. (2011). A trust-based approach against IP-spoofing attacks [Conference Proceedings]. In *Privacy, Security and Trust (PST), 2011 Ninth Annual International Conference on* (p. 63-70). IEEE.

Goseva-Popstojanova, K., Anastasovski, G., Dimitrijevikj, A., Pantev, R., & Miller, B. (2014). Characterization and classification of malicious web traffic [Journal Article]. *Computers & Security*, *42*, 92-115.

Grigorik, I. (2013a). *High performance browser networking: What every web developer should know about networking and web performance* [Book]. O'Reilly Media, Inc.

Grigorik, I. (2013b). Making the web faster with HTTP 2.0 [Journal Article]. *Communications of the ACM*, *56*(12), 42-49.

Guo, F., Krishnan, R., & Polak, J. (2012). Short-term traffic prediction under normal and incident conditions using singular spectrum analysis and the k-nearest neighbour method [Conference Proceedings]. In *Road Transport Information and Control (RTIC 2012), IET and ITS Conference on* (p. 1-6). IET.

Haddadi, F., Morgan, J., & Zincir-Heywood, A. N. (2014). Botnet behaviour analysis using IP flows: With HTTP filters using classifiers [Conference Proceedings]. In *Advanced Information Networking and Applications Workshops (WAINA), 28th International Conference on* (p. 7-12). IEEE.

Haykin, S., & Lippmann, R. (1994). Neural networks, a comprehensive foundation [Journal Article]. *International Journal of Neural Systems*, *5*(4), 363-364.

Heron, S. (2010). Denial of Service: Motivations and trends [Journal Article]. *Network Security*, *2010*(5), 10-12.

Hubballi, N., & Suryanarayanan, V. (2014). False alarm minimization techniques in signature-based intrusion detection systems: A survey. *Computer Communications*, *49*, 1–17.

Igure, V., & Williams, R. (2008). Taxonomies of attacks and vulnerabilities in computer systems [Journal Article]. *Communications Surveys & Tutorials,* *IEEE*, *10*(1), 6-19.

The internet of things [Magazine Article]. (2014, July 12). *The Economist*.

Juels, A., & Brainard, J. G. (1999). Client puzzles: A cryptographic countermeasure against connection depletion attacks [Conference Proceedings]. In *NDSS* (Vol. 99, p. 151-165).

Jung, J., Krishnamurthy, B., & Rabinovich, M. (2002). Flash crowds and Denial of Service attacks: Characterization and implications for CDNs and web sites [Conference Proceedings]. In *Proceedings of the 11th International Conference on World Wide Web* (p. 293-304). ACM.

Kandula, S., Katabi, D., Jacob, M., & Berger, A. (2005). Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds [Conference Proceedings]. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2* (p. 287-300). USENIX Association.

Karagiannis, T., Papagiannaki, K., & Faloutsos, M. (2005). BLINC: Multilevel traffic classification in the dark [Conference Proceedings]. In *ACM SIGCOMM Computer Communication Review* (Vol. 35, p. 229-240). ACM.

Katkar, V. D., & Kulkarni, S. V. (2013). Experiments on detection of denial of service attacks using naive bayesian classifier [Conference Proceedings]. In *Green Computing, Communication and Conservation of Energy (ICGCE), International Conference on* (p. 725-730). IEEE.

Keane, J. (2015, August). *DDoS attack hit record numbers in Q2 2015.* Digital Trends. Retrieved from `http://www.digitaltrends.com/computing/ddos-attacks-hit-record-numbers-in-q2-2015/`

Khandelwal, S. (2016, January). *602 Gbps! This may have been the largest DDoS attack in history.* The Hacker News. Retrieved from `http://thehackernews.com/2016/01/biggest-ddos-attack.html`

Kim, H., Claffy, K. C., Fomenkov, M., Barman, D., Faloutsos, M., & Lee, K. (2008). Internet traffic classification demystified: Myths, caveats, and the best practices [Conference Proceedings]. In *Proceedings of the 2008 ACM CoNEXT conference* (p. 11). ACM.

Kitten, T. (2013, January 14). *DDoS: Lessons from phase 2 attacks* [Web Page]. Information Security Media Group, Corp. Retrieved from `http://www.bankinfosecurity.com/ddos-attacks-lessons-from-phase-2-a-5420/op-1` (Accessed: 2016-09-06)

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps [Journal Article]. *Biological cybernetics*, *43*(1), 59-69.

Kotsiantis, S. B., Zaharakis, I. D., & Pintelas, P. E. (2006). Machine learning: A review of classification and combining techniques. *Artificial Intelligence Review*, *26*(3), 159–190.

Kumar, K., Joshi, R., & Singh, K. (2007). A distributed approach using entropy to detect DDoS attacks in ISP domain. In *Signal Processing, Communications and Networking, 2007. ICSCN'07. International Conference on* (pp. 331–337).

Labovitz, C. (2010, December 14). *The Internet goes to war* [Web Page]. Arbor Networks, Inc. Retrieved from `http://www.arbornetworks.com/asert/2010/12/the-internet-goes-to-war/`

Lakhina, A., Crovella, M., & Diot, C. (2005). Mining anomalies using traffic feature distributions. In *ACM SIGCOMM Computer Communication Review* (Vol. 35, pp. 217–228).

Li, Z., Yuan, R., & Guan, X. (2007). Accurate classification of the Internet traffic based on the SVM method [Conference Proceedings]. In *Communications, 2007. ICC'07. IEEE International Conference on* (p. 1373-1378). IEEE.

Liu, H., Zhang, Y., Lin, H., Wu, J., Wu, Z., & Zhang, X. (2013). How many zombies around you? [Conference Proceedings]. In *Data Mining (ICDM), IEEE 13th International Conference on* (p. 1133-1138). IEEE.

Loukas, G., Gan, D., & Vuong, T. (2013). A taxonomy of cyber attack and defence mechanisms for emergency management networks [Conference Proceedings]. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on* (p. 534-539). IEEE.

Lu, L. F., Huang, M. L., Orgun, M. A., & Zhang, J. W. (2010). An improved wavelet analysis method for detecting DDoS attacks [Conference Proceedings]. In *Network and System Security (NSS), 2010 4th International Conference on* (p. 318-322). IEEE.

Lu, W., & Ghorbani, A. A. (2009). Network anomaly detection based on wavelet analysis [Journal Article]. *EURASIP Journal on Advances in Signal Processing*, *2009*, 4.

Ma, X., & Chen, Y. (2014). DDoS detection method based on chaos analysis of network traffic entropy. *IEEE Communications Letters*, *18*(1), 114–117.

Malialis, K., & Kudenko, D. (2013). Large-scale DDoS response using cooperative reinforcement learning [Conference Proceedings]. In *11th European Workshop on Multi-Agent Systems (EUMAS).*

Mansfield-Devine, S. (2011). DDoS: Threats and mitigation [Journal Article]. *Network Security*, *2011*(12), 5-12.

Meng, W., Li, W., & Kwok, L.-F. (2014). EFM: Enhancing the performance of signature-based network intrusion detection systems using eenhanced filter mechanism [Journal Article]. *Computers & Security*, *43*, 189-204.

Meng, Y.-X. (2011). The practice on using machine learning for network anomaly intrusion detection [Conference Proceedings]. In *Machine Learning and Cybernetics (ICMLC), International Conference on* (Vol. 2, p. 576-581). IEEE.

Mirkovic, J., & Reiher, P. (2004). A taxonomy of DDoS atack and DDoS defense mechanisms [Journal Article]. *ACM SIGCOMM Computer Communication Review*, *34*(2), 39-53.

Mizrak, A. T., Savage, S., & Marzullo, K. (2008). Detecting compromised routers via packet forwarding behavior [Journal Article]. *Network, IEEE*, *22*(2), 34-39.

Mohamed, W. N. H. W., Salleh, M. N. M., & Omar, A. H. (2012). A comparative study of reduced error pruning method in decision tree algorithms. In *Control System, Computing and Engineering (ICCSCE), 2012 IEEE International Conference on* (pp. 392–397).

Moore, A. W., & Papagiannaki, K. (2005). Toward the accurate identification of network applications [Book Section]. In *Passive and Active Network Measurement* (p. 41-54). Springer.

Moore, A. W., & Zuev, D. (2005). Internet traffic classification using bayesian analysis techniques [Conference Proceedings]. In *ACM SIGMETRICS Performance Evaluation Review* (Vol. 33, p. 50-60). ACM.

Moustis, D., & Kotzanikolaou, P. (2013). Evaluating security controls against HTTP-based DDoS attacks [Conference Proceedings]. In *Information, Intelligence, Systems and Applications (IISA), 2013 Fourth International Conference on* (p. 1-6). IEEE.

Mukherjee, S., & Sharma, N. (2012). Intrusion detection using naive bayes classifier with feature reduction [Journal Article]. *Procedia Technology*, *4*, 119-128.

Negnevitsky, M. (2005). *Artificial intelligence: A guide to intelligent systems.* Pearson Education.

Nguyen, T. T., & Armitage, G. (2008). A survey of techniques for Internet traffic classification using machine learning [Journal Article]. *Communications Surveys & Tutorials, IEEE*, *10*(4), 56-76.

Ni, T., Gu, X., Wang, H., & Li, Y. (2013). Real-time detection of application-layer DDoS attack using time series analysis [Journal Article]. *Journal of Control Science and Engineering*, *2013*, 4.

Northcutt, S., & Novak, J. (2002). *Network intrusion detection* [Book]. Sams Publishing.

Oikonomou, G., & Mirkovic, J. (2009). Modeling human behavior for defense against flash-crowd attacks. In *Communications, 2009. ICC'09. IEEE International Conference on* (pp. 1–6).

Olszewski, D. (2014). Fraud detection using self-organizing map visualizing the user profiles [Journal Article]. *Knowledge-Based Systems*, *70*, 324-334.

Paganini, P. (2013, May 28). *Dangerous DDoS (Distributed Denial of Service) on the rise* [Web Page]. InfoSec Institute, Inc. Retrieved from `http://resources.infosecinstitute.com/dangerous-ddos-distributed-denial-of-service-on-the-rise/`

Panchev, C., Dobrev, P., & Nicholson, J. (2014). Detecting port scans against mobile devices with neural networks and decision trees [Book Section]. In *Engineering applications of Neural Networks* (p. 175-182). Springer.

Panda, M., Abraham, A., & Patra, M. R. (2015). Hybrid intelligent systems for detecting network intrusions. *Security and Communication Networks*, *8*(16), 2741–2749.

Park, K., & Lee, H. (2001). On the effectiveness of probabilistic packet marking for IP traceback under denial of service attack [Conference Proceedings]. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (Vol. 1, p. 338-347). IEEE.

Paxson, V. (1999). Bro: A system for detecting network intruders in real-time. *Computer networks*, *31*(23), 2435–2463.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: Networks of plausible inference* [Book]. Morgan Kaufmann.

Peng, T., Leckie, C., & Ramamohanarao, K. (2007). Survey of network-based defense mechanisms countering the DoS and DDoS problems [Journal Article]. *ACM Computing Surveys (CSUR)*, *39*(1), 3.

Petkov, V., Rajagopal, R., & Obraczka, K. (2013). Characterizing per-application network traffic using entropy [Journal Article]. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, *23*(2), 14.

Pressman, R. S., & Jawadekar, W. S. (1987). Software engineering [Journal Article]. *New York 1992*.

Rahmani, H., Sahli, N., & Kamoun, F. (2012). Distributed Denial-of-Service attack detection scheme-based joint-entropy [Journal Article]. *Security and Communication Networks*, *5*(9), 1049-1061.

Ramos, V., & Abraham, A. (2005). ANTIDS: Self organized ant-based clustering model for intrusion detection system [Book Section]. In *Soft Computing as transdisciplinary science and technology* (p. 977-986). Springer.

Randall, W. D., & Martinez, T. R. (2000). Reduction techniques for instance-based learning algorithms. *Machine Learning*, *38*(3), 257–286.

Rokach, L., & Maimon, O. (2014). *Data mining with decision trees: Theory and applications*. World scientific.

Ryan, J., Lin, M.-J., & Miikkulainen, R. (1998). Intrusion detection with neural networks. *Advances in Neural Information Processing Systems*, 943–949.

Sachdeva, M., & Kumar, K. (2014). A traffic cluster entropy based approach to distinguish DDoS attacks from flash event using DETER testbed [Journal Article]. *ISRN Communications and Networking*, *2014*.

Salah, K., Sattar, K., Sqalli, M., & Al-Shaer, E. (2011). A potential low-rate DoS attack against network firewalls [Journal Article]. *Security and Communication Networks*, *4*(2), 136-146.

Saleh, M. A., & Abdul Manaf, A. (2015). A novel protective framework for defeating HTTP-based denial of service and distributed denial of service attacks. *The Scientific World Journal*, *2015*.

Savage, S., Wetherall, D., Karlin, A., & Anderson, T. (2000). Practical network support for IP traceback [Conference Proceedings]. In *ACM SIGCOMM Computer Communication Review* (Vol. 30, p. 295-306). ACM.

Sen, S., Spatscheck, O., & Wang, D. (2004). Accurate, scalable in-network identification of P2P level traffic using application signatures [Conference Proceedings]. In *Proceedings of the 13th International Conference on World Wide Web* (p. 512-521). ACM.

Shannon, C. E. (2001). A mathematical theory of communication [Journal Article]. *ACM SIGMOBILE Mobile Computing and Communications Review*, *5*(1), 3-55.

Sharma, A. K., & Parihar, P. S. (2013). An effective DoS prevention system to analysis and prediction of network traffic using support vector machine learning [Journal Article]. *International Journal of Application or Innovation in Engineering & Management*, *2*(7), 249-256.

Siripanadorn, S., Hattagam, W., & Teaumroong, N. (2010). Anomaly detection in wireless sensor networks using self-organizing map and wavelets [Journal Article]. *International Journal of Communications*, *4*(3), 74-83.

Srinivasan, T., Vijaykumar, V., & Chandrasekar, R. (2006). A self-organized agent-based architecture for power-aware intrusion detection in aireless ad-hoc networks [Conference Proceedings]. In *Computing & informatics. icoci'06. international conference on* (p. 1-6). IEEE.

Stebila, D., Kuppusamy, L., Rangasamy, J., Boyd, C., & Nieto, J. G. (2011). Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols [Book Section]. In *Topics in cryptologyâĂŞct-rsa 2011* (p. 284-301). Springer.

Stenberg, D. (1996–2016). *cURL* [Software]. Retrieved from `https://curl.haxx.se/download.html`

Stroeh, K., Madeira, E. R. M., & Goldenstein, S. K. (2013). An approach to the correlation of security events based on machine learning techniques [Journal Article]. *Journal of Internet Services and Applications*, *4*(1), 1-16.

Su, M. Y. (2011). Using clustering to improve the kNN-based classifiers for online anomaly network traffic identification [Journal Article]. *Journal of Network and Computer Applications*, *34*(2), 722-730.

Swamy, K., & Lakshmi, K. V. (2012). Network intrusion detection using improved decision tree algorithm [Journal Article]. *IJCSIS) International Journal of Computer Science and Information Security*, *10*(8).

Tanenbaum, A., & Van Steen, M. (2007). *Distributed systems* [Book]. Pearson Prentice Hall.

Tang, Y., Lin, P., & Luo, Z. (2014). Obfuscating encrypted web traffic with combined objects [Book Section]. In *Information security practice and experience* (p. 90-104). Springer.

Tavallaee, M., Bagheri, E., Lu, W., & Ghorbani, A.-A. (2009). A detailed analysis of the KDD CUP 99 data set. In *Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defence Applications 2009.*

The third great wave [Magazine Article]. (2014, October 4). *The Economist.*

Thomas, B., Jurdak, R., & Atkinson, I. (2012). SPDYing up the web [Journal Article]. *Communications of the ACM*, *55*(12), 64-73.

Triukose, S., Al-Qudah, Z., & Rabinovich, M. (2009). Content delivery networks: Protection or threat? In *European symposium on research in Computer Security* (pp. 371–389).

Tsai, C. F., Hsu, Y. F., Lin, C. Y., & Lin, W. Y. (2009). Intrusion detection by machine learning: A review [Journal Article]. *Expert Systems with Applications*, *36*(10), 11994-12000.

Tsujikawa, T. (2015). *Nghttp2: HTTP/2 C library* [Computer Program]. Retrieved from `https://nghttp2.org/`

University of Waikato. (1993–2016). *Weka (version 3.8)* [Software]. Retrieved from `http://www.cs.waikato.ac.nz/ml/weka/downloading.html`

Vapnik, V. N., & Vapnik, V. (1998). *Statistical learning theory* (Vol. 2) [Book]. Wiley New York.

von der Weth, C., & Hauswirth, M. (2013). DOBBS: Towards a comprehensive dataset to study the browsing behavior of online users. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on* (Vol. 1, pp. 51–56).

Wang, G., Hao, J., Ma, J., & Huang, L. (2010). A new approach to intrusion detection using artificial neural networks and fuzzy clustering [Journal Article]. *Expert Systems with Applications*, *37*(9), 6225-6232.

Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques* [Book]. Morgan Kaufmann.

WorldCup98 dataset. (1998). Retrieved from `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`

Wu, D., Chen, X., Chen, C., Zhang, J., Xiang, Y., & Zhou, W. (2014). On addressing the imbalance problem: A correlated kNN approach for network traffic classification [Book Section]. In *Network and system security* (p. 138-151). Springer.

Wu, S. X., & Banzhaf, W. (2010). The use of computational intelligence in intrusion detection systems: A review [Journal Article]. *Applied Soft Computing*, *10*(1), 1-35.

Wu, X., Kumar, V., Quinlan, J. R., Ghosh, J., Yang, Q., Motoda, H., ... Philip, S. Y. (2008). Top 10 algorithms in data mining. *Knowledge and Information Systems*, *14*(1), 1–37.

Wu, Y. C., Tseng, H. R., Yang, W., & Jan, R. H. (2011). DDoS detection and traceback with decision tree and grey relational analysis [Journal Article]. *International Journal of Ad Hoc and Ubiquitous Computing*, *7*(2), 121-136.

Xie, Y., Tang, S., Xiang, Y., & Hu, J. (2013). Resisting web proxy-based HTTP attacks by temporal and spatial locality bsehavior [Journal Article]. *Parallel and Distributed Systems, IEEE Transactions on*, *24*(7), 1401-1410.

Yang, J., Tiyyagura, A., Chen, F., & Honavar, V. (1999). Feature subset selection for rule induction using RIPPER. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Vol. 2, p. 1800).

Ye, C., & Zheng, K. (2011). Detection of application layer distributed denial of service [Conference Proceedings]. In *Computer Science and Network Technology (ICCSNT), International Conference on* (Vol. 1, p. 310-314). IEEE.

Yu, S., Guo, S., & Stojmenovic, I. (2012). Can we beat legitimate cyber behavior mimicking attacks from botnets? In *INFOCOM, 2012 Proceedings IEEE* (pp. 2851–2855).

Yu, S., Zhou, W., Jia, W., Guo, S., Xiang, Y., & Tang, F. (2012). Discriminating DDoS attacks from flash crowds using flow correlation coefficient [Journal Article]. *Parallel and Distributed Systems, IEEE Transactions on*, *23*(6), 1073-1080.

Yu, Z., & Tsai, J. J. (2011). *Intrusion detection: A machine learning approach* (Vol. 3) [Book]. World Scientific.

Zargar, S. T., Joshi, J., & Tipper, D. (2013). A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks [Journal Article]. *Communications Surveys & Tutorials, IEEE*, *15*(4), 2046-2069.

Zhang, G. Q., Zhang, G. Q., Yang, Q. F., Cheng, S. Q., & Zhou, T. (2008). Evolution of the Internet and its cores [Journal Article]. *New Journal of Physics*, *10*(12). doi: 10.1088/1367-2630/10/12/123027

Zhang, J., Chen, C., Xiang, Y., Zhou, W., & Xiang, Y. (2013). Internet traffic classification by aggregating correlated naive bayes predictions [Journal Article]. *Information Forensics and Security, IEEE Transactions on*, *8*(1), 5-15.

Zhou, W., Jia, W., Wen, S., Xiang, Y., & Zhou, W. (2014). Detection and defense of application-layer DDoS attacks in backbone web traffic [Journal Article]. *Future Generation Computer Systems*, *38*, 36-46.

Appendix A

Machine Learning Parameter Values

A.1 Naïve Bayes

Remark	**Parameter**	**Value**
The preferred number of instances to process if batch prediction is being performed	batchSize	100
The number of decimal places to be used for the output of numbers in the model	numDecimalPlaces	2
Use kernel estimator for numeric attributes rather than a normal distribution	useKernelEstimator	false
Use supervised discretization to convert numeric numeric attributes to nominal ones	useSupervised Discretization	false

A.2 Decision Tree

Remark	**Parameter**	**Value**
The preferred number of instances to process if batch prediction is being performed	batchSize	100
Whether to use binary splits on nominal attributes when building the trees	binarySplits	false

Whether parts are removed that do not reduce training error	collapseTree	true
Pruning confidence factor, smaller values incur more pruning	confidenceFactor	0.25
If true, the split point is not relocated to an actual data value	doNotMakeSplitPoint ActualValue	false
The minimum number of instances per leaf	minNumObj	2
The number of decimal places to be used for the output of numbers in the model	numDecimalPlaces	2
Whether reduced-error pruning is used	reducedErrorPruning	false

A.3 JRip

Remark	**Parameter**	**Value**
The preferred number of instances to process if batch prediction is being performed	batchSize	100
If true, the check for error rate > 50% is included in stopping criterion	checErrorRate	true
The amount of data for pruning	folds	3
The minimum total weight of the instances in a rule	minNo	2
The number of decimal places to be used for the output of numbers in the model	numDecimalPlaces	2
The number of optimization runs	optimizations	2
The seed used for randomizing the data	seed	1
Whether pruning is performed	usePruning	true

A.4 Support Vector Machines

Remark	**Parameter**	**Value**
The type of SVM to use	SVMType	C-SVC
The preferred number of instances to process if batch prediction is being performed	batchSize	100
The cache size in MB	cacheSize	40.0
The coefficient to use	coef0	0.0
The cost parameter C	cost	1.0
The degree of the kernel	degree	3

Whether to turn off automatic replacement of missing values	doNotReplace MissingValues	false
The tolerance of the termination criterion	eps	0.001
The gamma tu use, if - then 1/max_index is used	gamma	0.0
The type of kernel to use	kernelType	linear
Normalize the data	normalize	false
The number of decimal places to be used for the output of numbers in the model	numDecimalPlaces	2
Whether to generate probability estimates	probabilityEstimates	false
The random number seed to be used	seed	1
Whether to use shrinking heuristics	shrinking	true
The weights to use for the classes	weights	1

A.5 Self Organizing Maps

Remark	**Parameter**	**Value**
The number of epochs in convergence phase	convergenceEpochs	1000
The height of lattice	height	2
The initial amount the weights are updated	learningRate	1.0
Normalize the attributes	normalizeAttributes	true
The number of epochs in ordering phase	orderingEpochs	2000
The width of lattice	width	2