A Systems Approach to Network Modelling for DDoS Attack Detection using Naìve Bayes Classifier

A THESIS

submitted by

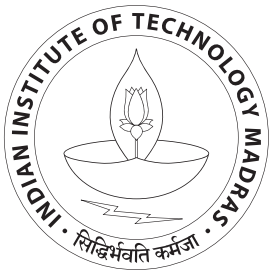
R VIJAYASARATHY

for the award of the degree

of

MASTER OF SCIENCE

(by Research)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

February 2012

THESIS CERTIFICATE

This is to certify that the thesis entitled "**A Systems Approach to Network Modelling for DDoS Attack Detection using Naìve Bayes Classifier**" submitted by **R Vijayasarathy** to the Indian Institute of Technology Madras, Chennai for the award of the degree of **Master of Science (by Research)** is a bonafide record of research work carried out by him under my supervision and guidance. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. B. Ravindran
Research Guide
Associate Professor
Dept. of Computer Science and Engineering
IIT-Madras, Chennai – 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

It feels good to get paper space, to express gratitude to those who have contributed to making this work a reality. Not so long ago, I would have only been dreaming about this situation. Unlike others, my journey towards reaching this stage has literally been an epic, as my supervisor Prof. Ravindran will acknowledge. The epic has seen long periods of inactivity, more than one year of hard work, and some last minute magic which re-kindled the dying interests in me to earn a degree at a prestigious institute like the IIT M.

I am grateful to my first mentor Dr. Kapali Viswanathan, who motivated me towards this idea of doing formal academic research at IIT Madras for a masters degree. He has also motivated me psychologically when my chips were down. I would also like to acknowledge and thank Prof. Pandurangan, for his encouragement during times of inactivity. I rue missing a good opportunity to work with him.

I feel grateful and eternally indebted to my research guide Prof. Ravindran for his support and assistance in all forms – academic and non-academic. In many situations, he assumed more than a technical supervisor's role and helped me out. On the research front, the discussions I have had with him have been very insightful, and I stood to benefit in multiple ways from our interactions throughout this period. His professional and personal ethics has been an inspiration to me. I also thank Prof. Sreenivasa Kumar, Prof. Narayanaswamy, and Dr. Rama for their guidance in research. I also like to thank Prof. Shankar Balachandran for his association in our discussions and for providing useful inputs.

Being an external scholar, I have not had the privilege of sitting through the day in the campus and socialising with fellow students at the Reconfigurable and Intelligent Systems Engineering (RISE) lab. However, my association with Ranga

from RISE lab has always been fun, and importantly, useful to both of us. I thank him for providing me with help whenever I wanted from him.

I would also like to thank my parent organisation, the Society for Electronic Transactions and Security (SETS) for their support in letting me pursue this degree as an external scholar. Technically, my colleagues at SETS Mr. Uppili, Mr. Dillibabu, Mrs. Suganya have all been helpful, mostly in implementing the system which incorporates the developed algorithms, and sharing with me performance related details whenever I wanted.

With regards to the thesis, I would like to thank my collaborator Dr. Ejaz Ahmed, Queensland University of Technology (QUT), Brisbane, for his honest attempt at comparing our respective methods. I also acknowledge that the flow of the related works section in this thesis is inspired by his introduction in [3]. My thanks go to my other collaborators at QUT.

I realised the importance of proof-reading only out of writing this thesis, and I am thankful to everyone who has taken the efforts to proof-read this document. This includes Mr. Uppili, who at his age, took the pains to meticulously go through some chapters of this thesis. I was heartened and more than happy to let the document be proof-read by my parents, who volunteered for it. Even here, I admire the remarkable proof-reading skills of Prof. Ravindran.

And last but not the least, I cannot forget my family's support during this entire phase – My parents, sisters and my wife. I know that I have given them enough confusion and disappointment throughout the journey, but I am glad that I have been able to catch up finally, to see them happy. I dedicate this thesis to my parents.

ABSTRACT

Keywords: *DoS and DDoS attacks*; *TCP and UDP*; *Naìve Bayes Classifier*; *FPGA*

Denial of Service attacks are a major threat to the modern electronic society. Carefully crafted attacks of large magnitude, better referred to as Distributed Denial of Service attacks (DDoS) have the ability to cause havoc at the highest level, national information infrastructure. The ease at which such attacks can be performed today is startling given the number of free online tools available in the open. An ideal situation would be to differentiate between good and bad packets as generally attempted and accomplished in the case of intrusion attempts, which in the case of DoS attacks is extremely hard as only the intent differs between a genuine user and an attacker.

There are multiple perspectives of dealing with the DoS attack problem, in order to mitigate, detect and cope with detected attack. Techniques dealing with each of these aspects can be integrated to make a fool-proof system. An important such perspective in terms of detecting DoS attacks is to view the problem as that of a classification problem on network state (and not on individual packets or other units) by modelling normal and attack traffic and classifying the current state of the network as good or bad, thereby detecting attacks when they happen. Classical machine learning algorithms are used to solve the problem.

In this work, the approach to a carefully engineered, practically realisable system to detect DoS attacks based on a Naìve Bayesian classifier is described. The system is designed to be near the target. The focus of the system is on two transport layer protocols – TCP and UDP, both of which work on a common framework of mechanisms, although with different input parameters.

For the TCP protocol, TCP flags are taken as the feature, and the probability space is engineered taking into account hardware implementability and operability at line speeds. Simple independence assumptions are made inside a window of packets, and the probability of a window is determined as a function of the probabilities of flag combinations observed inside the window. Threshold probability is determined by cross validation – a technique commonly used in data mining. For the UDP protocol, the feature is the Windows Arrival Time (WAT), the time taken for a window of packets to arrive at the mitigation device. Techniques similar to TCP are used.

Experiments are carried out with both tagged and untagged datasets, and the results are found to be promising. In order to assert the efficiency and usefulness of our system, the system was compared with a one-class SVM classifier, and found to be more suitable for hardware implementation.

TABLE OF CONTENTS

List of Algorithms

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Problem Description

The world has seen rapid advances in science and technology in the last two decades, which has enabled dealing with a wide spectrum of human needs effectively. These needs vary from simple day-to-day needs like paying electricity bills, booking train tickets, etc., to sophisticated needs like power grids for power generation and sharing. These technologies have taken human life into much higher levels of sophistication and ease. But in the middle of this phenomenon, the rise and growth of a parallel technology is startling – that of compromising security, thereby resulting in different effects detrimental to the use of technology. This includes attacks on information, such as stealing of private information, hacking, and outage of services. Media and other forms of network security literature report the possibility of the existence of underground anonymous attack networks which can effectively attack any given target at any time. This only shows a possible shift in attack perspective in current days and in times to come – from wars causing physical damage and destruction to what is termed "information warfare", compromising of attacks mentioned above. The twist in the latter is that these attacks are mostly performed by attackers/networks who can conceal themselves. Recent attacks on the Sony PlayStation network and the website of the CIA among others (see Figure 1.1) stand testimony to this fact. In particular reference to the cyber attack on the Sony network, the personal data (including credit card information) of an estimated 10 million customers was compromised [59]. Intrusions[1] [38] are a typical example of cyber attacks.

While the range of attacks that can be performed on targets is as broad as the spectrum of constructive technology itself, this thesis deals with a particular class of attacks known as **Denial of Service (DoS)** attacks.

[1]The act of exploiting vulnerabilities to compromise a system or a network for a variety of benefits, including stealing information, infecting system or network for service disruption, exploitation for carrying out attacks etcetera. For a more formal definition, refer to [38].

Denial of Service (DoS) attacks are a class of attacks on targets, which aims at exhausting target resources, thereby denying service to valid users. The target resources could be in terms of space and/or time. For example, servers providing SSL service could be time-attacked by making them perform a lot of expensive cryptographic operations (public key decryption in this case) thereby preventing them from serving their genuine clients. Alternately, servers could also be space-attacked by exhausting their bandwidth or connection buffers with lot of bogus packets/requests.

Distributed Denial of Service (DDoS) attacks are a scaled form of DoS attacks where multiple attack bots[2] are employed in a coordinated fashion to form an attack network for attacking a specific target. DoS and DDoS attacks are catastrophic particularly when applied to highly sensitive targets like Critical Information Infrastructure. A classical and well noted example of a DoS attack is a SYN flooding attack [16], which is most popular to this day.

A fact which is often acknowledged by security researchers throughout the globe is the ease with which serious DoS and DDoS attacks can be possibly carried out today. Software tools available for free download on the Internet [11] give a *script kiddie*[3] the ability to carry out low volume to high volume DoS and DDoS attacks [38, 60]. It is possible that a compromised system may be used as a client (with purportedly low activity so as not to raise suspicion with the user of the computer) for performing DDoS attack on a target. Major security players and security agencies in the globe have reported such DDoS attacks [18, 12, 39, Symantec, 15] and have published a fair bit of analysis. Shockingly, various sources report the fact that complex botnets can be *rented* to perform an attack on a chosen target! [24].

The one important issue that differentiates DoS and DDoS attacks from the other cyber attacks like intrusions is the enormous volume of traffic that the protection systems may have to deal with in very short time. For example, it is reported that an estimated 10 million bots participated in the attack against the Estonian gateways [33], causing total disruption of access to services outside the country. This fact essentially calls for light weight techniques (and implementations) which will be able to handle such large traffic, and work at "wire speeds" online.

[2]Software applications which are clandestinely installed in vulnerable computers, with instructions to respond to their masters and act at designated times to perform coordinated attacks on chosen targets.

[3]Script kiddie is a term used to describe people who use scripts or programs developed by others to perform attacks.

The targets of a DoS attack could be servers which offer specific services, or clients which could be used as bots to attack other servers. The service may be an open service, available to all users of the Internet, or a closed group service. The target service may also be a service like mail service offered by `www.gmail.com`, news offered by web server at `www.timesofindia.com`, a DNS server, an Internet gateway, or an entity as sophisticated as a power grid. DoS attacks are carried out on different targets using the target service vulnerabilities. A detailed view of DoS and DDoS attacks, their targets, methods, etc., could be obtained from a book on the subject, like [40]. For the limited purpose of focussed research, the target of a DoS attack is assumed to be a server providing a particular service online. Alternately, the service may be a sophisticated, closed group service.

An ideal solution with respect to DoS and DDoS attacks would be to differentiate between good and bad packets (alternately, connections, sessions, flows, etc.,) in a network. Works in [27] propose techniques which use packet TTLs to differentiate between normal packets and (spoofed) attack packets. Authors of [49] give an interesting insight into differentiating between DDoS bot requests and normal user requests, which may help understand differentiating characteristics of good and bad packets. However, an ideal solution to this problem is still widely regarded as an extremely difficult proposition, since only the intentions differ in most DoS attacks. Alternatively, certain characteristics of normal traffic can be tapped to differentiate them from attack traffic, thereby letting us differentiate between attack and normal "states" of the network (rather than differentiating packets). For instance, a well researched result suggests that incoming source IP addresses could be a good candidate for detecting normal traffic and attack traffic respectively [4, 2]. But this argument may also stand to get defeated, primarily due to two aspects: a) [35] reports how the actual composition of normal traffic is both diverse and continuously evolving, and b) It is reported that an attack traffic can be crafted to mimic normal behaviour [10].

Another interesting scenario which provides a further twist to the problem is that of a flash crowd [10], where there is a sudden huge influx of traffic to select servers due to the occurrence of an unexpected event (for e.g. news servers being flooded with requests on account of a sudden world incident). In the case of a flash crowd, the effect is the same as that of a DoS attack, although the intention is genuine.

The problem to the research community takes multiple facets from here. Some of them are the following:

- **Attack Detection.** Can we detect attacks quickly? Simple solutions like bandwidth metering can take time for the attack to get detected. Hence an intelligent method of quickly and efficiently detecting attacks should be in place. Classical machine learning techniques like [19, 52, 34, 66, 64, 20] have been used to propose new methods of DoS and DDoS attack detection. The primary challenges lie in performance, availability of training data, and accommodating changing traffic profiles.

- **Differentiating Attacks from flash floods.** As already introduced, flash floods are sudden and extreme bursts of legitimate traffic, and it is important to differentiate between a genuine flash flood and a DDoS attack. Various features have been discussed as possible candidates to achieve this. For instance, [70] uses various measures of "distance" between flows of DDoS traffic and flash floods, motivated by the fact that attack programs exhibit a uniform flow property. Work done in [58] discusses use of packet arrival information in either situations to differentiate them. The authors of [49] present another interesting angle to the problem – by attempting to model different aspects of human behaviour, in order to differentiate between DDoS bots and normal users.

- **Mitigation.** What next after an attack has been detected? After an attack has been detected, how does the target react to it. For instance, the service could simply be closed down for some time to let the attack recede (when we are causing DoS to ourselves). [31] mentions random early dropping as one of the earliest solutions to handling attacks, although it clearly risks dropping legitimate requests. [53, 28] proposes intelligent and probabilistic packet dropping mechanisms based on Traffic Rate Analysis. Sophisticated filtering techniques have also been proposed to mitigate attacks. [27, 62] propose hop-count filtering techniques by using the Time To Live (TTL) of a network packet. [67] introduces a capability based flow filtering method to selectively suspicious packet flows when an attack is on. [57] proposes IP traceback to filter packets during attack.

  In the case that the service has been affected by the attack and it is sure that the server may not be able to function in full health, the server is expected to do preferential servicing [47] or accommodate graceful degradation. Hash cash [7] is a classical example of graceful degradation. Research in the cryptographic community introduces a novel way of handling this problem, by models such as client puzzles [31], which discourage attack attempts, and also help in coping

up with attacks by the use of moderately hard problems [6, 63, 56, 1].

- **Post-Mortem.** How to perform a post-mortem analysis after the attack? This effects in trying to answer questions like
  - **Can we trace back the attack to it's source?** Literature reports a lot of IP traceback mechanisms [21, 22, 69, 71]. These methods in essence do some kind of packet marking, in order to trace packets/collection/flows at a later point. For instance, [22] proposes use of checksum in a scalable fashion in order to relate sequence of message fragments. The problems with IP traceback are generally two pronged – Efficiency and cooperation requirement from Internet Service Providers. These are the focal points for research.
  - **Can we use some kind of pushback mechanism?** Pushback mechanisms not only drop packets near the target, but recursively network with neighbouring network devices to propagate dropping of path-identified sources. This will ensure that the attacks are pushed more and more away from the target. For example, [26] presents an architecture for router pushback of UDP attack packets, by considering the problem as a congestion-control problem after attack has been flagged. These measures again require cooperation from ISPs or intermediate core routers, some of which may not even be known to the entity intending to protect a target. Research along these lines focus on such problems.

All the above directions of research look to be equally promising in their own right, given the fact that there already exists popular DDoS attack mitigation devices in the market from world class network security solutions providers like Cisco [14], Radware [51], Arbor [5], etc., and still attacks have kept happening with open declarations (Figure 1.1).

This work is in the direction of the first of these problems – detecting DoS and DDoS attacks at "*line-speeds*" using machine learning techniques. The tactic is to convert the problem into that of a classification problem on network state by modelling normal and/or attack traffic and classify the current state of the network as normal or abnormal/attack. The chosen traffic is captured into objects of a mathematical model, and a given online traffic pattern is determined to be part of normal or attack traffic based on it's proximity to the modelled traffic.

1.2 Scope of Research

The primary objective of this work is to develop solutions to DoS and DDoS attacks using machine learning techniques and implement them in hardware. Hence, the scope of this work is to analyse and design a system based on machine learning techniques, to detect DDoS attacks on public services running over TCP or UDP protocols, with specific suitability on the detection process for hardware implementation. The training phase is assumed to be offline, and the detection phase online. The scope of testing the developed concepts is confined to testing with standard datasets and limited number of real time datasets.

1.3 Contribution of this work

The contribution of this work is the following:

1. This work discusses the design of an anomaly based system developed to detect DoS and DDoS attacks using a Naìve Bayes classifier approach, coupled with elegant tricks to make the system usable in real-time. The system is designed to be near the target.

2. The focus of the system is on two transport layer protocols – TCP and UDP, both of which work on a common framework of mechanisms, although with different input parameters. While other protocols in the Internet stack may be different, we believe that the basic solution framework can still be retained while working out solutions for other protocols.

3. The algorithms have been tested with representatives of both tagged and untagged traffic. While tagged traffic represents an ideal situation where the users exactly know what is what, a more practical situation is when there is only a vague understanding of the traffic. In many cases, there may have been minor attack attempts which may have failed or discontinued. It may be difficult for the user to give accurate tags in such cases.

4. The work has been ported into a real-time, embedded system prototype, whose design and implementation details, along with recorded performance parameters are described.

1.4 Structure of the Thesis

The thesis is organised in the following manner:

- Chapter 2 discusses related work including approaches, drawbacks and challenges, and ends with a discussion on the history of the accomplished design;
- Chapter 3 discusses the general requirements and considerations towards design of a practical DoS mitigation system;
- Chapters 4 and 5 discuss in detail the algorithms used for modelling TCP and UDP traffic with a note on their complexities for both the training phase and the deployment phase;
- Chapter 6 discusses experiments done on the system towards parameter tuning and performance estimation. The chapter ends with details of an effort to compare the system with existing methods; and
- Finally, Chapter 7 concludes on the thesis, and discusses scope for future work.

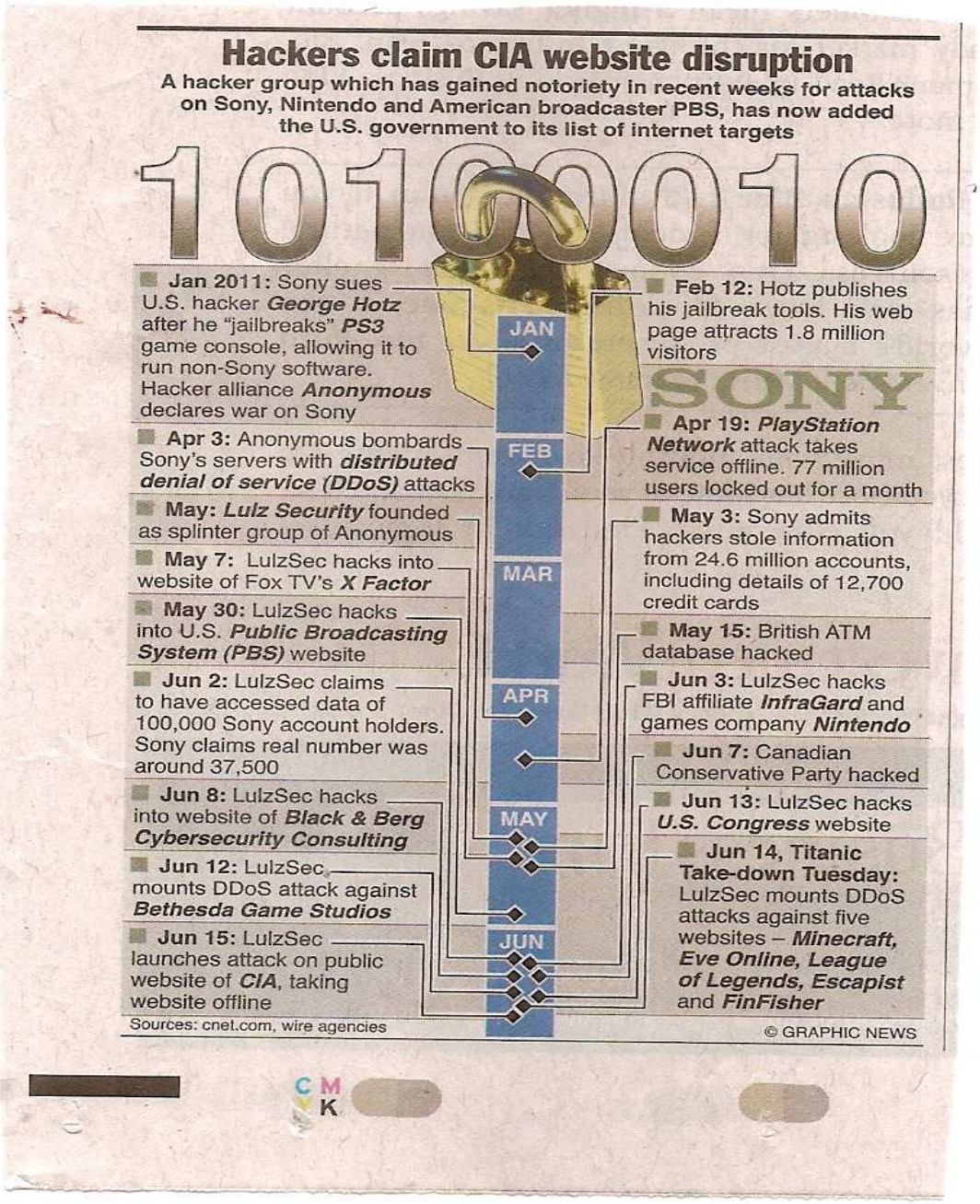


Figure 1.1: Recent News on Attacks performed on Major websites – Published by THE HINDU, article dated 18 Jun 2011

CHAPTER 2

Related Work

2.1 Transmission Control Protocol (TCP) – An Introduction

The Transmission Control Protocol (TCP) is a transport layer protocol which is widely used by applications which require reliable transmission. This is provided by TCP by means of mechanisms for time-out and retransmission of packets in a sliding window based protocol.

TCP establishes reliable connections between endpoints by means of different handshake mechanisms which will ensure that both endpoints are assured of the connection and it's activities. The entities are defined by an IP and a logical port. For example, TCP ensures connection establishment between two entities by a three way handshake between them, the popular SYN-SYN/ACK-ACK handshake. TCP maintains handshakes and mechanisms for reliability, error correction, congestion control, flow control, etc. The TCP packet header contains fields such as flags and sequence numbers which will determine the state of connectivity between entities, and also bind them while letting multiple sessions to execute in parallel. The TCP in that sense, is a Finite State Machine.

The header of the TCP packet running over IPv4 is as shown in Figure 2.1.

Most popular application layer applications run over TCP. This list includes, web services (port 80), mail services (port 25), and file transfer services (port 21).

The very fact that TCP accommodates time-outs and retransmissions makes it a hunting ground for malign entities which exploit these properties and cause attacks such as DoS by using these properties.

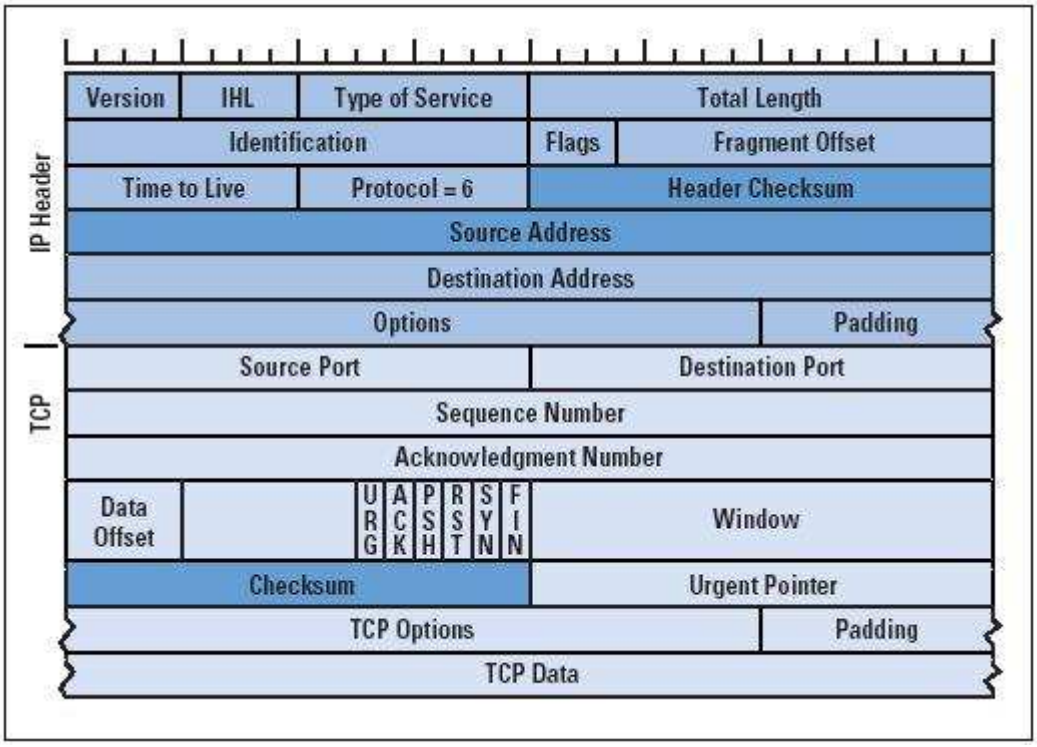


Figure 2.1: TCP running over IPv4 – Header

2.2 User Datagram Protocol (UDP) – An Introduction

The User Datagram Protocol (UDP) is a transport layer protocol which is used by applications which do not necessarily call for reliability of transmissions, thereby giving them a non-guaranteed datagram delivery. This gives applications direct access to the datagram service of the underlying IP layer.

Since UDP does not take care of communication reliability issues, it is upto the applications themselves to be dealing with end-to-end communication problems like retransmission, reordering, congestion control, flow control, etc. In general, the applications which run over UDP are those which do not require the level of service offered by TCP. Additionally, the applications are also those which are time-critical and can afford to drop packets instead of waiting for retransmission (while managing loss of such packets).

Examples of applications running over UDP include Domain Name Services (DNS),

and video conferencing applications.

The header of UDP running over IPv4 is as shown in Figure 2.2.

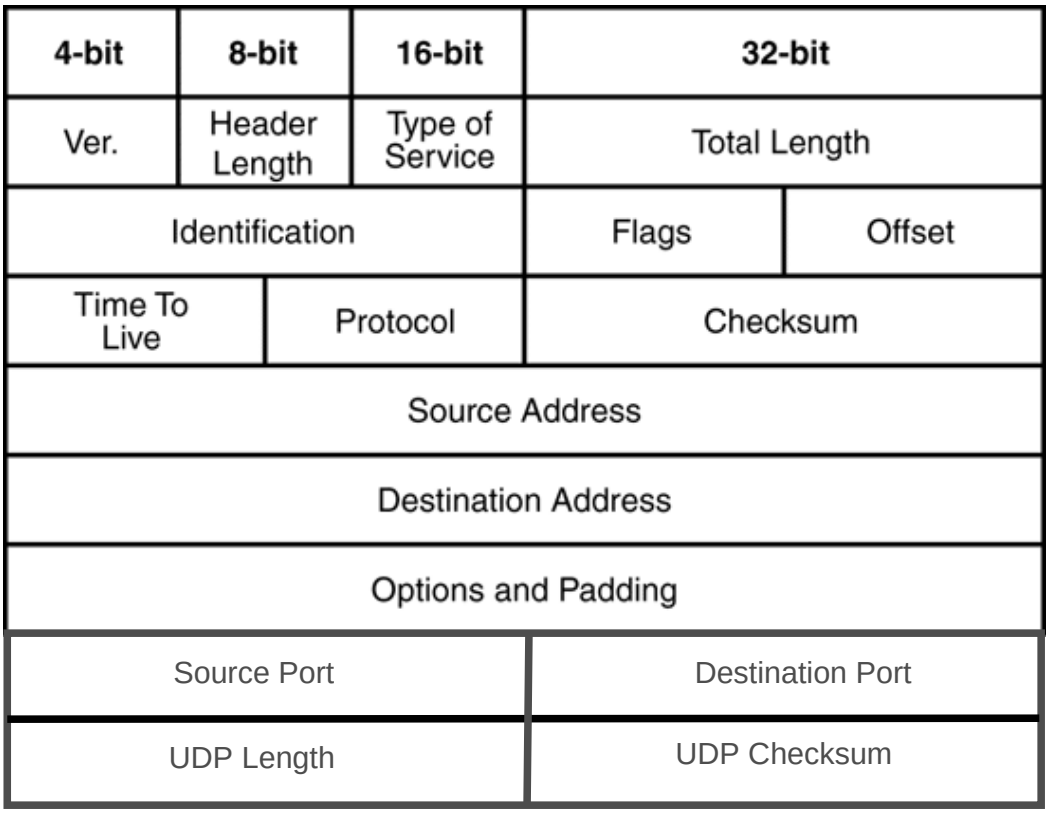


Figure 2.2: UDP running over IPv4 – Header

Since basic UDP does not provide for orderly communication, it does not inherit the vulnerabilities of TCP. However, services running over UDP can still be DoS attacked through vulnerabilities in the application itself. Bandwidth based attacks are also common with applications running over UDP.

2.3 DoS Attack Protection using Machine Learning – Literature

The very nature of high-rate flooding DDoS attacks suggests that the primary point of focus of corresponding attack-detection techniques should be on detecting anomalies in network traffic. For the limited purpose of this discussion, it is assumed that the detection process comprises of three main components:

- Measurement of parameters of interest;
- Data analysis, and model evolution, both part of the *learning phase*;
- Decision making on whether the observed behaviour is normal or anomalous, part of *deployment*.

2.3.1 Measuring Parameters

The method of data-analysis, combined with the constraints posed on the system primarily in terms of performance, decides the number and quality of parameters to be measured. The most common measurements are the typical Management Information Base (MIB) measurements like volume of traffic entering network (ingress), number of packets/second, etc. These are the ones which are easy to manage, and can also be implemented in a modular fashion. Going one step deeper into choosing of parameters, protocol type, source/destination IP addresses have been considered in order to differentiate protocol, and to channelise traffic [10].

Traffic Rate Analysis (TRA) [52] is a commonly used parameter which captures some of these parameters mentioned above. It is generally described as the proportion of a specific collection of packets (the collection depends on the parameters chosen for observation) to the total number of packets. For example, [52] defines TCP flag rate as "the number of a specific TCP flag to the total number of TCP packets" (TCP flag is the examination parameter here).

The use of multiple parameters for examination has also been considered. Authors of [32] consider three different features which are trained under independent HMMs and later "fused". Another alternative reported in literature corresponds to functions pertaining to interlinking and sequencing of packet or packet information pertaining to protocols/IPs, leading to the analysis of "flows of traffic". The flows are defined in terms of combinations of source IPs, destination IPs, source ports, destination ports. For example, [23] analyses sessions of traffic by separating traffic based on <source ip,source port, destination ip,destination port> of TCP traffic, thereby capturing the entire sequence of packets within the session. But then the entire session information will not be available until the session is closed during detection, which calls for avoidance of session related techniques during detection.

In the interests of performance, header analysis has been widely preferred and his-

torically used in different forms for defining examination parameters. These include source and destination IP addresses, packet type, and other queries related to arrival and departure of packets with marked addresses or types. However, the advent of high-performance hardware has created an environment which can accommodate examination of more complex levels of parameters. "Application aware firewalls", as they are so called, exhibit the capability to perform "deep-packet inspection", including the ability to parse the application level content of each packet [25]. The availability of high performance hardware adds an important dimension to the performance of detection, although there are a few concerns – privacy of user data, and unavailability of data in the case of encrypted content, which will leave flow related information incomplete, and may mislead analysis in some cases.

In a nutshell, the research community has mostly focussed on a small number of base fields/values from the packet header, based on which multiple features are derived, ranging from simple features such as observing the rate of occurrence of single bits in the TCP flag field, to features such as rate of change of source IPs for detection. Source and destination IPs, destination ports, TCP flags have been the major fields in observation with most works. Many works have relied on design similar to Management Information Bases, where queries could be placed on required features. Further, solutions to the problem can be broadly classified as those based near the source or near the target, or at network transit points (routers).

2.3.2 Data Analysis

Various machine learning techniques have been proposed to analyse data based on chosen input parameters. Performance has been a major factor driving the choice of the model to be employed for data analysis, importantly during detection.

Some of these are mentioned below:

- *Statistical Approaches*:
  - In [19], a Chi-Square-Test based modelling technique is proposed on the entropy values of the packet headers.
  - [29] discusses the effects of multivariate correlation analysis on the DDoS detection and presents an example of SYN flooding.

  - [30] uses the covariance matrix to present the relationship between each pair of network feature and identifies attacks.
  - In [72], the CUSUM algorithm is used to detect the change in the amount of packets to destination.

- Traditional Machine Learning techniques
  - Markov Models [65] consider application layer DDoS attacks and use objects called hidden semi-markov models to describe browsing behaviour of users for anomaly detection at the application layer.
  - [52] uses the Traffic Rate Analyser (TRA) to model the TCP flag rate, a measure of proportion of packets with chosen flag set to the total number of TCP packets, and the protocol rate, a measure of the proportion of packets from chosen protocol to the total number of inbound packets.
  - Hidden Markov Model based DoS attack solutions have also been proposed in [32, 66], highlighting the use of multiple features for sequence analysis, and coordinated efforts to spread the word during an attack situation to improve performance.
  - Other classification algorithms such as Genetic Algorithms [55], Artificial Neural Networks (ANN) [20, 64] and Bayesian Learning [37, 35].

- Hybrid modelling techniques which use multiple algorithms for modelling the same data in parallel, like [55, 54] have also been attempted, and provide interesting results.

- Recent works [34, 17] have discussed use of Bayesian classifiers towards intrusion detection in general, which includes DoS attacks.

A broad taxonomy of DDoS attacks and defence mechanisms has been documented in [43].

2.3.3 Decision Making and Mitigation

The decision to accept or discard a packet or a group of packets (which let suspect the state of the network) can be made on a white list or black list basis. The list can take a variety of forms based on the performance of hardware in which the

system is running. These could be based on individual IPs/groups of IPs/domains. Alternatively, these could also be based on state information if the system is capable of maintaining it, or it could be based on sophisticated rule sets built upon complex parameters. For the limited purpose of building the system, *a white list IP based mitigation strategy* is described in this work.

2.3.4 Description of selected works from Literature

A few selected works pertaining to DDoS attack detection, using multiple features, different datasets is described below:

- In [66], the authors model the rate of change of new and old IP addresses at nodes in the network into a Hidden Markov Model, and determine attack based on the above characteristic. Their work further explains how results appear to improve with attack information exchange between distributed nodes using Cooperative Reinforcement Learning techniques. The solution is a target end solution.

- [52] uses the Traffic Rate Analyser (TRA) to model the TCP flag rate, a measure of proportion of packets with chosen flag set to the total number of TCP packets, and the protocol rate, a measure of the proportion of packets from chosen protocol to the total number of inbound packets. A sum of 10 features, each representative of either of the above are monitored and separately modelled for both normal packets and attack packets as a series of SVMs, and attacks detected as DoS, DDoS, or DrDoS attacks based on the class to which the pattern belongs to. The solution is based near the source.

- The work in [29] presents a generalised view of multivariate correlation analysis for DDoS detection, and a subsequent illustration with SYN flooding attacks. The idea projected in the paper is to differentiate between normal and attack traffic by considering the correlation between possible pairs of chosen features among a set of features $f_1 \ldots f_p$ at different intervals, and the distance of the formed correlation matrix with the average of correlation matrices seen during training. The authors consider the 6 flags in the TCP field as separate features and show the effectiveness of the method under these conditions. The solution is a target end solution.

- The solution in [32], deployed at the source end, uses three features – a 3 tuple {*source IP, destination IP, destination port*}, a 16-bit packet identifier, and the TCP header flags on the same dataset, which is trained for each of the above three as independent HMMs before the HMMs are coupled to form the multi-stream fused HMM. If the probability of the traffic pattern is lesser than the threshold probability, the pattern is flagged anomalous, and a suitable threshold on proportion of anomalous patterns to total patterns determines an attack.

- A good example of using multiple anomaly detection algorithms in parallel is the work of Shanbag and Wolf in [54]. They exploit the potential of high-performance parallel processors to operate several anomaly detection algorithms in parallel. The output from each algorithm is normalised and then aggregated to produce a single anomaly metric. Such classifiers are generally termed "ensemble of classifiers".

2.4 Motivation

In many of these works, a few drawbacks are generally observed. In these works, DoS attacks have been classified to fall into the broad category of Intrusion attacks, and so solutions have been oriented more towards formal intrusions, including root escalation, scripting attacks, etc. A major factor that is often missed by classifying DoS attacks into intrusion attacks is the enormous volume that DoS detection solutions have to handle in comparison with intrusion attacks. This fact straight away differentiates DoS attacks from other types of intrusion, and renders learning models such as SVMs, HMMs, ANNs impractical when it comes to real time attack detection. The detection mechanism is expected to support speeds in the order of at least a few Gbps[48]. Another motivational fact is that the very nature of DoS attacks (the system can tolerate some attack traffic unlike intrusions) helps in modelling traffic using simpler notions than the notions which are described above.

Another practical drawback of supervised/unsupervised learning models for DoS detection is the accuracy of (supposedly normal) traffic supplied to learning[1]. In practical user sites, it is a very difficult proposition to be fully confident that the

[1]During the training phase, the system does learning. Learning and training are used interchangeably

input to learning is absolutely normal. The system may cause false alarms if traces of the training traffic contained abnormalities. For making practical systems, additional work has to be done to eliminate such abnormalities which may be present in traffic supplied to learning.

Another common problem to the entire research community on DoS attacks is the lack of training data. The DoS research community depends extensively on standard datasets (KDD dataset [61] and DARPA dataset [45]) for the purpose of learning and analysis, which are better suited to analysis of intrusion attacks. Further more, the datasets used are well-tagged, which may generally not be the case with real user sites.

So a practical system using any machine learning algorithm to detect DoS attacks should be carefully engineered to be a) Light weight; b) Accommodative of practical difficulties in learning. Different systems for DoS detection and mitigation have different approaches to the problem based on their position in the network. The system could be placed either near the target, or the source, or anywhere at an intermediate point in the network[72].

2.5 Issues and Challenges

In summary, the following are the important issues and challenges towards the objective of implementable DoS detection design:

- Statistical and machine learning approaches are best known to be effective given large training datasets, which capture a variety of possible patterns. But in practical user sites, the training data has to be gathered separately apriori for initial configuration, in a process which will require reasonably long running times. This only increases deployment time, which makes it undesirable. The system, therefore, may have to work with comparatively lesser training data.
- Although works on machine learning techniques mostly assume that the input traffic given to the training phase is absolutely normal, this may not always be true. So, the system should be able to tolerate some abnormalities inside input training traffic. This factor should be appropriately quantified into the system.
- Since it is ultimately aimed to have the system implemented in hardware, the

algorithm should ensure that the implementation will occupy as little space as possible. This in particular reference to probabilistic models, may require bringing down to the extent possible the number of floating point values to be stored and used.

- Since the detection phase is assumed to be online, and also it is desired to have detection running at close to line speeds, the detection mechanism should be as light-weight as possible in terms of the number of operations required for a verdict on traffic patterns.

2.6 History on Current Design

We feel it is important to discuss the history behind the current design, which led to the realisation of important practical aspects like the examination parameters to be taken for each protocol, the model to be used, the assumptions that could be made considering the nature of the DDoS problem and the location assumed, etc. It should be kept in mind that the research work was driven by a project which required the implementation of (parts of) the system in hardware. The reference upon which system design was mandated was the work by Vipul Hattiwale, IIT Madras as described in [23].

Further subsections briefly describe this work, and discuss why it was necessary to look beyond it for the purpose of implementation and system design.

2.6.1 Intrusion (and DoS) Detection using HMM[23] – An overview

In this work, Hidden Markov Model (HMM) [50] has been used for modelling network traffic for TCP, primarily for the purpose of detecting intrusions in the form of anomalies. The system also reports detection of the following DoS attacks:

1. *Back attack* Denial of Service against Apache web server where a client requests a URL containing many backslashes.

2. *Neptune*. SYN flood denial of service on one or more ports.

Hidden Markov Models – Fundamental Blocks

The HMM is represented with the help of two components – number of states (N), and the number of observables (M) at each state. The HMM starts with a finite number of states. Transitions among states are governed by a set of probabilities called transition probabilities, which are associated with each state. In a particular state, an outcome or observation can be generated according to a state probability distribution associated with the state. It is only the outcome, not the state, which is visible to the observer. The states are hidden to the outside world, and hence the name Hidden Markov Models. The Markov model used here is a first layer model, which means that the probability of being in a state depends on the previous state. One of the goals of using an HMM is to deduce from the set of emitted observables the most likely path in state space that was followed by the system. A HMM is represented as a triplet $\{A, B, \pi\}$, where

- A: The state transition probability matrix which is a square matrix with size equal to the number of states ($N \star N$). Each element represents the probability of transitioning from a given state to another state.
- B: The observable probability distribution which is a non-square matrix, with dimensions equal to number of states by number of observables ($N \star M$). The observable probability distribution represents the probability that a given observable will be emitted by a given state.
- π: The initial state probability vector, which is a vector of size equal to number of states ($1 \star N$). It gives the probability that the state sequence will start with a given state.

Three problems are central to a HMM:

1. Given a HMM with $\{A, B, \pi\}$, estimate the probability of occurrence of an observation sequence O_t , where t is the length of the sequence.
2. Given an observation sequence O_t , choose the most optimal state sequence Qt which would have caused O_t.
3. Adjust model parameters A, B and π to maximise probabilities of any observation sequence.

In this work, network modelling is dealt with in respect of problems 1 and 3. While the forward variable is used to solve problem 1, Baum-Welch re-estimation algorithm uses forward-backward variables to solve problem 3.

Basic Approach

The work answers the following important aspects with respect to network modelling for TCP:

1. **How is a HMM state defined?** States of the HMM in the network scenario are assumed to be non-existent, imaginary entities which are representative of the network.

2. **How many number of states?** Since states are imaginary, the number of states was determined empirically. The number of states is the number of states of the HMM with the most effective classification (lowest number of false alarms). The work reports 10 states as experimentally found effective to model network data in the DARPA dataset [45], the test dataset for the experiments.

3. **What are the observables, and how many?** Various candidates for observables were examined:

   (a) TCP flags

   (b) Payload size

   (c) Source/Destination Port number and count

   (d) Source/Destination IP number and count

   (e) Inter-packet time gaps

   (f) Total connection time till current packet

   (g) Number of packets in connection

   It is assumed that the overall traffic is streamed for analysis based on destination details like IP address, port number, etc. Also, source details are not considered because of the assumption that spoofing is the common trend among attackers.

   Parameters 6 and 7 make sense only when complete connection sequence data is supplied. So if the system is utilised on-line, these attributes won't be available

for on-line usage. Hence, those attributes are also excluded. Amongst the remaining 3 parameters, it was experimentally found that *TCP flags* are best suited among the three.

This knowledge was later borrowed for our work.

4. **How many such models to use for best results?** After analysis of distribution of connections per port, the work build one model for each frequently occurring port.

Methodology

The basic methodology is as follows:

1. During a training/learning phase, the input traffic is source-separated, and fed into a HMM, whose parameters are iteratively re-estimated and established using Baum-Welch re-estimation.
2. During the testing phase, the probability of a streamed, source-separated packet sequence of length n is estimated using the trained HMM.
3. If the probability of the packet sequence is lesser than a threshold probability, then the sequence is declared abnormal.

Experiment

Experiments are done with offline traffic reported in the DARPA dataset [45], considering complete user sessions, using the knowledge of the entire dataset. It should be noted that DARPA makes public a series of well-tagged datasets for normal traffic and attack traffic, which is generally not the case with a real-life situation. It was later observed that only a small fraction of an entire attack traffic dataset was originally (tagged) DoS attack. The experiment were based on the following:

1. During the learning phase, normal traffic was separated at three levels, based on layer 3/4 protocols used, based on destination information, and based on source information. Each stream is subjected to a HMM for the purpose of learning. Therefore, multiple HMMs are trained with the normal traffic.

2. During the testing phase, streaming is done again on all the three levels mentioned above, and the probability of the packet sequence in the stream coming from corresponding trained HMM is estimated, and flagged as anomaly if it falls below a threshold.

The different operational phases are diagrammatically described in Figures 2.6.1 and 2.6.1.

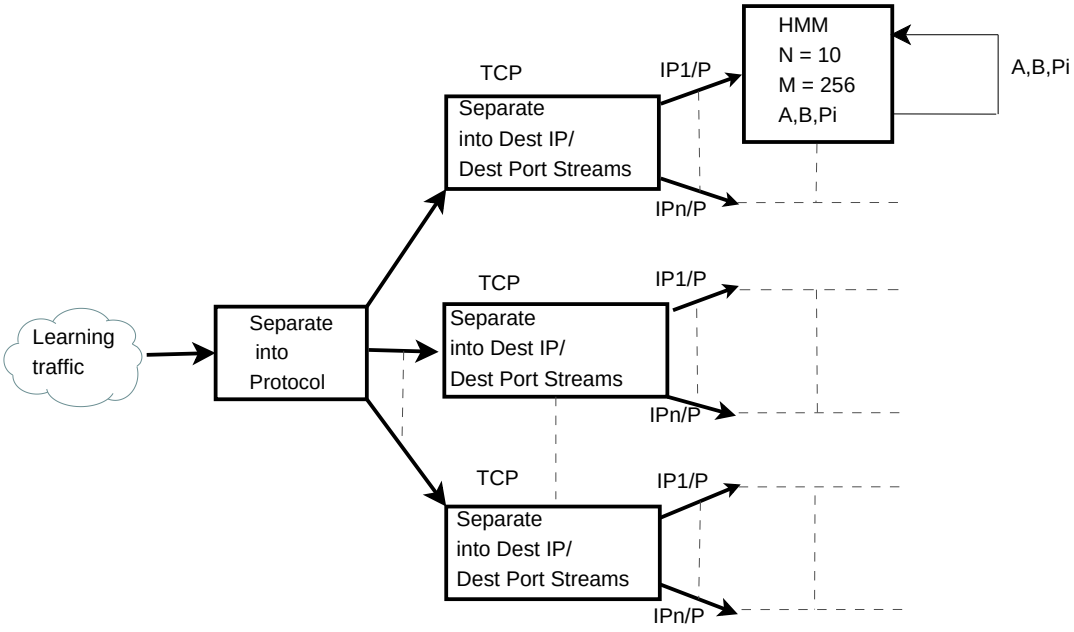


Figure 2.3: Intrusion (and DoS) detection with HMM – Training Phase

Boosting

HMM trained using the design explained above was found to have almost 100 percent detection rate. But the false alarm rate was still a concern, reporting around 5-10 percent of normal traffic as attack. To reduce false alarm rate, boosting techniques are applied. A boosting approach which is different from the conventional approaches is proposed.

After building a first classifier (HMM), some probability value is fixed as a threshold probability. All the connections having occurrence probability below the threshold are flagged as attacks. Now a second HMM is built using the suspicious samples. This essentially means that, second HMM is trained with normal traffic that was flagged as attack by first model. Similarly, a third HMM using samples with

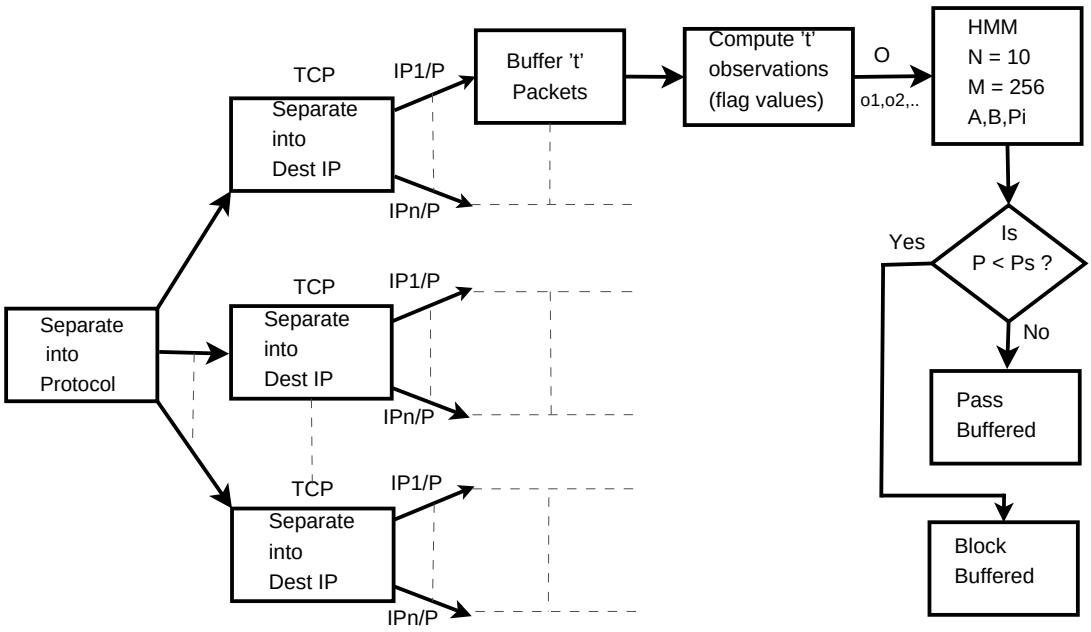


Figure 2.4: Intrusion (and DoS) detection with HMM – Testing Phase

probability of occurrence lower than threshold probability calculated for the second classifier. The process continues as many times as needed. It was observed that three levels of filters were sufficient to capture known attacks.

2.6.2 Implementation Difficulties

In an attempt to implement the detection module of the above system for detection of DoS attacks, the following difficulties were observed:

1. It could be possible that different user sites may have to start with different number of states in order to be effective. It is essential to determine the number of states dynamically. This may call for implementing accommodation of varying number of states during detection, which is undesirable, given the general space and look-up limitations on hardware[2].

2. It was observed that the number of computations in estimating the probability of a sequence of packets for each protocol may be too intensive to be implemented in hardware. This was theoretically estimated to be $O(N^2t)$, where t

[2]The hardware in consideration was a general purpose Field Programmable Gate Array (FPGA)

denotes the number of observations in input sequence, and N denotes the number of states. This is aggravated by the fact that the detection module may have to deal with high speed traffic during detection, which will create serious system consequences.

2.6.3 System Traffic Separation based on Source IPs – A Dead End

Traffic can be split at three levels – based on the protocol at the first level, this further split according to destination IPs and destination ports (that is, traffic according to each service that is offered). Separation at these two levels (either in the order mentioned or vice versa) is essential since it will lessen the complexity of the problem that we are dealing with and will make solutions effective.

Traffic can be separated at a third level in the system – based on the source IPs. The third level of separation, done in the above work, is arguable with respect to detecting a DoS attack. The arguments are given below.

Why one should separate traffic on source IPs

After the traffic has been separated based on the second level, various streams which are a mix of same protocol packets for the same destination IP/port but from various clients would be available for analysis. If the traffic were not to be separated according to source IPs, then the learning model would be encountering a stream of interleaved packets coming at random times from random clients to a particular server. These are seen as different packet sequences.

It should be seen that in a practical situation, various packets may reach the server at various time intervals, and the resulting packet sequence may be totally different from what was observed by the learning model during the learning phase otherwise. In a normal (non-attack) situation, this variation might be primarily due to latencies associated with the network, considering that the clients accessing the service are spread across the network and different clients may have different latencies for reaching the server. This also aggravates due to device failures and network failures that may happen in a practical situation, where the packets may have to be retransmitted and hence may reach the server after a delay. In an attack

situation, the bandwidth might be so occupied by the attack packets, that the packets from the legitimate clients may take unusually long time to reach the server. Since packet sequences depend on the latency factor, **there are too many such practical variations in packet sequences, all of which may not have been seen during learning**. Hence, there may be too many false positives (since they may not have occurred during learning phase).

The following example illustrates the problem: Let the learning model be HMM (Hidden Markov Model), and during the testing phase, consider that the following packet sequence would have occurred –

[SYN (from client a) –SYN (from client b) –ACK (from client a) [3]–SYN (from client c) –SYN (from client d) –SYN (from client e) –ACK (from client d) –SYN (from client f) –ACK (from client c) –ACK (from client f)]

One can observe that all clients (a,b,c,d,e,f) successfully close their connection, and the packet sequence is absolutely indicative of a normal situation. But for network latency related inconsistencies, if patterns almost same as these patterns were not seen during learning phase, then the probability of these packet sequences occurring would fall below threshold probability of the HMM, and hence these would be labelled as abnormal sequences.

Conclusion: Traffic has to be separated based on source IPs, as otherwise, the learning model based system might not have captured all variations, and will produce enormous amount of false alarms.

Why one should not separate traffic on source IPs

If the traffic were to be separated on source IPs, then a few questions arise:

1. **How to address the problem of source IP spoofing.** For example, in a TCP SYN flooding attack, if the source IPs were spoofed, then the system would end up doing only traffic separation. Every such stream will have only one or two packets (in the case of TCP connection establishment), either a SYN packet (if it were from an attack client with spoofed IP), and a SYN packet

[3]This is a sequence of incoming traffic only. Hence an ACK packet after SYN packet assumes that SYN/ACK packet would have gone to the server

and ACK packet (if it were from a normal client who cared to respond back with ACK packet). With a large number of spoofed SYN packs flowing at an enormously high rate, the packet sequences (streams) would mostly be one packet sequences. Streaming takes most of the time, and the learning system will cause enormous delay.

2. **When to conclude to go ahead processing the packet sequence from a client.**

   (a) Due to the same problem of network latencies, the ACK packet in the above example from a legitimate client may take a long time to reach the system. But if the system were to conclude due to the delay that the client did not respond and goes ahead with processing only the SYN packet sent by the client, then there is every chance that this might be termed an attack (but it really was not). This behaviour of the client packet is also typical of a bandwidth flooding attack, where packets from legitimate clients are known to have a lot of delay to reach server since bandwidth is occupied by attack packets. So the question of when to stop collecting packets from a stream and going ahead with processing by learning model remains an extremely difficult question to answer, since this is tied to the magnitude and velocity of attack, which is not known prior to the attack itself.

   (b) Another serious problem in sequel to the above is the following:[4] Let the system receive a SYN packet from client A, and A (after having received SYN/ACK from the server) sends back an ACK packet. But due to network latencies, the ACK packet takes unusually long to reach the system, before which the system concludes that A has not replied, and has already passed on this stream to the learning model. Regardless of the learning model's verdict, the ACK packet from A may be received by the system at a later time by which the stream corresponding to A would already have been cleared from buffer. Now, by streaming again, a new stream for A would have been opened, and the first packet for the stream would have been an ACK packet !! This will certainly be classified as anomalous stream by the learning model, since it will definitely expect that the first packet in any new stream would be a SYN packet (since client first requests connection with a SYN packet only).

[4]TCP SYN situation is consistently followed in most examples.

Conclusion: Traffic cannot be separated based on source IPs since in a practical situation, separating traffic based on source IPs may make the system raise too many false alarms due to above reasons.

2.6.4 Conclusion

From the above arguments, one can conclude that the approach as above for packet processing for systems based on learning models appears to be inconclusive – evident by logically arriving at two contradictory conclusions on separation of traffic based on source IPs. Such systems may have sound theoretical results, but may not be practically realisable. The root cause for system failure here appears to be the fact that we are modelling our learning based on the actual data itself – packet sets, preserving the sequence of arrival of packets also. DoS, fortunately is a problem, which can absorb a slight amount of delay in detecting attacks, since a small amount of attack packets do not cause catastrophe on a server, unlike intrusion packet sequences. If we were to work on metadata like packet statistics, then we need not worry about the legitimacy of specific packet sequences (although specific packet sequences mean a lot for intrusion based attacks), although we may be able to sufficiently pass a verdict on the network state – Attack or Normal.

Examples of such parameters are no. of SYN packets, no. of ACK packets given no. of SYN packets, no. of source IP packets, etc. The time/sequence dimension associated with such modelling will be lost, but we will still be able to design a working system which will conclude reasonably successfully the state of the network.

Such a system can be as simple as a straightforward system based on packet thresholds: Only N SYN packets are allowed to hit the server in unit time, and excess SYN packets may not be allowed. The system can also be as complex as a probabilistic system which models various probability distributions for the various TCP flags, arrives at conditional probabilities (ACK is only followed by SYN) during a learning phase, and during deployment phase shall determine probabilities of conditions on packet statistics, answering questions like "What is the probability that 5 ACK packets occur during unit time, given that 8 SYN packets have occurred during the same unit time".

2.6.5 Important Lessons Learnt

From the experience of trying to implement a hardware based DDoS protection system, the following lessons were learnt:

1. Modelling traffic by preserving the sequence may be an overkill for the DoS and DDoS detection problem, and may also prove to backfire in a DDoS situation.
2. Given the nature of the DoS problem, and also the requirement of detection at line speeds, simpler models which work with aggregate data may prove to be effective.
3. Trying to separate traffic based on source IP may not be a good idea, particularly in the case of DDoS attacks. Instead, the algorithm should be able to work with interleaved data.

CHAPTER 3

General Aspects of System Design

In this chapter, we discuss certain important aspects of DDoS protection system design, which paves way for crucial decisions taken on the detection process.

3.1 Crucial Parameters in System Design

Any system design process begins with user requirements. It will be easy to see that every decision made with respect to the system contributes to one or more such requirements. Since this work this was part of a project which was of research nature, the design started with a broad set of user requirements. The requirements and the issues surrounding them which have been addressed are mentioned below.

3.1.1 User Requirements

The following were the broad requirements considered at the outset:

- **A DDoS mitigation system**, which
    1. Can protect multiple targets at a time;
    2. Can work with minimal deployment delays;
    3. Can work at "line speeds", ensuring minimal processing time in both normal/attack situations;
    4. Can reduce reaction time by speedy attack detection;
    5. Is capable of detecting new attacks; and
    6. Is finally implementable in dedicated hardware.

3.1.2 System Design Issues

The primary issues pertaining to system design, keeping in mind the above set of requirements, manifest themselves in the following questions:

- **What could the detection strategy be**. The detection strategy should be in a way that it will aid the detection of new attacks.
- **What features of network packet could be the input**. The features/attributes should be chosen in a way which will let the system differentiate between normal state and an attack state.
- **What classifier model to use**. The classifier model should be able to work a) At line speeds; and b) Occupy lesser hardware space, making it more suitable for hardware implementation. An obvious cost would be the accuracy with which the traffic is monitored. Hence, the right balance should be struck between time, space, and accuracy of modelling by choosing such a classifier.
- **How to ensure faster reaction times**. The feature and the pattern of observation chosen for modelling should let the system detect attacks then and there, and not wait in anticipation for an event to finish. An obvious question that has to be answered is "Should monitoring be temporal or spatial ?".
- **How to make statistical system work with small amounts of data**. The system should be able to work with lesser amount of training data.
- **When and how to restore system after an attack has been detected**. The system should provide a means of reacting to an attack, and have a strategy to restore services at some point.

Further sections precisely try to cater to one or more of these questions.

3.2 Location of the Protection system in the network

A crucial decision on the position assumption of the system is the first step towards network modelling for DDoS attacks. Based on the location of the protection system,

the research problem and it's associated constraints could be different. For example, if it is assumed that the location of the system is near the source of attacks, then an important problem to be addressed is to look at the formation of bots inside the source network, and prevent such formations. However, This will also involve dealing with lesser volumes of traffic flow at a time, and may allow slightly heavier computational techniques to be employed.

The protection system could be in one of the following locations:

- **Near the Target**. The location of the protection system is near the target[4, 29, 55]. This automatically implies that the protection system has to deal with large volumes of traffic striking it at very high speeds. However, there may be a limit on the number of such targets being protected.
- **Near the Source**. The location of the protection system is near the source [41, 52, 32]. In this case, the system should ensure that the host network or part of it does not become a botnet. This calls for traffic modelling for botnet formation, encryption patterns, command and control network monitoring and similar such methods. The methods intending to do protection near the source may not have to necessarily deal with very large volumes of traffic.
- **Intermediate Point on the Network**. The location of the protection system is an intermediate point on the network, where the intermediate network devices possibly talk to each other, to form a distributed solution[44, 66]. It can be assumed that the location is at the periphery of an intermediate network, or on a very large network like the Internet. This means that the volumes to be handled by methods assuming protection at intermediate networks is reasonably high. Also, a fact to be considered in modelling traffic is that the entire traffic for a particular target may not be available with a single such point. In that case, modelling of only a subset of entire traffic may be possible. Alternately, the devices should exchange statistics in order to have a full view of inbound traffic to particular targets.

The protection system, it is assumed, is located near the target. A pictorial representation is as shown in Figure 3.2.

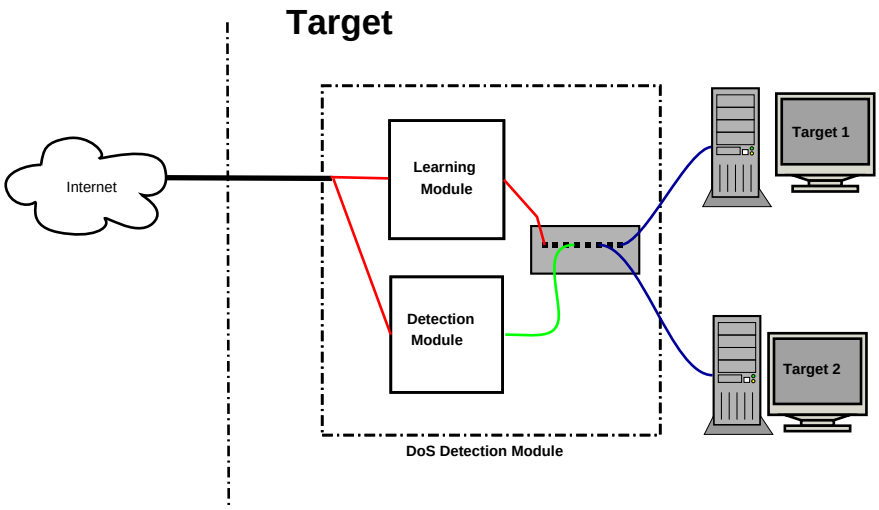


Figure 3.1: Location of the DoS Mitigation Device

3.3 Detection Approach

While signature based detection models build model on attacks and detect attacks based on attack signatures, anomaly based detection models build model on normal traffic and determine attack as one which deviates significantly from normal traffic. The latter is more suited to detecting novel attacks and hence was chosen.

3.4 The Classifier Model

The following promising models were discussed for being the classifier. Emphasis was on implementation complexity and detection time.

- *Hidden Markov Models (HMM)* – Best at temporal sequence modelling, and reported good performance statistics for detecting some DoS attacks by previous work [32]. But heavy weight considering implementation, adding to the fact that sequence modelling may be an overkill for the DoS detection problem.
- *Markov Models (MM)* – Simpler models with lesser number of computations in comparison with HMMs. However, an optimal number of states has to be developed on the fly per user site, which makes it unsuitable for practical implementation.

- *Support Vector Machines (SVM)* – Simpler than the above two models, but is not ideal for zero day response to new attacks, as the model needs to be trained in all classes for accurate detection.

- *Naìve Bayes (NB) Classifiers* – Simplest of the models and ideally suited for the DoS problem, given it's nature of allowing for simplified independence assumptions and the inherent nature of the DoS detection problem, which lets us make such assumptions.

Naìve Bayes Classifier – An Introduction

A Naìve Bayes (NB) classifier is a simple probabilistic classifier based on applying Bayes' theorem with naìve independence assumptions. A Naìve Bayes classifier assumes that the presence (or absence) of a particular feature of a class is unrelated to the presence (or absence) of any other feature. NB classifiers can be trained very efficiently in a supervised learning setting. The fact that NB classifiers can work with small amounts of training data, and can also accommodate a large number of attributes makes them a good choice for network modelling for DoS attacks.

A Naìve Bayes Classifier is formally defined below:

- Let D be a training set of tuples, along with their associated class labels. Each element in the set is an attribute vector $X = \{x_1, x_2, x_3, \ldots, x_n\}$. Let there be m classes, and the set C represent the set of class labels, $C = c_1, c_2, \ldots c_m$. So, each tuple in D is represented as $\{C_i, X\}$.

- Given that an attribute vector X has occurred, classifying X amounts to maximising the value of $P(c_i|X)$, that is, finding out the maximum association of the evidence X with any of the classes in C.

- The value of $P(c_i|X)$ is derived from Bayes theorem

$$P(c_i|X) = \frac{P(X|c_i) \star P(c_i)}{P(X)},$$

  where $P(c_i)$ denotes a priori probability.

- Naìve Bayes Assumption: The attributes $x_1, x_2 \ldots x_n$ are independent. Thereby, $P(X|c_i) = P(x_1|c_i) \star P(x_2|c_i) \star \cdots P(x_n|c_i)$

- Classifier output $i = \arg\max_{c_i} P(c_i|X)$

A very intuitive description of Naìve Bayes classifiers and their applications is presented in [46].

3.5 Stream based Separation

If there are multiple targets behind the DMM, mitigation involves identifying precisely the target being attacked, so that mitigation could be applied only on the affected target. Hence, traffic needs to be separated. separation is done on two levels – based on the layer 3/4 protocol employed (TCP or UDP), and based on destination IP/destination port (application layer) combination.

3.6 Windowing

Windowing essentially means splitting input traffic into traffic subsets, called windows. Analysis will be done on every single window (traffic subset) to arrive at a verdict on the network state. Windowing is important for the following reasons:

1. For practical DoS mitigation systems, it is important to quickly detect DoS attacks at some point after they have been launched, act upon the attack, and also resume services after a point. For all these purposes, analysis needs to be done in small subsets of input packets. Hence, windowing is essential in such cases.

2. During the entire learning phase, a few million packets may have been observed, but we may not be considering variations in traffic at different times of the day if windowing were not done. While there would have been variations in traffic, the average traffic over the entire learning period (as a single window) is what would have been represented by the probabilities that we have learnt during learning phase. This essentially means that we would be failing in totality to differentiate between varying volumes of traffic.

3. It is understood that learning is performed to empower the testing phase with decision making on the state of the network, and hence, analysis in the learning

phase should be done on equal scales as the analysis done during the testing phase. The importance of windowing during testing phase (operation) is established in the first point above, and hence, the same should be followed in the learning phase too in order to provide a verdict on the network state.

4. By not windowing, we are trying to draw conclusions on smaller packet subsets, given likelihood of packets on very large subsets taken collectively during learning. Although uncertainties during different phases of learning are captured by the probability measures, it is always a better technical idea (for reasons stated above) to window the systems so that the system may be more accurate.

3.7 Packet based Windowing

Windowing may be done in two ways:

- Time Windows: Windows based on time slices, occurring every n time units.
- Packet count Windows: Windows based on packet count, occurring for every n network packets.

The important factor that drives the choice among the two is the fact that analysis during learning should help analysis to be done real time.

Packet windows are considered a better choice of the two for the following reasons:

- Packet windows provide smaller reaction times during an attack situation, because of the fact that the system may have to wait for the time window to complete before deciding to flag an attack (or anomaly).
- Packet windows may provide for more accurate modelling, since the number of possible events to be considered is bound to be limited, whereas the number of events to be considered may be large in time windows, since there is no control on how many packets could be received within a time window.

One obvious concern with a packet window is the fact that the periodicity of probability computations may not be regular or even known, since one does not know when n packets would be available for computing probabilities. This may

happen particularly when there is high traffic rate at one time, and low traffic rate at another. But the server can easily handle low volume traffic and related probability computations in any case.

3.8 Smoothing

Smoothing is a common technique used in situations where the system has to deal with sparse training data. Smoothing is the mechanism of artificially injecting a trivial non-zero value (for example, assuming even before the beginning of the experiment that every event has occurred once) to every element in the data in order to avoid ending up with sparse input. Smoothing is very important in the design of practical systems modelling network traffic for DoS attacks. Further information about how smoothing is performed on TCP and UDP is available in later sections.

3.9 Attack Detection

Flagging an abnormal situation as an attack is a function of a series of abnormal windows, not necessarily consecutive. For example, if attack were to be flagged after observing 5 consecutive abnormal windows (say), then the attacker can cleverly escape attack detection by keeping attack traffic to just below 5 windows.

To overcome this problem, a step based mechanism is used. At any point until a reasonable time out period in the course of deployment, if the number of abnormal windows is greater than a particular number[1], then the system flags an attack. This will ensure that attacks such as the one explained above will be caught. The function of the system during deployment phase is formally described in Algorithm 1. It is to be noted that this algorithm applies to both TCP and UDP protocols.

3.10 Practical Design Considerations

The most important factors driving design are the following:

[1]Quantified as a parameter called Abnormal Window Count (AWC)

Algorithm 1: Function *Deploy (S,IT,AWC)*

```
   Input: Stream S, Input Traffic IT, Abnormal Window Count (AWC)
   /* Initialisation                                              */
 1 A ← 0;
 2 for every packet window W in T do
 3 |   P ← determineProbability (W,S) ;
 4 |   if P < Threshold probability of S then
 5 |   |   increment A;
 6 |   else
 7 |   |   decrement A;
 8 |   end
 9 |   if A ≥ AWC then
10 |   |   return attack;
11 |   end
12 end
```

1. **Availability of Training Data**: One of the most important requirements for establishing confidence over a statistical model is to have large volumes of test data. In this context, this amounts to having large volumes of training data to determine what traffic patterns are part of a good network state. This is generally impractical, considering the time for which learning has to take place before the system can be deployed online, particularly in sites where the traffic is not expected to be very high at any point of time. So the system should be able to work with small training data.

2. **Authenticity of Training Data**: The general assumption in a learning based system is that the data supplied to training is tagged and completely normal. This will help the system to completely take all patterns seen during training as normal and determine anomalies accordingly. But in practical situations, it is very hard to guarantee such purely normal training data. The training data might even contain data pertaining to unsuccessful attack attempts, in which case, these attempts will never be classified as anomalies during the deployment phase. So the system should be able to tolerate some amount of abnormal data in the training data and still be able to classify them as abnormal.

   To work around the problem, the error in "normal" traffic is quantified into the system in the form of a parameter called the *Error Proportion (t)*. This proportion indicates the proportion of training traffic (supposedly normal) that is likely to be abnormal. From experiments in limited datasets (tagged and

untagged), it is seen that the proportion varies from 1% in tagged datasets like the DARPA dataset to close to 2% in other untagged datasets.

3. **Dealing with sparse training data**: In many user sites, the training data is observed to be very sparse, that is, the data contains occurrence of very less events out of a large number of possibilities. In this case, a large number of probabilities remain zero, due to which crucial probability information may be lost because of one zero probability event.

3.11 Phases of Operation

The system design consists primarily of two phases of operation:

1. (**Training Phase**) The system takes stream information and traffic statistics corresponding to the stream as input, and produces a data structure S which consists of the NB model probabilities for the different TCP events. This model is used in the deployment phase to determine the probability of a window. *It is to be noted that operations done during training phase are offline operations.*

2. (**Deployment Phase**) The actual operations phase in which the system takes the stream S containing the NB model probabilities for the stream as input, and determines the state of the network at any point in time. In real time, the system takes input traffic and determines if the network state is normal or anomalous.

CHAPTER 4

Design for TCP

4.1 Technical Background

Given that the basic framework of packet based windows has been established, the next task is to decide upon a suitable parameter for monitoring and modelling TCP traffic. Modelling is done based on the protocol headers, to enable faster processing. The choice of parameter for monitoring is driven by the fact that the parameter should be let us differentiate between a normal pattern and an attack pattern.

In a packet window, modelling is done based on the TCP flags set in the packet, building upon the experimental knowledge acquired in [23]. The TCP flags field is a collection of 8-bits, each bit representing one flag. Occurrence of individual flags or combinations of flags in a series of inbound and outbound packets symbolise specific actions in TCP - For example, connection establishment, connection closure, requesting data, etc. The different TCP flags are: SYN, ACK, PSH, FIN, URG, RST (standard TCP flags), ECE, and CWR. The last two flags, it was observed, are seldom used in contemporary Internet traffic, and so we are concerned only about events arising out of the remaining 6 flags.

Hence, there are 2^6 different observables. A packet window is technically defined as a collection of observables (flags set) observed with every packet in the window. The probability of a window is a function of individual probabilities of different observables observed inside the window.

4.1.1 Independence Assumption

Since TCP is a causal system, there exists dependencies between the occurrence of packets with different flag combinations inside a connection. For example, it is likely that we get to see atleast an equal number of ACK packets compared to that of SYN packets, in the assumption that all clients close their connections properly. More

specifically, if we are looking at sequence of occurrence, then we can see that an ACK packet in a session with a client would be preceded by atleast one SYN packet. This implies that there exists dependencies between different observables in TCP.

However, this applies more strongly only when the traffic is source-separated. In our case, the objective is to work with interleaved traffic, where the traffic is not source separated. Furthermore, analysis will be done in terms of small packet windows of fixed size. Also given that different sources may be in different stages of a connection not necessarily occurring in the same window, the dependencies between the observables tend to become loose inside such small windows (in comparison to the total training traffic), that they may be statistically overlooked. Hence, these are assumed to be independent.

Independence Assumption: Packets with individual observables occur independent of each other inside smaller windows.

It is also well known that not all these observables are prescribed for use by TCP literature. For example, TCP does not define the behaviour of servers on observing packets with "abnormal" flag combinations set (for example, SYN and FIN flags in the same packet!). This observation is important in the perspective of implementation – observing these observables separately may give a lot of trivial probabilities, thereby wasting space and also possibly resulting in loss of probability information because of a single zero-probability event. Hence, some kind of grouping of observables has to be done in order to overcome this problem, and also save space.

A set of experiments were carried out with the broad objective of determining the most frequently observed observables (which deserve separate probabilities), and in the process, also identify seldom used observables (in the idea of grouping them in appropriate fashion). Section 6.2.2 gives an account of the datasets used for doing the analysis described below.

- Experiments were conducted on a local LAN for observing communications with server of different packet types. Depending upon the other fields set, for example, acknowledgement number and sequence number, and payload sometimes, responses varied from situation to situation.

  For example, the flag combination of SYN, URG and PSH is a seldom-seen combination in normal traffic, and this packet was sent across to server using a packet generator (without payload). The response from the server was a

SYN/ACK packet.

- Experiments on observing seldom-seen packets at the target were done. Using a standalone machine connected to the Internet, a public web server was injected with unusual packet types – For example, SYN and FIN set. When such unusual packet types were sent, it was observed that these packets never reached the server at all [1] (unlike a LAN environment as mentioned above)

After further examination of frequently occurring observables in the used datasets, an initial grouping on inbound packets was done in the following manner:

1. Packets with RST bit set (irrespective of other bits) – 32 packet types
2. SYN packets – 1 packet type
3. ACK packets – 1 packet type
4. FIN/ACK packets – 1 packet type
5. PSH/ACK packets – 1 packet type
6. (Possibly) Invalid flag combinations – 15 identified combinations of flags
7. (Possibly) Valid but seldom seen combinations – 13 combinations.

But the above classification has a caveat in terms of implementation. During the deployment phase, an inbound packet has to be classified into one of these classes based on the flags set in the packet. It would be of great use if this can be done with minimum number of comparisons. For example, even if the first class (packets with RST bit set) has 32 different flag combinations grouped under it, classifying a packet into this class needs only one wild card comparison (if (packet flag `&` 0X40) !=0) to search inside it. Unfortunately, our classification of the last two classes did not have such a pattern. They contained un-uniform flag combinations grouped into them. During deployment, this may require that the observed combination has to be compared with every combination in the group. For example, to classify a packet belonging to class 6, it may take in the worst case 20 comparisons (5+15), in the absence of a wild card pattern search.

[1] These packets were getting dropped on their way through the Internet, before reaching the target.

An additional noticeable fact was that normal traffic of observed datasets showed almost no signs of appearance of packets belonging to the last two groups. This hinted that without loss of precision, these two groups could be grouped into one.

Finally, the following grouping of observables was arrived at:

1. T_1: Packets with RST bit set (irrespective of other bits) – 32 packet types
2. T_2: SYN packets – 1 packet type
3. T_3: ACK packets – 1 packet type
4. T_4: FIN/ACK packets – 1 packet type
5. T_5: PSH/ACK packets – 1 packet type
6. T_6: Rest of the packets – 28 packet types. Includes seldom used packets and invalid packets.

4.1.2 The Distribution – An Overview

The resulting probability distribution considered is broadly described by the following:

1. Observables – Grouped into 6 (T_1 to T_6).
2. Discrete Random Variable X_i per observable group i = number of packets of type T_i observed in a window.
3. Outcome set $O_i = \{0,1,2,3 \ldots N\}$, where N = window size.
4. The probability of observing c_i packets of type T_i in a window is $P(X_i = c_i)$.
5. $P(X_i = c_i) = \frac{number\ of\ windows\ with\ c_i\ packets}{T}$, where T = Total number of training windows.
6. Probability of a window W with c_1 packets of T_1, c_2 packets of $T_2 \ldots c_6$ packets of T_6 ($c_1 + \cdots c_6$ = window size N) = $P(X_1 = c_1) * \cdots P(X_6 = c_6)$, by NB independence assumptions.

Since a window is characterised by the occurrence of packets, each of which belongs to one of these observable groups, the probability of a window is a function of the probability of observing different counts of packets belonging to each of these observable groups. Since the observable surrounding the occurrence of a number of TCP packet types is being modelled, the number of possible events depends on the size of the window, that is, the number of packets in a window. Hence, from the distribution, it can be seen that for a packet window of size N, there are $(N+1)$ instances to be monitored, thereby making the total number of instances in probability space (and hence the total number of floating points in memory) $6 * (N+1)$. But again, for previously mentioned reasons, many instances are very unlikely to occur in normal traffic. For example, the probability of seeing a window full of SYN packets in normal traffic is (and should be) very less. The TCP traffic, in summary, is highly sparse in terms of TCP flag based events, and lesser number of events could be sufficient in modelling the traffic, a reason for choosing NB classifiers with simplistic assumptions.

This may again result in zero probability events, and hence, may be further grouped in order to compress the space occupied in hardware. Furthermore, smoothing may have to be done on these data in order to avoid zero probability events, and simple techniques like Laplacian smoothing (used for speedy smoothing) may produce inaccurate results when done on large number of instances, given the fact that we started with the assumption of working with smaller amounts of training data. A typical example would be that of window size 100 and a training data of 1000 windows, in which case, 606 values may have to be smoothed given that there are only 1000 training windows. Hence, it is important to group these instances of observable events in some meaningful fashion.

Initially, grouping was tried with the following strategy:

- Groups with contiguous instances, and with fixed band size. For example, for window size $N = 100$, and for the SYN type (T_2), group instances in the following manner:
  - Band 1: Events (windows) with SYN = 0 - 10.
  - Band 2: Windows with SYN = 11 - 20; and so on.

  Although this was a simple and straight-forward way of grouping and could result in a constant number of bands making it implementer-friendly, this method

may not allow accurate capture of events. Datasets followed specific patterns with respect to the number of windows seen for each instance, and this strategy will only result in completely neutralising it, by arbitrarily grouping some high probability events (instances with most number of windows) with some low probability events, by catering to contiguous band grouping.This was also experimentally confirmed.

- With a closer observation of datasets, considering separate bands for the first few highly likely events (since these events will bias the probabilities of other lesser likely events on being grouped with them), and grouping the other less likely events into bands. For example, for the SETS dataset, the following grouping was attempted:

  - RST: Windows with 0 RSTs into band 1, windows with 1 RSTs into band 2, windows with 2 RSTs into band 3, windows with >3 RSTs into band 4.
  - SYN: Windows with 0 SYNs into band 1, windows with 1 SYNs into band 2, windows with 2 SYNs into band 3, ... windows with 9 SYNs into band 10, windows with >9 SYNs into band 11.
  - ACK: Windows with <31 ACKs into band 1, windows with 31 ACKs into band 2, windows with 32 ACKs into band 3 ... windows with 100 ACKs into band 70.
  - FIN/ACK: Windows with 0 FIN/ACKs into band 1, windows with 1 FIN/ACKs into band 2, windows with 2 FIN/ACKs into band 3,... windows with 9 FIN/ACKs into band 10, windows with >9 FIN/ACKs into band 11.
  - PSH/ACK: Windows with 0 PSH/ACKs into band 1, windows with 1 PSH/ACKs into band 2, windows with 2 PSH/ACKs into band 3, ... windows with 69 PSH/ACKs into band 70, windows with >69 PSH/ACKs into band 71.
  - Others: Windows with 0 other packets into band 1, windows with 1 other packets into band 2, windows with 2 other packets into band 3, windows with >2 other packets into band 4.

  Obviously, this was a more accurate way of modelling traffic as the experimental results suggested, but again there were two important drawbacks – This kind of

grouping may not be a one-size-fits-all solution, as experimental results again suggested that when the same logic was applied to the DARPA dataset, the model probabilities were skewed. The more important drawback with respect to implementation is that the number of bands per observable group is not maintained as a constant across observable groups. This may result in varied number of lookups during detection, which is undesirable when it comes to managing space on hardware.

A view of the pattern of the number of windows with different events for each observable group per dataset is shown in Figure 4.1. The graphs are completely different from each other, which proves that

- Grouping with contiguous instances will not work, as can be seen in the graphs, particularly in the SETS SSH server pattern of FIN/ACK packets (number of windows with 17 FIN/ACK packets). If contiguous instances were to be brought into the same group, then the probability of the FIN/ACK group containing the above said event will be dominated by it, skewing the actual likelihoods of the other events in the group.
- Custom grouping for bands as explained above will not work, given the diversities of traffic to different destinations, as shown in Figure 4.1.

Lesson Learnt from Grouping Experiments

The wisdom acquired in the above experiments got translated into the following important design decisions:

1. It was decided that the number of bands to which these instances have to be grouped would be a constant, in order to have a fair idea of memories that may be required to store the entire distribution. This will bring down the total number of instances in the event space to a standard (lesser) constant $6 * K$ from $6 * (N + 1)$, where N = window size.
2. While grouping instances, it is to be ensured that only those which are as likely as each other are to be in the same group. These instances may not be contiguous in nature – instances $k = l$ and $k = l + 1$ may not fall into the same group.

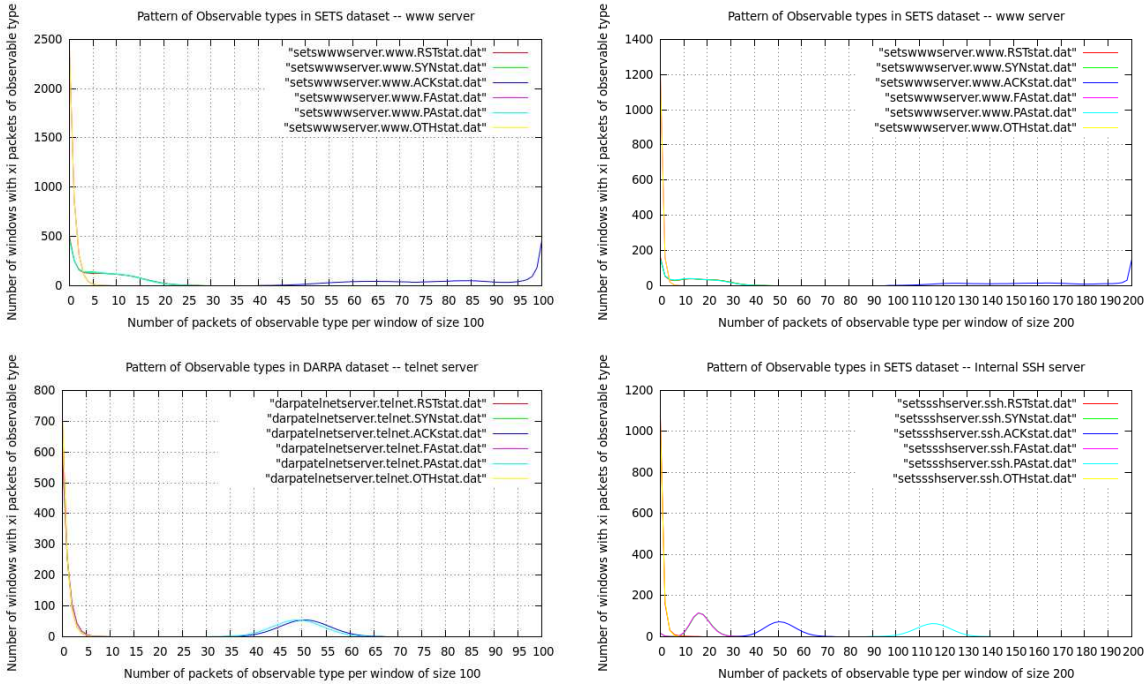


Figure 4.1: Patterns in observable groups in two datasets – SETS and DARPA dataset

For the purpose of implementation, it is assumed that instances per group T_i have to be grouped to K constant bands, and this number is independent of window size. For example, if SYN observable group was considered, and window size $N = 100$, then the bands for number of windows in which n SYN packets (n varies from 0 to 100) are found could be the following: First band $b_1 = \{k = 0, k = 23, k = 54, k = 13, \ldots\}$, second band $b_2 = \{k = 18, k = 27, k = 56, k = 99, \ldots\}$ and so on. It is to be noted that these bands are disjoint sets, and the union of these sets results in the event space. These indices may also vary from site to site.

4.2 Training Phase

4.2.1 Training Goals

In summary of above arguments, the following can be considered the primary goals of the learning phase:

- To determine the optimal grouping of observable instances into constant K

number of bands seen during learning for each TCP observable group, depending on their occurrences;

- To observe every window of normal traffic and update probabilities for each of the above bands and determine their final probabilities at the end of learning; and
- To determine threshold probability (window probability used during deployment, below which the window will be termed as abnormal) based on the error proportion (t) of learning traffic. The value t denotes the percentage of learning traffic which may possibly be abnormal, and captures the practical view that even learning traffic may contain abnormalities.

4.2.2 Input

Input to the training phase is the following information, organised into a data structure.

1. Stream information – Containing the following:
   (a) Target IP address
   (b) Target (canonical) host name
   (c) Target port number (application layer)
   (d) Target port name (canonical)
   (e) Window size (N)
   (f) Error proportion t of learning traffic.
2. TCP Stream traffic statistics (per stream) : Traffic statistics denote the number of packets for each of the 6 TCP packet classes in a window of the stream. For example, if $N = 100$, then a typical example of statistics for one window could be the tuple (2 5 49 43 4 0), where the 6 numbers represent the counts corresponding to the 6 packet classes T_1 to T_6.

4.2.3 Data Structures

Every TCP stream is captured into a single data structure. The definition of the data structure in C language notation is as follows:

```
struct TCPstream
{
char *destination_ip;
char *destination_hostname;
char *destination_portnum;
char *destination_portname;
int N;/*window size*/;
int K;/*Number of instance groups*/
float t;/*error proportion*/
int total_windows_in_learning;/*Training window count*/
int C[6];/*packet counters falling into 6 observable groups*/
float **W;/*Array of learnt probabilities*/
float threshold_probability;/*Function of t*/
};
```

Here, variable W is a 2-D array which represents the actual learnt probabilities of each event for each packet type. For example, S.W [3][7] denotes the probability of observing 7 packets of type T_3 in a window. Variable C is a 1-D array which represents local count of packet types per window. Variable K denotes the number of bands of events per packet type.

Note on Algorithm Inputs

The primary inputs to all algorithms are a subset of the following parameters:

1. **Number of training windows (W).** Denotes the number of input packet windows used for training the classifier. Typical values of W could be in the range of a few thousands.
2. **Window size (N).** Size of packet window. This is a limited variable whose scope is within a few hundreds.
3. **Number of Bands (K).** Number of bands for grouping Observable instances. This is assumed to be a *constant*.

A generalised relation between these three variables could be expressed as the following:

$$K < N << W$$

Typical values of the above are $K = 40$, $N = 200$, $W = 2000$.

4.2.4 Algorithms

Algorithms 2 and 3 describe the training phase[2] function for TCP. Stream information as obtained from the user is updated into the stream structure S before passing as input to the training phase.

Complexity

The complexity of Algorithm 2 is the combined complexity of function TCPtrain () and TCPdetermineThresholdProbability (). The complexity of the algorithm is dominated by two important operations[3] – a O (W) operation, and a O (N^2) operation. Even though N is considered to be small compared to W, N^2 could easily outclass W in a typical setting. Hence, the complexity of the algorithm is O (N^2). This is easy to perform for typical values of N, which range between 50 to 250.

The complexity of Algorithm 3 involves the following operations:

- A O (W) operation in updating instance counters for every window;
- The instance grouping function TCPdetermineOptimalBands (), an O (N^2) function
- The probability updation function TCPupdateProbabilities (), a O (N) function.

Hence, the complexity of the algorithm is O (N^2).

[2]The terms "learning" and "training" are used interchangeably
[3]explained in the complexity arguments below

Algorithm 2: Training phase functionality for TCP

Input: Partially update stream structure S, Traffic statistics TF, window size $S.N$, number of bands $S.K$, error proportion $S.t$.

Output: Updated Stream Data Structure S with learnt probabilities and threshold probability for stream.

```
/* Update model probabilities                                    */
```
1 $S \leftarrow$ TCPTrain (S,TF,S.N,S.K) ;
```
/* Determine Threshold probability for the stream                 */
```
2 S.threshold probability $\leftarrow$ TCPdetermineThresholdProbability (S.N,TF,S.t) ;
3 **return** S;

4.2.5 Instance Grouping

The motivation behind grouping observables into different events has already been discussed in the previous section.

Problem

Given an array of numbers (size of array equivalent to window size+1) $A = \{a_0, a_1, a_2, a_3, \ldots a_N\}$, where N =window size, Divide the array into K groups, such that each element is closest in likelihood to every other element in the group.

Methodology

From the description of the above problem, it can be seen that the problem translates to clustering events to a fixed number of clusters (K).

Objective Function

The objective function of clustering is formally described in the following manner:

- Terminology:
  - Let the input array be A.
  - Let cluster I be represented as C_I, and i_{th} element of C_I be represented as C_I^i.

Algorithm 3: Function *TCPTrain (S,TF,N,K)* – Learning Module Algorithm for TCP

Input: Stream S, Traffic Statistics TF, Window size for stream $S.N$, number of bands $S.K$.

Output: Updated Stream Data Structure S with learnt probabilities.

```
1  for Every window in TF do
2  |  Populate S.C_i s for each of the 6 T_i s;
3  |  for i = 1 to 6 do
   |  |  /* Update corresponding event occurrence.                    */
4  |  |  Increment S.W[i][S.C_i];
5  |  end
6  |  Increment S.total windows in learning;
7  |  Reset S.C_i s;
8  end
   /* Determine Optimal bands for each row of W, compute
      probabilities                                                    */
9  for i = 1 to 6 do
10 |  band ← TCPdetermineOptimalBands (S.W[i],S.N,S.K) ;
   |  // Determine band ranges for row i
11 |  S ← TCPupdateProbabilities (S,S.K,band,i) ;
   |  // Determine probabilities for row i
12 end
13 return S;
```

  - Let C_I^{min} and C_I^{max} represent the smallest and largest elements of cluster C_I respectively.
  - Let the distance between two elements of C_I be represented as $d_I^{ij} = C_I^i - C_I^j$.

- *Objective Function*: Group A into fixed K clusters, such that
  - Value $\sum_{I=1}^{K} \sum_{C_I^i, C_I^j} d_I^{ij}$ is minimised.
  - Value $\sum G_{IJ}$ is maximised, where $G_{IJ} = C_I^{max} - C_I^{min}, \forall$ clusters I, J.

Two approaches were attempted in solving the above problem – a) K-means clustering, a well known, scientific approach to solve the problem of clustering one-dimensional data, and whose objective function is an approximation of the above objective function; and b) A Heuristic clustering approach, termed as "Jump based clustering", which aims to maximise the distance between clusters.

K-Means clustering – Formal Approach

K-means clustering is one of the most popular clustering methods used to cluster single dimensional array points, when the number of clusters is a known constant. This property of K-means clustering, along with the additional fact that the small input size ($N + 1$ elements) does not encourage sophistication in clustering, made K-means the obvious choice.

The objective of K-means clustering is to cluster the input array into constant K clusters in such a way that the individual cluster elements are closest to the cluster means.

In technical parlance, the objective function of K-means clustering is described below:

- Terminology:
  - Let the input array be A.
  - Let cluster I be represented as C_I, j_{th} element of C_I be represented as C_I^j.
  - Let the mean of cluster I be represented as C_I^μ.
- *Objective Function*: Group A into fixed K clusters such that
  - The value $\sum_{I=1}^{I=K} \sum_{j \in I} ||C_I^j - C_I^\mu||^2$ is minimised.

The k-means clustering algorithm follows a greedy strategy, by starting with arbitrary chosen values for cluster means for each of the K clusters. An array point A_{val} is classified as falling into the cluster I with which it's distance from the mean $(A_{val} - C_I^\mu)$ is minimal (since we are trying to minimise this error). After a classification of each array point has been obtained, the cluster means C_I^μ for all Is are recomputed with the new cluster points. The algorithm is said to have converged when the changes between means in subsequent iterations is negligible or zero.

K-means algorithm also inherits the concern of greedy strategy – of getting struck at local optima. In a practical situation, this can be overcome by running the K-means algorithm with multiple initial means (50 in this case) and choosing an optimal set of means which give the best results in terms of the problem – the one which provides the least deviation from actual probabilities (or it's derivatives).

At the end of banding, the probabilities of all elements inside a single cluster will be the same, and is the mean of the cluster. Since the probabilities will be impacted by the number of windows for each event (which is what the input array in this case signifies), the values of array are recomputed to have the cluster mean value, in a modified array A_{mod}. In order to determine the initial means which provide the least error, the sum of errors of actual input values (elements of original array $\{A\}$) with the modified values (elements of modified array $\{A_{mod}\}$, whose values are recomputed based on clustering) is computed for each of the 50 iterations. The initial means which has the least sum of errors is chosen as the most optimal.

In summary, the K-means clustering algorithm is explained in detail in Algorithm 4.

Jump-based Clustering – Heuristic Approach

The basic principle behind jump based clustering is the fact that events which are equally likely (events with equal number of windows) should be on the same cluster. Since the objective is to maximise the distance between clusters, the cluster boundaries would be on positions in a sorted array, where the distance between two elements is maximum.

The objective is accomplished in the following fashion – the input numbers are first sorted. Sorting these numbers brings like instances closer. In order to do optimal grouping, each group is characterised by a jump (in values) that it has with the last element of the previous group. With the sorted array, the jumps between successive numbers (it is to be seen that estimating jumps between successive numbers will clearly isolate surges into a single group, since one big number between small number neighbours will automatically induce two jumps. This situation is not desirable.) is computed. The job on hand is to estimate the boundaries of fixed number of partitions in the sorted sequence in accordance with the jumps. If K bands are to be formed, then the top $(K-1)$ jumps are taken and partitions are drawn in the sorted sequence in front of them. The indices corresponding to the original sequence of numbers in the sorted sequence in each partition will form the elements of the band. The number of windows for each band is computed as the average number of windows occurring for events in the band. The band probability is computed as the average number of windows for the band by the total number of windows during learning.

In summary, the following algorithm is used:

1. Observe occurrences for each event, and consider jumps between event occurrences in descending order. The jump between two observations of events in the array is denoted as $J_i = O_i - O_{i+1}$.
2. Consider top $K-1$ jumps in J_i s and form K bands bordering them.

This also reaffirms an important property of input data – the banding may not always result in contiguous instances falling in the same band. Sorting of input data automatically jumbles up the numbers and defeats contiguity.

Conclusion on Clustering

A performance comparison was made between the two methods, along with K-means clustering with a single arbitrary initial means. The False Positives, True Negatives, True Positives and False Negatives were measured for experiments pertaining to four datasets – SETS dataset, DARPA dataset, Auckland - I dataset, and Auckland - II dataset. The methodology of performing these tests are as explained in Section 6.2.3.

The following were the tried clustering methods:

1. **K-means clustering with single arbitrary set of initial K means (C_1)** : K-means clustering is used to divide the observable instances based on their occurrence. However, only a single arbitrary set of K numbers is taken as the initial set of K means. Tolerance for convergence of means in subsequent iterations is taken to be 1 percent.
2. **Jump based clustering (C_2)** : The observable instances are grouped into K groups or clusters based on the top $(K-1)$ jumps of values found between them.
3. **K-means clustering with optimal initial means (C_3)** : K-means clustering is used to cluster the observable instances. However, multiple random sets of initial means are tried (50 attempts), and the one which introduces the smallest error in the actual observable instance values is chosen. Tolerance for convergence of means in subsequent iterations is 1 percent. Initial seed for random number generator = 2.

The results are shown in Table 4.1.

From the table, it can be seen that the two methods appeared to be equally efficient. Hence, a more scientific analysis technique was considered to compare these two clustering techniques. The number of false-negatives was the yardstick, and the **students t-test for independent samples** was conducted with 20 random (training) input samples.

The results of the t-test prove that **both mechanisms show no significant difference in performance**. For want of simpler and more efficient handling techniques, the *jump-based clustering was preferred over K-means clustering.*

Algorithm

The jump-based clustering described above is presented formally in Algorithm 5.

Example

Let window size $N = 10$.

Let the set representing number of windows during learning against each event (from $n = 0$ to $n = 10$) be $A = \{500, 291, 271, 36, 222, 111, 1211, 3, 1, 31, 1\}$, corresponding to packet type T_i. Let the fixed number of bands $K = 5$.

Sorted array $A_s = \{1211, 500, 291, 271, 222, 111, 36, 31, 3, 1, 1\}$, which brings like events closer.

Jump array $J = \{711, 209, 20, 49, 111, 75, 5, 28, 2, 0\}$.

Top 4 jumps $J_s[0] \dots J_s[3] = \{711, 209, 111, 75\}$.

Index array $I = \{0, 1, 4, 5\}$.

Sorted Index array $I_s = \{0, 1, 4, 5\}$.

Band groups: $Bg_1 : A_s[0]$ (1 value), $Bg_2 : A_s[1]$ (1 value), $Bg_3 : A_s[2] - A_s[4]$ (3 values), $Bg_4 : A_s[5]$ (1 value),$Bg_5 : A_s[6]A_s[10]$ (5 values).

Bands[indices]: $B_1 : \{6\}$, $B_2 : \{0\}$, $B_3 : \{1, 2, 4\}$, $B_4 : \{5\}$, $B_5 : \{3, 9, 7, 8, 10\}$.

Complexity

The function TCPdetermineOptimalBands () is dominated by the sorting of numbers, and hence has a theoretical complexity of O ($NlogN$), although for practical implementations, N will be small and the complexity in that case would be O (N^2).

Memory for the Detection Phase

In summary, the detection system maintains the model probabilities which are described by the following per observable group T_i:

B_1: $(I_{11}, I_{12}, \ldots\ldots, P_1)$
B_2: $(\ldots, \ldots, \ldots, P_2)$ and so on, with K number of bands, where B_is are the bands, I_{11}, I_{12} are the instance indices falling into the particular band, and P_is are the band probabilities.

4.2.6 Probability Updation

Problem

Given 2-dimensional array band of K number of rows (that is, K number of bands), determine the probability for all possible events for the packet type.

Methodology

After determining the band indices, determining probabilities is done by the following principle: Probability of all events within a group should be the same, and should be the mean probability of all events. However, smoothing needs to be done prior to determining probabilities.

The method explained above is described in Algorithm 6.

Complexity

The complexity of the function TCPupdateProbabilities () is dominated by operations done on individual elements of each band, whose sum total is the window size

N. Hence, the complexity of the function is O (N).

4.2.7 Thresholding

Problem

Given learning traffic statistics for a site, determine the threshold probability below which an event will be considered abnormal.

Methodology

For the purpose of determining a threshold probability, a technique called *Cross Validation* is adopted. The following is the philosophy behind cross validation: While there is variability in traffic in terms of some events appearing more likely than a few others, it is assumed that this variability is reflected in the learning traffic also. So, the idea is to simulate the fact that detection actually has to work on unseen data, by making some of the training data itself unseen, and get trained with the seen data. For the assumed unseen data, the detection module estimates probabilities. This in essence, gives us candidate probabilities for threshold, as these are determined by training only with a subset of the learning traffic. A suitably low probability estimated in detection becomes a candidate for threshold probability.

An important design consideration is to eliminate the effect of bad traffic that may be present in the training traffic. During cross validation, there is a possibility that most of "unseen" traffic set (in spite of randomly picking the traffic for the unseen set) might be bad traffic, in which case the threshold probability will be lesser than an ideal choice (since the probability of most unseen windows would be very low as estimated by the model built with good traffic). This is undesirable. So the unseen traffic should be sampled at random more than once, before a threshold probability is arrived at. Our algorithm does 10 such iterations.

A single experiment consists of training the model using a group G consisting of $\frac{9}{10}$ th of learning traffic (picked at random from the complete traffic), and estimating probabilities of a blob B consisting of the remaining $\frac{1}{10}$ th of traffic. The learning traffic (that is, window statistics) is split into 10 groups (10-fold cross validation[4]) of

[4]cross validation done by splitting input dataset into n groups is called n-fold cross validation.

traffic (that is, windows) at random. The system is trained on every possible 9-group combination, and probabilities computed on the remaining blob of traffic. This will result in a number of probabilities, from which a suitably low probability will be chosen as the threshold probability.

The error proportion (t) is taken as an input to the algorithm. If the error proportion were t percent, then the lower t percent of probabilities calculated in the learning-cum-estimation phase described above will be eliminated. The threshold probability will be the least probability in the set after t percent of probabilities are left without being considered. The algorithm uses a Random Number Generator Rng () which will be used for determining which group a particular window statistic shall belong to.

Algorithm

The methodology is formally described in Algorithm 7.

Complexity

The complexity of the algorithm is dominated by the iterative TCPTrain () function run on the individual groups split from the training traffic, and hence the complexity of the algorithm is O (N^2).

4.3 Deployment Phase

The deployment phase is very similar in functionality to the training phase in terms of computing statistics. The system takes as input the data structure S containing the NB model probabilities for the given stream, and determines if the network state for the stream is normal or anomalous at any point in time. It is important to have a light-weight deployment phase activity for the system since this is done online. The system is designed to only perform preliminary window statistics computation and a few look ups to determine the probability of a window.

4.3.1 Probability Computation

Probability computation involves computing statistics of a window and determining the probability with which the window would have occurred in the training phase.

Algorithm

The algorithm is formally described in Algorithm 8.

Algorithm 8: Function *TCPdetermineProbability (W,S)* – Probability of a window W for TCP stream S

Input: Stream with NB probabilities S, packet window statistics W
Output: Probability P of window W

```
1 Initialise P to 1;
2 Determine counts of packets with flags of all 6 groups T_1 ... T_6 and update
  counts C_1 ... C_6;
3 for i = 1 to 6 do
4 |   P = P * S.W[i][C_i];
5 end
6 return P;
```

Complexity

The detection mechanism involves a set of constant, light weight operations. More precisely, it includes a constant number of additions for updating TCP flag statistics (N of them) until a window of packets is received, 6 lookups to determine probabilities of individual events in the window, 5 multiplications (or additions in the case of log space operations) to determine the probability of the window, and one comparison operation to arrive at a verdict.

Algorithm 4: Function *TCPoptimalKmeans (A[])* – K-means clustering for TCP

Input: Array $A = \{a_0, a_1 \dots a_N\}$, window size for stream N,number of bands K.

Output: 2-dimensional array *band*, containing indices of per group elements in A.

`/* Iterative K-means process with different random initial means */`

for *i = 1 to 50* **do**
  Set random initial means C_I^{μ}, for $I = 1 \dots K$;
  Set previous values of initial means $C_I^{\mu_p}$ to C_I^{μ}, for $I = 1 \dots K$;
  `/* Apply K-means to the input */`
  for *j = 0 to N* **do**
    `/* Classify point */`
    Associate array element A_j with cluster C_I for which error $e_j = (A_j - C_I^{\mu})$ is minimal;
  end
  `/* Recompute cluster means */`
  Update mean C_I^{μ} with the mean of new cluster elements for each cluster, for $I = 1 \dots K$;
  `/* Compare new cluster means with old values for convergence */`
  if $|C_I^{\mu} - C_I^{\mu_p}| < 0.1 \star C_I^{\mu_p} \forall I = 1 \dots K$ **then**
    `/* Substitute into` A_{mod} `cluster elements with cluster mean value */`
    for *j = 1 to K* **do**
      $A_{mod}[\text{index of } C_j^l \text{ in } A] = C_j^{\mu}, \forall C_j^l \in C_j$;
    end
    `/* Find sum of errors over actual probabilities */`
    for *j = 0 to N* **do**
      $soe_i = \sum |A_{mod}[j] - A[j]|$;
    end
  end
end

`/* Determine the index of initial means with minimal sum of errors */`

Determine the index of the smallest *soe*, i_{min};

return *cluster elements for initial means corresponding to* i_{min};

Dataset	FPs			TNs			TPs			FNs		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
DARPA	5	0	1	701	706	705	7897	11585	11585	3688	0	0
SETS	40	31	29	3982	3991	3993	4064	18162	18162	14098	0	0
Auckland-I	27	25	15	3266	3268	3278	16271	18162	18163	2332	1	0
Auckland-II	57	69	63	6840	6828	6834	130	129	130	1	2	1

Table 4.1: System performance analysis with three clustering methods – Results

Algorithm 5: Function *TCPdetermineOptimalBands (S.W[],K)* – Instance Clustering Algorithm for TCP

Input: Array $A = \{a_0, a_1 \dots a_N\}$, window size for stream N,number of bands K.

Output: 2-dimensional array *band*, containing indices of per group elements in A.

$J[N] \leftarrow \{0\}$;// stores jumps between consecutive values of array A
$J_s[K-1] \leftarrow \{0\}$;// stores the top $(K-1)$ jumps.
upper,lower,nelements $\leftarrow 0$;
$I[K-1] \leftarrow 0$, $I_s[K-1] \leftarrow 0$;// stores indices in A of top $((K-1))$ jumps.
Sort array A in descending order and copy into array A_s;
/* Determining jumps between consecutive values of A_s */
for *i = 0 to N* **do**
  $J[i] \leftarrow A_s[i] - A_s[i+1]$;
end
Determine the top $(K-1)$ jumps (largest numbers) from array J. Store them in $J_s[0], J_s[1] \dots J_s[K-2]$;
Store indices of $J_s[0] \dots J_s[K-2]$ in J into array I;
Sort array I in ascending order and store in I_s;
for *i = 0 to* $K-1$ **do**
  upper $\leftarrow I_s[i]$;
  nelements $\leftarrow$ upper - lower + 1;
  k $\leftarrow 0$;
  band[i][k++] $\leftarrow$ nelements;
  for *j = lower to (lower+nelements)* **do**
    band[i][k++] $\leftarrow$ index of $A_s[j]$ in A;
  end
  lower $\leftarrow$ upper + 1;
end
band[i][K++] $\leftarrow$ remaining elements of A;
return *band*;

Algorithm 6: Function *TCPupdateProbabilities (S,K,band,L)* – Probability determination algorithm

Input: TCP Stream structure S, 2-D array *band* containing band indices of groups of A, number of bands K, Index of observable type L (for writing into appropriate index at S.W)

Output: Probabilities of all events updated into corresponding W index in S

```
1  avgwcount ← 0;
   /* Increment total window to accommodate smoothing                */
2  S.total windows in learning+= K;
3  for j = 0 to K do
4  |  nelband ← band[j][0];
5  |  for k = 1 to (1+ nelband) do
6  |  |  avgwcount ← avgwcount + S.W[L][band[j][k]];
7  |  end
   |  /* Smooth the event group and compute mean                     */
8  |  avgwcount++;
9  |  avgwcount ← avgwcount/nelband;
   |  /* Write the mean value into the corresponding W indices of
   |     the observable type.                                        */
10 |  for k = 1 to (1+ nelband) do
11 |  |  S.W[L][band[j][k]] ← avgwcount;
12 |  end
   |  /* Compute probabilities                                       */
13 |  for k = 1 to (1+ nelband) do
14 |  |  S.W[L][band[j][k]] ← S.W[L][band[j][k]] / S.total windows in learning;
15 |  end
16 |  avgwcount ← 0;
17 end
18 return S;
```

Algorithm 7: Function TCPdetermineThresholdProbability (S,N,TF,t) – Thresholding algorithm for TCP

Input: Stream S, Window size for stream N, Traffic statistics TF, Error proportion t percent
Output: Threshold probability P_t for S

`/* Initialisation */`
Seed Rng () ;
parray[] $\leftarrow 0$;
`/* Grouping Learning traffic */`
Split traffic TF uniformly at random into 10 groups $G_1, G_2 \ldots G_{10}$;
for *i = 1 to 10* **do**
    $G \leftarrow TF - G_i$;
    $B \leftarrow G_i$;
    `/* Learn from` $\frac{9}{10}$`th of traffic */`
    $S_{temp} \leftarrow$ *TCPTrain (SF,G)* ;
    `/* Compute probability of remaining` $\frac{1}{10}$`th of traffic */`
    for *every window W in B* **do**
        $P \leftarrow$ determineProbability (W,S_{temp}) ;
        add P to *parray*;
    end
end
`/* Sieving lower probabilities */`
result $\leftarrow$ ($\frac{t}{100} \star parraylen + 1$)th smallest element of parray;
return *result*;

CHAPTER 5

Design for UDP

5.1 Technical Background

The objective of monitoring protocol headers is to extract a suitable feature, which will aid in detecting DoS attacks on the protocol. The feature about UDP is that it is connection-less, and hence ensures speedy communication. However, the header of the UDP protocol does not contain fields like flags, which will determine the state of the communication, unlike TCP. Since UDP is not connection-oriented, most DoS attacks performed on UDP are only band-width based attacks, essentially trying to exhaust bandwidth resources available to connect to the server. In line with this view of thought, it can be seen that header information cannot be a critical parameter for detecting DoS attacks on UDP.

Hence, the parameter considered here is the **Window Arrival Time (WAT)** of a packet window. Arrival time of a packet window is the duration in which a packet window has arrived. Technically, it is the difference in time between packets P_1 and P_N of a window, where N is the size of the window. Due to various constraints, only non-overlapping windows are being considered. During learning phase, the WATs of windows are monitored, and a model is computed to accommodate these events (WATs). During the operations phase, the probabilities of these models are used to determine the probability of an incoming window. If the probability is less than a threshold probability, then the window is classified as abnormal.

As is the case with TCP, it is observed that the events are so sparse in nature that there appears to be a lot of scope to group these events in order to reduce the number of probabilities handled by the model. This requires that WATs be further grouped into a constant number of bands defined by time bounds. Hence, each band is a collection of a contiguous range of WATs. The probability of a band will be common to all WATs falling under the band. During detection, the WAT of each inbound window is computed, and the probability of the window is the probability of the band inside which the WAT falls. If this probability were to be lesser than

a threshold probability, then the window is considered abnormal. The threshold probability is determined by adopting cross validation methods similar to that done in TCP.

5.1.1 The Distribution

The probability distribution in the case of UDP is a very simple function, owing to the fact that the probability space is a fixed constant, independent of the size of the packet window. It is to be noted that this is not the case with TCP, where the probability space varied with the value of N.

The distribution is best described in the following fashion:

1. Observable: Windows Arrival Time (WAT) of a packet window.
2. Discrete Random Variable X_i: Band under which the WAT of the window number i falls.
3. Outcome set O: $\{B_1, B_2, \ldots B_K\}$.
4. Probability P_j of band B_j $(j = 1, 2, \ldots K) = \frac{number\ of\ windows\ \in B_j}{total\ number\ of\ training\ windows}$
5. Probability of a window i falling into band $j = P(X_i \in B_j) = P_j$

5.2 Training Phase

5.2.1 Training Goals

The primary goals of training with respect to UDP are the following:

1. To compute per UDP stream, the upper and lower bounds of each band of WATs, given that a constant number of bands (K) exist. For example, if window size N = 100, then the different bands could be B_1 = (0–10) seconds[1] , B_2 = (11–235) seconds, B_3 = (236–499) seconds and so on. It should be noted that the band intervals shall span the entire time.

[1]Unit of time is assumed to be seconds.

2. To compute per stream, the probabilities of each of these bands (for example, probability of WAT in B_2) at the end of the learning phase.

3. To compute per stream the threshold probability associated with the stream. This threshold probability is the probability below which the window will be considered abnormal.

5.2.2 Inputs

Inputs to the training phase with respect to UDP stream are the following:

1. Stream Information: Containing the following:

   (a) Target IP address

   (b) Target (canonical) host name

   (c) Target port number (application layer)

   (d) Target port name (canonical)

   (e) Window size (N).

   (f) Number of bands (K)

   (g) Error proportion t.

2. Normal traffic statistics (file) containing WATs of windows.

5.2.3 Data Structures

Every UDP stream is captured into a single data structure. The definition of the data structure is best described in the form of a standard C structure as follows:

```
struct UDPstream
{
char *destination_ip;
char *destination_hostname;
char *destination_portnum;
char *destination_portname;
```

```
char *traffic_stats_file;
int N;/*window size*/
int K;/*Number of bands for partitioning*/
float t;/*Error proportion*/
struct UDPband *udpbands;/*Band intervals and probabilities*/
int total_windows_in_learning;
float threshold_probability;/*Function of t*/
}
```

The structure UDPband is defined as follows:

```
struct UDPband
{
int upperbound;
int lowerbound;
double probability;
};
```

The definition of a band is fundamentally different for UDP due to the way UDP traffic is modelled. The variables *upperbound* and *lowerbound* are the time bounds for the band of WATs and the variable probability denotes the probability of band.

Note on Algorithm Inputs

The primary inputs to all algorithms are/depend upon the following parameters:

1. **Number of training windows (W).** Denotes the number of input packet windows used for training the classifier. This is the largest input, and hence the complexity of algorithms which get directly impacted by W are dominated by it's value. Typical values of W is in a few thousands.
2. **Number of Bands (K).** This is assumed to be a *constant*, and has been experimentally found to be optimal at a value close to 20.
3. It is to be noted that the memory taken for the model probabilities depends on K (unlike in TCP where the space varied with N), and the window size N has no impact on the algorithm complexities.

5.2.4 Algorithms

The functionality of training phase with respect to UDP streams is described in Algorithm 9 and Algorithm 10.

Complexity

The complexity of Algorithm 9 depends on the complexities of functions UDPTrain () and UDPdetermineThresholdProbabilities (). Hence, the complexity of Algorithm 9 is O ($WlogW$).

The complexity of Algorithm 10 is dominated by the complexity of the UDPdetermineOptimalBands () function, which has a complexity of O ($WlogW$). The rest of the operations are pertaining to constant O (K) operations, a few O (W) operations on WAT computation for every packet window, and probability update function UDPupdateProbabilities (). Hence the complexity of the algorithm is O ($WlogW$).

Algorithm 9: Training phase functionality for UDP

Input: Stream Information SF, Traffic statistics TF, window size N, number of bands K, error proportion t.

Output: Updated Stream Data Structure S with learnt probabilities and threshold probability for stream.

1 Initialise S with the number of windows N and error proportion t;
`/* Update model probabilities */`
2 $S \leftarrow$ UDPTrain (SF,TF,S.N,S.K) ;
`/* Determine Threshold probability for the stream */`
3 S.threshold probability $\leftarrow$ UDPdetermineThresholdProbability (S.N,TF,S.t) ;
4 **return** S;

5.2.5 Instance Grouping

Problem

Given an array of numbers (WATs) with size of array equivalent to the total number of windows during learning, $A = \{a_0, a_1, a_2, a_3, \ldots a_{TW}\}$, where TW = Total number of windows seen during learning, Divide the array of WATs into K time bands.

Algorithm 10: Function *UDPTrain (SF,TF,S.N,K)* – UDP Learning Algorithm

Input: Stream information file SF,traffic statistics file TF, window size $S.N$, number of bands $S.K$

Output: Update stream data structure S with learnt probabilities

`/* Initialisation */`

Initialise UDP stream S with values for B, destination IP, destination port, etc., from SF;

Allocate $S.K$ values to array $S.udpbands$;

WATarray ← {0};

i,j ← 0;

bandindex ← 0;

`/* Read statistics and update WATarray */`

Open traffic statistics file TF;

for *every window in TF* **do**

    Write WAT of window into WATarray;

    S.total windows in learning ← S.total windows in learning + 1;

end

Close TF;

`/* Determine optimal bands, expand bands, compute probabilities */`

band ← UDPdetermineOptimalBands (WATarray,S.K,S.total windows in learning) ;

`// Determine band ranges for row` i

`/* Write band values into S.udpbands */`

bandindex ← 0;

for $j = 0$ *to* $(S.K - 1)$ **do**

    S.updbands[j].upperbound ← bands[bandindex];

    S.udpbands[j].lowerbound ← bands[bandindex+1];

    bandindex ← bandindex + 2;

end

`/* Expand bands */`

S ← UDPexpandBands (S) ;

`/* Compute probabilities of each band */`

S ← UDPupdateProbabilities (WATarray,S) ;`// Determine probabilities for row` i

return S;

Methodology

The traffic is modelled in terms of the WAT, that is, the time taken for a (non-overlapping) window of packets to arrive. An approach similar to the approach taken in the case of TCP is taken, although only contiguous time ranges are being considered for grouping, since the learning phase may not have encountered all time variations, unlike in the TCP case where there are only a finite set of events to model. This approach of making contiguous bands has a drawback – inaccuracy of estimation of probabilities is possible if events with like probabilities are distributed non-contiguously like the case of TCP, that is, equally likely events will be put into different bands and may be estimated different probabilities. However, it is assumed that in the case of UDP, consecutive WAT events (example, WAT=500,WAT=501,WAT=502, etc.,) are *most likely* to be equally probable, and hence this strategy can be adopted.

Based on the input set of WATs, K bands of contiguous time intervals have to be arrived at. The input set is first sorted in order to bring WATs likely to be in the same band together. The job on hand is to draw partitions in this sorted set. An approach similar to the approach for TCP is taken. Whenever the jump between consecutive numbers in the sorted list is significantly higher than the rest of the jumps, then a partition is most likely to exist in that point. Since the bands represent time, the sum total of all bands should span the entire time. Hence, $(K-2)$ bands have to be computed. Two more bands will be included to the list – The first band will be any WAT greater than max (WAT s), and the last band will be any WAT lesser than min (WAT s).

In summary, a simple technique is used, as described below:

1. Observe occurrences for each event, and consider jumps between event occurrences in descending order. The jump between two observations of events in the array is denoted as $J_i = O_i - O_{i+1}$.
2. Consider top $K-3$ jumps in J_i s and form $K-2$ bands bordering them.
3. Include two other bands – a band which represents all WATs greater than the maximum WAT, and another band which represents all WATs lesser than the minimum WAT. The total number of bands becomes K.

Algorithm

The functionality described above is formally described in Algorithm 11. The output of the algorithm is an array *result*, which stores elements in the following order: result $= \{B_1^U, B_1^L, B_2^U, B_2^L, \ldots, B_K^U, B_K^L\}$, where B_i^L and B_i^U represent the lower and upper bounds of band B_i.

Example

The algorithm is illustrated with the following example:

- Let array A of WATs be $A = \{500, 291, 271, 36, 222, 111, 1211, 3, 2, 31, 2\}$, and the number of bands $B = 8$.
- Sorted array $A_s = \{1211, 500, 291, 271, 222, 111, 36, 31, 3, 2, 2\}$.
- Jump array $J = \{711, 209, 20, 49, 111, 75, 5, 28, 1, 0\}$.
- Sorted jump array $J_s = \{711, 209, 111, 75, 49, 28, 20, 5, 1, 0\}$.
- Indices with top 5 ranks are stored in $I = \{0, 1, 4, 5, 3\}$.
- Sorted Indices array $I_s = \{0, 1, 3, 4, 5\}$.
- Output Bands[2]: {inf–1212},{1211},{500},{291–271},{222},{111},{36–2},{1–0}
- Output array $result = \{inf, 1212, 1211, 1211, 500, 500, 291, 271, 222, 222, 111, 111, 36, 2, 1, 0\}$.
- **NOTE**: One can see the distinct gap of bounds between upper bound of a band and lower bound of neighbouring band.

Complexity

The complexity of Algorithm 11 is dominated by the sorting function, which sorts the entire input array of length W, apart from a few O (W) operations in determining

[2]If a band contains only a single WAT after clustering, then in the output of the algorithm, the same WAT will be copied into both the upper and lower bounds.

jumps, finding indices of values corresponding to jumps, and a few O (K) operations in building K bands. Hence, the complexity of the algorithm could be taken as O ($WlogW$).

5.2.6 Band Expansion

Methodology

In order to arrive at $(K-2)$ partitions, the jump array is sorted, and the WATs contributing to the top $(K-2)$ elements in the jump array will help form $(K-2)$ middle partitions. However, it has to be noted that the exact boundaries of these bands may not be known from the WATs observed during learning, since the learning phase may not have encountered all possible values of WATs. For example, if the sorted WAT array was $A_s = \{259, 245, 200, 177, 99, 97, 45, 40\}$, and if the bands have been partitioned as $B_i = \{259 - 200\}$ and $B_{i+1} = \{177 - 97\}$, it is difficult to come to a conclusion on the probabilities of events which have not occurred in the learning traffic, for example, WAT = 190. Hence, there should be a mechanism for meaningful expansion of bands in order to accommodate events which were not seen in learning phase.

A simple strategy to expand bands is followed: The difference between the upper bound of a band, and the lower bound of its neighbouring band (the conflict zone) is equally shared between the neighbouring bands. Hence every band (except the first and the last band) can be expanded on both its bounds to accommodate the entire time.

Algorithm

The methodology is described in Algorithm 12.

Complexity

As can be seen with the algorithm, the complexity of Algorithm 12 is a constant set of operations based on the constant number of bands K. The complexity of the algorithm, hence, is O (K).

Algorithm 12: Function *UDPexpandBands (S)* – UDP Band Expansion Algorithm

Input: UDP Stream structure S.

Output: S with *S.udpbands* expanded.

```
1 difference_between_bands ← 0;
2 for j = 1 to (S.K - 1) do
3 |   difference_between_bands ← S.udpbands[j].lowerbound -
  |   S.udpbands[j+1].upperbound + 1;
4 |   S.udpbands[j].lowerbound ← S.udpbands[j].lowerbound -
  |   difference_between_bands / 2;
5 |   S.udpbands[j+1].upperbound ← S.udpbands[j].lowerbound - 1;
6 end
7 return S;
```

5.2.7 Updation of Learning Probabilities

Methodology

Updation of Learning probabilities for UDP streams is straightforward, after time bands have been frozen. It involves re-counting number of windows in the learning traffic falling under each band (according to the WAT of the window) and estimating probability as the number of windows seen in the band by the total number of windows seen during learning. It is to be seen that the window counters against each band have to be smoothed before estimating probabilities.

Algorithm

The above functionality is described in Algorithm 13.

Complexity

The complexity of Algorithm 13 is dominated by an O (W) operation, which involves re-associating windows with bands based on the newly computed bands[3], apart from

[3]The complexity of this operation is O ($K \star W$), but K is assumed to be a small constant compared to W.

a few O (K) operations in recomputing probability of each band based on the new window members. Hence, the overall complexity of the algorithm could be taken as O (W).

5.2.8 Thresholding

Problem

Given learning traffic statistics for a site, determine the threshold probability.

Methodology

The same strategy of cross validation is employed to arrive at a threshold probability for UDP streams based on the learning traffic. Please refer to section 4.2.7 for more details.

Algorithm

The algorithm uses a Random Number Generator Rng () used for determining which group a particular window statistic shall belong to. Algorithm 14 explains Cross Validation for threshold probability determination.

Complexity

The complexity of Algorithm 14 is dominated by iterative calls of the UDPTrain () function, which has a complexity of O ($WlogW$). Hence, the complexity of the algorithm could be taken as O ($WlogW$).

5.3 Deployment Phase

The deployment phase is very similar in functionality to the training phase in terms of computing statistics. The system takes as input the data structure S containing the NB model probabilities for the given stream, and determines if the network state for the stream is normal or anomalous at any point in time.

5.3.1 Probability Computation

Probability computation involves computing statistics of a window and determining the probability with which the window would have occurred in the training phase. It is described in Algorithm 15.

Algorithm 15: Function *UDPdetermineProbability (W,S)* – Probability of a window W for UDP stream S

Input: Stream with NB probabilities S, packet window statistics W
Output: Probability P of window W

```
1 for Every Window W do
2 |   Determine WAT for window W;
3 |   Determine the range r to which the WAT belongs.
  |   P = S.updbands[r].probability;
4 end
5 return P;
```

Complexity

The number of operations involved in determining probability is a set of simple, light weight operations. This involves a counting operation per packet, and at the end of the window, a look up on the band values to determine which band the window belongs to (worst case $2K$ comparisons), along with a look up of band probability.

Algorithm 11: Function *UDPdetermineOptimalBands (A,TW,K)* – Optimal Band Determination for UDP

Input: WAT array $A = \{a_0 \dots a_{TW}\}$, where TW = number of windows, K = number of bands

Output: Array *result*, containing upper and lower bounds of partitioned bands.

1 $A_s[TW] \leftarrow \{0\}$;
2 $J[N] \leftarrow \{0\}$;
3 $J_s[K-1] \leftarrow \{0\}$;
4 upper,lower $\leftarrow 0$;
5 $I[K-1] \leftarrow \{0\}$, $I_s[K-1] \leftarrow \{0\}$;
6 i,j,k $\leftarrow 0$;
7 resultindex $\leftarrow 0$;
8 Allocate $2 \star K$ values for array *result*;
9 Sort array A in descending order and copy into array A_s;
`/* Determining jumps between consecutive values of` A_s `*/`
10 **for** *i = 0 to TW* **do**
11 $\quad J[i] \leftarrow A_s[i] - A_s[i+1]$;
12 **end**
13 Determine the top $(K-2)$ jumps (largest numbers) from array J. Store them in $J_s[0] \dots J_s[K-3]$;
14 Store indices of $J_s[0] \dots J_s[K-3]$ in J into array I;
15 Sort array I in ascending order and store in I_s;
`/* The first band will be any value greater than max (`I_s`) */`
16 result[resultindex++] $\leftarrow$ inf;
17 result[resultindex++] $\leftarrow A_s[0] + 1$;
18 lower,upper $\leftarrow 0$;
19 **for** *i = 0 to* $(K-1)$ **do**
20 $\quad$ upper $\leftarrow I_s[i]$;
21 $\quad$ result[resultindex++] $\leftarrow A_s[lower]$;
22 $\quad$ result[resultindex++] $\leftarrow A_s[upper]$;
23 $\quad$ lower $\leftarrow$ upper + 1;
24 **end**
25 **if** $K \geq 2$ **then**
26 $\quad$ lower $\leftarrow I_s[K-2] + 1$;
27 **end**
28 upper $\leftarrow$ TW - 1 ;
29 result[resultindex++] $\leftarrow I_s[lower]$;
30 result[resultindex++] $\leftarrow I_s[upper]$;
`/* Last band will be any value less than min (`I_s`) */`
31 result[resultindex++] $\leftarrow I_s[TW-1] - 1$;
32 result[resultindex++] $\leftarrow 0$;
33 **return** *result*;

Algorithm 13: Function *UDPupdateProbabilities (WATarray,S)* – Updation of probabilities for UDP streams.

Input: Array containing WATs observed during learning WATarray, UDP stream S.

Output: UDP stream S with updated probabilities in S.udpbands.

```
   /* Initialisation                                                    */
 1 j, l ← 0;
 2 interval ← 0;
   /* Recount windows for determining windows under bands               */
 3 for j=0 to S.total windows in learning do
 4 |   interval ← WATarray[j];
 5 |   for l=0 to S.K do
   |   |   // Check to which band this value falls into.  Increment
   |   |      it's count.
 6 |   |   if interval > S.udpbands[l].lowerbound and interval ≤
   |   |   S.udpbands[l].upperbound then
 7 |   |   |   S.udpbands[l].probability ← S.udpbands[l].probability + 1.0 ;
 8 |   |   |   break;
 9 |   |   end
10 |   end
11 end
   /* Smooth window counts                                              */
12 for j= 0 to S.K do
13 |   S.udpbands[j].probability ← S.udpbands[j].probability + 1.0;
14 end
   /* Update total number of windows                                    */
15 S.total windows in learning ← S.total windows in learning + S.K;
   /* Compute probabilities for each band                               */
16 for j=0 to S.K do
17 |   S.udpbands[j].probability ← S.udpbands[j].probability / S.total windows in learning;
18 end
19 return S;
```

Algorithm 14: Function UDPdetermineThresholdProbability (N,TF,t) – Thresholding algorithm for UDP

Input: Window size for stream N, Traffic statistics TF, Error proportion t percent

Output: Threshold probability P_t for S

`/* Initialisation */`

Seed Rng () ;

parray[] $\leftarrow 0$;

`/* Grouping Learning traffic */`

Split traffic TF uniformly at random into 10 groups $G_1, G_2 \ldots G_{10}$;

for *i = 1 to 10* **do**

  $G \leftarrow TF - G_i$;

  $B \leftarrow G_i$;

  `/* Learn from` $\frac{9}{10}$`th of traffic */`

  $S_{temp} \leftarrow$ *UDPTrain (SF,G)* ;

  `/* Compute probability of remaining` $\frac{1}{10}$`th of traffic */`

  for *every window W in B* **do**

    $P \leftarrow$ UDPdetermineProbability (W,S_{temp}) ;

    add P to *parray*;

  end

end

`/* Sieving lower probabilities */`

result $\leftarrow (\frac{t}{100} \star parraylen + 1)$th smallest element of parray;

return *result*

CHAPTER 6

Experiments and Results

6.1 Organisation – TCP experiments

The experiments done as part of this work are grouped into the following components:

1. Choice of instance grouping algorithm;
2. Tuning System parameters;
3. Evaluating performance;
4. Comparison with other DDoS protection methods;

A broad block diagram of the organisation of these experiments is shown in Figure 6.1.

All these experiments use an offline implementation of the learning and testing phases of the system, a C language implementation. The executable takes as input traffic file in text format, and produces text output logs, which report the number of misclassified windows (based on which, the number of True and False Positives, and True and False Negatives are computed).

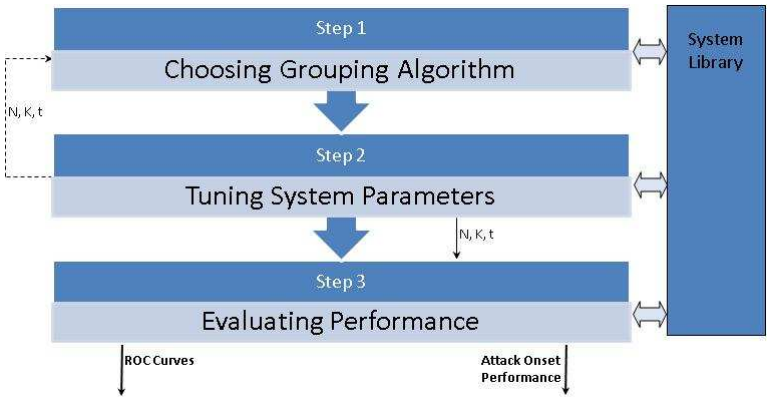


Figure 6.1: Block Diagram showing organisation of TCP Experiments

6.1.1 Choice of Grouping Algorithm

Two candidates exist for grouping model probabilities for reasons mentioned in section 4 - a heuristic jump based clustering technique, and K-means clustering, a well known technique. This experiment describes the student's t-test for independent samples, which was conducted in order to compare the distributions arising out of the two techniques. Values of system parameters are retrospectively fed into system implementation at this point.

6.1.2 Tuning System Parameters

After choosing the grouping algorithm, the system implementation proceeds with the chosen grouping technique. Experiments are done to find the optimal values or relations between system parameters, at which the system reaches good performance levels. At the end of these experiments, optimal values are obtained for each system parameter for different user data.

6.1.3 Evaluating Performance

Experiments are carried out with the parameters chosen in the above experiments, and the false positives and false negatives are reported. ROC curves are plotted for different datasets and are analysed. System performance in the case of windows representing onset of attack is discussed (this is not considered in the previous experiments).

6.1.4 Comparison Experiments

The developed system is compared with a one-class Support Vector Machine (SVM) classifier operating on the same set of features.

6.2 Datasets and Approach

6.2.1 Note on Experiments and Datasets

- It is to be noted that analysis is done per stream, which is a combination of <destination IP, destination port>. In one of these instances (DARPA dataset), there were many such streams available in an interleaved fashion inside a single traffic dataset. The stream with the maximum number of inbound packets was chosen as the target.

- In a practical user site, the deployment phase cannot take the advantage of the entire traffic (attack or normal) being available in entirety from the beginning. But for the limited purpose of experimentation here, this is assumed to be available. In order to differentiate between these two situations, we use the term "testing phase" instead of the term "deployment phase" in this section.

- Data (traffic) is required for both phases of operation – learning phase and the testing phase. For the learning phase, the traffic required is inbound traffic to a prospective target ("normal" traffic). For the testing phase, the traffic required is primarily attack traffic, optionally mixed with normal traffic.

- It is generally observed that while various tools and methods exist for generating attack traffic to a particular target, the difficulty lies in generating or synthesising normal traffic profiles for the target.

- For the limited purpose of testing, the learning and testing modules were implemented in C, and the datasets were printed to standard output by using the *tcpdump* program, and in turn redirected to text files. These text files are the input to the TCP learning and testing modules.

- Since design started with the basic motive of emphasising on practical user sites, even the first experiment started with two datasets – an off-the-shelf, well tagged dataset (ideal traffic representative), and an untagged dataset (user site traffic representative).

It is to be seen that text mode processing is done on the inputs. This makes it easy to "synthesise" any scenario, by copying the textual details of a packet, and appropriately changing interpreted fields, wherever necessary. For example, in

order to synthesise a SYN flooding attack, the textual signature of a SYN packet to the target is copied repetitively (if analysis is done by considering fields other than flags, then the values of the corresponding fields have to be changed in the textual signature). This trick has been extensively employed when considering the need to make a variety of attacks, which otherwise would have been difficult to be generated under lab conditions. Since the analysis for TCP is done only on TCP flags, and importantly, NOT on time of arrival of packets, while synthesising the attack "traffic", it was chosen to conveniently ignore synthesising details of packet arrival times of the attack packets, which may otherwise be required in order to mimic a real attack.

6.2.2 Datasets Used

The following datasets were used for experimentation:

1. **DARPA dataset (DARPA)** : Consists of
   (a) Tagged Normal traffic to a telnet (port 23) server with IP 172.16.112.50, consisting of 70,603 packets.
   (b) Attack traffic to the mentioned telnet server, synthesised as explained in Table 6.1.
2. **SETS dataset (SETS)** : Consists of
   (a) Untagged Normal traffic to the SETS webserver (`www.setsindia.org`) (local IP: 196.168.1.40, port:80) consisting of 4,02,288 packets (collected over ten days at SETS).
   (b) Attack traffic to the mentioned webserver, synthesised as per Table 6.2.
3. **Auckland dataset, Profile I (AUCK-I)** : Consists of
   (a) Untagged Normal traffic to a webserver (IP: 10.0.0.63, Port:80), consisting of 3,23,980 packets.
   (b) Attack traffic to the mentioned webserver, synthesised as per Table 6.2, but with target IP/port changed.
4. **Auckland dataset, Profile II (AUCK -II)** : Consists of

(a) Untagged Normal traffic to a webserver (IP: 10.0.0.63, Port:80), consisting of 6,41,978 packets.

(b) Attack traffic to the mentioned webserver, consisting of 15,000 attack packets.

5. **SETS dataset for UDP**: Consists of

(a) Untagged Normal Traffic to a UDP DNS server: Consists of 64,313 inbound packets to the SETS internal domain name server, collected over a period of 3 days;

(b) Attack Traffic: Consists of 1, 32, 783 attack packets addressed to the SETS internal domain name server, spread over a period of 35 seconds.

Table 6.1: Synthesised Attack Traffic for the DARPA dataset

Packet Numbers	**Nature of Attack**
1-1046	SYN flooding
1047-2277	FIN/ACK flooding
2278-2543	SYN/ACK flooding
2544-2911	PSH/ACK flooding
2912-3629	SYN flooding
3630-4806	RST flooding
4807-6077	FIN/ACK flooding
6078-16115	SYN flooding
16116-22934	ACK flooding
22935-31864	PSH/ACK flooding
31865-42349	RST flooding
42350-97649	FIN/ACK flooding
97650-192549	SYN flooding
192550-545549	ACK flooding
545550-1158549	SYN flooding

6.2.3 Experiment Approach

The basic objective of the experiments was to evaluate the system performance under various conditions, and either make decisions accordingly, or report results in appropriate form. Since the system performance is measured with different functional variants of number of False Positives (FP), True Negatives (TN), True Positives

Table 6.2: Synthesised Attack Traffic for the SETS dataset

Packet Numbers	**Nature of Attack**
1-1566	SYN flooding
1567-2758	PSH/ACK flooding
2759-3859	RST/ACK flooding
3860-19669	FIN/ACK flooding
19670-34655	SYN/ACK flooding
34656-115258	SYN flooding
115259-303265	PSH/ACK flooding
303266-960266	SYN flooding
960267-1146266	FIN/ACK flooding
1146267-1816266	SYN flooding

(TP) and False Negative (FN), every experiment was driven by the requirement to determine one or more of these values.

Terminology

- A particular state of network is flagged as "positive" when it is under attack.
- The state of network is flagged "negative" when it is normal.
- A True Negative (TN) window is a packet window which is part of the normal traffic, and has been classified as normal.
- A False Positive (FP) window is part of the normal traffic, but classified as abnormal.
- A False Negative (FN) window is part of the attack traffic, but classified as normal.
- A True Positive (TP) window is part of the attack traffic, and classified as abnormal.

It is to be noted that the attack traffic described in Tables 6.1 and 6.2 is meant to have windows whose states we are very sure about, while in a practical situation, the system will encounter windows with a mix of attack and normal traffic. The latter situation is discussed in detail in Section 6.5.2.

Methodology

The conventional method of conducting experiments is to have a single dataset, a mix of normal and abnormal traffic, and run it through the testing phase to determine false alarms, after learning with normal traffic. This could be done by choosing at random either the normal traffic distribution or the attack traffic distribution for each instance, in order to provide a mix of traffic. In our case, a mix of normal and abnormal windows has no significance with respect to false alarms, since the window probability will not change with it's position in the sequence. Hence, without loss of precision, the experiment can be split into two phases, where normal traffic and abnormal traffic are fed separately into the testing phase to determine false alarms. The results reported in the document will be the same if a single combined (mixed) dataset was used in the testing phase to determine false alarms.

It should be noted that the above applies only when the normal traffic and abnormal traffic are mixed window by window (considering that the smallest unit of normal and abnormal traffic distributions is a packet window). It may not apply when the normal traffic and abnormal traffic distributions are mixed packet by packet (the smallest unit of the respective distributions are packets). In the latter case, a window could contain both normal and abnormal traffic packets in varying proportions, and the actual tag of windows could be fuzzy. Experiments related to this aspect are discussed in detail in Section 6.5.2.

For reasons mentioned above, the general methodology of conducting experiments is done as follows:

1. **To determine False Positives (and True Negatives)** :
    - Train the classifier with normal traffic.
    - Give normal traffic as input to the testing phase.
2. **To determine False Negatives (and True Positives)** :
    - Train the classifier with normal traffic.
    - Give attack traffic as input to the testing phase.

6.3 Choice of Grouping Algorithm – Student's t-test for Independent Samples

6.3.1 Quick Introduction to Student's t-test

The student's t-test for independent samples [9] is used to compare two small sets of quantitative data when samples are collected independently of one another. It is one of the most commonly used techniques for testing a hypothesis on the basis of a difference between sample means. In other words, the t-test determines a probability that two populations are the same with respect to the variable tested. The probability (based on probability tables originally determined by Gossett) determines the likelihood of accepting or rejecting a hypothesis.

The t-test starts with a null hypothesis, which states that the two distributions are the same. Given the sample set and the sample lengths, the means of the two sample sets, standard deviation, and number of data points in each sample set are computed. At the end of the t-test, the output is a statistic t (not to be confused with the error proportion t), in which

$$t = \frac{|\bar{x_1} - \bar{x_2}|}{\sqrt{A \star B}},$$

where

$$A = \frac{1}{n_1} + \frac{1}{n_2},$$

and

$$B = \frac{[(n_1 - 1) \star s_1^2 + (n_2 - 1) \star s_2^2]}{n_1 + n_2 - 2}$$

$\bar{x_1}, \bar{x_2}$ are the means of the distributions P_1 and P_2, s_1, s_2 are the respective standard deviations, and n_1, n_2 are the sample sizes of the distributions. The term $(n_1 + n_2 - 2)$ refers to the degrees of freedom.

The critical value corresponding to the number of degrees of freedom, and at expected levels of confidence is looked up from Gossett's T-distribution table. If this

value is lesser than the computed t, the null hypothesis can be rejected. Otherwise, it can be concluded that the two distributions present no significant difference.

6.3.2 Tuning to perform t-test

In order to compare the K-means clustering method and the heuristic jump based clustering method, a suitable comparison parameter should be chosen. Since we are concerned about the performance of the system under both these clustering techniques, the t-test input from both clustering methods could be the False Positives/False Negatives/both produced by the clustering techniques.

To accomplish this, we train the system with normal traffic. In order to have multiple false positives, a subset of normal traffic (measured in terms of windows instead of packets, so that we can have absolute control over the expected tags of the windows) chosen at random is taken as input to the testing phase[1]. This is iterated through 20 trials, and the number of false positives tabulated. Similarly, to have false negatives, a subset of attack traffic chosen at random is taken as input to testing phase, and the number of false negatives through each iteration tabulated. These two sets from each clustering technique, {trial index,False Positives in trial} and {trial index, false negatives in trial} are the inputs for comparison.

But it was observed that both clustering techniques were catching all attack windows in all iterations, which means that this information may not be very useful for comparison[2]. Hence, the false positive set from each clustering technique was taken as the input for t-test.

6.3.3 Experiment

In a nutshell, the following procedure is used:

1. Complete system $S_{k-means}$, with K-means used for instance grouping, and system S_{jump}, with jump based clustering used for instance grouping.

2. Generate 20 subsets of normal traffic, with each window of the subset chosen at random from the normal traffic set.

[1] $\frac{1}{5}$ th of normal traffic

[2] This may also imply that the attacks considered were probably too strong to be missed.

Table 6.3: t-test: Number of False Positives for the SETS dataset

FP	**Trial Number** (Total number of testing windows–805)																		
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
C_j	8	3	8	5	9	7	5	8	8	8	5	10	7	3	5	6	6	7	4	5
C_{km}	8	3	7	5	9	5	5	7	8	7	4	10	7	4	6	8	8	8	6	5

Table 6.4: t-test: Number of False Positives for the Auckland dataset-I

FP	**Trial Number** (Total number of testing windows–659)																		
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
C_j	3	7	4	2	9	5	8	6	4	7	5	7	2	4	9	5	7	4	3	4
C_{km}	2	7	3	2	9	6	7	5	7	9	5	7	3	5	9	6	7	4	3	4

3. Training phase: Train with complete normal traffic set.

4. Testing phase: Pass the subset of normal traffic as input. Determine the number of False Positive windows for each clustering method. .

5. Repeat Steps 3 and 4 through 20 iterations, each with a different normal traffic subset for testing.

6.3.4 Results

The experiment mentioned above was conducted for 2 different datasets – the SETS dataset and the Auckland dataset-I. Standard inputs to other system parameters were considered (Refer to Section 6.4) – error percent $t = 1$, number of bands $K = 20$, and window size $N = 100$. The results are for each dataset are tabulated in Tables 6.3 and 6.4. C_j refers to Jump-based clustering and C_{km} refers to K-means clustering.

The following are the results:

- For the SETS dataset,
  - $\bar{x_1} = 6.35, \bar{x_2} = 6.5, n_1 = n_2 = 20, s_1 = 1.9045, s_2 = 1.8027.$
  - $A = 0.1, B = 3.4284.$
  - t-test statistic $t = 0.2558$.
  - Gossett's critical value at $p = 0.05$ is 1.686.
  - Since $t <$ observed critical value, we can conclude that the distributions are not significantly different at a confidence level of $p = 0.05$.

- For the Auckland-I dataset,
  - $\bar{x_1} = 5.25, \bar{x_2} = 5.5, n_1 = n_2 = 20, s_1 = 2.0946, s_2 = 2.2022.$
  - $A = 0.1, B = 4.6185.$
  - t-test statistic $t = 0.3678.$
  - Gossett's critical value at $p = 0.05$ is 1.686.
  - Since $t <$ observed critical value, we can conclude that the distributions are not significantly different at a confidence level of $p = 0.05$.

6.3.5 Complexity Comparison

- The jump based clustering algorithm is dominated by sorting of $N + 1$ numbers, where $N=$ window size. Since for practical applications N is small, the complexity of the jump-based algorithm is $O(N^2)$.
- The K-means algorithm is an iterative algorithm which converges when the difference between cluster means in subsequent iterations converge. The complexity of K-means algorithm is $O(N \star K)$, where $N=$ window size, and $K=$ number of clusters. It is observed that typical values of K should be of the order of $\frac{N}{5}$ (see Section 6.4.1), thereby making the complexity of K-means algorithm $O(N^2)$. The significant constant term involved in the complexity is m, the number of iterations for convergence. However, the number of iterations it takes for convergence is unknown. This shows that for all $m > 5$ (and in general for large m), it is more likely that the K-means algorithm will become more expensive than the jump-based algorithm, due to the constants involved in the respective complexity arguments.

  The above discussion is under the assumption that the first selected mean combination is ideal, but however it is known that K-means get struck on local optima. Hence in a practical situation, the best results out of many iterations should be taken. More accurately, the significant constant term in the complexity argument should have been $m \star I$, where I refers to the number of iterations the K-means algorithm has to be performed.
- In general, although both the jump-based clustering algorithm and K-means algorithm have similar complexities, the former edges out the latter due to the

smaller constants involved (in the case of reasonably large m and I for K-means algorithm) in practical situations.

6.3.6 Conclusion

Although the t-test shows no significant difference between the two distributions of false positives computed with the two techniques – Jump based clustering and K-means clustering, *further experiments will be reported with the system based on jump-based clustering*, due to reasons mentioned above.

6.4 Tuning System Parameters

After the choice of the grouping algorithm has been made, the next task is to tune system parameters in order to maximise performance. It is seen that some of these parameters could be generalised across user profiles, while the others revolve around the user traffic profile.

The following are the important system parameters in consideration:

1. Window Size (N);
2. Number of bands (K); and
3. Error Percent (t).

The optimal values for each of the above parameters were empirically determined, based on the following factors:

- Maximum Performance – lesser number of false alarms.
- Light-weight detection and quicker response – decrease response time.

6.4.1 Experiments on N and K

Approach

The objective of the experiment is to determine the ideal values of $\{N, K, t\}$ at which the performance will be maximal. It is obvious that N and K are related to each other – K refers to the amount of compression allowed on the total number of events in the event space, which in turn, is determined by the value of N, for good performance. A higher value of N will result in more probabilities, which may (or may not) call for an increase/decrease in the value of K in order to accommodate performance requirements. But it is observed that although N and K are related to each other, the value of the two tuple $\{N, K\}$ need not be tied to a particular user traffic profile, and can be generalised across user sites. However, N and K should be chosen in a way that the relation (if it exists) between N and K is still preserved across all user sites. On the other hand, t by definition is specific to the user traffic considered. Hence, t may have to be varied from site to site, and involves trial-and-error.

For reasons mentioned in the previous section, the performance is mentioned in terms of False Negatives (FN) or True Positives (TP). The parameter considered for measurement is the True Positive Rate (TPR), the ratio of True Positive windows to the total number of Attack windows.

The approach taken to finding the right $\{N, K, t\}$ is as follows:

- The value of t is kept at a constant, and the system performance for different $\{N, K\}$ combinations is observed to derive the relation between N and K.
- The value of t is chosen in a way that it exceeds the maximum value of t for all datasets available for experimentation. This is based on the fact that the system performance saturates at a maximum, beyond a value of t for any dataset. It is seen that $t = 1\%$ is a suitable value for the experiment. Precise values of t will be determined for every dataset separately (described in the next section).
- The varying values of N and K are taken from the following discrete values: $N^{'} = \{50, 100, 150, 200\}$ and $K^{'} = \{5, 10, 15, 20, 30, 40\}$.
- A performance matrix, which describes the performance (TPR) of the system at each possible $\{N, K\}$ combination from $N^{'}$ and $K^{'}$ is arrived at. The resultant

Table 6.5: Performance Matrix for the SETS dataset

K	**N**			
	50	**100**	**150**	**200**
5	11.94%	11.97%	11.99%	11.91%
10	100%	22.37%	11.99%	11.91%
15	100%	100%	11.99%	22.39%
20	100%	100%	100%	22.39%
30	100%	100%	100%	22.39%
40	100%	100%	100%	100%

matrix is a 6*4 matrix.

- The matrix is analysed to determine $\{N, K\}$ values for which performance is maximum. Bounds are determined on the values for which the system performance is guaranteed to get maximised. This will also bring to light the relation between N and K, if it is quantifiable.

- Based on the above observation, choose good values of N, K, and determine the exact value of t for every dataset.

Experiment

The experiment is the same as the general experiment approach – for a chosen value of $\{N, K\}$, train with normal traffic and pass the training traffic into the testing phase in order to determine the number of False Positives. Train with normal traffic and pass the attack traffic into the testing phase, in order to determine the number of False Negatives. Determine TPR for each $\{N, K\}$ and populate a cell of the performance matrix.

Results and Interpretations

The experiment was conducted (with $t = 1\%$) for the above said values of K for three datasets – DARPA dataset, SETS dataset and the AUCK-I dataset. The performance matrices for the three datasets are indicated in Tables 6.5, 6.6 and 6.7.

The following are the important observations:

- The performance matrix can be analysed in two ways – Column wise analysis

Table 6.6: Performance Matrix for the AUCK-I dataset

K	**N**			
	50	**100**	**150**	**200**
5	0.83%	0.84%	0.9%	0.91%
10	0.84%	0.85%	0.9%	0.85%
15	89.58%	89.57%	0.86%	0.86%
20	89.58%	99.99%	0.87%	0.9%
30	99.99%	100%	100%	89.6%
40	99.99%	100%	100%	100%

Table 6.7: Performance Matrix for the DARPA dataset

K	**N**			
	50	**100**	**150**	**200**
5	68.14%	68.16%	68.16%	68.18%
10	100%	100%	68.17%	100%
15	100%	100%	100%	100%
20	100%	100%	100%	100%
30	100%	100%	100%	100%
40	100%	100%	100%	100%

is done when N is fixed and a suitable K has to be arrived at, and row wise analysis to be done when the right N is chosen for fixed K. The former is more useful.

- The matrices show an important fact discussed above – for a given N, performance saturates after a particular value of K is reached, and addition of a few more bands does not improve the performance any more.

- The well tagged DARPA dataset appears to have ideal properties – for even higher values of N, performance starts converging from small values of K. In essence, it means that the dataset contains very few event occurrences for all N, which could be captured with fewer bands.

- From the results, it can be seen that when K is upper bound by $\frac{N}{5}$, the system performance peaks for almost all values of N and K in all three datasets.

Conclusion

Hence, it is hypothesised that K should be atleast $\frac{N}{5}$ for the system to achieve good performance levels. The absence of constant values for K for any N is expected to create a few space overheads in hardware implementation (of detection), since N is expected to vary across multiple user sites, depending on their traffic profiles. This can be worked around by implementing detection assuming a reasonably large value of N, and choosing to keep $\frac{N}{5}$ bands for model probabilities. This space can be dynamically handled for all values of N lesser than the chosen value. This may amount to wastage of reasonably small space, which can be easily handled by modern hardware.

To illustrate, let the implementation be done with the assumption that the maximum value of N could be 2000. In this case, a straight forward implementation of model probabilities will take space of $(N + 1) * 6$ floating points = (2001 * 6 * 4) bytes = 480.2KB. Wastage of space of the magnitude of these are easy to handle by modern hardware.

6.4.2 Experiments on Error Percent t

The objective of this experiment is to determine precise values of t for different datasets. The previous experiment hypothesised the relation between N and K, and experiments on t preserve this relation. For the purpose of these experiments, N is assumed to be 100, and K, according to our hypothesis is chosen to be 20. With these values fixed, experiment is conducted by varying the value of t until a saturation point in performance, in terms of True Positives is reached. Since it has already been explained that increasing values of t will increase false positives, we are concerned more about the attack traffic and how efficient the system is with respect to classifying attack windows as attack, with a given t.

Experiment

The following procedure was used to conduct the experiment:

1. Training phase: Train with normal traffic.

2. Testing phase: Pass as input the attack traffic, and estimate the True Positives for every t.

3. Repeat the experiment for different values of t, and find out the value of t at which the number of True Positives converges to a maximum. This value is the actual value for the error percent t.

Results and Interpretations

The results of the experiment are graphically described in Figure 6.2.

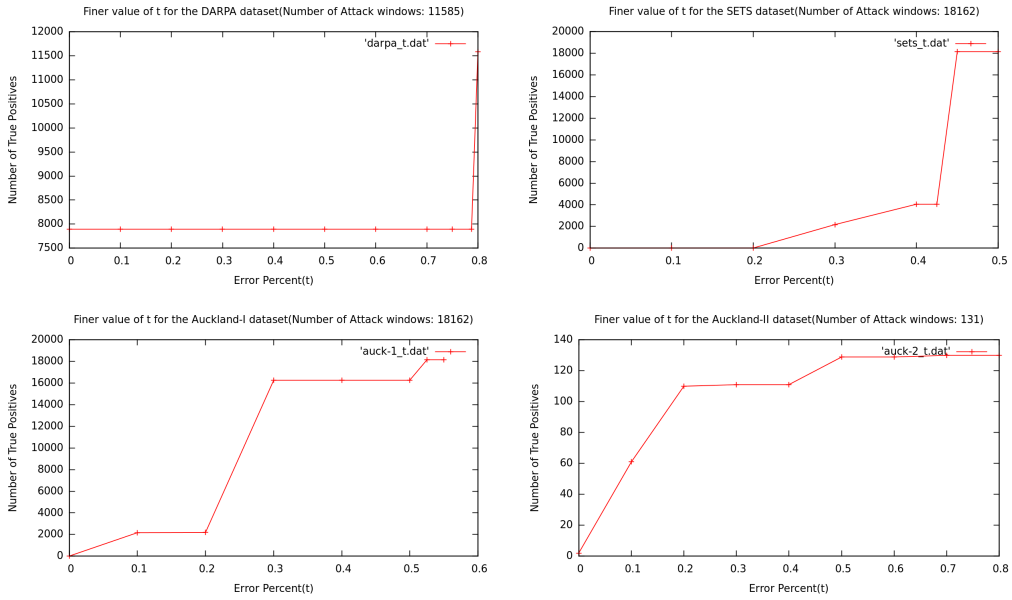


Figure 6.2: Experiments to determining finer values of t

6.4.3 Conclusion on System Input Parameters

- The value of t differs from site to site. This could be determined in an experimental fashion during the learning phase.

- The number of bands K determines the level of accuracy of modelling, and should be carefully chosen during system design, to provide a balance between implementation constraint and performance. N and K appear to be related to each other.

6.5 Evaluating Performance

While there could be multiple ways of evaluating performance of the system, we have taken two such representations of evaluating the performance of the system – A straightforward tabular representation on false alarms resulting out of different datasets, and an ROC curve representation. Further sections describe experiments done towards the above objective.

6.5.1 ROC Curves

Quick Introduction to ROC Curves

Receiver Operating Characteristic (ROC) curves are graphical plots of False Positive Rate (FPR) versus True Positive Rate (TPR) for a binary classifier, when the threshold for classification is varied. ROC analysis generally provides for comparison of different models and possible elimination of the sub-optimal ones. ROC curves are known to depict trade-off between True Positives and False Positives. Each point in the ROC curve represents the value of FPR and TPR for a particular threshold. These thresholds are varied from 0 to 1 (thresholds are in terms of probabilities in this case) to obtain different such points. An ROC curve also assists in estimating a reasonable value for threshold – Point for which FPR is minimum and TPR is maximum, towards the top left corner of the graph. An example of an ROC curve is given in Figure 6.3.

While different interpretations are done in the machine learning research community based on the Area Under Curve (AUC) of the ROC curve, certain facts about ROC curves would let us take certain straight-forward viewpoints on the used classifiers:

- A completely random guess gives a point along with diagonal line from the left bottom to the top right corners.
- The best possible prediction method would yield a point in the upper left corner or coordinate (0,1) of the ROC space, representing no false negatives and no false positives.
- The diagonal divides the ROC space. It is generally observed that points above

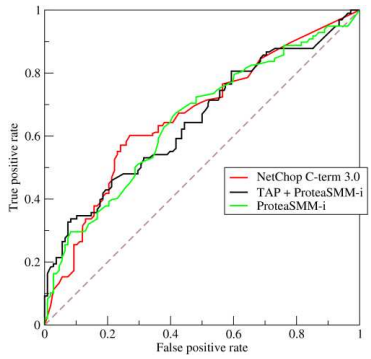


Figure 6.3: A Sample ROC curve (source: `www.wikipedia.org`)

the diagonal represent good classification results, and points below the line represent poor results. Hence the curves lying above the diagonal represent good classification results, and the ones below the diagonal represent poor results.

These intuitive principles will be applied to judge our system performance.

Methodology

The objective of the experiment is primarily to determine FPRs and TPRs at different thresholds. This includes keeping track of the actual tags of windows input to the testing phase, and their classification tags at each threshold. Since we have at our disposal different datasets representing normal and abnormal traffic, tagging of windows as normal and abnormal is easy. The probabilities of both normal traffic and abnormal traffic windows (after training with normal traffic) are recorded during the testing phase, and the actual tags are associated with them in accordance with where the windows come from – normal or abnormal traffic.

The impact of the error percent (t) on window tagging has to be highlighted here. During the tagging of windows, all windows in the normal traffic dataset would generally be termed normal. But with t in place, t percent of the total number of windows is assumed (likely) to be abnormal. Hence, the actual tags of the lowest t percent of normal traffic windows are marked abnormal.

At the end of this exercise, what we have with us is a set of window probabilities (of both normal and attack windows) and the associated actual tags. The job left is to classify the windows corresponding to these probabilities as normal or abnormal

based on varying thresholds. These will denote the classification tags of the windows for the different chosen thresholds. Based on the false classification, the FPR and TPR are computed for each thresholds, and are used for generating points on the ROC curve.

In order to obtain P points on the ROC curve, P corresponding thresholds have to be determined along the window probability space. In order to do this, the window probabilities are first sorted (with their corresponding actual tags preserved). This would let us easily handle the count of true and false alarms based on a chosen threshold. If P points are desired in the ROC curve, the sorted probability set is divided into $(P + 2)$ partitions[3], and the last probability of each partition (the highest probability of the partition, since the probability set is sorted) is taken as the threshold. For each value of threshold, every window (probability) is classified as abnormal (if the probability is lesser than the threshold probability) or normal (if the probability is greater than the taken threshold). This would mark the "classified tag" of the window.

At the end of the above operation, each window has an actual tag and a classified tag. The number of classifications and misclassification are then computed and recorded based on their actual and classification tags. These represent the performance parameters, namely, True Positives, True Negatives, False Positives, and False Negatives. The FPR and TPR are estimated for each set of values.

Hence, we have P number of FPR and TPRs, which are plotted on the ROC curve.

Experiment

The following procedure is followed for the above experiment:

1. Training phase: Train with normal traffic dataset.
2. Testing phase:
   (a) Pass windows from normal traffic dataset into testing phase, and record the probabilities. These constitute (in part) windows with actual tags marked as normal.

[3]We are not taking the values of the highest probabilities as thresholds, and hence ignoring thresholds from last two blocks.

(b) Pass windows from attack traffic dataset into testing phase, and record the probabilities. These constitute windows with actual tags marked as "abnormal/attack".

3. Tagging of windows with actual tags:

   - Tagging of attack traffic: Mark the actual tag of the attack traffic windows as "abnormal".
   - Tagging of normal traffic: Sort normal traffic window probabilities, and mark the actual tags of lowest t percent of windows as "abnormal". Mark the actual tags of rest of the windows as "normal".

4. Data for ROC:

   - Determine the number of points to be plotted on the ROC curve. Let this be P.
   - Sort the combined set of window probabilities, and divide the sorted set into $P + 2$ partitions.
   - Take the last element of each partition as the threshold.
   - For each threshold, mark the classified tag for each window as either normal (if probability greater than threshold) or abnormal (if probability lesser than threshold).
   - For each threshold, computed and record the following values:
     - True Negative (TN) – No of windows whose actual tag is normal, and classified tag is normal;
     - False Negative (FN) – No. of attack whose actual tag is abnormal, and classified tag is normal;
     - True Positive (TP) – No. of windows whose actual tag is abnormal, and classified tag is abnormal;
     - False Positive (FP) – No. of windows whose actual tag is normal, and classified tag is abnormal
     - False Positive Rate (FPR) = $\frac{FP}{FP+TN}$
     - True Positive Rate (TPR) = $\frac{TP}{TP+FN}$
   - Plot the P values of FPRs and TPRs as points on the ROC curve.

Results

The ROC curves were drawn for two datasets – Auckland-I dataset, and the SETS dataset. These are depicted in Figure 6.4. In order to understand the effect of the error percent t (which directly affects the thresholding in our case, and system performance), tagging of normal windows was attempted at 4 different values of t - 0, 0.1,0.2 and 1 percent for window size $N = 100$ and number of bands $K = 20$, and the behaviour of the ROC curves observed.

It is seen that when t moves towards the optimal value for every dataset, the curve moves to the left, with lesser FPR, and thereby better performance. A zoomed version of the ROC curves is shown.

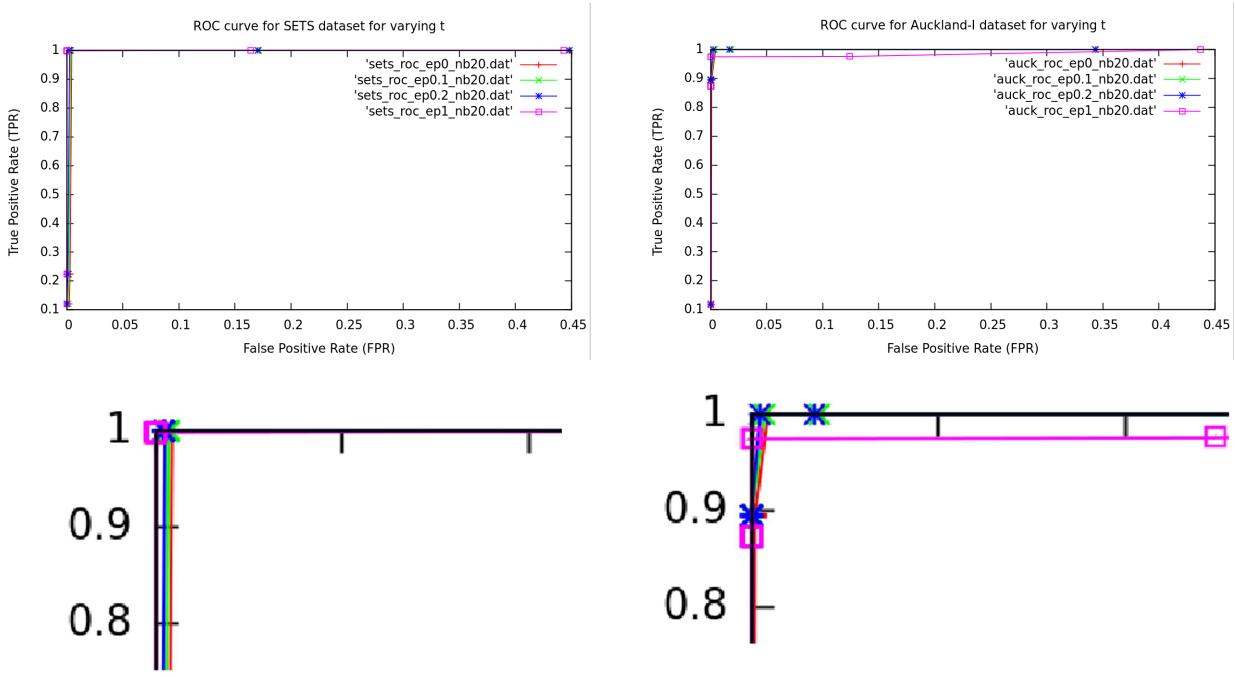


Figure 6.4: ROC Curves – SETS and Auckland-I datasets

6.5.2 Performance against Onset of Attack

Methodology

While the training traffic datasets considered here are untagged datasets, which characterise a realistic situation, the datasets used for the experiments in testing are

mostly well tagged – either normal traffic datasets coupled with error percent t, or attack traffic datasets whose windows are clearly tagged. But in real user sites, such a clear understanding of the window tags during testing is not available. In many cases, the packets in a window may be a mix of both normal traffic and attack traffic, which introduces a fuzziness in classification of the packet window.

While the system performance parameters including false alarms generally characterise how well the classification is done, we are concerned about a specific such scenario where the system has to respond to an onset of attack. While an attack is building, it is possible that we see windows which contain variable amount of attack traffic, mixed with normal traffic.

The objective of this experiment is to look at how well the system can respond to an onset of attack. SYN flooding attack is taken as the example. During the onset of a SYN flooding attack, it can be seen that in (mostly) subsequent windows, the proportion of SYN packets gradually rises until in a couple of windows (depending on the velocity of the attack), we see windows with full of SYN packets. We aim to examine the classification of these intermediate windows.

In order to synthesise such intermediate windows, different representatives of normal windows (with varying proportion of each of the 6 observable types) were chosen from the normal traffic. It should be noted that these normal windows should be equally spaced in terms of probabilities so that we get a good representation set of normal window.

The intermediate windows (representing the onset of SYN flooding attack) are synthesised by gradually increasing the proportion of SYN packets in the window, while lessening the number of packets of other types, based on heuristic interpolation. For every input normal window, 10 such intermediate windows were computed by increasing the proportion of SYN packets heuristically (window with 10 percent increase in SYN packets, window with 20 percent increase in SYN packets and so on until window full of SYN packets). Hence, each interpolation contributes to 11 classes of windows – from (0 % attack, 100 % normal) to (100 % attack, 0 % normal). An example of such interpolation is shown in Table 6.8, for a single normal window chosen from the Auckland-I dataset.

If we started with W normal windows, then at the end of the exercise, we have $10\star W$ additional windows, with different proportions of attack traffic in them. Windows with the same proportion of attack traffic (for example, available W windows with

%		RST	SYN	ACK	F/A	P/A	Oth
N	**A**						
100	0	**2**	**3**	**71**	**1**	**23**	**0**
90	10	2	13	63	1	21	0
80	20	2	23	57	1	17	0
70	30	1	32	53	1	13	0
60	40	1	40	50	0	9	0
50	50	1	51	42	0	6	0
40	60	0	60	39	0	1	0
30	70	0	72	27	0	1	0
20	80	0	81	18	0	1	0
10	90	0	90	10	0	0	0
0	100	**0**	**100**	**0**	**0**	**0**	**0**

Table 6.8: Crafting windows representing onset of attack

10 percent of SYN flooding attack traffic) represent different variants of the same trait – the same proportion of attack traffic. Such windows are grouped into a single dataset, and these W windows are subject to the testing phase, and the number of windows classified as attack computed. These are graphically represented.

Experiment

In a nutshell, the following experiment is conducted:

1. Train the system with normal traffic.
2. Pass the normal traffic as input to testing. Based on the probabilities of the windows, choose W equally spaced windows (in terms of probabilities) from the set of windows classified as normal. These W windows are the representations of normal traffic.
3. For each of the W windows, construct 10 intermediate windows, which have an increasing proportion of SYN flooding traffic, namely windows with 10 percent increase in attack traffic, 20 percent increase in attack traffic and so on. The result is a set of $10 \star W$ windows.
4. Among the $10 \star W$ windows, group windows with the same proportion of increase in attack traffic, to form 10 groups $G_{10}, G_{20} \ldots G_{100}$, each with W windows.

5. Train the system with normal traffic, and pass each of these groups into testing phase. Compute the number of windows classified as abnormal.

Results and Interpretations

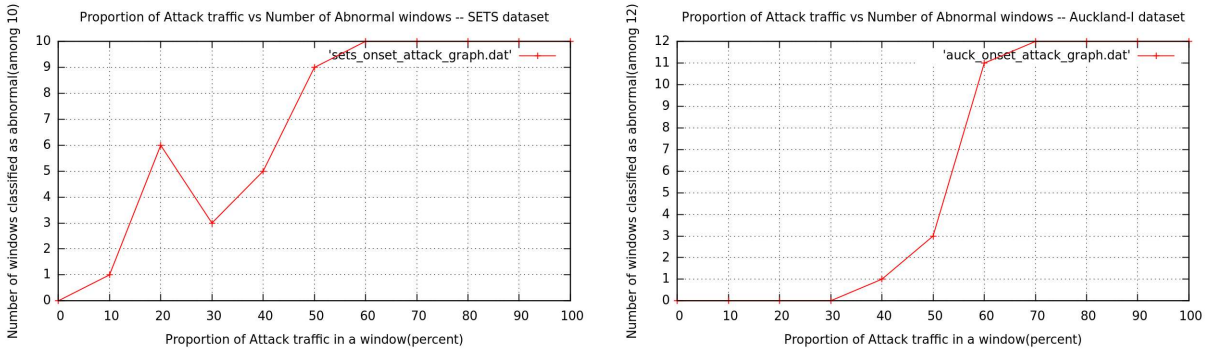


Figure 6.5: Performance against onset of attack – SETS and Auckland-I datasets

- Experiments were performed on two datasets – Auckland-I dataset, and the SETS dataset. Experiments were run with window size $N = 100$, number of bands $K = 20$, and error percent $t = 1$ for both datasets.
- The results are graphically described in Figure 6.5. The graph describes the number of these intermediate windows classified as attack versus the proportion of increase in attack traffic.
- The portion between the rise of the curve from the x-axis, to reaching the peak of the y-axis represents the fuzzy region. The rise happens at different proportions of attack traffic for the two datasets. For example, in the Auckland-I dataset, for a 30 percent of increase in attack traffic from any of the chosen normal windows, none of the resultant windows were classified as attack. This is probably due to the fact that windows with such proportions of SYN packets were also frequently observed in the normal traffic itself.
- When the proportion of attack traffic is more than 50 percent, the system seems to have detected most variants of such windows as abnormal.
- As expected, the number of windows detected as abnormal follows a rising trend – with increase in proportion of attack traffic, more number of attack windows get detected.

- The graph of the SETS dataset presents an interesting observation – for a 30 percent increase in proportion of attack traffic, there is a sudden dip in number of windows detected as abnormal, after which it follows the normal trend. This may probably be because of a strange fact in the SETS dataset – The normal traffic of SETS consists of more number of windows with SYN=30 compared to number of windows with SYN=20!

- Another unusual observation with the SETS dataset is the fact the one window with 10 percent of SYN flooding traffic was classified as abnormal, while the rest of the windows seemed normal to the system. This in one sense, is indicative of the diversity in the SETS normal traffic.

6.6 Comparison with existing Methods

6.6.1 Challenges

In an attempt to compare the complexity and performance of the developed technique with other techniques in literature, it was observed that different works have been done in different test environments, including datasets used, number of features, performance parameters reported, constraints on efficiency reports, etc. Here are a few excerpts of the variety of environments in which experiments pertaining to different works are reported:

- [17] does NB modelling on the KDD99 dataset, utilising 41 attributes and reports for DoS attacks a detection rate of 99.75% and a false positive rate of 0.04%.

- [66] does HMM based training on the DARPA99 dataset, testing on a mix of DARPA99 dataset traffic and DDoS traffic injected at (supposedly) arbitrary times, reports a detection rate of 100% with zero false alarms, subject to the condition that the length of the input observation sequence is 120, while their Discrete Reinforcement Learning algorithm reports 79.2% detection rate.

- Authors of [72] assume a distributed network for detection, perform customised experiments based on CUSUM and report a Detection rate of 91.5% with a false positive rate of 8.8%

- The authors of [19] use portions of the NZIX dataset, BELL labs dataset along with proprietary datasets for their experiments.

Also, it is seen that very few previous works comment on the latency involved in detection. Owing to the above, it is understood that comparison efforts may be inaccurate.

For the above reasons, we were left with two strategies for comparison; a) Identify and implement methods in literature which are closest to ours in terms of constraints and inputs (for example, methods taking flags as features) in our settings, in the assumption that the methods will be comparable; or b) Assume the same constraints and inputs on standard classifier, implement them and compare with our method.

In the course of examination, it was discovered that strategy (a) was also a difficult proposition, owing to the fact that only very few works in literature are actually described in reasonably good detail, to the extent of implementing them in a straightforward fashion. It was then decided that our method be compared with an implementation of a suitable 1-class classifier under the same settings and inputs.

The reason behind choosing 1-class classifiers was that our method is also viewed as a 1-class classifier in some sense, based on a) The single class of input (normal traffic) that it is trained on ; and b) The decision that the system makes – normal/not normal. We chose to use the *libsvm* [13] package to take standard implementation of 1-class SVM for comparison. For a detailed account of SVMs and their applications in classification, the reader is suggested to through literature on SVMs like [8].

6.6.2 Comparison with 1-class SVM

Our method was compared with 1-class SVM implemented by the libsvm package. After experimentation, it was seen that RBF kernel was giving the best performance among different kernel inputs to 1-class SVM. The SVM was trained with the same input as that of our method – 6 counters $C_1 - C_6$, corresponding to the count of flags set in packets per window, with different values of RBF kernel input ν. The data was scaled before being used.

The experiment was done similar to the way our method was experimented – as described in section 6.2.3. The accuracy of our method was compared with the accuracy produced by the trained 1-class SVM. With regards to comparison, the

Dataset	Accuracy of 1-class SVM			Accuracy of our algorithm	
	$\nu = 0.1$	$\nu = 0.01$	$\nu = 0.001$	Accuracy	t
SETS dataset	96.93%	99.91%	99.98%	99.91%	0.45
Auck-I dataset	98.44%	99.65%	99.94%	99.93%	0.525
DARPA dataset	99.31%	99.93%	95.21%	100%	0.8

Table 6.9: Performance comparison of 1-class SVM with our algorithm

motive was to compare the performance of 1-class SVMs with our algorithm, tuned with best parameter values for each dataset. The values of N and K are maintained at 100 and 20 respectively (in line with the hypothesis that K should be equal to $\frac{N}{5}$). The remaining (best) value is that of the error percent t, which varies for each dataset, and whose value was experimentally determined more accurately as described in section 6.4.2. The corresponding values of t as determined in section 6.4.2 was chosen to be used for every dataset.

Table 6.9 shows the comparison between accuracy of 1-class SVM with different ν inputs and that of our algorithm, executed with the precise value of t for each dataset.

From the performances of our algorithm reported in the above table, it should be noted that certain proportion of false positives (t percent) in our algorithm is self-induced, since we consider that t percent of training traffic may be abnormal. Hence, this should be readjusted with correct classifications (True Positives), thereby inducing changes in other classifications also. This in general, will improve the accuracy of classification.

For instance, for the DARPA dataset, t=0.8% out of a total number of 706 training windows. The classification was as follows: TP = 11585, FP = 0, FN = 0, TN = 706. So the number of attack windows P = TP + FN = 11585, and the number of normal windows N = FP + TN = 706.

t = 0.8% of 706 = 6 windows, which should be added to total number of positive (attack) windows. Hence, P = P + 6 = 11591, and N = 706-6 = 700.

Since t = 0.8%, we are expecting FP to be atleast 6. But the actual observed FP is 0, which means 6 windows which should have been classified as abnormal, have been missed. This should be adjusted against False Negatives (attack windows that

Dataset	Accuracy of 1-class SVM			Accuracy of our algorithm
	$\nu = 0.1$	$\nu = 0.01$	$\nu = 0.001$	
SETS dataset	96.93%	99.91%	99.98%	100%
Auck-I dataset	98.44%	99.65%	99.94%	99.98%
DARPA dataset	99.31%	99.93%	95.21%	99.95%

Table 6.10: Performance comparison of 1-class SVM with our algorithm (readjusted with t)

were missed). Hence, FN = FN + 6 = 6.

Hence, the actual classification values after readjustment based on considered t are given by TP = 11585, FP = 0, FN = 6, TN = 700, P = 11591, and N = 700.

Hence accuracy = $\frac{TP+TN}{P+N} = 99.95\%$

The resultant readjusted results for our algorithm produce accuracy as shown in Table 6.10.

6.6.3 Conclusion

From the results of comparison, it can be seen that our algorithm performs as good as a 1-class SVM trained with the same data. But our system outscores a 1-class SVM with respect to practical implementation, due to the following reasons:

- The detection operation is easily less computation intensive in our algorithm, when compared to that of 1-class SVMs. This is because the intermediate probabilities are stored in a lookup table in our algorithm, while they have to be computed on-the-fly in the case of 1-class SVMs.

  In order to support this argument, a C implementation of our algorithm was compared with the detection phase of the *libsvm* package (configured to run 1-class SVM detection), by executing both algorithms in a PC hosting Intel core i5 processor and 4GB RAM, under similar conditions. The Average Running Times (ART) of 10, 000 iterations of detection of a dataset containing 18, 162 windows was analysed (see Table 6.11). It can be seen that **our method is nearly 25 times**[4] **faster than the 1-class SVM of libsvm**;

[4]Due to measurement errors, the synopsis of this work wrongly reported the speed gain as 70.

ART of 1-class SVM detection	ART of our detection algorithm
157.33 seconds	6.5 seconds

Table 6.11: Average Running Time (ART) comparison of our method and 1-class SVM

- If we were to work around the above problem by pre-computing and storing window probabilities in the case of 1-class SVMs, the space consumed (in terms of number of floating points to store intermediate probabilities) in the process grows rapidly with the window size N (all combinations of all 6 observable types, making N^6 combinations, limited by the fact that the sum of observable counts is N). On the contrary, our algorithm consumes space linear with N (to be more precise, $6 \star (N + 1)$ floating points for a straightforward implementation).

6.7 Experiments for UDP

6.7.1 Introduction

The experiments done as part of testing the design for UDP protocol were very less compared to those done for TCP, primarily because of the lack of sufficient training sets in the open domain for the UDP protocol. The only traffic set available for UDP testing was that of the internal DNS server at SETS. With the assistance of a few traffic generators, the attack traffic pertaining to the feature for UDP (WAT) was generated at SETS and used.

The experiment method was same as that done for TCP – Normal traffic for training and Normal traffic for testing to find False Positives, and normal traffic for training and attack traffic for testing to determine False Negatives. Similar to TCP, the error percent t was appropriately included to be part of the system.

6.7.2 Effect of System Parameters

The system parameters for UDP design are the values of the window size (N), number of bands for grouping WATs (K), and the error percent (t). The impact of the above parameters is discussed below:

- **Window Size (N).** The value of window size does not affect accuracy of modelling, or the system performance, because the probability space is modelled on WATs, which are completely independent of the window size (although a drastic increase in window size tends to increase the WATs). Hence the system performance is not affected for typical values of N. Common values could be chosen for N based on experience.

- **Number of bands (K).** Although innumerable number of WATs can be modelled and the considered design still groups such WATs based on K into constant number of bands, it is believed that system critical values of WATs (the ones with reasonably small WATs) can all be captured by K constant groups in most situations for most datasets. Hence, the value of K is more constant for UDP than TCP.

- **Error percent (t).** The value of error percent (t) appears to be comparatively significant with respect to UDP design. It should be noted that unlike the case of TCP, the abnormal windows in normal traffic have a stronger and more pronounced signature with respect to their WATs.

6.7.3 Experiment

The experiment done with respect to UDP is similar to the experiments done with TCP. The training phase involved training system with the training traffic, with a particular value of t, and the value of N set to 100 and K set to 10. During the testing phase, the training traffic or the attack traffic were passed as inputs to have an idea of false positives and false negatives.

6.7.4 Results

The results for UDP testing with different values of t are tabulated in Table 6.12.

- It is seen that even the number of false positives does not increase linearly with t. Since the probability of a window is the probability of the band under which the window falls, the total number of unique probabilities recorded at the end of testing phase will be not more than K. The normal window probabilities hence have clear demarcations between them, and there is very limited variability

Inputs		FPR	TPR
t	K		
0	5	0	0.998
1	5	0.009	0.998
1	8	0.009	0.999
1	9	0.003	1
1	10	0.003	1
2	10	0.017	1
2	20	0.023	1
3	10	0.032	1
3	20	0.032	1
5	10	0.032	1
5	20	0.032	1
10	1	0.032	1
10	20	0.032	1

Table 6.12: Results for UDP Design

in probability values between different windows unlike in the case of TCP. Therefore, for a particular value of t (for a particular threshold probability, that is), an entire series of windows (all of them belonging to a same band) will be classified as normal/abnormal based on whether the probability of the band is less than the threshold or greater. The proportion of false alarms will change only when t is increased to a value which will make it greater than the probability of another band, in which case, all the windows belonging to the band will be classified as abnormal.

- Due to the above fact, it is seen that the value of t has a lesser binding with the likely proportion of abnormal windows in learning traffic.
- The results are good in the case of $t = 0$, mostly because of the presence of no abnormal events.
- Just as it was in the case of TCP, the values of t and K (unlike in the case of TCP, the value of K saturates faster) saturate at a point, beyond which the results are not significantly different.

6.8 Implementation and System Design Details

The ideas developed in this work were formally put together into the design of a practical system, and was implemented as an embedded system. A brief on the system design details is given below. It describes a general system framework on which TCP and UDP are implemented.

6.8.1 System Description

The implemented Denial of Service (DoS) Mitigation System is comprised of five modules:

1. Control Unit (CU);
2. Traffic Capture Module (TCM);
3. Learning Module (LM);
4. Detection Module (DM); and
5. Mitigation Module (MM).

Control Unit (CU)

The Control Unit is a software module, which centrally manages communication between the rest of the modules. CU is resident on the computer hosting the Learning Module (LM), and interacts with the user (through a Graphical User Interface) to obtain inputs for the above modules. The CU also is responsible for information exchange and dissemination between the modules, and establishes a communication network between each of these modules for this purpose. The CU runs different protocols with the various modules for the task. This is a software module.

Traffic Capture Module (TCM)

The Traffic Capture Module (TCM) is a hardware module, which captures traffic required for learning phase, and computes traffic statistics as required for the Learning Module. The TCM communicates with the Control Unit (CU) to obtain it's inputs,

namely stream information. Based on triggers, the values of stream configuration are loaded (by CU), traffic captured and statistics supplied back to the Control Unit.

Learning Module (LM)

The Learning Module (LM) is a software module, which resides in the host computer which is connected to the rest of the hardware modules. The LM acccepts configuration details of the server targets, and learning traffic statistics (as captured by the TCM) from the CU, and arrives at model probabilities which will later be used by the Detection Module (DM) for classification of normal traffic/anomalous traffic. It is into this module that the ideas developed as a result of this research work are implemented.

Detection Module (DM)

The Detection Module (DM) is a hardware module, which on input the set of model probabilities of normal traffic per stream (output by LM, and supplied by CU), determines the state of the network (normal or abnormal) in real-time. This module is the operations module for the system. The DM talks with the CU to obtain it's inputs and interrupts the CU in case an attack flag is asserted. This will cause further action by the CU on the Mitigation Module (MM).

Mitigation Module (MM)

The Mitigation Module is a hardware module, which allows input traffic unaltered when attack flag is not set in any stream, and when attack flag is set in one or more streams, it filters traffic for the stream (s) on the basis of a white list of source IPs corresponding to the stream.

Module Operation Sequence

Every module communicates with the CU for Input and output. The sequence of information exchange and functionality for the prototype is given in Figure 6.6.

The circled numbers (in sequence) represent the protocols that the respective modules run with the CU. A sample protocol which is run between the TCM and

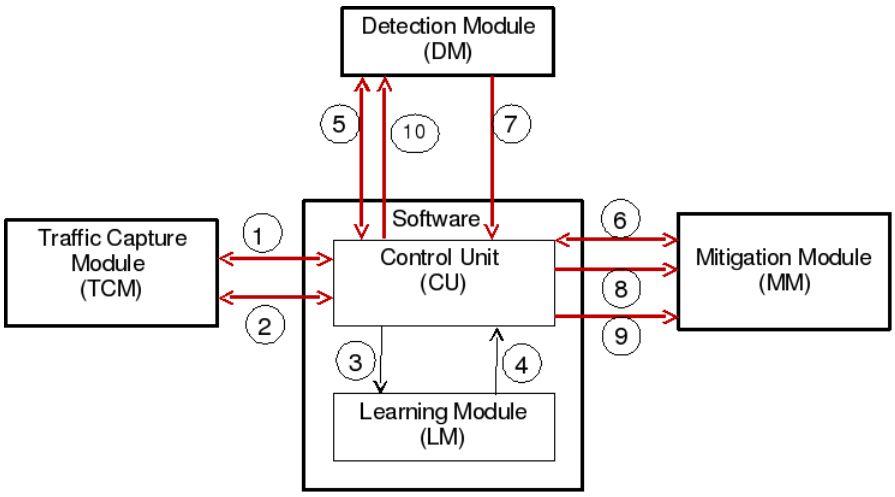


Figure 6.6: Overall System Diagram

CU is shown in Figure 6.7.

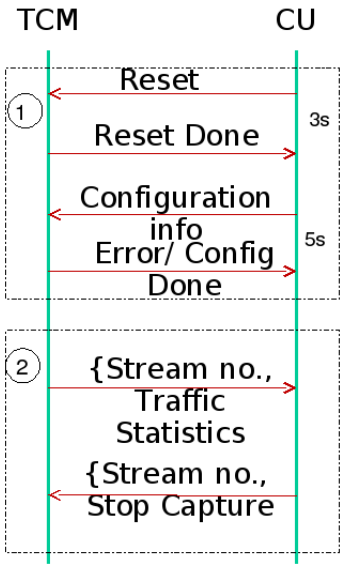


Figure 6.7: Protocol between TCM and CU

Implementation Performance

The performance in terms of the number of false alarms raised has already been discussed in previous sections. Based on the system built around the concepts, and it's implementation, the latencies involved in the implementation were determined. It should be seen that the bottleneck lies mostly in the hardware implementation (since that is the online module).

- **Description of FPGA hardware platform**. All hardware modules involved in the system were implemented in a Celoxica RC340 FPGA board, running at a speed of 65Mhz.

- **Performance** The following parameters were experimentally estimated:

    1. **Worst-case Per-Packet-Latency**: The worst case per-packet-latency is the additional time taken for processing a packet, due to the presence of the detection module. It was observed that the system took 1.64 clock cycles for the additional processing, for a window size of 100. This amounts to 30 nano seconds.

    2. **Attack Detection Time**: This is the time between an onset of attack and the moment the attack flag is asserted. This was observed to be 3 seconds, for an attack done at close to 100Mbps. In general, the faster the attack, the faster it will get detected.

CHAPTER 7

Conclusion and Future Work

7.1 Contribution of the Thesis

The aim of the work leading to this thesis is to address the problem of DoS and DDoS detection at the target end using machine learning techniques. With emphasis on hardware implementation, the thesis provides the following key contributions:

- A systems approach for DoS and DDoS detection at target using a Naìve Bayes classifier for TCP and UDP, with a design engineered for real time use and implementability.
- Design considerations taking into mind practical user problems with machine learning techniques.
- Light weight detection algorithm, with a note on processing latency and reaction time.
- Experiments were conducted to show the system efficiency and comparability with similar works.

7.2 Issues and Future Scope of Work

The model is simple and light weight in terms of detection, but while the design has been made keeping in mind flooding attacks (which form most of attacks carried out to the day), input parameters may have to be further fine tuned in order to be detecting convincingly a broader class of DoS attacks. This may involve including payload features as well. Also, at a user level, making the choice of some inputs to the system is very difficult even for an expert user. For example, it may be extremely difficult for even a seasoned network administrator to determine the right choice for

the proportion of abnormal traffic in the training input, unless manual checking is done over the entire dataset.

Further more, the scalability of the model to very high volume servers has to be examined. For example, it is quite possible to see a window full of SYN packets with a high volume server, while the signature for SYN flooding attacks still remain the same. Although it may look like increasing the size of the window may solve this problem, this has to be carefully examined before a conclusion could be given. This essentially calls for testing the system with variety of traffic at various volumes.

The independence assumption with respect to TCP has to be more critically examined. Even though it was a convenient option to assume complete independence with respect to flag related events based on our understanding of a variety of factors including sessions, the distribution of flags in sessions, the appearance of flags inside a single window, it was seen that this was a major source of false alarms (cost of simplification).

It was observed particularly during the onset of an attack, that there were a few windows which had reasonable amounts of attack traffic in them, but still parts of the window (which are assumed to be completely independent from the rest of the window parts) were good enough to drag the window just above the threshold, resulting in a false negative.

In a specific instance for example, a window with <RST=2, SYN=35, ACK=49, FIN/ACK=3, PSH/ACK=11, Others=0> (an onset of attack), it was seen that the probability of seeing 35 SYNs in a window was clearly low, but the probabilities of the other flag types were high owing to their prominent presence in some other windows (and not on windows which could have had 35 SYNs in training traffic). As a result, the window just managed to cross the threshold and be called normal. Such cases are certainly undesirable.

It is inferred that though sequence preservation may be an overkill, building some kind of a dependence between these flag events in a collective fashion will help model the system better, and also reduce false alarms. For example, some kind of a loose correlation coefficient between these six types could be formulated. These coefficients will also be considered during determining probability of a window, indicating shift to the conventional Bayes' probabilities.It should also be taken care that this does not result in too many extra computations.

Outside of the detection problem itself, a few other interesting areas of research surface when the intention is to build a fool proof DDoS protection system. Some of them are the following:

- **Extending to other protocols**. While we believe that a similar framework will work for other protocols, the research could be extended to include other protocols in the Internet protocol stack. Examples of such protocols include gateway level protocols, or application level protocols.
- **Research on sound mitigation techniques**. An equally important area of research is to find ways of handling attacks once they are detected. It is also to be remembered that these techniques in many cases may have to be applied in-line, which calls for extremely efficient methods to handle attacks. Possible mitigation techniques include graceful degradation, cryptographic methods to stateless monitoring and making launching further attacks tougher.

To conclude, in order to make a complete system for DoS attacks at all levels, the protocol spectrum should span other layers of the Internet stack, while taking care of the concerns mentioned above.

LIST OF PAPERS BASED ON THESIS

Vijayasarathy, R., Ravindran, B., and Raghavan, S. V. (2011) "A System Approach to Network Modeling for DDoS Detection using Naive Bayesian Classifiers". In the Proceedings of the Third International Conference on Communication and Networks (COMSNETS 2011), IEEE Press.

REFERENCES

[1] **Abadi, M.**, **M. Burrows**, **M. Manasse**, and **T. Wobber** (2005). Moderately hard, memory-bound functions. *ACM Trans. Internet Technol.*, **5**, 299–327. ISSN 1533-5399.

[2] **Ahmed, E.**, **A. Clark**, and **G. Mohay**, A novel sliding window based change detection algorithm for asymmetric traffic. *In Proceedings of the 2008 IFIP International Conference on Network and Parallel Computing*. IEEE Computer Society, Washington, DC, USA, 2008.

[3] **Ahmed, E.**, **G. Mohay**, **A. Nadarajan**, **B. Ravindran**, **A. Tickle**, and **R. Vijayasarathy**, *An Investigation into the Detection and Mitigation of Denial of Service(DoS) Attacks*, chapter 5. Springer, 2011. Under Publication.

[4] **Ahmed, E.**, **G. Mohay**, **A. Tickle**, and **S. Bhatia**, Use of ip addresses for high rate flooding attack detection. *In* **K. Rannenberg**, **V. Varadharajan**, and **C. Weber** (eds.), *Security and Privacy: Silver Linings in the Cloud*, volume 330 of *IFIP Advances in Information and Communication Technology*. Springer Boston, 2010, 124–135.

[5] **Arbor Networks** (2011). Arbor Networks. `www.arbornetworks.com`. Official Website of Arbor Networks.

[6] **Aura, T.**, **P. Nikander**, and **J. Leiwo**, DoS-resistant authentication with client puzzles. *In Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[7] **Back, A.** (2002). Hashcash - a denial of service counter-measure. Technical report.

[8] **Burges, C. J.** (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, **2**, 121–167. ISSN 1384-5810. URL `http://dx.doi.org/10.1023/A:1009715923555`. 10.1023/A:1009715923555.

[9] **Caprette, D. R.** (2005). Student's t-test for Independent Samples. `http://www.ruf.rice.edu/~bioslabs/tools/stats/ttest.html`.

[10] **Carl, G.**, **G. Kesidis**, **R. R. Brooks**, and **S. Rai** (2006). Denial-of-service attack-detection techniques. *IEEE Internet Computing*, **10**, 82–89.

[11] **CERT** (2000). Cert advisory Denial-of-Service developments. `http://www.cert.org/advisories/CA-2000-01.html`.

[12] **CERT** (2001). Distributed Denial of Service Tools. `http://www.cert.org/incident_notes/IN-99-07.html`. CERT Incident Note.

[13] **Chang, C.-C.** and **C.-J. Lin** (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, **2**, 27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[14] **Cisco** (2011). Cisco Guard DDoS Mitigation Appliances. `http://www.cisco.com/en/US/products/ps5888/index.html`. Last Accessed: July 2011.

[15] **Dittrich, D.** (1999). The Stacheldraht Distributed Denial of Service attack tool. `http://staff.washington.edu/dittrich/misc/stacheldraht.analysis`.

[16] **Eddy, W.** (2007). TCP SYN flooding Attacks and Common Mitigations. `http://tools.ietf.org/html/rfc4987`. RFC 4987, IETF published document.

[17] **Farid, D. M.**, **N. Harbi**, and **M. Z. Rahman** (2010). Combining Naive Bayes and decision tree for adaptive intrusion detection. *CoRR*, **abs/1005.4496**.

[18] **FBI National Press Office** (2011). Search warrants executed in the United States as part of ongoing Cyber Investigation. `http://www.fbi.gov/news/pressrel/press-releases/warrants_012711`.

[19] **Feinstein, L.**, **D. Schnackenberg**, **R. Balupari**, and **D. Kindred**, Statistical approaches to DDoS attack detection and response. *In Proc. DARPA Information Survivability Conference and Exposition, 2003*, volume 1. 2003.

[20] **Gavrilis, D.** and **E. Dermatas** (2005). Real-time detection of Distributed Denial-of-Service attacks using RBF networks and statistical features. *Comput. Netw. ISDN Syst.*, **48**(2), 235–245. ISSN 0169-7552.

[21] **Goodrich, M. T.**, Efficient packet marking for large-scale ip traceback. *In Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02. ACM, New York, NY, USA, 2002.

[22] **Goodrich, M. T.** (2008). Probabilistic packet marking for large-scale ip traceback. *IEEE/ACM Trans. Netw.*, **16**, 15–24.

[23] **Hattiwale, V.** (2008). Intrusion detection using Hidden Markov Models. Technical report, IIT Madras.

[24] **Henry, A.** (2008). Wannabe hackers can now rent a Botnet. `http://www.pcworld.com/businesscenter/article/145931/wannabe_hackers_can_now_rentabotnet.html`. PC World Article.

[25] **Hruby, T.**, **K. van Reeuwijk**, and **H. Bos**, Ruler: high-speed packet matching and rewriting on npus. *In Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS. ACM, 2007.

[26] **Ioannidis, J.** and **S. M. Bellovin**, Implementing pushback: Router-based defense against ddos attacks. *In In Proceedings of Network and Distributed System Security Symposium*. 2002.

[27] **Jin, C.**, **H. Wang**, and **K. G. Shin**, Hop-count filtering: an effective defense against spoofed ddos traffic. *In Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03. ACM, New York, NY, USA, 2003.

[28] **Jin, H.**, **Y. Pan**, and **N. Xiao** (eds.) (2004). *Grid and Cooperative Computing - GCC 2004 Workshops: GCC 2004 International Workshops, IGKG, SGT, GISS, AAC-GEVO, and VVS, Wuhan, China, October 21-24, 2004. Proceedings*, volume 3252 of *Lecture Notes in Computer Science*. Springer, 2004.

[29] **Jin, S.** and **D. Yeung**, A Covariance Analysis model for DDoS attack detection. *In Communications, 2004 IEEE International Conference*. China, 2004.

[30] **Jin, S.-Y.** and **D. Yeung**, DDoS detection based on Feature Space Modeling. *In Proc. of 2004 International Conference on Machine Learning and Cybernetics*, volume 7. IEEE Press, 2004.

[31] **Juels, A.** and **J. G. Brainard**, Client Puzzles: A Cryptographic countermeasure against connection depletion attacks. *In NDSS*. 1999.

[32] **Kang, J.**, **Y. Zhang**, and **J. bin Ju**, Detecting DDoS attacks based on multistream fused HMM in source-end network. *In International Conference on Cryptography and Network Security*. 2006.

[33] **Kerk, J.** (2007). Estonia recovers from massive DDoS attack. `http://www.computerworld.com/s/article/9019725/Estonia_recovers_from_massive_DDoS_attack`.

[34] **Khor, K.-C.**, **C.-Y. Ting**, and **S.-P. Amnuaisuk** (2009). From feature selection to building of Bayesian classifiers: A network intrusion detection perspective. *American Journal of Applied Sciences 6 (11)*, 1949– 1960.

[35] **Kline, J.**, **S. Nam**, **P. Barford**, **D. Plonka**, and **A. Ron**, Traffic anomaly detection at fine time scales with bayes nets. *In Proceedings of the 2008 The Third International Conference on Internet Monitoring and Protection*. IEEE Computer Society, Washington, DC, USA, 2008.

[36] **Longman, W.** (2005). Trends in Botnet Command and Control, giac general security essentials (gsec), 2005. www32.giac.org.

[37] **Mahoney, M. V.** and **P. K. Chan**, Learning nonstationary models of normal network traffic for detecting novel attacks. *In KDD 02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2002.

[38] **Malik, S.**, *Network Security Principles and Practices*, chapter 14. Cisco Press, 2002.

[39] **Mcafee Inc.** (2011). DDoS-Trinoo. `http://www.mcafee.com/threat-intelligence/malware/default.aspx?id=98488`. Last Accessed: Jul 2011.

[40] **Mirkovic, J.**, *Internet Denial of Service Attacks and Defense Mechanisms*. Pearson Education Inc., Newjersey, 2004.

[41] **Mirkovic, J.**, **G. Prier**, and **P. L. Reiher**, Attacking *ddos* at the source. *In Proceedings of the 10th IEEE International Conference on Network Protocols*, ICNP '02. IEEE Computer Society, Washington, DC, USA, 2002.

[42] **Mirkovic, J.**, **G. Prier**, and **P. L. Reiher**, Attacking ddos at the source. *In Proceedings of the 10th IEEE International Conference on Network Protocols*, ICNP '02. IEEE Computer Society, Washington, DC, USA, 2002.

[43] **Mirkovic, J.** and **P. Reiher** (2004). A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, **34**(2), 39–53. ISSN 0146-4833.

[44] **Mirkovic, J.**, **M. Robinson**, and **P. Reiher**, Alliance formation for ddos defense. *In Proceedings of the 2003 workshop on New security paradigms*, NSPW '03. ACM, New York, NY, USA, 2003.

[45] **MIT Lincoln Laboratory** (2000). DARPA Intrusion Detection Datasets. `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html`. Last Accessed: Jul 2011.

[46] **Moore, Andrew W.** (2004). Naive Bayes Classifiers. `http://www.autonlab.org/tutorials/naive02.pdf`.

[47] **Nam, S. Y.** and **T. Lee**, Memory-efficient IP filtering for countering ddos attacks. *In Proceedings of the 12th Asia-Pacific network operations and management conference on Management enabling the future internet for changing business and new computing services*, APNOMS'09. Springer-Verlag, Berlin, Heidelberg, 2009.

[48] **Nazario, J.**, Political DDoS: Estonia and beyond(invited talk). *In Seventeeth USENIX Security Symposium*. San Josa, CA, USA, 2008.

[49] **Oikonomou, G.** and **J. Mirkovic**, Modeling human behavior for defense against flash-crowd attacks. *In Proceedings of the 2009 IEEE international conference on Communications*, ICC'09. IEEE Press, Piscataway, NJ, USA, 2009.

[50] **Rabiner, L.** (1989). A tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, **77**(2).

[51] **Radware** (2011). Guard against DoS and DDoS Attacks. `http://www.radware.com/Solutions/Enterprise/Security/DoSProtection.aspx`. Last accessed: Jul 2011.

[52] **Seo, J.**, **C. Lee**, **T. Shon**, **K.-H. Cho**, and **J. Moon**, A new DDoS detection model using multiple SVMs and TRA. *In Machine Learning and Cybernetics*, volume 3823. Springer, 2005.

[53] **Seo, J.-T.**, **C. Lee**, and **J. Moon**, Defending ddos attacks using network traffic analysis and probabilistic packet drop. *In GCC Workshops*. 2004.

[54] **Shanbhag, S.** and **T. Wolf**, Evaluation of an online parallel anomaly detection system. *In GLOBECOM*. 2008.

[55] **Shon, T.**, **Y. Kim**, **C. Lee**, and **J. Moon**, A Machine Learning framework for network anomaly detection using SVM and GA. *In Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*. Nagoya, Japan, 2005.

[56] **Stebila, D.**, **L. Kuppusamy**, **J. Rangasamy**, **C. Boyd**, and **J. G. Nieto**, Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols. *In Proceedings of the 11th international conference on Topics in cryptology: CT-RSA 2011*, CT-RSA'11. Springer-Verlag, Berlin, Heidelberg, 2011.

[57] **Sung, M.** and **J. Xu**, Ip traceback-based intelligent packet filtering: A novel technique for defending against internet ddos attacks. *In Proceedings of the 10th IEEE International Conference on Network Protocols*, ICNP '02. IEEE Computer Society, Washington, DC, USA, 2002.

[Symantec] **Symantec** (). DDoS Stacheldraht Client Request. `http://www.symantec.com/business/security_response/attacksignatures/detail.jsp?asid=20008`.

[58] **Thapngam, T.**, **S. Yu**, **W. Zhou**, and **G. Beliakov**, Discriminating DDoS attack traffic from flash crowd through packet arrival patterns. *In Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference*. School of Information Technology, Deakin University, Burwood, VIC 3125, Australia, Shanghai, China, 2011.

[59] **THE HINDU** (2011). Sony executives apologise for network security breach. `http://www.thehindu.com/sci-tech/technology/article1983272.ece`.

[60] **Trinity Security Services** (2003). The Distributed Denial of Service Attack. `http://archive.networknewz.com/networknewz-10-20030924TheDistributedDenialofServiceAttack.html`.

[61] **University of California, I.** (1999). KDD Cups 1999 data. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`.

[62] **Wang, H.**, **C. Jin**, and **K. G. Shin** (2007). Defense against spoofed ip traffic using hop-count filtering. *IEEE/ACM Trans. Netw.*, **15**, 40–53.

[63] **Wang, X.** and **M. K. Reiter**, Defending against denial-of-service attacks with puzzle auctions. *In Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-1940-7.

[64] **Xiang, Y.** and **W. Zhou**, Mark-aided distributed filtering by using Neural Network for DDoS defense. *In Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE*. 2005.

[65] **Xie, Y.** and **S.-Z. Yu**, A Novel model for detecting Application layer DDoS attacks. *In IMSCCS '06: Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences - Volume 2 (IMSCCS'06)*. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2581-4.

[66] **Xu, X.**, **Y. Sun**, and **Z. Huang**, Defending DDoS attacks using Hidden Markov Models and Cooperative Reinforcement Learning. *In PAISI'07: Proceedings of the 2007 Pacific Asia conference on Intelligence and security informatics*. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 978-3-540-71548-1.

[67] **Yaar, A.**, **A. Perrig**, and **D. Song**, Siff: A stateless internet flow filter to mitigate ddos flooding attacks. *In In IEEE Symposium on Security and Privacy*. 2004.

[68] **Yamamura, M.** (2000). `http://service1.symantec.com/sarc/sarc.nsf/html/W32.DoS.Trinoo.html`.

[69] **Yao, G.**, **J. Bi**, and **Z. Zhou**, Passive ip traceback: capturing the origin of anonymous traffic through network telescopes. *In Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, SIGCOMM '10. ACM, New York, NY, USA, 2010.

[70] **Yu, S.**, **T. Thapngam**, **J. Liu**, **S. Wei**, and **W. Zhou**, Discriminating ddos flows from flash crowds using information distance. *In Proceedings of the 2009 Third International Conference on Network and System Security*, NSS '09. IEEE Computer Society, Washington, DC, USA, 2009.

[71] **Yu, S.**, **W. Zhou**, **R. Doss**, and **W. Jia** (2011). Traceback of ddos attacks using entropy variations. *IEEE Trans. Parallel Distrib. Syst.*, **22**, 412–425.

[72] **Zhou, Z.**, **D. Xie**, and **W. Xiong** (2009). A novel distributed detection scheme against DDoS attack. *Journal of Networks (JNW)*, **4**(9), 921–928.