DEEP PACKET INSPECTION APPLICATIONS FOR TRAFFIC CLASSIFICATION AND SECURITY MONITORING

Ph.D. Thesis

By

MAYANK SWARNKAR

DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

SEPTEMBER 2019

DEEP PACKET INSPECTION APPLICATIONS FOR TRAFFIC CLASSIFICATION AND SECURITY MONITORING

A THESIS

Submitted in partial fulfillment of the
requirements for the award of the degree
of
DOCTOR OF PHILOSOPHY

by
MAYANK SWARNKAR

DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE
SEPTEMBER 2019

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **DEEP PACKET INSPECTION APPLICATIONS FOR TRAFFIC CLASSIFICATION AND SECURITY MONITORING** in the partial fulfillment of the requirements for the award of the degree of **DOCTOR OF PHILOSOPHY** and submitted in the **DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from July 2014 to September 2019 under the supervision of Dr. Neminath Hubballi, Associate Professor, Indian Institute of Technology Indore.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other institute.

Signature of the student with date
(MAYANK SWARNKAR)

This is to certify that the above statement made by the candidate is correct to the best of my/our knowledge.

Signature of Thesis Supervisor with date
(Dr. NEMINATH HUBBALLI)

MAYANK SWARNKAR has successfully given his Ph.D. Oral Examination held on ______________.

Signature of Chairperson (OEB)
Date:

Signature of External Examiner
Date:

Signature(s) of Thesis Supervisor(s)
Date:

Signature of PSPC Member #1
Date:

Signature of PSPC Member #2
Date:

Signature of Convener, DPGC
Date:

Signature of Head of Discipline
Date:

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to people who in one or the other way contributed by making this time as learnable, enjoyable, and bearable. At first, I would like to thank my supervisor **Dr. Neminath Hubballi**, who was a constant source of inspiration during my work. Without his constant guidance and research directions, this research work could not be completed. His continuous support and encouragement has motivated me to remain streamlined in my research work. I am also grateful to **Dr. Surya Prakash**, HOD of Computer Science for all his extended help and support.

I am thankful to **Dr. Aruna Tiwari** and **Dr. Vimal Bhatia**, my research progress committee members for taking out some valuable time to evaluate my progress all these years. Their valuable comments and suggestions helped me to improve my work at various stages.

My sincere acknowledgment and respect to **Prof. Pradeep Mathur**, Director, Indian Institute of Technology Indore for providing me the opportunity to explore my research capabilities at Indian Institute of Technology Indore.

I would like to appreciate my colleagues and friends especially, Dr. Robin Bhadoria, Mr. Rajendra Choudhary, Dr. Mayank Modak, Mr. Ram Sharma, Mr. Aditya Chouhan, Mr. Saurabh Yadav, Dr. Syed Sadaf Ali, Mr. Animesh Chaturvedi, Dr. Rakesh Sharma, Mr. Chandan Gautam and Mr. Navneet Pratap Singh. Hearty thanks to my lab mates Dr. Nikhil Tripathi and Ms. Pratibha Khandait for their cooperation during my research. I would like to express my heartfelt respect to my parents for their love, care and support they have provided to me throughout my life. Special thanks to my younger brother for his support and encouragements. Finally, I am thankful to all who directly or indirectly contributed, helped and supported me. To sign off, I write a quote by Zora Neale Hurston -

> “Research is formalized curiosity. It is poking and prying with a purpose.”

Mayank Swarnkar

Dedicated to My Family

ABSTRACT

Deep Packet Inspection (DPI) is a commonly used network traffic monitoring technique which finds applications in variety of network management activities. Two prominent use cases of DPI are in traffic classification and security monitoring. DPI based traffic monitoring techniques screen the payload or content within network packets to identify applications and detect security issues like worm breakouts. Network management activities based on DPI are known to be accurate and at the same time computationally expensive. In this thesis, we seek to design effective DPI based network traffic monitoring methods for three tasks - traffic classification, zero day attack detection in web traffic and detecting spam users in Voice over Internet Protocol (VoIP) network.

DPI based traffic classification methods generate application signatures using invariant payload content. Both supervised and unsupervised methods are proposed in the literature for this task. We propose three DPI based traffic classification methods namely *RDClass*, *BitCoding* and *BitProb* in this thesis. *RDClass* is an unsupervised traffic classifier which automatically identifies a set of keywords for an application when presented with unknown network flows. It finds the relative distance between identified keywords to generate application specific signatures. *RDClass* is designed to handle only text based protocols as it requires identifying meaningful keywords from network flows. *BitCoding* and *BitProb* are supervised traffic classification methods proposed to handle all types of application protocols (text, binary, open standard and proprietary). These two methods generate application specific bit level signatures by identifying invariant bits from network flows of a particular application. We experiment with two publicly available datasets and one private dataset containing traffic of a variety of applications and show that these methods can classify applications with very high accuracy.

Detecting zero day attacks is usually done with payload based anomaly detection systems. We propose two DPI based anomaly detection methods, *Rangegram* and *OCPAD*, to detect zero day attacks in web traffic. *Rangegram* and *OCPAD* generate

short sequences from benign application packet payloads and find deviations in occurrence range or probability of short sequences to identify anomalous packets (attacks). We evaluate the detection performance of both the detection methods with few HTTP based attacks and show that they can detect anomalies in the web traffic accurately.

We propose a DPI based method *SpamDetector* to detect VoIP spam callers. It uses DPI to extract a set of call related parameters from Session Initiation Protocol (SIP) packets. Using these call parameters, it generates a directed weighted graph representing social interaction among the users. *SpamDetector* identifies nodes which are different from their local neighborhood as anomaly and hence as spam users. We evaluate the detection performance of *SpamDetector* with a large simulated user base and show that it can detect the spam users with a good detection rate.

Publications from Thesis

Published/Accepted

Journals

J1. Neminath Hubballi, **Mayank Swarnkar**, “BitCoding: Network Traffic Classification through Encoded Bit Level Signatures”, IEEE/ACM Transactions on Networking, Vol 26, Issue 5, pp 1-13, 2018.

J2. **Mayank Swarnkar**, Neminath Hubballi, “RDClass: On Using Relative Distance of Keywords for Accurate Network Traffic Classification”, IET Networks, Vol 7, Issue 4, pp. 1-8, 2018.

J3. **Mayank Swarnkar**, Neminath Hubballi, “OCPAD: One Class Naive Bayes Classifier for Payload based Anomaly Detection”, Expert Systems with Applications, Elsevier, Vol 64, Issue 1, pp. 330-339, 2016.

J4. **Mayank Swarnkar**, Neminath Hubballi, “SpamDetector: Detecting Spam Callers in VoIP with Graph Anomalies”, Security and Privacy, Wiley, pp. 1-18.

International Conferences

C1. Neminath Hubballi, **Mayank Swarnkar**, “BitCoding: Protocol Type Agnostic Robust Bit Level Signatures for Traffic Classification”, Proceedings of 36th IEEE Global Communication Conference (GLOBECOM’17), Singapore, pp. 1-6, 2017.

C2. Sainath Batthala, **Mayank Swarnkar**, Neminath Hubballi, Maitreya Natu, “VoIP Profiler: Profiling Voice over IP User Communication Behaviour,” Proceedings of 9th International Conference on Availability, Reliability and Security (ARES’16), Salzburg, pp. 312-320, 2016.

C3. **Mayank Swarnkar**, Neminath Hubballi, “Rangegram: A Novel Payload based Anomaly Detection Technique against Web Traffic,” Proceedings of 9th IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS’15), Kolkata, pp. 1-6, 2015.

Submitted

Journals

J5. Hubballi, N., **Swarnkar, M.,** and Conti, M., “BitProb: Probabilistic Bit-Level Signatures for Accurate Network Traffic Classification”, *IEEE Transactions on Network and Service Management*, pp. 1-11, 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network traffic monitoring has several applications including network management, identifying security issues, behavioural analysis, etc [53]. Network packets carry application data and monitoring these packets gives insight into what type of applications are being used and type of data being carried. Other prominent use of traffic monitoring is in identifying security issues with behavioral profiles of users and applications [69]. Tools like Intrusion Detection Systems (IDSs) [110] are used for this purpose. In general, traffic monitoring can be done at two levels. At an abstract level, traffic statistics are used to draw inferences and in a detailed analysis, actual content of network traffic is inspected as described below.

I. Statistical Methods: Statistical network traffic monitoring methods inspect the traffic at an abstract level by analyzing only packet headers. Details like IP addresses, port numbers, flow level details, inter-arrival time between packets, etc. are used for this purpose. These monitoring methods can be used to identify cases like worm breakouts, port scanning, etc. These methods are computationally less expensive and can also be used when application data is encrypted.

II. Deep Packet Inspection (DPI) based Methods: DPI based traffic monitoring methods inspect the protocol header as well as data contained within a packet. For example, a packet carrying HTTP data can be identified with certain keywords taken from the HTTP payload. Due to this, these methods can not be used to monitor encrypted application data. Despite of this limitation, DPI based methods are widely used for traffic monitoring as they are more accurate compared to statistical methods [40, 48].

Rest of this chapter is organized as follows. In Section 1.1, we give an overview of DPI and its applications for different purposes. In Section 1.2, we present the motivation behind the research work presented through this thesis. We give a summary of thesis contributions in Section 1.3 and in Section 1.4, we give the outline of the rest of the thesis.

1.1 Deep Packet Inspection (DPI) and Its Applications

DPI is a traffic monitoring method which inspects the protocol header as well as content within a packet. DPI inspects the content of packets passing through a given checkpoint like IDS sensors, URL filtering engines, etc. Some of the popular tools [33] which use DPI to monitor the network traffic are Protocol and Application Classification Engine (PACE) [20], OpenDPI [18], L7-filter [12], nDPI [16], Libprotoident [13], Network Based Application Recognition (NBAR) [15] and Snort [23]. DPI based traffic monitoring methods find their applications in various fields such as network traffic classification, network security and management, etc. These applications of DPI are described below.

1.1.1 Network Traffic Classification

Network traffic classification [41, 43] is an automated process which categorizes traffic into different classes based on the application/service. Traffic classification is performed for purposes such as bandwidth allocation, defining security rules, etc. For example, live streaming applications and applications which involve sending files, multimedia, etc. have different bandwidth requirements and thus, should be treated differently. For this purpose, it is necessary to differentiate the traffic generated from these applications using appropriate network traffic classification methods. Traffic classification is also used to filter the traffic generated from applications that violate network policies [95]. Various traffic classification methods use DPI to analyze the content of the packets passing through a checkpoint. Based on the packet content, these methods identify the packet as belonging to a particular application. For example, a DPI based firewall can classify a HTTP packet by analyzing the content of its payload.

1.1.2 Network Security

An adversary on Internet can harm a benign end user or a network by infecting it with malicious programs. The adversary can bind such malicious programs either to a genuine application executable files or simply attach it to an image and send the resulting file to the end user. In order to detect the traffic generated by such malicious programs, Intrusion Detection Systems (IDSs) [102] are deployed within the network. Broadly, IDSs can be categorized into two classes as follows:

Signature based IDSs: In these types of IDSs, signatures of various malicious applications and their actions are stored in a signature database. An alarm is generated when any application and/or its activity matches with any signature in the database. Snort [23] and Bro [88] are the two popular open source signature based IDS. These tools allow to write signatures using keywords, regular expressions and also their position within the payload along with packet header parameters and the direction of packet (incoming and outgoing). These IDSs use DPI to extract the payload content from the packet under consideration and compare it with the signatures stored in the database to detect malicious activity.

Anomaly based IDSs: Anomaly based IDSs detect network intrusions by identifying anomalies in the network traffic. These IDSs operate in two phases - training and testing phases. During training phase, DPI is used to extract candidate features to create a normal traffic profile using which IDS is trained. These features are in the form of keywords, n-grams, bit sequence, etc. During testing phase, IDS creates profiles using the same candidate features and compares it with the normal profile generated during training phase. If the profile generated during testing phase differs significantly from the one generated during training phase, IDS detects the presence of anomalous traffic in the network and thus, raises an alarm.

1.1.3 Network Management

DPI is widely used for network management tasks such as load balancing [111] in specific applications like VoIP. Load balancing refers to efficiently distributing incoming network traffic across a group of back-end servers, also known as a server farm. For load balancing, payload of the packets is first analyzed by a load balancer using DPI to find out the details such as type of application, attached multimedia, etc. Based on these details, the load balancer forwards the packets to one of the multiple back-end servers.

1.2 Motivation

Internet penetration is increasing across the globe. This is reflected from the fact that more than half of the world's population has access to Internet today [7]. Figure 1.1 [7] shows the continuous growth in the number of Internet users in last 23 years. Today Internet is used for variety of applications such as online gaming, video streaming, e-mail, e-commerce, banking, etc. There are millions of mobile applications developed

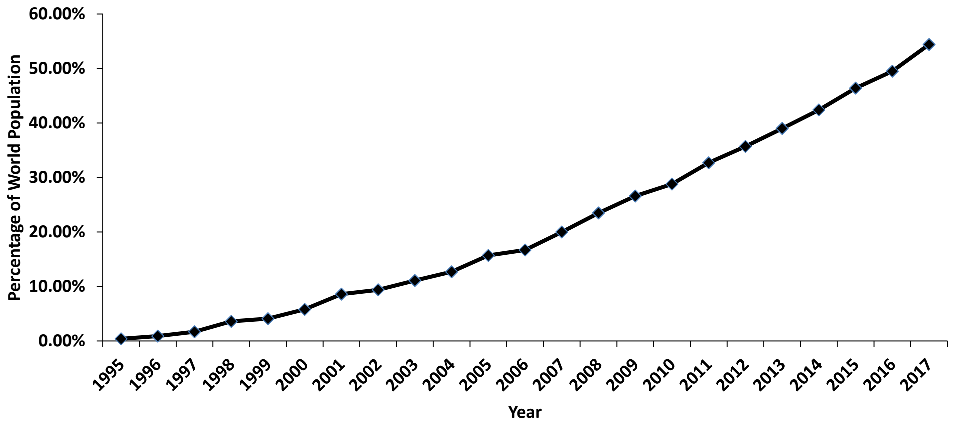


Figure 1.1: Growth in Number of Internet Users over the Years [7]

for different mobile platforms [28, 103]. All these applications generate a heterogeneous mix of network traffic. Every application has some requirements of Quality of Service (QoS) in terms of jitter, end to end delay, etc. for its useful usage. Further these applications need to be monitored for security compliance and any violations need to be detected. This can be done by traffic monitoring as described previously. Inspecting only packet header or statistics based methods are computationally less expensive and are recently shown to be less accurate [86]. This lead to the development of Deep Packet Inspection (DPI) based network monitoring methods. These methods analyze the actual content of packet and thus, give more accurate results. However, there are few limitations of using DPI based methods for network traffic monitoring. First, analyzing the packet payload is proved to be computationally expensive. Second, analyzing the actual packet content also hinders users' privacy. Third, with the development of applications which use binary or proprietary protocol, the DPI based methods eventually fail to monitor the traffic. Thus, we argue that a DPI based network traffic monitoring method should possess the following two properties:

A. Maximum Accuracy by Examining Minimum Payload Content: Normally DPI based methods use entire payload content of a packet for inspection. This provides good accuracy but it comes with the cost of high computational overhead and compromises users' privacy. Thus, the traffic monitoring method should require inspecting minimum payload content without compromising with the accuracy.

B. Protocol Agnostic: It is desirable that a DPI based traffic classification method should be able to accurately classify different types of text-based, binary and/or proprietary protocols.

An important application of Deep Packet Inspection is in the area of Zero Day attack detection (attacks which are not seen before). DPI based methods analyze the payload portion of incoming network traffic to identify attacks carrying malicious content.

In this thesis, we propose different DPI based network traffic monitoring methods possessing these properties for three different purposes - 1) Network traffic classification, 2) Zero day attack detection in web traffic and 3) Profiling user behavior and detecting spam users in VoIP network traffic. The proposed methods can analyze network packets with less computational overhead and thus, can be deployed in the real networks for real time monitoring.

1.3 Thesis Contribution

In this thesis, we describe new methods for network traffic classification, zero day attack detection in web traffic and VoIP spam detection using DPI. The contribution of our thesis are as follows:

I. Byte Level Payload Analysis for Network Traffic Classification: Traffic classification approaches proposed in the literature use only a set of keywords to identify the applications which leads to misclassification. Thus, as our first contribution, we propose *RDClass*, an automated content based traffic classifier, for classifying network flows[1]. *RDClass* uses a set of keywords extracted from an application flow and uses the relative distance between these keywords to generate application specific signatures. These signatures are then used for traffic classification. We represent the set of keywords and their relative distances in the form of a state transition machine called *Relative Distance Constrained Counting Automata (RDCCA)*. This state transition machine can check both ordering of keywords and their relative distance within the payload to classify the flow under consideration. *RDClass* can automatically extract a set of keywords from the application flows and find their relative ordering to generate *RDCCA* when presented with unknown application flows. We experiment with different public and private datasets containing traffic generated from a variety of applications and show that *RDClass* has better classification performance as compared to previously known traffic classification methods which use only ordering of keywords.

II. Bit Level Payload Analysis for Network Traffic Classification: With the

[1] Flow is the combination of packets having same source IP address, destination IP address, source port number, destination port number and transport layer protocol

number of proprietary network protocols on the rise and many of them using bit level encoding, byte level signatures are not effective in identifying applications. Thus, we propose two supervised bit-level traffic classification methods *BitCoding* and *BitProb* that can generate application specific signatures using only first n bits extracted from a flow. Our first method *BitCoding* encodes generated signatures of each application in order to compress them and then transforms the compressed signatures into a state transition machine called *Transition Constrained Counting Automata* (*TCCA*). This *TCCA* is subsequently used for classification purpose. Since *BitCoding* considers only invariant bits while matching application signatures, this increases the chances of signature overlap which may lead to misclassifications. However, our second method *BitProb* considers the occurrence probability of bit values at each position without omitting any bit value during signature match due to which it can identify applications more accurately. Similar to *BitCoding*, *BitProb* also transforms the generated signatures into a state transition machine called *Probabilistic Counting Deterministic Automata* (*PCDA*). Our proposed methods generate short application specific signatures and thus, are computationally less expensive. We evaluate the classification performance of *BitCoding* and *BitProb* on two public and one private datasets containing different text, binary and/or proprietary protocols and furnish the results. We show that both these methods can classify the applications with very high accuracy independent of the protocol type which makes them protocol-type agnostic. We extend our experiments and show that these methods can be ported from site to site with little compromise in detection performance. We also compare our proposed methods with recently proposed traffic classification methods and show that our methods outperform the previous methods.

III. Zero Day Attack Detection in Web Traffic: The attacks whose signatures are not known to traffic monitoring systems such as IDS are commonly known as zero day attacks. As our third contribution, we propose two methods - *Rangegram* and *OCPAD* to detect zero day attacks in web traffic. As a representative, we choose web traffic as attacks against HTTP are very common. Our first payload based anomaly detection method *Rangegram* has two phases of operations - training and testing phases. During training phase, *Rangegram* generates n-grams (short sequences of length n from a string) from a set of benign packets and finds the occurrence frequency range of generated n-grams in a packet and stores it in our proposed efficient data structure called *Min-Max-Tree*. During testing phase, n-grams are extracted from a test HTTP packet. These n-grams are compared with n-grams stored in *Min-Max-Tree* and those n-grams which are not found in *Min-Max-Tree* or their occurrence frequency within the packet is not in the normal range are counted. If this count is greater than a threshold, *Rangegram* considers the packet as anomalous. Our second anomaly detection method *OCPAD* uses a version of Multinomial Bayesian one class classification technique for accurately detecting anomalous payloads. In particular, *OCPAD* uses likelihood of each n-gram occurrence in a payload of known non-malicious HTTP packets as a measure to derive the degree of maliciousness of a packet. *OCPAD* also has two phases of operations as training and testing phases. During training phase, *OCPAD* generates the likelihood range of each n-gram occurrence from every packet and stores it in our proposed efficient data structure called *Probability-Tree*. During testing phase, if the n-grams generated from the payload of HTTP packet under consideration is not found in the database or its likelihood of occurrence in a packet is not in the range generated during training phase, *OCPAD* considers the packet as anomalous. We evaluate the detection performance of both the detection approaches (*Rangegram* and *OCPAD*) on three publicly available attack datasets and one normal dataset generated in our testbed setup. We show that both of our proposed methods outperform a closely related work.

IV. Detecting Spam Callers in VoIP Network Traffic: As our last contribution, we propose *SpamDetector*, a graph based method to detect VoIP spam users in a network. *SpamDetector* uses DPI to extract some parameters from Session Initiation Protocol (SIP) (an application layer VoIP signaling protocol) [69] packets. These details are further used to generate *Call Detail Record (CDR)* of VoIP users. Subsequently, *CDR* is used to obtain a set of call parameters and create a directed, weighted call graph (who call whom graph). *SpamDetector* uses the generated call graph for spam user detection. *SpamDetector* identifies anomalies in the graph by considering the local neighbourhood of a node and assigns a label based on how similar the node is in comparison to its neighbours. To find how similar a node is with its neighbours, *SpamDetector* uses a parameter called *Spam Outlier Factor (SOF)*. If the calculated *SOF* of the node under consideration is greater than the threshold *SOF* of a neighbouring node, it votes the node under consideration as spam. In a similar way, other neighbouring nodes also votes the considered node as normal or spam depending on their threshold *SOF*. If the spamming votes are found in majority, *SpamDetector* eventually declares the node as spam. We evaluate the detection performance of *SpamDetector* by creating a large simulated user base and show that it can detect the spam users with a very high accuracy. We also compare *SpamDetector* with one of the recent and closely related work and show that *SpamDetector* outperforms it.

1.4 Organization of Thesis

The rest of this thesis is organized as follows:

Chapter 2: In this chapter, we discuss previously known DPI based methods for the purpose of network traffic classification, zero day attack detection and VoIP spam detection.

Chapter 3: In this chapter, we describe our first contribution *RDClass* that uses keywords specific to application layer protocols and relative distance between the consecutive keywords to classify applications.

Chapter 4: In this chapter, we describe two traffic classification methods *BitCoding* and *BitProb* which generate bit level signatures and subsequently use it for traffic classification.

Chapter 5: In this chapter, we describe two methods *Rangegram* and *OCPAD* to detect zero day attacks in web traffic using n-gram analysis on HTTP packet payload.

Chapter 6: In this chapter, we describe a graph based method *SpamDetector* to detect VoIP spam users by analyzing calling behaviour of VoIP users.

Chapter 7: This chapter summarizes the work presented in this thesis and provides directions to future work in this area.

Chapter 2

Literature Survey

Deep Packet Inspection (DPI) is a crucial task in network management and network security domains. Several network management related tasks such as traffic classification [50, 82] and Quality-of-Service (QoS) of applications [35] can be enhanced if an efficient payload analysis scheme is deployed in the monitored network. Also, several network security monitoring tools like Intrusion Detection Systems use DPI for identifying intrusions [50]. As discussed in the last chapter, in this thesis, we propose DPI based methods for three different tasks - 1) Network traffic classification, 2) Zero day attack detection in web traffic and 3) Detecting spam callers in VoIP network traffic. Several works in the literature discuss methods that use DPI for the above tasks. We categorize these methods into different classes as shown in Figure 2.1. In this chapter, we review some of the methods which are closely related to DPI based methods proposed in this thesis. We also discuss few shortcomings in previously known methods, implications and research gaps in this area which motivated us to pursue the work presented in this thesis.

The structure of the rest of this chapter closely follows Figure 2.1. In Section 2.1, we describe previously known methods for network traffic classification. We discuss anomaly detection systems used to detect zero day attacks in the network traffic in Section 2.2. In Section 2.3, we describe previously known methods to detect spam callers in the VoIP network. In Section 2.4, we finally conclude the chapter by describing the research gaps in this area which motivated us to pursue the work presented

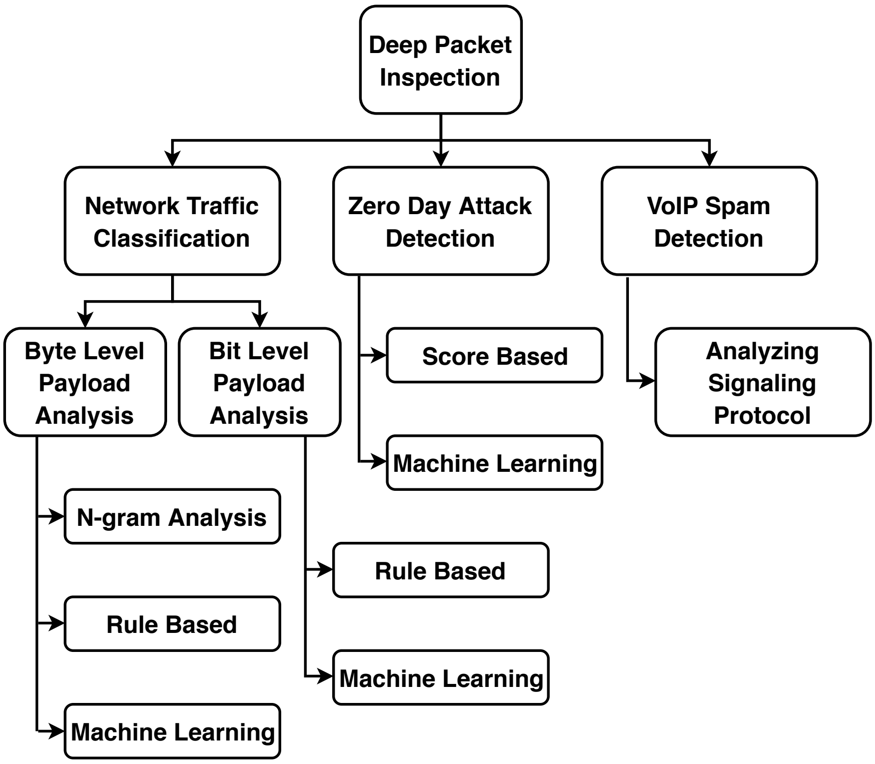

Deep Packet Inspection
Network Traffic Classification
Zero Day Attack Detection
VoIP Spam Detection
Byte Level Payload Analysis
Bit Level Payload Analysis
Score Based
Analyzing Signaling Protocol
Machine Learning
N-gram Analysis
Rule Based
Rule Based
Machine Learning
Machine Learning

Figure 2.1: Deep Packet Inspection Taxonomy

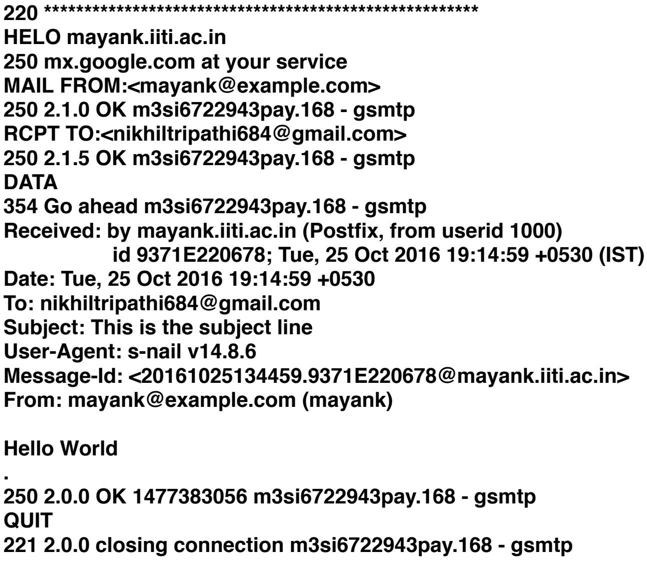
in rest of the thesis.

2.1 Network Traffic Classification

Several approaches have been proposed in the literature for traffic classification using payload analysis. These approaches can be divided into two different categories - i) *Byte level* and ii) *Bit level* payload analysis. Most of the approaches available in the literature use byte level payload analysis for traffic classification. However, due to various drawbacks in byte level payload analysis based approaches, researchers in the networking community have recently started following a paradigm shift towards developing bit level payload analysis based traffic classification approaches. In this section, we first describe previously known byte level payload analysis based works and their limitations. Subsequently, we discuss approaches that use bit level payload analysis for traffic classification.

2.1.1 Byte Level Payload Analysis for Traffic Classification

For network traffic classification, byte level payload analysis methods extract byte level information such as keywords and n-grams from the payload. Keywords are the most frequently occurring words in the flow payloads of any application protocol while n-grams are the sub-sequences extracted from the payload. For example, an SMTP payload is shown in Figure 2.2 and the keywords and n-grams that can be extracted from this payload are shown in Table 2.1. In this subsection, we describe previously known methods which extract these byte level information from payloads to classify network traffic.



```
220 ********************************************************************
HELO mayank.iiti.ac.in
250 mx.google.com at your service
MAIL FROM:<mayank@example.com>
250 2.1.0 OK m3si6722943pay.168 - gsmtp
RCPT TO:<nikhiltripathi684@gmail.com>
250 2.1.5 OK m3si6722943pay.168 - gsmtp
DATA
354 Go ahead m3si6722943pay.168 - gsmtp
Received: by mayank.iiti.ac.in (Postfix, from userid 1000)
          id 9371E220678; Tue, 25 Oct 2016 19:14:59 +0530 (IST)
Date: Tue, 25 Oct 2016 19:14:59 +0530
To: nikhiltripathi684@gmail.com
Subject: This is the subject line
User-Agent: s-nail v14.8.6
Message-Id: <20161025134459.9371E220678@mayank.iiti.ac.in>
From: mayank@example.com (mayank)

Hello World
.
250 2.0.0 OK 1477383056 m3si6722943pay.168 - gsmtp
QUIT
221 2.0.0 closing connection m3si6722943pay.168 - gsmtp
```

Figure 2.2: Simple Mail Transfer Protocol Payload

Table 2.1: Keywords and n-grams Extraction from a Simple Mail Transfer Protocol Payload

Type	Example
Keyword extracted from SMTP payload	HELO, MAIL, FROM, RCPT, TO, DATA
n-gram of size 2 generated from “HELO” string	“HE”, “EL”, “LO”

Rule Based Classification: Rule based network traffic classification methods extract keywords from the payload. These keywords are then utilized to form rules or signatures which are further used for classification purpose. These rules can be in the form of automata, regular expressions, trees, substring sequences, etc. Laser [84] is one such method that generates application signatures using Longest Common Subsequences (LCSs) from each application trace. Figure 2.3 shows the method of generating LCS. The generated LCSes are then used as signatures and

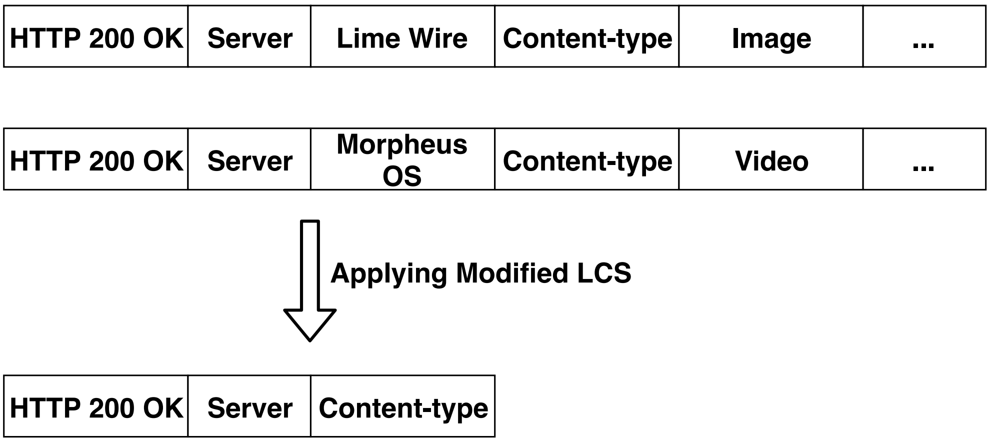


Figure 2.3: Generating Longest Common Subsequences [84]

matched against test application flow using Deoxyribo Nucleic Acid (DNA) sequence matching algorithm used in bioinformatics [85] for traffic classification. However, it is cumbersome to maintain up-to-date signatures for a large number of applications. To overcome this issue, Ye et al. proposed AutoSig [120] which automatically generates application signature by extracting multiple common substring sequences from sample flows of the same application. These substrings use adaptive merging algorithm to form regular expressions. These regular expressions are then used as signatures for application classification. Since AutoSig can generate application signatures automatically, it saves a lot of time on manual analysis and updates signatures in time. Laser and AutoSig generate signatures of very large length because of this, matching test flow payload against these signatures either takes a very long time to complete, or requires prohibitively large space. As a result, these methods are not appropriate for traffic classification in real networks. As a solution to this problem, authors in [122] proposed CUTE which generates lightweight regular expressions from weighted keyword set. This weighted keyword set is generated by assigning weights to the words based on their appearance frequency for each application. These lightweight regular expressions result into faster traffic classification. However, in a recent work, [104], authors argued that methods such as Laser [84] and CUTE [122] which extract application signatures based on longest common substrings in application flows, suffer from high false-positive and false-negative rates due to the lack of context in signatures. Authors, thus, proposed a solution SANTaClass [104, 105] which uses keywords and its sequence in payload for protocol identification. SANTaClass starts with zero knowledge of protocols and takes payload of unidentified flows and automatically identifies the required keywords and their sequence in the payload to generate Prefix Tree Acceptor (PTA). Each PTA is a signature for an application which is subsequently used for classification. However, the drawback of this method is that it does not consider the position of the keywords in the payload and keywords with same sequence can be found in the payload of other application protocol also but at different position. For example, to identify BitTorrent protocol, SANTaClass [104, 105] generates two keywords namely 'BitTorrent' and 'protocol' and search for these keywords (appearing in order) in any given payload. However, these two keywords appear in order not only in BitTorrent protocol payload but it can also appear in a HTTP payload. By comparing these two keywords, BitTorrent protocol can be correctly identified but HTTP can also be classified as BitTorrent protocol. This results into misclassification. In order to resolve this issue, in Chapter 3 of the thesis, we propose a traffic classification method *RDClass* which uses relative distance between selected keywords. For example, in a BitTorrent protocol payload, the terms 'BitTorrent' and 'protocol' appear at the beginning of payload and in a HTTP payload, they can appear at arbitrary positions but within the first few bytes of the payload. Thus by incorporating the ordering and relative distance between the keywords, our method *RDClass* is able to classify the network traffic more accurately as described in Chapter 3 of this thesis.

N-gram based Traffic Classification: N-grams are the subsequences of length 'n' which are extracted consecutively from any string. An n-gram of size 1 is known as unigram, size 2 as bigram, size 3 as trigram and so on. For example, trigrams that can be generated from the string 'Hello World' are 'Hel','ell','llo','lo ', 'o W', 'Wor', 'orl' and 'rld'. For traffic classification, these n-grams are first extracted from the application payload and then used as features in various statistical and machine learning based models. Pro-Decoder [116] and Securitas [125] are two such methods which generate n-grams from payload content and combine them to form keywords based on the latent relationship between n-grams. These two methods rely on the fact that the probability distribution of n-grams generated from payload content of applications exhibit skewness in the distribution with few n-grams appearing more frequently and others appearing rarely. This skewness in the distribution allows to use the frequently appearing n-grams as a method of identifying applications. This is done by treating content of any protocol as a message made up of words and each word is made up of n-grams. By estimating the conditional probability of appearance of an n-gram given that a particular word has appeared in a message, the n-grams necessary to generate keywords are identified. Subsequently these keywords are aligned to form signature for application. However, Pro-Decoder [116] and Securitas [125] are highly computationally expensive as they use iterative algorithms to generate keywords. Wang et al. proposed ProHacker [117, 118] which is a non-parametric approach to identify applications in the network traffic. ProHacker first extracts the different order n-grams from the payload of application flows. These n-grams are then combined using Pitman–Yor process [96] to form keywords. These keywords are the vector of n-grams that can be used to distinguish the network traces of individual protocols. These keywords are passed to ensemble learning algorithm called Tri-training [127] for training. This trained model is then used for classification.

Payload Structure Based Classification: Some of the network traffic classification methods proposed in the literature show that the traffic belonging to different application layer protocols can be classified by identifying payload structure of protocols. This is because many application layer protocols follow Backus Naur Format (BNF) [93]. BNF is a meta-syntactic notation procedure used to specify the syntax of application layer protocol. BNF is used when a word and its related description is required. For example, in SMTP, a word is always mapped to a description in the format *"word: description"*. Table 2.2 shows some words and their meaning present in SMTP payload. Similar to SMTP, other application layer protocols such as HTTP and FTP follow some structure and this structural information is leveraged for traffic classification. AutoFormat [75] is one such method which assumes that each protocol and its associated fields are expressed in Backus Naur Form (BNF). AutoFormat correlates the usage of each byte in payload with its associated binary executable code for identifying header fields in the packet. Since each field content is processed differently, the fields of an application can be uniquely identified by simply looking at their usage. Once fields are identified, their combination defines the protocol as per BNF format. Ferdous et al. [47] also proposed a method to classify various types of SIP messages in VoIP traffic by performing structural analysis of SIP headers. Since SIP follows BNF format, lexicon are extracted from SIP header and passed as features to Support Vector Machine (SVM) for training. The trained model is then used for classification. The limitation of these methods is that they can identify an application layer payload only if it follows BNF structure. There are many new peer-to-peer, binary and proprietary application protocols like BitTorrent, Dropbox, etc. which do not follow BNF and hence these methods can not classify these application layer protocols. Also, these methods require reverse engineering a protocol implementation which is a time-consuming, tedious and error-prone process.

Table 2.2: Simple Mail Transfer Protocol Payload Keywords and Their Meaning

Word	**Description**
MAIL FROM	Sender's E-mail ID
RCPT-TO	Receiver's E-mail ID
DATA	Variable length data

Classification using Machine Learning Algorithms: Machine learning methods use features such as bag of words, subsequences or the payload content as a whole from different application layer protocols. During training phase, these methods create a profile of each application layer protocol using extracted features and then the generated profile is used for traffic classification during testing phase. Patrick et al. [60] proposed one machine learning based traffic classification method called Automated Construction of Application Signatures (ACAS) which extracts first 'n' bytes of payload and generates a binary feature vector of size n × 256 by setting each feature to either 0 or 1 based on whether a particular byte is present in the payload or not. This feature vector is subsequently fed to three statistical machine learning algorithms for classifying network traffic. However, in [49], authors argued that ACAS [60] is computationally expensive and can be made lightweight by changing feature set. They proposed a new method KISS which uses Stochastic Packet Inspection (SPI) for payload based traffic classification. KISS is specifically designed for identifying applications which uses UDP as layer 4 protocol. This method aims to classify the applications by analyzing application layer protocol header. Therefore, KISS first extracts few initial bytes of an application layer protocol header. The header fields of application protocol are identified by statistical characterization of byte values. This is done using Chi-Square statistical test. The byte values at different fields are passed as feature vector to Support Vector Machine (SVM) [56] for training. After training, this model is used for traffic classification. This method is designed to classify only UDP based applications and thus, can not classify TCP based applications.

2.1.2 Bit Level Payload Analysis for Traffic Classification

Network traffic classification methods which generate signatures using byte level content [76, 115, 122] do not perform well on binary protocols such as DNS, NTP, SSH and RPC. Even if some of the known byte level payload analysis based approaches such as SANTaClass [104, 105] can classify traffic belonging to binary protocols, they require extracting large amount of payload content (first 1024 bytes in case of SANTaClass). This results into two serious concerns. First, parsing such a large payload content is computationally expensive and second, it affects user's privacy. To address these issues, researchers in the networking community attempted to extract only initial few bits from the payload for traffic classification. Yuan et al. proposed one such bit level payload analysis based approach called BitMiner [124] that generates application specific signatures using bit-level data. BitMiner generates bit-level signature by learning the association between the bit value and the position of bit within the payload. Those bit values which appear more than a certain threshold number of flows are expressed using extended regular expressions as signature. The generated bit level signatures for some of the applications are shown in Table 2.3 [124]. These signatures are then used for classification purpose. In another such approach called BitsLearning [123], which is an extended idea of BitMiner [124], authors used machine learning algorithms to learn the bit values and their positions. In particular, the authors proposed two methods BitFlow and BitPack based on two types of features extracted from application flows. In BitFlow, features are extracted from first 'n' bits of each application flow and in BitPack, features are extracted from first 'm' bits of first few packets of each flow. These features are then passed to decision tree algorithm for learning which is then used for classification. Authors showed that BitPack performed slightly better than BitFlow. Since traffic classification using bit level payload analysis is a recent approach, only preliminary work has been done in this area and a comprehensive evaluation of this method is still lacking. Thus, in Chapter 4 of this thesis, we propose two bit level payload analysis based methods called *BitCoding* and *BitProb* that uses only a small number of initial bits of flows to generate signatures of different application layer protocols. We test the proposed classification method while taking 20 protocols into account and report the results.

Table 2.3: Example BitMiner Signatures [124]

Applications	**Application Signature**
Skype	^(002_0x02) + (002_0_0&002_4_1&002_5_1&002_6_0&002_7_1) ∗ $
Xunlei (Thunder)	^(001_0_0&003_0_0&003_1_1&003_2_0) ∗ $
Google Hangouts	^(000_0_1&000_1_0&001_1_1&001_3_0) ∗ $

2.2 Anomaly Detection Systems for Zero Day Attack Detection

Payload analysis is also used for detecting anomalies in network traffic [50, 68]. However, it is a challenging task to detect zero day attacks as signatures corresponding to them are not known before hand. Several schemes are proposed in the literature to detect these attacks. These detection methods have two phases of operation - training and testing phases. During training phase, a normal traffic profile is generated by extracting features such as n-grams, bag of words, sequence of words, etc. from normal traffic. The generated profile is then used to detect anomalies in the network during testing phase. We classify the previously known methods of zero day attack detection into two categories as follows:

Score Based Detection Methods: A class of anomaly detection methods generate a score from the n-grams extracted from the normal traffic. During training phase, a database of n-grams is created as part of training model and in the testing phase, n-grams extracted from the packet under consideration are compared with n-grams stored during the training phase. An anomaly score is then generated based on how many n-grams of considered packet are found in the training database. If this score is more than a predefined threshold, the packet is detected as anomalous. PAYL [114] is one such method which uses 1-gram to extract 256 features with their respective frequencies from normal traffic. Each feature corresponds to one of the 256 ASCII values. Average of all the feature vectors is called centroid and represents the normal profile of an application. In order to decide a packet as normal or malicious, authors used a simplified version of Mahalanobis distance. This measure finds the distance between feature vector of the packet under test with the centroid. Any packet having a distance larger than the predefined threshold value with respect to the centroid is labeled as anomalous. In [112], authors extended PAYL to detect worms in the network traffic. In this work, packets in incoming and outgoing traffic are correlated to detect worms as they propagate in the network which results into high similarity between a pair of an incoming packet and outgoing packet in the traffic. PAYL is used to find those packets whose 1-grams have overlapping frequencies between incoming and outgoing traffic. If such 1-grams are seen in a higher number, the packet under consideration is reported as anomalous. However, PAYL can be evaded using mimicry attacks [51] wherein an adversary can launch attack in such a way that the malicious traffic is almost similar to the normal traffic used for training purpose. To resolve this issue, Wang et al. proposed Anagram [113] which uses n-grams generated from all the packets of training dataset and stored in the space efficient bloom filters [4] and for detection, a score is assigned to each packet which is the ratio of n-grams which are not found in bloom filter to the total number of n-grams in the packet. If a packet gets score higher than a predefined threshold score, the packet is detected as anomaly. Anagram was shown to be space efficient and can thwart mimicry attacks [113]. However, recent experiments [87] have shown that it is possible for an adversary to bypass Anagram by having conversation with target for a while to discover randomization mask and subsequently use this to craft packets.

The anomaly detection methods described earlier use lower order n-grams because higher order n-grams suffer from curse of dimensionality [89]. However, it is also known that higher order n-grams show higher accuracy because they avail more number of features for anomaly detection. In this direction, Hubballi et al. [65] described a score based anomaly detection model which stores higher order n-grams from training dataset with their corresponding frequencies. Several bins of frequencies are created using a clustering algorithm and a score is assigned to each bin. Each n-gram from test packet found in a particular bin receives score of the bin. A scoring function is proposed to calculate the score of a packet which combines scores of all n-grams. If anomaly score of a packet is greater than a preset threshold, the packet is detected as anomalous. PCkAD [32] by Angiulli et al. is another such approach which uses higher order n-grams for anomaly detection. PCkAD uses the position and frequency of n-gram within payload to identify anomalous packets. During training, n-grams are extracted from the payload of packets found in normal traffic. Locations and frequency of these n-grams are used to create a normal profile. If a test packet contains too many unseen n-grams or the location of the seen n-grams differ considerably from the normal profile, the packet is considered as anomaly. To check these deviations, PCkAD uses Mahalanobis distance. One of the issue with this approach is that its dependency on location of n-gram within payload is prone to false alarms as the same n-gram can occur at different positions in the packet payload.

Zero day attacks can evade previously described n-gram analysis methods which use either binary approach to find n-grams in the database or use absolute frequency values for deriving anomaly score. However, such approaches may result into higher false detection rates. Thus, in chapter 5 of this thesis, we propose a zero day attack detection approach called *Rangegram* that considers the minimum and maximum appearance frequency of a particular n-gram generated from a packet in training dataset and finds deviations from this range to detect anomalies.

Machine Learning based Methods: There are several methods in the literature that use machine learning algorithms to detect zero day attacks. Due to the scarcity of labeled datasets and skewed distribution of normal and malicious content in network traffic, many of these methods use one class classification methods. In these methods, features are extracted from the normal application payload and provided as input to one class classification algorithms for training purpose. After this, the trained profile is used to detect anomalous packets/flow in network traffic during testing phase. In [89, 90], authors proposed a detection method McPAD that uses multiple one class Support Vector Machines (SVMs) to detect anomalous packets by majority voting. During training phase, McPAD uses modified form of n-grams called n_v grams which are substrings of a string in which each substring of length ‘n’ is separated from other substring of length ‘n’ by length ‘v’. By varying parameter ‘v’, each payload gets represented in different feature space. For each value of ‘v’, the generated features are passed into a different one class SVM for training. During testing, n_v grams generated from packet under consideration are passed to the trained SVM according to the value of ‘v’. Majority of votes from different SVMs decide whether the packet is normal or anomalous. However, the major drawback of this method is that it requires training of multiple one class SVMs. In [34], authors proposed HMMPayl that uses multiple hidden markov models for classification. Görnitz et al. [57] suggests a much larger performance improvement with a novel machine learning algorithm based on unsupervised learning method. Similarly authors of [83] experimented with different distance measures and used χ^2 hypothesis testing method [121] to report improved performance over [34].

Similar to the previously known score based anomaly detection methods, the machine learning based approaches described above also rely on the presence or absence of n-grams in the database generated during training phase to detect anomalies in the traffic. As a result, these methods are also prone to false alarms. Thus, in Chapter 5 of this thesis, we propose a method called *OCPAD* wherein we adapt one class Multinomial Naive Bayes classifier for the purpose of anomaly detection.

2.3 VoIP Spam Detection

VoIP spam detection methods proposed in the literature analyze payload of packets belonging to signaling protocols such as Session Initiation Protocol (SIP) [94], H.323 [77], etc. These spam detection methods extract call features like call duration, time of call, caller-callee pair, call frequencies, etc. to detect the VoIP spam. In this section, we discuss previously known methods that analyze signaling protocol payload to detect VoIP spam.

Analyzing Signaling Protocol Payload: Signaling protocols use various types of messages to manage VoIP calls. For example, SIP, which is a text based light-weight signaling protocol, makes use of INVITE and BYE messages for call establishment and call disconnection respectively. Various call parameters such as call-id, caller-id, callee-id, call duration, etc. can be extracted by processing SIP header. SIP is considered as most popular signaling protocol [70, 97] and thus, most of the VoIP spam detection methods are proposed in the literature considering SIP as the signaling protocol. We discuss these VoIP spam detection methods in the next paragraph.

Authors of [62, 67] proposed methods which maintain different lists of users such as *white list* and *black list* and calls are allowed or rejected based on these lists. In white list, subscriber (callee) stores only those caller-ids which are allowed to make a call to the callee where as in black list, subscriber stores those caller-ids which are blacklisted by the callee and are not allowed to make a call to it. These blacklisted caller-ids are considered as potential spam call generators. These lists are maintained and updated manually which is tedious, error prone and also difficult to update frequently. Thus, a determined spam user can make use of a new caller-id each time it calls a particular callee. To overcome this issue, Hansen et al. [62] introduced the concept of maintaining a *grey list* whose behaviour depends on the circumstances of the calls. A caller receives a busy signal on its first call attempt and the phone of the callee does not ring. However, if the caller is reattempting to reach the callee, the caller is allowed to get connected to the callee as it is likely that a normal user (human) is actually interested in talking to user on the opposite side. However, this may result into blocking of important calls also. Another limitation of this method is that since these lists are populated with entries manually, it is difficult to maintain them.

Along with the list based approaches, few works in the literature proposed methods that prevent spam calls in a VoIP network by analyzing calling patterns of VoIP users and subsequently use it for allowing or disallowing calls. A call is allowed if the current call pattern is similar to the previous call pattern of a caller. However, if the current call pattern is deviated from the last one, the call is blocked. Features like call frequency and average call duration are used to compare previous call with the current one [99]. In another work, Sengar et al. [98] used parameters such as time and day of calling and the call duration to create a pattern and then used Mahalanobis Distance [64] as a metric to differentiate the current call pattern from the previous one in order to detect spam calls. Few other approaches in the literature extract features such as call duration and frequency of calls and provide it as input to various machine learning algorithms for training purpose. Once the detection model is trained, it is used for detecting spam callers in VoIP networks. In [37], authors used three call parameters -

call frequency between a caller and a callee, call duration between a caller and a callee and number of outgoing partners associated with a caller - to calculate a trust score for a pair of caller and callee. Using this trust score, a global reputation is calculated for each caller. Based on this global reputation, clusters are generated using K-means clustering algorithm. After this, a predefined threshold is used to separate the cluster containing spam users from the clusters containing normal users. In another work [119], Wu et al. proposed an approach that uses a modified version of MPCK-Means [63] to detect spam calls. Similarly, Toyoda and Sasase in [107] proposed an approach that extracts different set of features from the VoIP traffic and feed it to random forest algorithm to detect spam calls.

A class of works in the literature also proposed reputation based approaches wherein *buddy lists* are used to collect user ratings and these ratings are then used to generate reputation scores. The generated scores decide the acceptance or rejection of a call from a caller. Various parameters such as social network linkage between users and feedback from users are used to generate scores. These schemes require an appropriate reputation threshold to detect spam calls. If the reputation of a user is less than the predefined threshold, a user is considered as spam user. In this direction, Dantu and Kolan [45, 73] use trust, reputation and friendship of the calling party and recipient's mood to generate a score. In [39], authors proposed a reputation based approach in which reputation of a caller is built on the basis of its call duration with a callee. Authors introduced a parameter named as call credential which decides the level of trust for a user. Call credential is the ratio of average call duration between a pair of caller and callee to the overall average call duration of the caller. Authors also designed an algorithm called CallRank which manages the call credential of users and changes the values calls after calls. This algorithm declares a user as spam user if its call credential goes below a predefined threshold. The drawback of this approach is that it is dependent on only one call parameter (i.e. call duration) for detecting spam users. Azad et al. [36] also proposed a reputation based approach to detect VoIP spam but used more than one call parameter in their work. Authors used call duration, interaction rate, and caller out-degree distribution to establish a trust network between VoIP users across the network. These call parameters are used to calculate a score called Direct trust score for a pair of users (caller to callee). This score is then used to calculate reputation of a caller by normalizing trust scores of all the users in its database. A threshold reputation is then decided using reputation of normal users. If the reputation of a particular user goes below a predefined threshold, s/he is considered as a spam user. This approach is tested by generating artificial calls from a set of users designated as spam users. However, authors of [36] mentioned that if a community of VoIP users contains spam users less than 10% of the total users, the proposed approach may not be able to detect them.

Most of the previously proposed reputation based approaches [106, 107] use average values of features such as number of calls, call duration, etc. due to which they may fail to detect spam users if a determined spam user maintains these averages using artificial calls. For example, to maintain call duration average, a spam user can generate some calls of large duration to other spam users and hence avoid detection. To overcome this issue, in chapter 6 of this thesis, we propose *SpamDetector*, a graph based anomaly detection approach, that can identify such determined spam users too by taking the one-to-one neighbourhood relationship (in a graph) into account and by identifying nodes which have different behavioural patterns compared to its neighbour nodes.

2.4 Conclusion

With this literature review, we attempted to highlight the gaps that exist in payload analysis based methods for three different purposes - network traffic classification, zero day attack detection and VoIP spam detection. We discussed how previously known traffic classification methods suffer from drawbacks such as high computational complexity, false classification and hindering users' privacy. Similarly, the known zero day attack detection methods are also prone to generate false detection results either due to binary comparison or mimicry attacks. We also discussed how existing VoIP spam detection methods can be evaded by a determined spam user. Thus, it is evident from the existing literature that there is need of developing payload analysis based traffic classification methods which have low computational overhead and can identify applications accurately. Also, there is a need for developing anomaly detection methods which can detect anomalies in the network traffic accurately with false detection rates as low as possible.

Taking motivation from the existing research gap in the literature, we present different traffic classification and anomaly detection methods in the subsequent chapters of this thesis.

Chapter 3

Byte Level Payload Analysis for Network Traffic Classification

3.1 Introduction

Network traffic classification is performed for various network management purposes such as defining security policies and implementing QoS rules. For accurate classification, well-defined signatures for different protocols/applications must be known before hand. New application layer protocols require new signatures so that signature database is up-to-date and coverage is comprehensive. However, a report [100] suggests that nearly 50% of Internet traffic belong to unknown or proprietary applications whose details of design are not made public. This results into lack of availability of signatures for these applications which makes the traffic classification difficult. Also, it is tedious to manually generate signatures for all applications given large number of applications [104]. This strongly advocates the need for a traffic classification method which can automatically extract features unique for each application and generates signature using the extracted features. Few works [105, 125] recently proposed in the literature generate signatures of different applications using a set of potential keywords and their positions in the packet payload. These signatures are subsequently used for traffic classification. However, as discussed in Section 3.2.1 of this chapter, simply using potential keywords and their ordering in a flow may lead to misclassifications. Taking motivation from this, in this chapter, we describe a new traffic classification method called *RDClass* in which we first extract keywords automatically and subsequently measure the relative distance between the extracted keywords to form more accurate protocol identities. The idea has its motivation in the fact that, each application has keywords in certain specific portion of the payload. We show that the use of relative distance of keywords increases the accuracy of classification compared to other similar methods [105, 125] which simply use ordering of keywords. Our contributions in this chapter are as follows.

1. We describe a robust payload based traffic classifier called *RDClass* for application layer protocol detection. *RDClass* is completely automated and can generate unique keywords for protocol identification when presented with payloads of different application.

2. We define a new state transition machine $RDCCA$ which uses both keywords and their relative distance to generate signature for each application.

3. We evaluate the classification accuracy of *RDClass* by testing it against applications which use either TCP or UDP as transport layer protocols.

4. We perform experiments using different applications and compare *RDClass*'s performance with other related methods and show that the relative distance measurement minimizes the misclassifications.

Rest of this chapter is organized as follows. In Section 3.2, we describe the motivation and working of proposed traffic classifier. In Section 3.3, we describe the experiments performed to evaluate the classification accuracy of the proposed classifier. Finally the chapter is concluded in Section 3.4.

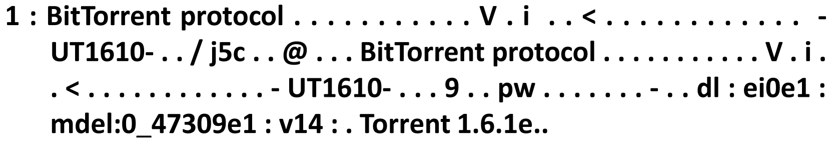
3.2 Proposed Traffic Classifier

Several rule-based techniques [84, 104, 120, 122] have been proposed in the literature to classify network traffic. These techniques generate rules for each application from keywords frequently occurring in the application flows. In this section, we first present the motivation behind proposing *RDClass* in terms of limitations of the previously known rule-based traffic classification techniques and subsequently describe the working of *RDClass*.

3.2.1 Motivation

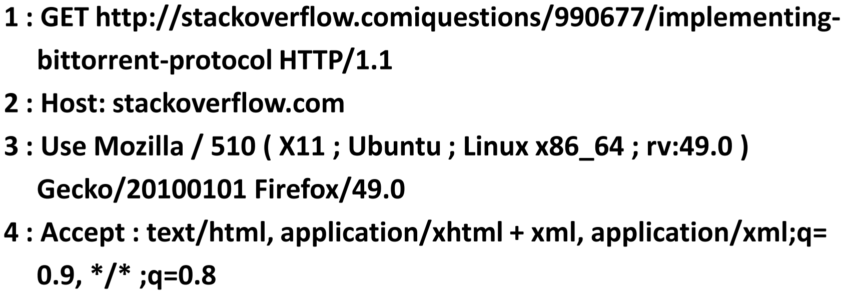
As described earlier in Section 3.1, several network classification techniques in the literature [105, 125] generate an ordered list from extracted keywords of payload and use it to generate application signature.

1. **Scenario 1:** Consider two sample payloads generated from flows of BitTorrent protocol and HTTP as shown in Figures 3.1 and 3.2 respectively. To identify BitTorrent protocol, techniques described in [105, 125] generate two keywords namely "BitTorrent" and "protocol" and search for these keywords (appearing in order) in any given payload. However, as we can see from both the figures, these two keywords appear in the same order in both BitTorrent and HTTP payload



```
1 : BitTorrent protocol . . . . . . . . . . . V . i  . . < . . . . . . . . . . . .  -
    UT1610- . . / j5c . . @ . . . BitTorrent protocol . . . . . . . . . . . V . i .
    . < . . . . . . . . . . . . - UT1610- . . . 9 . . pw . . . . . . . - . . dl : ei0e1 :
    mdel:0_47309e1 : v14 : . Torrent 1.6.1e..
```

Figure 3.1: BitTorrent Payload



```
1 : GET http://stackoverflow.comiquestions/990677/implementing-
    bittorrent-protocol HTTP/1.1
2 : Host: stackoverflow.com
3 : Use Mozilla / 510 ( X11 ; Ubuntu ; Linux x86_64 ; rv:49.0 )
    Gecko/20100101 Firefox/49.0
4 : Accept : text/html, application/xhtml + xml, application/xml;q=
    0.9, */* ;q=0.8
5 : Accept-Language: en-US, en;q=0.5
```

Figure 3.2: Hyper Text Transfer Protocol Payload

as a result there is a chance that HTTP packet is identified as belonging to BitTorrent protocol.

2. **Scenario 2:** Consider two sample payloads generated from SMB and FTP protocols as shown in Figure 3.3 and Figure 3.4. An SMB protocol signature contains keywords “SMBr, SMBs and NTLMSSP” (shown in bold fonts in the figure) in order. This can lead to misclassification with the sample FTP flow shown in Figure 3.4 as it contains all the keywords of SMB protocol in the same order. These keywords within FTP flow originate from a file content which is transferred using FTP protocol.

```
1 : SMBr..C..PC NETWORK PROGRAM 1.0.
    MICROSOFT NETWORKS 1.03..MICROSOFT NETWORKS 3.0..
    LANMAN1.0..Windows for Workgroups 3.1a..LM1.2X002..
    DOS LANMAN2.1..LANMAN2.1..Samba..NT LANMAN 1.0..NT LM 0.12......
    SMBs....................Cy..........02.......J..........`H..+......>0<..0..
    .NTLMSSP........`.... .......%...VNET3BLU.U.n.i.x...S.a.m.b.a
```

Figure 3.3: SMB Payload

```
 1 : 220 Domain FTP Server Ready
 2 : USER anonymous
 3 : 331 Anonymous login ok, send your complete email address as your password.
 4 : PASS <password>
 5 : 230 Anonymous access granted, restrictions apply.
 6 : LIST
 7 : 150 Opening data channel for directory list.
 8 : 226 Transfer successful.
 9 : PASV
10 : 227 Entering Passive Mode (10,203,4,50,4,29)
11 : SIZE /../SMB-Logs/Requesters/SMBR/Server/SMBs/NTLMSSP.txt
12 : 213 0
```

Figure 3.4: FTP Payload with SMB Keywords

To prevent such misclassifications, we use relative distance between selected keywords. For example, in a BitTorrent protocol payload, the terms “BitTorrent” and “protocol” appear at the beginning of payload but in HTTP payload, they can appear at any arbitrary position and the same applies to SMB and FTP protocols too. Thus, by incorporating the ordering and relative distance between the keywords, we design a new state transition machine to classify the network traffic more accurately. In the next few subsections, we describe the working of our proposed classification technique *RDClass* and how it generates the state transition machines for different applications.

3.2.2 *RDClass* Working

RDClass has 3 main components - i) Flow Reconstruction, ii) Term Extraction and iii) State Transition Machine for Flow Classification. These components are shown in Figure 3.5 and working of these components is elaborated below:

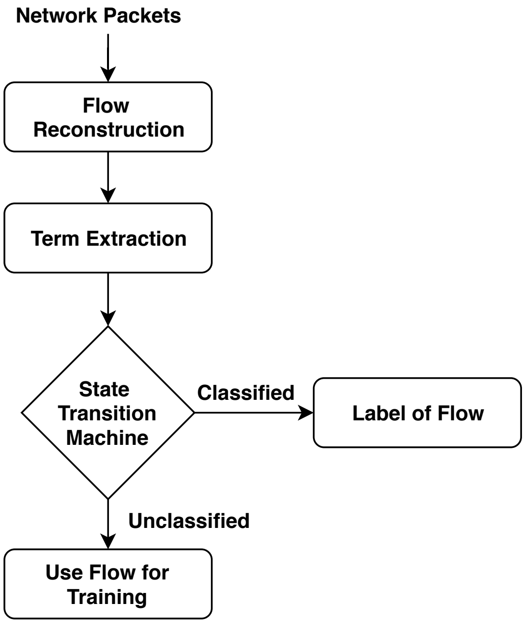


Figure 3.5: *RDClass* Components

1. Flow Reconstruction: A network flow is a series of packets exchanged between two hosts in their sequence of generation. These hosts are identified by two unique IP addresses. A flow is uniquely identified by the 5 tuple as *SrcIP, DstIP, SrcPort, DstPort, Protocol.* These terms carry the following meaning:
All the packets with complete payload from the identified TCP/UDP flow are taken for the subsequent stage of signature generation. A TCP flow starts with a three-way

$SrcIP$	Source IP address of host
$DstIP$	Destination IP address of host
$SrcPort$	Source Port number
$DstPort$	Destination Port numbers and
$Protocol$	Layer 4 Protocol (TCP/UDP)

handshake and results in a bidirectional flow of packets between two hosts (A and B) as shown below.

$(A) \dashrightarrow [SYN] \dashrightarrow (B)$

$(A) \dashleftarrow [SYN/ACK] \dashleftarrow (B)$

$(A) \dashrightarrow [ACK] \dashrightarrow (B)$

To terminate the established TCP connection, the communicating hosts exchange FIN packets. Thus, the packets exchanged after connection establishment but before its termination belong to one TCP flow. In case of a UDP flow, all packets sharing the same source IP address, source port, destination IP address and destination port having an inter packet arrival timing less than δ are considered as part of one flow. The threshold δ is a user defined threshold and is fixed suitably.

2. Term Extraction: All the packets of a flow are taken and payload portion of every packet is extracted and concatenated to get a flow payload which is then parsed to generate terms. *RDClass* uses newline character, white space and special character as delimiter for parsing. For example if a flow f_i has payload content "This is my first payload", the set of terms generated from this are {This, is, my, first, payload}.

3. Processing Terms with State Transition Machine: The set of terms generated by term extractor is given as input to a set of state transition machines (there may be one or more state transition machines for every application) where each machine has transitions defined with the frequently occurring terms (frequently occurring byte sequences or keywords) of an application. Each such state transition machine makes allowed transitions after reading terms generated from payload and a flow is classified (labeled) if any of the state transition machine reaches to an accepting state. If none of the state transition machines are able to reach an accepting state, the flow is not classified. We provide the formal definition of the state transition machine and its working in the next subsection.

3.2.3 State Transition Machine for Flow Classification

We propose a new state transition machine called "*Relative Distance Constrained Counting Automata*" ($RDCCA$) which acts as a signature for the application and hence subsequently can be used to classify applications. $RDCCA$ has the ability to see specific terms in a certain order and enforce constraints on their relative distance. Relative distance and ordering of keywords is enforced by $RDCCA$ with a counter at each state. The machine revisits a state if a non keyword term is read. It also has the ability to count the number of times a state has been revisited and validate a transition only if counter value is in a defined range. The counting ability of the machine is required to see how many other terms appear in between two selected terms (which are found to be invariant across flows of an application). $RDCCA$ is formally defined using six tuple as $\mathcal{M} = (Q, \Sigma, C, \delta, q_0, F)$ where

Q : A finite set of states

$q_0 \in Q$: An initial state

Σ : A finite set of input symbols

C : A finite set of Counters with each $c_i \in C$ taking values in $\mathcal{N} \bigcup 0$

$F \subseteq Q$: Is a set of final states

δ : A set of transitions defined as $\delta : Q \times C \times \Sigma \rightarrow Q \times (C \rightarrow C)$

where each transition $\delta_i \in \delta$ is defined using six tuple as $\langle q_i, q_j, c, \sigma, \phi(c_i), Inc(c_j) \rangle$ with the elements representing

q_i : Current state

q_j : Next state

c_i : Counter value at the state q_i

$\sigma \in \Sigma$: Input symbol

$\phi(c)$: Is a constraint on counter value c_i at state q_i on this transition

$Inc(c_j)$: Is a function which initialize the counter in the next state q_j to a new value

For our implementation, we use Σ as set of terms possible in any payload. It can be noticed that each term is a combination of alphanumeric characters and each term is of variable length which results into an infinite possible terms. However for the machine, alphabet set needs to be finite. We address this by calculating the length of largest term appeared in the training set of that application flow and including all combination of alphanumeric characters upto that length in Σ.

$RDCCA$ accepts a set of strings (after reading terms from payload) starting from start state to a final state. The set of strings accepted by the $RDCCA$ is the language generated by it and is denoted as $\mathcal{L} \subseteq \Sigma^*$. The string acceptance can be interpreted as a recursive process. For example, the string with $S = \sigma_1, \sigma_2, \cdots, \sigma_n$ is read by machine with a set of transitions $\delta^*(q_0, \sigma_1, \sigma_2, \cdots, \sigma_n)$ where δ^* denote the extended transition function which is defined recursively as $\delta(\delta^*(q_0, \sigma_1, \sigma_2, \cdots, \sigma_{n-1}), \sigma_n)$. Each payload is processed and terms are read to check whether it contains a valid string that $RDCCA$ accepts. An example $RDCCA$ generated (how to generate application specific $RDCCA$ is described in next subsection) to identify BitTorrent protocol is shown in Figure 3.6. In this $RDCCA$, there are three states (q_0, q_1, q_2) and two transitions as shown in Table 3.1. In the beginning, the machine starts at state q_0 and initializes counter c_0 at q_0 to 0. The first transition is from state q_0 to q_1 and this

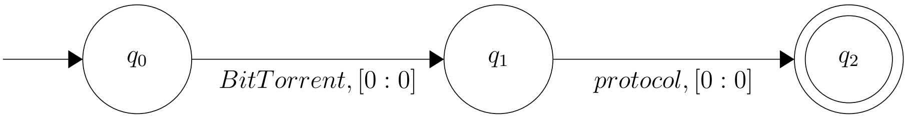


Figure 3.6: $RDCCA$ Generated for BitTorrent Protocol

Table 3.1: Transitions of BitTorrent $RDCCA$

Current State	Next State	Input Symbol	Counter Constraint
q_0	q_1	BitTorrent	[0:0]
q_1	q_2	protocol	[0:0]

transition occurs on reading term "BitTorrent" and there is a constraint on minimum and maximum counter value of c_0 at state q_0 which in this case is [0:0]. A counter value of 0 indicates the state q_0 is not revisited and in turn implies that there are no other terms preceding the term "BitTorrent" in the flow and it is the very first term to appear in the payload of a flow. When transition δ_1 fires, it sets the counter value c_1 at q_1 to 0. Second transition is from the state q_1 to q_2 and has symbol of "protocol". This keyword also has a constraint of [0:0] which means that there are no revisits at state q_1 also and the keyword "protocol" immediately follows the keyword "BitTorrent". Often *RDCCA* may have self loops at states which are indicated with wildcard "*" as input symbol and - symbols for constraints. The wild-card "*" in this case takes a slightly different meaning of matching any symbol (term) in payload except those which have an explicit labeled outgoing edge from that state. The "-" symbol means there is no constraint on the transition.

3.2.4 *RDCCA* Generation from Unknown Flows

As discussed in previous subsection, *RDClass* generates *RDCCA* from a set of unlabeled network flows. In this subsection, we describe how *RDClass* automatically generates new set of *RDCCA* instances for classification of applications. Following are the important considerations for design of *RDClass* while generating machine instances.

1. Input to this phase is extracted terms of unlabeled flows.
2. Keywords are selected by processing only initial few bytes of payload which pertain to mostly application header, it contains terms which appear in majority of the flows. Such frequently appearing terms are used to generate application specific state machine instances.

The architectural components of *RDClass* state transition machine generation phase is shown in Figure 3.7. It has five components as described below.

1. Flow Binning: We can recall that *RDClass* learns and extracts right key-

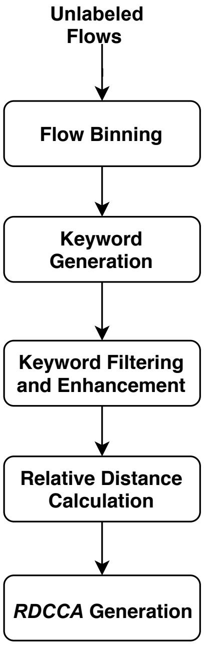


Figure 3.7: $RDCCA$ Generation from Unknown Flows

words from unknown flows. Thus any flow which is not classified by classifier in the previous step is passed to binning component which groups similar flows together. There will be many bins which represent unknown application flows and the idea of collecting sufficient number of flows is to identify right keywords which are invariant across the flows. This step is critical for extracting right set of keywords and hence generating a state transition machine with right sequence of keywords and constraints. If clustering is inefficient, it results into clusters containing flows of multiple protocols which eventually leads to poor traffic classification. To cluster payloads, this component calculates a similarity score between payloads. In order to calculate the similarity between payloads of two flows, the terms of two payloads are compared with each other. Similarity between two flows f_i and f_j is calculated using Equation 3.1.

$$Sim_{ij} = \frac{|f_i \bigcap f_j| \times 2}{|f_i| + |f_j|} \times 100 \tag{3.1}$$

This equation calculates the ratio of common terms in two flows and sum of the total terms in both the flows. Let us begin with the first training flow f_1 and set a threshold value of flow similarity for binning as 70%. Let P_1 be the payload in the first flow f_1 with content as “This is my first payload”. The terms set of P_1 contains alphanumeric words extracted from P_1 and can be represented as $term_{P_1} = \{This, is, my, first, payload\}$. Since P_1 is the first payload, it is stored in a new partial bin say B_1. Now let P_2 be the payload corresponding to the second flow f_2 with content as “This is my second payload”. The terms set of P_2 also contains alphanumeric words extracted from P_2 and can be represented as $term_{P_2} = \{This, is, my, second, payload\}$. The similarity score between f_1 and f_2 is calculated from Equation 3.1 using $term_{P_1}$ and $term_{P_2}$. As there are 4 common terms and total 10 terms (5 in each flow), the similarity score between f_1 and f_2 is calculated as 80% (((4×2)/10)×100). As this value is greater than the predefined threshold value (70%), P_2 is stored in the same partial bin B_1 with payload P_1. Let P_3 be the payload corresponding to the third flow f_3 with content as “This is your third payload”. The term set of P_3 can be represented as $term_{P_3} = \{This, is, your, third, payload\}$. The similarity score between f_3 and each payload in partial bin B_1 is now calculated. The similarity scores between f_1 and f_3 and f_2 and f_3 are calculated as 60% each. Since these values are below the predefined threshold value, P_3 is not stored in the partial bin B_1 but stored in a new partial bin, say B_2. This clustering step continues until sufficient number of flows are there in a bin to extract meaningful set of keywords. Thus, there are many incomplete clusters/bins generated during this phase when it accumulates unknown flows. Thus for every new flow f_n, similarity between f_n and all existing Partial Bins (PBs) is calculated and f_n is put into that partial cluster with which it has maximum similarity (average of similarities between every flow in that bin and the flow under consideration). This clustering step is represented in Algorithm 3.1. This algorithm takes flows along with two thresholds as input. One threshold is for minimum similarity between bin and a flow to be included in the bin and second is a threshold on minimum number of flows required in a bin to forward the flows of a bin to keyword generation component. For each new flow received, it calculates the similarity with each of the partially clustered flows and adds flow into that bin with which it shares maximum similarity and this is done only if this similarity is greater than a predefined threshold. Otherwise, a new partial bin (an entry in PB) is created and maintained. After successful addition of

Algorithm 3.1 Binning Flows

```
Input: A set of unclassified flows $f_1, f_2, \cdots$
Input: $Sim_{threshold}$ - Minimum similarity threshold between flow and bin
Input: $Bin_{threshold}$ - Minimum number of flows in a bin for keyword extraction
Output: A set of clusters/bins of flows
while ! interrupted do
    $f_{new} \leftarrow$ Receive a new unclassified flow from classifier
    $PB \leftarrow$ List of Partial Bins
    for all $b_i \in PB$ do
        Calculate $Sim_{nb_i}$
    end for
    $Sim_{max} \leftarrow$ Maximum similarity combination $Sim_{nb_j}$
    if $Sim_{max} \geq Sim_{threshold}$ then
        Add $f_n$ to bin $b_j$
        $flow_{count} \leftarrow$ GetFlowCount($b_j$)
        if $flow_{count} \geq Bin_{threshold}$ then
            Forward bin $b_j$ to Keyword Generation component
            Remove $b_j$ from PB
        end if
    else
        Create a new bin $b_{new}$ with $f_n$ and add it to $PB$
    end if
end while
```

flow into a bin, it checks if the number of flows in the bin has crossed the minimum number of flows and based on this, it either forwards flows of bin to keyword generation component or keeps it in PB list.

It is worth noting that, computing similarity between two whole payloads is computationally expensive as packets can be of arbitrarily large size[1] and moreover, we need to compute similarity between flows instead of packets. This makes it computationally even more prohibitive. To address this issue, several previous works [104, 105] proposed approaches wherein first few bytes of flows were used to estimate the similarity. These works also showed that processing first 1024 bytes of a flow is good enough to find similarity[2]. We also confirm this observation in our experiments.

2. Keyword Generation: Keyword generation takes flow cluster (in the form of extracted terms) generated in the binning phase as an input to generate keywords. To identify keywords from set of terms, the probability of occurrence of each term is calculated as in Equation 3.2. All terms whose probability is greater than certain threshold are considered as keywords. For example, if payload of a flow f_i is “Hello

[1] Ethernet network has an MTU of 1500 bytes, thus each packet can have approximately 1500 bytes

[2] If the payload length of a flow is less than 1024 bytes, whole payload is considered

World..!! How @re you 123." then the term set generated is { Hello, World, How, re, you, 123 } and if a second flow f_j="Hello World" then term set of this flow is { Hello, World}. Considering only these two flows, the probability of term "Hello" is 100% and that of "you" is 50%.

$$P(term_i) = \frac{\text{Number of flows having } term_i}{\text{Total number of flows}} \tag{3.2}$$

3. Keyword Filtering and Enhancement: In the keyword generation phase discussed previously, only the probability of occurrence of a term is used for selecting a keyword. In some cases although a term is having high probability, it may not be a suitable candidate for keyword. For example, terms which denote time, machine names, software name, etc do not make good keywords. Thus, these ill suited keywords are removed in this keyword filtering and enhancement phase. This is necessary because if these dependencies change over time, the state transition machine fails to detect applications correctly. In our implementation, we used a dictionary of "bad words" to filter such ill-suited terms. Following are some of the criteria we used for filtering keywords:

(i) Removing pure numeric terms: All keywords which contain only numbers such as "001", "702", etc are removed.

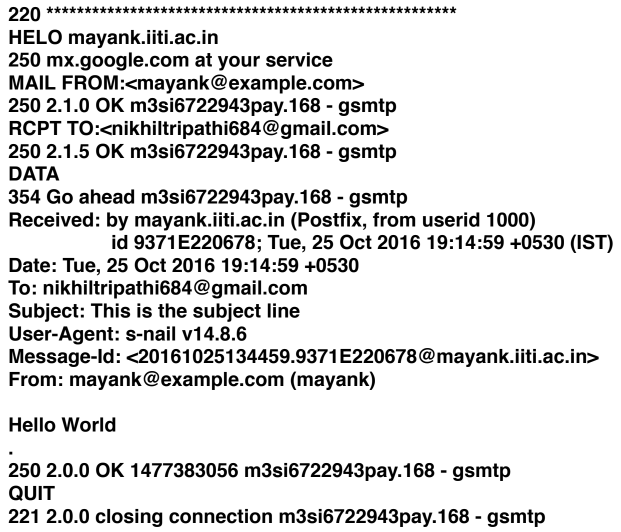
(ii) Removing short terms: All keywords which contain only lower case letters, having length less than 3 and does not form strong adjacency pair with any other keyword are removed. Strong adjacency here indicates those keyword pairs which are always adjacent to each other in all payloads.

(iii) Removing time dependent words: Time dependent terms include name of days ("Sunday","sun", "Monday", "mon", etc.), months ("january", "jan", "february", "feb", etc.) and other time dependent words (hrs, min, sec, hours, minutes, seconds, etc.). These may change over a period of time and hence do not qualify to be good keywords. Hence these terms are also filtered from the keyword set.

(iv) Removing application dependent words: Words like "Mozilla", "Explorer", "Asterisk" or any word describing name of software. It also contains names of operating systems like "Windows", "Ubuntu", "Kali", etc. and architectures "x64"

and “x86” also do not make good candidate for keywords and these are also filtered. *(v) Adding mutually exclusive terms: RDClass* selects keywords on the basis of occurrence probability. The drawback of $RDCCA$ is that it eliminate some of the important words. For example, in HTTP protocol, GET, PUT, POST, DELETE, etc. methods are mutually exclusive. It is also seen that majority of flows contains GET keyword whereas other terms like PUT or POST occur less frequently. Therefore, they never get converted from term to a keyword. Therefore, these terms are added manually in the keyword set.

4. Relative Distance Calculation: One of our main contribution in this chapter is the use of relative distance between keywords along with their ordering. Relative distance of two selected keywords k_i and k_j is the number of other terms appearing in the payload between these two keywords. Consider a sample payload shown in Figure 3.8. This payload is of Simple Mail Transfer Protocol (SMTP) used for email communication. As usual, terms are generated by parsing this payload with white space and special character as delimiters. The set of terms generated from the first 3 lines of payload are {HELO, mayank.iiti.ac.in, 250, mx.google.com, at, your, service, MAIL, FROM, mayank, example, .com}. If HELO and MAIL are two



```
220 ******************************************************
HELO mayank.iiti.ac.in
250 mx.google.com at your service
MAIL FROM:<mayank@example.com>
250 2.1.0 OK m3si6722943pay.168 - gsmtp
RCPT TO:<nikhiltripathi684@gmail.com>
250 2.1.5 OK m3si6722943pay.168 - gsmtp
DATA
354 Go ahead m3si6722943pay.168 - gsmtp
Received: by mayank.iiti.ac.in (Postfix, from userid 1000)
              id 9371E220678; Tue, 25 Oct 2016 19:14:59 +0530 (IST)
Date: Tue, 25 Oct 2016 19:14:59 +0530
To: nikhiltripathi684@gmail.com
Subject: This is the subject line
User-Agent: s-nail v14.8.6
Message-Id: <20161025134459.9371E220678@mayank.iiti.ac.in>
From: mayank@example.com (mayank)

Hello World
.
250 2.0.0 OK 1477383056 m3si6722943pay.168 - gsmtp
QUIT
221 2.0.0 closing connection m3si6722943pay.168 - gsmtp
```

Figure 3.8: Simple Mail Transfer Protocol Payload

selected keywords then the relative distance between these two is 6.

Relative distance calculation component takes filtered set of keywords as input and finds a range of relative distance values. For example, if there are 20 flows within a cluster/bin, for a particular pair of ordered keywords k_i and k_j the relative distance is calculated for each flow and the minimum and maximum value of these distances are subsequently used as permissible range of separation between k_i and k_j. Output of this component is a list of filtered keywords and their permissible relative distances. If the filtered keyword set is K =$\{k_1, k_2, \cdots k_n\}$ and this set is ordered, their relative distances set is R_{dist}=$\{[d^{min}_{k0,k1}, d^{max}_{k0,k1}], [d^{min}_{k1,k_2}, d^{max}_{k1,k_2}], \cdots, [d^{min}_{k_{n-1},k_n}, d^{max}_{k_{n-1},k_n}]\}$
For example, consider 3 SMTP flows which are having keywords as HELO, MAIL, FROM, RCPT and TO and the relative distance between first term of payload (say B) and HELO is calculated as 2, 6 and 10 in the three flows. Similarly, the relative distance between HELO and MAIL is calculated as 3, 6 and 4, relative distance between MAIL and FROM is calculated as 0, 0 and 0, relative distance between FROM and RCPT is calculated as 5, 6 and 4 and relative distance between RCPT and TO is calculated as 0, 0 and 0 in the three flows. This set is ordered and their relative distances set is R_{dist}=$\{[2_{B,HELO}, 10_{B,HELO}]$,$[3_{HELO,MAIL}, 6_{HELO,MAIL}]$,$[0_{MAIL,FROM}, 0_{MAIL,FROM}]$, $[4_{FROM,RCPT}, 6_{FROM,RCPT}]$,$[0_{RCPT,TO}, 0_{RCPT,TO}]\}$

5. *RDCCA* Generation: As described previously, we generate a novel state transition machine which enforces constraints on ordering and relative distance between selected keywords. This machine when fully constructed and put into use reads payload content from incoming flows and labels them if it is able to transition from start state to a final state.

To construct $RDCCA$, the ordered pairs and their relative distances are received from previous phase and a state transition machine is generated with $n + 1$ states for a set of keywords of size n. The state q_0 is denoted as start state. The $(n + 1^{th})$ state q_n (if all keywords in the list are ordered in the sequence in which they are appearing in the payload) is marked as final state. The keyword k_i is the input symbol for transition between q_i and its next state q_{i+1} and this transition will have counter constraint set to $[d^{min}_{k_i,k_{i+1}}, d^{max}_{k_i,k_{i+1}}]$. Considering the example given in previous step, we have 5 keywords and thus $RDCCA$ in this case has 6 states. If the keyword HELO is the input symbol for transition between q_0 and its next state q_1 as shown in Figure 3.9 then this transition will have counter constraint set to $[2_{B,HELO}, 10_{B,HELO}]$.

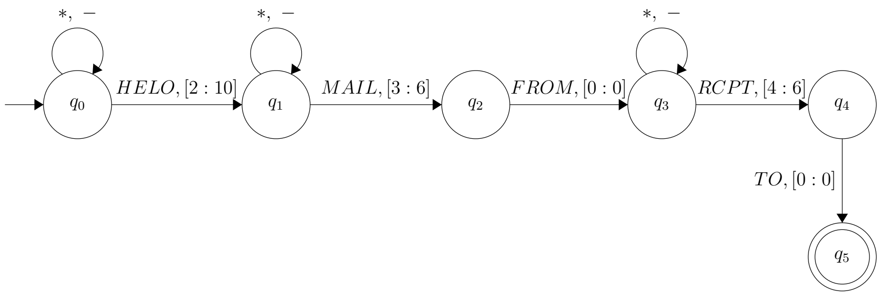


Figure 3.9: Sample $RDCCA$ for Simple Mail Transfer Protocol

3.2.5 Complexity Analysis

As described previously in Section 3.2.4, computing similarity between two payloads is computationally expensive. Thus, in this subsection, we describe the asymptotic complexity of each component of *RDClass* model. Table 3.2 shows the complexity of each component of our proposed model indicating which steps are done in training (offline) mode and testing (online) mode.

Table 3.2: Component-wise Complexity of *RDClass*

Component Name	**Complexity**	**Online/Offline**	**Explanation**
Flow Reconstruction	O($l \times p$)	Both	l= number of flows, p = number of packets in the trace.
Term Extraction	O(j)	Both	j = total number of bytes in the payload of test flow
Flow Classifier	O($m \times t$)	Online	m= number of state transition machines and t = number of terms in the payload of test flow
Flow Binning	O($t\ log\ t$)	Offline	n = number of terms in the flow.
Keyword Generation	O(f×t)	Offline	f = number of flows in bin. t = number of terms in flow.
Keyword Filtering and Enhancement	O($k\ log\ k$)	Offline	k = number of words in the set of keywords
Relative Distance Calculation	O(t)	Offline	t = number of terms in the flow
$RDCCA$ Generation	O(1)	Offline	Relative distance and ordering of keywords is found in previous step. Just convert this into a graph

1. Flow Reconstruction: Flow reconstruction inspects each packet header and adds it into appropriate flows. If there are l flows and p packets then it has a complexity of O($l \times p$).

2. Term Extraction: Term extraction reads entire payload once and identifies separator (delimiter) to parse the payload and scanning payload once it is sufficient for this operation, hence its complexity is O(j) where j is length of payload.

3. Flow Classifier Flow classifier is a state transition machine and it has to match every term with every possible instance of state transition machine. In the worst case, it has a complexity of O($m \times t$) where m is number of machine instances and t is number of terms in a flow.

4. Flow Binning: Flow binning component needs to find the intersection of number of terms in two flows which can be implemented by sorting the two sets separately which involves a complexity of O($t\ log\ t$) + O($t\ log\ t$) for flow 1 and flow 2 each containing t terms and O(t) time is needed to check the term equivalence. Thus, the overall complexity is O($t\ log\ t$).

5. Keyword Generation: Keyword generation needs to count the number of flows containing a particular term which can be done in O($f \times t$) time where f is the number of flows in bin and t is the number of terms in the flow.

6. Keyword Filtering and Enhancement: Keyword filtering and enhancement component requires matching set of keywords with set of bad words list. It has a complexity of O($k\ log\ k$) where k is number of keywords (or bad words) in the list which has similar operations as that of binning component.

7. Relative Distance Calculation: Relative distance calculation also needs to scan the payload one more time identifying the position of keyword which has complexity of O(t) where t is number of terms in payload.

8. *RDCCA* Generation: This component creates *RDCCA* by encoding identified keywords and their relative distances which can be performed in constant time.

3.3 Experiments

In order to evaluate the performance of *RDClass*, we conducted experiments using three different datasets which are either publicly available or generated in our testbed setup. In this section, we first describe how the first dataset is generated in our testbed setup and also present the details of other two public datasets. Subsequently, we present the experiments conducted to evaluate the classification accuracy of *RDClass* and report the obtained results.

3.3.1 Testbed Setup and Dataset Description

To evaluate the classification accuracy of *RDClass*, we performed experiments on three different datasets. First dataset was generated in Networks and Security Lab of IIT Indore and contains flows belonging to five applications - BitTorrent, Dropbox, Hypertext Transfer Protocol (HTTP), Session Initiation Protocol (SIP) and Secure Shell (SSH). We used the testbed setup similar to the one shown in Figure 3.10. The setup consisted of three computers, each having configuration as Intel Core i5-4590 processor with clock rate of 3.30 G Hz and 8 GB of physical memory. All these computers were running Ubuntu 16.04 LTS operating system. We captured the generated dataset using tcpdump tool [24]. The details of how these flows were generated are as follows:

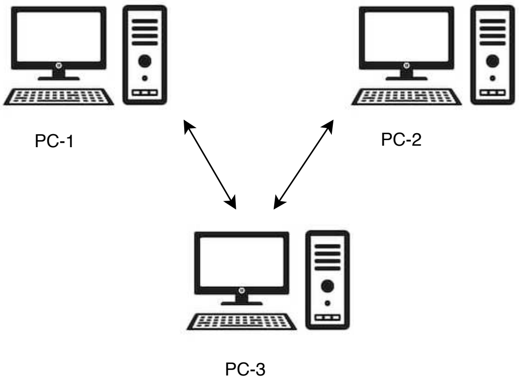


Figure 3.10: Testbed Setup used to Generate Dataset

1. BitTorrent: We used only one computer to generate BitTorrent traffic. The flows of this application were generated by downloading some open source ".iso" files from BitTorrent application [3]. The ".torrent" files of these ".iso" files were first downloaded from website *https://isohunts.to/* and then given as input to BitTorrent application in order to download the required content. The resulting traffic was then collected using tcpdump for 1 hour.

2. Dropbox: Here also we used only one computer from the testbed setup. We first installed Dropbox software [6] and provided the required credentials so that software can synchronize with the server. We then used few files which we put in dropbox and removed from dropbox at intervals for synchronization. Simultaneously, we started capturing the flows belonging to Dropbox and stopped the capture after 2 hours.

3. HTTP: Here also we used only one computer from the testbed setup. The flows belonging to HTTP were generated by browsing top 200 websites from a computer according to Alexa rankings [1]. The generated traffic was captured for 3 hours.

4. SIP: To generate SIP traffic, we designated one PC in our setup as VoIP PBX [27] installed with Asterisk version 11.18.0 [2]. Other two machines were client machines mimicking two enterprises. We created 90 simulated users using these two client machines. These users were created using a modified version of VoIP Bot program that uses Jain SIP API [8] and Java Media Framework [9]. The created users were executed as threads such that each thread acted as a VoIP user to make and receive calls. In total, there were approximately 500 VoIP calls made with each call duration varying from 1 minute to 10 minutes. We captured the generated traffic for about 2 hours.

5. SSH: To generate the SSH traffic, we again used the whole testbed setup. In one of the PC, we installed OpenSSH server [19] and in other two computers which were designated as SSH clients, we created few user accounts and established SSH connections from each of the two clients to the server. The resulting traffic was captured for approximately 30 minutes.

The flows belonging to each of these five protocols were captured in the form of *pcap* traces.

Our second dataset is a publicly available dataset made available by Digital Corpora [5]. This dataset has traffic of 4 text based protocols - Dropbox, HTTP, SIP and SMTP and contains about 110 thousand application flows. Third dataset is also made publicly available by FOI Information warfare lab [17]. This dataset has traffic of 5 text based application protocols - CLDAP, FTP, HTTP, POP and SMB and contains about 29 thousand application flows. Henceforth in this chapter, we name the first, second and third dataset as private, public-1 and public-2 dataset respectively. We divided each dataset into two parts such that the first part was used for training purpose in which signature of each application layer protocol was generated while the second part was used for testing purpose. The details of the private, public-1 and public-2 datasets such as their size and number of flows in training and testing phases are shown in Tables 3.3, 3.4 and 3.5 respectively. In the next subsection, we describe the experiments performed to evaluate the classification performance of *RDClass* and present the obtained results.

Table 3.3: Private Dataset Statistics

Protocol	**Flows for Training**	**Size (in MB)**	**Flows for Testing**	**Size (in MB)**
BitTorrent	001816	145.8	001892	218.4
Dropbox	003005	059.3	003010	192.3
HTTP	127444	117.3	131474	431.5
SIP	000678	068.6	000695	109.6
SSH	002208	006.2	002000	005.5
Total	**135151**	**397.2**	**139071**	**957.3**

Table 3.4: Public-1 Dataset Statistics

Protocol	**Flows for Training**	**Size (in MB)**	**Flows for Testing**	**Size (in MB)**
Dropbox	00014	000.1	00026	000.3
HTTP	52164	207.5	53354	210.3
SIP	00676	069.5	00694	105.6
SMTP	01510	031.5	01574	032.3
Total	**54364**	**308.6**	**55648**	**348.5**

Table 3.5: Public-2 Dataset Statistics

Protocol	**Flows for Training**	**Size (in MB)**	**Flows for Testing**	**Size (in MB)**
CLDAP	1820	00.4	1782	00.4
FTP	4378	23.1	4376	20.1
HTTP	0872	01.1	0856	01.3
POP	0994	00.4	1050	00.8
SMB	6322	21.2	6222	21.9
Total	**14386**	**46.3**	**14286**	**44.5**

3.3.2 Evaluation

We wrote a Java program to implement *RDClass*. This program was run on a computer having Ubuntu 16.04 LTS operating system and Intel Core i5 quad-core processor and 4 GB physical memory. The program uses jNetPcap programming library [10] to read and process packets from traces. jNetPcap library also has a module for bidirectional flow reconstruction. We used this module for reconstructing flows from packets in our experiments. During training phase, $RDClass$ started with zero knowledge of any application and generated $RDCCA$ instances for each application using the training dataset. We measured the classification performance of *RDClass* in terms of $Recall$. *Recall* is the ratio of flows correctly labeled as a particular application (True Positive (TP)) to the total number of flows (TP + False Negative (FN)) belonging to that application and is given by Equation 3.3

$$Recall = \frac{TP}{TP + FN} \tag{3.3}$$

We performed three experiments as homogeneous, heterogeneous and grand experiments with the previously described three datasets to assess the classification performance of $RDClass$. In our experiments, we selected first 1024 (n = 1024) bytes of the flow for generating signatures. The details of these experiments and obtained performance results are described below

1. Homogeneous Experiment: In the homogeneous experiment, we used the training portion of each of the three datasets (Private, Public-1 and Public-2) to generate signatures for each of the protocols. The other portion of the same dataset is used as the testing dataset. In this experiment, *Recall* obtained for each protocol in private, public-1 and public-2 datasets are shown in Tables 3.6a, 3.6b and 3.6c respectively. From these results, we can notice that *RDClass* has an average *Recall* greater than 99% and thus, this experiment suggests that *RDClass* performs very well if the testing dataset is from the same site as that of training dataset.

2. Heterogeneous Experiment: In heterogeneous set of experiments, we used the training portion of each dataset to generate signatures for application protocols in

Table 3.6: *Recall* for Homogeneous Experiments

(a) Private Dataset

Protocol	*Recall* (in %)
BitTorrent	100.00
Dropbox	100.00
HTTP	099.23
SIP	099.98
SSH	100.00

(b) Public-1 Dataset

Protocol	*Recall* (in %)
Dropbox	100.00
HTTP	098.37
SIP	099.13
SMTP	100.00

(c) Public-2 Dataset

Protocol	*Recall* (in %)
CLDAP	100.00
FTP	100.00
HTTP	099.29
POP	099.04
SMB	098.50

that dataset. The testing portions of the other two datasets were used to calculate the *Recall*. The idea was to assess robustness of signatures generated in classifying applications when presented with dataset collected from different sites[3]. The benefit of using data-sets from different sites is that, if the signatures are generated from the dataset of same site, it may have site dependent words such as User-Agent for HTTP can be Mozilla or User-Agent of SIP can be Asterisk. These words can become keywords (if selection is automated as is the case with *RDClass*) as they appear in nearly all flows of the application. However, multiple sites are likely to have different end devices and software which can avoid these site specific terms to be selected as keywords. Therefore we used datasets generated from multiple sites.

Tables 3.7, 3.8 and 3.9 show the *Recall* for the cases where the signatures were generated with only the training portion of the Private, Public-1 and Public-2 datasets and tested with testing portion of the other two datasets respectively.

Table 3.7: *Recall* for Training with Private Dataset and Testing with Public-1 and Public-2 Datasets

(a) Public-1 Dataset

Protocol	*Recall* (in %)
Dropbox	100.00
HTTP	099.17
SIP	099.91

(b) Public-2 Dataset

Protocol	*Recall* (in %)
HTTP	99.34

It is worth noting that, in this set of experiments, testing was done only for the overlapping set of protocols (training and testing). We can notice from the three tables that in all the cases of cross evaluation, *Recall* is over 98% which indicates that

[3] All three datasets are collected from different sites

Table 3.8: *Recall* for Training with Public-1 Dataset and Testing with Private and Public-2 Datasets

(a) Private Dataset

Protocol	*Recall* (in %)
Dropbox	100.00
HTTP	099.08
SIP	100.00

(b) Public-2 Dataset

Protocol	*Recall* (in %)
HTTP	99.78

Table 3.9: *Recall* for Training with Public-2 Dataset and Testing with Private and Public-1 Datasets

(a) Private Dataset

Protocol	*Recall* (in %)
HTTP	98.99

(b) Public-1 Dataset

Protocol	*Recall* (in %)
HTTP	99.57

the generated signatures were robust in detecting applications when presented with datasets from other sites.

3. Grand Experiment: In grand experiment, we merged the training portion of all the protocols of the three datasets and generated a grand training dataset. This grand training dataset is used to generate *RDCCA* for each protocol. During training phase, *RDClass* automatically generated an instance of *RDCCA* for each protocol and we observed that it took approximately 175 seconds to process 210 thousand application flows as shown in Table 3.10 to generate *RDCCA* for each protocol whose flows were present in the training dataset. Figures 3.6, 3.11, 3.12, 3.13, 3.14, 3.15,

Table 3.10: *RDCCA* Generation Time

Protocol	Training Time (in seconds)
BitTorrent	011.134
CLDAP	001.135
Dropbox	023.484
FTP	008.968
HTTP	064.396
POP	000.975
SIP	053.747
SMB	006.905
SMTP	003.055
SSH	001.997
Total time	**175.796**

3.16, 3.17, 3.18 and 3.19 show the *RDCCA*s generated for BitTorrent, SMTP, HTTP, SIP, SMB, CLDAP, Dropbox, FTP, POP and SSH respectively after successful training.

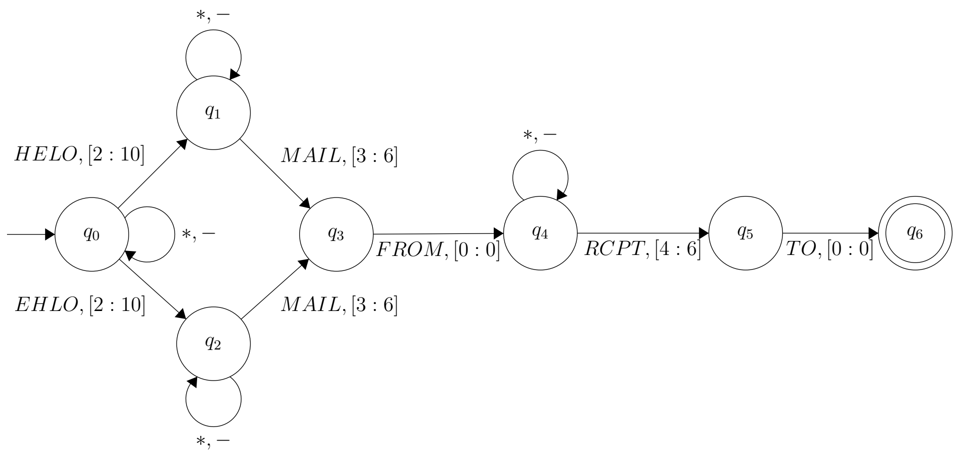


Figure 3.11: *RDCCA* Generated for Simple Mail Transfer Protocol

For testing purpose, we used the three dataset's respective testing portions and also a combined dataset (of all the three testing portions) to evaluate the performance. Table 3.11 shows the *Recall* for each protocol. We can notice from this table that, *RDClass* show an average *Recall* rate of more than 99%.

Table 3.11: *Recall* for Grand Experiment

Protocol	***Recall*(in %)**
BitTorrent	100.000
CLDAP	100.000
Dropbox	100.000
FTP	100.000
HTTP	098.138
POP	099.047
SIP	099.568
SMB	098.505
SMTP	100.000
SSH	100.000

We further evaluated the robustness of *RDCCA*s generated for each protocol with a $n \times n$ evaluation where each flow of every protocol in the testing portion of dataset is given as input to every *RDCCA*. This evaluation shows the uniqueness of *RDCCA*'s keywords and their ordering in identifying applications. Table 3.12 shows the classi-

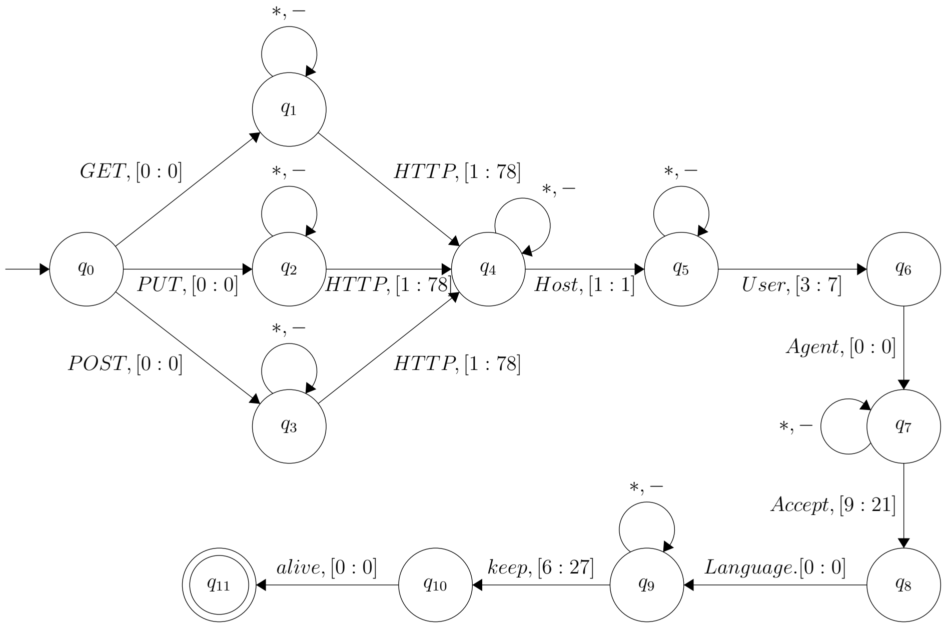


Figure 3.12: *RDCCA* Generated for Hyper Text Transfer Protocol

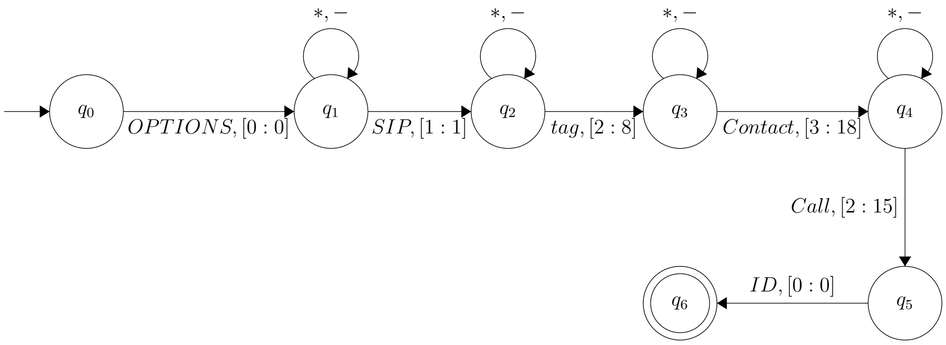


Figure 3.13: *RDCCA* Generated for Session Initiation Protocol

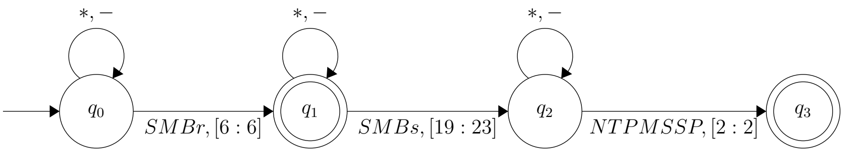


Figure 3.14: *RDCCA* Generated for Server Message Block

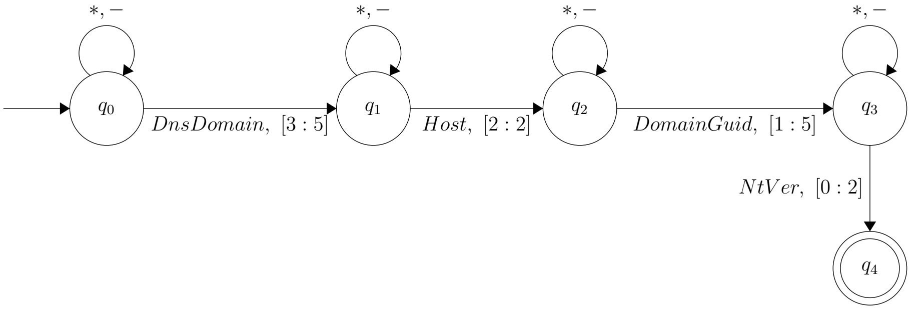


Figure 3.15: *RDCCA* Generated for Connectionless Lightweight Directory Access Protocol

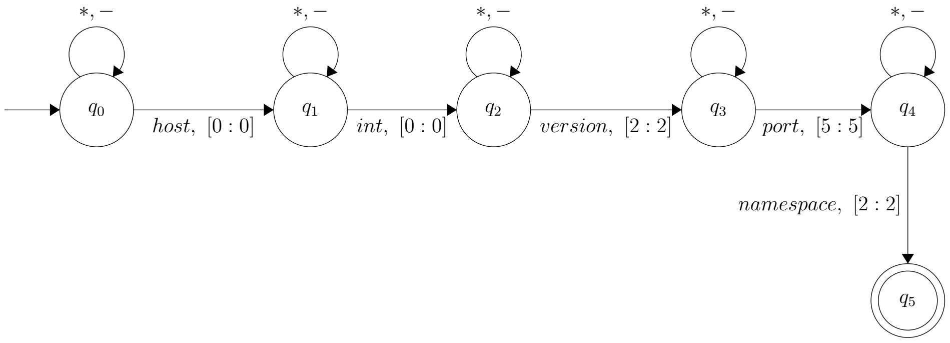


Figure 3.16: *RDCCA* Generated for Dropbox

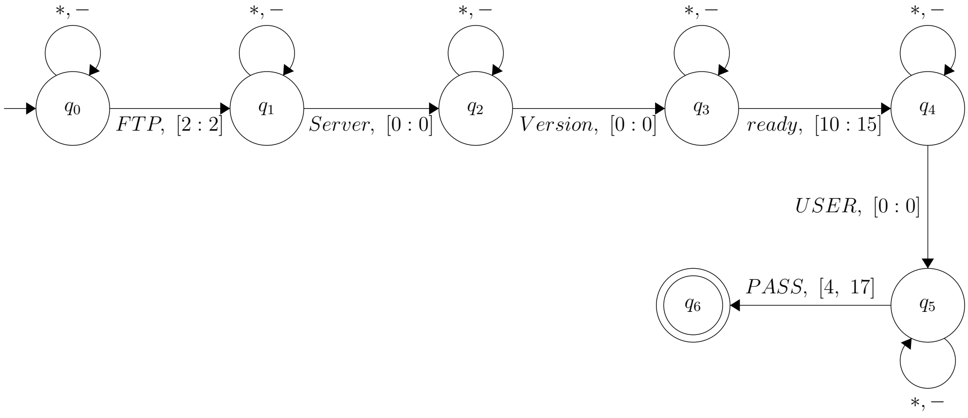


Figure 3.17: *RDCCA* Generated for File Transfer Protocol

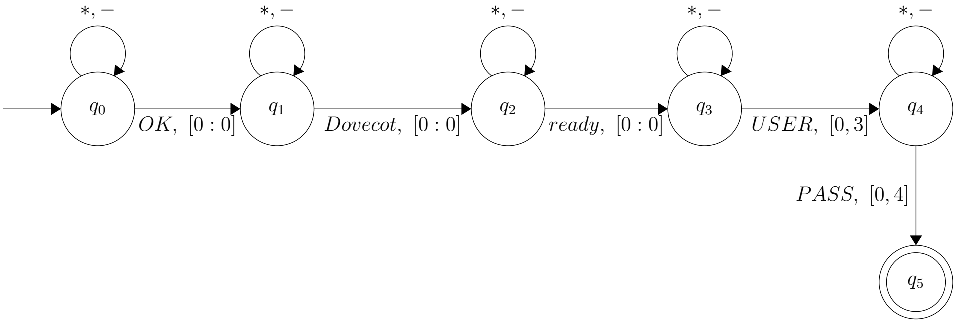


Figure 3.18: *RDCCA* Generated for Post Office Protocol

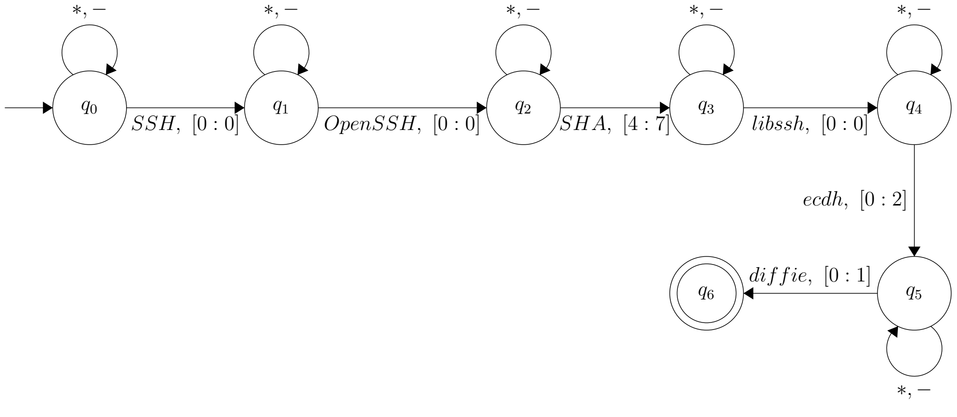


Figure 3.19: *RDCCA* Generated for Secure Shell

fication performance of ten application layer protocols used in our grand experiment. An entry in a particular row and column indicates the number of flows of column header classified against *RDCCA*s of row header. For example, the first column BitTorrent has 1892 number of flows in testing dataset and all of them are correctly classified as BitTorrent by the corresponding *RDCCA* and none of these flows are classified as any other protocol. Similar interpretation is done for other cases too. We can notice that there is no case where one protocol is interpreted as other protocol which indicates that generated *RDCCA*s are robust. However there are some flows which are not classified. For example, 10 out of 1050 flows of POP are not classified while rest of the 1040 flows were classified correctly.

Table 3.12: $n \times n$ Evaluation Matrix for Flow Classification in *RDClass*

	Bit-Torrent (1892)	**CL-DAP (1782)**	**Drop-box (3036)**	**FTP (4376)**	**HTTP (185684)**	**POP (1050)**	**SIP (1389)**	**SMB (6222)**	**SMTP (1574)**	**SSH (2000)**	**Unclassified**
BitTorrent	1892	0	0	0	0	0	0	0	0	0	0
CLDAP	0	1782	0	0	0	0	0	0	0	0	0
Dropbox	0	0	3036	0	0	0	0	0	0	0	0
FTP	0	0	0	4376	0	0	0	0	0	0	0
HTTP	0	0	0	0	182228	0	0	0	0	0	3456
POP	0	0	0	0	0	1040	0	0	0	0	10
SIP	0	0	0	0	0	0	1383	0	0	0	6
SMB	0	0	0	0	0	0	0	6129	0	0	93
SMTP	0	0	0	0	0	0	0	0	1574	0	0
SSH	0	0	0	0	0	0	0	0	0	2000	0

3.3.3 Performance Comparison and Discussion

We compared classification accuracy of *RDClass* with a recent network traffic classification method *SANTaClass* [104] on our grand dataset. *SANTaClass* is an unsupervised learning method which automatically generates signature from network flow. In this method, keywords are extracted from flow payload and arranged in the same sequence in which they occur in the flow. These sequences are then called as signatures. Each application protocol has unique signature which is used for classification purpose. We experimented with *SANTaClass* on our dataset and the obtained classification results are shown in Table 3.13. We can notice that this method resulted into few signature overlaps (shown in red colour in Table 3.13). The reason

Table 3.13: $n \times n$ Evaluation Matrix for Flow Classification in *SANTaClass*

	Bit-Torrent (1892)	CL-DAP (1782)	Drop-box (3036)	FTP (4376)	HTTP (185684)	POP (1050)	SIP (1389)	SMB (6222)	SMTP (1574)	SSH (2000)	Unclassified
BitTorrent	1892	0	0	0	0	0	0	0	0	0	0
CLDAP	0	1782	0	0	0	0	0	0	0	0	0
Dropbox	0	0	3036	0	0	0	0	0	0	0	0
FTP	16	0	0	4352	0	0	0	8	0	0	0
HTTP	10	0	0	0	182212	0	0	6	0	0	3456
POP	0	0	0	0	0	1040	0	0	0	0	10
SIP	0	0	0	0	0	0	1383	0	0	0	6
SMB	0	0	0	0	0	0	0	6129	0	0	93
SMTP	0	0	0	0	0	0	0	0	1574	0	0
SSH	0	0	0	0	0	0	0	0	0	2000	0

for signature overlap was that the signature generated by *SANTaClass* checks only sequence of keywords and the same sequence was found in other protocol signatures too particularly with those having smaller number of keywords in their signatures. On the other hand, *RDClass* checks the sequence as well as the relative position of the keywords which did not result into false classifications.

3.3.4 Limitations

From the experimental results presented in this chapter, it can be concluded that *RDClass* can classify applications with very high accuracy. However, there are also few limitations of the proposed approach as described below:

1. Keyword Filtering and Enhancement: *RDClass*'s performance is based on accurate keyword selection and hence, presence or absence of ill suited or important keywords respectively has an impact on its performance.

2. Classification of Text based Protocols Only: *RDClass* is designed to classify text based protocols only. For binary protocols, this model is not suitable. *RDClass* currently cannot handle such protocols hence it can be best used along with another method which purely works on header and flow level statistics to deal with such traffic.

3.4 Conclusion

In this chapter, we described a DPI based traffic classification method called *RDClass*. This method is automated and can select keywords automatically when presented with unknown application payloads. *RDClass* uses relative distance between keywords along with their ordering to accurately identify application flows. A new state transition machine which enforces constraints on ordering and their relative distances of keywords was also proposed. We performed extensive experiments with a range of applications and showed that *RDClass* can accurately identify applications. However, *RDClass* is limited to classifying only text based protocols and it can not classify binary protocols. Thus, in next chapter of the thesis, we overcome this issue by proposing a traffic classification approach that can classify both text as well as binary protocols.

Chapter 4

Network Traffic Classification through Encoded Bit Level Signatures

4.1 Introduction

Traditional network traffic classification methods extract byte level information from the application flows to generate the signatures. However, an important challenge for classifying network flows is to capture the signatures which are unique and minimal in length as signature length governs the computational overhead. Some recent works [105] propose to automate the signature generation process for application network traffic classification addressing these concerns. However, there is a surge in data-transfer formats and increasingly network protocols and applications are encoded at the bit-level. This makes the byte-level signatures ineffective in traffic classification. Few works [123, 124] recently proposed in the literature generate application signatures at bit level granularity of payload. However, traffic classification using bit level payload analysis is a recent approach and only preliminary work has been done in this area. Borrowing motivation from this, we propose two DPI-based bit-level signature generation methods for accurate traffic classification. Our methods generate bit-level signatures using first n bits of bidirectional flows belonging to an application. Our contributions in this chapter are as follows.

1. We propose *BitCoding* and *BitProb* which are bit-level signature based application classification methods.

2. Our proposed methods are computationally less expensive as they generate signatures from the first n bits of bidirectional flow. Our experiments show that even with first 40 bits of flow, the proposed methods can accurately identify applications.

3. *BitCoding* and *BitProb* translate the application signatures into state transition machines called $Transition\ Constrained\ Counting\ Automata$ ($TCCA$) and *Probabilistic Counting Deterministic Automata* (*PCDA*) respectively for bit-level comparison with application flows.

4. We perform extensive experiments on three different datasets consisting of text based protocols, binary protocols and proprietary protocols to assess the performance of proposed classification methods and show that the proposed methods are protocol-type agnostic.

5. We perform cross evaluation of the proposed methods on datasets from different sites and show that they are able to classify flows with a small compromise in *Recall*.

Rest of this chapter is organized as follows. In Section 4.2, we describe the working principle of *BitCoding* and *BitProb*. We describe the experiments performed to evaluate the classification performance of proposed methods in Section 4.3. In Section 4.4, we compare the classification performance of *BitCoding* and *BitProb* with some of the prior works. We end this chapter with concluding remarks in Section 4.5.

4.2 Proposed Traffic Classification Methods

As discussed in the last section, we propose two bit-level signature based application classification methods *BitCoding* and *BitProb*. In this section, we first describe the motivation behind proposing these methods which is subsequently followed by the explanation of working of these methods.

4.2.1 Motivation

The choice of generating application signatures using bit-level content is motivated by the fact that there is a surge in data-transfer formats and increasingly network protocols and applications are now encoded at the bit-level. For example, consider the first 32 bits of NTP packet structure shown in Figure 4.1. The first two bits (0th and 1st) indicate whether the last minute of the day will have a leap second or not. Next 3 bits indicate the version number of NTP and the bits from 5 to 7 indicate different modes of NTP. In this case, even if the mode of NTP is changed (hence change in bit positions of 5 to 7), we can still use the first 5 bits to guess the protocol or even if the first two bits are changed, the bits corresponding to version number (which are not likely to change) can be indicators of protocol type. Thus, the bit-level details can be used for robust signature generation[1].

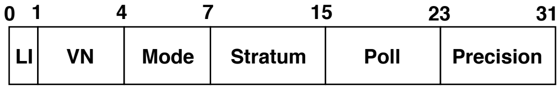


Figure 4.1: First 32 Bits of Network Time Protocol Packet Structure

4.2.2 *BitCoding*

Our first application identification/traffic classification technique *BitCoding* uses signatures generated from bit-level content of flow payload to identify different application flows. These signatures are efficiently encoded so as to reduce the total signature length. These encoded signatures are converted into a state transition machine. *BitCoding* uses first n bits from bidirectional flow to generate signatures. The first step for signature generation is bidirectional flow reconstruction from network traffic. Subsequent to the reconstructed flow, first n bits of the flow are extracted for signature generation. In the next two subsections, we describe signature generation from training data and classifying flows from generated signatures.

[1] Bit-level signatures guarantee a lower bound performance of byte-level signatures for the same content length.

4.2.2.1 Signature Generation and Encoding

BitCoding generates application specific signatures with training data. This process has four stages - 1) Flow Reconstruction, 2) Bit signature generation, 3)Run Length Encoding and 4) State Transition Machine construction. These four stages are shown in Figure 4.2 and elaborated subsequently.

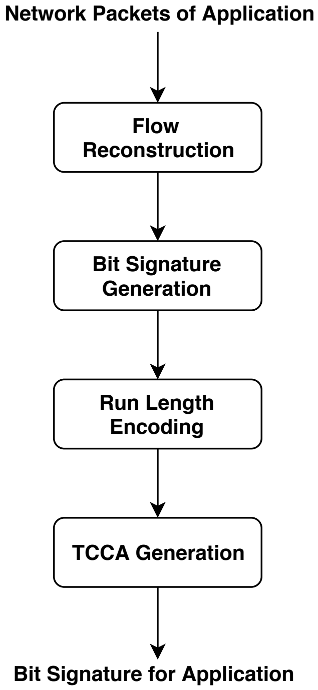


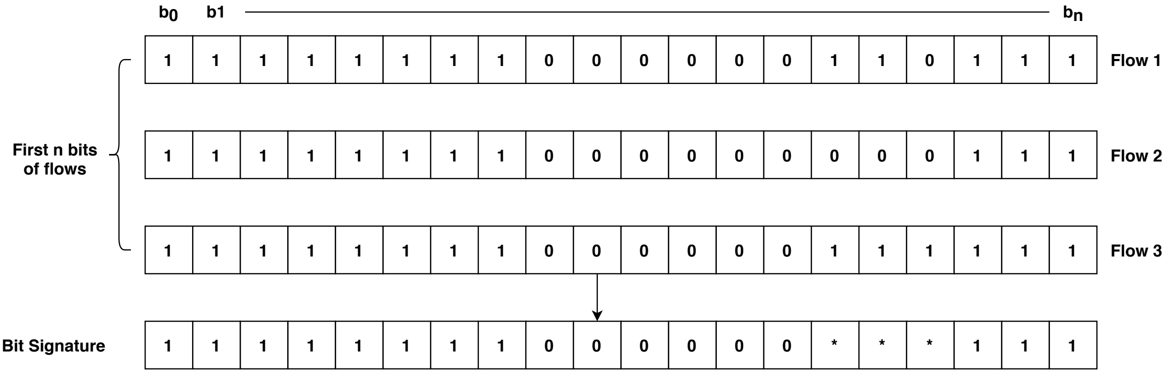
Figure 4.2: Bit-level Signature Generation for *BitCoding*

1. Flow Reconstruction: The bidirectional TCP and UDP network flows are constructed using the method discussed in Section 3.2.2 of the Chapter 3. Once the bidirectional flows are reconstructed, they are given as input to the next stage.

2. Bit Signature Generation: *BitCoding* generates application specific bit signatures using a set of invariant bits of the payload selected in the previous stage. Assuming there are K ($K \in I$) such bidirectional flows of an application in training set, it collects the first n bits from each of the K bidirectional flows of application $\mathcal{A}$ and generates n bit signature A_{Sig} for that application. The first n bits of the i^{th} flow ($1 \leq i \leq K$) are of the form $f_1^i, f_2^i, \cdots, f_n^i$. All K flow extracts are used for generating signatures as follows. The j^{th} bit $[1, n]$ position of every flow extract are used to decide the j^{th} signature bit of A_{Sig}. The j^{th} signature bit is set to 0 if the j^{th} bit of every flow $(1 \leq j \leq K)$ has a value of 0 and the j^{th} signature bit is set to 1 if the j^{th} bit of every flow $(1 \leq j \leq K)$ has a value of 1. If some of these bit positions have 0's and 1's, the j^{th} signature bit is set to *. The signature generation method is shown in Table 4.1 where each s_i is a signature bit. A sample application bit signature

Table 4.1: Flow Extracts and Signature Generation

Flow 1	f_1^1	f_2^1	....	f_n^1
Flow 2	f_1^2	f_2^2	....	f_n^2
Flow 3	f_1^3	f_2^3	....	f_n^3
.	.	.	.	.
.	.	.	.	.
Flow K	f_1^K	f_2^K	....	f_n^K
Signature	s_1	s_2	....	s_n



Flow 1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	1	0	1	1	1
Flow 2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1
Flow 3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1
Bit Signature	1	1	1	1	1	1	1	1	0	0	0	0	0	0	*	*	*	1	1	1

Figure 4.3: Bit Signature Generation

generation using bits is shown in Figure 4.3. In this example, there are 3 flows each with 20 bits and are used for signature generation. The first bit of all the three flows are having a value of 1, thus this bit is included in the signature as 1. Also the next 7 bits of each of the flows are also having 1's, hence the next 7 signature bits are also labeled with 1's. Similarly the bit positions from 9-14 of all the 3 flows are having 0's and hence the signature bits 9-14 also will have 0's in these positions. The 15^{th} bit of flow 1 is having a bit value of 1, flow 2 is having a value of 0 and flow 3 is having a value of 1. Hence the corresponding bit in signature is set to * (indicating 0/1 and insignificant for detecting this application). Similar labeling is done with remaining bits and the final signature generated is $11111111000000 * * * 111$.

3. Run-length Encoding: We can notice that signature bits consist of 1's, 0's and *'s and each signature is of n bits. For efficient representation, storage and comparison purposes, we perform Run-length Encoding (RLE) [91] of these n bits. RLE is a technique used in loss-less data compression. RLE reduces the size of a repeating string of characters by specifying number of repetitions. In RLE, runs of data i.e. sequences of the same data value in many consecutive data elements are stored as a single value and count of number of times that data value is repeated is stored. Consider the signature sequence for the above example which has bit values of $11111111000000 * * * 111$ and after encoding with RLE, it is converted to $8W6Z3 * 3W$ where W and Z represent bit values of 1 and 0 respectively.

4. *Transition Constrained Counting Automata* ($TCCA$) Generation: Subsequent to the RLE, an encoded signature needs to be compared with network traffic flows to identify applications. For this purpose, we convert the encoded bit signature into a new state transition machine $Transition\ Constrained\ Counting\ Automata$ ($TCCA$). This $TCCA$ is the final bit signature for an application. To create $TCCA$ of an application, the compressed RLE bit signature of that application is given as input. The machine $TCCA$ is formally defined as $\mathcal{M} = (Q, \Sigma, C, \delta, q_0, F)$ where

Q is a finite set of states
Σ is a finite set of input symbols
C is a finite set of Counters with each $c_i \in C$ taking values in $\mathcal{N} \bigcup 0$
δ is a set of transitions defined as $\delta : Q \times C \times \Sigma \to Q \times (C \to C)$
$q_0 \in Q$ is an initial state
$F \subseteq Q$ is a set of final states

Each transition $\delta_i \in \delta$ is a six tuple $\langle q_i, q_j, c, \sigma, \phi(c_i), Inc(c_j) \rangle$ with the elements representing

q_i is current state

q_j is next state

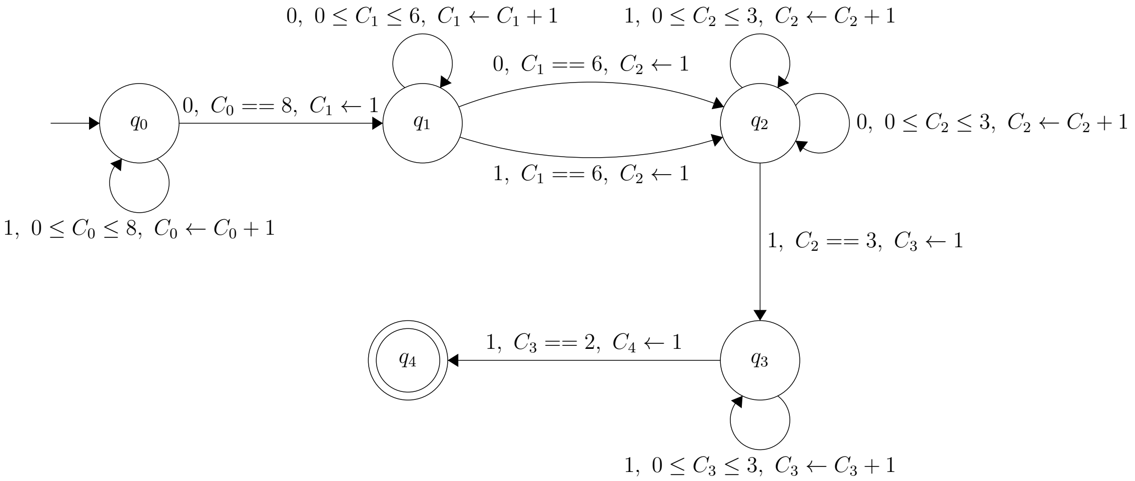
c_i is counter value at the state q_i

$\sigma \in \Sigma$ is an input symbol

$\phi(c_i)$ is a constraint (invariant condition) on counter value c_i at state q_i on this transition

$Inc(c_j)$ is a function which initialize the counter in the next state q_j to a new value

A sample $TCCA$ generated with a 20 bit compressed signature generated in the last step is shown in Figure 4.4. There are 5 states in the $TCCA$ labeled from q_0 to q_4 with q_0 being start state and q_4 being final state. Each state has a counter (C_0 to C_4) which will be initialized to a new value every time a transition visits the state. The machine starts in state q_0 with counter at q_0 being set to 0, read bits from test flow and makes allowed transitions to reach the final state. The transitions of $TCCA$ have an input symbol (bit value) and a constraint on counter value which acts as a guard and the transition is allowed only if the constraint is satisfied (evaluated to be true). For example, in Figure 4.4, state q_0 has a transition defined to itself (self loop)



$0,\ 0 \le C_1 \le 6,\ C_1 \leftarrow C_1 + 1$
$1,\ 0 \le C_2 \le 3,\ C_2 \leftarrow C_2 + 1$
$0,\ C_1 == 6,\ C_2 \leftarrow 1$
q_0
$0,\ C_0 == 8,\ C_1 \leftarrow 1$
q_1
q_2
$0,\ 0 \le C_2 \le 3,\ C_2 \leftarrow C_2 + 1$
$1,\ C_1 == 6,\ C_2 \leftarrow 1$
$1,\ 0 \le C_0 \le 8,\ C_0 \leftarrow C_0 + 1$
$1,\ C_2 == 3,\ C_3 \leftarrow 1$
q_4
$1,\ C_3 == 2,\ C_4 \leftarrow 1$
q_3
$1,\ 0 \le C_3 \le 3,\ C_3 \leftarrow C_3 + 1$

Figure 4.4: $TCCA$ Generated for the Example Signature 8W6Z3*3W

on input 1. This transition has a constraint on counter value of C_0 being in between 0 to 8. This constraint maps the requirement of reading 8 consecutive 1's in the flow at the beginning, while incrementing the counter value by 1 each time it is traversed. The next transition from q_0 to q_1 is on input 0 and is valid only when the counter value at C_0 is 8 (having read 8 consecutive 1's) and sets the counter C_1 at q_1 to 1 (read a 0 after 8 consecutive 1's). Whenever there is * in signature, it will have two transitions one with input 0 and other with input 1 both increment the counter values at the next states. Similarly all bits and their positions are enforced by $TCCA$. The training algorithm ($TCCA$ generation for an application A) is shown in Algorithm 4.1. The inputs given to this algorithm are the network traffic of application A and the value of n for extracting first n bits from each bidirectional flow. The output of this algorithm is the $TCCA$ for application A. This algorithm reads the packets of an application A and reconstructs the bidirectional flows. Subsequently, it counts the total number of flows. From each flow, the algorithm extracts *BitString* which is the first n bits of that flow. After this, the algorithm finds the invariant bits in *BitString* and updates *BitCode* accordingly. Once updated *BitCode* is obtained after analyzing all *BitString*s, *RLE* is applied to it which is then used to construct $TCCA$. *ConstructTCCA* function starts with a start state, reads the *EncodedBitCode* and constrains the counters according to it. Subsequently, it adds the next state and repeats the process till final state.

Algorithm 4.1 $TCCA$ Generation for Application A

```
Input: Network Traffic of Application $A$
Input: $n$- Number of Bits
Output: $TCCA$ for Application $A$
while There are Packets to Read do
    $pkt \leftarrow ReadPacket()$
    $Flows \leftarrow$ FlowReconstructor($pkt$)
end while
$FlowCount \leftarrow$ GetFlowCount($Flows$)
for $t = 1$ to $FlowCount$ do
    $BitString \leftarrow$ GetBitString($F_{Test}$, $n$)
    $BitCode \leftarrow$ UpdateBitCode($BitString$, $t$)
end for
$EncodedBitCode \leftarrow$ RLE($BitCode$)
$TCCA \leftarrow$ ConstructTCCA($EncodedBitCode$, $n$)
Return $TCCA$
```

4.2.2.2 Flow Classification

Similar to signature generation process, in flow classification phase too, bidirectional flows are first reconstructed. Subsequently, first n bits of the flow are extracted for comparison with $TCCA$s generated for different applications as shown in Figure 4.5. Each of these $TCCA$s is in their respective start states and the n bits from a test flow F_{test} are given as input (a bit at a time from first to last bit) to all the application signatures/$TCCA$s. Every $TCCA$ makes allowed transitions and the flow is labeled with application A if the corresponding $TCCA$ of application A reaches the final state. If none of the $TCCA$s reach the final state, the flow is labeled as unclassified. Consider two example bit sequences as 11111111000000101111 and 00111111000000101111. The first sequence 11111111000000101111 which is given as input to the $TCCA$ of Figure 4.4, it is easy to see that it reaches the final state as it contains eight 1's in the beginning and six 0's next, subsequent three bits are accepted invariably and then the last three digits are 1's. However second sequence is not accepted by $TCCA$ as it starts with a 0 and there is no transition with input 0 when the counter C_0 is 0 at state q_0. This classification procedure is shown in Algorithm 4.2. Algorithm 4.2 classifies the

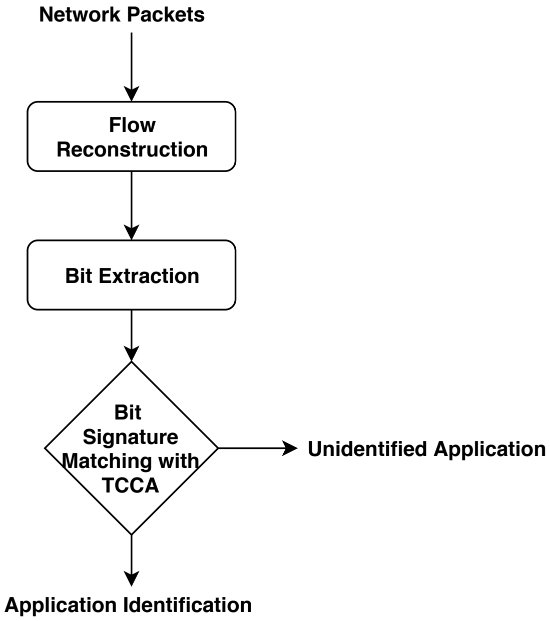


Figure 4.5: Flow Classification in $BitCoding$

Algorithm 4.2 Classifying Network Flows

```
Input: $LIST_{TCCA}$ - List of $TCCA$s for all applications
Input: $F_{Test}$- Test Flow
Output: Label of $F_{Test}$

$Size$ ← SizeOf($LIST_{TCCA}$)
$BitString$ ← GetBitString($F_{Test}$)
for t=1 to $Size$ do
    MatchString($BitString$, $LIST_{TCCA}[t]$)
end for
if  $LIST_{TCCA}[t]$ at final state then
    Label $F_{Test}$ is of Application $t$
else
    Label $F_{Test}$ as Unknown Application
end if
```

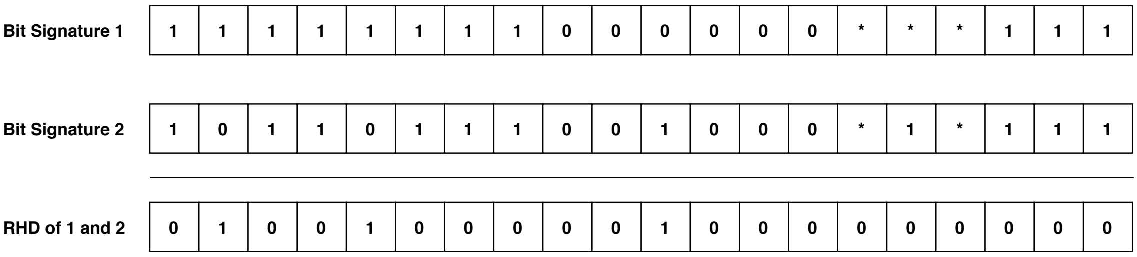
network flow under consideration. The inputs given to this algorithm are the list of $TCCAs$ and a test flow. The output of this algorithm is the labeled test flow. This algorithm first counts the number of $TCCAs$ generated by Algorithm 4.1 and then extracts $BitString$ in the same way it is extracted in Algorithm 4.1. Subsequently, the algorithm compares the binary string with the application signature using its $TCCA$. If the binary string of test flow reaches to the final state (accept state) then the flow is classified as belonging to that application. Otherwise, the algorithm declares the test flow as unclassified (unknown application).

4.2.2.3 Addressing Signature Overlap

It has to be noted that every application has one signature. While short signatures are computationally less expensive (for both storage and comparison) and as number of applications increases, there are chances of signature overlap i.e. two different applications having same signatures. This leads to misclassification of flows of one application as other one[2]. This can be addressed by increasing the signature length of applications. These two are contradictory goals (short signatures and no overlap) to be achieved together. To avoid such overlaps we compute the similarity between signatures of different applications using a form of $Hamming\ Distance$ (HD). Hamming distance is measured between two strings of equal length and its value is the number of bit positions at which the corresponding bits are differing. It

[2] Signature overlap is a common problem in firewalls and intrusion detection systems

is worth noting here that bit signatures also contain *s along with 0s and 1s. We do a minor change in the standard calculation of HD between a pair of signatures, which is ignoring * bits from the comparison. We name this changed distance measurement as *Relaxed Hamming Distance* (*RHD*). An example *RHD* calculation is shown in Figure 4.6. In this figure, there are two bit signatures and the corresponding *RHD* is 3. We measure the *RHD* between every pair of applications and find that the reason for cross signature matching is due to low *RHD* between them. We label those signatures which exhibit maximum similarity (or minimum distance) with other signatures as *weak signatures*. In order to improve the classification performance, we identify a set of application signatures which show maximum similarity and increase the signature lengths only for those protocols.



Bit Signature 1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	*	*	*	1	1	1
Bit Signature 2	1	0	1	1	0	1	1	1	0	0	1	0	0	0	*	1	*	1	1	1
RHD of 1 and 2	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

Figure 4.6: RHD Calculation between Bit Signatures

4.2.2.4 Complexity Analysis of *BitCoding*

Our proposed traffic classification method *BitCoding* has different modules such that some complexity is associated to each of them. Table 4.2 shows the complexity of each module of *BitCoding* which is also indicating the steps done in training (offline) mode and testing (online) mode.

1. Flow Reconstruction: Flow reconstruction needs to inspect each packet header and adds it into appropriate flows. If there are l flows and p packets then it has a complexity of O($l \times p$).

2. Bit Extraction: Bit extraction has to read the first n bits which is a constant time operation, hence its complexity is O(1).

Table 4.2: Module-wise Complexity of *BitCoding*

Module Name	**Complexity**	**Online/Offline**	**Explanation**
Flow-Reconstructor	O($l \times p$)	Both	l= number of flows, p = number of packets in the trace.
Bit Extractor	O(1)	Both	Constant number of bits to be read from the payload training flow
Bit Signature Generation	O(K)	Offline	For K flows of an application
RLE	O(X)	Offline	Need to read n bits for an application and there are X applications
TCCA Generator	O(X)	Offline	Convert *RLE* into a graph for X applications
Flow Classifier	O($m \times n$)	Online	m= number of machines and n = number of bits in the payload of test flow

3. Bit Signature Generation: Bit signature generation has to operate on all flows of an application. If there are K flows of an application then complexity of signature generation is $O(K)$ as number of bits is constant n for every application which is $O(1)$ operation.

4. Run-Length-Encoding (*RLE*): *RLE* has to read n bits of signature which is $O(1)$ operation for each application and for X different applications, it is $O(X)$.

5. *TCCA* Construction: *TCCA* construction is just the encoding of *RLE* signature into a state transition machine which can be done in constant time $O(1)$ and as there are X applications, total complexity is $O(X)$.

6. Flow Classification: Test flows are classified by *TCCA* and it has to match every bit with every possible instance of *TCCA*. This, in the worst case, has a complexity of O($m \times n$) where m is number of machine instances and n is number of bits in a flow.

It has to be noted that we do not consider variant bits (represented by '*') while comparing application signatures and ignores them from comparison even if very small number of flows are having different values. Thus, there will be few misclassifications (as shown later in experiments section). So we propose another bit-level traffic classification method *BitProb* that considers the occurrence probability of bit values at each position without omitting any bit value during signature comparison.

4.2.3 *BitProb*

Our second bit-level traffic classification method *BitProb* is also a supervised learning method where bit-level signatures are generated from bit sequences of a particular application and subsequently used for identifying flows belonging to that application. These signatures are converted into a state transition machine called as *Probabilistic Counting Deterministic Automata* (*PCDA*). *BitProb* uses first n bits from bidirectional flows to generate signatures. The motivation for *BitProb* is to use those bits which are not consistently either '1' or '0' ('*') and are not used in *BitCoding*. By using their probability values of a bit being either '1' or '0' these bits also participate in decision making. The first step for signature generation is bidirectional flow reconstruction from network traffic. Subsequent to the reconstructed flow, first n bits of the flow are extracted for signature generation. In the next two subsections, we describe signature generation from training data and classifying flows from generated signatures.

4.2.3.1 Signature Generation

BitProb generates application specific signatures with training data. This process has three stages - 1) Flow Reconstruction, 2) Bit Signature Generation and 3) State Transition Machine construction. These three stages are shown in Figure 4.7 and described below.

1. Flow Reconstruction: The bidirectional TCP and UDP network flows are constructed using the method discussed in Section 3.2.2 of the Chapter 3. Once the bidirectional flows are reconstructed, they are given as input to the next stage.

2. Bit Signature Generation: *BitProb* takes K bidirectional flows belonging to an application A and generates n bit signature A_{Sig} for that application. Subsequently it calculates the probability of a particular bit b_i at i^{th} position ($1 \leq i \leq n$) having a value of 1 and also 0 i.e. $P(b_i = 1)$ and $P(b_i = 0)$ ($P(b_i = 0) = 1 - P(b_i == 1)$). A sample probability calculation is shown in Figure 4.8 where three hypothetical flows of

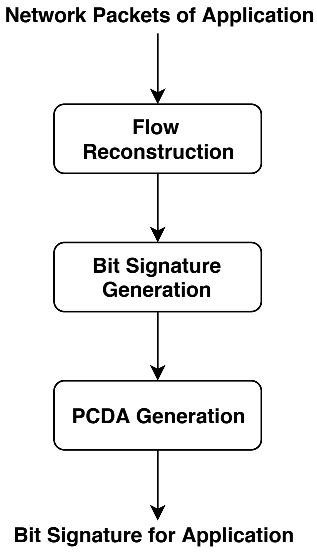
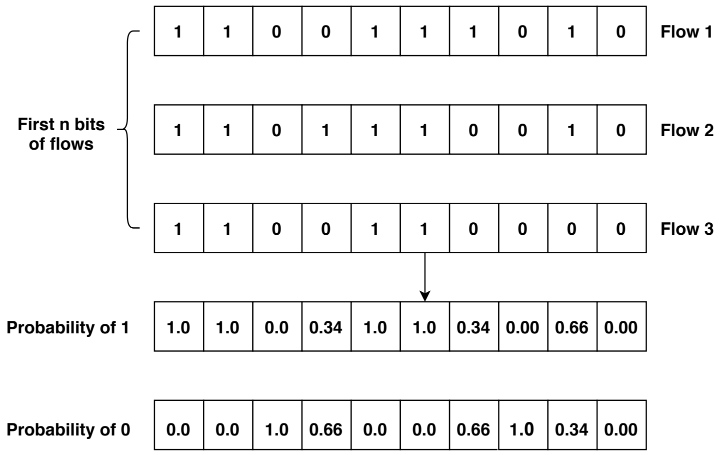


Figure 4.7: Bit-level Signature Generation for $BitProb$

an application are considered for deriving bit probability values. After bit probability values of all n bits are calculated, these bit probability values are used to generate an automata represented as a directed graph as shown in Figure 4.9[3]. This graph is a binary tree with every intermediate node representing one of the two possible cases i.e. either probability of a '1' is read or a '0' is read next. The path from root node to a leaf node indicates the probability with which the bit sequence leading to that leaf node appears in that application. Thus every possible bit string (of length n) has a probability of occurrence in that application. For example, consider the first four bits of example sequence shown in Figure 4.8. The first bit has $P(b_1 = 1) = 1$ and $P(b_1 = 0) = 0$. These two are represented with two transitions from root node to two nodes q_2 and q_1 respectively. Similarly the second bit probability values $P(b_2 = 1) = 1$ and $P(b_2 = 0) = 0$ is denoted with four transitions - two from each of the first level nodes q_2 and q_1 respectively. It can be noticed that, the above representation is not space efficient and it will have $2^n - 1$ number of states for a n bit signature. Thus the above automata is a candidate for optimization. We can notice that certain transitions in the above transition diagram have zero probability values which means that there is a zero probability of that automaton taking that transition

[3] Only first 4 bits out of 10 bits are considered for space constraints



	1	1	0	0	1	1	1	0	1	0	Flow 1
	1	1	0	1	1	1	0	0	1	0	Flow 2
	1	1	0	0	1	1	0	0	0	0	Flow 3
Probability of 1	1.0	1.0	0.0	0.34	1.0	1.0	0.34	0.00	0.66	0.00	
Probability of 0	0.0	0.0	1.0	0.66	0.0	0.0	0.66	1.0	0.34	0.00	

Figure 4.8: Bit Probability Calculation

and hence all the descendant states can be safely removed as they are never visited. This pruning will reduce the size of automaton considerably as network protocols usually have fixed values at fixed positions particularly in the fields of header. A pruned state transition diagram for the above four bit signature is shown in Figure 4.10. We can see that this automaton has only six states as opposed to thirty one in the previous case. Further, we also propose another optimization for encoding these signatures which is merging consecutive same value bit positions with same probability to one state. This is similar to the *RLE* used by *BitCoding* to encode signatures. In the above example, first two bits b_1 and b_2 values are 1 with probability of 1 i.e. $P(b_1 = 1) = P(b_2 = 1) = 1$. These two can be merged by having a counter at the first state. The resulting state transition machine is shown in Figure 4.11. With these two optimization incorporated, we define a new state transition machine and give an algorithm to directly construct it from the probability bit values in the next phase.

3. *Probabilistic Counting Deterministic Automata* ($PCDA$) Generation: We define a new state transition machine called *Probabilistic Counting Deterministic Automata* ($PCDA$) to represent the probabilistic bit signatures. This state transition machine takes the bit probability values and convert them into a probabilistic state transition machine. $PCDA$ is formally defined as

$\mathcal{M} = (Q, \Sigma, C, \delta, P, q_0, F)$ where

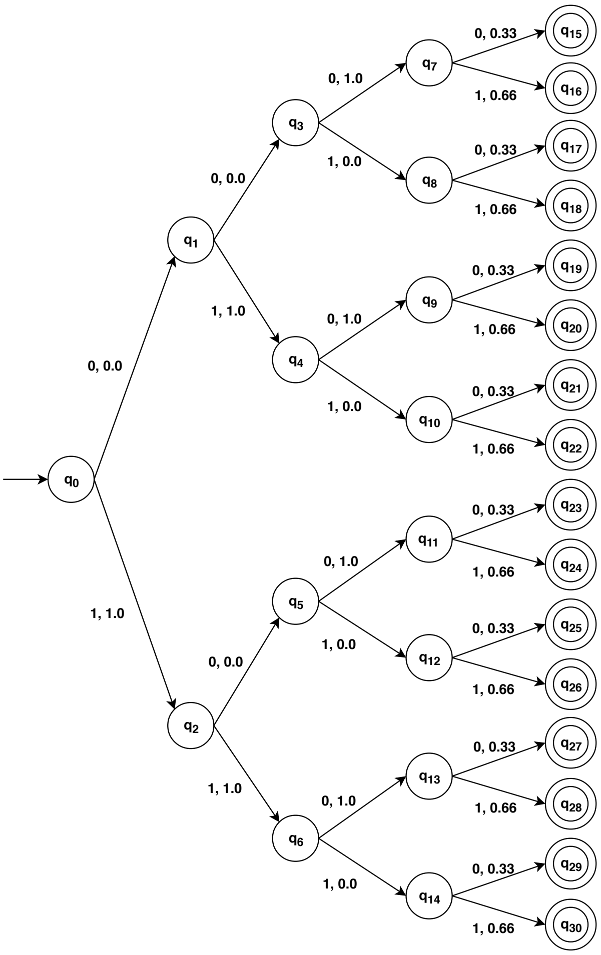


Figure 4.9: Probabilistic Bit Signature as a State Transition Machine

Each transition $\delta_i \in \delta$ is a six tuple $\langle q_i, q_j, c, \sigma, \phi(c_i), Inc(c_j)\rangle$ with the elements representing

The transition probabilities of the machine are governed by the following Equation 4.1.

$$\sum_{\sigma \in \Sigma} \delta(q_i, \sigma) = 1 \tag{4.1}$$

The $PCDA$ is a machine which generates a set of strings starting from start state to an accepting state. The set of strings generated by the $PCDA$ is the language generated by it and is denoted as $\mathcal{L} \subset \Sigma^*$. The string generation can be defined as a recursive process. For example the string with $S = \sigma_1, \sigma_2, \cdots, \sigma_n$ can be generated as $\delta^*(q_0, \sigma_1, \sigma_2, \cdots, \sigma_n)$ where δ^* denotes the extended transition function which is defined recursively as $\delta(\delta^*(q_0, \sigma_1, \sigma_2, \cdots, \sigma_{n-1}), \sigma_n)$. While generating a string if q_i is

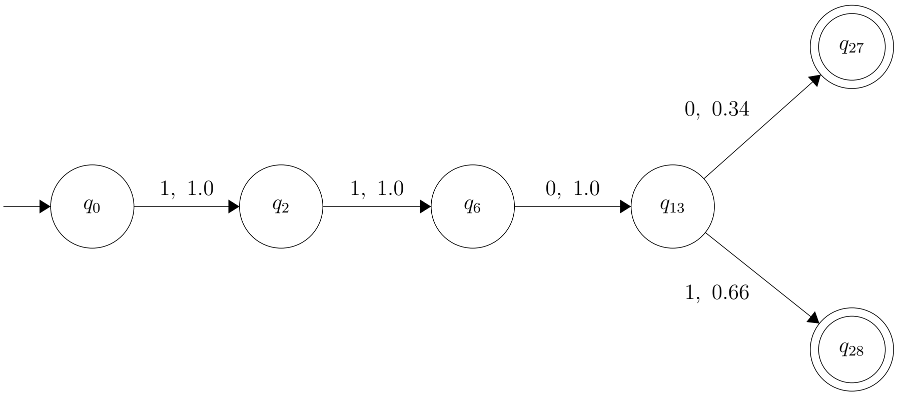


Figure 4.10: Pruned State Transition Diagram for First 4 Bits

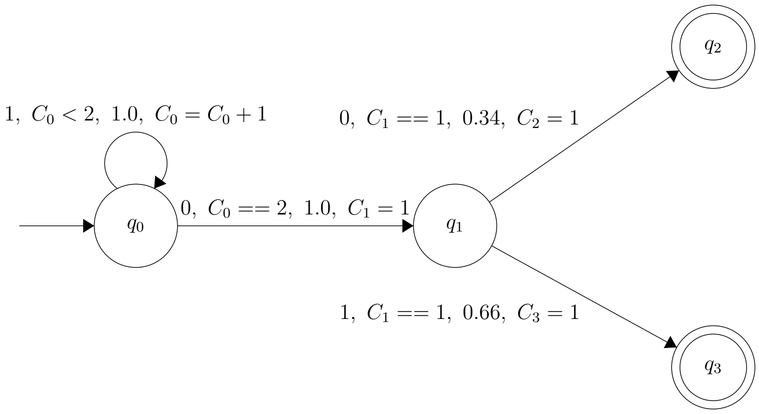

1, $C_0 < 2$, 1.0, $C_0 = C_0 + 1$
q_0
0, $C_0 == 2$, 1.0, $C_1 = 1$
q_1
0, $C_1 == 1$, 0.34, $C_2 = 1$
q_2
1, $C_1 == 1$, 0.66, $C_3 = 1$
q_3

Figure 4.11: Pruned and Merged State Transition Diagram for First 4 Bits

Q	A finite set of states
$q_0 \in Q$	Is an initial state
Σ	Is a finite set of input symbols
$F \subseteq Q$	is a set of final states
C	Is a finite set of Counters with each $c_i \in C$ taking values in $\mathcal{N} \bigcup 0$
δ	Is a set of transitions defined as $\delta : Q \times C \times \Sigma \rightarrow Q \times (C \rightarrow C)$
P	Is a set of transition probabilities with $p_{ij}(\sigma) \rightarrow \mathbb{R}^+$ corresponding to a transition $\delta(q_i, \sigma) = q_j$ for $\sigma \in \Sigma$

q_i Current state

q_j Next state

c_i Counter value at the state q_i

$\sigma \in \Sigma$ Input symbol

$\phi(c_{)}$ Is a constraint on counter value c_i at state q_i on this transition

$Inc(c_j)$ Is a function which initialize the counter in the next state q_j to a new value

the current state and σ_k is the input symbol, the next state is chosen according to the probability of transitions defined. Thus any string $S \in \Sigma^*$ ($|S| \leq n$) is generated with a probability given by Equation 4.2. Later, during flow classification, we use this probability of a string to decide whether a test string belongs to the application represented by $PCDA$.

$$P(S) = \prod_{i=0}^{n} p(q_i, \sigma_{i+1}) \tag{4.2}$$

The training algorithm ($PCDA$ generation for an application A) is shown in Algorithm 4.3. The inputs given to this algorithm are the network traffic of application A

Algorithm 4.3 *PCDA* Generation for Application *A*

```
Input: Network Traffic of Application $A$
Input: $n$- Number of Bits
Output: $PCDA$ for Application $A$
while There are Packets to Read do
    $pkt \leftarrow ReadPacket()$
    $Flows \leftarrow$ FlowReconstructor($pkt$)
end while
$FlowCount \leftarrow$ GetFlowCount($Flows$)
for $t = 1$ to $FlowCount$ do
    $BitString \leftarrow$ GetBitString($F_{Test}$, $n$)
    $BitProb \leftarrow$ UpdateBitProb($BitString$, $t$)
end for
$PCDA \leftarrow$ ConstructPCDA($BitProb$, $n$)
Return $PCDA$
```

and the value of n for extracting first n bits from each bidirectional flow. The output of this algorithm is the $PCDA$ for application A. This algorithm reads the packets belonging to application A and reconstructs the bidirectional flows. Subsequently, it counts the total number of flows. From each flow, the algorithm extracts $BitString$ which is the first n bits of that flow. After this, the algorithm calculates the probability of each bit at its respective position which is then used to construct *PCDA*. Since our input strings are of finite length with n being the order of 40 to 80 bits (see experiments section), we used a heuristic method to add transitions and states incrementally. *ConstructPCDA* function starts with one start state and reads one bit probability at a time to decide one of the two things, either merge it with existing state with a self loop or create one/two state(s) and subsequently adding one/two transitions to the newly created states.

4.2.3.2 Flow Classification

BitProb classifies network flows as belonging to different applications as shown in Figure 4.12. For this, it uses the probabilistic bit signature represented as *PCDA* in the previous phase. The first phase of classification is to reconstruct the bidirectional flows as in training phase and extract first n bits of binary string. This test binary string is given as input to all the signatures (all *PCDA*s) and is classified as that application to whose *PCDA* it has maximum occurrence probability exceeding the

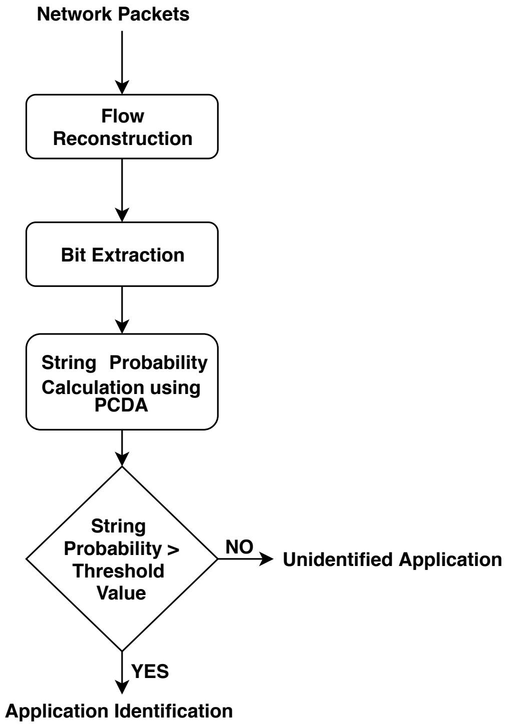

Network Packets
Flow Reconstruction
Bit Extraction
String Probability Calculation using PCDA
String Probability > Threshold Value
NO
Unidentified Application
YES
Application Identification

Figure 4.12: Flow Classification in *BitProb*

threshold probability for that application. This classification procedure is shown in Algorithm 4.4. This algorithm classifies the network flow under consideration. The

Algorithm 4.4 Classifying Network Flows

```
Input: $LIST_{PCDA}$ - List of $PCDA$s for all applications
Input: $A_\delta$ - Probability threshold vector for all applications
Input: $F_{Test}$- Test Flow
Output: Label of $F_{Test}$

$Size \leftarrow$ SizeOf($LIST_{PCDA}$)
$BitString \leftarrow$ GetBitString($F_{Test}$)
for t=1 to $Size$ do
  $ProbS \leftarrow$ CalculateProb($BitString$, $LIST_{PCDA}[t]$)
  if $t == 1$ then
    $CurrentProb \leftarrow ProbS$
    $CurrentApp \leftarrow t$
  else if $ProbS > CurrentProb$ then
    $CurrentProb \leftarrow ProbS$
    $CurrentApp \leftarrow t$
  end if
end for
if $CurrentProb \geq A_\delta[t]$ then
  Label $F_{Test}$ is of Application $t$
else
  Label $F_{Test}$ as Unknown Application
end if
```

inputs given to this algorithm are the list of $PCDAs$ and probability threshold vector A_δ for all applications and a test flow. The output of this algorithm is the labeled test flow. This algorithm first counts the number of $PCDAs$ generated by Algorithm 4.3 and then extracts *BitString* in the same way it is extracted in Algorithm 4.3. Subsequently, the algorithm calculates the string probability value *ProbS* for the test flow as in Equation 4.2 using each $PCDA$ over *BitString*. It calculates the *ProbS* values of the test flow with every $PCDA$ and remembers its highest value and associated $PCDA$. If this *ProbS* value is greater than the threshold value of that application, current flow is classified as belonging to that application. Otherwise, the algorithm declares the test flow as unclassified (unknown application).

4.2.3.3 Complexity Analysis of *BitProb*

Our proposed traffic classification method *BitProb* has different modules such that some complexity is associated to each of them. Table 4.3 shows the complexity of each module of *BitProb* which is also indicating the steps done in training (offline) mode and testing (online) mode.

1. Flow Reconstruction: For bidirectional flow reconstruction, *BitProb* has to inspect each packet header and add it into appropriate flow. If there are l flows and p packets, then it has a complexity of O($l \times p$).

2. Bit Extraction: Since extracting first n bits from the reconstructed payload is a constant time operation, the complexity of bit extraction is O(1).

Table 4.3: Module-wise Complexity of *BitProb*

Module Name	Complexity	Online/Offline	Explanation
Flow Reconstruction	O($l \times p$)	Both	l= number of flows, p = number of packets in the trace.
Bits Extraction	O(1)	Both	Constant number of bits to be read from the payload training flow
Bit Signature Generation	O(K)	Offline	For K flows of an application
$PCDA$ Generation	O(X)	Offline	Generate $PCDA$ for X applications
Flow Classification	O($m \times n$)	Online	m= number of machines and n = number of bits in the payload of test flow

3. Bit Signature Generation: Bit signature generation module has to operate on all the flows of an application. Thus, if there are K flows of an application, complexity of signature generation is $O(K)$.

4. *PCDA* Generation: $PCDA$ generation module generates $PCDA$ from the bits and their respective probabilities which can be done in constant time $O(1)$ and as there are X applications, total complexity is $O(X)$.

5. Flow Classification: Test flows are classified by $PCDA$ and it has to match every bit with every possible instance of $PCDA$. This, in the worst case, has a complexity of O($m \times n$) where m is number of machine instances and n is number of bits in a flow.

4.3 Experiments and Results

To evaluate the classification performance of *BitCoding* and *BitProb*, we used three datasets. One dataset was generated in our testbed setup while the other two datasets are publicly available. In this section, we first describe how the first dataset was generated in our testbed setup and present the details of other two public datasets.

Subsequently, we present the experiments conducted to evaluate the classification performance of *BitCoding* and *BitProb* and report the obtained results.

4.3.1 Dataset Description

For our experiments with *BitCoding* and *BitProb*, we used three different datasets containing flows belonging to 20 different application layer protocols listed in Table 4.4 along with their types (text/binary, open/proprietary). The first dataset was generated by adding flows belonging to SMTP and DNS protocols to the private dataset described in chapter 3. SMTP and DNS flows were generated using the same testbed setup we used in chapter 3 to generate dataset. Generation of SMTP and DNS network traffic from the testbed setup is described below.

Table 4.4: Application Protocols used in the Experiments

Abbreviation	Protocol	TCP/UDP	Type	Proprietariness
BACnet	Building Automation and Control network	UDP	Binary	ASHRAE
BitTorrent	Bit torrent protocol	TCP	Text	No
BJNP	Used to communicate with printer	UDP	Binary	Canon
BOOTP	Bootstrap protocol	UDP	Binary	No
CUPS	Common Unix Printing System	UDP	Text	Apple Inc.
DNS	Domain Name System	UDP	Binary	No
Dropbox	Dropbox LAN Sync protocol	UDP	Text	Dropbox
GsmIp	GSM over Internet protocol	TCP	Text	No
HTTP	Hyper Text Transfer Protocol	TCP	Text	No
Kerberos	Kerberos protocol	UDP	Binary	No
MWBP	Microsoft Windows Browsing Protocol	UDP	Text	Microsoft
NBNS	NetBIOS Name Service	UDP	Binary	No
NBSS	NetBIOS Session Service	TCP	Binary	No
NTP	Network Time Protocol	UDP	Binary	No
POP	Post Office Protocol	TCP	Text	No
QUIC	Quick UDP Internet Connections	UDP	Binary	No
RPC	Remote Procedure Call	TCP	Binary	No
SIP	Session Initiation Protocol	UDP	Text	No
SMTP	Simple Mail Transfer Protocol	TCP	Text	No
SSH	Secure Shell	TCP	Binary	No

1. SMTP: We configured Postfix SMTP server [22] with default configurations on one of the system of our testbed setup. Using this server, mails were exchanged and the resulted traffic was collected at server using tcpdump [24] with SMTP as filter.

2. DNS: We ran tcpdump [24] on all the three systems of the setup for about an hour with DNS as protocol filter. The network traces captured from each system was then merged using mergecap [14].

The flows belonging to these two protocols were captured in the form of *pcap* traces and merged to the private dataset described in chapter 3 which contained flows of BitTorrent, Dropbox, HTTP, SIP and SSH protocols. Thus, the resulting *pcap* trace contained flows of seven protocols, thereby, forming the first dataset which was to be used for our experiments. Second and third datasets are publicly available. One of this public dataset is from 'Digital Corpora' [5] and the other one is provided by 'FOI Information Warfare Lab' [17]. For the subsequent discussion, we name the first, second and third datasets as Private, Public-1 and Public-2 datasets respectively. In the next two subsections, we describe the experiments performed to evaluate the classification performance of *BitCoding* and *BitProb* respectively.

4.3.2 Evaluation of ***BitCoding***

To evaluate the classification performance of *BitCoding*, we divided each dataset into nearly equal parts (approximately 50% for every protocol) and used the first part for generating signatures and the other part for testing the detection performance. The statistics of flows in each case (training and testing) for every protocol and for each dataset are shown in the Tables 4.5, 4.6 and 4.7.

Table 4.5: Private Dataset Statistics

Protocol	**TCP/UDP**	**Flows for Training**	**Size (in MB)**	**Flows for Testing**	**Size (in MB)**
BitTorrent	TCP	01578	245.8	01582	150.4
DNS	UDP	65158	005.7	65528	005.7
Dropbox	UDP	02256	098.2	02276	153.4
HTTP	TCP	97668	220.4	97756	328.3
SIP	UDP	01218	194.1	01280	191.4
SMTP	TCP	01194	010.1	01216	022.9
SSH	TCP	02208	006.2	02212	006.2
Total	-	**171280**	**780.5**	**171850**	**858.3**

We implemented *BitCoding* as a standalone Java program with jNetPcap programming library [10]. We used training portion of each protocol taken from each of the three datasets and generated signatures for every protocol. These signatures were then used for classification of traffic flows. We used *Recall* as a measure of

Table 4.6: Public-1 Dataset Statistics

Protocol	TCP/UDP	Flows for Training	Size (in MB)	Flows for Testing	Size (in MB)
BACnet	UDP	00018	000.097	00022	000.074
BJNP	UDP	00068	000.026	00076	000.031
BOOTP	UDP	00162	004.400	00172	004.500
CUPS	UDP	00090	000.107	00094	000.218
DNS	UDP	50938	012.900	51700	011.100
Dropbox	UDP	00050	000.109	00052	000.319
HTTP	TCP	35928	151.100	35936	133.600
MWBP	UDP	00016	000.565	00014	000.574
NBNS	UDP	01964	007.800	01964	007.500
NTP	UDP	00402	000.652	00402	000.141
QUIC	UDP	00186	000.110	00254	000.115
SMTP	TCP	01040	010.100	01042	009.900
Total	-	**90862**	**187.996**	**91728**	**168.072**

Table 4.7: Public-2 Dataset Statistics

Protocol	TCP/UDP	Flows for Training	Size (in MB)	Flows for Testing	Size (in MB)
BOOTP	UDP	0182	00.080	0182	0.096
DNS	UDP	1926	00.865	1916	1.200
GsmIp	TCP	0018	00.007	0018	0.015
HTTP	TCP	0514	04.800	0506	9.000
Kerberos	UDP	1338	01.600	1344	1.900
NBNS	UDP	0578	00.853	0580	0.680
NBSS	TCP	0754	02.700	0746	3.900
NTP	UDP	0400	00.145	0404	0.648
POP	TCP	0112	00.035	0114	0.036
RPC	TCP	0014	00.020	0014	0.141
Total	-	**5836**	**11.105**	**5824**	**17.616**

classification performance for *BitCoding*. We performed three experiments with the above three datasets to assess the detection performance of *BitCoding*. In our experiments, we selected first 40 ($n = 40$) bits of the flow for generating signatures. Another point worth noting with *BitCoding* is that if there are few flows which are corrupted or incomplete, these may influence the signature generation due to mismatch in bit positions. One way to address this issue is to filter such flows from training set or use a threshold on percentage of times a particular bit is either 1 or 0. In our experiments, we used second approach by setting threshold to 99%[4]. The three experiments and results obtained are furnished below.

1. Homogeneous Experiments: In the homogeneous experiments, we used the training portion of each of the three datasets (Private, Public-1 and Public-2) to generate signatures for each of the protocols. The other portion of the same dataset

[4] All results shown in this chapter are with this threshold, although there are only few protocols which had incomplete/corrupted flows.

was used as the testing dataset. The *Recall* obtained for each protocol is shown in Table 4.8.

Table 4.8: *Recall* for Homogeneous Experiments

(a) Private Dataset

Protocol	*Recall* (in %)
BitTorrent	100.00
DNS	099.69
Dropbox	100.00
HTTP	099.23
SIP	097.97
SMTP	100.00
SSH	100.00

(b) Public-1 Dataset

Protocol	*Recall* (in %)
BACnet	095.45
BJNP	100.00
BOOTP	100.00
CUPS	093.61
DNS	099.93
Dropbox	100.00
HTTP	097.54
MWBP	100.00
QUIC	100.00
NBNS	099.59
NTP	100.00
SMTP	100.00

(c) Public-2 Dataset

Protocol	*Recall* (in %)
BOOTP	100.00
DNS	099.79
GsmIp	100.00
HTTP	100.00
Kerberos	100.00
NBNS	098.97
NBSS	098.91
NTP	100.00
POP	100.00
RPC	100.00

From these results we can notice that, *BitCoding* has an average rate of *Recall* greater than 99%. This experiment suggests that *BitCoding* performs very well if the testing dataset is from the same site as that of training dataset.

2. Heterogeneous Experiments: In heterogeneous set of experiments, we used the training portion of each dataset to generate signatures for application protocols in that dataset. The testing portion of the other two datasets are used to calculate the *Recall*. The idea is to assess robustness of signatures generated in detecting applications when presented with dataset collected from other sites[5]. Tables 4.9, 4.10 and 4.11 show the *Recall* for the cases where the signatures were generated with only the training portion of the Private, Public-1 and Public-2 datasets and tested with testing portion of the other two datasets respectively. It is worth noting here that, in this set of experiments testing is done only for the overlapping set of protocols (training and testing). This experiment gives an idea of how robust the generated signatures are "in not including site specific information in the signatures". For example, some protocols include the name of hosts, user names, etc in the communication. If the training data is collected from a particular site there is a chance that these keywords also be part of invariant bits of signature which should be

[5] All three datasets are collected from different sites

Table 4.9: *Recall* for Training with Private Dataset and Testing with Public-1 and Public-2 Datasets

(a) Public-1 Dataset

Protocol	***Recall* (in%)**
DNS	100.00
Dropbox	100.00
HTTP	099.52
SMTP	100.00

(b) Public-2 Dataset

Protocol	***Recall* (in%)**
DNS	095.19
HTTP	093.28

Table 4.10: *Recall* for Training with Public-1 Dataset and Testing with Private and Public-2 Datasets

(a) Private Dataset

Protocol	***Recall* (in%)**
DNS	096.39
Dropbox	100.00
HTTP	095.96
SMTP	100.00

(b) Public-2 Dataset

Protocol	***Recall* (in%)**
BOOTP	100.00
DNS	094.99
HTTP	091.69
NBNS	099.68
NTP	100.00

Table 4.11: *Recall* for Training with Public-2 Dataset and Testing with Private and Public-1 Datasets

(a) Private Dataset

Protocol	***Recall* (in%)**
DNS	096.39
HTTP	093.89

(b) Public-1 Dataset

Protocol	***Recall* (in%)**
BOOTP	090.69
DNS	099.93
HTTP	099.93
NBNS	099.59
NTP	099.51

avoided. We can notice from the three tables that in all the cases of cross evaluation, *Recall* is over 90% (with many having 95 to 100%) which indicates that the generated signatures are indeed robust (with little compromise in performance) in detecting applications when presented with data from other sites.

3. Grand Experiments: In grand experiments, we merged the training portion of all the protocols of all the three datasets and generated a grand training dataset. This grand training dataset was used to generate the signatures for each protocol. For testing purpose, we used the three dataset's respective testing portions and also a combined dataset (of all the three testing portions) to evaluate the performance. The signatures generated after the run length encoding (*RLE*) in this case are shown in Table 4.12. These signatures are subsequently converted into $TCCA$. One such $TCCA$ for DNS protocol is shown in Figure 4.13. The time taken to generate the signatures and converting them to $TCCA$ in each case are also shown in Table 4.13[6]. The time taken to generate the signatures depends on the number of flows in the training dataset as from every flow bits are extracted and processed to derive

Table 4.12: Signatures Generated for Different Protocols with $n = 40$ Bits in Grand Experiments

Protocol	**Signature**
BACnet	1W6Z1W4Z4*11Z1W1*1Z1*1Z1*1Z1*1Z2*1Z1*
BJNP	1Z1W1Z1W5Z1W2Z3W2Z1W2Z1W1Z1W2Z1W1Z1W11Z1W
BitTorrent	3Z1W2Z2W1Z1W4Z1W2Z2W1Z1W2Z1W1Z3W1Z1W3Z1W1Z1W1Z1W2Z
BOOTP	6Z2*7Z1W5Z2W8Z9*
CUPS	2Z2W2*1Z1*2Z2W1*4Z1*1W5*1Z1*1W5*1Z1*1W1*1Z3*
DNS	17*6Z1W1*15Z
Dropbox	1Z4W1Z2W2Z1W3Z1W2Z2W1Z1W4Z2W1Z4W1Z3W2Z2W
GsmIp	1Z1W1Z2W1Z1W2Z1W4Z1W2Z1W1Z2W4Z1W3Z1W9Z1W
HTTP	1Z1W1Z5*1Z1W2Z1*1W1*1W1Z1W1Z5*1Z6*2Z2*1Z4*
Kerberos	1Z8*4Z2*5Z20*
MWBP	3Z1W3Z1W4Z1*1Z1W1Z16*2W6Z
NBNS	17*1Z1*1Z2*1Z1W3Z1*12Z
NBSS	1*6Z1*16Z16*
NTP	7*1W5Z3*3Z13*8Z
POP	2Z1W1Z1W1Z2W1Z1W2Z4W1Z1W2Z1W1Z2W2Z1W6Z1W2Z1*1W2Z
QUIC	1Z15*6Z1W1Z16*
RPC	1W24Z1W4*2Z2W6*
SIP	1Z1W1Z5*1Z1W1Z5*1Z1W1Z1*1Z3*1Z2*1Z1W2*1W1Z7*
SMTP	2Z2W2Z1W3Z2W2Z1W3Z2W6Z1W6Z1*1W1Z3*1Z
SSH	1Z1W1Z1W2Z2W1Z1W1Z1W2Z2W1Z1W2Z1W5Z1W1Z2W1Z1W2Z2W2Z1W1Z

[6] We show the signatures and $TCCA$ for this case as this experiment is comprehensive

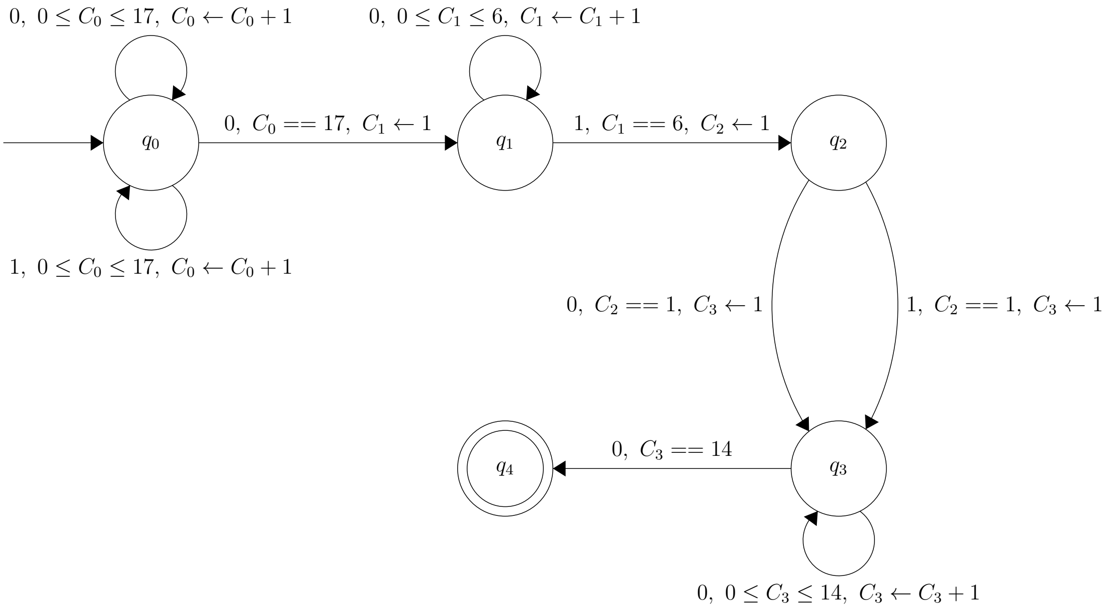

$0,\ 0 \le C_0 \le 17,\ C_0 \leftarrow C_0 + 1$
$0,\ 0 \le C_1 \le 6,\ C_1 \leftarrow C_1 + 1$
q_0
$0,\ C_0 == 17,\ C_1 \leftarrow 1$
q_1
$1,\ C_1 == 6,\ C_2 \leftarrow 1$
q_2
$1,\ 0 \le C_0 \le 17,\ C_0 \leftarrow C_0 + 1$
$0,\ C_2 == 1,\ C_3 \leftarrow 1$
$1,\ C_2 == 1,\ C_3 \leftarrow 1$
q_4
$0,\ C_3 == 14$
q_3
$0,\ 0 \le C_3 \le 14,\ C_3 \leftarrow C_3 + 1$

Figure 4.13: *TCCA* Generated for DNS with 40 Bit Length Compressed Signature "17*6Z1W1*15Z"

Table 4.13: Signature Generation Time for Grand Experiments

Protocol	**Training time (in seconds)**
BACnet	00.445
BJNP	00.414
BitTorrent	04.018
BOOTP	00.704
CUPS	00.500
DNS	34.500
Dropbox	03.367
GsmIp	00.288
HTTP	46.416
Kerberos	01.226
MWBP	00.373
NBNS	01.820
NBSS	00.871
NTP	00.815
POP	00.368
QUIC	00.493
RPC	00.237
SIP	03.629
SMTP	01.587
SSH	01.502
Total time	**103.573**

the signature. We can notice that *BitCoding* is very fast in processing flows and generating signatures. Even for HTTP with 134110 number of flows in training set, it is taking less than a minute to generate the signature. Table 4.14 shows the *Recall* rate for four different experiments. We can notice that the results are similar to the previous cases (with many having *Recall* close to 100%). We can also notice that, *Recall* has improved for many protocols compared to heterogeneous experiments, indicating signatures are more robust.

Table 4.14: *Recall* for Training with Grand Dataset

Protocol	**Grand Dataset**	**Private Dataset**	**Public-1 Dataset**	**Public-2 Dataset**
BACnet	095.45	-	095.45	-
BJNP	100.00	-	100.00	-
BitTorrent	100.00	100.00	-	-
BOOTP	100.00	-	100.00	100.00
CUPS	093.61	-	093.61	-
DNS	099.75	099.69	100.00	097.91
Dropbox	100.00	100.00	100.00	-
GsmIp	100.00	-	-	100.00
HTTP	098.18	098.44	097.42	091.69
Kerberos	100.00	-	-	100.00
MWBP	100.00	-	100.00	-
NBNS	099.44	-	099.69	098.96
NBSS	098.90	-	-	98.90
NTP	100.00	-	100.00	100.00
POP	100.00	-	-	100.00
QUIC	100.00	-	100.00	-
RPC	100.00	-	-	100.00
SIP	097.96	097.96	-	-
SMTP	100.00	100.00	100.00	-
SSH	100.00	100.00	-	-

4.3.2.1 Measuring Robustness of Signatures with Cross Application Flows

As discussed earlier in Section 4.3, signatures of one application might match with the flows of other applications. The chances of this cross signature matching increases as number of protocols increase particularly with short signature lengths. To understand the robustness and uniqueness of signatures generated by *BitCoding*, we performed an experiment with $n \times n$ signature matching. In this experiment, we evaluated the flows of one application protocol with signatures of all other protocols. For this experiment, we used the grand experiments dataset (both training and testing). The results of these experiments are summarized in the Table 4.15. The rows in the table show the number of flows of type, indicated in the column, matched against the signature of protocol in the row. For example, the third column BitTorrent has 1582 number of flows in testing dataset and 1582 of them are matched with BitTorrent signature. Similarly the second row first column has a value of 0 indicating none of BJNP flows are matched against signatures of BACnet. Similar interpretation was done for other cases too. We can notice that there are indeed some cases where signatures of one protocol match with flows of other protocols (shown in red colour). For example, out of 119144 flows of DNS, 116646 flows match with NBNS protocol.

As described in Section 4.3, we identified *weak signatures* by computing the *Relaxed Hamming Distance* (*RHD*) between the signatures generated. We computed the *RHD* values between every pair of application protocol signatures generated. The *RHD* values obtained are shown in Table 4.16. We can notice that most of the cross signature matching happened with those applications whose signatures have a zero RHD values (Ex. DNS and Kerberos). However there were few cross signature matching even with non-zero distance values too (28 HTTP flows matching with CUPS). We manually screened these flows and noticed that these were either incomplete or corrupted just like the ones found in training set.

4.3.2.2 Effect of Number of Bits and Number of Flows on Signature Quality

The quality of generated signatures is governed by two important parameters - *Number of bits* and *Number of flows.* In this section, we study the impact of these two parameters on the quality of signatures generated. First study is on increasing the number of bits used for signature generation and second is the number of flows used for signature generation. These two are elaborated below.

1. Effect of Increasing the Number of Bits: As we noticed in the last section (with $n \times n$ evaluation experiment and *RHD* calculation), there are indeed few cases where flows of one type of protocol are matched with signatures of other type of protocol leading to incorrect classification or at-best a guess of application

Table 4.15: Matrix for $n \times n$ Evaluation Scenario with $n = 40$ Bits

	BACnet	**BJNP**	**BitTorrent**	**BOOTP**	**CUPS**	**DNS**	**Dropbox**	**GsmIp**	**HTTP**	**Kerberos**	**MWBP**	**NBNS**	**NBSS**	**NTP**	**POP**	**QUIC**	**RPC**	**SIP**	**SMTP**	**SSH**	**Not**
	(22)	(76)	(1582)	(354)	(94)	(119144)	(2328)	(18)	(134198)	(1344)	(14)	(2544)	(728)	(806)	(114)	(254)	(14)	(1280)	(2258)	(2212)	**Classified**
BACnet	21	0	0	0	0	16	0	0	0	0	0	14	0	0	0	0	0	0	0	0	1
BJNP	0	76	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BitTorrent	0	0	1582	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BOOTP	0	0	0	354	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CUPS	0	0	0	0	88	0	0	0	28	0	0	0	0	0	0	0	0	0	0	0	6
DNS	0	0	0	0	0	118848	0	0	0	0	0	0	0	0	0	0	0	0	0	0	296
Dropbox	0	0	0	0	0	0	2328	0	0	0	0	0	0	0	0	0	0	0	0	0	0
GsmIp	0	0	0	0	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0
HTTP	0	0	0	0	0	0	0	0	131756	0	0	0	0	0	0	0	0	0	0	0	2442
Kerberos	0	0	0	0	0	1710	0	0	0	1344	0	36	0	408	0	0	0	0	0	0	0
MWBP	0	0	0	0	0	4	0	0	0	0	14	0	0	0	0	0	0	0	0	0	0
NBNS	0	0	0	0	0	116646	0	0	0	0	0	2530	0	0	0	0	0	0	0	0	14
NBSS	0	0	0	0	0	0	0	0	0	0	0	0	720	0	0	0	0	0	0	0	8
NTP	0	0	0	0	0	1792	0	0	0	0	0	0	42	806	0	0	0	0	0	0	0
POP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	114	0	0	0	0	0	0
QUIC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	254	0	0	0	0	0
RPC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	0	0	0	0
SIP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1254	0	0	26
SMTP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2258	0	0
SSH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2212	0

Table 4.16: Matrix for $n \times n$ Relaxed Hamming Distance with $n = 40$ Bits

	BACnet	**BJNP**	**BitTorrent**	**BOOTP**	**CUPS**	**DNS**	**Dropbox**	**GsmIp**	**HTTP**	**Kerberos**	**MWBP**	**NBNS**	**NBSS**	**NTP**	**POP**	**QUIC**	**RPC**	**SIP**	**SMTP**	**SSH**
BACnet	-	9	13	4	7	2	18	13	4	1	3	1	0	0	12	2	3	6	11	14
BJNP	9	-	14	12	10	7	20	8	1	3	7	5	9	5	14	2	12	4	18	17
BitTorrent	13	14	-	14	7	10	14	12	5	3	5	7	8	6	12	5	11	6	14	13
BOOTP	4	12	14	-	7	3	17	13	4	1	3	2	3	0	11	1	5	4	11	12
CUPS	7	10	7	7	-	3	4	10	8	3	6	2	5	4	10	1	7	5	2	8
DNS	2	7	10	3	3	-	15	7	1	0	2	0	1	0	6	2	4	3	5	10
Dropbox	18	20	14	17	4	15	-	14	8	3	10	13	10	8	20	4	15	5	14	11
GsmIp	13	8	12	13	10	7	14	-	2	3	8	6	9	4	16	4	12	3	14	13
HTTP	4	1	5	4	8	1	8	2	-	3	5	1	5	2	2	1	7	1	9	4
Kerberos	1	3	1	1	3	0	3	3	3	-	0	0	0	0	4	0	1	2	4	4
MWBP	3	7	5	3	6	2	10	8	5	0	-	2	2	2	9	0	4	3	7	10
NBNS	1	5	7	2	2	0	13	6	1	0	2	-	1	0	5	2	3	3	4	9
NBSS	0	9	8	3	5	1	10	9	5	0	2	1	-	0	12	1	0	3	8	9
NTP	0	5	6	0	4	0	8	4	2	0	2	0	0	-	5	0	3	2	5	6
POP	12	14	12	11	10	6	20	16	2	4	9	5	12	5	-	3	16	5	16	17
QUIC	2	2	5	1	1	2	4	4	1	0	0	2	1	0	3	-	2	1	3	3
RPC	3	12	11	5	7	4	15	12	7	1	4	3	0	3	16	2	-	6	5	11
SIP	6	4	6	4	5	3	5	3	1	2	3	3	3	2	5	1	6	-	8	1
SMTP	11	18	14	11	2	5	14	14	9	4	7	4	8	5	16	3	11	8	-	14
SSH	14	17	13	12	8	10	11	13	4	4	10	9	9	6	17	3	15	1	14	-

(matching with more than application signature). In this section, we provide results of the experiments done to find the effect of increasing the signature bits. We performed experiments by increasing the number of bits from 40 to 80 and 120 and evaluated the number of cross signature matches for those cases which showed cross matching in previous experiment ($n \times n$ evaluation). Table 4.17 shows the new signatures generated with 80 bits of payload extract. Table 4.18 shows the number of signatures matched for the protocol type which had issue with 40 bits signature. We can notice that even with 80 bits too, the cross signature matching continued, however the number of cross matching cases decreased as compared to the previous case. Similarly, Table 4.19 shows the signatures generated with 120 bits of payload extract (for only those cases which had cross matching even with 80 bits signature) and Table 4.20 shows the cross signature matching performance. We can notice that with signature length of 120 bits, the cross signature matching issue is completely addressed. We computed the RHD values between all pairs of protocols and the distances are shown in Table 4.21.We can notice that there are no zero RHD values in this case which also justifies the no cross signature matching. With this experiment, we can conclude that for different protocols, we need signatures of different lengths.

Table 4.17: Signatures Generated for Different Protocols with $n = 80$ Bits

Protocol	**Signature**
BACnet	1W6Z1W4Z4*11Z1W1*1Z1*1Z1*1Z1*1Z2*1Z1*2Z2*1Z1*2Z2*3W1*3W1*1W1*4W1*1Z3*1Z1*1Z2W6*
CUPS	2Z2W2*1Z1*2Z2W1*4Z1*1W5*1Z1*1W5*1Z1*1W1*1Z3*1Z1*1W1*1Z3*1Z1*1W2Z1*1Z1*1Z1*1W2*1Z2*1Z1*1W1*5Z1* 1W2*2Z1*
HTTP	1Z1W1Z5*1Z1W2Z1*1W1*1W1Z1W1Z5*1Z6*2Z2*1Z4*1Z7*1Z7*1Z7*1Z7*
DNS	17*6Z1W1*22Z1W14Z1*17Z
Kerberos	1Z8*4Z2*5Z31*1Z28*
MWBP	3Z1W3Z1W4Z1*1Z1W1Z16*2W6Z1W1Z1W1Z1W5Z1*1Z1*1Z3*1Z1*1Z4*8Z1W3Z1W1Z1W1Z
NBNS	17*1Z1*1Z2*1Z1W3Z1*19Z1*15Z1*16Z
NTP	7*1W5Z3*3Z13*15Z1*2Z14*15Z1*

Table 4.18: Matrix for $n \times n$ Evaluation Scenario with $n = 80$ Bits

	BACnet	**CUPS**	**DNS**	**HTTP**	**Kerberos**	**MWBP**	**NBNS**	**NTP**	**Unclassified**
BACnet	20	0	0	0	0	0	0	0	0
CUPS	0	94	0	14	0	0	0	0	0
DNS	0	0	117400	0	0	0	0	0	1744
HTTP	0	0	0	131756	0	0	0	0	2442
Kerberos	0	0	1710	0	1338	0	36	382	6
MWBP	0	0	0	0	0	14	0	0	0
NBNS	0	0	116646	0	0	0	2530	0	14
NTP	0	0	1792	0	0	0	42	796	10

Table 4.19: Signatures Generated for Different Protocols with n = 120 Bits

Protocol	**Signature**
CUPS	2Z2W2*1Z1*2Z2W1*4Z1*1W5*1Z1*1W5*1Z1*1W1*1Z3*1Z1*1W1*1Z3*1Z1*1W2Z1*1Z1*1Z1*1W2*1Z2*1Z1*1W1*5Z1*1W2*2Z1*1Z1*1W2*1Z2*1Z1*1W5*1Z1*1W5*2Z1W5*2Z1W2*3W
DNS	17*6Z1W1*22Z1W14Z1*36Z5*1Z7*1Z7*
Kerberos	1Z8*4Z2*5Z31*1Z39*1Z5*1Z1*2Z4*1Z3*1Z2*5Z3*
NBNS	17*1Z1*1Z2*1Z1W3Z1*19Z1*15Z1*31Z1*2Z1W6Z1W3Z3*1Z1W1Z5*
NTP	7*1W5Z3*3Z13*15Z1*2Z14*15Z18*1Z10*1Z3*3Z1*2Z2*

Table 4.20: Matrix for $n \times n$ Evaluation Scenario with n =120 Bits

	CUPS	**DNS**	**Kerberos**	**NBNS**	**NTP**	**Unclassified**
CUPS	94	0	0	0	0	0
DNS	0	117374	0	0	0	1770
Kerberos	0	0	1226	0	0	118
NBNS	0	0	0	2530	0	14
NTP	0	0	0	0	788	18

For correct detection of application, this length needs to be empirically found out. One way to address this is to generate a signature for an application and test it for overlap with RHD values and increase the length if there is a overlap. As $BitCoding$ uses RLE to compress the signature bits for efficient representation, we computed the compression ratio achieved by RLE for signatures of different lengths. The compression ratio is the ratio of length of signature in the RLE encoded signature to the length of signature before RLE. This is given by Equation 4.3.

$$CompressionRatio = \frac{Length\ After\ RLE}{Length\ Before\ RLE} \tag{4.3}$$

Figure 4.14 shows the compression ratio achieved by RLE for 5 different protocols[7]. We can notice that increasing the length of signature bits achieve better compression ratio. Similar observations were made for other protocols too.

2. Effect of Number of Flows on Signature Quality: As we generate signatures from the payloads of training application flows, the number of flows present in the training dataset is an important parameter for generating quality signatures. It is a natural question to ask how many flows are required to generate good signatures. To understand this, we performed an experiment with different number of flows to generate the signatures and tested the performance using our grand dataset. Figure

[7] We show for only 5 protocols here for clarity

Table 4.21: Matrix for $n \times n$ Relaxed Hamming Distance with $n = 120$ Bits

	BACnet	**BJNP**	**BitTorrent**	**BOOTP**	**CUPS**	**DNS**	**Dropbox**	**GsmIp**	**HTTP**	**Kerberos**	**MWBP**	**NBNS**	**NBSS**	**NTP**	**POP**	**QUIC**	**RPC**	**SIP**	**SMTP**	**SSH**
BACnet	-	29	35	7	14	16	34	29	13	4	14	17	13	8	30	2	15	23	19	35
BJNP	29	-	54	12	23	9	55	19	20	3	17	10	17	9	54	2	13	29	29	55
BitTorrent	35	54	-	33	17	39	44	47	19	19	39	42	29	33	28	5	42	27	28	45
BOOTP	7	12	33	-	13	4	35	15	9	1	9	3	4	4	30	1	5	17	15	35
CUPS	14	23	17	13	-	13	12	21	11	7	19	14	14	12	23	1	19	12	7	25
DNS	16	9	39	4	13	-	48	11	12	2	13	1	8	6	37	2	7	22	15	29
Dropbox	34	55	44	35	12	48	-	45	22	16	38	49	26	26	50	4	41	22	24	43
GsmIp	29	19	47	15	21	11	45	-	18	7	26	16	19	14	57	4	21	26	26	47
HTTP	13	20	19	9	11	12	22	18	-	5	20	11	10	14	15	4	17	6	15	17
Kerberos	4	3	19	1	7	2	16	7	5	-	6	4	2	1	19	2	1	8	8	20
MWBP	14	17	39	9	19	13	38	26	20	6	-	15	17	9	42	2	11	25	15	41
NBNS	17	10	42	3	14	1	49	16	11	4	15	-	9	5	39	2	9	23	16	43
NBSS	13	17	29	4	14	8	26	19	10	2	17	9	-	5	29	1	4	14	15	30
NTP	8	9	33	4	12	6	26	14	14	1	9	5	5	-	30	2	6	16	16	27
POP	30	54	28	30	23	37	50	57	15	19	42	39	29	30	-	3	47	23	29	47
QUIC	2	2	5	1	1	2	4	4	4	2	2	2	1	2	3	-	2	2	3	3
RPC	15	13	42	5	19	7	41	21	17	1	11	9	4	6	47	2	-	25	18	45
SIP	23	29	27	17	12	22	22	26	6	8	25	23	14	16	23	2	25	-	21	30
SMTP	19	29	28	15	7	15	24	26	15	8	15	16	15	16	29	3	18	21	-	25
SSH	35	55	45	35	25	39	43	47	17	20	41	43	30	27	47	3	45	30	25	-

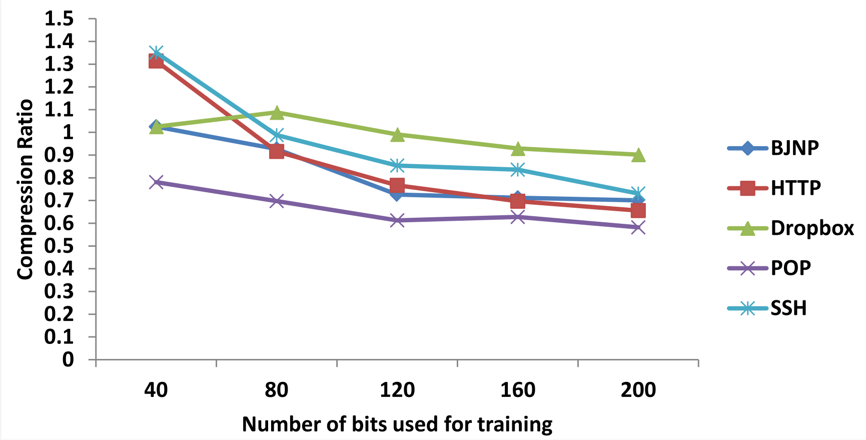


Figure 4.14: Compression Ratio Achieved for Different Protocols

4.15 shows the effect on the *Recall* with different number of flows for four protocols. We can notice that for two of the protocols (Dropbox and SSH) the *Recall* rate is 100% from the beginning. However with other two protocols it gradually increases and approaches to 100% after 256 and 512 number of flows. The same trend is shown with other protocols too. Thus we can infer that, for few protocols, small number of flows are sufficient for generating the signatures and for others relatively more (few hundred) are required to generate good signatures.

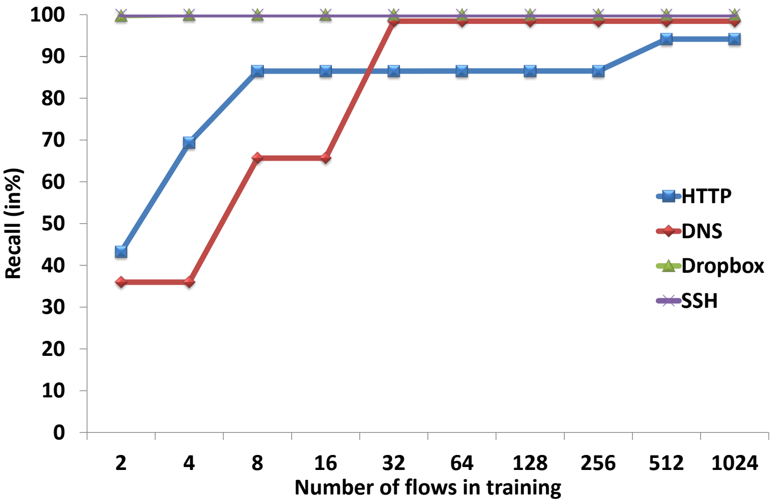


Figure 4.15: Effect of Number of Flows on *Recall*

4.3.3 Evaluation of ***BitProb***

To evaluate the detection performance of *BitProb*, we divided each dataset into nearly two equal halves such that one part was used for training while other part was used for testing purpose. The training part of each dataset was further divided into the ratio of 2:1 where the first part was used for signature generation while second part was used for threshold string probability value calculation. The statistics of dataset division for all three datasets is given in Table 4.22, 4.23 and 4.24.

Table 4.22: Private Dataset Statistics

Protocol	**Flows for Training**	**Size (in MB)**	**Flows for Threshold Setting**	**Size (in MB)**	**Flows for Testing**	**Size (in MB)**
BitTorrent	01041	162.228	00537	083.572	01582	150.400
DNS	43004	003.762	22154	001.938	65528	005.700
Dropbox	01482	065.500	00774	032.700	02276	153.400
HTTP	64460	145.464	33208	074.936	97756	328.300
SIP	00803	128.106	00415	065.994	01280	191.400
SMTP	00788	006.666	00406	003.434	01216	022.900
SSH	01457	004.092	00751	002.108	02212	006.200
Total	**113035**	**515.814**	**58245**	**264.686**	**171850**	**858.300**

Table 4.23: Public Dataset-1 Statistics

Protocol	**Flows for Training**	**Size (in MB)**	**Flows for Threshold Setting**	**Size (in MB)**	**Flows for Testing**	**Size (in MB)**
BACnet	00011	00.064	00007	00.032	00022	00.074
BJNP	00044	00.017	00024	00.008	00076	00.031
Bootp	00103	02.970	00059	01.430	00172	04.500
CUPS	00058	00.072	00032	00.035	00094	00.218
DNS	33619	08.514	17319	04.386	51700	11.100
Dropbox	00032	00.073	00018	00.036	00052	00.319
HTTP	23712	99.726	12216	51.374	35936	133.60
MWBP	00010	00.372	00006	00.192	00014	00.574
NBNS	01296	04.950	00668	02.250	01964	07.500
NTP	00265	00.435	00137	00.217	00402	00.141
QUIC	00124	00.075	00062	00.039	00254	00.115
SMTP	00686	06.666	00354	03.434	01042	09.9000
Total	**59960**	**123.934**	**30902**	**64.062**	**91728**	**168.072**

Table 4.24: Public Dataset-2 Statistics

Protocol	**Flows for Training**	**Size (in MB)**	**Flows for Threshold Setting**	**Size (in MB)**	**Flows for Testing**	**Size (in MB)**
Bootp	0120	0.063	0062	0.031	0182	0.096
DNS	1271	0.570	0655	0.294	1916	1.200
GsmIp	0011	0.004	0007	0.002	0018	0.015
HTTP	0339	3.168	0175	1.632	0506	9.000
Kerberos	0883	1.056	0455	0.544	1344	1.900
NBNS	0380	0.569	0198	0.284	0580	0.680
NBSS	0497	1.782	0257	0.918	0746	3.900
NTP	0262	0.097	0138	0.048	0404	0.648
POP	0073	0.023	0039	0.011	0114	0.036
RPC	0009	0.013	0005	0.006	0014	0.141
Total	**3845**	**7.345**	**1991**	**3.760**	**5824**	**17.616**

We also implemented *BitProb* as a standalone Java program with jNetPcap programming library [10]. We used first (larger) portion of training dataset taken from each of the three datasets and generated signatures for every protocol. We used the second (smaller) portion of the training dataset from each of the three datasets to calculate the threshold string probability value. The testing dataset is then used for classification. We used *Recall* as a measure of classification performance for *BitProb*. *Recall* is the ratio of flows correctly labeled as a particular application to the total number of flows belonging to that application.

Similar to *BitCoding*, we performed three experiments using these datasets to evaluate the classification performance of *BitProb* also. In our experiments, we extracted first 40 bits (n=40) to generate probabilistic signatures and calculated threshold values. We performed three types of experiments namely Homogeneous experiments, Heterogeneous experiments and Grand experiments. The details of these experiments and the results obtained from each experiment are presented below.

1. Homogeneous Experiments: In the homogeneous experiment, we used training part and testing part of same dataset (Private, Public-1 and Public-2) for evaluation. The *Recall* obtained for each application protocol is shown in Table 4.25. We can notice that the average *Recall* is more than 99% for homogeneous experiment.

Table 4.25: *Recall* for Homogeneous Experiments

(a) Private Dataset

Protocol	***Recall* (in %)**
BitTorrent	100.00
DNS	099.97
Dropbox	100.00
HTTP	099.99
SIP	100.00
SMTP	100.00
SSH	100.00

(b) Public-1 Dataset

Protocol	***Recall* (in %)**
BACnet	100.00
BJNP	099.99
BOOTP	100.00
CUPS	100.00
DNS	099.98
Dropbox	100.00
HTTP	099.92
MWBP	100.00
QUIC	100.00
NBNS	099.90
NTP	100.00
SMTP	100.00

(c) Public-2 Dataset

Protocol	***Recall* (in %)**
BOOTP	097.70
DNS	099.34
GsmIp	100.00
HTTP	100.00
Kerberos	100.00
NBNS	100.00
NBSS	100.00
NTP	100.00
POP	100.00
RPC	100.00

The results indicate that *BitProb* showed very high accuracy for homogeneous experiment.

2. Heterogeneous Experiments: In the heterogeneous experiment, we used training part of one dataset whereas we used testing part of other two datasets for evaluation. We performed this experiment to evaluate the robustness of signatures i.e. whether signatures generated from one site can classify the flows of same application protocol on other sites[8] or not. These results are shown in terms of

Table 4.26: *Recall* for Training with Private Dataset and Testing with Public-1 and Public-2 Datasets

(a) Public-1 Dataset

Protocol	***Recall* (in%)**
DNS	100.00
Dropbox	100.00
HTTP	100.00
SMTP	100.00

(b) Public-2 Dataset

Protocol	***Recall* (in%)**
DNS	097.91
HTTP	100.00

Table 4.27: *Recall* for Training with Public-1 Dataset and Testing with Private and Public-2 Datasets

(a) Private Dataset

Protocol	***Recall* (in%)**
DNS	098.54
Dropbox	100.00
HTTP	099.50
SMTP	100.00

(b) Public-2 Dataset

Protocol	***Recall* (in%)**
Bootp	100.00
DNS	095.20
HTTP	098.82
NBNS	100.00
NTP	099.50

Table 4.28: *Recall* for Training with Public-2 Dataset and Testing with Private and Public-1 Datasets

(a) Private Dataset

Protocol	***Recall* (in%)**
DNS	096.40
HTTP	099.89

(b) Public-1 Dataset

Protocol	***Recall* (in%)**
Bootp	095.83
DNS	099.93
HTTP	099.93
NBNS	099.99
NTP	100.00

[8] All three datasets belong to different sites

Recall in Table 4.26, 4.27 and 4.28 where the signatures were generated with only the training portion of the Private, Public-1 and Public-2 datasets and tested with testing portion of the other two datasets respectively. This experiment was important because some protocols contain site information like host machine name, user name, etc. which may become a part of signature and it needs to be avoided. From the results, we can notice that minimum *Recall* obtained is greater than 95% which indicates that the generated signatures are indeed robust (with little compromise in performance) in detecting application protocols when presented with data from other sites.

3. Grand Experiments: In the grand experiment, we merged the training part of all the datasets and generated a grand training dataset. From this dataset, we generated signatures in the form of $PCDA$ for each application protocol. Time taken to generate the signatures for this experiment for different protocols are shown in Table 4.29. The time taken to generate the signatures depends on the number of flows in the training dataset as from every flow bits are extracted and processed to derive the signature. We can notice that *BitProb* is fast having minimum and

Table 4.29: Training time for Application Protocols

Protocol	**Training time (in seconds)**
BACnet	000.571
BitTorrent	037.134
BJNP	001.068
Bootp	001.670
CUPS	000.934
DNS	068.343
Dropbox	014.227
GsmIp	000.654
HTTP	275.363
Kerberos	007.995
MWBP	000.708
NBNS	003.787
NBSS	002.341
NTP	002.145
POP	000.818
QUIC	001.095
RPC	000.062
SIP	010.553
SMTP	003.234
SSH	003.256
Total time	**497.841**

maximum time taken being 0.062 seconds and 275.363 seconds for processing flows and generating signatures. Subsequently, we calculated threshold string probability values for each application protocol. These threshold values are shown in Table 4.30. For testing purpose, we used testing part of all three datasets and also the combinations of all three datasets. *Recall* for the grand experiment is shown in Table 4.31.

Table 4.30: Threshold Values for Application Protocols

Protocol	**Threshold Values**
BACnet	4.1943×10^{-5}
BitTorrent	0.96038263
BJNP	0.91115702
Bootp	4.3783×10^{-5}
CUPS	3.6323×10^{-9}
DNS	1.4471×10^{-17}
Dropbox	0.93933648
GsmIp	1
HTTP	7.2987×10^{-26}
Kerberos	4.1212×10^{-10}
MWBP	8.6973×10^{-6}
NBNS	2.7617×10^{-13}
NBSS	2.7942×10^{-8}
NTP	1.1363×10^{-10}
POP	1
QUIC	1.3882×10^{-11}
RPC	0.82458621
SIP	2.3409×10^{-20}
SMTP	0.00423356
SSH	1

4.3.3.1 Measuring the Uniqueness of Probability Values and Thresholds

In this subsection, we present the experiments performed to understand the uniqueness of string probabilities generated and also threshold values. For this experiment we relaxed the criteria of selecting maximum probability application ($PCDA$) and comparing with the probability threshold of that application. Here for every test application flow, probability value of binary bit string is calculated by giving it as input to every application $PCDA$ and compared with the threshold of that application against whose the string is matched. The idea was to understand if the probability values of two bit strings are of two different applications being closer so as to mislead the classification. The results of these experiments are shown in Table 4.32. The rows

Table 4.31: *Recall* for Training with Grand Dataset

Protocol	Grand Dataset	Private Dataset	Public-1 Dataset	Public-2 Dataset
BACnet	100.00	-	100.00	-
BJNP	099.99	-	099.99	-
BitTorrent	100.00	100.00	-	-
Bootp	097.17	-	093.82	100.00
CUPS	100.00	-	100.00	-
DNS	099.99	100.00	100.00	099.95
Dropbox	100.00	100.00	100.00	-
GsmIp	100.00	-	-	100.00
HTTP	098.19	098.64	097.42	091.69
Kerberos	100.00	-	-	100.00
MWBP	100.00	-	100.00	-
NBNS	099.44	-	099.69	098.96
NBSS	100.00	-	-	100.00
NTP	100.00	-	100.00	100.00
POP	100.00	-	-	100.00
QUIC	100.00	-	100.00	-
RPC	100.00	-	-	100.00
SIP	100.00	100.00	-	-
SMTP	100.00	100.00	100.00	-
SSH	100.00	100.00	-	-

in the table show the number of flows of type indicated in the column are evaluated against the signature ($PCDA$) of protocol in the row. For example, the sixth column DNS has 119144 number of flows in testing dataset and 119142 of them are correctly identified with DNS signature while remaining 2 flows could not be classified as they fell short of meeting the minimum probability thresholds of DNS. Similar interpretation can be done for other cases as well. We can notice that there are indeed some cases where signatures of one protocol match with flows of other protocol (shown in red colour). For example, out of 119144 flows of DNS 254 flows were matched with NBNS protocol. However these numbers are very small and manageable.

4.3.3.2 Effect of Number of Flows on Signature Quality

Similar to the experiments performed in *BitCoding* method, we performed an experiment in *BitProb* also to show the effect of varying number of flows of four protocols on *BitProb*'s *Recall*. The results of this experiment is shown in Figure 4.16. We can notice from this figure that for two of the protocols (Dropbox and SSH), *BitProb*'s *Recall* is 100% independent of the number of flows used to generate signatures. However, for other two protocols, it gradually increases after 256 and 512

Table 4.32: Matrix for $n \times n$ Evaluation Scenario with $n = 40$ Bits

	BACnet	**BJNP**	**BitTorrent**	**Bootp**	**CUPS**	**DNS**	**Dropbox**	**GsmIp**	**HTTP**	**Kerberos**	**MWBP**	**NBNS**	**NBSS**	**NTP**	**POP**	**QUIC**	**RPC**	**SIP**	**SMTP**	**SSH**	**Not**
	(22)	(76)	(1582)	(354)	(94)	(119144)	(2328)	(18)	(134198)	(1344)	(14)	(2544)	(728)	(806)	(114)	(254)	(14)	(1280)	(2258)	(2212)	**Classified**
BACnet	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BJNP	0	74	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
BitTorrent	0	0	1582	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bootp	0	0	0	344	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10
CUPS	0	0	0	0	94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DNS	0	0	0	0	0	119142	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
Dropbox	0	0	0	0	0	0	2328	0	0	0	0	0	0	0	0	0	0	0	0	0	0
GsmIp	0	0	0	0	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0
HTTP	0	0	0	0	0	0	0	0	133952	0	0	0	0	0	0	0	0	0	0	0	246
Kerberos	0	0	0	0	0	914	0	0	2	1344	0	0	0	22	0	0	0	0	0	0	0
MWBP	0	0	0	0	0	0	0	0	0	0	14	0	0	0	0	0	0	0	0	0	0
NBNS	0	0	0	0	0	254	0	0	0	0	0	2530	0	22	0	0	0	0	0	0	14
NBSS	0	0	0	0	0	0	0	0	0	0	0	0	728	0	0	0	0	0	0	0	0
NTP	0	0	0	0	0	8	0	0	0	0	0	0	0	806	0	0	0	0	0	0	0
POP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	114	0	0	0	0	0	0
QUIC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	254	0	0	0	0	0
RPC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	0	0	0	0
SIP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1280	0	0	0
SMTP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2258	0	0
SSH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2212	0

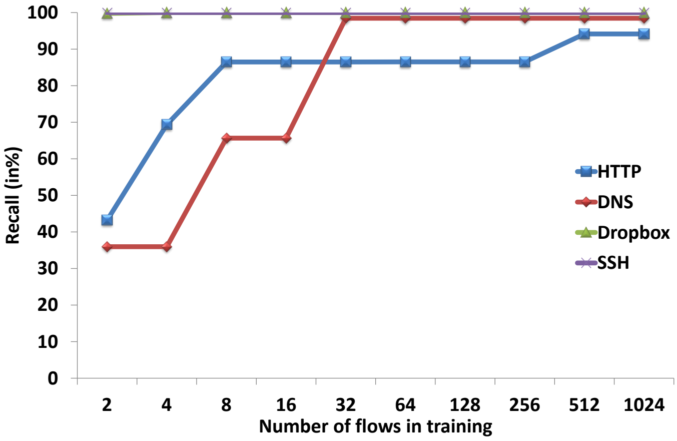


Figure 4.16: Effect of Number of Flows on *Recall*

number of flows. The same trend is shown with other protocols too. Thus we can infer that, for few protocols, small number of flows are sufficient for generating the signatures and for others, relatively more (few hundred) are required to generate good signatures.

4.4 Performance Comparison and Discussion

We also performed experiments to compare the classification performance of our proposed methods with two other methods in the literature. In this section, we describe these experiments and furnish the obtained results.

4.4.1 Comparison between *BitCoding* and *BitProb*

Our two proposed methods *BitCoding* and *BitProb* perform decently in identifying the application flows. In order to understand the performance difference between these two we performed an experiment using grand dataset. We report the results in the form of confusion matrices as in Table 4.33 and Table 4.34 for *BitCoding* and *BitProb* respectively. We can notice that *BitProb* has performed better in comparison to *BitCoding*. The reason for the better performance of *BitProb* is attributed to the contribution of those bits which were otherwise ignored as ('*') in *BitCoding*.

Table 4.33: Confusion Matrix for *BitCoding* with $n = 40$ Bits

	BACnet	**BJNP**	**BitTorrent**	**Bootp**	**CUPS**	**DNS**	**Dropbox**	**GsmIp**	**HTTP**	**Kerberos**	**MWBP**	**NBNS**	**NBSS**	**NTP**	**POP**	**QUIC**	**RPC**	**SIP**	**SMTP**	**SSH**	**Not**
	(22)	(76)	(1582)	(354)	(94)	(119144)	(2328)	(18)	(134198)	(1344)	(14)	(2544)	(728)	(806)	(114)	(254)	(14)	(1280)	(2258)	(2212)	**Classified**
BACnet	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BJNP	0	76	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BitTorrent	0	0	1582	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bootp	0	0	0	354	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CUPS	0	0	0	0	78	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16
DNS	0	0	0	0	0	116706	0	0	0	2	0	2286	0	0	0	0	0	0	0	0	160
Dropbox	0	0	0	0	0	6	2322	0	0	0	0	0	0	0	0	0	0	0	0	0	0
GsmIp	0	0	0	0	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0
HTTP	0	0	0	0	28	0	0	0	116052	4	0	0	0	0	0	0	0	104	0	0	18010
Kerberos	0	0	0	0	0	0	0	0	0	1344	0	0	0		0	0	0	0	0	0	0
MWBP	0	0	0	0	0	0	0	0	0	0	14	0	0	0	0	0	0	0	0	0	0
NBNS	0	0	0	0	0	0	0	0	0	12	0	2520	0	0	0	0	0	0	0	0	12
NBSS	0	0	0	0	0	0	0	0	0	0	0	0	728	0	0	0	0	0	0	0	0
NTP	0	0	0	0	0	66	0	0	0	0	0	0	0	740	0	0	0	0	0	0	0
POP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	114	0	0	0	0	0	0
QUIC	0	0	0	0	0	0	0	0	0	10	0	0	0	0	0	210	0	0	0	0	32
RPC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	0	0	0	0
SIP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1168	0	0	112
SMTP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2258	0	0
SSH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2212	0

Table 4.34: Confusion Matrix for *BitProb* with $n = 40$ Bits

	BACnet	**BJNP**	**BitTorrent**	**Bootp**	**CUPS**	**DNS**	**Dropbox**	**GsmIp**	**HTTP**	**Kerberos**	**MWBP**	**NBNS**	**NBSS**	**NTP**	**POP**	**QUIC**	**RPC**	**SIP**	**SMTP**	**SSH**	**Not**
	(22)	(76)	(1582)	(354)	(94)	(119144)	(2328)	(18)	(134198)	(1344)	(14)	(2544)	(728)	(806)	(114)	(254)	(14)	(1280)	(2258)	(2212)	**Classified**
BACnet	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BJNP	0	74	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
BitTorrent	0	0	1582	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bootp	0	0	0	338	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16
CUPS	0	0	0	0	80	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14
DNS	0	0	0	0	0	119066	0	0	0	0	0	88	0	0	0	0	0	0	0	0	0
Dropbox	0	0	0	0	0	0	2322	0	0	0	0	0	0	0	0	0	0	0	0	0	6
GsmIp	0	0	0	0	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0
HTTP	0	0	0	0	0	0	0	0	133864	0	0	0	0	0	0	0	0	186	0	0	114
Kerberos	0	0	0	0	0	0	0	0	0	1322	0	0	22		0	0	0	0	0	0	0
MWBP	0	0	0	0	0	0	0	0	0	0	14	0	0	0	0	0	0	0	0	0	0
NBNS	0	0	0	0	0	0	0	0	0	0	0	2536	0	0	0	0	0	0	0	0	8
NBSS	0	0	0	0	0	0	0	0	0	0	0	0	728	0	0	0	0	0	0	0	0
NTP	0	0	0	0	0	66	0	0	0	0	0	0	0	740	0	0	0	0	0	0	0
POP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	114	0	0	0	0	0	0
QUIC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	242	0	0	0	0	10
RPC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	0	0	0	0
SIP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1280	0	0	0
SMTP	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2258	0	0
SSH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2212	0

4.4.2 Comparison with *BitFlow*

We compared the classification performance of *BitCoding* and *BitProb* with a recent method *BitFlow* [123] using our grand dataset. *BitFlow* extracts first n bits from each flow of an application in training dataset and use AdaBoost machine learning algorithm [92] for learning the bit patterns. The experiments described in [123] used first 32 bits for TCP flows and first 40 bits for UDP flows. In our experiments, we considered 40 bits long signatures to compare the classification performances. Figure 4.17 shows performance comparison in terms of *Recall* for these experiments. We can notice that *BitProb* performed slightly better as compared to *BitCoding* for all protocols but Bootp. Also, both these methods performed consistently well in comparison to *BitFlow* with some of the protocol flows were not identified at all by *BitFlow*. One of the reason for under-performance of *BitFlow* is because of uneven number of flow samples in training dataset and other reason being similar bit patterns of protocols which exhibit higher similarity when compared with a similarity measurement (which is common with machine learning algorithms). These issues are bound to happen in any dataset collected from any network. We believe that this issue will further aggravate with increase in number of application protocols. In

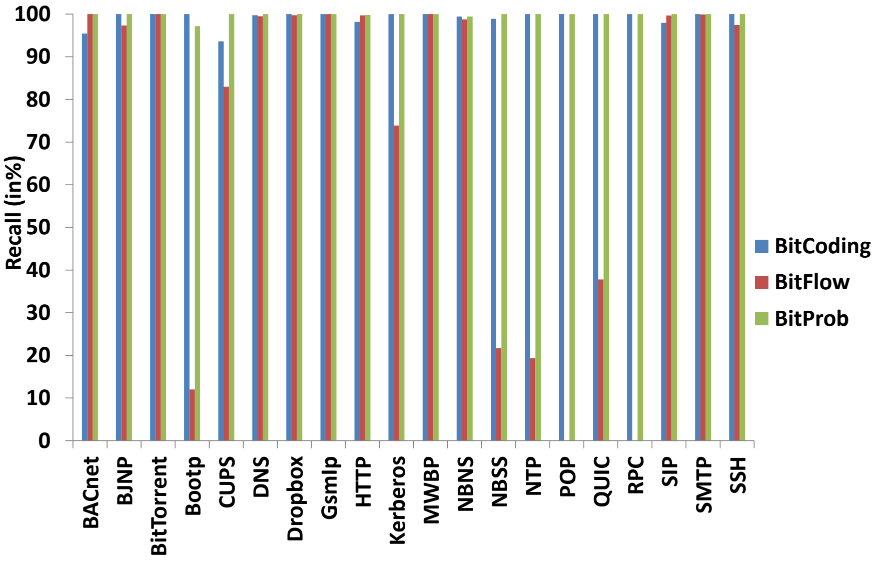


Figure 4.17: *Recall* Comparison Between *BitFlow* and Proposed Methods

contrast, *BitCoding* and *BitProb* can generate signatures with limited number of flows. Further, both these methods can work with variable number of bits for each protocol which is not possible for machine learning algorithms.

4.4.3 Comparison with *ACAS*

We compared *BitCoding* and *BitProb* with another classical method *ACAS* [60] which works with byte level content. *ACAS* generates feature vector from first 'n' bytes of the payload. It generates a binary feature vector of size $n \times 256$ where ASCII position value (1-256) of a byte at i^{th} position sets the binary value at $256 \times i + c[i]$ where $c[i]$ is the position value of byte at i^{th} position in payload. Feature vectors so generated are classified using machine learning algorithms (AdaBoost [92], Naive Bayes [31], Max-entropy [42]). The authors experimented and reported results for first 64 bytes of payload. We repeated this experiment with public-2 dataset and compared the performance with 40 bit signatures of *BitCoding* and *BitProb*. Figure 4.18 shows the *Recall* rate for different approaches. We can notice that *BitCoding* performed better than *ACAS* for many protocols while *BitProb* performed better than *ACAS* for all the protocols but using orders of magnitude less time and data.

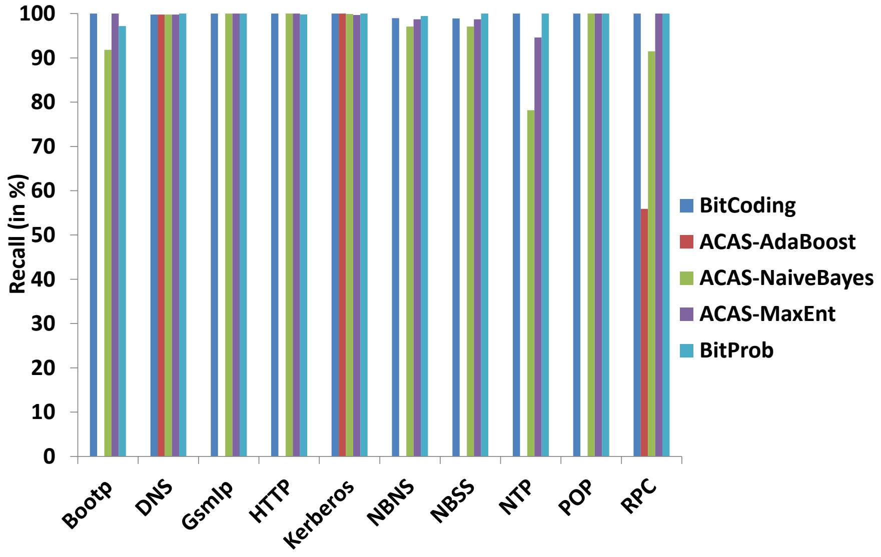


Figure 4.18: *Recall* Comparison Between *ACAS* and Proposed Methods

4.4.4 Discussion

We can notice from the Figure 4.17 and Figure 4.18 that particularly for few protocols like Bootp, NTP and RPC, *BitFlow* and *ACAS* showed poor recall rate. This is because of the less number of flows in training dataset for these protocols as compared to other protocols in the same dataset. This results into an under trained machine learning model for these protocols which leads to poor recall rate. However, there are few other protocols also like BACnet and GSM over IP which has less number of flows in training dataset but still they showed good recall rate. This is because they have approximately same binary string sequence in all training flows which are quite different from the string sequence of other protocols in the training dataset which leads to their decent recall rate. This observation also shows that our proposed methods can perform well even if we have less number of flows in training dataset.

4.5 Conclusion

In this chapter, we presented two bit-level application classification methods *BitCoding* and *BitProb*. *BitCoding* uses first n bits of data extracted from the payloads of bidirectional flow and subsequently, it encodes the signatures with *RLE* and transforms into a state transition machine *TCCA* for efficient representation and comparison. Similar to *BitCoding*, *BitProb* also uses first n bits of data extracted from the payloads of bidirectional flow. It then transforms the generated signatures into a state transition machine *PCDA*. With extensive experimentation we showed that bit-level signatures generated by *BitCoding* and *BitProb* are robust in detecting applications and can be ported from site to site with little compromise in detection performance. Further, we also showed that the signature quality enhances on increasing the number of flows used for signature generation. We also compared *BitCoding* and *BitProb* with a recently proposed bit-level classification method and a byte-level classification method and showed that our methods outperformed previously known bit-level and byte-level classification methods.

Chapter 5

Zero Day Attack Detection in Web Traffic using Payload Analysis

Application specific attacks are on an ever increasing trend [80]. These attacks include buffer overflows, command injection attacks, scripting attacks, etc. These attacks can evade detection from an Intrusion Detection System (IDS) which inspect only header or flow level data as there may not be any visible changes in patterns at header or flow level [30]. To detect such application level attacks, anomaly based detection systems which use payload analysis are proposed [71]. Some of these methods detect anomalous packets by creating a database of short sequences (n-grams discussed in Section 5.1.2) from payload of known non-malicious packets and checking the presence or absence of these sequences in the test packet [113]. However, this approach of binary comparison (presence or absence) is prone to evasion as discussed later in Section 5.1.1. To overcome this issue, we propose two methods, *Rangegram* and *OCPAD* in this chapter to detect zero day attacks in the web traffic. As a representative, we choose web traffic as attacks against HTTP are very common. Both these methods extract n-grams from HTTP packet payload to detect anomalous packets in web traffic. These methods not only check the presence or absence of n-grams but also check the occurrence frequencies and probabilities of n-grams in a packet. Our contributions in this chapter are as follows:

1. We propose two methods, *Rangegram* and *OCPAD*, to detect zero day attacks in web traffic by analyzing payload content of HTTP packets.
2. We propose efficient data structures called *Min-Max-Tree* and *Probability-Tree* used by *Rangegram* and *OCPAD* respectively to store n-grams extracted from packet payload. Along with the extracted n-grams, *Min-Max-Tree* and *Probability-Tree* store the frequency range and occurrence probability range, respectively, of n-grams found in attack free payloads to create a normal profile.
3. We experiment with two HTTP datasets and report the performance of our anomaly detection methods in terms of its *Recall* and *False Positive Rate (FPR).*
4. We compare the detection performance of both the methods with a closely related work Anagram [113].

Rest of this chapter is organized as follows. In Section 5.1, we describe the working of our proposed anomaly detection methods *Rangegram* and *OCPAD.* The experiment details and results are presented in Section 5.2 followed by the conclusion in Section 5.3.

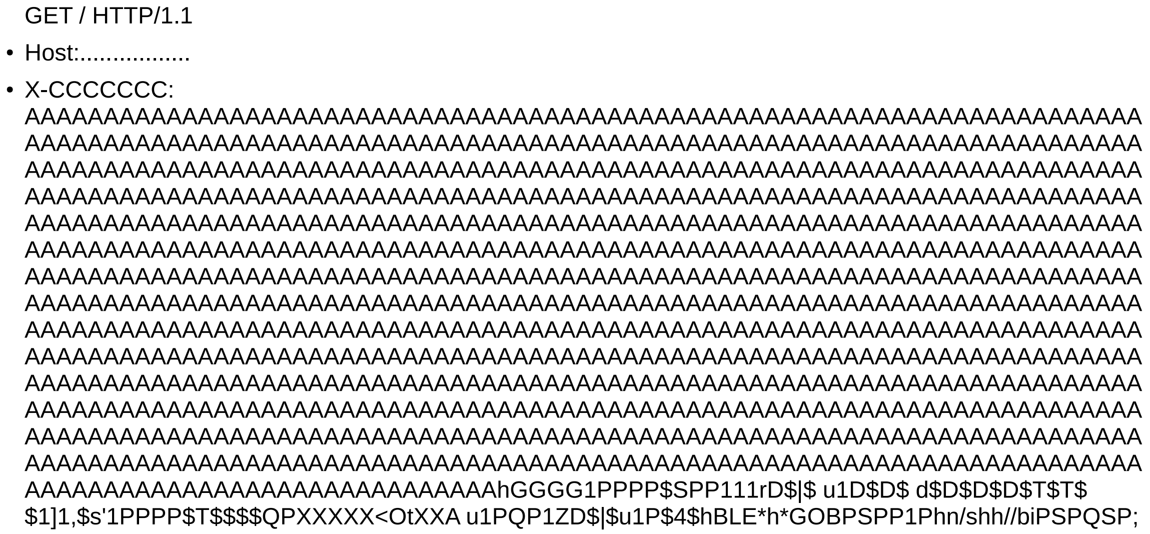
5.1 Proposed Detection Approaches

As discussed in the last section, we propose two anomaly detection methods *Rangegram* and *OCPAD* to detect zero day attacks in web traffic. In this section, we first describe the motivation behind proposing these two anomaly detection methods which is subsequently followed by the explanation of working of these methods.

5.1.1 Motivation

Several payload based anomaly detection methods [51, 65, 89, 90, 112, 113, 114] are proposed in the literature to detect application specific zero day attacks. These methods work by creating a database of short sequences (n-grams) from payload of known non-malicious packets of that application. The created database represents the normal behaviour profile of the application. In order to detect anomalies, short sequences of a test packet are created and compared against the sequences stored in the database.

A score is generated either by counting number of n-grams not found in the database or using their occurrence frequency. This score indicates the degree of deviation from normal profile. If the score crosses a decided threshold value, the packet is declared as anomalous. However, the detection method [112] can be evaded using mimicry attacks [51] wherein an adversary can launch attack in such a way that the malicious traffic is almost similar to the normal traffic used for training purpose. Moreover, it is recently observed [87] that anomaly detection method Anagram proposed in [113] can be bypassed by having conversation with target for a while to discover randomization mask and subsequently use this to craft packets. Also, Anagram declares a packet as anomaly if the generated score is greater than a predefined threshold. This approach of binary comparison (presence or absence) is prone to evasion because an accidental presence of a short sequence in the normal packet which is otherwise only seen in a malicious packet may result into misclassification of the malicious packet if only the sequence's presence or absence is taken into account. For example, the partial content of a sample payload of a buffer overflow attack is shown in Figure 5.1. It can be seen from this figure that, it contains a large number of 'A's inside its payload which are meant to overflow the buffer of a target process to hijack its execution. If binary comparison of short sequences is done and by chance one short sequence containing



```
  GET / HTTP/1.1
• Host:.................
• X-CCCCCCC:
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAhGGGG1PPPP$SPP111rD$|$ u1D$D$ d$D$D$D$T$T$
  $1]1,$s'1PPPP$T$$$$QPXXXXX<OtXXA u1PQP1ZD$|$u1P$4$hBLE*h*GOBPSPP1Phn/shh//biPSPQSP;
```

Figure 5.1: Payload of a Buffer Overflow Attack

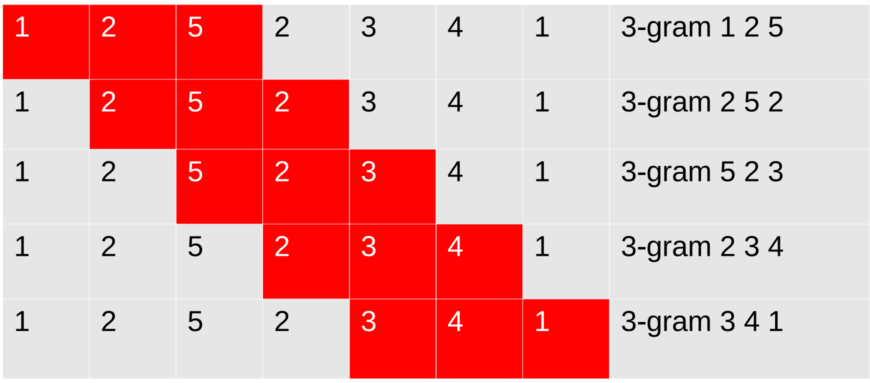
only ‘A’s is found in normal profile, this packet is not detected as anomalous as almost all sequences of this packet are declared as normal. This may result into large number of false negatives.

Drawing motivation from this, we propose two methods: *Rangegram* and *OCPAD* which also use short sequences to analyze payload content of HTTP packets. First method *Rangegram* is a preliminary work towards the detection of zero day attacks. *Rangegram* not only checks the presence or absence of short sequences but it also checks their occurrence frequency range. We also propose another anomaly detection technique *OCPAD* which uses a version of Multinomial Bayesian one class classification technique for accurately detecting anomalous payloads. In the next section, we describe the working of these two detection approaches.

5.1.2 *Rangegram*

Rangegram is an n-gram based zero day attack detection method which uses statistics of n-grams extracted from payload of a HTTP packet to calculate a score known as *anomaly_score*. This score is used to identify the anomalous packets. *Rangegram* has two phases of operation - training and testing. During training phase, it creates a normal profile from an attack free dataset. Subsequently, during testing phase, *anomaly_score* of each test packet is calculated by comparing with normal profile generated during training phase. If *anomaly_score* is greater than the decided threshold value, *Rangegram* labels the test packet as anomalous. In the next few subsections, we explain the method of generating n-grams from a packet payload followed by the training and testing phases of *Rangegram*.

N-gram Generation from Packet Payload: N-gram is a substring of length ‘n’ generated from any string. In our case, payload of a HTTP packet is the string from which the n-grams are generated. These n-grams form the features of a packet. For example, if the payload of a packet has the sequence of elements $1, 2, 5, 2, 3, 4, 1$ then 2-grams and 3-grams generated from this sequence are 1 2, 2 5, 5 2, 2 3, 3 4, 4 1 and 1 2 5, 2 5 2, 5 2 3, 2 3 4, 3 4 1 respectively. From the given sequence, 3-gram generation is shown in Figure 5.2 where a sliding window is used to denote the generated 3-grams. In general, from a packet with payload length of l with $l > n$ there will be $l - n + 1$ generated n-grams.



1	2	5	2	3	4	1	3-gram 1 2 5
1	2	5	2	3	4	1	3-gram 2 5 2
1	2	5	2	3	4	1	3-gram 5 2 3
1	2	5	2	3	4	1	3-gram 2 3 4
1	2	5	2	3	4	1	3-gram 3 4 1

Figure 5.2: 3-gram Generation From a Sequence

Training Phase: In the training phase, n-grams generated from the training dataset (network packet trace) are used to create a model of normal profile. This model is represented in the form of an efficient data-structure called *Min-Max-Tree* which stores each n-gram and its minimum and maximum occurrence frequency in a packet across training dataset. The idea behind using occurrence frequency range of n-grams is to increase the *Recall* and minimize *FPR*. Formally, *Min-Max-Tree* is an n-array tree. The operations that can be performed on *Min-Max-Tree* are:

- Find(x): This operation returns true if the n-gram 'x' is found in the *Min-Max-Tree* otherwise returns false.

- Fetch(x): This operation returns the existing minimum and maximum occurrence frequency range of n-gram 'x' from the *Min-Max-Tree*.

- Update(x, $freq_x$): This operation updates existing occurrence frequency value (either minimum or maximum) of n-gram x with its current occurrence frequency $freq_x$ in the *Min-Max-Tree*.

- Insert(x, $freq_x$): This operation inserts a new n-gram 'x' in the *Min-Max-Tree* and set its minimum and maximum occurrence frequency value to $freq_x$.

Min-Max-Tree can store n-grams of all ranges in one tree using anti-monotonicity property [61] of n-grams which makes it space efficient. Let us construct an example *Min-Max-Tree* of height 2 and 3 packets with payload values of $P_1 = 1,2,2,1,3,3,1;\ P_2 = 1,2,1,2,1,2,2;\ P_3 = 1,1,2,3,3,2,2$. First, 1-grams and 2-grams from the payloads of each packet are extracted and then inserted in the tree one by one. The unique 1-grams of P_1 with their current occurrence frequencies are [$< 1 : 3 >$, $< 2 : 2 >$, $< 3 : 2 >$] and unique 2-grams of P_1 with their current occurrence frequency are [$< 1,2 : 1 >$, $< 2,2 : 1 >$, $< 2,1 : 1 >$], [$< 1,3 : 1 >$, $< 3,3 : 1 >$, $< 3,1 : 1 >$]. Each n-gram is first checked for its previous occurrences using $Find$(n-gram) operation. Since all values appeared for the first time, $Find$(n-gram) operation always returns false and then $Insert$(n-gram, $freq_{n-gram}$) operation is used to insert each n-gram in the *Min-Max-Tree*. After P_1, 1-grams and 2-grams of P_2 are extracted with their current occurrence frequencies as $< 1 : 3 >$, $< 2 : 4 >$ and $< 1,2 : 3 >$, $< 2,1 : 2 >$, $< 2,2 : 1 >$ respectively. Here also, all n-grams are first check for its previous occurrence in the Min-Max-Tree using $Find$(n-gram) operation. Since 1-gram '1' occurred previously, $Find(1)$ operation returns true. Hence $Update$(n-gram. $freq_{n-gram}$) operation is performed as $Update(1,3)$ which updates the occurrence frequency range of 1-gram '1' as $[1,3]$ in the *Min-Max-Tree*. Similarly other 1-grams and 2-grams of P_2 gets updated or inserted in the *Min-Max-Tree*. Same step is repeated for P_3. The final tree after inserting payloads of all three packets is shown in Figure 5.3.

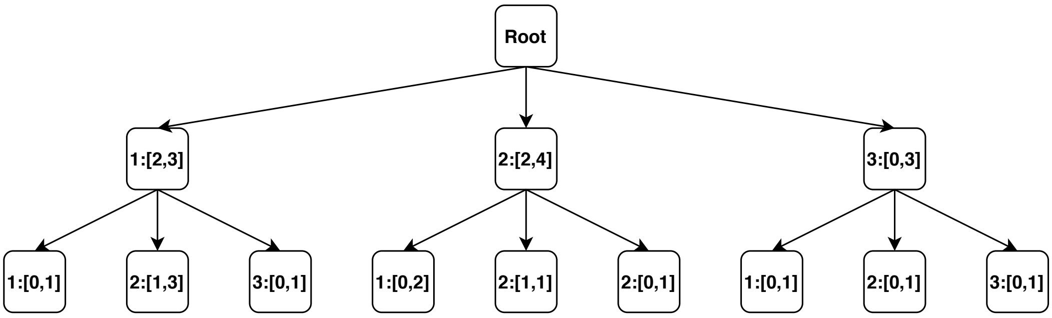


Figure 5.3: *Min-Max-Tree* Constructed up to 2-gram

Each node except the root node in the tree is of the form $< a_i : [min, max] >$.

First component in the node is the i^{th} element of i-gram starting from first node to i^{th} node and second component is its min-max frequency range. Algorithm 5.1 describes the procedure for constructing *Min-Max-Tree* incrementally. It takes network trace containing M packets as input along with highest order of n-grams N and generates *Min-Max-Tree*. It reads each packet, finds its payload length and generates n-grams of order in between 1 and N and updates their min-max range if the n-gram is already found; otherwise the n-gram is inserted in the tree by initializing its min and max value to the found frequency in the packet. The min-max range of the packet gives the occurrence frequency range for each n-gram. Any deviations from this range can be considered abnormal. It is to be noted that *Min-Max-Tree* can be constructed in only one pass.

Algorithm 5.1 *Min-Max-Tree* Generation

```
Input: Network Trace with Packets $P_1, P_2, \cdots, P_M$
Input: $N$ - Highest order n-grams to be generated.
Output: Min-Max-Tree $T$
for $i = 1$ to $M$ do
  $length \leftarrow PayloadLength(P_i)$
  for $j = 1$ to $N$ do
    $list[1, \cdots length - j + 1] \leftarrow$ n-grams of order $j$ from $P_i$
    $a[1, \cdots k] \leftarrow$ Count distinct n-grams from $list$ and generate pair $< n_t : freq_t >$
    for $x = 1$ to $k$ do
      if $Find(a[x]) ==$ True then
        $[min, max] = Fetch(a[x])$
        $min \leftarrow$ Existing minimum frequency for $a[x]$
        $max \leftarrow$ Existing maximum frequency for $a[x]$
        if $freq_{a[x]} < min$ then
          Update($a[x], freq_{a[x]}$) by replacing current value of $min$ with $freq_{a[x]}$
        else if $freq_{a[x]} > max$ then
          Update($a[x], freq_{a[x]}$) by replacing current value of $max$ with $freq_{a[x]}$
        end if
      else
        Insert($a[x], freq_{a[x]}$)
      end if
    end for
  end for
end for
```

Testing Phase: During testing phase, *Rangegram* detects an anomalous packet not only on the basis of presence or absence of n-gram generated from testing packet in trained model but also whether these n-grams occur in [min, max] range defined in the trained model. In the instance of an attack, the content of the malicious packet is different from the payloads used during training otherwise and/or the occurrences frequency of n-grams generated from payloads of such a packet is significantly different. For example, a buffer overflow attack has several n-grams repeated in the payload. A sample packet content of a buffer overflow attack is partially shown in Figure 5.1. We can notice that there is a long run of 'A's in the payload which is rarely seen in any other normal packet. This run of 'A's generate several n-grams with only 'A's whose count is larger than what is found in a normal packet. Resultant skewed range of an n-gram compared to the frequency of same n-gram found in normal packet is the key for detection. Let us consider a test packet P_t with payload as $P_t = 1, 2, 1, 2, 1, 2, 1, 2$. 1-grams extracted from P_t with their current occurrence frequencies are $[< 1 : 4 >, < 2 : 4 >]$. First $Find$(n-gram) operation is performed to check the presence of n-gram in the *Min-Max-Tree* and if n-gram is found then $Fetch$(n-gram) operation is performed to get the permitted frequency range of that n-gram. For 1-gram '1', $Find(1)$ operation returns true and $Fetch(1)$ operation returns permitted frequency range from *Min-Max-Tree* shown in Figure 5.3. The permitted frequency range of '1' obtained is [2, 3]. The current frequency of 1-gram '1' in P_t is 4 which is more than the permitted frequency range, therefore this 1-gram is treated as anomalous 1-gram. This is how anomalous n-grams are identified and counted in test packets which are then used to calculate *anomaly_score* of the test packets. If the *anomaly_score* of the test packet is greater than the threshold value then the packet is identified as anomalous packet.

The procedure used for labeling the packet as either normal or anomalous is shown in Algorithm 5.2. This algorithm takes a testing packet as input and generates n-grams of desired order N. The occurrence frequency of each n-gram is found for a particular value of N. Each n-gram is tested for its presence or absence in the training dataset (in *Min-Max-Tree*). If it is present, its min-max range is compared with the range obtained from testing packet. This is repeated for all n-grams generated from testing packet and *anomaly_score* is generated using Equation 5.1.

$$anomaly\_score = \frac{\sum Anomalous \text{ (n-grams)}}{\sum Normal \text{ (n-grams)} + 1} \tag{5.1}$$

In this equation, anomalous n-grams refer to those n-grams which are either not

Algorithm 5.2 Detecting Anomalous Packets

```
Input: Min-Max-Tree from Training Phase
Input: $P$ - A Packet read from interface
Input: $N$ - Order n-grams to be generated.
Input: $\alpha$ - Threshold for $anomaly\_score$
Output: Label of $P$
$length \leftarrow PayloadLength(P)$
$list[1, \cdots length - N + 1] \leftarrow$ n-grams of order $N$ from $P$
$a[1, \cdots k] \leftarrow$ Count distinct n-grams from $list$ and generate pair $< n_t : freq_t >$
$anomalous\_count = 0$
for $x = 1$ to $k$ do
  if Find($a[x]$) == false then
    $anomalous\_count \leftarrow anomalous\_count + 1$
  else
    $[min, max] = Fetch(a[x])$
     $min \leftarrow$ Existing minimum frequency for $a[x].ngram$
     $max \leftarrow$ Existing maximum frequency for $a[x].ngram$
     if $freq_{a[x]} < min$ or $freq_{a[x]} > max$ then
       $anomalous\_count \leftarrow anomalous\_count + 1$
     end if
  end if
end for
$anomaly\_score = \frac{anomalous\_count}{length - N - anomalous\_count + 1}$
if $anomaly\_score > \alpha$ then
  $P$ is anomalous
else
  $P$ is normal
end if
```

present in the *Min-Max-Tree* or whose occurrence frequency falls outside the range stored in the *Min-Max-Tree*. On the other hand, normal n-grams are the n-grams which are present in the *Min-Max-Tree* and their occurrence frequency falls within the range stored in the *Min-Max-Tree*. If the *anomaly_score* obtained is more than a threshold α, the packet is considered as anomalous otherwise it is considered as normal. We use Laplacian correction method [31] where '1' is added to the denominator to avoid 'divide-by-zero' situation while calculating *anomaly_score*.

Complexity Analysis of *Rangegram*: Our proposed method *Rangegram* has different operations and with each operation is associated some complexity. Table 5.1 shows the complexity of each operation of *Rangegram* including the indication of type of operation i.e. whether the operation is for training(offline) or for testing(online) or both.

1. Find(x): In this operation, an n-gram is searched in the *Min-Max-Tree* of

Table 5.1: Operation-wise Complexity of *Rangegram*

Operation Name	Complexity	Online/Offline	Explanation
$Find$(x)	O($log(N)$)	Both	N = highest order n-gram generated
$Fetch(x)$	O($log(N)$)	Both	N = highest order n-gram generated
$Insert$(x, $freq_x$)	O($log(N)$)	Offline	N = highest order n-gram generated
$Update$(x, $freq_x$)	O(1)	Offline	Constant time as this operation only updates the range values in already fetched n-gram

height N. The value of N is also the highest order of n-grams generated from the payload and thus, the complexity of this operation is O($log(N)$).

2. Fetch(x): This operation returns the frequency range of n-gram 'x' by fetching this n-gram from *Min-Max-Tree* of height N and hence the complexity is O($log(N)$).

3. Insert (x, $freq_x$): This operation inserts an n-gram 'x' at the n^{th} level of *Min-Max-Tree* of height N. Thus the complexity of this operation is O($log(N)$)

4. Update (x, $freq_x$): This operation only updates the occurrence frequency of an already fetched n-gram 'x' with its current frequency $freq_x$ which is a constant time operation. Hence the complexity is O(1)

5.1.3 *OCPAD*

Our second proposed method, *OCPAD*, is a one class payload based anomaly detection method to detect anomalies in web traffic. One class classification methods in machine learning are those methods which require training data of only one class. Similar to *Rangegram*, *OCPAD* also has two phases of operations as training and testing phases. During training phase, *OCPAD* generates the occurrence probability range of each n-gram from every packet and stores it along with the n-gram in our proposed data structure called *Probability-Tree*. This tree is similar to the *Min-Max-Tree*. However, instead of storing occurrence frequency ranges, it stores the occurrence probability ranges. This occurrence probability range serves as an indicator of permitted lower and upper bound probability of occurrence of a particular n-gram in a packet. During testing phase, if an n-gram generated from the payload of HTTP packet under consideration is not found in the database or its occurrence probability in a packet is not in the range stored in *Probability-Tree*, *OCPAD* considers the n-gram as anomalous. Subsequently, the class of test packet (i.e. normal or anomaly) is calculated with probability of each n-gram using a version of Multinomial one class Naive Bayes classifier. In subsequent paragraphs, we describe Naive Bayes classification algorithm and how we adapt it for *OCPAD*.

Naive Bayes Classifier: Naive Bayes classifier is a supervised learning algorithm which is based on Bayes rule shown in Equation 5.2.

$$P(C_j|X) = \frac{P(X|C_j)P(C_j)}{P(X)} \tag{5.2}$$

In the training phase, it reads the feature vectors of d dimension ($X = X_1, X_2, \cdots, X_d$) belonging to any of c classes $C_1, C_2, \cdots, C_c$. Using these feature vectors it estimates prior probabilities of each C_j and also the conditional prior probability $P(X|C_j)$s (where $1 \leq j \leq c$). In the Equation 5.2, $P(C_j)$ is the prior probability of occurrence of class C_j, $P(X|C_j)$ is the conditional probability of vector X given that C_j is the class and $P(X)$ is the probability of feature vector. In the testing stage, the posterior probability of class C_j given a test vector X is estimated and is denoted as $P(C_j|X)$. The probability $P(X)$ is calculated by adding the sum of prior probabilities of all c classes multiplied with conditional probabilities of X given class C_j as shown in Equation 5.3.

$$P(X) = \Sigma_{j=1}^{c} P(X|C_j)P(C_j) \tag{5.3}$$

This ensures that $P(C_j|X)$ is in between (0,1) and probabilities of all C_j's given X (i.e., $P(C_j|X)$) add up to 1. After calculating posterior probability of all c classes, the vector X is labeled with the class C_i ($1 \leq i \leq c$) which has the highest posterior probability among all.

Naive Bayes classifier is prominently used in text classification [72]. Depending on type of feature vectors used, there are two models of Naive Bayes classifier as Multivariate Bernoulli model and Multinomial model [79]. Multivariate Bernoulli model is used for binary feature vectors where each feature indicates presence or absence of each word from a dictionary in a document. In the Multinomial model each feature indicates the number of times each word appears in all the documents of a particular type.

Adapting Multinomial One Class Naive Bayes Classification for *OCPAD*: In *OCPAD*, the n-grams generated from packet form the feature vectors. We use Multinomial model for classifying payload data. A feature vector F is generated from payload content and is denoted by Equation 5.4

$$F = f(n_1), f(n_2)...f(n_k) \tag{5.4}$$

where $f(n_i)$ is the frequency of i^{th} n-gram (here $1 \leq i \leq K$) of a particular order generated from a packet. Since there are 256 ASCII values, $k = 256^N$ where N is the order of n-gram. Probability of each n-gram is calculated using Equation 5.5 where M is the total number of packets in training dataset.

$$P(n_i|Class) = \frac{\sum_{t=1}^{M} f_t(n_i)}{\sum_{t=1}^{M} \sum_{j=1}^{k} f_t(n_j)} \tag{5.5}$$

This equation denote the probability as the ratio of frequency of a particular n-gram from all the M packets to the total number of n-grams from all M packets. Here $Class$ is either normal or attack. Probability of an n-gram n_i's occurrence in normal or attack payloads $P(n_i|Class)$ can be estimated if both normal and malicious datasets are available for training. According to Multinomial model, probability of feature vector F for a class $Class$ is written as in Equation 5.6. This estimates the conditional probability of each feature vector given the class (normal or malicious) as required by the Equation 5.1.

$$P(F|Class) = \prod_{j=1}^{k} \left(P\left(n_j|Class\right)\right)^{f(n_j)} \tag{5.6}$$

It has to be noted that there is a skewed distribution of normal and malicious traffic in any network with malicious traffic being marginal fraction of the total traffic. Further it is difficult to collect an attack or intrusion dataset which covers all possible attacks against an application. This makes a perfect case for the use of one class classification methods. In order to train *OCPAD*, only normal class dataset can be used and deviations seen during testing can be identified as intrusions. We adapt the previously described multinomial method as one class classifier by comparing the probability of a particular packet payload being normal to a threshold value and if it is less than threshold value, it can be detected as malicious. However, adapting this algorithm to our context has few challenges. First challenge is only a small fraction of n-grams actually appear in the packets of normal dataset compared to the total number of possible n-grams (which is 256^n). This results into many n-grams probabilities becoming zero. Second is each packet has different length ranging from 0 to 1500 bytes. This results in unequal number of n-grams hence feature vector F is not of same length consistently. We address these two by modifying the way probability of each component of F is calculated during training and testing phases and these changes are elaborated in subsequent paragraphs.

Training Phase: In the training phase, n-grams with their frequencies are generated from every packet of normal dataset. Frequency of each n-gram is used to find its probability. Unlike Equation 3.2, we calculate the probability of n-gram as the ratio of frequency of a n-gram in the packet to the total number of n-grams in that packet. The modified probability calculation[1] is shown in Equation 5.7.

$$P(n_i|P_t) = \frac{f(n_i)}{\sum_{j=1}^{k} f(n_j)} \tag{5.7}$$

This measure indicates how probable an n-gram is within a packet P_t where $0 \leq t \leq M$. Thus, for each distinct n-gram there can be maximum of M such probabilities with each one from a different packet in training dataset.

In order to identify both under occurrence and over occurrence of a particular n-gram in a packet, we store these probabilities in an efficient data-structure called *Probability-Tree.* This tree stores the minimum and maximum occurrence probabilities of each n-gram. Similar to *Min-Max-Tree*, *Probability-Tree* also stores n-grams of different order in a single tree and thus, it is also very space efficient. Formally, *Probability-Tree* is an n-array tree. The operations that can be performed

[1] This probability calculation helps us to find both over occurrence and under occurrence of n-grams in a packet.

on *Probability-Tree* are:

- Find(x): This operation returns true if the n-gram 'x' is found in the *Probability-Tree* otherwise returns false.

- Fetch(x): This operation returns the existing minimum and maximum occurrence probability range of n-gram 'x' from the *Probability-Tree.*

- Update(x, $P(x)$): This operation updates existing occurrence probability value (either minimum or maximum) of n-gram x with its current occurrence probability $P(x)$ in the *Probability-Tree.*

- Insert(x, $P(x)$): This operation insert a new n-gram 'x' in the *Probability-Tree* and set its minimum and maximum occurrence probabilities value to $P(x)$.

Probability-Tree can also store n-grams of all ranges in one tree using anti-monotonicity property [61] of n-grams which makes it space efficient. Let us construct an example *Probability-Tree* of height 2 and 3 packets with payload values of $P_1 = 1, 2, 2, 1, 3, 3, 1$; $P_2 = 1, 2, 1, 2, 1, 2, 2$; $P_3 = 1, 1, 2, 3, 3, 2, 2$. First, 1-grams and 2-grams of the payloads of each packet is extracted and then inserted in the tree one by one. The unique 1-grams of P_1 with their current occurrence probabilities [$< 1 : [0.42] >$, $< 2 : [0.28] >$, $< 3 : [0.28] >$] and unique 2-grams of P_1 with their current occurrence probabilities are [$< 1, 2 : [0.16] >$, $< 2, 2 : [0.16] >$, $< 2, 1 : [0.16] >$, $< 1, 3 : [0.16] >$, $< 3, 3 : [0.16] >$, $< 3, 1 : [0.16] >$]. Each n-gram is first checked for its previous occurrences using $Find$(n-gram) operation. Since all values appeared for the first time, $Find$(n-gram) operation always returns false and then $Insert$(n-gram, P(n-gram)) operation is used to insert each n-gram in the probability tree. After P_1, 1-grams and 2-grams of P_2 are extracted with their current probabilities as [$1 : [0.42] >$, $< 2 : [0.57] >$ and $< 1, 2 : [0.5] >$], [$< 2, 1 : [0.33] >$, $< 2, 2 : [0.16] >$] respectively. Here also, all n-grams are first checked for its previous occurrences in the *Probability-Tree* using $Find$(n-gram) operation. Since 1-gram '2' already occurred previously, $Find$(2) operation returns true. Here, $Update$(n-gram, P(n-gram)) is performed as $Update(2, 0.57)$ which updates the permitted probability range of '2' as $[0.28, 0.57]$

in the *Probability-Tree*. Similarly other 1-grams and 2-grams pf P_2 gets updated or inserted in the *Probability-Tree*. Same step is repeated for P_3. The final tree after inserting payloads of all three packets is shown in Figure 5.4

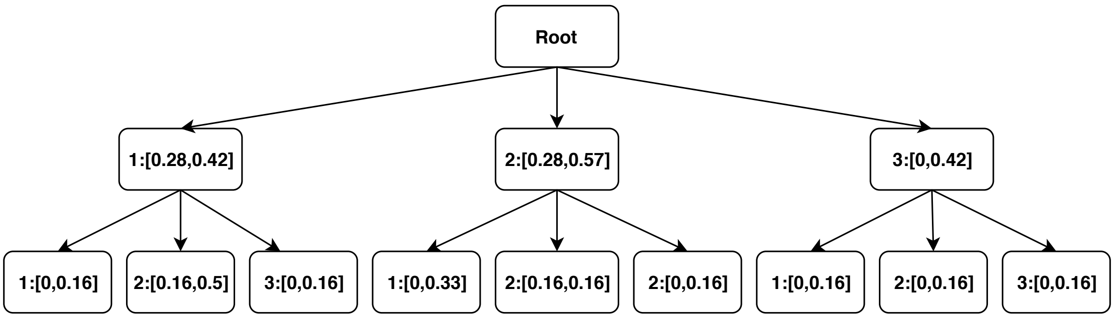


Figure 5.4: *Probability-Tree* Constructed up to 2-gram

Each node except the root node in the tree is of the form $< a_i : [P_{min}, P_{max}] >$. First element in the node is the i^{th} element of i-gram starting from first node to i^{th} node and second component is its min-max probability range. Algorithm 5.3 describes the procedure for constructing *Probability-Tree* incrementally. It takes a

Algorithm 5.3 *Probability-Tree* Construction

```
Input: Network trace with packets $P_1, P_2, \cdots, P_M$
Input: $N$ - Highest order n-grams to be generated.
Output: Probability-Tree $T$ of height $N$
for $t = 1$ to $M$ do
  $length \leftarrow PayloadLength(P_t)$
  for $j = 1$ to $N$ do
    $list[1, \cdots length - j + 1] \leftarrow$ n-grams of order $j$ from $P_t$
    $totalngrams \leftarrow$ size of $list$
    $a[1, \cdots k] \leftarrow$ Generate probabilities of distinct n-grams from $list$ and generate pair $\langle n_i : P(n_i) \rangle$
    for $x = 1$ to $k$ do
      if $Find(a[x]) == True$ then
        $[min, max] = Fetch(a[x])$
        $min \leftarrow$ Existing minimum probability for $P(a[x])$
        $max \leftarrow$ Existing maximum probability for $P(a[x])$
        if $P(a[x]) < min$ then
          $Update(P(a[x]))$ minimum probability with $P(a[x])$
        else if $P(a[x]) > max$ then
          $Update(P(a[x]))$ maximum probability with $P(a[x])$
        end if
      else
        $Insert(a[x], P(a[x]))$
      end if
    end for
  end for
end for
```

set of M packets from training dataset as input along with highest order of n-grams

N and generates *Probability-Tree.* It reads each packet, finds its payload length and generates n-grams of a particular order (in between 1 and N) and subsequently generates a list of unique n-grams and their corresponding probabilities. If the n-gram is already found in the *Probability-Tree*, probability values are used to update each n-gram's probability range in the tree otherwise the n-gram is inserted in the tree by initializing its minimum and maximum probabilities to the probability of n-gram from current packet.

Testing Phase: In the testing phase, feature vectors are generated from each test packet in the same way it is generated in training phase. Using the prior probabilities, each of these feature vectors (packets) are classified as either normal or anomalous. In *OCPAD*, anomalous packets are detected not only by the presence or absence of n-gram but depends on the probabilities of n-grams in the training dataset. Similar to the training phase, each n-gram's probability is calculated as the ratio of count of each n-gram to the total number of n-grams from that packet. Using individual n-gram's probabilities, we calculate the probability of each vector which is termed as *Packet Probability* (same as $P(normal|F)$). *Packet Probability* is equal to the product of probabilities of n-grams found in that packet. One key change in the calculation of the probability of a packet is that we consider only anomalous n-gram's probability. We consider an n-gram as normal if the n-gram is found in the *Probability-Tree* and its probability is within the range of probabilities i.e., within $[P_{min}, P_{max}]$ probability. However, if an n-gram is not found in the *Probability-Tree* or its probability is not in the $[P_{min}, P_{max}]$ range in the *Probability-Tree*, it is considered as anomalous. The anomalous n-gram's probability is multiplied with that of the current packet's probability. This variation serves two purposes. First, it determines the packets with large number of anomalous n-grams considering the occurrence range of each n-gram and second, the underflow problem that arises when large number of probabilities are multiplied which is a common issue with Bayes classifier.

Given the assumption that an attack packet's payload is different from the normal one or it may contain n-grams with different frequency in the payload, it results into more number of unseen n-grams or over/under occurred n-grams. This reduces the value of *Packet Probability*, hence it detects malicious payloads of type shown in Figure 5.1. When this packet's probability is calculated, these repeating n-grams does not fall within the $[P_{min}, P_{max}]$ range of probability found in the tree and hence is detected as anomaly. Let us consider a test packet P_t with payload as $P_t = 1, 2, 1, 2, 1, 1, 9$ whose *Packet Probability* is to be calculated for 1-gram. 1-grams extracted from P_t with their current occurrences probabilities are $< 1 : 0.57 >, < 2 : 0.28 >, < 9 : 0.14$. First $Find$(n-gram) operation is performed to check the presence of n-gram in the *Probability-Tree* and if n-gram is found then $Fetch$(n-gram) operation is performed to get the permitted probability range of that n-gram. For 1-gram '1', $Find(1)$ operation returns true and $Fetch(1)$ operations returns the permitted probability range for 1-gram '1' from the *Probability-Tree* as shown in Figure 5.4. The permitted probability range obtained is [0.28, 0.42]. The current probability of 1-gram '1' is 0.57 which is more than the permitted probability range, therefore this 1-gram is treated as anomalous. Moreover, when $Find(9)$ operation is performed on n-gram '9' it returns false because 1-gram '9' is not found in *Probability-Tree.* Therefore this 1-gram is also considered as anomalous. This is how anomalous n-grams are identified and used to calculate *Packet Probability* of the test packets. If the *Packet Probability* of the test packet is less than the threshold value then the test packet is identified as anomalous packet.

The procedure used for labeling the packet as either normal or anomalous is shown in Algorithm 5.4. This algorithm takes a testing packet as input and generates n-grams of desired order N. The probability of each n-gram is found for a particular value of N. Probability of each n-gram is tested with the range of *Probability-Tree.* If it is present within range, n-gram is assigned with probability 1 whereas when it is found out of range then its own probability is used for calculation of *Packet Probability.* This is repeated for all n-grams generated from testing packet and each packet is assigned a probability. If the packet's probability obtained is less than the threshold α, the packet is considered as anomalous.

Algorithm 5.4 Detecting Anomalous Packets

```
Input: Probability-Tree from training phase
Input: $P$ - A packet read from interface
Input: $N$ - Order n-grams to be generated.
Input: $\alpha$ - Threshold for a Feature Vector
Output: Label of $P$
$length \leftarrow PayloadLength(P)$
$list[1, \cdots length - N + 1] \leftarrow$ n-grams of order $N$ from $P$
$totalngrams \leftarrow$ size of $list$
$a[1, \cdots k] \leftarrow$ Generate probabilities of distinct n-grams from $list$ and generate pair $\langle n_i : P(n_i) \rangle$
$Probability(F) = 1$
for $x = 1$ to $k$ do
  if $Find(a[x]) == False$ then
    $Probability(F) = Probability(F) \times P(a[x])$
  else
    $[min, max] = Fetch(a[x])$
    $min \leftarrow$ Existing minimum probability for $P(a[x])$
    $max \leftarrow$ Existing maximum probability for $P(a[x])$
    if $P(a[x]) < min$ or $P(a[x]) > max$ then
      $Probability(F) = Probability(F) \times P(a[x])$
    else
      $Probability(F) = Probability(F) \times 1.0$
    end if
  end if
end for
if $Probability(F) < \alpha$ then
  $P$ is anomalous
else
  $P$ is normal
end if
```

Complexity Analysis of *OCPAD*: Our proposed method *OCPAD* has different operations and with each operation is associated some complexity. Table 5.2 shows the complexity of each operation of *OCPAD* including the indication of type of operation i.e. whether the operation is for training(offline) or for testing(online) or both.

1. Find(x): In this operation, an n-gram is searched in the *Probability-Tree* of height N. The value of N is also the highest order of n-grams generated from the payload and thus, the complexity of this operation is O($log(N)$).

2. Fetch(x): This operation returns the probability range of n-gram 'x' by fetching this n-gram from *Probability-Tree* of height N and hence the complexity is O($log(N)$).

3. Insert (x, $P(x)$): This operation inserts an n-gram 'x' at the n^{th} level of *Probability-Tree* of height N. Thus the complexity of this operation is O($log(N)$)

4. Update (x, $P(x)$): This operation only updates the occurrence probability of an already fetched n-gram 'x' with its current probability $P(x)$ which is a constant time operation. Hence the complexity is O(1)

Table 5.2: Operation-wise Complexity of $OCPAD$

Operation Name	Complexity	Online/Offline	Explanation
$Find$(x)	O($log(N)$)	Both	N = highest order n-gram generated
$Fetch(x)$	O($log(N)$)	Both	N = highest order n-gram generated
$Insert$(x, $freq_x$)	O($log(N)$)	Offline	N = highest order n-gram generated
$Update$(x, $freq_x$)	O(1)	Offline	Constant time as this operation only updates the range values in already fetched n-gram

5.2 Experiments and Results

In order to evaluate the detection performance of *Rangegram* and *OCPAD*, we conducted experiments on two datasets containing web traffic. First dataset contained only normal web traffic while the second dataset contained attack traffic. In the following subsections, we first describe the generated dataset details followed by the detection performance of proposed methods. Subsequently, we present a comparative study between the proposed methods and one of the closely related work Anagram [113].

5.2.1 Dataset Description

We used two datasets to evaluate the detection performance of proposed methods. The first one was normal dataset containing attack free HTTP packets while the second one was attack dataset containing malicious HTTP packets. The details of these datasets are described below.

Normal Dataset: This dataset was collected in our departmental network of IIT Indore. This network presently serves more than 600 users. Network traffic was collected at the gateway of this network for 3 hours. We collected only web traffic by putting appropriate HTTP filter in the traffic capturing tool tcpdump [24]. There were total of 1082336 packets in this dataset and the dataset size was

1002 MB approximately. This dataset was divided into 3 parts. First part was used for generating normal profile (*Min-Max-Tree* for *Rangegram* and *Probability-Tree* for *OCPAD*), second part was used for finding the threshold α for each value of N (n-gram order) and the third part was used for testing purposes to obtain the *FPR*s of *Rangegram* and *OCPAD*. Statistics of this traffic are shown in Table 5.3.

Table 5.3: Statistics of HTTP Normal Dataset

Type	Size (in MB)	Number of Packets
Training	0787	0838876
Threshold Setting	0074	0089151
Testing	0141	0154309
Total	**1002**	**1082336**

Attack Dataset: We used attack dataset provided by authors of [66, 89] for our experiments. This dataset has 3 types of attacks as follows.

1. Generic attacks: This part contains traces of 64 attacks. It contains few buffer overflow and remote execution attacks mounted against a web server.

2. Shell-code attacks: This part contains traces of shell-code attacks executed against web servers.

3. CLET attacks: This part contains trace of attacks generated using an evading tool CLET [21].

Statistics of this dataset are shown in Table 5.4.

Table 5.4: Statistics of HTTP Attack Dataset

Type	Size (in KB)	Number of Attacks	Number of Packets
Generic	0216.8	64	174
Shell-code	0125.5	11	087
CLET	1126.4	96	780

5.2.2 Training Phases of *Rangegram* and *OCPAD*

The training dataset contained 838876 packets as shown in Table 5.3. In the training phase, we constructed *Min-Max-Tree* and *Probability-Tree* for *Rangegram* and *OCPAD* respectively. To construct *Min-Max-Tree* and *Probability-Tree*, we wrote a Java program that uses jNetPcap programming library [10] to process the network traces in tcpdump format and generate *Min-Max-Tree* and *Probability-Tree.* This Java program was executed on a machine running Ubuntu 14.04 operating system and having quad-core processor (i5-pro) with 4 GB of RAM. The program took approximately 8 and 10 minutes for generating *Min-Max-Tree* and *Probability-Tree* respectively. In order to establish the fact that only a fraction of total number of possible n-grams actually appeared in the dataset, we collected statistics for number of distinct n-grams for different values of N. Table 5.5 shows the value of N, number of distinct n-grams found from training dataset, total number of possible n-grams for that order and fraction of appeared n-grams to the total possible n-grams. We can notice that, as the value of N increases, the fraction decreases considerably. The distinct number of n-grams generated during training phase shows the total number of nodes at each level of *Min-Max-Tree* and *Probability-Tree* where level is decided by the value of N in n-grams.

Table 5.5: Fraction of n-grams seen for Distinct Values of N

Value of N	**Distinct n-grams**	**Maximum Possible n-grams**	**Fraction**
1	97	256	37.891%
2	9313	65536	14.211%
3	893979	16777216	5.328%
4	8558096	4294967296	0.199%
5	24964563	1099511627776	0.002%
6	48713732	281474976710656	0.00000017%

5.2.3 Threshold Calculation

As discussed earlier in Section 5.1.3, a threshold value α was used to differentiate normal and anomalous packets. Instead of deciding the value of α manually, we derived the value experimentally. To calculate the threshold value, we used the second part of the normal dataset containing 89151 packets. We represent the threshold values for *Rangegram* and *OCPAD* as $\alpha_{Rangegram}$ and α_{OCPAD} respectively. For *Rangegram*, we first calculated the *anomaly_score* of each packet and subsequently, threshold $\alpha_{rangegram}$ was set to the average of all the anomaly scores calculated from all the normal packets and adding standard deviation to it. Similarly for *OCPAD*, we first calculated the *Packet Probability* of each packet and subsequently, threshold α_{OCPAD} was set to the average of all the probabilities calculated from all the normal packets and subtracting standard deviation from it. We added standard deviation in *Rangegram* but subtracted standard deviation in *OCPAD* because if the *anomaly_score* is high, the chance of *Rangegram* detecting a test packet as anomaly is also high. However, if the *Packet Probability* is less, the chance of *OCPAD* detecting a test packet as anomaly is high. Threshold values calculated for different values of N are shown in Table 5.6.

Table 5.6: Threshold Values for Different Values of N

Value of N	$\alpha_{rangegram}$	α_{OCPAD}
1	0.00024	0.9966
2	0.00135	0.9417
3	0.00293	0.4706
4	0.02682	0.1682
5	0.04926	0.0038
6	0.08340	0.0017

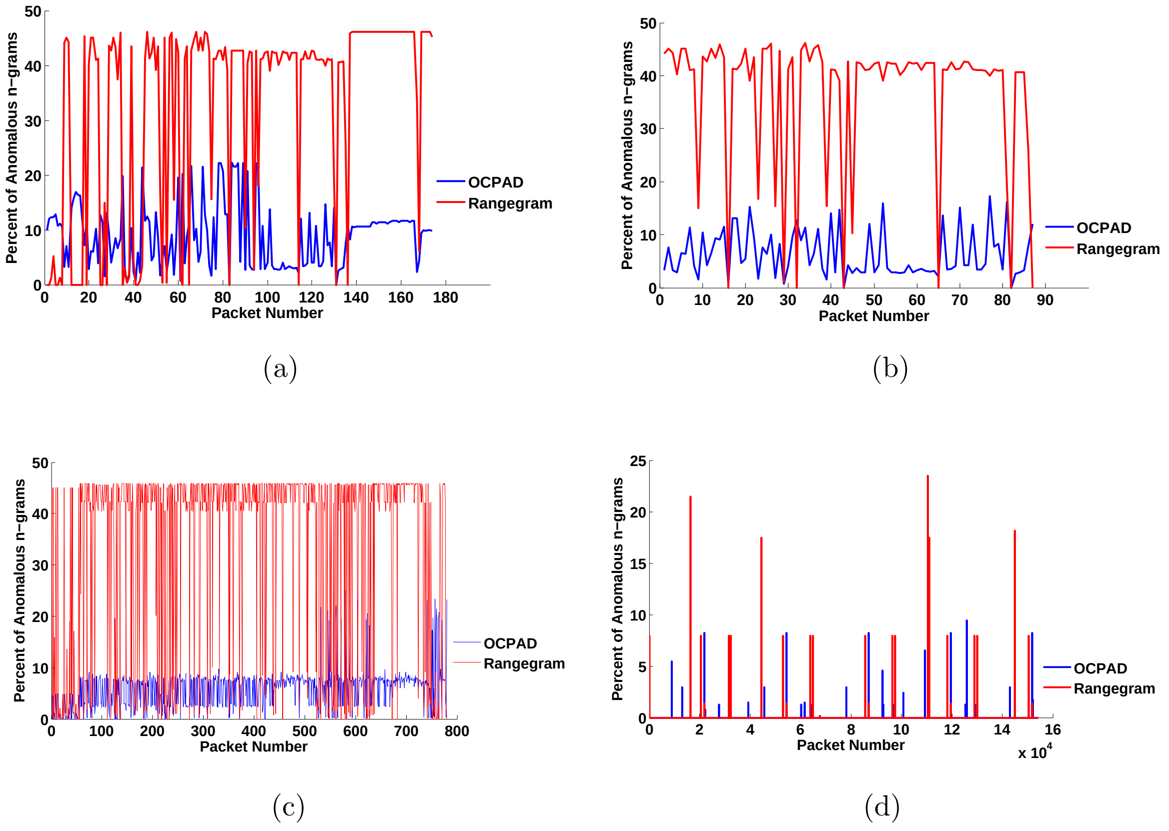
5.2.4 Testing Phases of *Rangegram* and *OCPAD*

The third part of normal dataset containing 154309 packets and the entire attack dataset were used for testing purpose. Figures 5.5a, 5.5b, 5.5c and 5.5d show the percentage of anomalous n-grams of order 3 in a packet belonging to Generic, Shellcode and CLET attack datasets and normal dataset respectively. We can notice that packets belonging to attacks contained more number of anomalous n-grams compared to normal packets. This fact was exploited by *Rangegram* and *OCPAD* to detect anomalous packets in the web traffic.

The detection performance of both the approaches is determined by the *Recall* for each attack type and *FPR* for normal dataset. *Recall* and *FPR* are calculated using equation 5.8 and 5.9 respectively.

$$Recall = \frac{TP}{TP + FN} \tag{5.8}$$

$$FPR = \frac{FP}{TN + FP} \tag{5.9}$$



(a)

(b)

(c)

(d)

Figure 5.5: Percentage of Anomalous 3-grams in (a) Generic Attack (b) Shellcode Attack (c) CLET Attack and (d) Normal Dataset

In these equations, TP, TN, FP and FN are as follows:

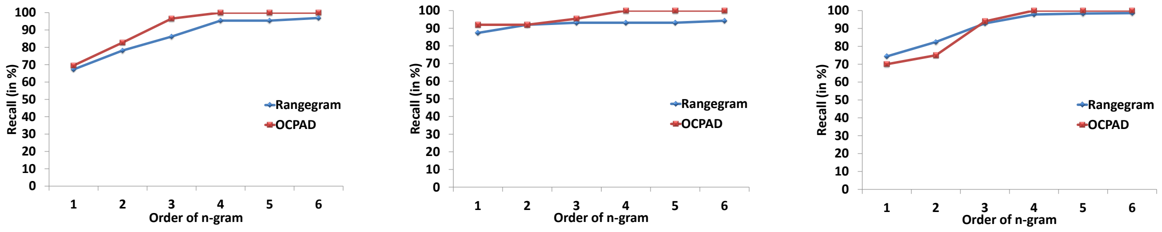
True Positive (TP): Attack packets correctly detected as attack

True Negative (TN): Normal packets correctly detected as normal

False Positive (FP): Normal packets incorrectly detected as attack

False Negative (FN): Attack packets incorrectly detected as normal

Recall of *Rangegram* and *OCPAD* for Generic, Shellcode and CLET attacks is shown in Figures 5.6a, 5.6b and 5.6c respectively. It can be noticed from these figures that in case of lower order n-grams (e.g. 1-grams and 2-grams), *Rangegram* gives better detection performance as compared to *OCPAD* for CLET attack. However, in case of higher order n-grams, *OCPAD* gives better detection performance as compared to



(a) (b) (c)

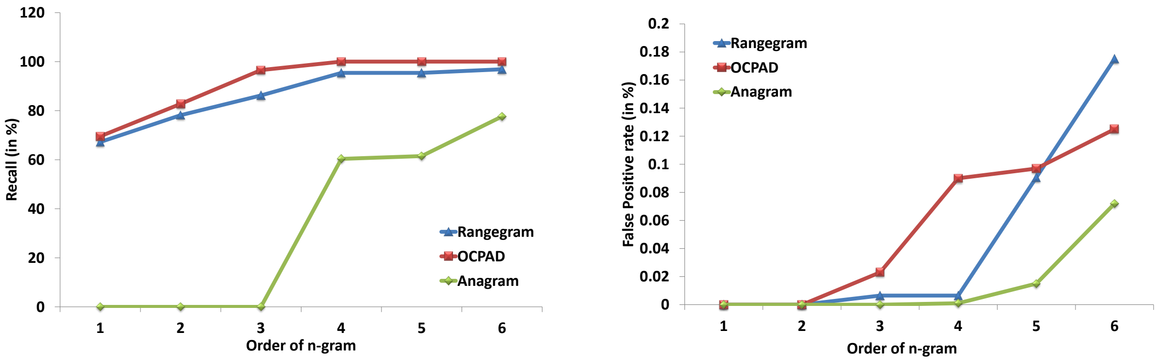
Figure 5.6: *Recall* of *Rangegram* and *OCPAD* for (a) Generic (b) Shellcode and (c) CLET Attack

Rangegram for CLET attack. For all other attacks, *OCPAD* gives better detection performance as compared to *Rangegram* irrespective of the order of n-gram.

5.2.5 Comparison with Anagram

We also compared the detection performance of *Rangegram* and *OCPAD* with a closely related work Anagram [113] in terms of *Recall* and *FPR*. Anagram computes the ratio of n-grams generated from the testing packet but not found in the database to the total number of n-grams of the packet as the anomaly score. Figure 5.7a shows the *Recall* of Anagram, *Rangegram* and *OCPAD* for generic attack. We can notice from the figure that both of our proposed methods achieved significantly higher *Recall* as compared to Anagram. Anagram's poor performance was due to the binary approach of finding the n-grams (i.e. presence/absence) in the database. Figure 5.7b shows the *FPR* comparison of *Rangegram*, *OCPAD* and Anagram on our normal dataset containing 154309 packets. It can be noticed from the figure that *FPR* of Anagram is slightly lower as compared to *Rangegram* and *OCPAD*. Nevertheless, *FPR* of our methods is still within acceptable limits (i.e. less than 0.2%).

We can notice from the Figure 5.7a that both of our proposed methods achieved significantly higher recall as compared to Anagram. Anagram's poor performance was due to the binary approach of finding the n-grams (i.e. presence/absence) in the database. Moreover, payload of the packets in generic attack dataset mostly contains



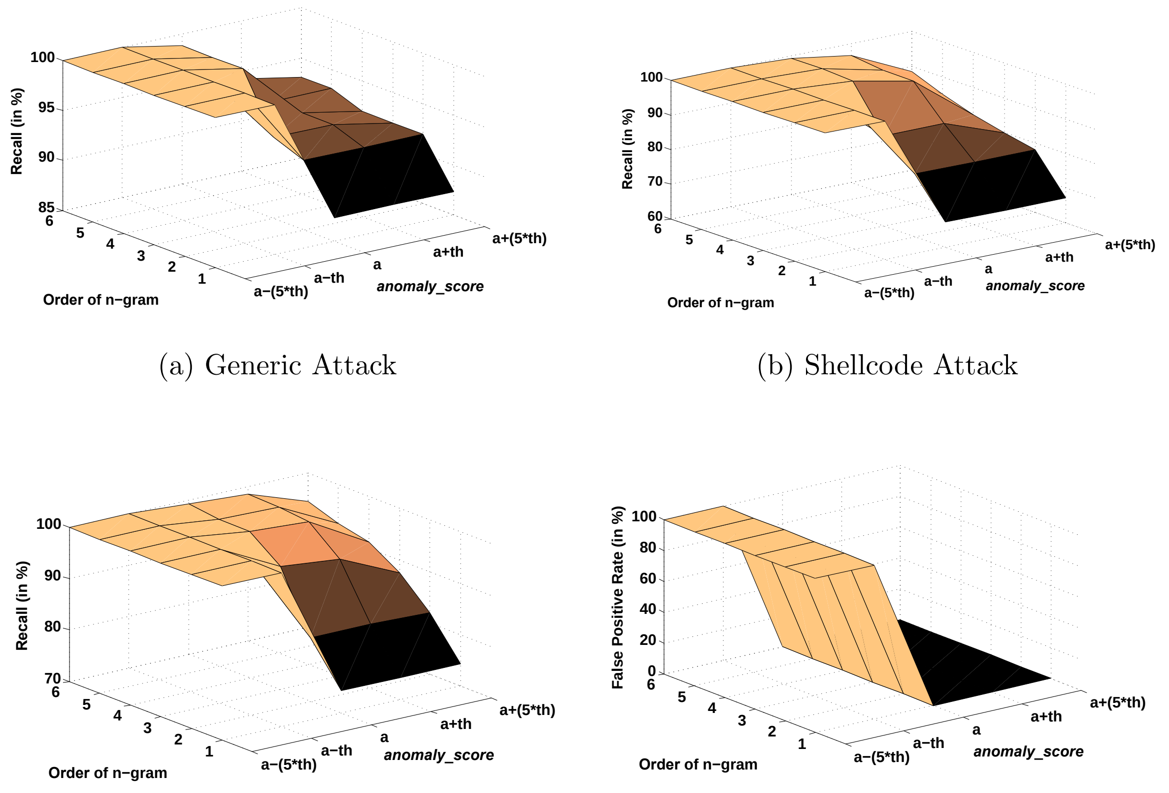
(a) (b)

Figure 5.7: Performance Comparison of Anagram and Proposed Methods in terms of *Recall* and *FPR*

the high occurrences of A. Since the occurrences of uni-gram, bi-gram and tri-gram having continuous A is common in an attack payload, the binary approach of finding n-grams fails here and the anomalous packets are not detected for 1-grams, 2-grams and 3-grams. However, the detection is possible if 4-grams and higher orders of n-grams are considered. On the other hand, our proposed methods, *Rangegram* and $OCPAD$ check the occurrence frequency and occurrence probability of n-grams for detecting anomalous packet. In the generic attack dataset, as the occurrence frequency and occurrence probability of $A's$ are much higher in the attack dataset as compared to their values found in training dataset, our approaches can detect anomalous packets for any order of n-gram.

5.2.6 Sensitivity Analysis

As discussed in Section 5.2, both of our proposed methods *Rangegram* and $OCPAD$ use threshold values $\alpha_{Rangegram}$ ($anomaly\_score$) and α_{OCPAD} (*Packet Probability*) respectively to differentiate between a normal packet and an anomalous packet. The accuracy of both these methods depend on these chosen threshold values and thus, it is important to choose them wisely in order to obtain high *Recall* but low *FPR*. To understand the impact of threshold values on the detection performance, we varied the threshold value and calculated *Recall* and *FPR* of both the methods for different orders of n-grams. The effect of varying the threshold value and n-gram order on the detection performance of *Rangegram* is shown in Figures 5.8a, 5.8b, 5.8c and 5.8d. In these figures, ‘a’ and ‘th’ are the average *anomaly_score* and standard deviation respectively. We can notice from these figures that independent of the attack type and the n-gram order, as the threshold *anomaly_score* increases, *Recall* decreases. Moreover, *FPR* also decreases with increase in threshold *anomaly_score* independent of n-gram order. Thus, the threshold *anomaly_score* is chosen in such a way that *Rangegram* should achieve high *Recall* but low *FPR*. In case of *OCPAD*, the effect of varying the threshold value and n-gram order on the detection performance is shown in Figures 5.9a, 5.9b, 5.9c and 5.9d. We can notice from these figures that independent of the attack type and the n-gram order, as the threshold *Packet Probability* increases, *Recall* increases. Moreover, *FPR* also increases with the increase in threshold *Packet Probability*. Thus, the threshold *Packet Probability* is chosen in such a



(a) Generic Attack

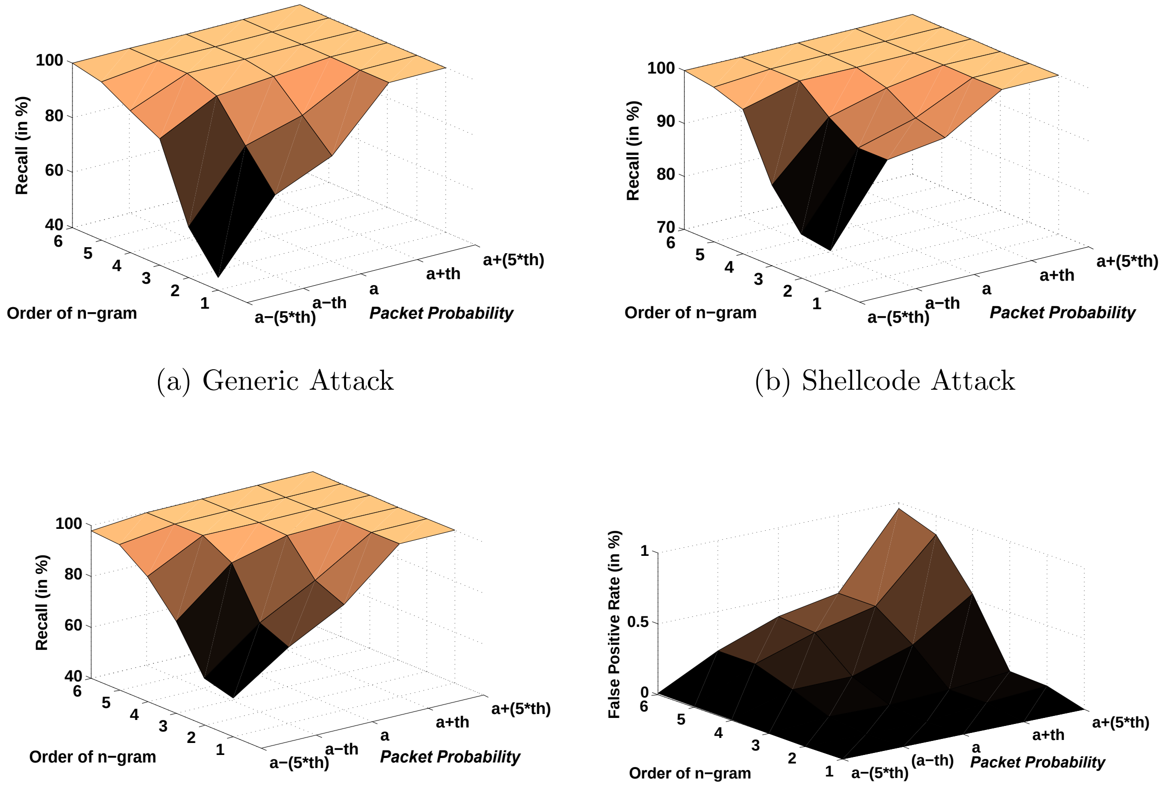
(b) Shellcode Attack

(c) CLET Attack

(d) Normal Dataset

Figure 5.8: Sensitivity Analysis for *Rangegram*

way that *OCPAD* achieves high *Recall* but low *FPR*.



(a) Generic Attack

(b) Shellcode Attack

(c) CLET Attack

(d) Normal Dataset

Figure 5.9: Sensitivity Analysis for *OCPAD*

5.3 Conclusion

Zero day attacks can evade state of the art payload anomaly detection methods which use either binary approach to find n-grams in the database or use absolute frequency values for deriving anomaly score. Thus, in this chapter, we described two new methods, *Rangegram* and *OCPAD* for detecting anomalous packets in web traffic. Unlike the previous methods, our proposed methods not only check the presence or absence of n-grams instead also check the occurrence frequency and probabilities of n-grams occurring in a packet. We experimented with one normal HTTP dataset and one publicly available attack dataset and showed that our proposed methods could detect different attack packets in HTTP traffic with high accuracy. We also compared the detection performance of our proposed methods with Anagram and showed that our methods outperformed Anagram in terms of *Recall*.

Chapter 6

Detecting Spam Callers in VoIP with Graph Anomalies

6.1 Introduction

Internet telephony or Voice over IP (VoIP) is a cheaper alternative for telephone communication compared to traditional Public Switched Telephone Networks (PSTN). In addition to supporting voice calls, VoIP also provides services like messaging, video calling, etc. The transmission of voice and multimedia content between users is carried over Internet Protocol (IP) networks. A VoIP call usually has a call setup/signaling operation (control plane operation) and subsequently media transmission session (data plane operation). Signaling is used to establish and disconnect calls. There are various signaling protocols like H.323, Session Initiation Protocol (SIP) [26] and others for this purpose. Of late, SIP has become a de-facto standard [44] signaling protocol because of its simplicity and easy implementation. SIP is an application layer signaling protocol which can establish, modify and terminate connection in Internet Telephony. For data transmission, most VoIP systems use Real time Transmission Protocol (RTP) . Growing popularity of VoIP and its low cost communication render it as a lucrative choice for spam users [39, 108][1]. Spam

[1] There are other known threat vectors to VoIP systems like flooding and coordinated attacks [54, 55], billing attacks [126], user enumeration, spoofing and man-in-the-middle attacks [46].

calls are made either through recorded audio files which are played during the call or by a customer care executive who speaks after dialing a number. There are also automated softwares available for generating such calls by taking a list of user-IDs as input. These calls may be promotional calls related to some business offers and are generally termed as SPam Over Internet (SPIT) calls. These calls are annoying to most VoIP users and need to be filtered. Many spam call detection methods use history and reputation [36, 67, 101] of a user as a metric to identify spam users. In this direction, authors of [106, 107] proposed methods which first extract a set of call details from SIP payload using DPI and subsequently use them to obtain parameters such as calling frequency, number of calls, call duration, etc. to detect spam users. However, these methods use only average values of these features and hence they fail to work if a determined spam user maintains these averages by generating artificial calls among spam users as discussed in Section 6.3.1. To address this issue, we describe a method that uses DPI to analyze SIP[2] packets in order to identify spam users in VoIP system. Spam user detection is performed using a form of social interaction among the users. The social interaction status of a user is collected from the history of calls made or received by that user and the calls' frequency, direction, etc. This interaction between users is represented as a weighted graph and to identify the spam users, we identify anomalies in the graph. The anomalies can be identified due to the fact that spam users usually have different interaction features like making more frequent calls, large number of calls, shorter duration calls, etc. as compared to normal users. Thus, these features when used to derive appropriate weights on edges help differentiate the spam users. In specific, our contributions in this chapter are:

1. We identify a set of differentiating call characteristics of normal and spam users and propose a mechanism to convert these features into weighted graphs.

2. We describe a graph anomaly detection system to identify spam users using the weighted graph. This system identifies anomalies in the graph by considering how

[2] We analyze SIP due to the fact that it is most popular and widely used signaling protocol for VoIP [97]

similar the node is compared to its neighbourhood.

3. We perform experiments with two different VoIP call datasets generated using a large number of simulated VoIP users and show that the proposed method can successfully identify the spam users.

Rest of this chapter is organized as follows. In Section 6.2, we describe the working of SIP and how it is used for VoIP call setup. Our proposed method *SpamDetector* is elaborated in Section 6.3. We describe the experiments conducted to evaluate the detection performance of proposed method in Section 6.4. Finally, we conclude this chapter in Section 6.5.

6.2 Session Initiation Protocol (SIP)

SIP is a signaling protocol which is used for VoIP call establishment, call management and call disconnection. SIP is the most popular VoIP signaling protocol as it is a text-based light weight protocol. Different building blocks of VoIP setup are shown in Figure 6.1 and are as follows:

1. **User agent:** These are VoIP phones with a valid Uniform Resource Indicator

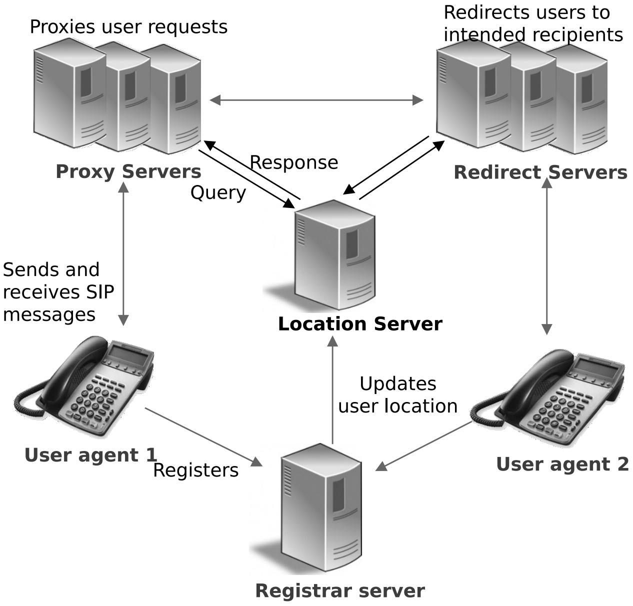


Figure 6.1: Voice over IP Building Blocks

(URI). These URIs represent user names used by the VoIP users. Multimedia sessions are established and terminated between user agents.

2. Registrar server: The user agents register with a registrar server when they connect to network and also update their location periodically.

3. Location server: The location server stores different locations of user agents which are identified by their IP addresses. There can be more than one location associated with a user.

4. Proxy server: The proxy server forwards the connection requests to the intended recipients on behalf of user agents.

5. Redirect server: If a user has more than one location, the redirect server helps to fork a connection request to different address.

The SIP messages that are exchanged between a caller (e.g. sip.caller@abc.com) and a callee (sip.callee@abc.com) in a complete call are shown in Figure 6.2 and are as follows:

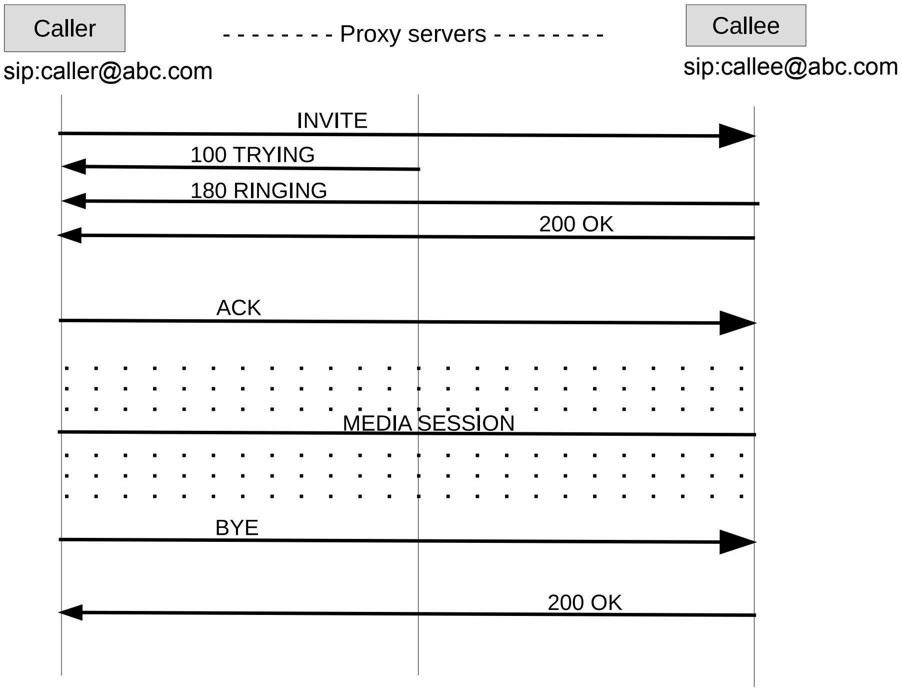


Figure 6.2: Voice over IP Call Setup using Session Initiation Protocol

1. An **INVITE** request message that is sent to a proxy server is responsible for initiating a session.

2. The proxy server immediately sends a **100 Trying** message as a response to the caller to stop the retransmissions of the **INVITE** request. Further, the proxy server searches the address of callee in the location server. After getting the address, it forwards the **INVITE** request message to the callee.

3. Thereafter, **180 Ringing** as a provisional response message is generated by callee and is returned back to caller.

4. A **200 OK** response message is sent immediately after callee picks up the phone to answer the call.

5. Callee receives an **ACK** message from the caller once it gets **200 OK** message.

6. After this, the media session gets established and data packets (actual conversations between caller and callee) are exchanged from both ends using media exchange protocols such as Real-time Transport Protocol (RTP) and RTP Control Protocol (RTCP).

7. After the conversation is complete, any participant (caller or callee) can send a **BYE** request message to terminate the call.

8. Finally the participant which gets **BYE** message, sends a **200 OK** response to confirm the reception of **BYE** message and the call is terminated.

6.3 Proposed Work

To detect VoIP spam users, we propose a method called *SpamDetector*. In this section, we first present the motivation behind the proposed method followed by its working.

6.3.1 Motivation

There are several VoIP spam detection methods available in literature [106, 107]. These methods use calling behaviour patterns of VoIP users to identify spam users in VoIP networks. These methods analyze the call parameters of VoIP users such as their average call duration, average successful call rate, time of arrival of call per day, etc. These methods detect spam users by finding those VoIP users whose average call duration, successful call rates are comparatively lower than normal VoIP users. However, a determined spam user can evade these detection methods by generating successful artificial calls to other spam users or bots with high call durations. In this way, they can maintain their average call parameters similar to that of other normal VoIP users. Thus, to detect such determined spam users, it is important to find the value of different call parameters between each caller-callee pair. Drawing motivation from this fact, we propose a graph based anomaly detection method *SpamDetector* that analyzes calling behaviour of each caller-callee pair using a weighted directed graph.

6.3.2 *SpamDetector*

Graph based anomaly detection methods identify nodes in the graph which look different from other nodes. To identify such nodes, these methods use a particular metric to find the similarity between the nodes [29]. Our proposed spam user detection method *SpamDetector* also detects anomalies in a graph representing social interaction of different VoIP users. *SpamDetector* has mainly three modules - 1) Call Detail Record (CDR) generation, 2) A weighted directed graph known as *Call Graph* generation and 3) Identification of anomalies in the *Call Graph*. Figure 6.3 shows these three phases of *SpamDetector* along with the input and output of the system. The CDR generation phase takes SIP packets as input and the anomaly detection phase generates the list of spam users as output. The working of the three phases of *SpamDetector* are described in next few subsections.

6.3.2.1 Call Detail Record Generator

Call Detail Record (CDR) generator processes the SIP packets collected from VoIP Private Branch Exchange (PBX) server and generates a summary of call records. Each entry in the CDR has details of call ID, name of caller, callee, status of call and call duration. Few sample call records are shown in Table 6.1[3]. Every VoIP call is uniquely

[3] We use the entries in this table as running example to elaborate the working of *SpamDetector* later in this section.

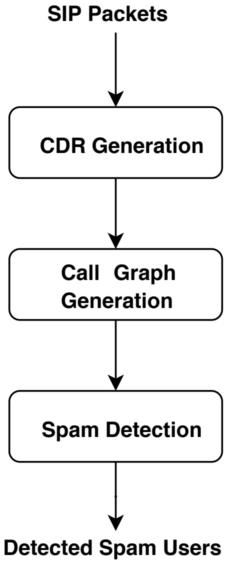


Figure 6.3: Modules of *SpamDetector*

identified by alphanumeric call ID. Status of call is either complete or incomplete and can be identified by whether the recipient answers the call or not. Whether the call is answered or not answered is identified by packet exchanges in control plane operation. For control plane operation, as discussed earlier in Section 6.1, mostly Session Initiation Protocol (SIP) is used. The duration between an ACKNOWLEDGMENT and a BYE message as shown in Figure 6.2 is the duration of call. If there is no ACKNOWLEDGMENT message in the conversation, the call is considered as incomplete.

6.3.2.2 Call Graph Generator

Call Graph Generator processes the CDR entries and generates a weighted graph $G = (V_G, E_G, W_E : E_G \rightarrow \mathbb{R})$ where

- $V_G = \{v_1,\ v_2,\ v_3,\ ...\ v_m\}$ is a set of nodes such that each node denotes a VoIP user among m VoIP users and $m \in \mathbb{N}$.

- $E_G = \{e_{12}, e_{13}..e_{mk}\}$ is a set of edges such that edge e_{pq} denotes a directed edge from node p to node q when a first call is attempted by a caller denoted by node p to a callee denoted by node q such that $p, q \in V_G$.

Table 6.1: Sample Call Data Records (CDR)

Call-ID	Caller	Callee	State of Call	Call Duration (in seconds)
00026a3d7178b1569f34f6e081a92ea0	A	B	Complete	30
000e2f4873ccd9c20e1ea5f87acfb789	A	C	Complete	20
000ae6dc45f6af6d4f2e8da46521757a	A	D	Complete	25
0021a02a64689959e8a6ce3ef5ecf0b0	A	B	Incomplete	00
005d927734961afe13538e05415972f4	A	C	Incomplete	00
00607bbc25274a5052a5ebf530c375fd	A	D	Incomplete	00
00c306591b57578a3c5a87270639d07e	B	E	Complete	60
0042914c5997f300143dbe076931181f	C	E	Complete	55
00c7af8ab0e2c3d9c519b83f0c5dd0b4	D	E	Complete	50
00ecf3df08bf040c6e778fd839dfac84	E	B	Complete	55
011e1c62a0b56f5fd85859eaf7944385	E	D	Complete	60
016a42db37b16e0c2ee537294020dda2	D	A	Complete	40
01d023b327cc58e446e5a6c5784a2043	E	D	Incomplete	00
00e336791b57574a3c5a87570339c06f	A	C	Incomplete	00
002d927734931ade12538e15412973a5	B	C	Complete	50

- $W_E = \{w_{12}, w_{13}, \cdots w_{mk}\}$ is a set of weights such that w_{pq} denotes the weight on the edge e_{pq}.

Call Parameters: The weights on edges of *Call Graph* are assigned based on the calling behaviour of VoIP users. There are many call parameters to detect behaviour of a VoIP user however we consider three parameters - 1) *Successful Call Rate*, 2) *Average Talk-time Per Call* and 3) *User Role in the Call* which can be leveraged to calculate edge weights. These three parameters are described below:

1. Successful Call Rate: This is the ratio of successful calls by a caller i to the total number of attempted calls by that caller to another user j. We denote successful call rate by α and its value is calculated using Equation 6.1. In Equation 6.1, the value α lies in the range of 0 to 1. If α is 0, it indicates that there are no successful calls by the user and if it is 1, all calls by user i to user j are successful.

$$\alpha_{ij} = \frac{\text{Successful Calls by User } i \text{ to User } j}{\text{Total Calls attempted by User } i \text{ to User } j} \tag{6.1}$$

2. Average Talk-time Per Call: This parameter measures the average duration of each call for a user i to user j. We use τ to denote average talk-time per call and its value is calculated using Equation 6.2. The value of $\tau \geq 0$.

$$\tau_{ij} = \frac{\text{Total Talk-time of Calls from User } i \text{ to User } j}{\text{Total Successful Calls by User } i \text{ to User } j} \tag{6.2}$$

3. User Role in the Calls: This is the ratio of number of times the user u calls to the number of times s/he receives the call. We denote this parameter by ρ and value is calculated using Equation 6.3. If ρ is less than 1, user acts as callee most of the time and if ρ is greater than 1, user acts as caller most of the time.

$$\rho_u = \frac{\text{Number of Times User } u \text{ is a Caller}}{\text{Number of Times User } u \text{ is a Callee}} \tag{6.3}$$

The choice of above three parameters is motivated by the fact that VoIP spam users make unwanted calls and often calls are generated from a recorded media file (message) for marketing purpose or advertisement. Sometimes, for promotional purposes of products, users in call centers also make such calls. Spam calls annoy VoIP users (receivers) and are likely to be disconnected prematurely by the receiver. Hence the average call duration of such callers is relatively small. Also spam users try calling many users in short period of time hence they act as callers in all these calls. Furthermore, many receivers notice the call ID used for such calls and are likely to reject the calls coming from these IDs in future. This makes success rate of calls very low. We consider these three observations about spam users [74] [109] [106] [78] and deduce that spam users usually have small values of α and τ but large value of ρ.

Weight Assignment: The weights on edges of *Call Graph* are derived from the three call parameters α_{ij}, τ_{ij} and ρ_{ij}. The weight w_{ij} on the directed edge e_{ij} from node i to node j is calculated by taking the average of α_{ij}, τ_{ij} and ρ_{ij} for all calls between these pair of nodes. Henceforth, in this chapter, α_{ij}, τ_{ij} and ρ_{ij} are referred as the average of the respective values between a pair of nodes. It should also be noted that ρ cannot be calculated for an edge as it is measured using both received and attempted calls therefore we calculate this value for the node and assign this value during weight calculation of the outgoing edge of that particular node. The weight assignment for an edge e_{ij} is a function of call parameters. We notice that, the weight w_{ij} is related to the three parameters as follows:

$$w_{ij} \propto \frac{1}{\alpha_{ij}} \;\; where \;\; \alpha_{ij} \in [0, 1] \tag{6.4}$$

$$w_{ij} \propto \frac{1}{\tau_{ij}} \;\; where \;\; \tau_{ij} \in \mathbb{R} \tag{6.5}$$

$$w_{ij} \propto \rho_i \;\; where \;\; \rho \in \mathbb{R} \tag{6.6}$$

We can notice that α_{ij}, τ_{ij} and ρ_i have different range of values. In order to avoid biasing in the assigned weight towards one of the parameters, we normalize each parameter using *Feature Scaling* method of normalization [58] as shown in Equation 6.7.

$$x' = \frac{x - min(x)}{max(x) - min(x)} \tag{6.7}$$

During normalization, the zero values are replaced with the lowest nonzero value from the processed CDR list. This is done to avoid the weight being converted to either 0 or ∞. We denote normalized values of α_{ij}, τ_{ij} and ρ_i by α'_{ij}, τ'_{ij} and ρ'_i. It is to be noted that the proportionality of weight and call parameters do not change even after normalization. After normalizing the values of call parameters in Equations 6.4, 6.5 and 6.6 and combining them, we obtain

$$w_{ij} \propto \frac{{\rho_i}'}{\alpha'_{ij} \times \tau'_{ij}} \tag{6.8}$$

or

$$w_{ij} = k \times \frac{{\rho_i}'}{\alpha'_{ij} \times \tau'_{ij}} \tag{6.9}$$

where k is a proportionality constant. If we set k=1 then

$$w_{ij} = \frac{{\rho_i}'}{\alpha'_{ij} \times \tau'_{ij}} \tag{6.10}$$

Example of *Call Graph* Generation and Weight Assignment: Consider CDR shown in Table 6.1. The *Call Graph* generated from this CDR is shown in Figure 6.4. The graph has 5 nodes denoting 5 users: A, B, C, D and E. A directed edge is generated from one node to another whenever a user attempts first call to another user (subsequent calls lead to change in weights as averages of parameters change). Final weights are assigned after processing whole CDR. We can see from the Table 6.1 that there are 15 calls out of which 10 are complete calls and 5 are

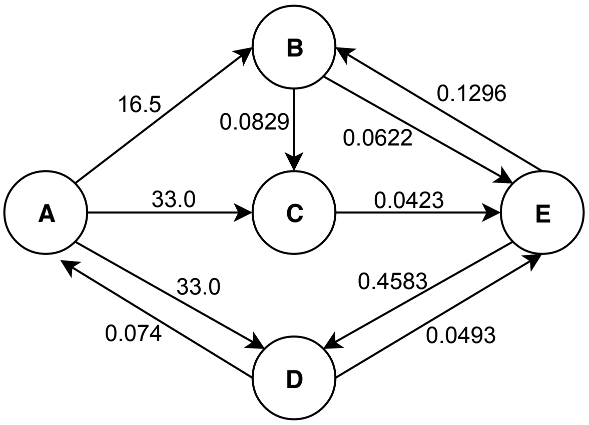


Figure 6.4: Example Call Graph

incomplete calls. The calculated values of α, τ and ρ using the above equations (normalized) and final weights on each edge are shown in Table 6.2. We can also see that there are 10 entries in this table representing 10 unique caller-callee pair.

Table 6.2: Values of α, τ and ρ for Example Call Detail Records

Edge (i→j)	α_{ij}	τ_{ij}	ρ_{ij}	**Norm(α_{ij})**	**Norm(τ_{ij})**	**Norm(ρ_{ij})**	w_{ij}
A→B	0.50	30	7.00	0.2424	0.250	1.0000	16.5000
A→C	0.34	20	7.00	0.2424	0.125	1.0000	33.0000
A→D	0.50	25	7.00	0.2424	0.125	1.0000	33.0000
B→C	1.00	50	0.67	1.0000	0.750	0.0622	00.0829
B→E	1.00	60	0.67	1.0000	1.000	0.0622	00.0622
C→E	1.00	55	0.25	1.0000	0.875	0.0370	00.0423
D→A	1.00	40	0.50	1.0000	0.500	0.0370	00.0740
D→E	1.00	50	0.50	1.0000	0.750	0.0370	00.0493
E→B	1.00	55	1.00	0.2424	0.875	0.1111	00.1296
E→D	0.50	60	1.00	0.2424	1.000	0.1111	00.4583

6.3.2.3 Spam Detection using ***SpamDetector***

Spam detection module detects spam users by identifying anomalies in graph. In our case, the *Call Graph* generated in the previous phase is the input given to this phase. Anomalies in the graph are identified by considering the local (1-hop) neighbourhood of a node. A node which is apparently different from its neighbourhood is considered as anomaly. In order to estimate the similarity of node v_i with respect to a neighbour v_j, we define a parameter *Spam Outlier Factor* (SOF). This SOF is subsequently used to identify anomalies in the graph. SOF of a node v_i with respect to an incident edge e_{ij} (hence with respect to an adjacent node v_j) is the ratio of weight (w_{ij}) to the average of weights of incident edges on node v_j (other than weight w_{ji}) as shown in Equation 6.11.

$$SOF(v_i(e_{ij})) = \frac{w_{ij}}{\frac{(\sum_{k=1}^{m} w_{jk}) - w_{ji}}{m-1}} \tag{6.11}$$

This calculated SOF is compared with the average SOF of node v_j for deciding whether the node v_i is different with respect to edge e_{ij}. Average SOF of node v_j is calculated using Equation 6.12

$$SOF_{av}(v_j) = \frac{(\sum_{k=1}^{m} SOF(e_{jk})) - SOF(e_{ji})}{m-1} \tag{6.12}$$

where m is the out-degree of v_j. If the ratio of SOF of node v_i with respect to edge e_{ij} is greater than N times the standard deviation of average SOF of node v_j, the node is considered as anomaly with respect to edge e_{ij}. Finally a node v_i is declared as outlier or anomaly if it is declared as anomaly with respect to majority of its neighbours. Standard deviation and the thresholds used for declaring node v_i as anomaly with respect to edge e_{ij} are shown in Equation 6.13 and Equation 6.14 respectively

$$SOF_{std-dev}(v_j) = \sqrt{\frac{\sum_{k=1}^{m-1}(SOF_{av}(v_j) - SOF(e_{jk}))}{m-1}} \tag{6.13}$$

$$SOF_{th}(v_i(e_{ij})) = SOF_{av}(v_j) + N \times SOF_{std-dev}(v_j) \tag{6.14}$$

where N is a natural number

Detecting Anomalies in Example Graph: Let us consider the example *Call Graph* shown in Figure 6.4. It has 5 nodes and 10 directed edges with each edge having a weight assigned to it. The weight on each edge is shown in Table 6.2. In this graph, let us consider that the node A is to be decided whether it is anomaly or normal node. As described earlier in this section, this is decided by majority voting of its neighbours using their respective SOF scores. Node A has B, C and D as its neighbours by virtue of edges e_{AB}, e_{AC} and e_{AD} having weights 16.5, 33 and 33 respectively. $SOF(A(e_{AB}))$ is calculated as the ratio of w_{AB} to the average of w_{BC}

and w_{BE} which results into 227.743 (=16.5/(0.0829+0.0622)/2)). $SOF_{th}(A(e_{AB}))$ is the sum of the average $SOF_{avg}(B)$ and $SOF_{std-dev}(B)$. $SOF_{avg}(B)$ is the average of $SOF(B(e_{BC})$ and $SOF(B(e_{BE})$ which have the values of 1.96 and 0.13 respectively. The average $SOF_{avg}(B)$ in this case is 1.045. Similarly the standard deviation value of $SOF_{std-dev}(B)$ using Equation 6.13 is 0.912. Considering N=1 in Equation 6.14 the threshold value on the $SOF_{th}(A(e_{AB}))$ is 1.96. As the $SOF(A(e_{AB}))$ value is 227.743 which is greater than this threshold value of 1.96 of $SOF_{th}(A(e_{AB}))$, it is interpreted that the behaviour of node A is not similar to that of B and hence with respect to B, node A is considered as an anomaly. Similar calculations on other two edges e_{AC} and e_{AD} also result into declaring the node A as anomaly. Hence through majority voting, node A is finally declared as anomaly. Similar calculations on other nodes and edges result into declaring all the remaining nodes other than A as normal. These results are summarized in Figure 6.5 where the anomalous node A is shown in red colour and all normal nodes are shown in green colour.

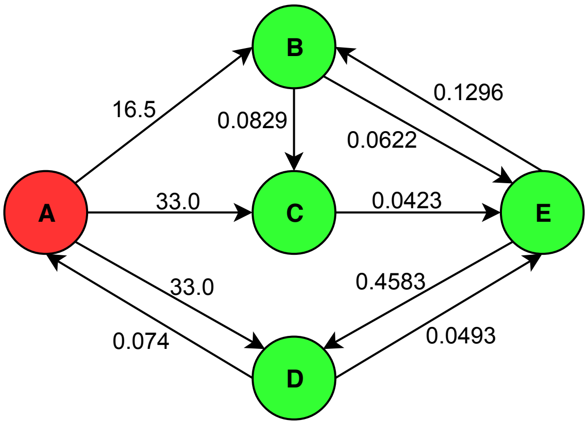


Figure 6.5: Identifying Anomaly in Example Call Graph

6.4 Experimental Evaluation

To evaluate the detection performance of proposed method *SpamDetector*, we conducted two experiments. In the first experiment, we created 1,000 simulated users in our testbed setup as discussed in next subsection. In the second experiment, we used a modified version of a CDR generator script [25] to generate CDRs of 10,000 users. The reason for conducting the second experiment with such a large user base was to test the scalability of our proposed method. In this section, we describe the testbed setup for generating VoIP call dataset for first experiment, datasets used, obtained values of different call parameters and *SpamDetector* evaluation results obtained in both the experiments.

6.4.1 Testbed Setup

To generate the dataset for our first experiment, we created a setup similar to the one shown in Figure 6.6. Using this setup, we simulated two enterprise networks with a set of users in each network. All the three machines in the setup had Intel Core i5-4590 processor with the clock rate of 3.30 GHz, 16 GB of RAM and were using Ubuntu 18.04 LTS. In one of the machines, we installed Asterisk Server 11.18.0 [2] which acted as VoIP PBX [27]. On other two machines, we installed VoIP clients which generated and received VoIP calls. These two client machines generated several VoIP calls using a modified version of VoIP bot program [81]. Client machines ran multiple instances of bot program and randomly generated one or more number of calls at a time. Each bot program mimicked a user and ran as an independent process on a different UDP port. This bot program used Jain SIP API [8] to handle the SIP signaling and Java Media Framework (JMF) [9] to generate RTP flow using recorded

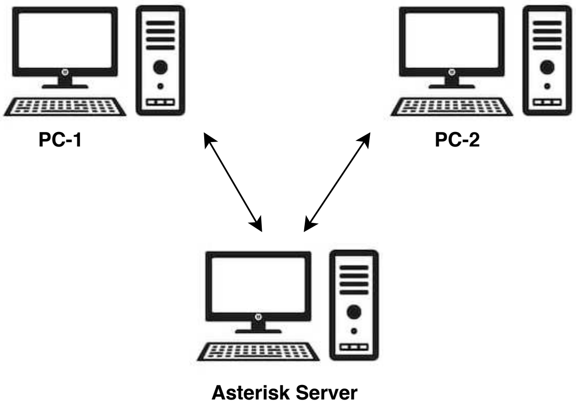


Figure 6.6: Testbed Setup used to Generate VoIP Calls

audio files of varying duration. We first created 1000 users by registering them on the server. We named these users as bot1, bot2, bot3 and so on upto bot1000. We then simulated these VoIP users in both the machines and generated calls among them.

6.4.2 Dataset Generation using 1000 users

As discussed earlier in this section, we created 1000 simulated users (bot1-bot1000) in our first experiment. Out of these 1000 users, we configured 2% of the users to mimic the behaviour of spam users (bot981-bot1000) and remaining as normal users (bot1-bot980). The simulation of calling behaviour of these normal and spam users is done as follows:

1. We configured the normal users to generate non-frequent calls in such a way that each normal user calls other normal users with a random call duration of 1 minute to 5 minutes. We also configured the normal users to make few calls to spam users.

2. We configured the spam users to generate two types of calls. First, they called normal users frequently with a random call duration of 5 seconds to 30 seconds. For spam users, we also generated large number of unsuccessful calls by sending only INVITE packets. Second type of calls were artificial calls made by spam users within the group. As discussed in Section 1, spam users may attempt to maintain the three parameters (used to derive weight) close to that of normal users by generating some artificial calls among themselves. To simulate this behaviour, we scheduled calls between spam users themselves. Using these calls, spam users maintained - 1) the average talk-time by calling to each other for longer call duration, 2) the high successful call rate with other spam users and 3) the user role in call by receiving calls from other spam users in a coordinated manner. This behaviour of VoIP spams is also described in [38].

We collected the network traffic generated through this VoIP communication in the Asterisk server machine using tcpdump software [24]. With appropriate filters in tcpdump, we collected only SIP traffic. Processing only SIP traffic is sufficient for generating CDRs and it also does not compromise users' privacy to some extent[4].

[4] RTP contains real conversation data.

Using this setup, we generated VoIP calls between all the 1000 users. This simulation continued for 3 days resulting into a filtered SIP dataset of 5.339 GB. We created CDRs from this dataset and found a total of 147507 calls. Out of these, 97037 were attended calls whereas 50470 were dropped calls. A summary of call statistics and collected dataset is shown in Table 6.3. Total number of calls between different user groups are shown in Table 6.4.

Table 6.3: Dataset Generated using 1000 VoIP Users

Parameter	**Value**
Total size of dataset (in GB)	005.3390
Number of packets (only SIP)	3453257
Number of complete calls	0097037
Number of incomplete calls	0050470
Total number of VoIP users	0001000

Table 6.4: Calls Between Different User Groups in First Experiment

Calls	**Number of calls**
Normal to Normal	49573
Spam to Normal	50000
Spam to Spam	47507
Normal to Spam	00427

6.4.3 Dataset Generation for 10000 users

As discussed earlier in this section, in our second experiment, we generated CDRs of 10,000 VoIP users by modifying a CDR-generator script [25] to test the scalability of our proposed method *SpamDetector*. Instead of simulating 10,000 users, we generated the CDRs using this script because it is difficult to simulate such a large number of users on a machine[5]. This CDR-generator script can be configured by varying different parameters such as the number of calls to be generated, call duration, call drop rate, etc. We configured the script such that it generated CDRs similar to one we obtained for 1000 users in our previous experiment. Also, we configured the script to designate 2% users as spam users (bot9801 to bot10000) and remaining as normal users (bot1 to bot9800). The generated CDRs showed that the call duration and call drop rate

[5] Each simulated user is executed as a process on a machine.

between normal users was between 1 minute to 5 minutes and less than or equal to 0.3 respectively. Moreover, call duration and call drop rate between spam to normal users was less than or equal to 0.5 minutes and more than 0.7 respectively whereas call duration and call drop rate between spam to spam users was between 3 to 6 minutes and less than 0.2 respectively. A summary of call statistics and dataset collected is shown in Table 6.5. Total number of calls between different user groups are shown in Table 6.6.

Table 6.5: Dataset Generated using 10000 VoIP Users

Parameter	**Value**
Number of complete calls	198868
Number of incomplete calls	100609
Total number of VoIP users	010000

Table 6.6: Calls Between Different User Groups in Second Experiment

Calls	**Number of calls**
Normal to Normal	099317
Spam to Normal	100000
Spam to Spam	099477
Normal to Spam	000683

6.4.4 Obtained Calling Parameters

As discussed earlier in Section 6.3.2.2, we consider three parameters to calculate edge weights in the generated *Call Graph*. In this subsection, we present the values of these calling parameters obtained in our experiments.

First Experiment: We can notice from Table 6.4 that the number of calls between normal users to normal users were 49573, normal users to spam users were 427 and spam users to normal users were 50000. Also, 47507 calls were between spam users to spam users to maintain their calling parameters similar to that of normal users. To understand the calling behaviour of normal users and spam users, we calculated the average values of α, τ and ρ for each user which are shown in Figures 6.7, 6.8 and 6.9 respectively. We can notice from Figure 6.7 that the average

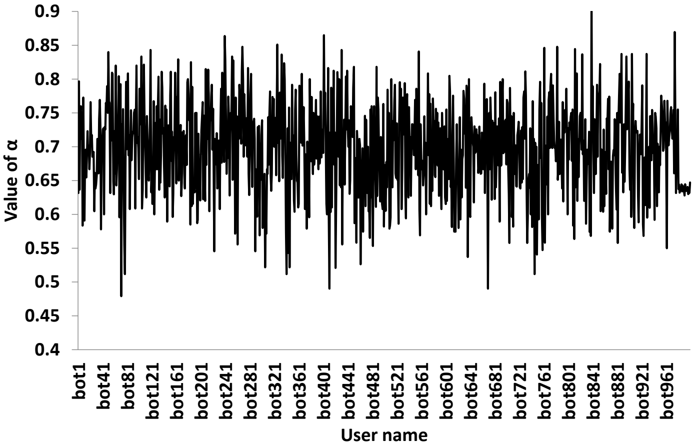


Figure 6.7: Average Successful Call Rate (α) for 1,000 Users

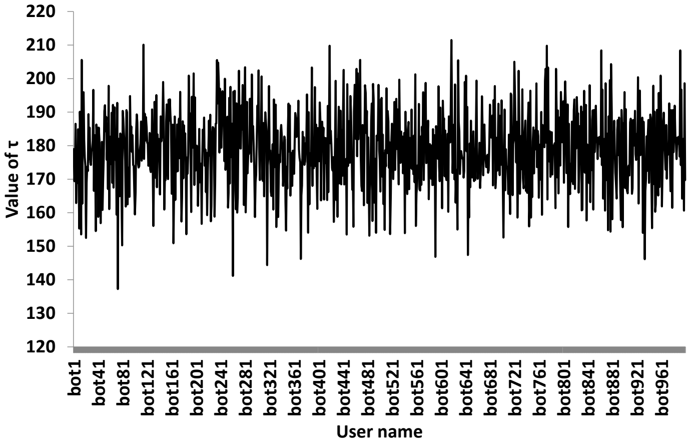


Figure 6.8: Average Talk-time (τ) for 1,000 Users

successful call rate of spam users (bot981 to bot1000) is same as that of normal users. The same is true for the other two parameters except the third parameter of few spam users (showing deviation in Figure 6.9). From these figures, we can clearly notice that spam users were trying to mimic as normal user by maintaining the averages of all the monitored parameters thus posing a challenge in detecting them by simply noticing the call parameters and their average values. However, the calling behaviour of spam users can be analyzed by comparing these parameter values between the spam users group and normal users group. Figures 6.10, 6.11 and 6.12 show these parameter values (for 20 spam users numbered from bot981 to bot1000) between spam to spam users and spam to normal users. We can notice that spam users have large successful

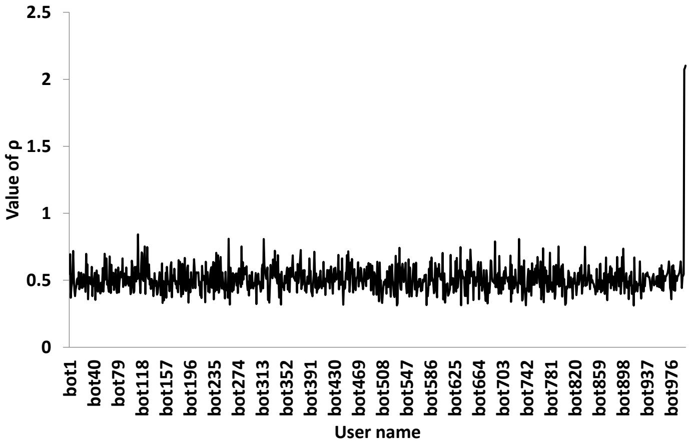


Figure 6.9: User Role In Call (ρ) for 1,000 Users

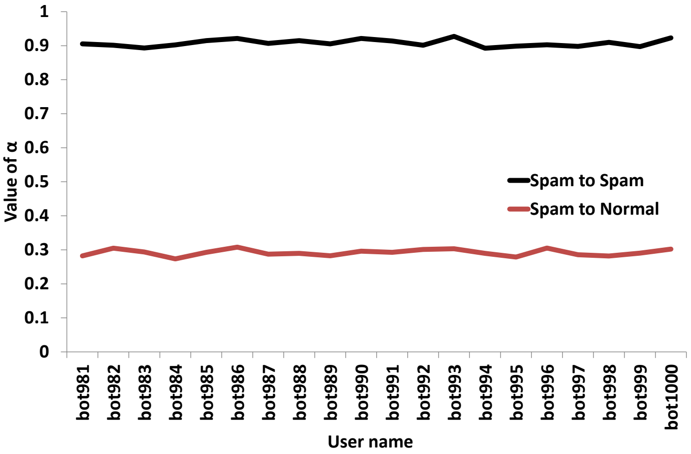


Figure 6.10: Average Successful Call Rate (α) for 20 Spam Users among 1,000 Users

call rate and average talk time among the spam users group that bring the average close to that of normal users. The same is also true for the third parameter as well. Thus any detection method based solely on these parameters can not differentiate a spam user from a normal user. However, since *SpamDetector* uses the social interaction among users and also the neighbourhood similarity, it can identify the spam users.

Second Experiment: We can notice from Table 6.6 that the number of calls between normal users to normal users were 99317, normal users to spam users were 683 and spam users to normal users were 100000. Also, 99477 calls were between spam users to maintain their call parameters similar to that of normal users. The

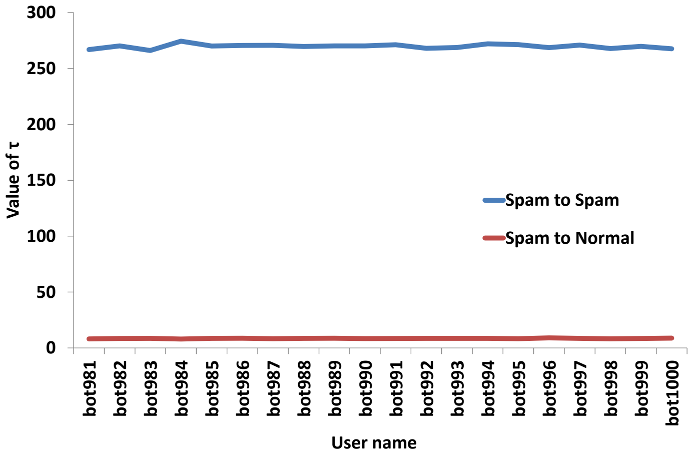


Figure 6.11: Average Talk-time (τ) for 20 Spam Users among 1,000 Users

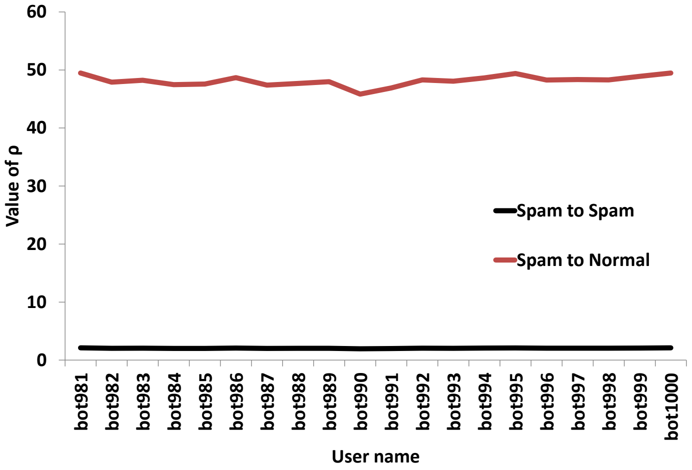


Figure 6.12: Average User Role In Call (ρ) for 20 Spam Users among 1,000 Users

calculated average values of α, τ and ρ for each user considering all their adjacent nodes are shown in Figures 6.13, 6.14 and 6.15 respectively.

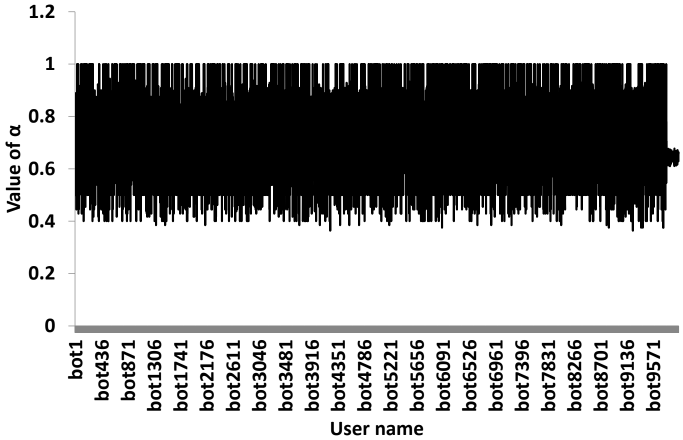


Figure 6.13: Average Successful Call Rate (α) for 10,000 Users

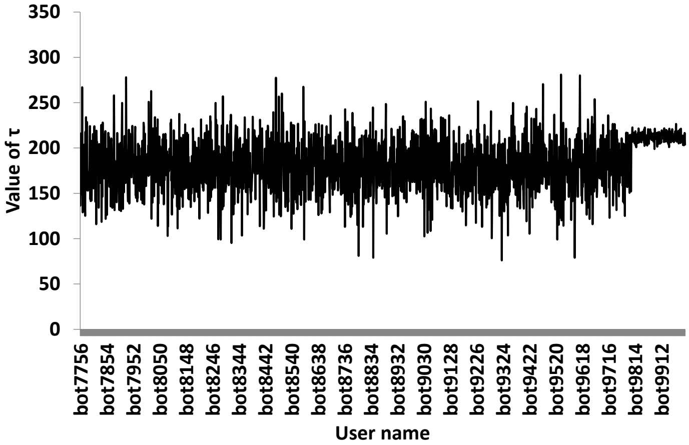


Figure 6.14: Average Talk-time (τ) for 10,000 Users

Similar to the first experiment, we can notice from Figures 6.13, 6.14 and 6.15 that the spam users were trying to mimic the behaviour of normal users in second experiment also. However, the calling behaviour of spam users can be differentiated from the behaviour of normal users by comparing these parameter values between the spam users group and normal users group. Figures 6.16, 6.17 and 6.18 show the average values of α, τ and ρ respectively for 200 spam users (bot9801 to bot10000) in two scenarios- 1) when spam users call spam users and 2) when spam users call normal

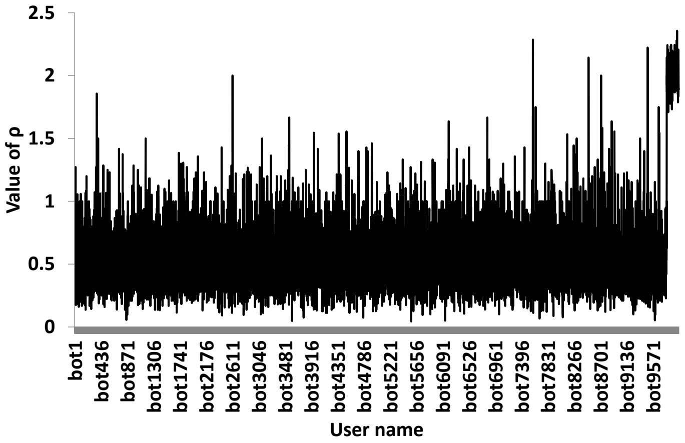


Figure 6.15: User Role In Call (ρ) for 10,000 Users

users. We can notice that in second experiment also, for spam users, the selected parameters average values are closer to the normal users.

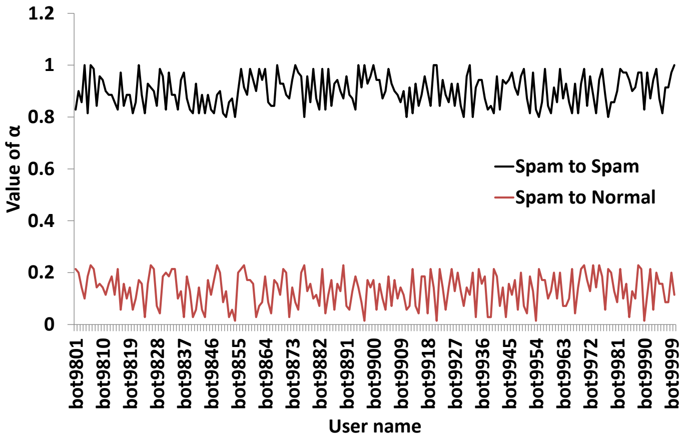


Figure 6.16: Average Successful Call Rate (α) for 200 Spam Users among 10,000 Users

6.4.5 *SpamDetector* Performance

We evaluated the detection performance of *SpamDetector* using the datasets generated from the setup described in the previous subsection. In this subsection, we present the detection results obtained while testing *SpamDetector* on these datasets.

First Experiment: Using the dataset generated from the testbed setup in

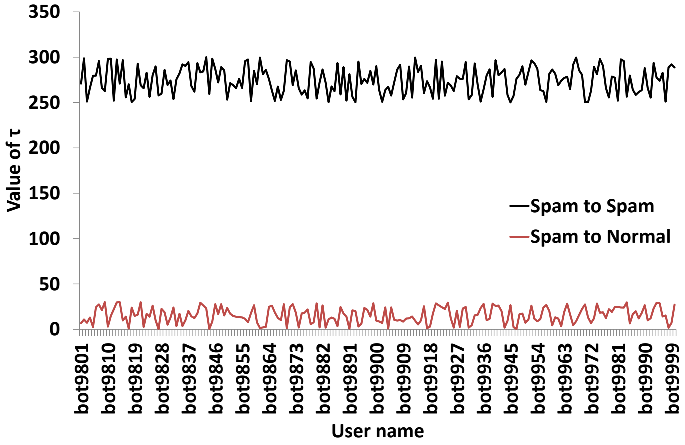


Figure 6.17: Average Talk-time (τ) for 200 Spam Users among 10,000 Users

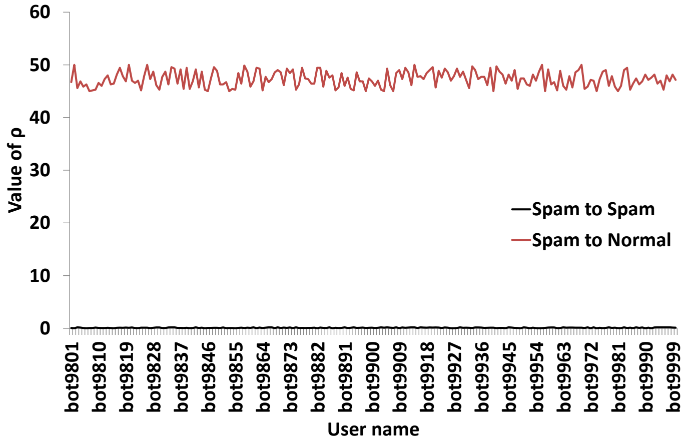


Figure 6.18: Average User Role In Call (ρ) for 200 Spam Users among 10,000 Users

our first experiment, we evaluated the performance of *SpamDetector* to identify spam users (nodes) in the graph. As a first step, the SIP packets were processed to generate CDRs. For processing SIP packets and generating the CDR file of VoIP calls, we wrote a Java program using jNetPcap [10] library which generated a list of 147507 entries. Subsequently, the CDR data was converted into weighted *Call Graph* (represented as adjacency matrix) using another Java program. This program also calculated the α_{ij}, τ_{ij} and ρ_{ij} (ρ_i is used for ρ_{ij}) values for each pair of nodes, normalized the values using Feature Scaling [52] and assigned the weights for each directed edge e_{ij}. As there were 1000 VoIP users in our setup, the generated graph had 1000 nodes and the total number of edges in the graph were 67190. We also implemented *SpamDetector* module in Java which calculated SOF values for each node and with voting, identified the spam users in the graph using the value of N=1 (i.e. standard deviation) in Equation 6.14. We evaluated the performance of our method on the basis of two parameters: *Recall* and *False Positive Rate (FPR)*. Since *SpamDetector* could correctly detect all spam users and no normal users were classified as spam user, the obtained *Recall* and *FPR* of *SpamDetector* were 100% and 0% respectively.

Second Experiment: We used a CDR generation script [25] to generate CDR of 10000 users. We first provided the list of users as input to this script and set flags for normal and spam users as 0 and 1 respectively. We also set a minimum and maximum value of successful call rate for normal users and spam users. The CDR generator script generated successful calls as per this range for different users. Similarly, we set a minimum and maximum value of every call duration for normal users and spam users. We modified the CDR generator script in such a way that when spam user calls to spam user then s/he uses high call duration range whereas when s/he calls to normal user, s/he uses low call duration range. The CDR generated in this case is having same fields as the CDR generated in the first experiment. From CDR, we obtained the *Call Graph* with 10000 nodes and 135698 edges connecting the nodes. Similar to the first experiment, in this case also, *SpamDetector* could correctly detect all spam users without any false positives and so the obtained *Recall* and *FPR* of *SpamDetector* was 100% and 0% respectively. Thus, our proposed method *SpamDetector* is scalable and can be implemented to identify spam users even if the VoIP network contains a large number of users.

6.4.6 Sensitivity Analysis

SpamDetector uses a threshold on the ratio of edge weight under consideration and average SOF of peer node to vote the node as anomaly or normal. Thus, this threshold value governs the detection performance of *SpamDetector*. To understand the impact of varying this threshold value on *Recall* and *FPR*, we varied the threshold value by setting different values of N in Equation 6.14 between -20 to 20 and obtained the detection results. Figure 6.19 shows the effect of varying N on *Recall* and *FPR* of *SpamDetector*. We can notice that as the threshold increases, *Recall* decreases and at the same time *FPR* also decreases. Thus an appropriate threshold value should be set balancing these two.

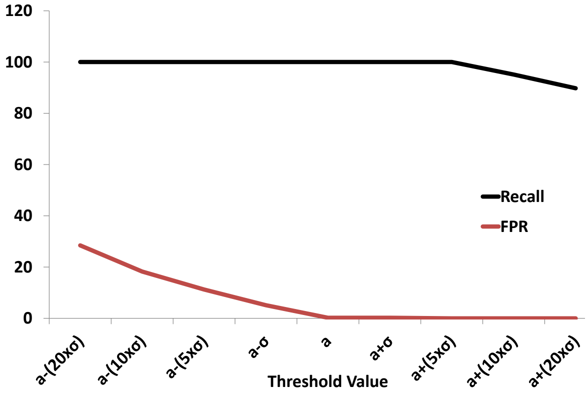


Figure 6.19: Effect of Varying N on Detection Performance

6.4.7 Performance Comparison and Discussion

We compared the performance of *SpamDetector* with a recent work described by Toyoda et al. [106] by testing their method on our dataset. Their method detects spam users by clustering VoIP users on the basis of their call behaviour. The call behaviour of each VoIP user is described using 5 call parameters as described below:

1. Average Call Duration (ACD): ACD of a user u is the average talk time per call of the user. ACD of a spam user is small as s/he has low talk-time per call.
2. Call Frequency Per Day (CPD): CPD of a user u is the average number of calls by the user per day. CPD of a spam user is high as s/he makes large number of calls per day[6].
3. Strong Ties (ST): ST of a user u corresponds to the ratio of total call duration of the top 5 callees (based on the call duration) to the total call duration of the user. ST of a spam user is high as s/he has small total call duration.
4. Weak Ties (WT): WT of a user u is the fraction of number of callees that talk for more than 60 seconds to the user. WT of a spam user is small as there are less number of users who talk to a spam user for more than 60 seconds.
5. Incoming to outgoing Ratio (IOR): IOR of a user u is the ratio of number of calls received by the user to the number of times the user called. IOR of a spam user is small as s/he mostly behaves as a caller.

Example: Consider a VoIP network having five users as u_1, u_2, u_3, u_4 and u_5. The CDR generated from the social interaction among these users in a day is shown in Table 6.7. From this table, we can notice that each user calls two times in a day. From the given CDR, ACD of u_1, u_2, u_3, u_4 and u_5 is calculated as 35, 60, 5, 100 and 15 seconds respectively. Moreover, CFD of all these users is 2 as all of them call two times in a day. In the given example, total call duration of all the (top) 5 callees is 430 seconds. Thus, ST for users is calculated as 6.14, 3.58, 43, 2.15 and 14.33. The user u_1 called u_4 and talked for 70 seconds whereas u_1 could not establish a call with u_2. Thus, WT for u_1 is calculated as 0.5. Similarly, for other four users, WT is calculated as 1, 0, 1 and 0.5. The last parameter IOR for users u_1, u_2, u_3, u_4

[6] For our experiments, we considered the total number of calls attempted by a user during the simulation period as the CPD of that user.

Table 6.7: Example Call Detail Records

Caller	Callee	Talk-time (in seconds)
u_1	u_4	070
u_1	u_2	000
u_2	u_3	040
u_2	u_1	080
u_3	u_5	000
u_3	u_1	010
u_4	u_1	120
u_4	u_2	080
u_5	u_2	030
u_5	u_2	000

and u_5 is calculated as 1.5, 2.0, 0.5, 0.5 and 0.5 respectively.

The values of these 5 parameters are passed to two clustering algorithms namely K-means clustering and combination of Random Forest (RF) with Partitioning Around Medoids (PAM). These clustering algorithms form two clusters on the basis of these call parameters. One cluster is of normal users and other cluster is of spam users such that values of all the 5 parameters for spam users is lower than that of normal users. We calculated all these parameters over our dataset and used both the clustering algorithms available in a machine learning tool Knime [11]. The performance comparison of both the methods, *SpamDetector* and that of Toyoda et al., is shown in the Table 6.8. We can notice that there is a significant drop in the detection performance of method proposed in [106]. This lower detection rate and higher false alarm rate is justified by the fact that they use the raw values of average call duration, success rate, etc. irrespective of which interactions among the users contribute to these values. However, *SpamDetector* uses the neighbourhood relationship and can precisely differentiate to which peers a node is behaving differently and hence can identify the spam users although the averages look similar.

Table 6.8: *SpamDetector* Performance Comparison with Toyoda et. al [106] Method

Parameter	*SpamDetector* (for 1000 Users)	Toyoda et al. [106]
Recall (in %)	100.00	061.23
FPR (in %)	000.00	022.71

6.4.8 Limitations

Although our proposed method *SpamDetector* performs well even if a spam user maintains the averages of parameters used for differentiating a spam user from a normal user, it also has few limitations as follows:

- As *SpamDetector* relies on the majority voting for declaring a node as spam in the graph, a tie in voting needs to be resolved. This can be done by either considering the node receiving equal votes as normal or anomaly.
- Since *SpamDetector* calculates the SOF values of all the neighbouring nodes of a node under consideration, this is computationally expensive and time consuming to perform, particularly, if the graph is dense (nearly complete graph).
- We tested the detection performance of *SpamDetector* on the datasets generated by mimicking behaviour of real VoIP users in a simulated environment only.

6.5 Conclusion

VoIP involves transmitting voice and multimedia content between users over IP networks. Most VoIP systems use Session Initiation Protocol (SIP) for control plane operation. However, VoIP's low cost communication makes it a popular target for spam users. In this chapter, we described *SpamDetector*, a method for VoIP spam user detection based on the VoIP user's calling behaviour. *SpamDetector* first generates a call graph using call detail records of VoIP users. Subsequently, it uses the calling behaviour between the users through a set of chosen parameters and identifies anomalies in the generated call graph. Anomalies are identified by considering the local neighbourhood of a node and deriving *Spam Outlier Factor* with each of its neighbours. A node is declared as anomaly if it is different with respect to majority of its neighbours. We evaluated the performance of *SpamDetector* to identify spam users on two simulated datasets and reported that it can detect spam users with very high accuracy. We also compared *SpamDetector* with a recent work and showed that *SpamDetector* achieves better *Recall* without any false positives.

Chapter 7

Conclusion

In this chapter, we summarize the DPI based methods presented in the thesis to monitor network traffic for three different purposes - 1) *Traffic classification*, 2) *Zero day attack detection in web traffic* and 3) *Spam user(s) detection in VoIP networks.* The goal of our work was to propose such methods which can monitor the network traffic for these purposes and give accurate classification/detection results.

We first motivated our thesis by stating that monitoring network traffic is essential for various purposes as it gives an in-depth insight of what type of data is flowing through a network. We presented some recent trends which report that the number of Internet users and hence the services offered through Internet are on a rapid rise that results into generation of voluminous and heterogeneous traffic. This makes the network monitoring a challenging task. We also pointed out that DPI based methods are more accurate than statistical methods, however, DPI based methods also suffer from problems such as high computational complexity and compromise user's privacy. Therefore, the goal of the research presented in this thesis was to provide novel solutions for network traffic classification, zero day attack detection and VoIP spam detection which are accurate and at the same time, computationally less expensive as well. In this direction, we first proposed a traffic classification method *RDClass* which analyzes payload at byte level and can accurately classify the application protocols in the network traffic. However, *RDClass* works only for text based protocols. In order to handle binary protocols, we proposed another traffic classification method *BitCod-*

ing and *BitProb* which analyze payload at bit level and can accurately classify text, binary and proprietary protocols in the network traffic. We subsequently proposed two methods - *Rangegram* and *OCPAD* to detect zero day attacks in web traffic. First method *Rangegram* extracts n-grams with their occurrence frequency from the payload of each HTTP packet and use it to detect zero day attacks. Our second anomaly detection method *OCPAD* detects anomalies in web traffic by checking the occurrence probability of n-grams of the payload of HTTP packets. As our last contribution, we proposed *SpamDetector* which can detect spam users in a VoIP network by analyzing their calling behaviour using a call-graph.

7.1 Thesis Contributions

We recap the thesis contributions in this section.

7.1.1 *RDClass* - A Byte Level Payload Analysis Method for Network Traffic Classification

We proposed *RDClass* which is an unsupervised payload based traffic classification method for classifying network flows. *RDClass* uses a set of keywords extracted from flow payload and uses the relative distance between the pair of consecutive keywords to identify application protocol. We represented the set of keywords and their relative distances in the form of a state transition machine called *Relative Distance Constrained Counting Automata* (*RDCCA*). We evaluated the performance of *RDClass* on three different public and private datasets and compared with a recent and closely related work and showed that *RDClass* gives better classification accuracy. We also discussed the limitation of *RDClass* as it can classify only text based protocols and fails to detect binary and/or proprietary protocols.

7.1.2 *BitCoding* and *BitProb* - Bit Level Payload Analysis Methods for Network Traffic Classification

To handle both text-based and binary protocol, we proposed two supervised traffic classification methods *BitCoding* and *BitProb* which generate application specific signatures using only a small number of initial bits extracted from a flow. *BitCoding* encodes generated signatures of each application in order to compress them and then transforms the compressed signatures into a state transition machine called *Transition Constrained Counting Automata* (*TCCA*). This *TCCA* is subsequently used for classification purpose. Since *BitCoding* considers only invariant bits while matching application signatures, this increases the chances of signature overlap which may lead to misclassifications. However, our second method *BitProb* considers the occurrence probability of bit values at each position without omitting any bit value during signature match due to which it can identify applications more accurately. Similar to *BitCoding*, *BitProb* also transforms the generated signatures into a state transition machine called *Probabilistic Counting Deterministic Automata* (*PCDA*). Our proposed methods generate short application specific signatures and thus, are computationally less expensive. We evaluated the classification performances of *BitCoding* and *BitProb* on two public and one private datasets and showed that they can classify different text, binary and proprietary protocols with very high accuracy which makes them protocol agnostic. We performed cross site experiments and showed that the generated signatures are robust. We also compared the classification performance of *BitCoding* and *BitProb* with a recent and closely related work and showed that our methods outperform *BitFlow*.

7.1.3 *Rangegram* and *OCPAD* - Zero Day Attack Detection in Web Traffic

We proposed two methods *Rangegram* and *OCPAD* to detect zero day attacks in the web traffic. These methods use DPI for extracting n-grams from packet payloads. The first method *Rangegram* uses occurrence frequencies of n-grams to detect anomalous packet payloads. *Rangegram* creates a normal HTTP profile by extracting n-grams (short sequences of length n from a string) from HTTP payload and storing the minimum and maximum occurrence frequency of each n-gram along with the n-gram in a data structure called *Min-Max-Tree*. Subsequently, *Rangegram* uses a rating function to find those HTTP packets for which the generated n-grams deviate from the normal HTTP profile. If the observed deviation is greater than a predefined threshold, the HTTP packet under consideration is declared as anomalous. Our second anomaly detection method *OCPAD* uses a version of Multinomial Bayesian one class classification technique for accurately detecting anomalous payloads. In particular, *OCPAD* uses likelihood of n-gram occurrence in a payload of known non-malicious HTTP packets as a measure to derive the degree of maliciousness of a packet. *OCPAD* first calculates the likelihood range of each n-gram occurrence from every packet and stores it along with the n-gram in a proposed efficient data structure called *Probability-Tree*. Subsequently, if the n-grams generated from the payload of HTTP packet under consideration is not found in the database or its occurrence probability in a packet is not in the range stored in the *Probability-Tree*, *OCPAD* considers the packet as anomalous. We evaluated both of our proposed methods on a common dataset and compared their detection performance. We also compared *Rangegram* and *OCPAD* with a closely related work and showed that both of our methods outperform the previously known method in terms of detection accuracy.

7.1.4 *SpamDetector* - Detecting Spam Users in VoIP Networks

As our last contribution, we proposed *SpamDetector*, a graph based method to detect VoIP spam users in a network. *SpamDetector* examines SIP packets in network traffic and creates a directed and weighted call graph using *Call Detail Records (CDR)* of users where a set of differentiating call parameters are used to derive weights on the edges. Subsequently, *SpamDetector* analyzes the call graph for spam user detection. *SpamDetector* identifies anomalies in the graph by considering the local neighbourhood of a node and assigning a label based on how similar the node is in comparison to its neighbours. To find how similar a node is with its neighbours, we defined a new parameter called *Spam Outlier Factor (SOF)*. If the calculated *SOF* of the node under consideration is greater than the threshold *SOF* of a neighbouring node, it votes the node under consideration as spam. In a similar way, other neighbouring nodes also votes the considered node as normal or spam depending on their threshold *SOF*. If the spamming votes are found in majority, *SpamDetector* eventually declares the node as spam. We evaluated the detection performance of *SpamDetector* on two different datasets and showed that *SpamDetector* could correctly detect all spam nodes. Using our datasets, we also compared the detection performance of *SpamDetector* with one of the closely related work and showed that *SpamDetector* outperforms the previously known method in terms of both *Recall* and *FPR*.

7.2 Future Work

Our work on traffic classification and detecting zero day attacks can be extended in several ways. Some of the future directions are discussed in next few subsections.

7.2.1 Classification of Mobile Applications in Network Traffic

There are approximately six million mobile applications available in the wild. These applications generate voluminous and heterogeneous network traffic and majority of them use HTTP for the purpose of communication and data transfer. As a result, misleading classifications are evident which make classifying mobile application traffic a very challenging task. For network traffic classification, we made three contributions in this thesis - *RDClass*, *BitCoding* and *BitProb*. We showed that these methods could accurately classify application layer protocols. However, these methods cannot detect those applications which are using these application layer protocols for the purpose of communication and data transfer. Thus, extending our classification methods to generate unsupervised bit level signatures for mobile applications is a useful extension so that they can detect mobile applications also.

7.2.2 Detection of Zero Day Attacks in Binary Protocols

We proposed two methods, *Rangegram* and *OCPAD*, to detect zero day attacks particularly in web traffic. These methods involve extraction of n-grams from the payload of HTTP packets to generate normal profile and subsequently use the generated profile to detect anomalous packets. Since HTTP is a text protocol, it is easy to generate n-grams and compare them from the normal profile. However, as we discussed in Chapter 4, several popular application layer protocols such as NTP or DNS are encoded at the bit level. As a result, several previously known methods (including our proposed methods *Rangegram* and *OCPAD*) which use n-grams as features to analyze payload may not work effectively [59]. Therefore, it is an interesting future research problem how the proposed methods can be modified so that they can detect zero day attacks in binary protocols also.

Bibliography

[1] Alexa Websites. https://www.alexa.com/topsites/countries/IN, Last accessed: January 2018.

[2] Asterisk. http://www.asterisk.org/, Last accessed: July 2018.

[3] Bittorrent. http://www.bittorrent.com, Last accessed: July 2018.

[4] Bloom Filters. http://llimllib.github.io/bloomfilter-tutorial/, Last accessed: January 2018.

[5] Digital Corpora. http://digitalcorpora.org/, Last accessed: January 2018.

[6] Dropbox. https://www.dropbox.com/, Last accessed: July 2018.

[7] Internet World Stats. https://www.internetworldstats.com/emarketing.htm, Last accessed: May 2018.

[8] Jain SIP API. https://www.oracle.com/technetwork/java/introduction-jain-sip-090386.html, Last accessed: August 2018.

[9] Java Media Framework. https://www.oracle.com/technetwork/java/javase/tech/index-jsp-140239.html, Last accessed: August 2018.

[10] jNetPcap. https://sourceforge.net/projects/jnetpcap/, Last accessed: October 2018.

[11] Knime. https://www.knime.com/, Last accessed, September 2018.

[12] L7-filter. http://l7-filter.clearos.com/, Last accessed: March 2018.

[13] libprotoident. https://research.wand.net.nz/software/libprotoident.php, Last accessed: March 2018.

[14] Mergecap. https://www.wireshark.org/docs/man-pages/mergecap.html, Last accessed: March 2018.

[15] NBAR. https://www.cisco.com/c/en/us/products/ios-nx-os-software/network-based-application-recognition-nbar/index.html, Last accessed: March 2018.

[16] nDPI. https://www.ntop.org/products/deep-packet-inspection/ndpi/, Last accessed: March 2018.

[17] Netresec. http://www.netresec.com/?page=pcapfiles, Last accessed: January 2018.

[18] OpenDPI. https://sourceforge.net/projects/opendpigt/, Last accessed: March 2018.

[19] OpenSSH. https://help.ubuntu.com/lts/serverguide/openssh-server.html.en, Last accessed: June 2018.

[20] PACE. https://www.ipoque.com/products/dpi-engine-rsrpace-2, Last accessed: March 2018.

[21] Polymorphic Shellcode Engine. http://phrack.org/issues/61/9.html, Last accessed: April 2018.

[22] Postfix Server. http://www.postfix.org/, Last accessed: March 2018.

[23] Snort. https://www.snort.org/, Last accessed: May 2018.

[24] Tcpdump. http://www.tcpdump.org/, Last accessed: September 2018.

[25] Telecom Record Generator. https://github.com/deshpandetanmay/cdr-data-generator, Last accessed: August 2018.

[26] Voice over IP: Protocols and Standards. https://www.cse.wustl.edu/ jain/cis788-99/ftp/voip-protocols/index.html, Last accessed: April 2018.

[27] VoIP PBX. https://www.voip-info.org/pbx/, Last accessed: June 2018.

[28] Aceto, G., Ciuonzo, D., Montieri, A., and Pescapé, A. Multi-classification Approaches for Classifying Mobile App Traffic. *Journal of Network and Computer Applications 103* (2018), 131–145.

[29] Akoglu, L., Tong, H., and Koutra, D. Graph based Anomaly Detection and Description: A Survey. *Data Mining and Knowledge Discovery 29* (2015), 626–688.

[30] Alizadeh, H., Khoshrou, S., and Zúquete, A. Application Specific Traffic Anomaly Detection Using Universal Background Model. In *Proceedings of the 5^{th} ACM International Workshop on International Workshop on Security and Privacy Analytics (IWSPA '15)* (2015), pp. 11–17.

[31] Angelov, P., and Parvanov, Gu, X. *Empirical Methods for Machine Learning*, 1^{st} ed. Springer, 2017.

[32] Angiulli, F., Argento, L., and Furfaro, A. Exploiting N-gram Location for Intrusion Detection. In *Proceedings of the 27^{th} IEEE International Conference on Tools with Artificial Intelligence (ICTAI '15)* (2015), pp. 1093–1098.

[33] Antonello, R., Fernandes, S., Kamienski, C., Sadok, D., Kelner, J., GóDor, I., Szabó, G., and Westholm, T. Deep Packet Inspection Tools and Techniques in Commodity Platforms: Challenges and Trends. *Journal of Network and Computer Applications 35* (2012), 1863–1878.

[34] Ariu, D., Tronci, R., and Giacinto, G. HMMPayl: An Intrusion Detection System based on Hidden Markov Models. *Computers & Security 30* (2011), 221–241.

[35] Aurrecoechea, C., Campbell, A. T., and Hauw, L. A Survey of QoS Architectures. *Multimedia Systems 6* (1998), 138–151.

[36] Azad, M. A., and Morla, R. Caller-REP: Detecting Unwanted Calls with Caller Social Strength. *Computers & Security 39* (2013), 219–236.

[37] Azad, M. A., Morla, R., Arshad, J., and Salah, K. Clustering VoIP Caller for SPIT Identification. *Security and Communication Networks 9* (2016), 4827–4838.

[38] Azad, M. A., Morla, R., and Salah, K. Systems and Methods for SPIT Detection in VoIP: Survey and Future Directions. *Computers & Security 77* (2018), 1–20.

[39] Balasubramaniyan, V., Ahamad, M., and Park, H. CallRank: Combating SPIT Using Call Duration, Social Networks and Global Reputation. In *Proceedings of the 4th International Conference on Email and Anti-Spam (CEAS '07)* (2007), pp. 1–8.

[40] Bujlow, T., Carela Español, V., and Barlet Ros, P. Independent Comparison of Popular DPI Tools for Traffic Classification. *Computer Networks 76* (2015), 75–89.

[41] Callado, A., Kamienski, C., Szabó, G., Gero, B. P., Kelner, J., Fernandes, S., and Sadok, D. A Survey on Internet Traffic Identification. *IEEE Communications Surveys & Tutorials 11* (2009), 37–52.

[42] Cohen, P. R. *Empirical Methods for Artificial Intelligence*, 1st ed. MIT Press, 1995.

[43] Dainotti, A., Pescape, A., and Claffy, K. Issues and Future Directions in Traffic Classification. *Networking Magazine of Global Internetwoking 26* (2012), 35–40.

[44] Dalgic, I., and Fang, H. Comparison of H.323 and SIP for IP Telephony Signaling. In *Proceedings of the 1st International Conference on Multimedia Systems and Applications (MSA'99)* (1999), vol. 3845, pp. 106–122.

[45] Dantu, R., and Kolan, P. Detecting Spam in VoIP Networks. In *Proceedings of the* 1st *Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI '05)* (2005), pp. 1–7.

[46] Dwivedi, H. *Hacking VoIP: Protocols, Attacks, and Countermeasures*, 1st ed. No Starch Press, 2008.

[47] Ferdous, R., Cigno, R. L., and Zorat, A. Classification of SIP Messages by a Syntax Filter and SVMs. In *Proceedings of the* 13th *IEEE International Global Communications Conference (GLOBECOM '12)* (2012), pp. 2714–2719.

[48] Fernandes, S., Antonello, R., Lacerda, T., Santos, A., Sadok, D., and Westholm, T. Slimming Down Deep Packet Inspection Systems. In *Proceedings of the* 28th *IEEE International Conference on Computer Communications (INFOCOM '09)* (2009), pp. 1–6.

[49] Finamore, A., Mellia, M., Meo, M., and Rossi, D. Kiss: Stochastic Packet Inspection Classifier for UDP Traffic. *IEEE/ACM Transactions on Networking 18* (2010), 1505–1515.

[50] Finsterbusch, M., Richter, C., Rocha, E., Muller, J.-A., and Hanssgen, K. A Survey of Payload-based Traffic Classification Approaches. *IEEE Communications Surveys & Tutorials 16* (2014), 1135–1156.

[51] Fogla, P., Sharif, M. I., Perdisci, R., Kolesnikov, O. M., and Lee, W. Polymorphic Blending Attacks. In *Proceedings of the* 15th *USENIX Security Symposium (USENIX '06)* (2006), pp. 241–256.

[52] Forman, G. BNS Feature Scaling: An Improved Representation Over TF-IDF for SVM Text Classification. In *Proceedings of the* 17th *ACM Conference on Information and Knowledge Management (CIKM' 08)* (2008), pp. 263–270.

[53] Gebali, F. *Analysis of Computer Networks*, 2nd ed. Springer, 2016.

[54] Golait, D., and Hubballi, N. VoIPFD: Voice over IP Flooding Detector. In *Proceedings of the* 22^{nd} *National Conference on Communication (NCC '16)* (2016), pp. 1–6.

[55] Golait, D., and Hubballi, N. Detecting Anomalous Behavior in VoIP Systems: A Discrete Event System Modeling. *IEEE Transactions on Information Forensics and Security 12* (2017), 730–745.

[56] Gómez Sena, G., and Belzarena, P. Early Traffic Classification using Support Vector Machines. In *Proceedings of the* 5^{th} *International Latin American Networking Conference (LANC' 09)* (2009), pp. 60–66.

[57] Görnitz, N., Kloft, M., Rieck, K., and Brefeld, U. Toward Supervised Anomaly Detection. *Journal of Artificial Intelligence Research 46* (2013), 235–262.

[58] Grus, J. *Data Science from Scratch: First Principles with Python*, 1^{st} ed. O' Reilly, 2015.

[59] Hadžiosmanović, D., Simionato, L., Bolzoni, D., Zambon, E., and Etalle, S. N-gram Against the Machine: On the Feasibility of the N-gram Network Analysis for Binary Protocols. In *Proceedings of the* 12^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID'12)* (2012), pp. 354–373.

[60] Haffner, P., Sen, S., Spatscheck, O., and Wang, D. ACAS: Automated Construction of Application Signatures. In *Proceedings of the* 1^{st} *ACM SIGCOMM Workshop on Mining Network Data (MineNet '05)* (2005), pp. 197–202.

[61] Han, J., Pei, J., Yin, Y., and Mao, R. Mining Frequent Patterns without Candidate Generation. *Data Mining and Knowledge Discovery 8* (2004), 53–87.

[62] Hansen, M., Hansen, M., Möller, J., Rohwer, T., Tolkmit, C., and Waack, H. Developing a Legally Compliant Reachability Management System

as a Countermeasure Against SPIT. In *Proceedings of the* 3^{rd} *Annual VoIP Security Workshop (VSW '06)* (2006), pp. 1–7.

[63] Hartigan, J. A., and Wong, M. A. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society 28* (1979), 100–108.

[64] Hastie, T., Tibshirani, R., and Friedman, J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2^{nd} ed. Springer, 2017.

[65] Hubballi, N., Biswas, S., and Nandi, S. Layered Higher Order N-grams for Hardening Payload based Anomaly Intrusion Detection. In *Proceedings of the* 5^{th} *International Conference on Availability, Reliability and Security (ARES '10)* (2010), pp. 321–326.

[66] Ingham, K. L., and Inoue, H. Comparing Anomaly Detection Techniques for HTTP. In *Proceedings of the* 10^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID '07)* (2007), pp. 42–62.

[67] J. Rosenberg, C. Jennings. (RFC 5039) The Session Initiation Protocol (SIP) and Spam, Last accessed: July 2018.

[68] Jamdagni, A., Tan, Z., He, X., Nanda, P., and Liu, R. P. Repids: A Multi Tier Real-Time Payload-based Intrusion Detection System. *Computer Networks 57* (2013), 811–824.

[69] Kang, H. J., Zhang, Z. L., Ranjan, S., and Nucci, A. SIP-based VoIP Traffic Behavior Profiling and its Applications. In *Proceedings of the* 3^{rd} *Annual ACM Workshop on Mining Network Data (MineNet '07)* (2007), pp. 39–44.

[70] Karopoulos, G., Kambourakis, G., and Gritzalis, S. Caller Identity Privacy in SIP Heterogeneous Realms: A Practical Solution. In *Proceedings of the* 13^{th} *IEEE Symposium on Computers and Communications (ISCC '08)* (2008), pp. 37–43.

[71] Kaur, R., and Singh, M. A Survey on Zero-Day Polymorphic Worm Detection Techniques. *IEEE Communications Surveys & Tutorials 16* (2014), 1520–1549.

[72] Kim, S. B., Han, K. S., Rim, H. C., and Myaeng, S. H. Some Effective Techniques for Naive Bayes Text Classification. *IEEE Transactions on Knowledge and Data Engineering 18* (2006), 1457–1466.

[73] Kolan, P., and Dantu, R. Socio-Technical Defense against Voice Spamming. *ACM Transactions on Autonomous and Adaptive Systems 2* (2007), 1–44.

[74] Lee, J., Cho, K., Lee, C., and Kim, S. VoIP-Aware Network Attack Detection Based on Statistics and Behavior of SIP Traffic. *Peer-to-Peer Networking and Applications 8* (2015), 872–880.

[75] Lin, Z., Jiang, X., Xu, D., and Zhang, X. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Proceedings of the* 15^{th} *Network and Distributed System Security Symposium (NDSS '08)* (2008), pp. 1–15.

[76] Liu, C., and Wu, J. Fast Deep Packet Inspection with a Dual Finite Automata. *IEEE Transactions on Computers 62* (2013), 310–321.

[77] Liu, H., and Mouchtaris, P. Voice over IP Signaling: H. 323 and Beyond. *IEEE Communications Magazine 38* (2000), 142–148.

[78] Liu, J., Rahbarinia, B., Perdisci, R., Du, H., and Su, L. Augmenting Telephone Spam Blacklists by Mining Large CDR Datasets. In *Proceedings of the 13th ACM Asia Conference on Computer and Communications Security(AsiaCCS'18)* (2018), pp. 273–284.

[79] McCallum, A., and Nigam, K. A Comparison of Event Models for Naive Bayes Text Classification. In *Proceedings of the* 15^{th} *International Workshop on Association for the Advancement of Artificial Intelligence (AAAI '98)* (1998), pp. 41–48.

[80] Micro, T. Targeted Attack Campaigns and Trends. Annual report, R & D Centre, Trend Micro, 2014.

[81] Nassar, M., Festor, O., et al. Labeled VoIP Data-Set for Intrusion Detection Evaluation. In *Proceedings of the* 4th *European Network of Universities and Companies in Information and Communication Engineering (EUNICE '10)* (2010), pp. 97–106.

[82] Nguyen, T. T., and Armitage, G. A Survey of Techniques for Internet Traffic Classification using Machine Learning. *IEEE Communications Surveys & Tutorials 10* (2008), 56–76.

[83] Oza, A., Ross, K., Low, R. M., and Stamp, M. HTTP Attack Detection using N-gram Analysis. *Computers & Security 45* (2014), 242–254.

[84] Park, B. C., Won, Y. J., Kim, M. S., and Hong, J. W. Towards Automated Application Signature Generation for Traffic Identification. In *Proceedings of the* 9th *IEEE Network Operations and Management Symposium (NOMS '08)* (2008), pp. 160–167.

[85] Parsian, M. *Data Algorithms*, 1st ed. O'Reilly, 2015.

[86] Parsons, C. Deep Packet Inspection in Perspective: Tracing its Lineage and Surveillance Potentials, 2008. Working Paper of University of Victoria.

[87] Pastrana, S., Orfila, A., Tapiador, J. E., and Peris Lopez, P. Randomized Anagram Revisited. *Journal of Network and Computer Applications 41* (2014), 182–196.

[88] Paxson, V. https://www.bro.org/, Last accessed: March 2018.

[89] Perdisci, R., Ariu, D., Fogla, P., Giacinto, G., and Lee, W. McPAD: A Multiple Classifier System for Accurate Payload-based Anomaly Detection. *Computer Networks 53* (2009), 864–881.

[90] Perdisci, R., Gu, G., and Lee, W. Using an Ensemble of One-Class SVM Classifiers to Harden Payload-based Anomaly Detection Systems. In *Proceedings of the* 6th *International Conference on Data Mining (ICDM'06)* (2006), pp. 488–498.

[91] Pountain, D. Run Length Encoding. *Byte 12* (1987), 317–319.

[92] Rätsch, G., Onoda, T., and Müller, K. Soft Margins for AdaBoost. *Machine Learning 42* (2001), 287–320.

[93] Rosen, S. *Programming Systems and Languages*, 1st ed. McGraw Hill, 1967.

[94] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and Schooler, E. (RFC 3261) SIP: Session Initiation Protocol, Last accessed: July 2018.

[95] Roughan, M., Sen, S., Spatscheck, O., and Duffield, N. Class-of-Service Mapping for QoS: A Statistical Signature-Based Approach to IP Traffic Classification. In *Proceedings of the* 4th *ACM SIGCOMM Conference on Internet Measurement (IMC '04)* (2004), pp. 135–148.

[96] Sato, I., and Nakagawa, H. Topic Models with Power-Law using Pitman-Yor Process. In *Proceedings of the* 16th *ACM International Conference on Knowledge Discovery and Data Mining (SIGKIDD' 10)* (2010), pp. 673–682.

[97] Schulzrinne, H., Rosenberg, J., et al. A Comparison of SIP and H. 323 for Internet Telephony. In *Proceedings of the* 8th *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '98)* (1998), pp. 83–86.

[98] Sengar, H., Wang, X., and Nichols, A. Call Behavioral Analysis to Thwart SPIT Attacks on VoIP Networks. In *Proceedings of the* 3rd *International Conference on Security and Privacy in Mobile Information and Communication Systems (MobiSec '11)* (2011), pp. 501–510.

[99] Shin, D., Ahn, J., and Shim, C. Progressive Multi Gray-Leveling: A Voice Spam Protection Algorithm. *IEEE Network 20* (2006), 18–24.

[100] St Sauver, J. A Look at the Unidentified Half of Netflow. In *Proceedings of the* 1st *International Conference of Networking Engineers (ICNE'07)* (2007), pp. 1–10.

[101] Stanek, J., Kencl, L., and Kuthan, J. Analyzing Anomalies in Anonymized SIP Traffic. In *Proceedings of the* 13th *International Networking Conference (IFIP' 14)* (2014), pp. 1–9.

[102] Sun, M., and Chen, T. Network Intrusion Detection System, 2010. US Patent App. 12/411,916, Google Patents.

[103] Taylor, V. F., Spolaor, R., Conti, M., and Martinovic, I. Robust Smartphone App Identification via Encrypted Network Traffic Analysis. *IEEE Transactions on Information Forensics and Security 13* (2018), 63–78.

[104] Tongaonkar, A., Keralapura, R., and Nucci, A. SANTaClass: A Self Adaptive Network Traffic Classification System. In *Proceedings of the* 13th *IFIP Networking Conference (IFIP '13)* (2013), pp. 1–9.

[105] Tongaonkar, A., Torres, R., Iliofotou, M., Keralapura, R., and Nucci, A. Towards Self Adaptive Network Traffic Classification. *Computer Communications 56* (2015), 35–46.

[106] Toyoda, K., Park, M., Okazaki, N., and Ohtsuki, T. Novel Unsupervised Spitters Detection Scheme by Automatically Solving Unbalanced Situation. *IEEE Access 5* (2017), 6746–6756.

[107] Toyoda, K., and Sasase, I. SPIT Callers Detection with Unsupervised Random Forests Classifier. In *Proceedings of the* 19th *IEEE International Conference on Communications (ICC '13)* (2013), pp. 2068–2072.

[108] Tu, H., Doupé, A., Zhao, Z., and Ahn, G.-J. SoK: Everyone Hates Robocalls: A Survey of Techniques against Telephone Spam. In *Proceedings of the 37th International Conference on Security and Privacy (S&P '16)* (2016), pp. 320–338.

[109] Tu, H., Doupé, A., Zhao, Z., and Ahn, G.-J. Sok: Everyone Hates Robocalls: A Survey of Techniques against Telephone Spam. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP'16)* (2016), pp. 320–338.

[110] Vasilomanolakis, E., Karuppayah, S., Mühlhäuser, M., and Fischer, M. Taxonomy and Survey of Collaborative Intrusion Detection Systems. *ACM Computing Surveys 47* (2015), 1–55.

[111] Vasquez, J., and Desai, P. Load Balancing based on Deep Packet Inspection, 2013. US Patent 8,606,921, Google Patents.

[112] Wang, K., Cretu, G., and Stolfo, S. J. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID '05)* (2005), pp. 227–246.

[113] Wang, K., Parekh, J. J., and Stolfo, S. J. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID '06)* (2006), pp. 226–248.

[114] Wang, K., and Stolfo, S. J. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID '04)* (2004), pp. 203–222.

[115] Wang, X., Jiang, J., Tang, Y., Wang, Y., Liu, B., and Wang, X. $StriD^2FA$: Scalable Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 11th IEEE International Conference on Communication (ICC '11)* (2011), pp. 1–5.

[116] Wang, Y., Yun, X., Shafiq, M. Z., Wang, L., Liu, A. X., Zhang, Z., Yao, D., Zhang, Y., and Guo, L. A Semantics Aware Approach to Automated Reverse Engineering Unknown Protocols. In *Proceedings of the* 20th *IEEE International Conference on Network Protocols (ICNP '12)* (2012), pp. 1–10.

[117] Wang, Y., Yun, X., and Zhang, Y. Rethinking Robust and Accurate Application Protocol Identification: A Nonparametric Approach. In *Proceedings of the* 23rd *IEEE International Conference on Network Protocols (ICNP'15)* (2015), pp. 134–144.

[118] Wang, Y., Yun, X., Zhang, Y., Chen, L., and Zang, T. Rethinking Robust and Accurate Application Protocol Identification. *Computer Networks 129* (2017), 64–78.

[119] Wu, Y.-S., Bagchi, S., Singh, N., and Wita, R. Spam Detection in Voice-Over-IP Calls through Semi-Supervised Clustering. In *Proceedings of the* 39th *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN '09)* (2009), pp. 307–316.

[120] Ye, M., Xu, K., Wu, J., and Po, H. AutoSig-Automatically Generating Signatures for Applications. In *Proceedings of the* 9th *IEEE International Conference on Computer and Information Technology (CIT'09)* (2009), pp. 104–109.

[121] Ye, N., Emran, S. M., Li, X., and Chen, Q. Statistical Process Control for Computer Intrusion Detection. In *Proceedings of the* 1st *DARPA Information Survivability Conference & Exposition (DISCEX '01)* (2001), pp. 3–14.

[122] Yeganeh, S. H., Eftekhar, M., Ganjali, Y., Keralapura, R., and Nucci, A. CUTE: Traffic Classification Using TErms. In *Proceedings of the* 21st *International Conference on Computer Communications and Networks (ICCCN '12)* (2012), pp. 1–9.

[123] Yuan, Z., Xu, J., Xue, Y., and Van Der Schaar, M. Bits Learning: User-Adjustable Privacy Versus Accuracy in Internet Traffic Classification. *IEEE Communications Letters 20* (2016), 704–707.

[124] Yuan, Z., Xue, Y., and van der Schaar, M. BitMiner: Bits Mining in Internet Traffic Classification. In *Proceedings of the* 34th *ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)* (2015), pp. 93–94.

[125] Yun, X., Wang, Y., Zhang, Y., and Zhou, Y. A Semantics-Aware Approach to the Automated Network Protocol Identification. *IEEE/ACM Transactions on Networking 24* (2016), 583–595.

[126] Zhang, R., Wang, X., Yang, X., and Jiang, X. Billing Attacks on SIP-based VoIP Systems. In *Proceedings of the* 1st *USENIX Workshop on Offensive Technologies (WOOT '07)* (2007), pp. 1–8.

[127] Zhou, Z. H., and Li, M. Tri-training: Exploiting Unlabeled Data using Three Classifiers. *IEEE Transactions on Knowledge and Data Engineering 17* (2005), 1529–1541.