AN INVESTIGATION ON DETECTING APPLICATIONS HIDDEN IN SSL STREAMS USING MACHINE LEARNING TECHNIQUES

by

Curtis S. McCarthy

Submitted in partial fulfillment of the
requirements for the degree of
Master of Electronic Commerce

at

Dalhousie University
Halifax, Nova Scotia
August 2010

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "AN INVESTIGATION ON DETECTING APPLICATIONS HIDDEN IN SSL STREAMS USING MACHINE LEARNING TECHNIQUES" by Curtis S. McCarthy in partial fulfillment of the requirements for the degree of Master of Electronic Commerce.

Dated: August 13, 2010

Supervisor:

Dr. Nur Zincir-Heywood

Readers:

Dr. Vlado Keselj

Dr. Evangelos Milios

DALHOUSIE UNIVERSITY

DATE: August 13, 2010

AUTHOR: Curtis S. McCarthy

TITLE: AN INVESTIGATION ON DETECTING APPLICATIONS HIDDEN IN SSL STREAMS USING MACHINE LEARNING TECHNIQUES

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: M.E.C. CONVOCATION: October YEAR: 2010

Table of Contents

List of Figures

Abstract

The importance of knowing what type of traffic is flowing through a network is paramount to its success. Traffic shaping, Quality of Service, identifying critical business applications, Intrusion Detection Systems, as well as network administration activities all require the base knowledge of what traffic is flowing over a network before any further steps can be taken. With SSL traffic on the rise due to applications securing or concealing their traffic, the ability to determine what applications are running within a network is getting more and more difficult. Traditional methods of traffic classification through port numbers or deep packet inspection have been deemed inadequate by researchers thus making way for new methods. The purpose of this thesis is to investigate if a machine learning approach can be used with flow features to identify SSL in a given network trace. To this end, different machine learning methods are investigated without the use of port numbers, Internet Protocol addresses, or payload information. Various machine learning models are investigated including AdaBoost, Naive Bayes, RIPPER, and C4.5. The robustness of the results are tested against unseen datasets during training. Moreover, the proposed approach is compared to the Wireshark traffic analysis tool. Results show that the proposed approach is very promising in identifying SSL traffic from a given network trace without using port numbers, Internet protocol addresses, or payload information.

Acknowledgements

I would like to thank my supervisor Dr. Nur Zincir-Heywood for her guidance throughout this work. The valuable advice I received from Riyad Alshammari was also of great benefit. The generation of the dataset used in this thesis wouldn't have been possible without the help from Dalhousie's UCIS and TARA. Specifically, Dana Echtner, Jeff Allenwood, and Krista Skodje. A big thank-you goes out to all of the teachers and professors who taught me how to learn and my family for their support.

Finalement, je voudrais remercier ma lapinoune pour toutes les biscuites. J'tm.

Chapter 1

Introduction

Correct classification of network traffic is a fundamental step required for many pivotal services required by various stakeholders including Internet Service Providers (ISPs), governments, and system administrators. These services include traffic shaping, ensuring the uptime of networked mission-critical applications, workload modeling, managing bandwidth budgets, detecting bottlenecks, and balancing Quality of Service (QoS) [4, 3, 2]. To this end, the classification of network traffic, is imperative for making the necessary calculations in order to resolve any of the above duties.

Successful methods pursued in the past have relied on deep packet inspection by examining the contents of the payload or using port numbers to correctly identify the application behind a traffic stream. Unfortunately they no longer hold as much weight as they once did due to encryption rendering packet payloads non-transparent and dynamic port allocation enabling applications to connect on alternate ports then those assigned by the Internet Assigned Numbers Association (IANA). For example, applications attempting to mask their traffic in order to circumvent firewalls may use the same ports as other white-listed services. Alternatively, applications may conceal packets by masquerading as a different protocol through tunneling or even employ the use of cryptographic protocols to protect their payloads.

Secure Socket Layer (SSL) is a fundamental security protocol belonging to the application layer in the Internet Transmission Control Protocol Internet protocol (TCP/IP) model but residing below higher level application protocols and above TCP. It enables e-commerce transactions and other applications to communicate securely over a public network by encrypting the packet payload. Despite its good intentions SSL also creates a black hole of encrypted network traffic, which may be used for illegitimate or non-network sanctioned purposes. Generally speaking, the disadvantages brought about by encrypting traffic create a security trade-off between a secure protocol and losing knowledge about a network [53].

Recent research has shown a sharp rise in the use of the SSL protocol amongst applications to encrypt network traffic [7]. This trend shows no signs of slowing down as application developers race to encrypt or mask their traffic as a different protocol while the demand of knowing all types of traffic becomes of greater interest to ISPs, governments, and network engineers. With the rise of applications utilizing SSL, the need for a different manner of traffic classification is paramount to resolving many network questions. To this date, there has been no reported classification measurement using a flow-based analysis on encrypted SSL traffic.

Past investigations into traffic classification have shown promising results primarily in the classification of unencrypted traffic with more recent explorations into encrypted traffic. These methods rely on the statistical patterns left behind by the packet attributes or flows to determine which application is within a given stream. Packet-based approaches by Bernaille et al. [7] have shown promising results by utilizing the first few packets for classification. Other methods using Hidden Markov Models (HMMs) by Wright et al. [52] have also shown similar results by taking an alternative flow-based approach using clustering. Machine Learning (ML) techniques using AdaBoost, C4.5, Naive Bayes, RIPPER algorithms by Alshammari et al. [4] have also produced promising results in the area of encrypted traffic classification. Nevertheless, few have incorporated the use of SSL in their training and those that have conglomerated the traffic together treating it as a single label with no regard to the underlying application [19, 18, 9].

The objective of this thesis is to investigate a statistical flow-based approach to classifying applications implementing or being tunneled through the SSL protocol using supervised ML techniques. The ML algorithms chosen by Alshammari et al. [4] were chosen primarily due to their human readability post classification. This readability is important for interpreting how a ML algorithm arrived at its results and for highlighting which feature set attributes were actually used. Thus, a similar approach is followed in this thesis. Since no prior dataset focused on SSL, a new one was automatically generated in the lab. The use of tunneling protocols was implemented to further obfuscate the traffic by introducing a degree of entropy into the dataset. Numerous precautions were taken to prevent a biased traffic set which does not represent real traffic.

In the rest of this thesis, Chapter 2 presents a literature review with more in-depth look at previous work. Chapter 3 describes the automated data collection process followed to generate the dataset used. Chapter 4 then breaks down the methodology steps taken to prepare and train the proposed system. An analysis of the results is offered in Chapter 5. Chapter 6 presents a baseline comparison of the proposed system using the well known Wireshark traffic analysis tool. Finally, conclusions are given and future work is discussed in Chapter 7.

Chapter 2

Literature review

The intent of this chapter is to present the reader with an overview of SSL as well as past research related to this thesis. Initially, a description and short history on the evolution of SSL is offered. Various methods of traffic classification are then discussed along with their advantages and disadvantages.

2.1 Secure Socket Layer — SSL Protocol

This section discusses the history and usage of SSL around the world today for security and e-commerce. An explanation on the SSL encryption process is then brought to light. The evolution of SSL to Transport Layer Security (TLS) is then described with emphasis on the enhancements and differences compared to its predecessor.

2.1.1 Overview

The concept behind SSL is to provide secure communication over a public network through the use of multiple algorithms for cryptography, digests, and signatures. By supporting this type of dynamic authentication, SSL servers are able to adapt to any legal obligations surrounding the use of cryptography by choosing which algorithms to use during the handshake. To accommodate every possible application, SSL was designed to be application independent, laying between the transport layer (specifically over TCP) and application layer of the TCP/IP protocol stack. It was originally designed by Netscape to secure e-commerce transaction over the HyperText Transfer Protocol (HTTP). However, the more prominent HyperText Transfer Protocol Secure (HTTPS) protocol is not the only application that can run over SSL. As stated by Bernaille et al. [7], other application protocols are realizing the need to encrypt and conceal their data from packet sniffers and are implementing Application Programming Interfaces (APIs) to use back-end SSL libraries. For those application instances that are not able to support SSL communication, unencrypted TCP traffic

can also be wrapped by programs such as Stunnel [48] to encrypt other applications such as e-mail. Regrettably, nefarious users may try to conceal their attacks within this protocol creating the illusion that a high amount of legitimate SSL traffic is occurring. Thus, a security trade-off exists between ensuring confidential e-commerce transactions via HTTPS and losing network flow knowledge [53].

As discussed earlier, the requirement for SSL stemmed from the need for a secure communication medium for web traffic. Since web server communication occurs in clear-text via HTTP, transmitting confidential information in this manner over an insecure network was not a viable option. Thus, in 1994 Netscape developed SSLv2 to protect the consumer during e-commerce transactions [49]. Unfortunately, the original implementation suffered from numerous security holes and it was later superseded by SSLv3 in 1995 due to security issues [49]. The upgrade permitted the use of a wider variety of encryption algorithms and certificate authentication [49]. To help the adoption of this new protocol, SSLv3 was designed to be backwards compatible with SSLv2 [32]. Thus, during the handshake process the server is able to fall back to the methods present in SSLv2 [32].

The SSL encryption process begins with a handshake exchange when a client requests an SSL-enabled service from a server. With the client's initial request comes a series of supported authentication methods called CipherSuites [33]. Upon receipt, the server calculates the strongest possible encryption methods based on the CipherSuite and sends its decision as well as a digital certificate back to the client. This certificate includes a public key as well as other pertinent information regarding the server [49]. Since the public key is freely available, additional precautions are taken to prevent tampering. As such, a trusted third party Certificate Authority (CA) is used to verify the legitimacy of the server. The CA maintains a collection of valid certificates and uses its private key to sign server certificates [49]. A client can then use the public key of the CA to verify the authenticity of the server. The next step involves generating session keys to create a secure stream between the client and the server. A random number is first generated by the client, encrypted using the server's public key, and sent to the server. While an attacker could easily capture this data, it would have to know the private key of the server to successfully decrypt the random number [49]. The client and the server then use the random number to encrypt and decrypt the

remaining connection. Figure 2.1 illustrates the overall process.

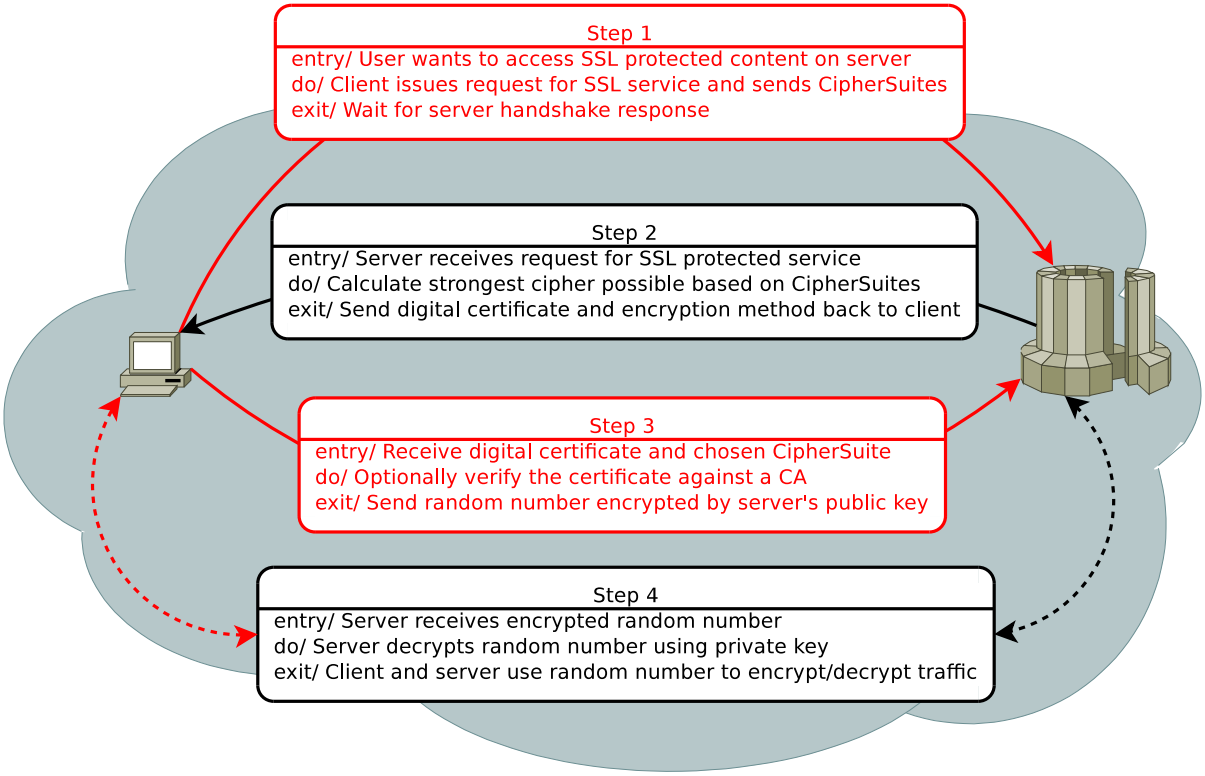

Figure 2.1: Overview SSL process

2.1.2 Transport Layer Security — TLS Protocol

In an attempt to standardize and have an open community supported standard, TLSv1 was created by the Internet Engineering Task Force (IETF) in 1999 [32]. It is important to note that TLS and SSL are not interchangeable and one protocol must be decided upon before the handshake is completed [32]. As such, it is widely accepted that SSLv3 has evolved into TLSv1 despite several minor differences in the following list [1]:

- The replacement of the Message Authentication Code (MAC) algorithm by the keyed-Hashing for Message Authentication Code (HMAC) algorithm for more secure hashes is implemented.

[1] http://technet.microsoft.com/en-us/library/cc784450(WS.10).aspx

- Additional alert messages are added for easier diagnostics.
- The usage of intermediary CAs is permitted instead of requiring confirmation from the root CA.
- The usage of padding block values for block cipher algorithms is added for TLS.
- Due to the closed source nature of the Fortezza algorithms, they are not included in the TLS RFC [32] because it contradicts IETF policy.
- A few minor changes are implemented in some of the message fields.

Additionally, TLS now supports the User Datagram Protocol (UDP) as the Datagram Transport Layer Security (DTLS) protocol [21]. At the time of writing, OpenSSL is currently compiled with TLS by default and is the most popular implementation of the SSL/TLS protocols [49]. The IETF is currently working on TLSv1.2 to remove the protocol's dependence on MD5 and SHA-1 digest algorithms, which have been compromised in recent research [10]. Finally, new authenticated encryption modes using Counter Mode Encryption (CTR) and combined encryption/authentication modes are planned [10].

2.2 Previous work

To this date, there is very little literature available focusing on encrypted SSL traffic classification. Most research on SSL analysis has concentrated either on building IDSs that rely on behavior-based anomaly detection [53] or developing algorithmic optimizations to reduce computational cost preventing DoS attacks [39, 45, 42, 56]. Yet few researchers have investigated the classification of SSL traffic across different application instances. Most traffic classification methods focus on unencrypted data, despite the rise in applications securing or concealing their data with SSL [7, 45]. For those works, which include encrypted traffic, whether it be SSL or another form of encryption, many simply state that their methods are not affected by the encryption process as they rely on packet or flow attributes [36, 55, 19, 9]. Nevertheless, few researchers actually attempt to parse those encrypted streams to determine the concealed application instances [46]. Thus, in many network classification papers SSL is

simply conglomerated together as a single application instance representing a small portion of the overall dataset [19, 18, 9]. This aggregate class approach does little to present Internet Service Providers and network administrators with the ability to determine the exact application hidden in a traffic flow [19, 38].

The level of classification granularity becomes further obfuscated by the use of tunneling programs in an attempt to encrypt or conceal network traffic. Tunneling is a frequently mentioned term amongst researchers [36], however it is simply referred to in the related work sections or used to point out potential drawbacks in methodology of others. With the exception of Tunnel Hunter [16, 17, 12], very few seem to have actually attempted to classify tunneled traffic, much less encrypted tunneled traffic. Research done by Dusi et al. [16, 17] has shown promising results, but it focuses solely on HTTP [12] and Secure Shell (SSH) [16, 17] tunnel detection. The authors claim although Domain Name Server (DNS) tunnels also run in the application layer and are generally allowed by most firewalls, they are ignored in Dusi et al.'s work due to bandwidth constraints and lower popularity of usage [16]. Even then, in order for Tunnel Hunter to work on encrypted traffic, network administrators must enforce only one type of user authentication [16]. Indeed, such a requirement will not work in practice.

Williams et al. [51] used similar classification methods with flow-based approaches utilizing NetMate [15] preprocessing and ML algorithms but simply do not account for SSL traffic. Other statistical fingerprinting techniques by Crotti et al. [13] follow a similar route using clustering but concentrate on a smaller subset of application instances without any encryption.

Recent research studying encrypted flow-based streams has clearly demonstrated that machine learning algorithms are able to classify encrypted network traffic with a greater degree of accuracy then expert driven systems [1, 4, 3, 2]. The purpose of those works was to determine the possibilities of traffic signatures amongst encrypted SSH traffic. SSL and SSH share many similarities but also differ greatly due to the following key points listed from Barrett et al. [5].

- Server-side authentication for SSL is optional whereas it is a requirement for SSH.
- TLS requires Public Key Infrastructure (PKI) using certificates whereas SSH is

limited to keys.

- SSH supports additional features compared to SSL via its connection protocol allowing tunneling and other services to function.

Bernaille et al. [7] classified SSL using the first few clear-text packets to determine the encryption algorithm used and application type. They achieve an 85% degree of accuracy but claim their application is not scalable due to SSL compression features, which would cause their detection system to fail.

The use of HMM by Dainotti et al. [14] and Wright et al. [52] produced good results demonstrating the ability of packet-level statistical analysis. Wright et al. [52] made use of encrypted SSL connections in their dataset prompting further investigation. In order for their system to work, additional preprocessing steps were taken by the authors to improve the recognition ability of their classifier using vector quantization and clustering. Only three post-encryption features from packets are used to build these models: size, timing, and direction. A comparison of the results obtained using the same methodology detailed in this paper is presented in Chapter 6.

Regardless of the methods used, many researchers agree that traditional port numbers used by systems such as Wireshark [11] are no longer a viable manner to detect traffic due to dynamic port allocation [36, 18, 55, 19]. Furthermore, the use of signature matching in deep packet inspection [37], such as Bro [41] or SNORT [43] to identify distinct applications presents several disadvantages. A database of all application signatures must be maintained and continually updated to match new versions, which may change the signature of an application [55, 18]. Additionally, user privacy concerns as well as its inability to be applied against encrypted traffic also make it an undesirable method [55, 18]. As such, many researchers have turned towards ML algorithms, which are built on the pretense of pattern recognition using some attributes of the network traffic [55, 51]. Their usage becomes especially important when considering how large the captured IP packet traces can grow as well as the fluctuations amongst the streams [38]. Williams et al. [50] demonstrated the performance of several ML algorithms including various implementations of Naive Bayes and C4.5 on network traffic in general without considering encrypted traffic specifically. They were evaluated in terms of the accuracy, training times, and computational speed to achieve their results. On the other hand, Alshammari et al. [4]

specifically focused on the classification of encrypted traffic. However, in those works the focus was on SSH and Skype only. Human interpretation of the results was made easy thanks to the ML algorithms used: C4.5, AdaBoost, and RIPPER.

Moreover, from Ruixi et al. [55] and Este et al. [19], it can be seen that applications carry distinct fingerprints in network traffic. What differs amongst the methods of most researchers is which packet or flow feature-set posses enough information to correctly classify an application or grouping. Many researchers agree that packet sizing and inter-arrival times tend to be amongst the most popular attributes offering the most information [52, 38, 13]. Williams et al. actually proved that greater classification accuracy can be obtained through reduced feature sets [50, 38], which was also confirmed in the encrypted traffic classification work of Alshammari et al [3, 2]. Additionally, the ML algorithms employed offer different approaches to classifying the traffic based on how they assign weights to the different flow attributes [38]. The other important characteristic to consider are the preprocessing steps as well as the datasets used as due to their large influence on the final results [38].

Aside from focusing on encrypted SSL traffic, this thesis also differs from previous literature on the datasets employed. The usage of public data sets for training can be misleading as they are only as accurate as the labeler employed [46]. Others have failed to properly represent their datasets by excluding parts of traffic, such as UDP [36], or using simulation techniques, which do not account for "real" background network traffic. Others have fixed the algorithms used to only account for one type of handshake encryption [52].

Nguyen et al. [38] highlighted the importance of balancing training sets by pointing out the differing results found by two researchers, Park et al. [40] and Erman et al. [54], using the same ML algorithms and datasets. Unfortunately, the resulting network traffic would fail to adequately represent real world traffic. High degrees of accuracy can be attributed to the high sensitivity of training for the ML algorithms since the training sets and data preprocessing methods greatly influence the resulting output. Nevertheless, these models would fail to scale to the real network scenarios where certain assumptions or restrictions are not a viable option. In order to demonstrate the robustness of the results in this thesis, additional tests are performed on different datasets to investigate the adaptability of the proposed approach.

Chapter 3

Data collection

The purpose of this chapter is to present an overview of the data collection process followed to generate the required traffic. The reasoning behind generating a new dataset can be attributed to the lack of publicly available labeled datasets. Also, requiring the absolute *ground truth* is imperative for the ML algorithms to accurately identify key classification features. Finally, many existing datasets have very little SSL traffic with the majority belonging to HTTPS. Thus, a training data set is generated to overcome these problems. In the following description of the network layout, housing where the dataset is generated is first described. Following this is an in-depth look at how the traffic was captured at multiple sources. The process behind centralizing all of the captured traffic into a repository is then brought to light. An explanation regarding the overall entropy of the captured traffic is then discussed. Finally, all of the applications and tunnels used in this work are described in further detail.

3.1 Network Setup

To prevent a single network's hardware from influencing the packet attributes, two networks were used to gather the data: one which operated under Dalhousie University[1] and the other independently under TARA[2]. The Dalhousie network contained one computer, *nims*, whereas the TARA network housed both *taraserver* and *taraclient* computers. Since there was only one physical computer at Dalhousie, *nims* hosted two virtual machines using VMware [3] - *nims-server* and *nims-client*. All machines ran the open source OS Ubuntu [4] Linux distribution, but each site differed in versions varying from 9.04 '*Jaunty Jackalope*' to 9.10 '*Karmic Koala*'. At each

[1] http://www.dal.ca
[2] http://web.archive.org/web/20080531140809/http://www.tara.ca/
[3] http://www.vmware.com/
[4] www.ubuntu.com

network, the machines were segregated on a separate subnet with static IP addresses to prevent contamination and respect network privacy policies [5]. Since some of the services required hostnames, dynamic DNS through Dynamic Network Services Inc. [6] was used for all of the TARA services with the exception of mail. The machines on the Dalhousie network were supplied with their own hostnames through Dalhousie's DNS servers.

Dalhousie's firewall rules allowed for any traffic on any port from TARA to flow through, while blocking the rest of the outside world except for SSH for remote administration. Additionally, an exception was required for *nims-server* to communicate with Dalhousie's mail servers. An MX record, i.e. a mail server record, was setup at Dalhousie to forward mail from their mail servers to *nims-server*. Dalhousie University's Information Technology Services (ITS) took the added precaution of white-listing email addresses flowing between the two networks. The TARA firewall setup mirrored that of Dalhousie's ensuring all traffic flowing between the two networks was locally generated. While remote administration was essential to ensuring a high degree of uptime, it unfortunately also left an open invitation to external probing. More information on how this was further secured can be found in section 3.4.1.

Originally, The Onion Router (TOR)[7] network setup was considered to help introduce additional randomness into packet arrival times and inject further entropy into the dataset. TOR allows users to re-direct their traffic through a network of nodes before reaching the final destination thus obscuring their traffic from network administrators. Despite the bandwidth overhead, the open source TOR program does an excellent job of anonymizing a client's requests as well as the responses back from the actual server. However it was deemed infeasible after encountering configuration difficulties with a majority of the services and incessant attacks from various countries. Furthermore, one of the tunneling services required root access to run, opening up a potential exploitable hole. An overview of above network setup plus additional TCPdump captures discussed in the section 3.2 is visualized in Figure 3.1.

[5]Dalhousie University Computing and Information Services policy on capturing network traffic http://its.dal.ca/policies/5.5.2-data-sets.pdf

[6]http://www.dyndns.com/

[7]http://www.torproject.org/

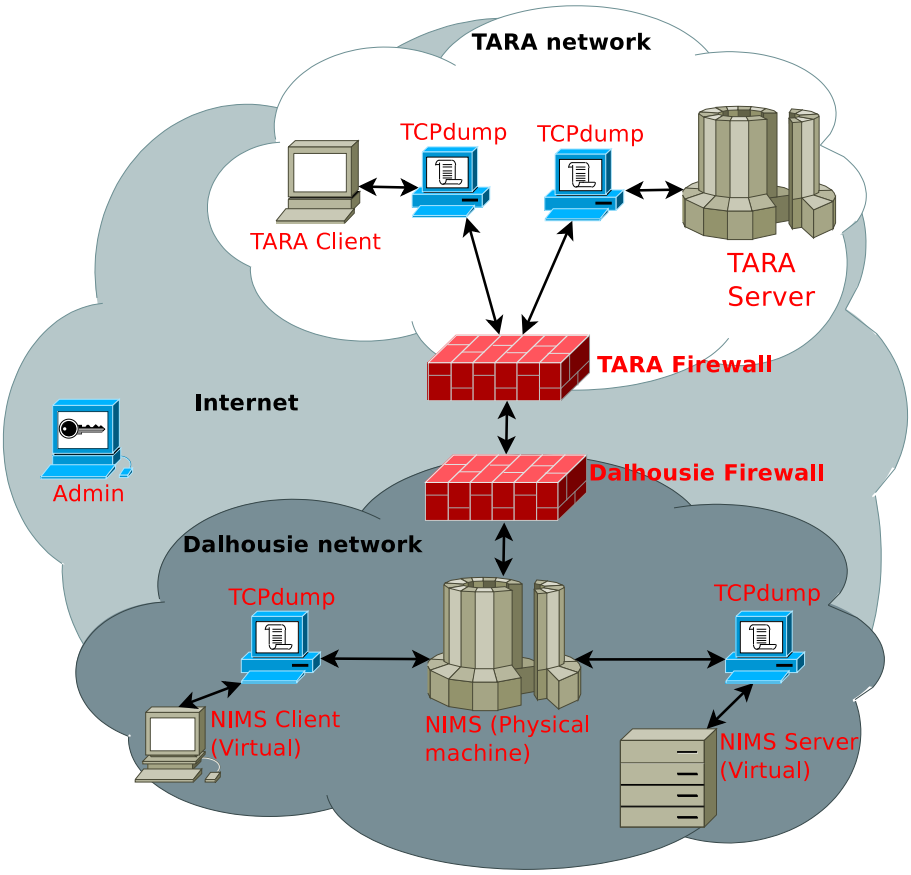

Figure 3.1: Overview of network and TCPdump captures

3.2 Capturing traffic

As shown in figure 3.1, packet captures were acquired using the TShark [8] TCPdump [9] program in libpcap binary format. With the exception of *nims*, each machine captured its own traffic to avoid any possibility of packet loss between the networks. The two virtual machines at Dalhousie performed exactly like separate physical boxes from a network point of view preventing any interference from occurring. All client and server machines ran a script initializing TShark at boot time to start recording TCPdump files. The command line options passed to TShark specified several important details. Firstly, traffic was captured on any interface with no network name resolution to ensure full coverage and reduce TShark's memory footprint respectively. In order to help automate the capture, the multiple file 'ring mode' was enabled. This mode of

[8] http://www.wireshark.org/docs/man-pages/tshark.html
[9] http://www.tcpdump.org/

operation continually captured packets until the TCPdump file reached a maximum size of 350MB. At this point, TShark would open up another file and begin writing to it, incrementing the filename counter by one. Traffic from the *nims* IP address was also explicitly ignored for backup purposes described in section 3.2.1. Finally, the file path location for storage of all of these files was specified.

3.2.1 Centralizing the Traffic Captures

To avoid using up the limited resources available on each client and server machine, an alternative mode of storage was required. As a fully automated solution, *nims* was chosen to act as a central server repository housing all of the completed TCPdump traffic files. This was accomplished in two phases: ensuring the integrity of the TCPdump files and subsequently transferring them to the repository. The first phase used a cron job script on each client and server machine to verify the file size of the TCPdump file. These files were then moved to a special folder location every 30 minutes. This verification step was paramount to ensuring the TCPdump file was available to be moved and not currently being written to or incomplete due to a power failure. The second phase executed a cron job script every 12 hours on the physical *nims* machine. This script secure copied (SCP) into each client and server machine and transferred the TCPdump files to their respective directories in *nims*, thus keeping all traffic segregated. Once transferred, the completed TCPdump files were then removed remotely. To avoid contamination of the dataset and recursive loops, *nims* IP address was specifically excluded in the TShark startup command. This allowed the transfer of these 350MB TCPdump files to occur without being captured by the local TShark processes. An overview of the backup and centralization steps is illustrated in Figure 3.2.

3.3 Traffic heterogeneity

The applications selected for the data set were based on their ability to be reproduced by being mainly open source, the heterogeneity of protocol traffic they provided, their popularity, and their inclusion in the literature. TCP, UDP, and DNS traffic were all important to provide a good mix of protocols and packet data. The reasoning behind all of this heterogeneity stems from the need for realistic background traffic to give the

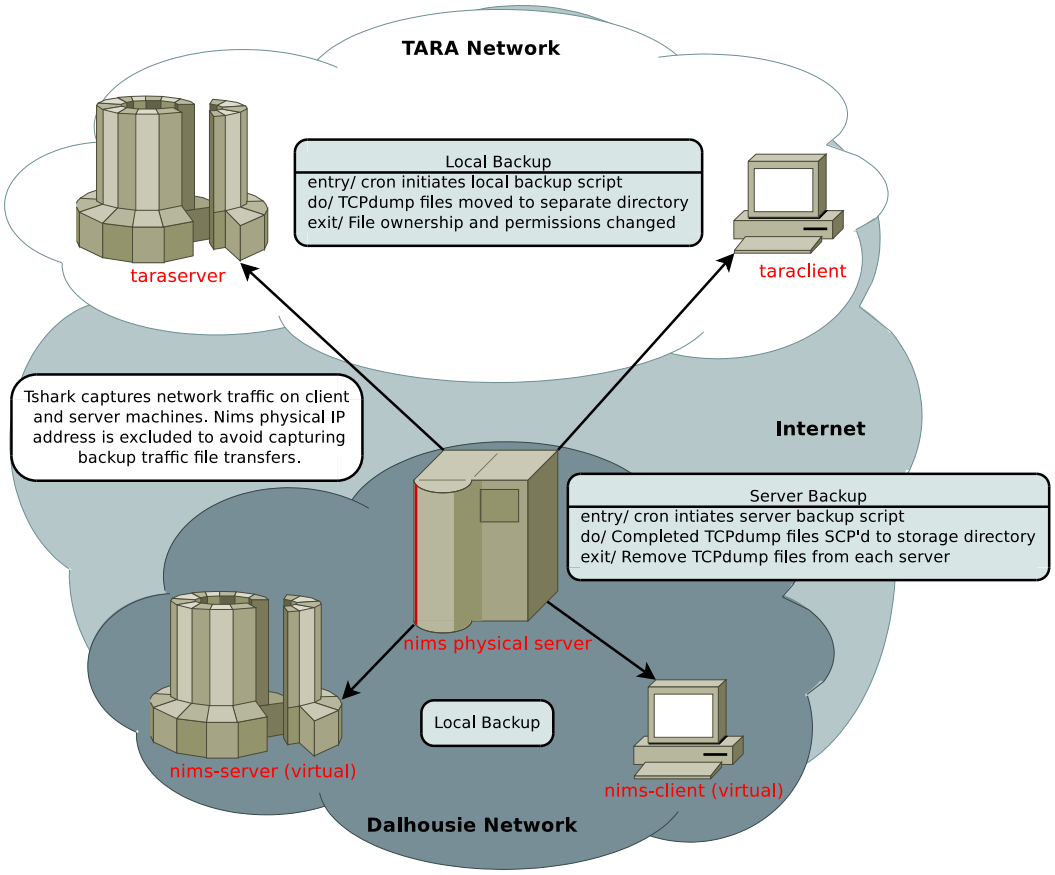

Figure 3.2: Overview of the automated backup and centralization process

illusion of real network traffic [1, 4]. Without this background noise to inject entropy into the dataset, the classifiers would be too restrictive and not scalable to different networks [1, 4]. The usage of a master port list for labeling purposes was important to determine the *ground truth* of which ports were being utilized for application specific traffic with everything else being labeled as *spurious* background traffic.

3.4 Application Instances and Tunnels

The applications used to populate the dataset as well as their corresponding protocols, initialization scripts, and required parameters are discussed in this section. Any modifications to application source code and configuration files are also brought to light. All services were set to startup on boot to prevent power-failures from creating zombie processes on the clients. With the exception of Skype, all of the application programs are both open source and freely available through use of non-commercial

licenses, thus promoting the reproducibility of the dataset. Traffic for all of the application instances was gathered over a period of 4 months with intermittent breaks for maintenance.

3.4.1 OpenSSH

SSH [30] enables secure communication, including data transfer and command execution, to a remote host using public key cryptography. It is a valuable tool for system administrators or anyone wanting to securely connect and even forward GUI programs back to their client computers.

OpenSSH version 5.1p1 [10] was installed through Ubuntu's Synaptic package manager [11]. Through use of the *sshd_config* configuration file, the OpenSSH server was locked down to only allow remote connections from a special administrative account. Root login was also prohibited from being authenticated. Additional filtering was applied forcing client logins to only be authenticated if the connection came from a Dalhousie or TARA IP address. Finally, in an effort to simulate user-interactive login, SSHPASS [12] was used for to better protect user passwords and avoid setting up public/private keys.

The SSH script was wrapped within SSHPASS to perform the initial authentication. Afterwards, commands were executed remotely, and files were transferred between the client and server. Approximately 852748 SSH flows were generated.

3.4.2 Web

HTTP [23] and HTTPS [24] web traffic was captured through use of a Linux Apache MySQL and PHP (LAMP) server setup. All web services were installed using Ubuntu's Synaptic package manager with the latest versions in the repository as follows: Apache v2.0 [13], MySQL v5.0.75 [14], and PHP v5.2.6 [15]. Additionally, a self-signed SSL certificate key was generated at both sites to accept HTTPS connections. Both HTTP

[10] http://openssh.org
[11] https://help.ubuntu.com/community/SynapticHowto
[12] http://sourceforge.net/projects/sshpass/
[13] http://httpd.apache.org/
[14] http://www.mysql.com/
[15] http://www.php.net/

and HTTPS websites possessed the same web pages consisting of HyperText Markup Language (HTML) and Hypertext Preprocessor (PHP) files.

Client side scripts made use of the wget [16] open source program to crawl over both sites following links to all the pages. To enable automated client web requests, a flag to force automatic certificate acceptance was used. Additionally, certain pages would throw exceptions, such as *404 page not found* errors, upon request from the recursive crawling client to further emulate a user browsing. Both text and binary data files were transferred between client and server. Approximately 10092691 HTTP and 8562626 HTTPS flows were generated.

3.4.3 Mail

Postfix v2.5.5 [17] and Dovecot v1.1.11 [18] were used to send and receive both encrypted and unencrypted email traffic. The Dalhousie site was responsible for unencrypted traffic whereas the TARA site was solely encrypted traffic. User account credentials were created and authenticated using the Simple Authentication and Security Layer (SASL) framework [28]. The network setup at Dalhousie forced mail to go through several hops before arriving at the Faculty of Computer Science's servers which then relayed the emails to *nims-server*. On the other hand, *taraserver* had a slightly different MX record setup and was capable of dealing with emails directly. Additionally, tunneling was impossible from the Dalhousie site due to external firewall rules, so all tunneling was done at the TARA site. In order to avoid email duplication and reduce the storage load on the servers, Post Office Protocol 3 (POP3) [26] was chosen over Internet Message Access Protocol (IMAP) [25]. By doing so, it restricted emails to the client program once downloaded. OpenSSL v0.9.8g [19] was installed to support Simple Mail Transport Protocol Secure (SMTPS) [29] and Post Office Protocol Secure (POP3S) [27] authentication with both SSL and TLS protocols.

On the client side, emails were sent with and without attachments using the mailsend program [20]. Mailsend allowed emails to be sent via the command-line with

[16] http://www.gnu.org/software/wget/
[17] http://www.postfix.org/
[18] http://www.dovecot.org/
[19] http://www.openssl.org/
[20] http://www.muquit.com/muquit/software/mailsend/mailsend.html

arguments specifying the sender, recipient, body, mail server, authentication credentials, and optional attachments. Custom scripts further complimented its capabilities by allowing optional tunneling ports to be specified.

The Thunderbird v2.0.0.24 mail program by Mozilla [21] was used as the incoming mail client. Due to the self-signed certificate nature of the mail services, the Remember Mismatched Domains v1.4.6 add-on [22] was used to automatically accept certificates. Different accounts, protocols, and tunnels were then specified to receive emails at regular intervals. Thunderbird was set to automatically download emails, so its cron job simply launched the program at specified times. To give an impression of the volume of mail traffic, the 260000 inbox limit of emails imposed by Thunderbird had to be cleared multiple times. Approximately 29818 POP3, 1479638 SMTP, 8786 POP3S, and 420502 SMTPS flows were generated.

3.4.4 VSFTP

File Transfer Protocol (FTP) [22] is a commonly used protocol for transferring files allowing clients to connect to servers and issue series of commands to either place or retrieve files. FTP traffic was generated through use of the Very Secure FTP Daemon (VSFTP) v2.2.0 server [23]. Command-line scripts using both encrypted and unencrypted protocols would retrieve and submit files. Both binary and text files were transferred with consistently changing sizes. Various configuration options were taken to restrict FTP users in a sandbox environment. This would help to reduce the spread of contamination should a single account become compromised. Anonymous FTP traffic was disabled to prevent any guest logins. The SSL certificate used was the one provided by Ubuntu during installation. Different ports for both data and control channels at each site were used to separate encrypted and unencrypted traffic. Approximately 130826 FTP and 151580 FTPS flows were generated.

[21] http://www.mozilla.org/projects/thunderbird/
[22] https://addons.mozilla.org/en-US/firefox/addon/2131/
[23] http://vsftpd.beasts.org/

3.4.5 Skype

Skype is a chat client, which allows both text, voice, and video to pass through its proprietary encryption algorithms for secure communication to any telecommunications device in the world. Representing the only closed-source application, Skype v2.1.0.47 (Beta) [24] was installed through the binary Debian packages provided by the Skype company. Configuration was limited to selecting which port to use for traffic. No automation was possible due to its closed-source nature so human-controlled conversations were executed at both sites. Specifically, the Skype accounts setup on each client chatted with each other with a physical user directing the conversation. Approximately 3489 Skype flows were generated.

3.4.6 Subversion

Subversion [25] is a version management system used by many software projects enabling multiple users to work on the same file without accidentally overwriting each others changes. It uses a central repository to house various versions of files supporting additional functionality such as branching, reverting to past files, and comparing file versions. Subversion v1.6.5 [26] was installed and a repository was created on the servers holding a variety of files. Command-line client-side scripts listed, checked-out, modified, and committed these files using subversion SVN commands to the repository. Since subversion's protocol is unencrypted clear-text, it relies on an SSH tunnel to provide secure communication. As such, secure subversion traffic was simply labeled as another type of SSH behavior. Approximately 40897 subversion flows were generated.

3.4.7 Vuze and Bit-torrent

Bit-torrent allows files to be transferred between peers without having to consume the bandwidth of a central server [8]. To simulate background traffic, as well as introduce wave of UDP traffic, bit-torrent files were downloaded. The original command-line

[24] http://www.skype.com/intl/en-us/home
[25] http://subversion.apache.org/
[26] http://subversion.apache.org/

bitTorrent client v3.4.2-11.1 in Ubuntu's Synaptic repositories was used for unencrypted bit-torrent traffic. Vuze v4.3 [27], the evolved form of Azureus [28], was used for encrypted SSL traffic bit-torrent downloads. The former program simply required a torrent file as a command-line argument — no other alterations were necessary. The configuration options in Vuze allowed for numerous options such as the ability to resume downloads automatically and forcing encryption to be required. The files selected for download were varying distributions of Ubuntu. Approximately 2332976 Vuze and 1422414 bit-torrent flows were generated.

3.4.8 Jabberd and Pidgin CHAT

Peer-to-Peer (P2P) chat was automatically generated using the Jabberd v1.4.3 server [29] and Pidgin v2.6.2 [30] instant messaging client - the default client for Ubuntu. Jabberd uses the open IETF supported Extensible Messaging and Presence Protocol (XMPP) [34] to allow P2P messaging as well as relaying client statuses. In contrast to other chat servers, Jabberd is entirely open source and cross-platform. It also supports SSL encryption for secure communication between two or more parties.

Unfortunately, the Ubuntu 9.10 *Karmic Koala* distribution used a different version of Jabberd, which is a completely different project. As such, the *nims-server* Jabberd service was built from source using the same version setup on *taraserver*. The source code was modified to allow SSL connections to avoid a reported bug [31].

Pidgin's integration with the back-end *lib-purple* library [32] allowed for automatic messaging and status changes to occur between multiple clients via the *purple-remote* package [33]. Once integrated with the proper Xserver display environment variables, cron jobs were able to launch and kill Pidgin processes. Additionally, Pidgin's *buddy pounces* allowed automatic conversations to occur amongst the clients by sending Instant Messages (IMs) based on events.

[27] http://www.vuze.com/
[28] http://azureus.sourceforge.net/
[29] http://jabberd.org/
[30] http://www.pidgin.im/
[31] http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=405461
[32] http://developer.pidgin.im/wiki/WhatIsLibpurple
[33] http://manpages.ubuntu.com/manpages/jaunty/man1/purple-remote.1.html

Encrypted and unencrypted conversations were conducted at both sites respectively. Custom scripts would only be responsible for launching the program, sleeping, then terminating the process after a period of time. Approximately 489282 Jabber and 457024 Jabber-SSL flows were generated.

3.4.9 Telnet-SSL

The telnet network protocol [31] allows unencrypted text-based communication to another port or telnet server. Due to its open nature, telnet has mainly been replaced by other remote access programs for added security. Thus to alleviate security concerns, an alternate implementation that supports SSL to encrypt the authentication or entire session between client and server has been developed. Telnet-ssl and telnetd-ssl versions 0.17.24 were both installed from Ubuntu's Synaptic repositories.

To simulate telnet connections and interactions with the connecting port, Expect [34] was used to create an automatic script. Login credentials and optional tunnel ports were passed as command-line arguments to separate scripts, which used either encrypted or unencrypted connections. Interactions with the responding port were accomplished by executing a series of commands before terminating the connection. Approximately 732541 Telnet and 122 Telnet-SSL flows were generated.

3.5 Tunnels

In order to introduce further entropy into the dataset, tunnels were used to conceal and sometimes encrypt many of the application instances. This section delves into further detail regarding how these tunnels function and what they contributed to the dataset. Firstly, HTTP tunneling is presented using its unique dual program technique. Secondly, Internet Control Message Protocol (ICMP) tunneling is discussed showing its ability to circumvent firewalls. Lastly, SSL tunneling is brought to light demonstrating its encryption ability of non-native SSL application instances. Supported application instances for tunneling included the following:

- FTP
- FTP-SSL

[34] http://www.nist.gov/mel/msid/expect.cfm

- SSH
- HTTP
- HTTPS
- POP3
- POP3S
- SMTP
- SMTPS
- Telnet-SSL
- Telnet

3.5.1 HTTP Tunnel

HTTP tunneling encapsulates the original data packet as an HTTP packet thus concealing, but not encrypting, the original payload. Developed by Lars Brinkhoff, HTTP tunnel [35] is a useful tool for users restricted by network firewalls, which only permits HTTP traffic by enabling them to tunnel other traffic through the firewall. The encapsulated HTTP packet travels between the client and the proxy server where it is accepted on a port being listened to on a separate proxy server process. The proxy server then sheds the HTTP layer and forwards the packet to its final destination. The client program, *htc*, listens locally on a specified port, which users have to explicitly set an application to use. For example, making use of the SSH port option, a command redirecting traffic to a localhost port 2010 would be as follows:

ssh user@localhost -p 2010

The htc tunneling program's command line options permit the specification of a proxy server to redirect the packets onto their final destination. In this situation, an htc instance would have to specify both port and destination addresses, as shown below.

htc -F 2010 proxyServer.com:2020

[35] http://www.nocrew.org/software/httptunnel.html

From the above command, we can see that all traffic going to localhost port 2010 is being redirected to the 2020 proxy server port. The proxy server makes use of its own *hts* process to redirect the packet traffic to the final destination, as shown in the command below.

hts -F destinationServer.com:22 2020

Each tunnel had to be established on startup since HTTP tunnel does not rely on any configuration file. As such, startup scripts on both client and server passed command line arguments containing the allocated port and proxy information to respective instances of the programs. Neither program required elevated root privileges and were run as a regular user.

Table 3.1 shows a breakdown of the number of flows from each HTTP tunneled application instance.

Application Instance	Number of flows
http_tunnel-HTTP	1394386
http_tunnel-HTTPS	810
http_tunnel-POP3	60969
http_tunnel-POP3S	299287
http_tunnel-SMTP	658702
http_tunnel-SMTPS	565264
http_tunnel-SSH	572852
http_tunnel-TELNET	590208
http_tunnel-TELNET-SSL	9341

Table 3.1: Total flow numbers of application instances supporting HTTP tunneling

3.5.2 ICMP Tunnel

Ptunnel [36] takes ordinary TCP packets and transforms them into ICMP (ping) echo request and reply packets, thus tunneling the connection between a server, proxy, and client. By doing so, many network firewalls can be bypassed since the TCP packets are concealed, but not encrypted, as ICMP packets. Ptunnel, developed by Daniel Stodle utilizes a proxy server to de-code the incoming ping request packets on a specific port and relay them onto their final destination [47]. On the way back, the TCP packets are once again transformed into ICMP reply packets and transmitted

[36] http://www.cs.uit.no/~daniels/PingTunnel/

back to the client. It is important to note that Ptunnel requires root privileges to run due to its interaction with raw sockets to send and receive ICMP packets. The proxy-side setup simply required the ICMP daemon to run, however, the client-side needed to know both proxy and final destination information. Similar to the HTTP Tunnel program, it ran strictly on command line arguments with no configuration files. The following command illustrates the command line arguments required for the client to redirect the ICMP traffic.

ptunnel -p proxyServer.com -lp 2030 -da destinationServer.com -dp 2040

Table 3.2 shows a breakdown of the number of flows from each ICMP tunneled application instance.

Application Instance	**Number of flows**
icmp_tunnel-HTTP	772
icmp_tunnel-HTTPS	2868
icmp_tunnel-POP3	4744
icmp_tunnel-POP3S	123
icmp_tunnel-SMTP	111
icmp_tunnel-SMTPS	1346
icmp_tunnel-SSH	989
icmp_tunnel-TELNET	179

Table 3.2: Total flow numbers of application instances supporting ICMP tunneling

3.5.3 SSL Tunnel

For application instances that do not support SSL encryption, Stunnel was used to wrap TCP-only traffic within a secure stream [48]. Unlike the other tunnels discussed thus far, Stunnel actually encrypts the tunneled application packets by making use of the external libraries found in OpenSSL.

Instead of using command line arguments to control the program, Stunnel opted for utilizing a configuration file to specify all of the tunneling instructions. The client Stunnel configuration file was altered to listen for specific services on different ports. These ports matched those on the proxy Stunnel configuration file, which would then forward the connection along to the final destination server. An example entry for *nims-client*, proxy *nims-server*, and final destination *taraserver* using the POP3 is listed below:

Client [POP3] accept = 2016 // Connect to localhost on port 2016 connect = nims-server // Forward connection to nims-server

Proxy Server [POP3] accept = 2016 // Listen on port 2016 for traffic connect = taraserver:110 // Send traffic to final destination server:port

Table 3.3 shows a breakdown of the number of flows from each SSL tunneled application instance.

Application Instance	**Number of flows**
stunnel-FTP	10212
stunnel-FTPS	94234
stunnel-HTTP	2273402
stunnel-HTTPS	3523
stunnel-POP3	24226
stunnel-POP3S	205
stunnel-SMTP	334743
stunnel-SMTPS	532162
stunnel-SSH	225955
stunnel-TELNET	245779
stunnel-TELNET-SSL	304

Table 3.3: Total flow numbers of application instances supporting SSL tunneling

Chapter 4

Methodology

This chapter discusses the steps followed to setup the data generation system, pre-process the data, and apply the ML algorithms using WEKA [20]. The first section deals with an overview of the client and server system setup from the point of view of the software and network resources. Following this is an in-depth look at the data pre-processing and flow labeling methods used. A description of the various class labels and independent multi-class runs is then highlighted. A detailed explanation of how the training sets were sampled is then discussed. The application of WEKA, the ML algorithms used, and the summarization of the resulting output are then brought to light. Figure 4.1 illustrates the overall process from data capture to the ML result output. Each training set size had to meet a performance threshold to judge which had the best results using various metrics from Section 4.5.3.

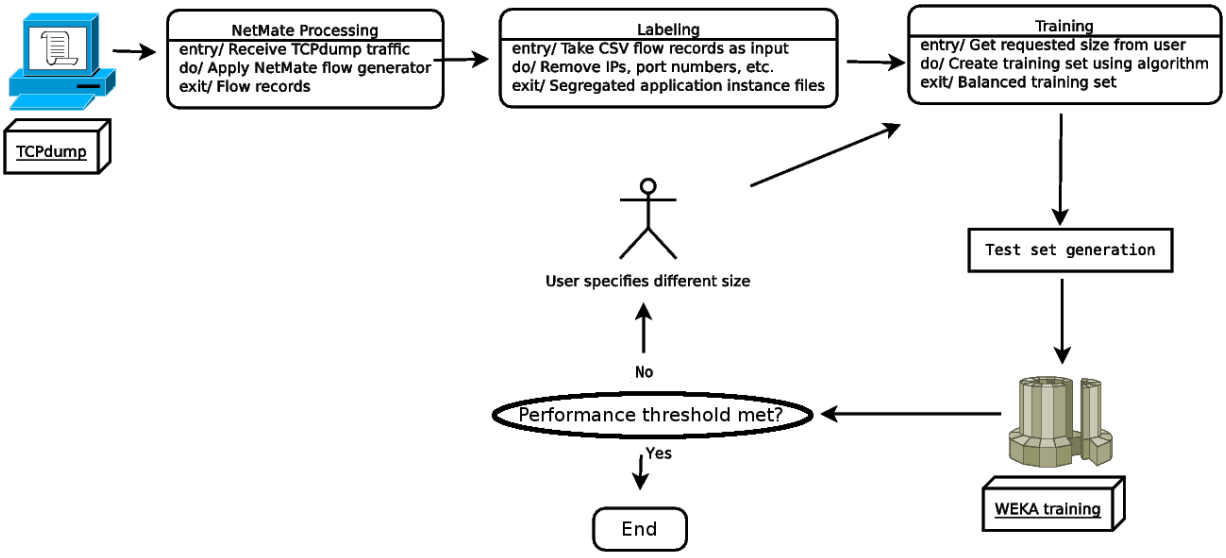

Figure 4.1: Process from the flow generation of captured traffic to the ML algorithm output.

4.1 System Overview

One of the primary goals in generating this dataset was to design the system to be self-sufficient. Only minimal amounts of user interaction were involved, despite wanting to mimic that same traffic interchange between actual users. In order to achieve this level of automation, the use of primarily open source programs, with the exclusion of the proprietary closed-source Skype application, were both preferred and implemented. This section discusses the client and server side setup, which enabled the aforementioned automation to occur.

4.1.1 Client Setup

The client machines, *nims-client* and *taraclient*, were responsible for generating the data through the use of customized shell scripts and cron jobs. The process began with a cron job being set to run at a certain time in order to automatically execute a certain shell script with specific command-line arguments. These shell scripts spawned processes for each of the specified applications. Different applications required different arguments, for example, the email script expected a sender and a recipient address whereas SSH needed commands to run once it created a connection with a remote host. Generally speaking, each of the shell scripts executed several commands relevant to the application and then terminated.

In order to avoid the classifiers from picking up on consistent packet sizes and trends due to automation, commands were chosen that would alter the packet sizes of the data being transfered. For instance, the SVN application instance script would not only check out with some files from the repository, but also alter and commit files to simulate human interaction. Additionally, some scripts were manually tuned throughout the experiment sending different commands or transferring new files. Due to the resource limits of each machine and available bandwidth, the sequence and timing behind the client cron jobs was crucial for enabling the server machines to properly respond to requests without being overburdened. Most of the post-setup administration time was spent tuning the systems to avoid spawning zombie processes caused by non-responding system services due to lagging servers. Finally, optional command-line arguments for each application's script, which supported tunneling,

was programmed. This allowed scripts to either send packets straight through the Internet, or traverse through an optional proxy tunnel. The client automation process is illustrated in Figure 4.2.

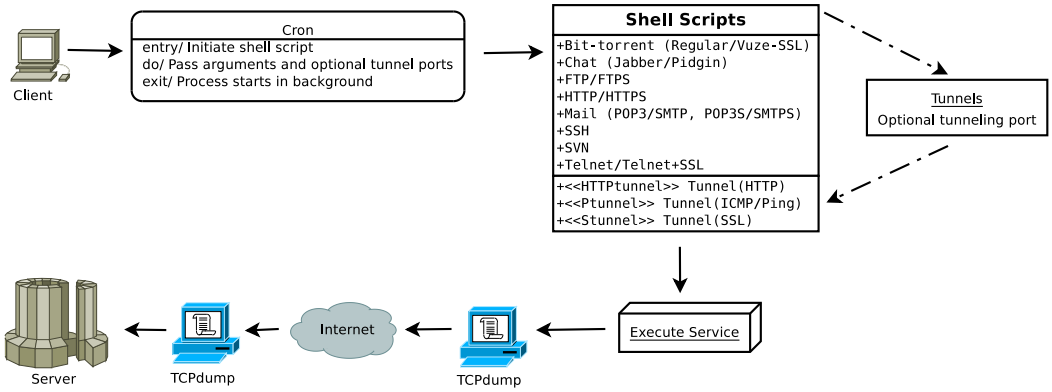

Figure 4.2: Automation from client machines

4.1.2 Server Setup

The server machines had primarily two responsibilities: *(i)* running a variety of services, which respond to various client requests, and *(ii)* acting as a proxy server for the tunneling programs. Both of these duties required *nims-server* and *taraserver* to become host connection platforms responding to client queries. Properly set configuration files for security purposes and functionality were pivotal in order to properly respond to client requests. User accounts for a collection of services, such as FTP or email, were also setup enabling the clients to login to their respective services. A thorough lock down of permissions restricted user accounts from interfering with each other should one become compromised.

To setup proxy tunnels for the tunneling programs, each client relied on its local network server to tunnel connections to the final destination. Depending on the tunneling program used, the proxy servers handled requests accordingly with configuration files either specifying background daemons to listen on a certain port or providing the end destination IP and port for incoming tunnels. To avoid clashing with well-known ports, both local and proxy ports were assigned numbers incrementally from 2000. Unfortunately, not all application instances were able to be tunneled by these programs due to tunneling program boundaries, such as not supporting

multi-port FTP connections, or other incompatibilities. An overview of the tunneling process for any of the tunneling programs used is presented in Figure 4.3.

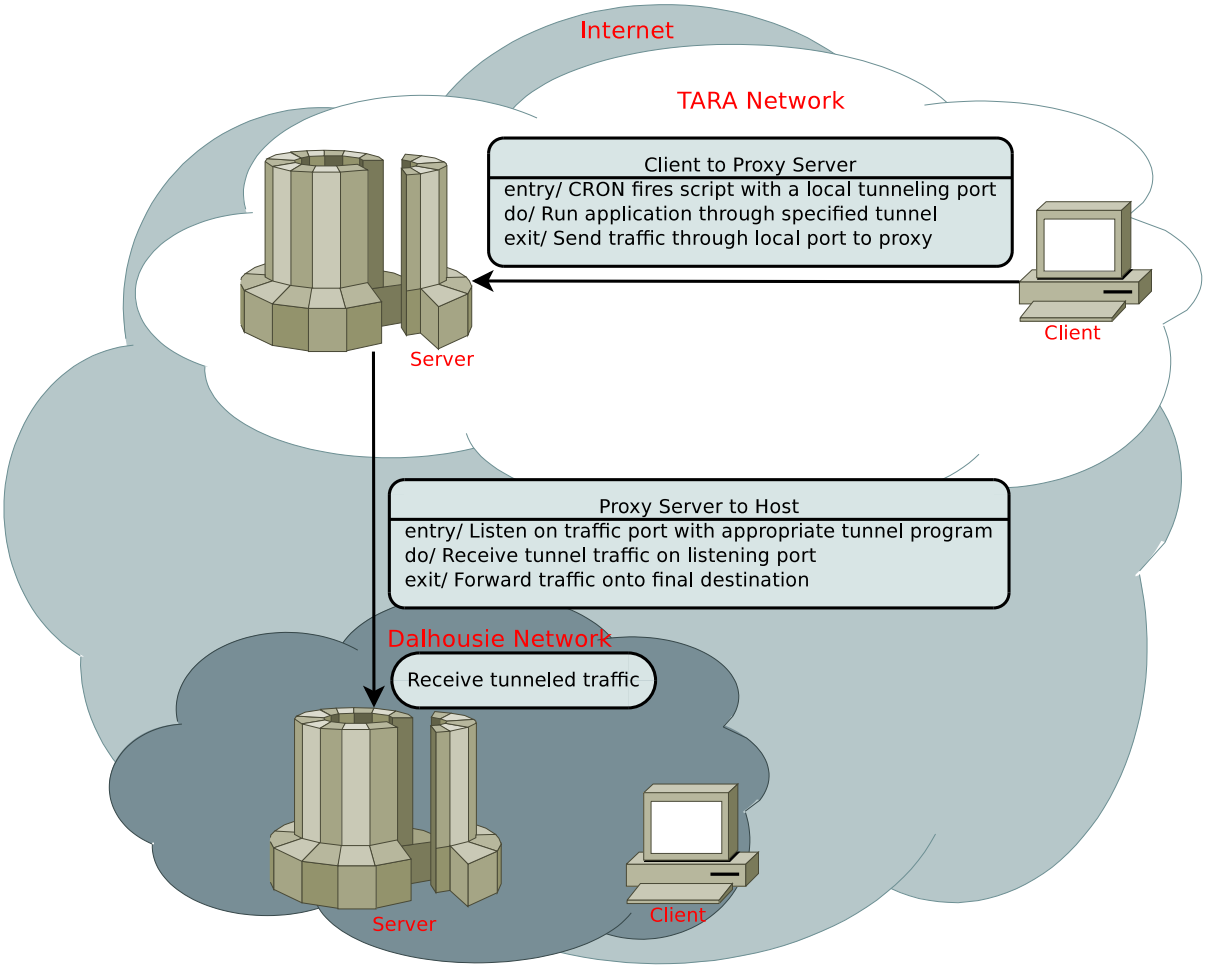

Figure 4.3: Tunneling from one network into another.

4.2 Data pre-processing

Once all of the TCPdump traffic was collected, several data pre-processing steps were taken before preparing the data for WEKA. An exploration into the usage of the NetMate program to translate the TCPdump traffic into flows is first highlighted. Following this is an explanation of the class labeling process to prepare the data for the independent multi-class runs. Finally, the methods behind the creation of the balanced training sets are then discussed.

4.2.1 NetMate

NetMate is a traffic monitoring tool, which converts IP packets into bi-directional flows and generates several metrics regarding these flows [15]. A flow is defined as a stream of network traffic during a single session of an application instance. Instead of simply combining all these packets together, NetMate actually computes several statistical characteristics about the data outputting them in a Comma Separated Values (CSV) flow record. For example, from the initial request to start an SSH connection, the subsequent commands, and final termination of the session would constitute a single flow. It is important to note that NetMate can be run live on an ethernet interface or on captured TCPdump files giving it additional flexibility [15].

NetMate supports many configuration options giving the user powerful filters to work with as well as several features to adjust. These filters can be restricted to generating flows from certain IPs and can even be as fine grained as the protocol employed [15]. Furthermore, to prevent flows from running indefinitely, a flow time-out feature is available to be adjusted. In this thesis, the flow time-out was set to 600 to follow the approach by Alshammari et al. [4]. Both TCP and UDP traffic were permitted by NetMate.

The installation of NetMate used the same source code as Alshammari et al. [4]. Due to the older GCC compiler required, another virtual machine running Ubuntu 8.04 '*Hardy Heron*' OS was installed. To start the transformation process, a shell script would loop through all the 350MB TCPdump files for each site and generate flows. These flows were then concatenated into a single 63GB CSV file containing all of NetMate's flow output attributes.

4.2.2 Assembling flows

In order to determine, which flow records belonged to the various application instances as well as their tunnels, the NetMate CSV file had to be further processed. Specifically, *IP and port information had to be removed* from the flow records leaving behind the same feature set used by Alshammari et al. [4] called *NetMate Flows*. This was accomplished through phase one of a Perl script. Many of the attributes have two types representing the forward and backward direction of the packets as they flowed between the source and destination. The feature set of flow attributes from NetMate

used in this thesis is listed below.

1. Minimum forward packet length (*min_fpkt*l)
2. Mean forward packet length (*mean_fpktl*)
3. Standard deviation forward packet length (*std_fpktl*)
4. Minimum backwards packet length (*min_bpktl*)
5. Mean backwards packet length (*mean_bpktl*)
6. Maximum backwards packet length(*max_bpktl*)
7. Standard deviation backwards packet length (*std_bpktl*)
8. Minimum forward inter-arrival time (*min_fiat*)
9. Mean forward inter-arrival time (*mean_fiat*)
10. Maximum forward inter-arrival time (*max_fiat*)
11. Standard deviation forward inter-arrival time (*std_fiat*)
12. Minimum backwards inter-arrival time (*min_biat*)
13. Mean backwards inter-arrival time (*mean_biat*)
14. Maximum backwards inter-arrival time (*max_biat*)
15. Standard deviation backwards inter-arrival time (*std_biat*)
16. Duration of the flow (*duration*)
17. Protocol used (*proto*)
18. Total amount of forward packets (*total_fpackets*)
19. Total forward volume in bytes of packets (*total_fvolume*)
20. Total amount of backward packets (*total_bpackets*)
21. Total backward volume in bytes of packets(*total_bvolume*)

22. Class label (added later by labeler)

Before discarding the IP and port information, phase two of the Perl script used this data to separate the flow records into their respective application instance files and assign an additional class label. The result consisted of 46 application instance and tunneled application instance files captured. For example, *HTTP*, *SSH*, and *SSL-Tunneled_SSH* were now all segregated in their own files containing 22 flow attributes listed above. Any traffic not belonging to an application instance or tunnel was labeled as spurious traffic and assigned the appropriate NON-SSL class label. Figure 4.4 illustrates the overall breakdown of flow traffic captured by each application instance. Further detail on the flow numbers can be found in Appendix D.

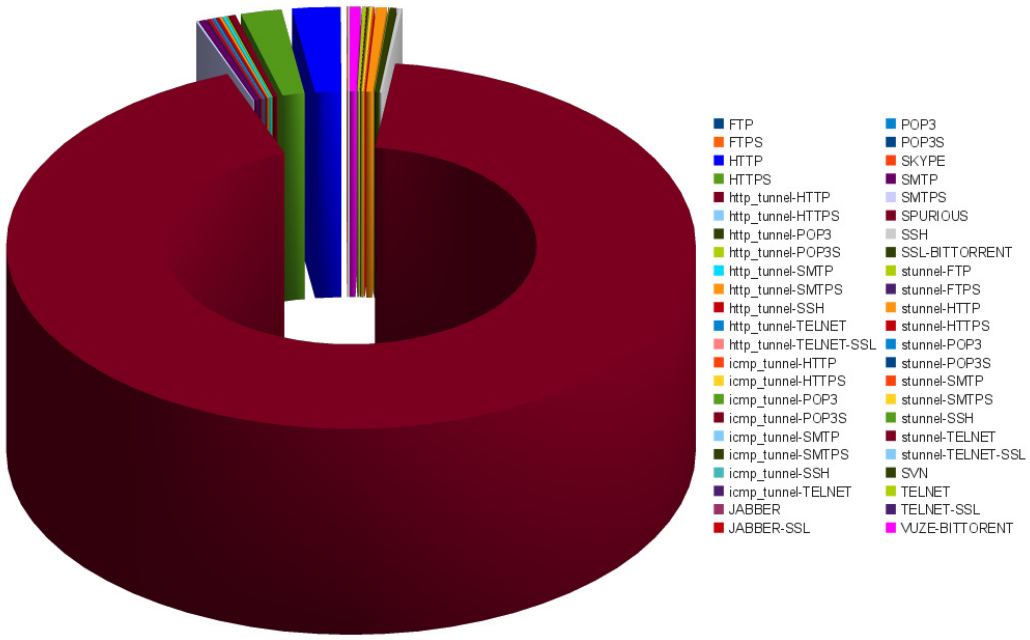

Figure 4.4: Pie chart illustrating the overall breakdown of NetMate flows for each application instance

As illustrated above in Figure 4.4, certain application instances generated many more flows then others. The culprit behind this is NetMate rejecting certain packets and not considering them part of a traffic session. In an effort to yield more packet traces, cron job frequencies were altered for certain tunnels and application instances to yield a higher volume. Despite the richer traffic set, only minimal change in quantity was observed.

4.3 Class labeling

Class labels had to be assigned to each flow record to indicate what type of traffic it represented. This section describes the class labels used for the different types of traffic found in the dataset. Following this is a brief overview of the independent multi-class runs used to test the ML algorithms.

4.3.1 Multiple classes

Although there exists many levels of granularity for class labeling, a high level labeling approach was pursued. This decision was influenced by many factors, such as some of the ML algorithms not supporting multi-class labeling, time constraints, and the investigative nature of the work. To test the performance of the classifiers on the dataset, traffic was split into several sub-classes. This section describes the different class labels used to identify the types of traffic within an application instance, specifically, *Native SSL*, *SSL-Tunneled*, and *Non-SSL*. In total, 5 runs were done on each class run using 10 fold cross-validation to minimize any data generation related biases. Additionally, the various types of independent multi-class runs are described.

Native SSL

This class consisted of applications, which employ the usage of SSL encryption through back-end libraries or APIs to OpenSSL. Traffic is encrypted using one of SSL's encryption algorithms and directly transmitted between client and server. Certain application instances have the choice of transmitting in the clear via unencrypted channels, or provide the additional security of SSL. For example, HTTP connections are unencrypted whereas HTTPS connections use SSL and are considered a native SSL application instance. Below is a list of all application instances, which were classified as native SSL.

- HTTPS
- SMTPS
- Jabber-SSL (Chat)
- POP3S

- FTP-SSL
- Telnet-SSL
- Bit-torrent SSL
- Skype[1]

SSL-Tunneled

Making use of the open source Stunnel program, SSL-Tunneled traffic utilizes the backend OpenSSL libraries to encrypt communication between a client and proxy. Many native SSL and non-SSL application instances can be tunneled in such a manner creating a distinct type of traffic. So long as the protocol employed and the application are compatible with Stunnel, the traffic was labeled as being tunneled. Below is a list of all application instances, which supported Stunnel.

- HTTP
- HTTPS
- SMTPS
- SMTP
- FTP
- FTP-SSL
- POP3
- POP3S
- Telnet-SSL
- Telnet
- SSH

[1]Skype was tested as both SSL and Non-SSL as it employs SSL for the initial login authentication [6] before utilizing its proprietary encryption protocols.

Non-SSL

All unencrypted non-SSL traffic fits into this class, including background traffic along with application instances such as SSH which do not use SSL. The non-SSL class represented the largest portion of data collected, mainly due to the enormous size of the *spurious* application instances. Below is a list of all application instances, which were classified as non-SSL.

- HTTP
- SMTP
- Jabber (Chat)
- SVN
- Bit-torrent
- POP3
- FTP
- Telnet
- SSH
- Skype

4.3.2 Multi-Class Runs

Once flow records were separated into their correct classes, multiple runs were performed to produce the final results. Each of these comparisons generated a different model showing how well the classifiers performed on a set of two classes, i.e. binary classification. Each class comparison, *SSL vs Non-SSL*, *SSL vs STunnel*, and *Non-SSL vs Stunnel*, are discussed in more detail below.

SSL vs Non-SSL

The comparison of SSL encrypted versus unencrypted traffic plays the base-case role in determining how well the classifiers can detect the differences between SSL and non-SSL traffic. Any application instance that used some form of SSL, whether it was native or tunneled, was included as such with the remaining traffic given a non-SSL label. The results from these runs demonstrated the flow attributes the ML algorithms selected as important for classification.

SSL vs STunnel

The results from this experiment indicated how well the classifiers were able to distinguish between applications using SSL natively and those being tunneled. Any application instance using SSL natively was labeled SSL, whereas any application being tunneled through Stunnel was labeled SSL-Tunneled. Furthermore, native SSL using Stunnel were labeled as SSL-Tunneled as their packets were still being tunneled through a proxy.

Non-SSL vs Stunnel

This class comparison demonstrated how well the classifiers were able to detect unencrypted traffic from the hybrid SSL and Non-SSL nature of SSL-Tunneled applications. This hybrid nature is due to the fact that traffic being SSL-Tunneled appeared both encrypted and unencrypted at one point throughout its path to the final destination. Non-SSL encrypted application instances were labeled NON-SSL and only SSL-Tunneled applications were labeled as SSL-Tunneled no native SSL application instances were included.

4.4 Training

Once all the application instances were labeled and contained all the NetMate flow attributes, a sample set had to be taken for training from these CSV files. Given the 46 different application instances and their varying sizes, performing subset sampling required a custom shell script and specialized algorithm. Following the training set size approach taken by Alshammari et al. [1], training sets measuring in sizes of

6000, 9000, 12000, 20000, 50000, 100000, 150000, and 500000 flows were taken. Each training set was balanced in a 50/50 split of the respective classes to be trained. Thus, for a total training set size of 6000 flows, 3000 from each class would be extracted from the application instance files representing the class.

While certain independent multi-class runs involved fewer application instances, the issue of balance amongst them all required special attention. Since application instances within the dataset measured between kilobytes to gigabytes of flow records, a specialized script ran an algorithm to help compensate for those with not enough flows. Generally speaking, for application instances where there was not enough traffic for training, only half of the initial total flow records were taken. In order to maintain the overall class balance, the other half was extracted from the largest file for that class. In some training set sizes, the application instances were able to provide enough flow records as required from the training Algorithm 1. In other cases, the application instances varied in contribution due to the aforementioned smaller sizes causing more to be extracted from the largest application instance of the respective class. The actual numbers of flow records extracted from the different application instances for the different training set sizes of the final training sets can be found in the analysis section. Algorithm 1 illustrates the pseudo-code followed to build the training sets.

4.5 WEKA

In order to apply the ML algorithms and determine which among them had the best performance given the different sizes of training sets, the WEKA platform was chosen. WEKA is a Java-based environment for applying ML techniques to output statistical information and visualizations [20]. Specifically, its purpose was to provide an implementation of the different ML algorithms used as well as output models and results. This section begins by highlighting the additional WEKA preprocessing steps that were taken. The different ML algorithms used to classify the data are then introduced. Finally, a brief description on how the results were summarized for analysis and an overview on how the algorithms were ranked are presented.

Algorithm 1 Training set generator

Input: A number N representing the training set size specified by the user.

Output: 50/50 class split balanced training set.

1: Calculate total number of flows from all application instances for the class run $totalFlows$
2: Count the number of individual application instance class files
3: Determine how many flows to extract from each file per class $numFlowPerClass$
4: Figure out which application instances posses the largest number of flows per class
5: **for** every applicationInstance $ineach$ class **do**
6: **if** ($applicationInstance$! = $largestClassFile$) **then**
7: **if** (size($applicationInstance$) < $numFlowPerClass$) **then**
8: Extract initial half of total flow records from application instance placing them in the training set
9: Calculate difference between half and requested $numFlowPerClass$
10: Add difference to $additionalFlows$ number to be extracted from largest class file
11: Increment $numCompensateFlows$ counter to keep track of any rounding discrepancies
12: **else**
13: Extract initial $numFlowPerClass$ from application instance placing them in the training set
14: **end if**
15: **end if**
16: **end for**
17: $numExtractedFromLargest = numFlowPerClass + additionalFlows + numCompensateFlows$ {Deal with largest application instance files for balance}
18: Output statistics regarding how many flow records each application instance contributed

4.5.1 WEKA preprocessing

Before training or testing could commence, the flow records for each application instance as well as the generated training sets had to be transformed into a WEKA readable format. This was accomplished using WEKA's built in CSV converter, *weka.core.converters.CSVLoader*, which outputs WEKA's native ARFF format [20]. Unfortunately, due to hardware memory resource constraints, WEKA can only operate with a maximum number of flow records pending available memory for the Java virtual machine before throwing an exception. While this had no affect on the training set files, the test sets required additional processing. As such, each application instance file was split into multiple files containing approximately 1000000 flow records. As a requirement of WEKA, an instance of both classes must appear in a given ARFF file. In order to be in compliance with this assertion, a script had to check to make sure each 1000000 split file contained at least once instance of each class.

4.5.2 Machine Learning Algorithms

While WEKA supports many different ML algorithms that could be used to classify the data, we wanted to choose those, which have had success in the past and could output the human readable solutions [1, 4]. To this end, a metaheuristic, decision tree, probabilistic, and rule learner ML algorithms were chosen. The following section is a brief description on their unique abilities. It is important to note that each algorithm was run with 10 fold cross-validation and all WEKA's default settings were used unless explicitly stated.

AdaBoost

AdaBoost is a ML algorithm using ensemble learning, which alters weak learning classifiers by assigning weights with the overall goal to properly identify instances. Generic boosting is accomplished by starting with all weights at one and then iteratively adjusting them by increasing the weights of misclassified examples and decreasing those of correctly classified examples. Each iteration generates a new hypothesis and holds a unique weight pending how accurately it classified the set. A hypothesis

achieving a higher degree of correctly classified examples will hold more weight in the final ensemble; meaning the size of any given hypothesis correlates to the weight it has in the total ensemble. The final hypothesis output is a weighted-majority collection of the entire hypothesis collection [44]. AdaBoosting differs from generic boosting by using weak learners to determine a weighted error on the example giving it a more accurate result then random guessing [44]. This action allows AdaBoost to generate a more accurate ensemble hypothesis for the training set.

WEKA's AdaBoost implementation deals with multi-class sets of data where the algorithm no longer accepts any weak hypothesis where the accuracy is less then half. This forces the algorithm to abandon certain weak learners whereas binary classification is less stringent [20]. Decision stumps representing single branch decision trees are also implemented generating useful output pending the size of the ensemble hypothesis [20]. A weighted-majority will be obtained eventually as the ensemble increases and the error rate heads towards zero [20]. This indicates that the number of decision stumps needed to accurately classify a training set is equal to the number of hypothesis making up the ensemble when the error rate equals zero.

As an optional AdaBoost attribute, WEKA supports the usage of decision trees instead of decision stumps. Unfortunately, testing indicated worse results compared to the default decision stumps so this approach was dropped. Finally, WEKA's output for AdaBoost lists the weight assigned to each decision stump allowing the user to easily see which attribute provided the greatest knowledge for the correct classification of an example.

C4.5

The C4.5 decision tree mimics the way a user would make up their mind between a choice of outcomes based on a known environment. By taking into account all available input attributes, numerous decisions can be made based on the data to predict the inevitable outcome with a high degree of accuracy. The reasoning behind calling it a decision tree can be attributed to its resulting structure resembling that of a tree with each decision creating a branch. It is also important to note that decision trees are fully transparent ML algorithms enabling researchers to understand how they arrived at the resulting decision.

Branches, or decisions, in a decision tree are created by attribute comparisons which relay the highest amount of information gain. Information gain represents how well a given decision will separate the output classes the most — that is, which attributes can be used to efficiently separate the two output classes. This process is repeated until all the data is properly classified, as such, they are particularly useful when dealing with noisy data.

Trees have many shapes and sizes indicating important characteristics about their classification ability on a data set. Generally speaking, preference is given towards short trees as they signify a relatively easy decision making process. Additionally, decision trees with high information gain attributes near the root of the tree separating a majority of the dataset are equally sought after. Both of these principals also coincide with Ockham's razor by striving towards the simplest tree model.

One of the main drawbacks of decision trees is their ability to over-fit the training data. Thankfully, this can be easily solvable by pruning the tree in the same manner one would alter a physical tree by removing excess branches. This can be accomplished by recursively removing these irrelevant attribute comparisons that provide little to no information gain [44]. WEKA's implementation of the C4.5 decision tree algorithm allows this post pruning to occur ensuring the optimal tree is output from the entire hypothesis space [20].

Naive Bayes

By utilizing prior knowledge about an event, we are able to sometimes predict the resulting outcome. As a probabilistic classifier, Naive Bayes uses conditional probabilities to arrive at a final decision by applying Bayes theorem.

Probability can be described as the degree of belief something will occur [44]. In some cases the probability of an event occurring is dependent on an action while in others those actions have no effect on the outcome. The latter case describes the notion of independence amongst variables. There are two types of probabilities; (i) unconditional and (ii) conditional. Unconditional probabilities are related to the belief that an action will occur given that no other information is available. If we let the probability P of an event A have a 0.1 likelihood of occurring, it would be written as P(A)=0.1. Conditional probabilities take this a step further by implying that if

a given action happens, the likelihood of another action occurring can be assigned a number. For example, given event B, the probability of that event A will happen is 0.2 according to the following notation: P(A|B)=0.2.

Making use of the above probability rules, Rev. Thomas Bayes invented the Bayes' Theorem to deduce the inverse probability of an event [44]. For example, if we know the probability of an event A and B occurring independently, as well as the probability of B given A, we can determine the probability of A given B using the following notation: P(A|B) = P(B|A)P(A) / P(B).

For classifying the flow traffic in this thesis, the classifier had multiple flow attributes to use. In this case, Naive Bayes was applied since all the Netmate flow attributes are conditionally independent. The correct implementation of the full joint distribution using Naive Bayes according to Russell et al. [44] is shown below.

$\mathbf{P}(Cause, Effect_1, ..., Effect_n) = \mathbf{P}(Cause) \prod_i \mathbf{P}(Effect_i|Cause)$

RIPPER

Following the same way a network administrator would apply access control lists, or rules, to a firewall, RIPPER uses a similar principle to classify traffic. By concatenating rules using logical **OR** and **AND** operators, it generates a rule set in which the classifier can accurately detect the out-classes. RIPPER contains two main phases, the first loops through a building stage where it grows, and prunes the classifier, while the second is responsible for optimization [20].

During the *growing* step, the algorithm continually adds conditional statements until the rule obtains complete and accurate classification. This process utilizes the same information gain principal discussed in the C4.5 decision tree and selects the condition with the highest gain [20]. The *pruning* step again applies the same principal as in the C4.5 decision tree by cumulatively pruning each rule.

According to the WEKA documentation, all of the above steps are then repeated in order to *build* the algorithm until the following conditions are met:

- The current description length of the rule set and examples is 64 bits greater then the smallest description length met so far
- No other positive examples exist

- The error rate exceeds 50

The second stage involves optimizing this initial set of conditionals, or rule set, that has been created thus far. Firstly, two pruned alternatives are produced for each rule of the initial rule set from a random data sample using the growth and pruning processes [20]. Specifically, one of these alternatives is produced from an empty rule whereas the other comes from greedily adding antecedents to the original rule [20]. At this point, computation to determine the smallest description length is undergone for both the alternative pruned rules as well as the original one. The rule with the smallest length is chosen as the final representative in the rule set [20]. More rules are then generated using the remaining newly calculated positives in the building stage [20].

4.5.3 Performance Metrics Employed

Once training and testing was completed, all that remained were the models generated by the training files and numerous WEKA text output files for each test. These output files included statistics on the True Positive (TP), False Positive Rate (FPR), and Recall rates in the form of confusion matrices. In order to start gathering the results, a shell script would first concatenate the last 23 lines from each of these files into a single result file. By including the ML algorithm used in the filename, the script was able to keep each algorithm's performance isolated. A separate Perl script would then parse the entire concatenated output file computing the necessary statistical measures. Once all the statistics were calculated, the Perl script would output the results in a CSV format for easy import into a spreadsheet.

In order to properly gauge the performance of the ML algorithm classifiers, various metrics popular amongst other network traffic classification methods were implemented [46, 35, 4, 1]. Specifically, accuracy, FPR, and recall are all utilized to determine the overall performance whereas the False Positive Rate Analysis (FPRA) breaks down the ML algorithm's robustness over each application instance in the class run.

Accuracy represents the ratio measurement of how closely classified the ML algorithm came to achieving 100% perfect classification. It is calculated using the ratio of the number of correctly classified instances and the total number of instances

belonging to that class.

The FPR, according to the WEKA documentation, represents how many instances the classifier misclassified as the other class. It is calculated using the ratio of the number of incorrectly classified instances and the total number of instances belonging to the misclassified class.

Recall is the same as True Positives and measures the precision of the ML algorithm for a certain class. It is calculated using the ratio of the number instances identified as a certain class and the total number of all possible instances truly belonging to that class.

Chapter 5

Results and Analysis

This chapter highlights the results obtained for the different class runs and presents an analysis behind the performance of the top algorithms. A more detailed explanation of the FPRA process is first presented highlighting the training data set size investigation process. Following this are the results from the base-case SSL vs Non-SSL run using the entire dataset. A closer look to see how the ML algorithms were able to differentiate between SSL and SSL-tunnel traffic is then brought to light. An examination of the Non-SSL vs SSL-Tunnel classes then adds additional knowledge behind how the application instances were classified. Both SSL vs SSL-tunnel and Non-SSL vs SSL-tunnel class runs used all the flows from every applicable application instance belonging to the classes involved for testing. Further information on how many flows each application instance contributed to the training files can be found in Appendix D. Finally, the robustness of the top performing SSL vs Non-SSL model is tested against an unseen dataset.

5.1 False Positive Rate Analysis — FPRA

The overall percentages on the performance of a classifier presented a generalized means to judge those ML algorithms, which could be fine-tuned for secondary optimization experiments. Nonetheless, the overall accuracy, FPR, and Recall metrics fail to provide any detail regarding the individualized results of the application instances within the classes. For example, the classifier could achieve a high degree of accuracy overall, but a certain group of application instances deemed important may be the ones being misclassified the most. As a result, another script was needed to keep all the application instances isolated from each other for testing and then gather results from the WEKA output.

The FPRA is a metric to determine the individual performance of a ML algorithm model against each application instance. From this, we can extrapolate how well the

ML algorithm performed on different types of traffic from the various standalone applications as well as those being tunneled. The algorithm, depicted in Algorithm 2, separates and runs the desired ML algorithm model from the training run against each application instance then proceeds to output individualized results.

Algorithm 2 FPRA algorithm

Input: NetMate output CSV file of total traffic

Output: Performance of ML algorithm training model on each application instance

1: Label the application instance traffic into the correct classes for the respective run
2: Separate and split the application instances into isolated folders
3: For each application instance, split the traffic into 1000000 flow record sized files
4: Preprocess the split files to convert them into ARFF format for WEKA
5: **for** every *ApplicationInstanceFolder in SeparatedDirectoryListing* **do**
6: **for** every *SplitFile in Directory* **do**
7: Run WEKA model against ARFF file saving output results in segregated application instance directories
8: **end for**
9: **end for**
10: Run modified version of summarizer to parse WEKA output for each application instance
11: Run Perl script on each of the files generating CSV output of the computed results

By carefully modifying existing scripts where applicable, this process guaranteed consistency in the results amongst the individualized application instances. The CSV records output holding the results on a per application instance basis were then ported into spreadsheet for graphical visualization.

5.1.1 Exploring different Sizes of training sets

The question of whether or not including additional flows of the misclassified application instances while maintaining the 50/50 class split balance would help improve the classifier's accuracy was raised. As such, a method for changing the size of the training sets based on the FPRA was developed. I decided to apply this principle

to the individual application instances with the highest degree of misclassification. A new training script was built using the application instances whose accuracy of correctly identified instances was lower than 50% from the FPRA as input. In this case, Algorithm 1 would add an additional 7% more flow records from the application instances listed as input if they were available. The resulting output training set still maintained the 50/50 class split of flow records, however, the number per application instance varied amongst the classes. For example, if SSH had poor performance in the FPRA and had enough flows were available, an additional 7% of the requested flows from SSH would be added to the training set. To ensure balance was maintained, that additional 7% of flow records was then subtracted from the other application instances.

5.2 Base-case: SSL vs Non-SSL

The **AdaBoost ML algorithm** achieved the best classification performance with the *largest 500000 training set size*. Results for all the ML algorithms with a training set size of 500000 are displayed in Figure 5.1 while the remaining experiments using other training set sizes can be found in Appendix A.

In general, the results for the base-case SSL vs Non-SSL run presented fluctuations in performance amongst the ML algorithms as the training set size increased. As shown in Appendix A, the poor performance rate of Naive Bayes suddenly increased dramatically when the training set size hit 100000 and continued achieving good results for each consecutive larger training set size. Performance on the smaller training set size runs was expected to be lower compared to the others as the number of flow records allowed was buffered down by the requested flows. By having such a high demand of representation from all the class application instances, it actually restricted the number of flow records from each that could be included thus preventing the ML algorithms from picking up on key learning features. For example, consider the smallest training set size of 6000, limiting each class to 3000 flow records. With a total of 46 application instances, 26 belonging to the SSL class and 20 Non-SSL, the total number of flow records per application instance is 115 for SSL and 150 Non-SSL. For the majority of application instances, this does not adequately represent the behavior amongst all the flow records leading to poor classification. The results

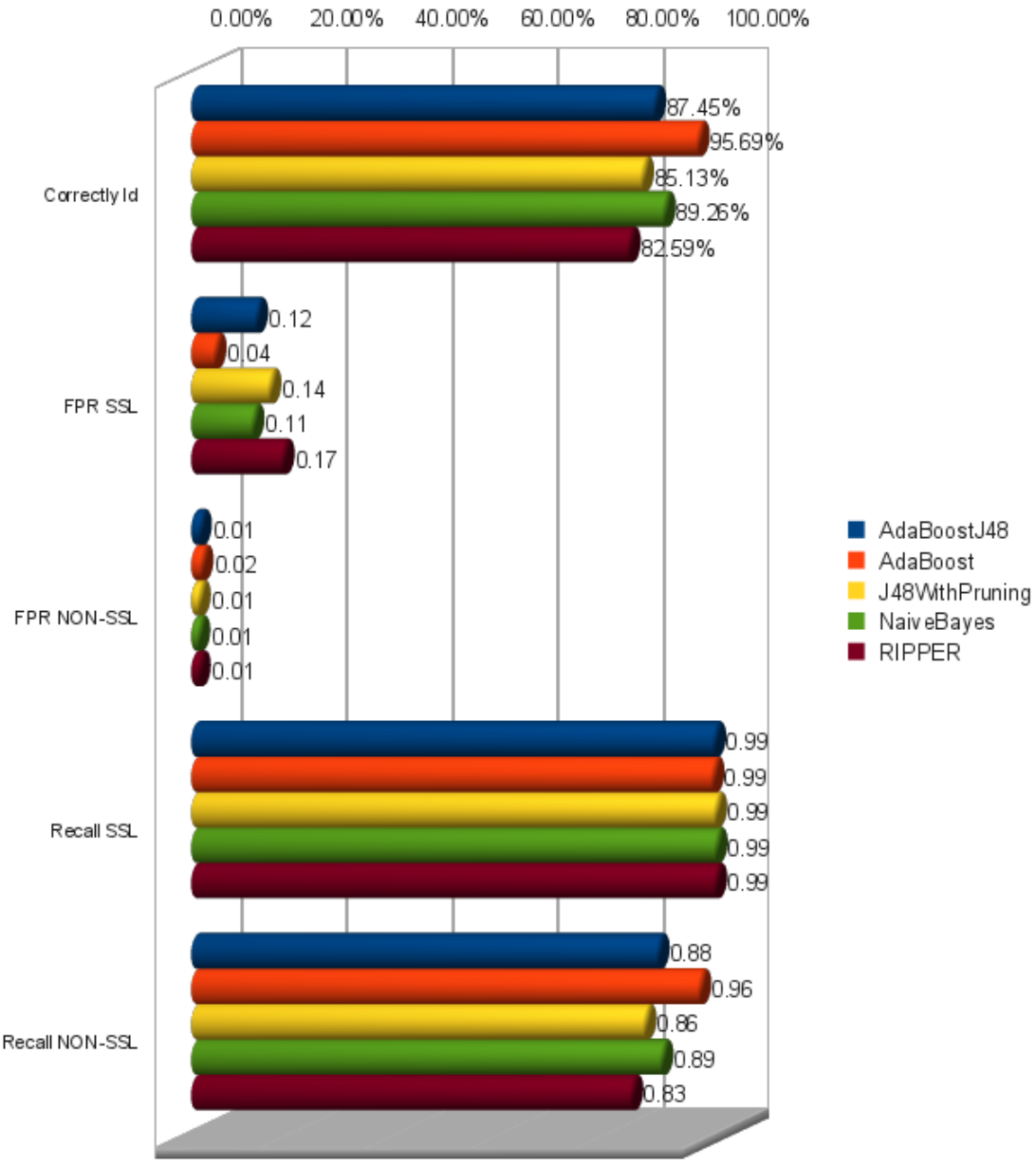

Figure 5.1: Adaboost performs best overall with a training set size 500000 in SSL vs Non-SSL run

corroborate with this theory as all ML algorithms witness improved performance as the training set sizes increased.

Another important point to mention about the training set sizes is how much impact Algorithm 1 had on the training set. As the training set sizes increased, so did the demand for the number of flow records from the application instances. According to Algorithm 1, if an application instance did not have enough flow records, it would

take 50% of the flows and leave the rest unseen for testing. Closer examination of the training logs revealed that half of the application instances for the SSL class were subjected to this statement with the largest training set size. With the exception of two, all of them were tunneled application instances using some form of SSL from different tunneling protocols. While it would seem this would introduce an unfair bias causing an overestimation of the ML algorithm's performance, the FPRA, illustrated in Figure 5.2, still showed how well the algorithm was able to distinctly separate the individual application instances. By implementing this segregation, we can see that almost all of the application instances that scored lowest in the FPRA *also contributed the least amount of flows during training.* This split amongst the SSL class shows that if those low ranked application instances had enough flows, the classifier performance of the ML algorithm should also increase.

In order to determine how AdaBoost achieved these results, an investigation into how the weights were assigned was required. As such, a Perl AdaBoost extraction script was used. The script would parse through the result files belonging to the AdaBoost algorithm and extract the weights assigned by WEKA. The overall weights for each attribute would then be computed and the script would output the results in CSV format. Table 5.1 shows the attributes chosen by AdaBoost in order to achieve the highest performance in this run.

mean_fpktl	37.33%
total_fvolume	20.12%
mean_bpktl	15.86%
std_biat	10.83%
total_bvolume	7.16%
min_bpktl	6.00%
max_bpktl	2.71%

Table 5.1: AdaBoost weights used to achieve results in SSL vs Non-SSL run

Of the total 22 flow attributes available, AdaBoost only used seven to classify the base-case SSL vs Non-SSL run. The algorithm focused on the *mean_fpktl* primarily as the attribute carrying the most weight throughout all of the larger training set sizes. By doing so, AdaBoost was classifying SSL and Non-SSL traffic by relying on the mean forward packet length sent from the client. This indicates AdaBoost was picking up on the extra SSL overhead, which adjusted the size of the packets. The

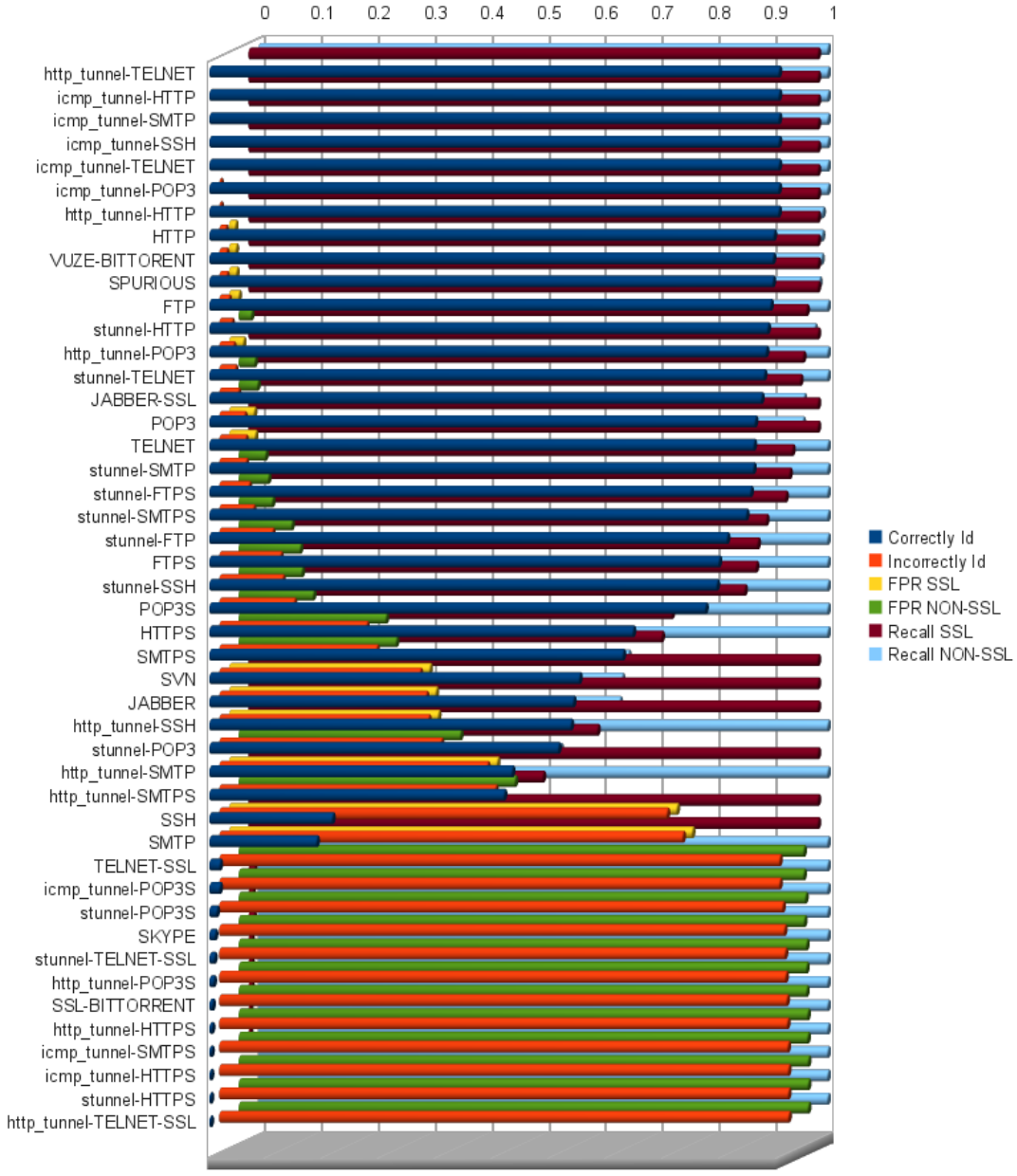

Figure 5.2: FPRA of AdaBoost 500000 training flows on SSL vs Non-SSL run. The graphic illustrates the results from each performance metric employed for each application instance. They are sorted by the best classification performance on top to the worst on the bottom.

packet's new mean size could then be used to separate encrypted SSL requests from the unencrypted requests coming from the client. Furthermore, the total amount of bytes sent from the client in the *total_fvolume* flow attribute supports this claim by indicating larger amounts of network traffic flowing from the client to handle the SSL

encryption. A similar case exists for the lesser weighted *total_bvolume* flow attribute responses from the server.

As the packets made their way back from the server to the client, AdaBoost picked up on the *mean_bpktl* attribute indicating a unique response from the server for many of the application instances. This was most likely accomplished by focusing on the SSL handshake process response from the server side. Due to the use of tunnels, this packet length would have varied since not all responses contained the same signature pattern as native SSL application instances. The results concur with this claim as the majority of the worse misclassified application instances were mainly tunneled traffic as shown in the FPRA in Figure 5.2.

The next highest ranked weight belong to the *std_biat* flow attribute illustrating a unique delay in the server responses. This could be explained by the extra processing power required to encrypt the network connection causing a statistically significant delay. In this case, the standard deviation spread shows the computing complexity required for SSL encryption was resource intensive enough to stagger the packet arrival and response times back to the client compared to the lack of processing power required for unencrypted traffic. This is further supported by AdaBoost using the *mean_bpktl* and remaining *min_bpktl*, and *min_biat* flow attributes to classify the traffic.

Several fine tuning approaches were taken to try and improve the performance of AdaBoost. In addition to changing the weak learner classifier used from decision stumps to C4.5, another available option through WEKA included using re-sampling instead of re-weighting, which produced no improvement throughout each training set size. Similarly, altering the weight threshold for weight pruning produced little difference. The results from both of these approaches can be found in Appendix E. Finally, in an attempt to offer more representation to capture the behavior of the application instances with the highest amount of misclassified flow records, the fine tuned training set algorithm was used to explore a better training set size (more applications instances). Unfortunately, only four of the 14 lowest scoring application instances had enough flow records to be added. To remedy this situation, the data collection process was once again initiated to capture specific application instances with limited flow records. Before sufficient flows packets could be captured to be

turned into flows, TARA announced it was going out of business and the network setup was subsequently terminated. As a result, the generation of additional traffic which could be captured in a heterogeneous network setup was impossible. A breakdown of the application instances composing each class in the training set is presented in Figure 5.3 and Figure 5.4.

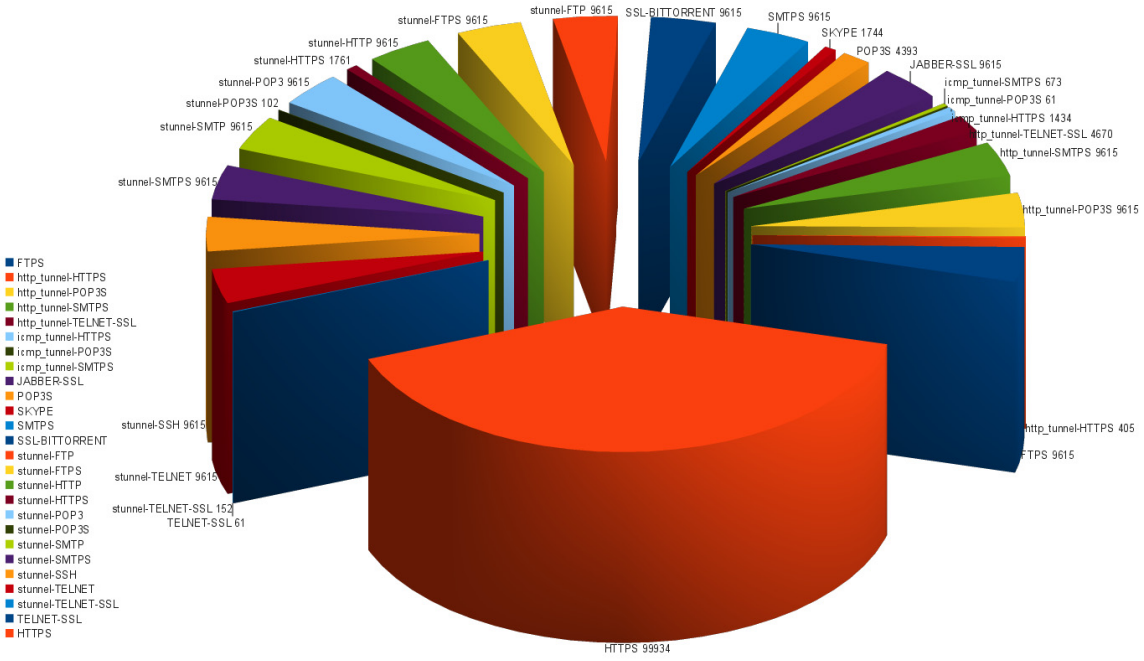

Figure 5.3: SSL class flow representation for training set size 500000 in SSL vs Non-SSL run

5.3 SSL vs SSL-Tunnel

With the exception of Naive Bayes, the phenomenal results shown in Figure 5.5 clearly illustrate the remaining the ML algorithms scoring well above 87% in almost every training set size. Unfortunately, a closer look at the performance revealed a high FPR for many of the ML algorithms. After sorting to see which ML algorithm had the highest accuracy of correctly identified instances as well as the lowest FPR, **AdaBoostJ48** using a *training set size of 6000* was ranked the highest. Standalone C4.5 came in a very close second place with key differences in the FPR and Recall rates creating a small performance gap compared to AdaBoostJ48.

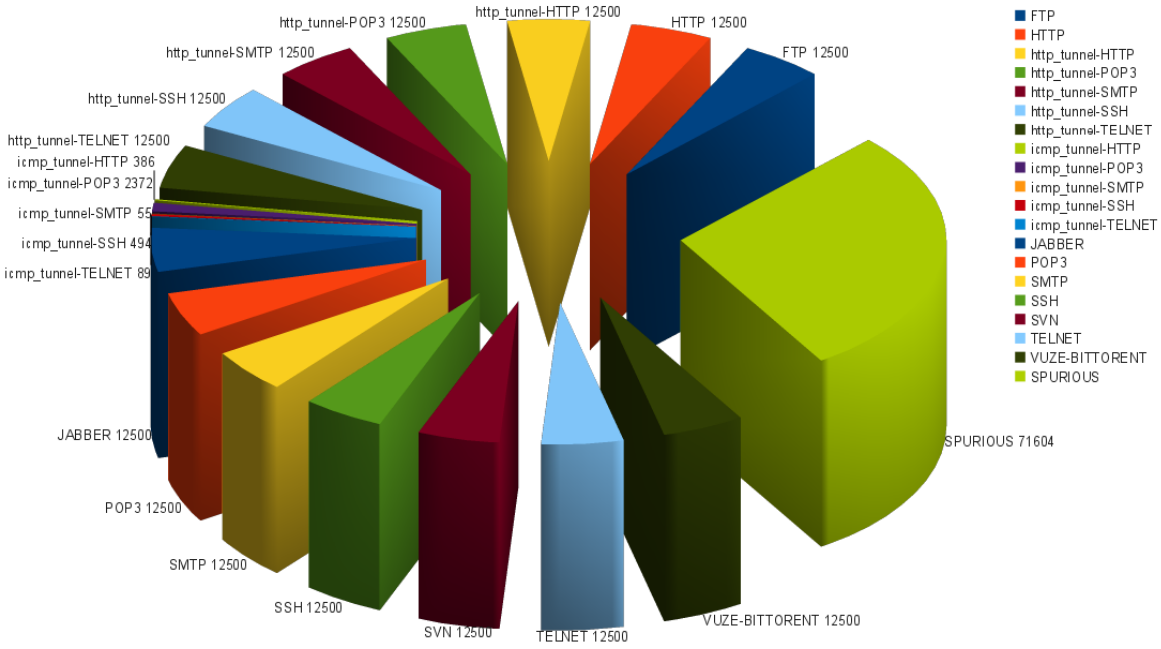

Figure 5.4: Non-SSL class flow representation for training set size 500000 in SSL vs Non-SSL run

Compared to the base case run, the increased FPR levels amongst most of the ML algorithms may be attributed to the fact that both classes use SSL at one point during packet communication between the client, optional proxy, and server. Omitting the extra hop in network traffic, this would cause the feature sets to become almost identical leading to a higher misclassification of SSL traffic. The results from the AdaBoostJ48 tree with the greatest weight, found in Figure 5.6, support this theory by assigning the primary root node weight to the *mean_bpktl* flow attribute. This clearly shows the AdaBoost classifier being forced to look at the packet length response from the server as either being encrypted for native SSL or unencrypted for SSL-Tunnel communicating with a proxy. Additionally, the difference in the flow attribute size is prevalent in the tree for the multiple packet length flow attributes due to the differences between native SSL and SSL-Tunneled traffic. The size difference can be attributed to how the tunneled packets shed their encryption overhead at one point during the flow. The other decision trees generated by AdaBoostJ48 held less weight but utilized the remaining flow attributes to reach their classification results.

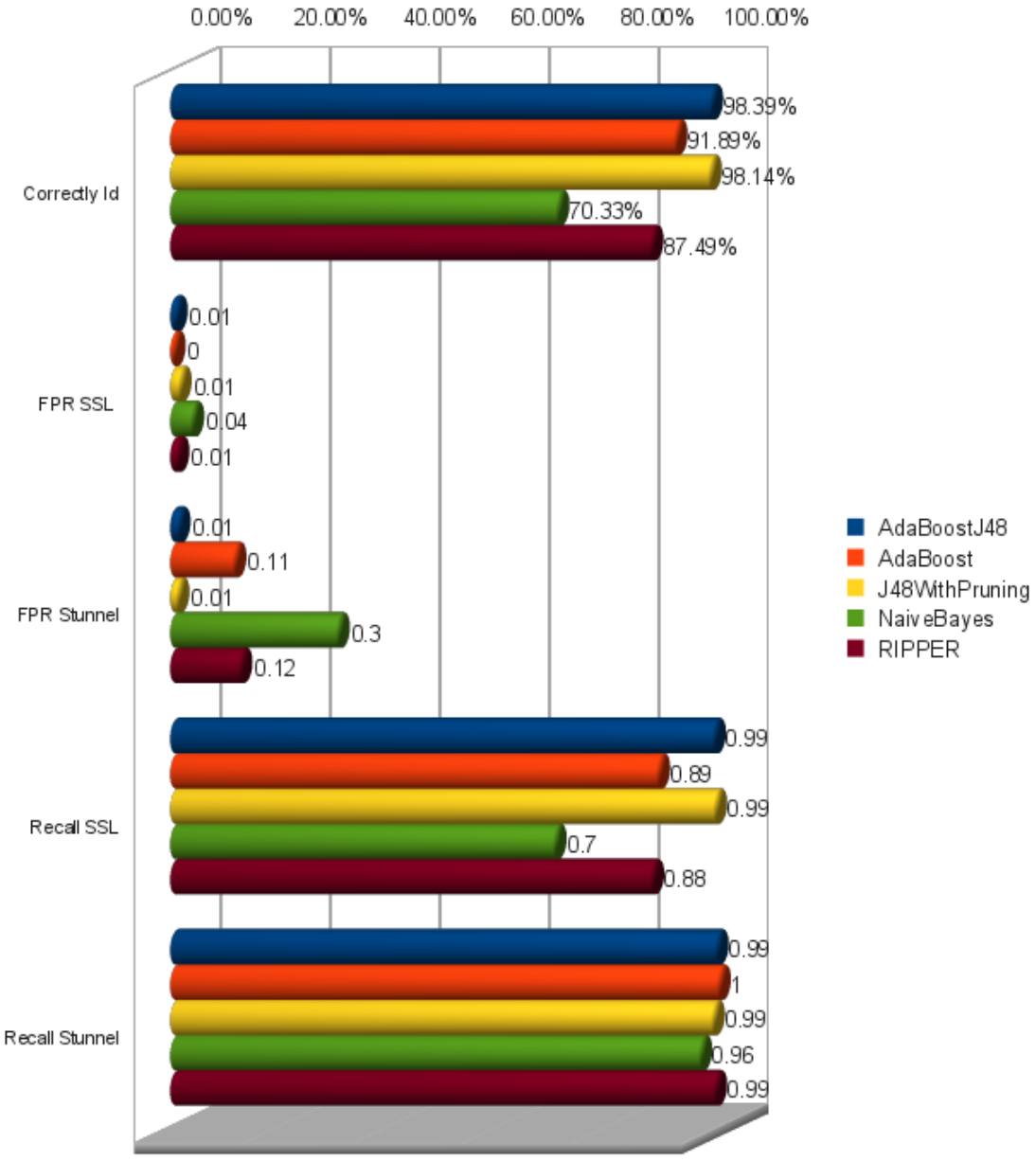

Figure 5.5: AdaBoost using C4.5 decision trees performs best overall with a training set size 6000 SSL vs SSL-Tunnel run

It should also be noted that as the training set size increased, even better performance was obtained as illustrated in Appendix B. Unfortunately, this is primarily due to the availability of flow records for each application instance. By analyzing the training set output, it became quickly apparent that instead of the classifiers achieving better results, the training was getting easier with the application instances unable to keep up with the flow record demands from Algorithm 1.

```
mean_bpktl <= 1062
|   min_fpktl <= 99
|   |   std_biat <= 5955590
|   |   |   min_bpktl <= 124: SSL (2143.0/428.0)
|   |   |   min_bpktl > 124
|   |   |   |   proto <= 20
|   |   |   |   |   std_biat <= 653997
|   |   |   |   |   |   mean_fpktl <= 222: SSL-Tunnel (62.0)
|   |   |   |   |   |   mean_fpktl > 222: SSL (21.0)
|   |   |   |   |   std_biat > 653997
|   |   |   |   |   |   std_biat <= 756462: SSL (62.0)
|   |   |   |   |   |   std_biat > 756462
|   |   |   |   |   |   |   std_biat <= 3133209
|   |   |   |   |   |   |   |   min_fpktl <= 96: SSL (9.0)
|   |   |   |   |   |   |   |   min_fpktl > 96: SSL-Tunnel (16.0)
|   |   |   |   |   |   |   std_biat > 3133209: SSL (62.0)
|   |   |   |   proto > 20: SSL-Tunnel (16.0)
|   |   std_biat > 5955590
|   |   |   min_fpktl <= 85
|   |   |   |   duration <= 0: SSL-Tunnel (237.0)
|   |   |   |   duration > 0: SSL (16.0)
|   |   |   min_fpktl > 85
|   |   |   |   max_fpktl <= 75: SSL (82.0/1.0)
|   |   |   |   max_fpktl > 75
|   |   |   |   |   proto <= 8: SSL-Tunnel (3.0)
|   |   |   |   |   proto > 8: SSL (2.0)
|   min_fpktl > 99
|   |   min_bpktl <= 84
|   |   |   mean_fpktl <= 414: SSL-Tunnel (63.0)
|   |   |   mean_fpktl > 414: SSL (309.0)
|   |   min_bpktl > 84
|   |   |   mean_fpktl <= 263
|   |   |   |   mean_bpktl <= 970: SSL (10.0)
|   |   |   |   mean_bpktl > 970: SSL-Tunnel (91.0)
|   |   |   mean_fpktl > 263: SSL-Tunnel (2083.0)
mean_bpktl > 1062: SSL (713.0)
```

Figure 5.6: Highest weighted AdaBoost C4.5 decision tree in SSL vs SSL-Tunnel run

The FPRA, found in Figure 5.7, highlights those few application instances, which had poor classification results, specifically *stunnel_POP3S*, *stunnel_TELNET-SSL*, and *icmp_tunnel_POP3S*. Closer investigation of the training log revealed that these application instances also presented the least number of flows to the training algorithm. This repeats the same analysis from the SSL vs Non-SSL run in regards to the ML algorithm being unable to adequately learn the behavior patterns with such few flows.

Finally, since these application instances also represented those without enough flows to meet the minimum required amounts from their respective classes, any increase in the size of the training set to contain more flow records would be futile. A breakdown of the application instances composing each class in the training set is presented in Figure 5.8 and Figure 5.9.

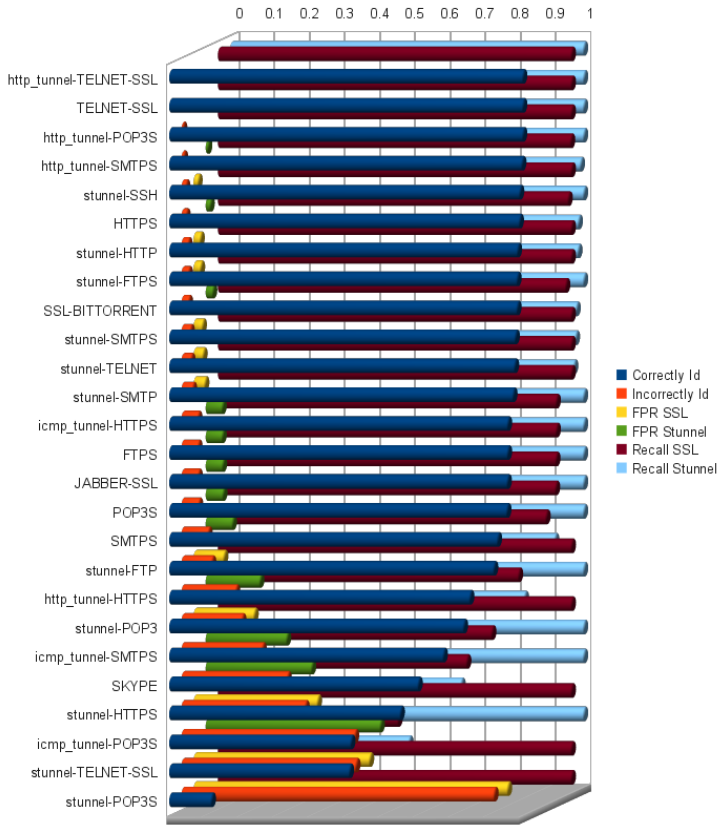

Figure 5.7: FPRA of AdaBoost C4.5 decision tree in SSL vs SSL-Tunnel run. The graphic illustrates the results from each performance metric employed for each application instance. They are sorted by the best classification performance on top to the worst on the bottom.

5.4 Non-SSL vs SSL-Tunneled

The original results using a subset of flows for this experiment showed a high degree of accuracy with low FPR, however, further FPRA investigation revealed that over half of the Stunnel application instances were being misclassified. Since this original experiment was performed on a subset of the flows with many application instances not supplying enough flow records to satisfy Algorithm 1, another run was initiated using all available class flows. These results, shown in Appendix C, with the additional Stunnel flows held a much higher classification rate for all algorithms with the exception of Naive Bayes. In the end, it was **RIPPER** with a *training set size of 12000* that held the best performance, as shown in Figure 5.10.

Despite the added Stunnel flow records, the FPR still remained quite high albeit

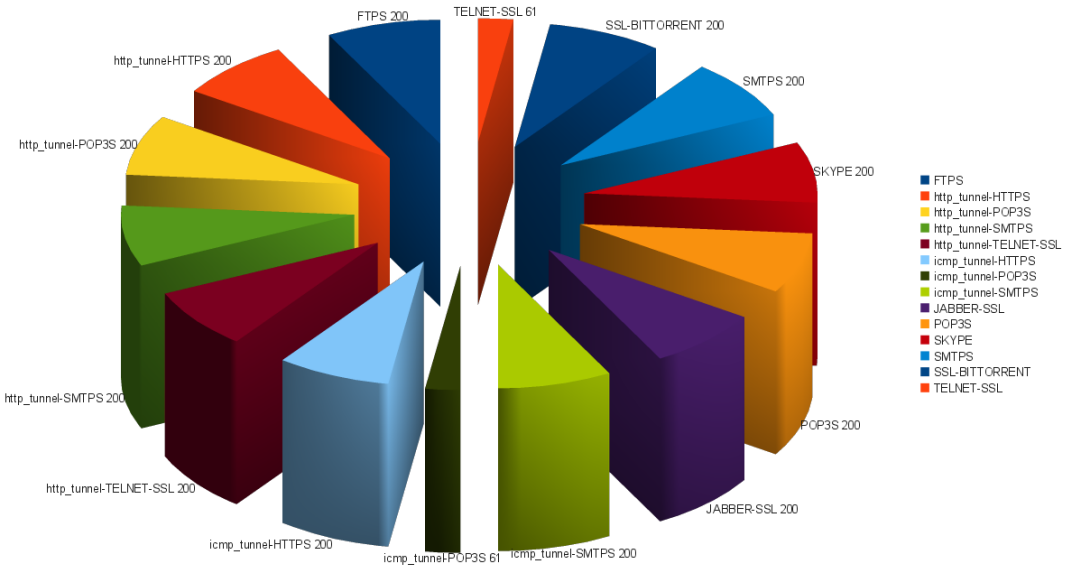

Figure 5.8: SSL class flow representation for training set size 6000 in SSL vs SSL-Tunnel run

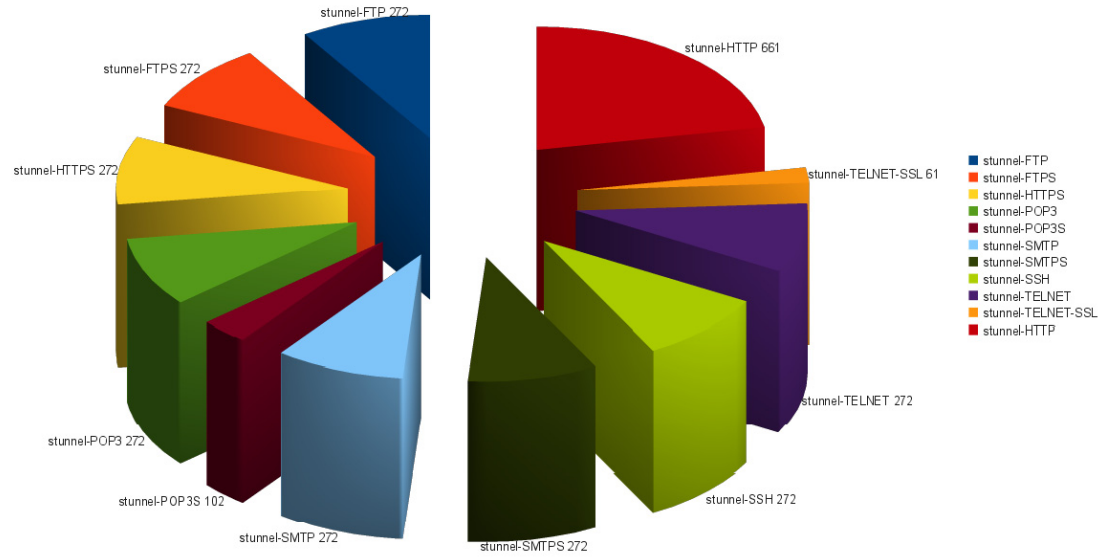

Figure 5.9: SSL-Tunnel class flow representation for training set size 6000 in SSL vs SSL-Tunnel run

with a more acceptable level of accuracy. The FPRA, shown in Figure 5.11, revealed high misclassification amongst the Stunnel application instances listed in Table 5.2. Since the purpose of the class run requires the proper detection of Stunnel traffic, further examination of the results was warranted.

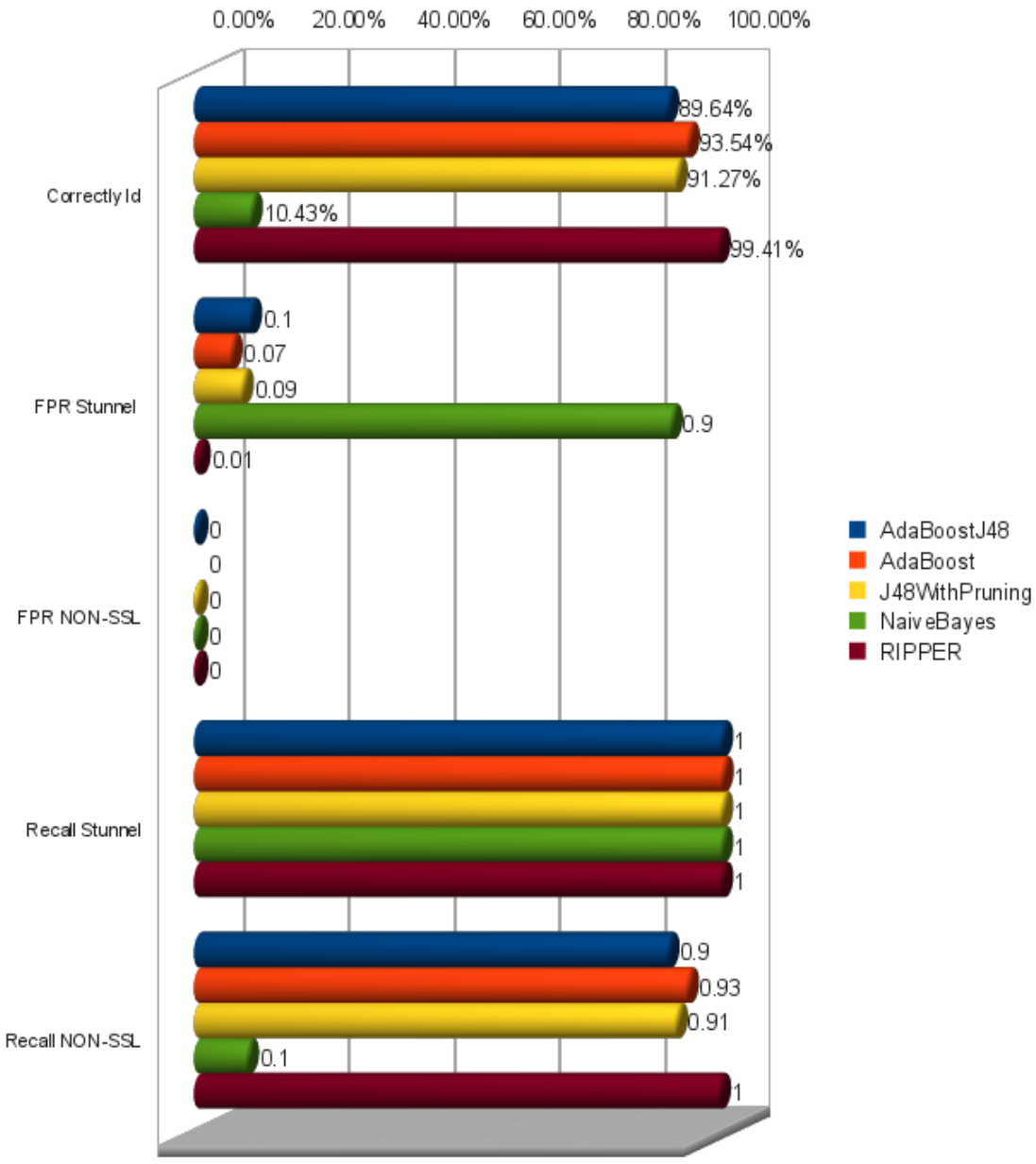

Figure 5.10: RIPPER performs best overall with a training set size of 12000 in Non-SSL vs SSL-Tunnel run

Analysis of the training set revealed that both *stunnel_POP3S* and *stunnel_TELNET-SSL* contained an extremely limited amount of flows causing inadequate representation in the training set. Additionally, *stunnel_POP3*, and *stunnel_SMTP* had enough flows but were misclassified due to high fluctuation in their flow records. Since the ML algorithms were unable to learn all the patterns within the application instance's flow records, they were not able to pick up on any discernible statistical pattern.

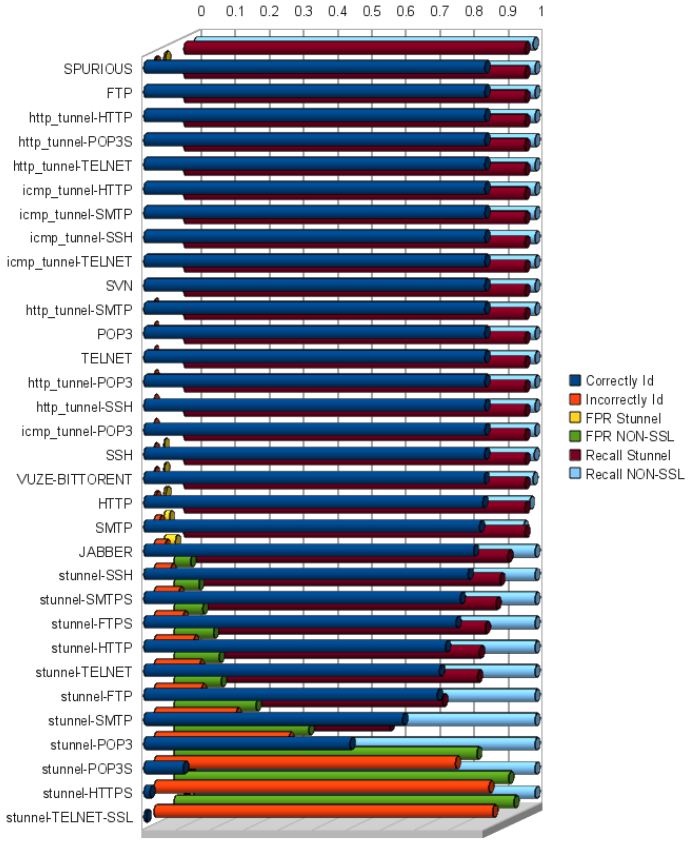

Figure 5.11: FPRA of RIPPER in Non-SSL vs SSL-Tunnel run

Application Instance	Incorrectly Id.	FPR STunnel	FPR Non-SSL
stunnel_TELNET-SSL	99.35%	0	1
stunnel_HTTPS	98.21%	0	0.98
stunnel_POP3S	88.41%	0	0.89
stunnel_POP3	39.55%	0	0.4
stunnel_SMTP	24.11%	0	0.24

Table 5.2: Application instances with lowest classification accuracy in Non-SSL vs SSL-Tunnel run

Similarly, the misclassification of *stunnel_HTTPS* can be attributed to memory restrictions on the training set size as the total behavior of the application instance was unable to be captured with the amount of flow records extracted. Further investigation revealed that the first few hundred flow samples consisted of a certain behavior type, whereas the remaining flow records differed greatly. While picking flow records at random may output better classification performance, the real solution is being able to capture all of the behavior of an application instance within the training set.

Due to memory restrictions imposed by WEKA requiring the entire training set to be loaded into memory, this is not possible and is thus left for future work.

As with the previous top performing ML algorithms, not all flow attributes were utilized. Only seven of the 22 flow attributes presented by NetMate were used by RIPPER to build a classifying ruleset. The following seven statements illustrate the rules built by RIPPER in order to classify the traffic.

1. ($min\_fpktl \geq 100$) and ($mean\_fpktl \leq 366$) and ($std\_biat \leq 32497904$) and ($min\_bpktl \leq 182$) and ($mean\_fpktl \geq 305$) $\Rightarrow$ class=SSL-Tunnel (2485.0/0.0)

2. ($min\_fpktl \geq 98$) and ($mean\_bpktl \geq 846$) and ($mean\_bpktl \leq 1062$) and ($mean\_bpktl \geq 1022$) $\Rightarrow$ class=SSL-Tunnel (1523.0/0.0)

3. ($total\_bvolume \leq 48$) and ($duration \geq 6936$) and ($mean\_bpktl \geq 846$) $\Rightarrow$ class=SSL-Tunnel (103.0/1.0)

4. ($duration \leq 0$) and ($total\_bvolume \leq 52$) and ($std\_biat \leq 6105665$) and ($std\_biat \geq 6105665$) $\Rightarrow$ class=SSL-Tunnel (61.0/0.0)

5. ($min\_fpktl \geq 98$) and ($mean\_fpktl \leq 574$) and ($min\_fpktl \geq 143$) and ($mean\_fpktl \geq 334$) and ($min\_fpktl \leq 244$) $\Rightarrow$ class=SSL-Tunnel (419.0/0.0)

6. ($total\_bvolume \leq 48$) and ($std\_biat \leq 2531334$) and ($min\_fpktl \geq 81$) $\Rightarrow$ class=SSL-Tunnel (31.0/0.0)

7. $\Rightarrow$ class=NON-SSL (7378.0/1379.0)

RIPPER's ability to achieve almost 100% accurate classification with 10 fold cross-validation and a very low misclassification error amongst the rule sets is paramount to its overall performance results. Immediately, the algorithm starts by focusing primarily on the bi-directional packet length flow statistics and inter-arrival times. By creating this restrictive window, it is able to classify a good portion of the flow records quickly and accurately. The use of the forward packet length statistics can be attributed to the added overhead of encryption while packets are being sent to the proxy. Furthermore, the inter-arrival times will differ for the packets as they have an extra hop to go through the proxy server. Finally, during the return trip the packets must maintain a minimum size to support the SSL encryption overhead.

The next set of rules manages to classify traffic using only bi-directional packet length flow statistics. The usage of the mean backwards packet lengths illustrates the algorithm's ability to recognize return trip packet connections further supporting the claims in the initial rule analysis.

Rule three takes into account two new flow attributes, the duration and backwards volume responses from the server with only one misclassification error. The duration of the flow plays an important role as Stunnel traffic would have a longer flow time due to the extra proxy hop. Rule four further adds to the restrictions of rule three by making use of the inter-arrival times of the packets. Rule five mimics rule two but concentrates exclusively on the client requests bi-directional packet length flow statistics. By defining such a restrictive window, it would seem RIPPER is isolating the Stunneled packets by their extra padding required. Rule six uses the backwards flow volume, backwards inter-arrival time, and minimum forward packet length to quarantine Stunnel flows by applying the same principals discussed above. Finally, any remaining flows not recognized by this *if-elseif-else* structure are classified as Non-SSL. This obviously also represents the point where most of the false positives occur in classification.

A breakdown of the application instances composing each class in the training set is presented in Figure 5.12 and Figure 5.13.

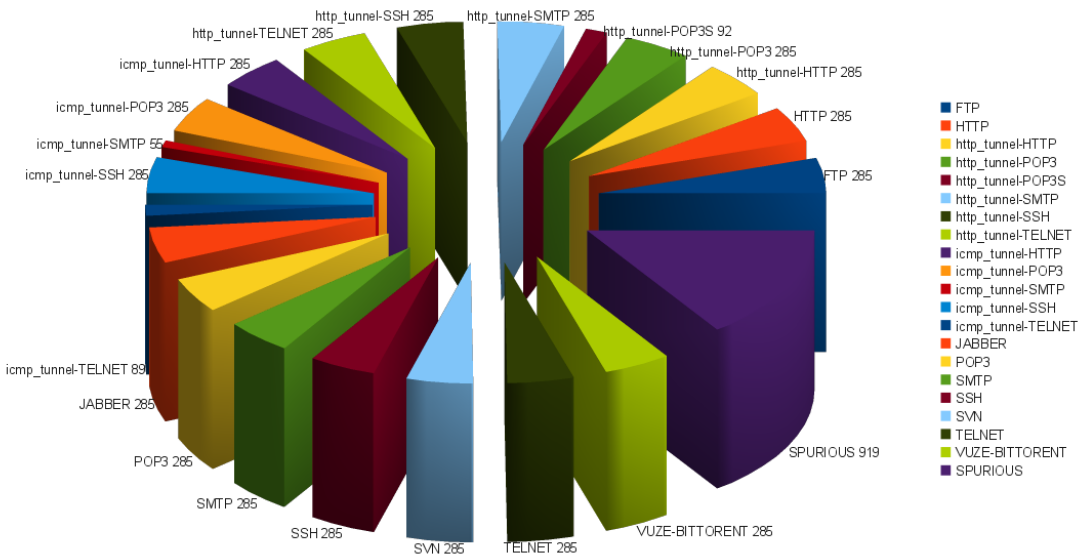

Figure 5.12: Non-SSL class flow representation for training set size 12000 in Non-SSL vs SSL-Tunnel run

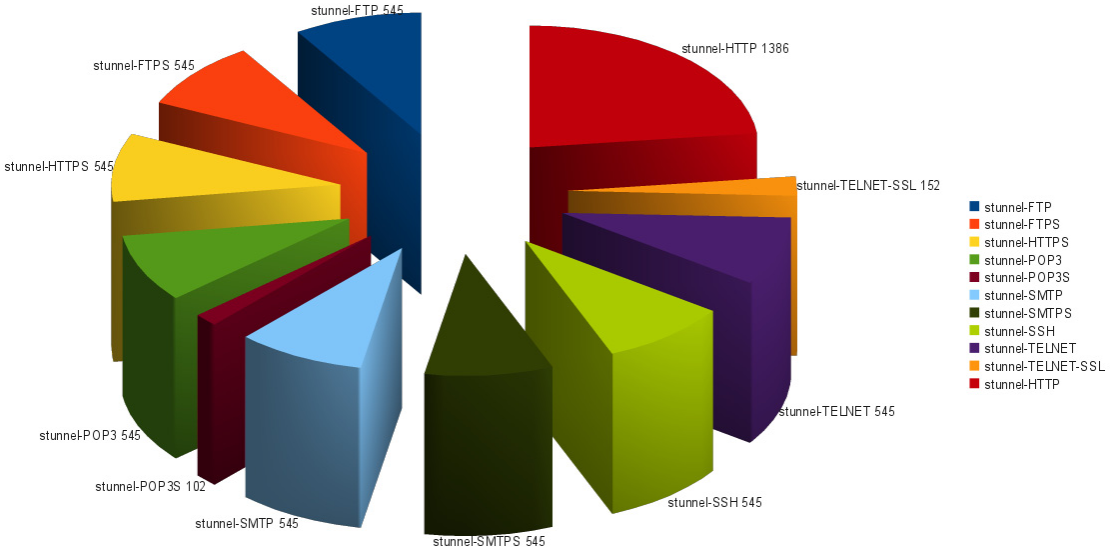

Figure 5.13: SSL-Tunnel class flow representation for training set size 12000 in Non-SSL vs SSL-Tunnel run

5.5 NIMS vs NIMSv2

In order to test the robustness of the classifiers, a different dataset called NIMS generated by Alshammari et al. [1, 4, 3, 2] was introduced. Since its purpose was judging the effectiveness of ML algorithms on other forms of encrypted traffic, little SSL traffic was included with more emphasis placed on SSH and Skype. Also, there were no tunneling experiments done in the NIMS dataset.

For labeling purposes, the flow records from the original NIMS dataset had to match those tested in this thesis. As such, only the subset of the application instances shown in Table 5.3 could be included for testing. A few assumptions had to be made regarding certain application instances in regards to labeling them as either SSL or Non-SSL. For instance, there is no way of knowing whether or not SMTP communication on port 25 or 587 occurred in an encrypted or unencrypted fashion. Nevertheless, this only serves to increase the entropy of the dataset underestimating the results of the classifiers [52].

Application Instance	Class label
SSH	NON-SSL
HTTP	NON-SSL
HTTPS	SSL
POP3	NON-SSL
POP3S	SSL
SMTP	NON-SSL
FTP	NON-SSL
SVN	NON-SSL
JABBER-SSL	SSL
JABBER	NON-SSL
TELNET	NON-SSL
SPURIOUS	NON-SSL

Table 5.3: Matching application instances that can be labeled and tested against in the NIMS vs NIMSv2 run

5.5.1 Robustness Analysis for SSL vs Non-SSL Classification

In an effort to determine how well the classifier would perform without any prior knowledge of the different dataset, the best performing model from the *SSL vs Non-SSL* run in Section 5.1 was chosen. This AdaBoost model with a training set size 500000 was tested against the original NIMS dataset on the application instances listed in Table 5.3. The results shown in Table 5.4 are quite promising despite the low number of available SSL application instances in the original NIMS dataset.

Correctly Id.	FPR SSL	FPR Non-SSL	Recall SSL	Recall Non-SSL
94.72%	0.05	0	1	0.947

Table 5.4: Results of the AdaBoost training set size 500000 model against the original NIMS traffic

5.5.2 Modifying the training data set

In an effort to further optimize the performance of AdaBoost, 20% of the original NIMS flow records were injected into the training set while maintaining the class balance. Specifically, 20% of the flow records required from any given application instance for training were supplied by the original NIMS dataset. By modifying the altered training script to implement this change, a new fine-tuned training set was

generated and run against the original NIMS dataset giving the slightly improved results in Table 5.5.

Correctly Id.	FPR SSL	FPR Non-SSL	Recall SSL	Recall Non-SSL
95.79%	0.04	0.01	0.986	0.96

Table 5.5: Results of the new AdaBoost training set size 500000 model with 20% of the flows from the original NIMS tested against the dataset from this thesis

As a final experiment, the new AdaBoost model with a training set size of 500000 and 20% of the flows from NIMS was then tested against the dataset generated by this thesis to see if there was any fluctuation in performance. The similar results obtained are found in Table 5.6.

Correctly Id.	FPR SSL	FPR Non-SSL	Recall SSL	Recall Non-SSL
94.80%	0.052	0	1	0.948

Table 5.6: Results of the new AdaBoost training set size 500000 model with 20% of the flows from the original NIMS tested against the dataset from this thesis

Combining the traffic from both datasets and following a similar permutation of testing was initially considered but the difference in size between the datasets would completely bias the results and was therefore not pursued. From these tests, it is safe to say a high degree of generalization exists solidifying the robustness of the AdaBoost model.

Chapter 6

Comparing the Proposed Approach to Wireshark

In an attempt to compare the methodology and results achieved in this thesis against others in the same area, this chapter presents a comparison using the Wireshark [1] traffic analysis tool. Due to Wireshark's high popularity within the networking field, this comparison was done to demonstrate the inadequacies of traffic classification tools available to network stakeholders. Since Wireshark uses packets and not flows, a discussion on the steps taken to extract those packets, which created the statistical Netmate flow attributes in the training set, is first presented. Several examples on Wireshark's poor performance on correctly identifying packet traces are then illustrated.

6.1 Wireshark for Comparison

Wireshark is a popular cross-platform open source traffic analysis tool used by network administrators [11]. It can capture or replay traffic, supports Deep Packet Inspection (DPI), and provides several useful tools for diagnosing and filtering packets. Wireshark's command line cousin, TShark, was discussed earlier in Section 3.2.

Since Wireshark uses packets and not NetMate flows, a direct comparison of the same packets NetMate used as input for the training sets is impractical. Nevertheless, it is possible to look at the same packets, which were included in the initial flow transformations given the temporal process followed in the methodology. The process began by filtering the date metadata included in the TShark capture files from each machine. Since the packets were changed into flows by NetMate in a first-capture-first-transformed manner, it is known that those first few packets appeared as the initial flow records in the CSV file. In turn, those flows were included as the initial records in the training sets for each application instance. The resulting process required loading the initial TShark captured files from each machine into Wireshark since

[1] http://www.wireshark.org/

they were stored in a compatible packet capture (pcap) binary format. Then using Wireshark's powerful filtering abilities, they were parsed using IP address, protocol, and port numbers. The remaining traffic represented a packet-based view of a specific application instance.

Wireshark's default labeling process uses the IANA official port number registry [2] to classify traffic. As such, it was able to correctly identify application instances that were in compliance with the IANA ruling. For instance, Wireshark had no problem detecting and subsequently classifying the application instance JABBER running on port 5222, as shown in Figure 6.1. However, any non-compliant application instances

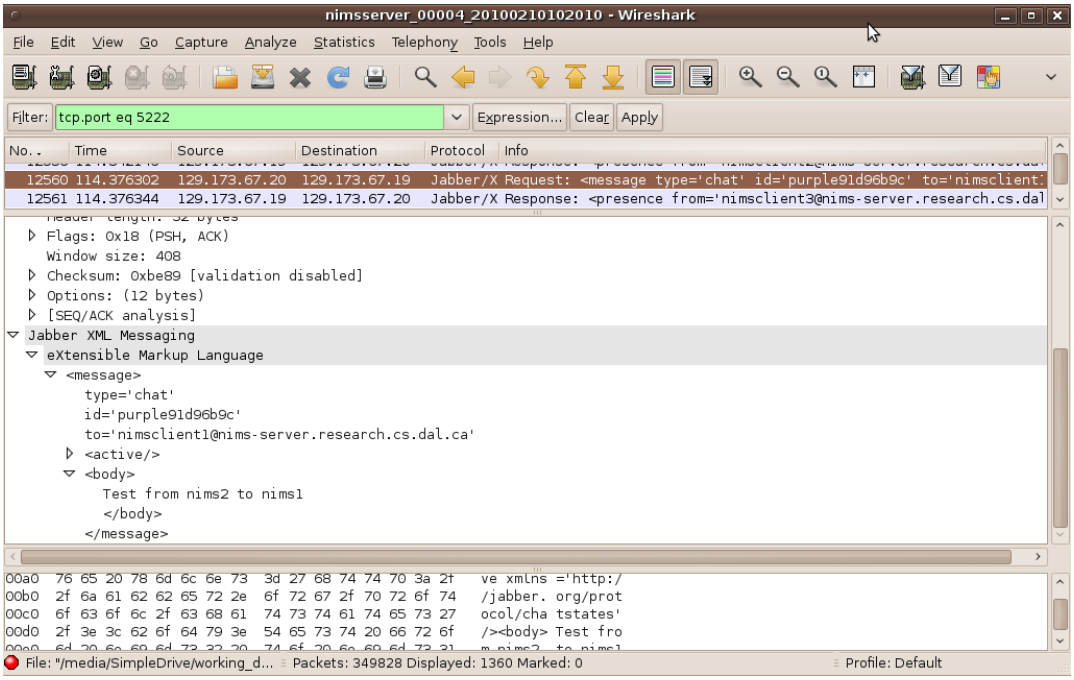

Figure 6.1: Wireshark view of correctly classified Jabber XMPP packet on port 5222

were misclassified as their IANA registered counterparts. For example, the application instance JABBER-SSL was classified as the *HP Virtual Machine Group Management* (*hpvirtgrp*) protocol, as illustrated in Figure 6.2. Further investigation of the packet data showed a hidden SSL certificate. The developers of Wireshark knew that it would sometimes mislabel traffic and provided a feature to decode packets as a specific protocol. Applying an SSL filter in this function on the same packet revealed it was actually a TLS handshake, shown in Figure 6.3. Unfortunately, the prior knowledge of

[2] http://www.iana.org/assignments/port-numbers

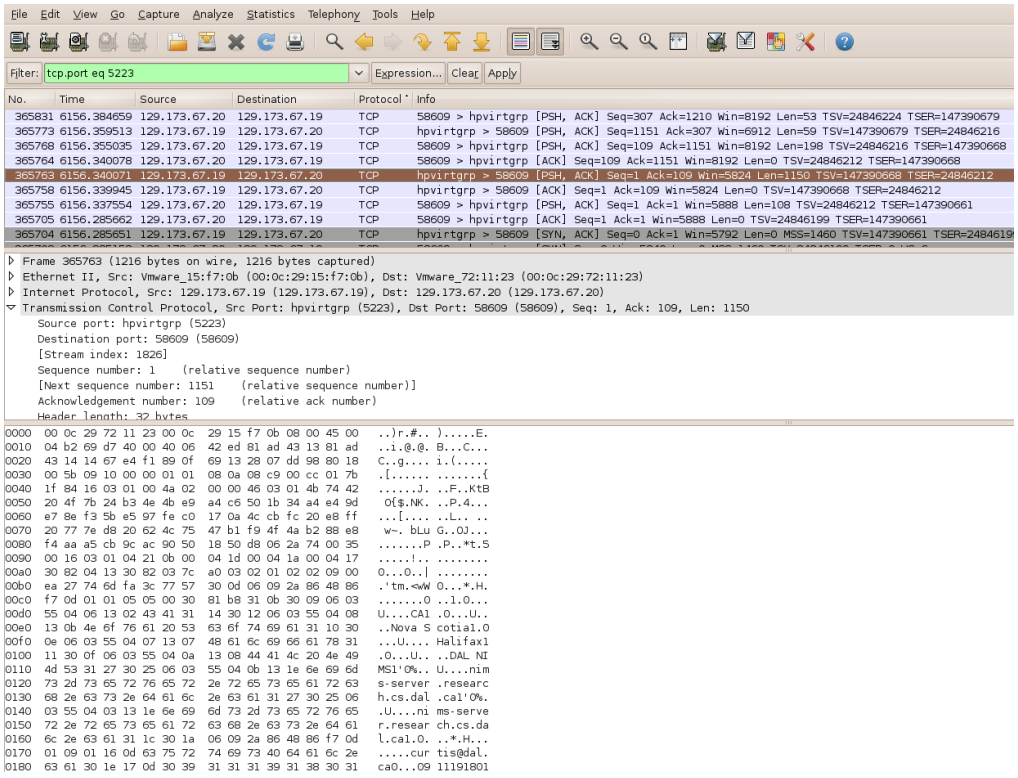

Figure 6.2: Wireshark view of mislabeled JabberSSL packet on port 5223 with a hidden SSL certificate

choosing which protocol to decode the packet as is not available in the real world. This causes misclassification without manually parsing the packet data field. All tunneling instances were affected by this same misclassification process. For instance, Stunnel-HTTP traffic from the client had packets labeled falsely as *mailbox* and correctly as an aggregate SSL class, shown in Figure 6.4. ICMP tunneling presented a unique characteristic as no ports could be used to further filter the packets. As such, packet payloads contained various pieces of data including the HTTP response shown in Figure 6.5

Overall the performance of Wireshark for traffic classification without using prior knowledge to decode packets was mediocre at best. Tunneled traffic suffered the most out of all the application instances tested and those using custom ports were also mislabeled. For those SSL/TLS encrypted packets properly labeled, there was no further knowledge presented about which application instance spawned the connection. As such, these SSL/TLS results forced an aggregate high level traffic representation but

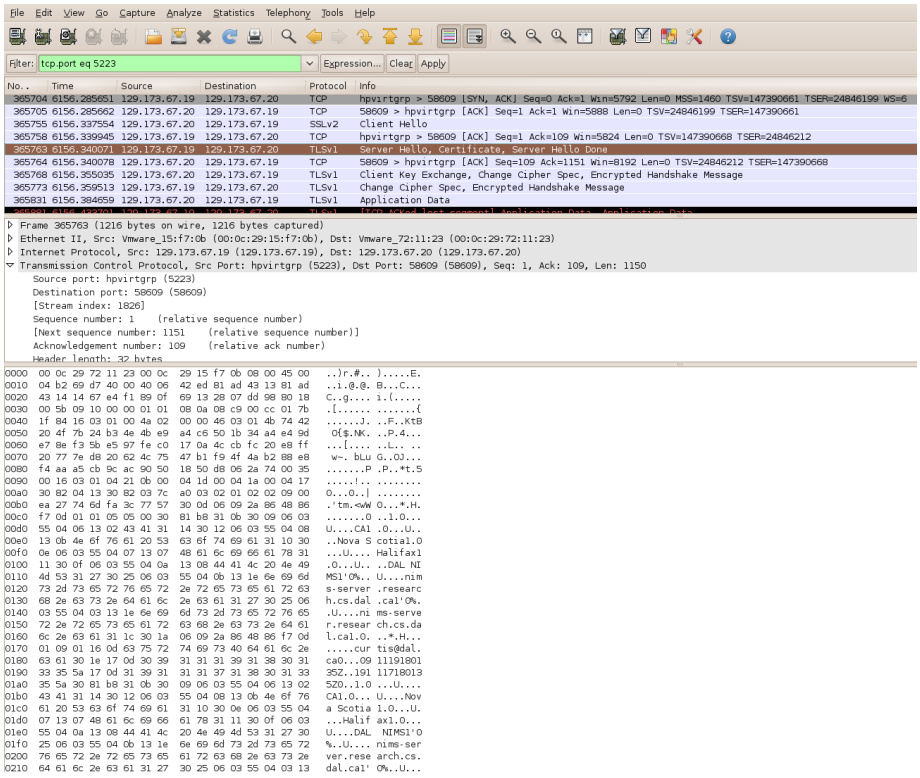

Figure 6.3: Wireshark view of JabberSSL packet after SSL decoding was applied showing TLS handshake

do not convey additional information to a network stakeholder. Nevertheless, this presented a good benchmark showing how well the ML algorithms were able to classify using a few flow attributes rather then entire packets.

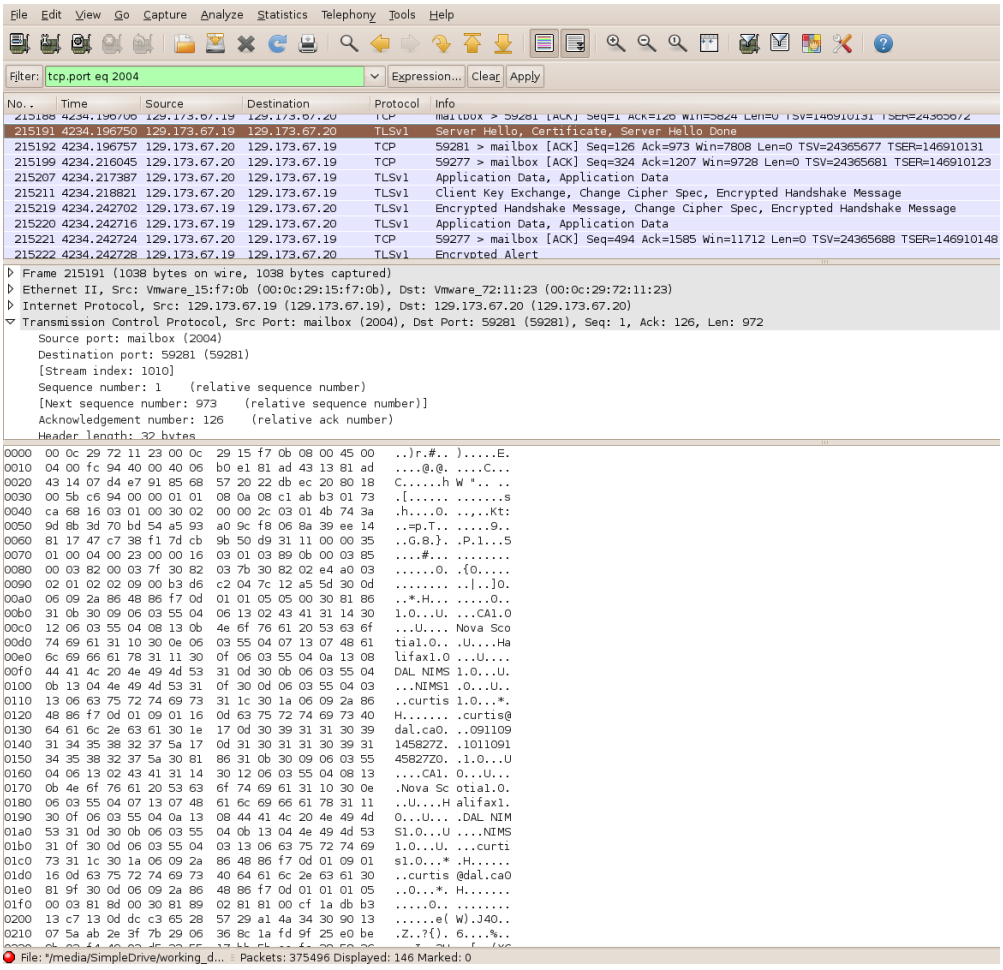

Figure 6.4: Wireshark view of SSL-Tunneled HTTP packets showing TCP and TLS classification

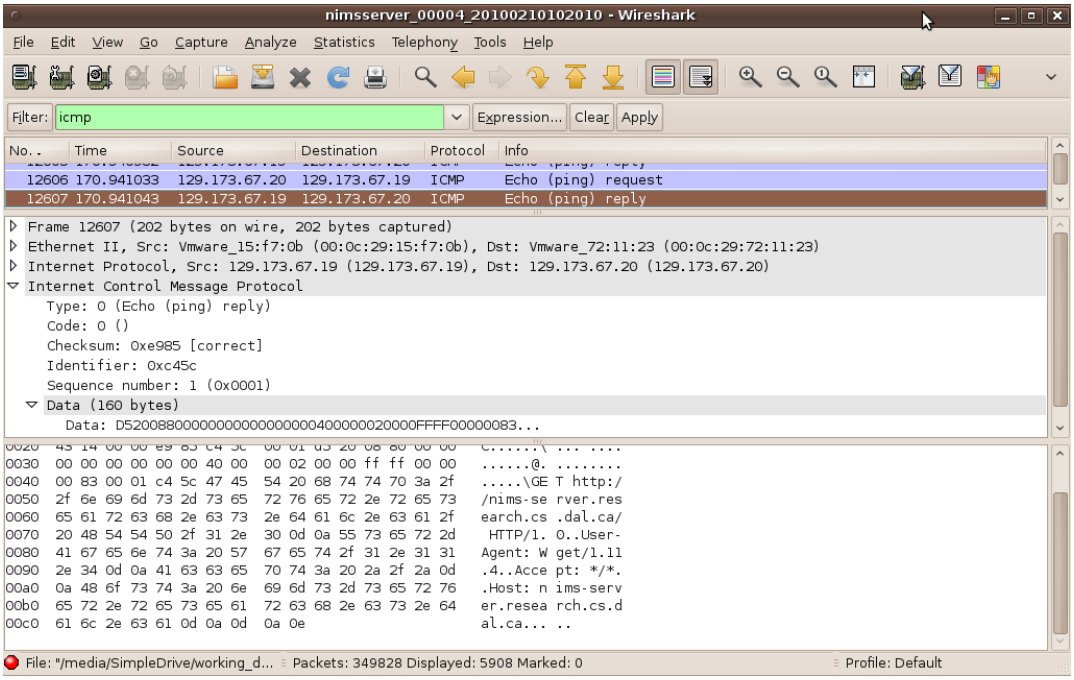

Figure 6.5: Wireshark view of ICMP-Tunneled HTTP packets showing HTTP response code

Chapter 7

Conclusion and Future Work

In conclusion, the investigation into classifying applications encrypted by SSL achieved promising results using flow-based statistics and ML algorithms. This was accomplished without employing IP addresses, port numbers, or packet payloads. The lack of traffic classification literature focusing on SSL will become a serious oversight as more and more applications encrypt their transmissions. As a result, network stakeholders will no longer have the means to implement services, such as traffic shaping or ensuring QoS. It is shown that the approach followed in this thesis presents promising preliminary results, which are robust enough to be scalable to different networks.

In this thesis, the generated training dataset represented real network traffic without imposing restrictions on SSL encryption algorithms, ensured the *ground truth* during the labeling process, and included the use of tunnels to increase the overall entropy of the dataset. The open source nature of almost all the application instances employed, as well as the WEKA ML algorithm implementation, further promotes the reproducibility of the results. Training algorithms ensured a balanced training set to prevent bias amongst the human-interpretable ML algorithm results. Additional investigation on the sizes of the training set helped to further optimize some of the results based off the feedback from individual application instance FPRA. The comparison of the results from the multiple class runs against recent literature as well as a popular traffic analysis tool further solidifies the methodology taken.

For the *SSL vs Non-SSL* class run, AdaBoost proved to have the best classification performance with a 96% classification accuracy, 4% SSL FPR, and 1.5% Non-SSL FPR. In the case of the native *SSL vs SSL-Tunnel* run, a modified version of AdaBoost using C4.5 decision trees instead of decision stumps performed the best with a 98% classification accuracy, 0.6% SSL FPR, and 1.1% SSL-Tunnel FPR. Finally, the ML algorithm best able to distinguish the *Non-SSL vs SSL-Tunnel* class run was RIPPER achieving a 99% classification accuracy, 0.5% SSL-Tunnel FPR, and 0.1% Non-SSL

FPR. After averaging the best performance of each ML algorithm across all three class runs, AdaBoost maintained the highest overall performance, as illustrated in Table 7.1. While the training set sizes varied amongst all the runs, it can be seen that capturing the correct amount of behavior is paramount for the ML algorithm to acquire acceptable results. Without adequate representation from an application instance, they will be unable to perform well during testing.

ML Algorithm	**Correctly Id. Average**
AdaBoost	*95.35%*
AdaBoostJ48	93.53%
RIPPER	92.94%
J48WithPruning	92.37%
NaiveBayes	59.31%

Table 7.1: Overall performance of the ML algorithms across all the class runs

Comparing the results found in this thesis to the Wireshark traffic analysis tool confirmed the incompetencies of DPI methods and inability to rely on port numbers for classification. Further investigation using this packet based approach showed high misclassification amongst the majority of the application instances not using standard IANA port numbers.

There are several areas where future work is warranted as this work was only investigative in nature. The unexpected closure of TARA prevented the capture of additional application instances whose overall flow count was much lower then others. The use of a different training set algorithm to better capture the behavior of an application instance would be valuable for increasing the classification performance of the ML algorithms. This might also lead to reduced training set sizes required to achieve similar or better results. Exploring alternative ML algorithms is another route, which should be investigated while attempting to maintain the same level of human interpretation. Furthermore with the changes implemented between IPv4 and IPv6, it would be interesting to see how much (if at all) the results achieved in this thesis may be affected. The difference in packet construction would require the NetMate flow generator to be modified to correctly detect the new flows. Additional packet data found in IPv6 may also lead to new flow statistical attributes from NetMate. Alternative options for flow timeouts could help improve the number of NetMate flows detected as several application instances failed to adequately represent themselves in

flow records after data collection was complete.Further investigation on parameter sensitivity and the affect they have on the ML algorithms may offer more explanation as to why the classifiers acted the way they did. Finally, a more thorough comparison of the findings in this thesis against others in the same research area should be investigated as well.

Bibliography

[1] R. Alshammari, A.N. Zincir-Heywood, and A.A. Farrag. Performance comparison of four rule sets: An example for encrypted traffic classification. *Privacy, Security, Trust and the Management of e-Business, 2009. CONGRESS '09. World Congress on*, pages 21 –28, 2009.

[2] Riyad Alshammari and A. Nur Zincir-Heywood. Generalization of signatures for ssh encrypted traffic identification. *Proceedings - IEEE Symposium on Computational Intelligence in Cyber Security*. pages 174 - 174, 2009.

[3] Riyad Alshammari and A. Nur Zincir-Heywood. Investigating two different approaches for encrypted traffic classification. *Proceedings - 6th Annual Conference on Privacy, Security and Trust*, pages 156 – 166, 2008.

[4] Riyad Alshammari and A. Nur Zincir-Heywood. Machine learning based encrypted traffic classification: Identifying ssh and skype. *IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pages 8 – 8, 2009.

[5] Daniel J. Barrett and Richard E. Silverman. *SSH, The Secure Shell: The Definitive Guide*. ISBN: 978-0-596-00011-0, O'Reilly & Associates, Inc., 2001.

[6] S. A. Baset and H. G. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications*, pages 1 –11, 2006.

[7] Laurent Bernaille and Renata Teixeira. Early recognition of encrypted applications. *Passive and Active Network Measurement*, 4427:165–175, 2007.

[8] Bit-torrent Bram Cohen. http://jonas.nitro.dk/bittorrent/bittorrent-rfc.html, Accessed June 2010.

[9] Arthur Callado, Judith Kelner, Djamel Sadok, Carlos Alberto Kamienski, and Stnio Fernandes. Better network traffic identification through the independent combination of techniques. *Journal of Network and Computer Applications*, 33(4):433 – 446, 2010.

[10] TLSv1.2 IETF Charter. http://www.ietf.org/dyn/wg/charter/tls-charter.html, Accessed June 2010.

[11] Gerald Combs. Wireshark. http://www.wireshark.org, Accessed May 2010.

[12] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Detecting http tunnels with statistical mechanisms. *IEEE International Conference on Communications*, pages 6162 – 6168, 2007.

[13] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic classification through simple statistical fingerprinting. *SIGCOMM Comput. Commun. Rev.*, 37(1):5–16, 2007.

[14] Alberto Dainotti, Walter De Donato, Antonio Pescape, and Pierluigi Salvo Rossi. Classification of network traffic via packet-level hidden markov models. *GLOBECOM - IEEE Global Telecommunications Conference*, pages 2138 – 2142, 2008.

[15] A. Dupay, S. Sengupta, O. Wolfson, and Y. Yemini. Netmate: A network management environment. *Network, IEEE*, 5(2):35 –40, 43, mar. 1991.

[16] M. Dusi, M. Crotti, F. Gringoli, and L. Salgarelli. Tunnel hunter: Detecting application-layer tunnels with statistical fingerprinting. *Computer Networks*, 53(1):81 – 97, 2009.

[17] Maurizio Dusi, Manuel Crotti, Francesco Gringoli, and Luca Salgarelli. Detection of encrypted tunnels across network boundaries. *IEEE International Conference on Communications*, pages 1738 – 1744, 2008.

[18] Jeffrey Erman, Anirban Mahanti, Martin Arlitt, Ira Cohen, and Carey Williamson. Offline/realtime traffic classification using semi-supervised learning. *Performance Evaluation*, 64(9-12):1194 – 1213, 2007.

[19] Alice Este, Francesco Gringoli, and Luca Salgarelli. Support vector machines for tcp traffic classification. *Computer Networks*, 53(14):2476 – 2490, 2009.

[20] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, 2009.

[21] DTLS RFC IETF. http://tools.ietf.org/html/rfc4347, Accessed June 2010.

[22] FTP: IETF. http://tools.ietf.org/html/rfc959, Accessed June 2010.

[23] HTTP: IETF. http://tools.ietf.org/html/rfc2616, Accessed June 2010.

[24] HTTPS: IETF. http://www.ietf.org/rfc/rfc2818.txt, Accessed June 2010.

[25] IMAP: IETF. http://www.ietf.org/rfc/rfc2060.txt, Accessed June 2010.

[26] POP3: IETF. http://tools.ietf.org/html/rfc1939, Accessed June 2010.

[27] POP3S: IETF. http://tools.ietf.org/html/rfc2595, Accessed June 2010.

[28] SASL: IETF. http://tools.ietf.org/html/rfc2222, Accessed June 2010.

[29] SMTPS: IETF. http://www.rfc-editor.org/rfc/rfc2487.txt, Accessed June 2010.

[30] SSH: IETF. http://www.ietf.org/rfc/rfc4252.txt, Accessed June 2010.

[31] TELNET: IETF. http://tools.ietf.org/html/rfc854, Accessed June 2010.

[32] TLS: IETF. http://www.ietf.org/rfc/rfc2246.txt, Accessed June 2010.

[33] TLS CipherSuite RFC IETF. http://tools.ietf.org/html/rfc5246, Accessed June 2010.

[34] XMPP: IETF. http://www.ietf.org/rfc/rfc3920.txt, Accessed June 2010.

[35] Li Jun, Zhang Shunyi, Lu Yanqing, and Zhang Zailong. Internet traffic classification using machine learning. *IEEE Communications and Networking in China, 2007. CHINACOM '07. Second International Conference on (978-1-4244-1009-5)*, pages 239–243, 2007.

[36] Abuagla Babiker Mohd and Dr. Sulaiman bin Mohd Nor. Towards a flow-based internet traffic classification for bandwidth optimization. *International Journal of Computer Science and Security*, 3:146–153, 2009.

[37] Andrew W. Moore and Konstantina Papagiannaki. Toward the accurate identification of network applications. *Lecture Notes in Computer Science*, 3431:41 – 54, 2005.

[38] T.T.T. Nguyen and G. Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys Tutorials, IEEE*, 10(4):56 –76, fourth 2008.

[39] David M. Nicol and Nabil Schear. Models of privacy preserving traffic tunneling. *Simulation*, 85(9):589 – 607, 2009.

[40] Junghun Park, Hsiao-Rong Tyan, and C.-C. Jay Kuo. Internet traffic classification for scalable QoS provision. *IEEE International Conference on Multimedia and Expo*, pages 1221–1224, 2006.

[41] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, pages 2435–2463, 1999.

[42] Luo Qing and Lin Yaping. Analysis and comparison of several algorithms in ssl/tls handshake protocol. *ITCS '09: Proceedings of the 2009 International Conference on Information Technology and Computer Science*, pages 613–617, 2009.

[43] Martin Roesch and Stanford Telecommunications. Snort — lightweight intrusion detection for networks. *USENIX - Proceedings of LISA '99: 13th Systems Administration Conference*, pages 229–238, 1999.

[44] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. ISBN: 0137903952, Pearson Education, 2003.

[45] Nabil Schear and David M. Nicol. Performance analysis of real traffic carried with encrypted cover flows. *Workshop on Principles of Advanced and Distributed Simulation*, pages 80 – 87, 2008.

[46] Murat Soysal and Ece Guran Schmidt. Machine learning algorithms for accurate flow-based network traffic classification: Evaluation and comparison. *Performance Evaluation*, 67(6):451 – 467, 2010.

[47] Daniel Stdle. Ptunnel icmp tunnel. http://www.cs.uit.no/ daniels/PingTunnel/, Accessed May 2010.

[48] Michal Trojnara. Stunnel: Ssl tunnel. www.stunnel.org. Accessed June 2010.

[49] Jon Viega, Pravir Chandra, and Matt Messier. *Network Security with Openssl*. ISBN: 059600270X, O'Reilly & Associates, Inc., 2002.

[50] Nigel Williams, Sebastian Z, and Grenville Armitage. A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. *Computer Communication Review*, 30, 2006.

[51] Nigel Williams, Sebastian Zander, and Grenville Armitage. A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. *SIGCOMM Comput. Commun. Rev.*, 36(5):5–16, 2006.

[52] Charles V. Wright, Fabian Monrose, and Gerald M. Masson. On inferring application protocol behaviors in encrypted network traffic. *Journal of Machine Learning Research*, 7(12):2745 – 2769, 2006.

[53] Akira Yamada, Yutaka Miyake, Keisuke Takemori, Ahren Studer, and Adrian Perrig. Intrusion detection for encrypted web accesses. *21st International Conference on Advanced Information Networking and Applications Workshops/Symposia*, 2:569 – 576, 2007.

[54] Caihong Yang and Benxiong Huang. Traffic classification using an improved clustering algorithm. *2008 International Conference on Communications, Circuits and Systems Proceedings*, pages 515 – 518, 2008.

[55] Ruixi Yuan, Zhu Li, Xiaohong Guan, and Li Xu. An svm-based machine learning method for accurate internet traffic classification. *Information Systems Frontiers*, Volume 12:149 – 156, 2010.

[56] Li Zhao, Ravi Iyer, Srihari Makineni, and Laxmi Bhuyan. Anatomy and performance of ssl processing. *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 197 – 206, 2005.

Appendix A

SSL vs Non-SSL runs

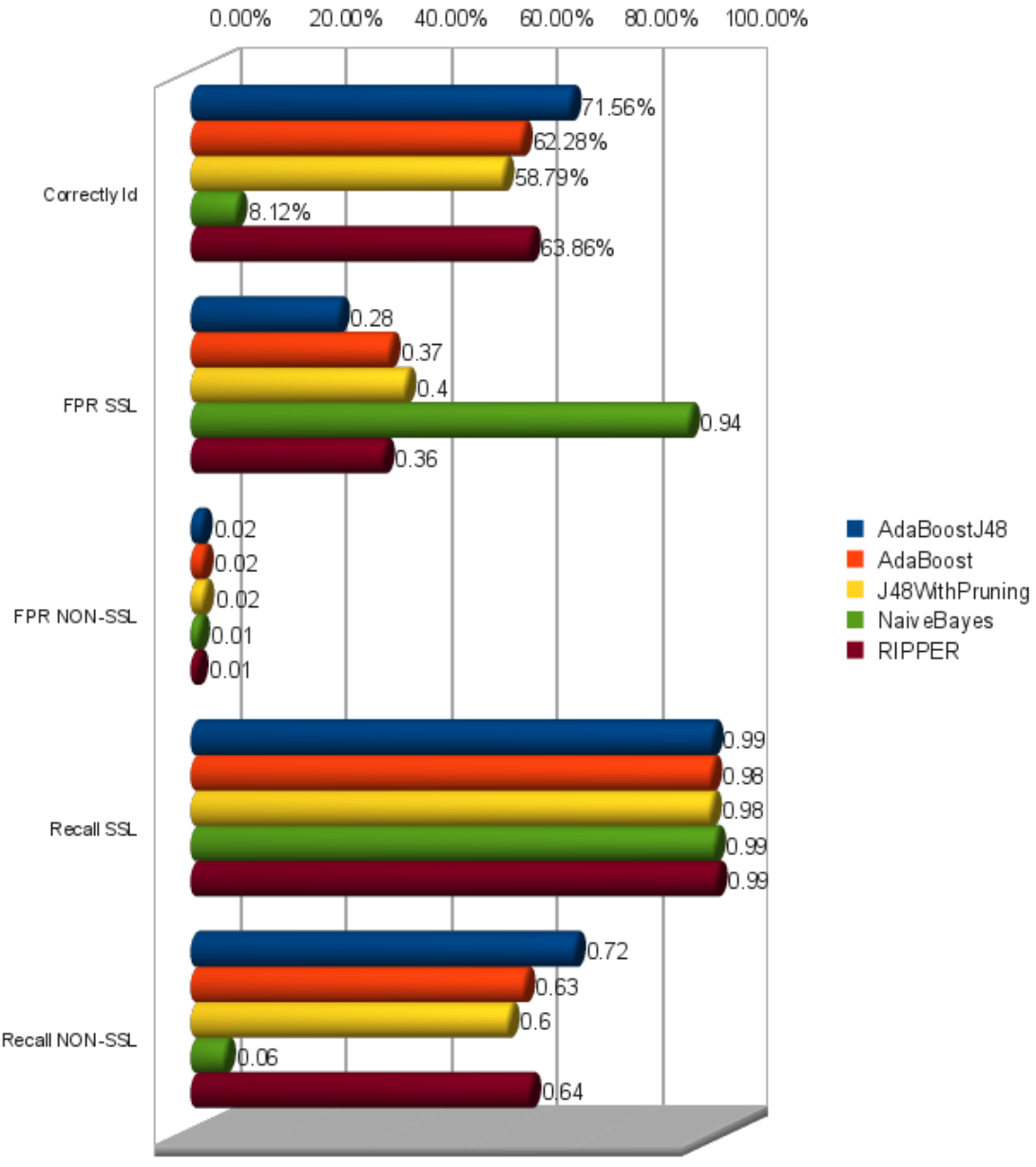

Figure A.1: Training set size 6000 results from each performance metric employed for each ML algorithm

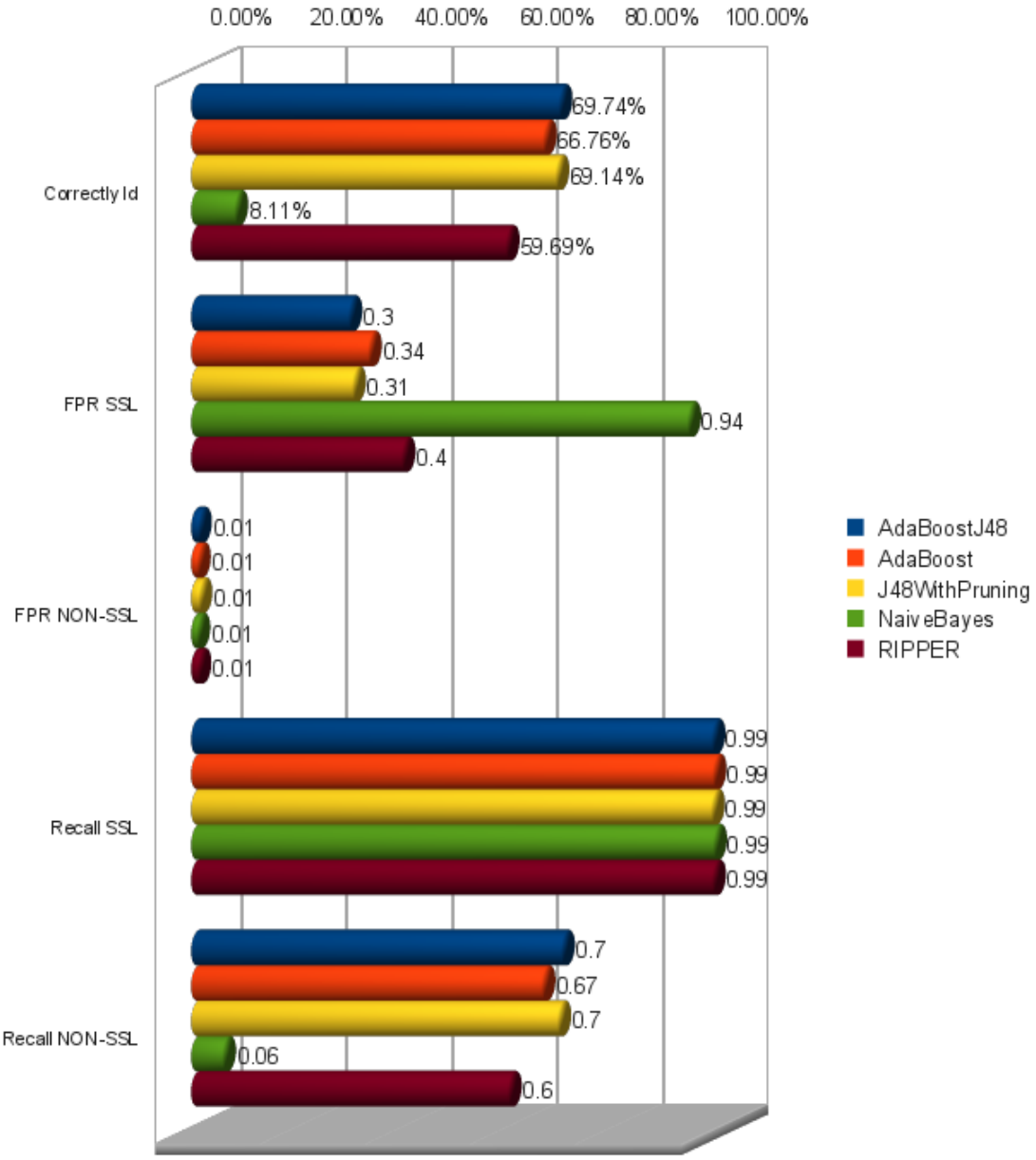

Figure A.2: Training set size 9000 results from each performance metric employed for each ML algorithm

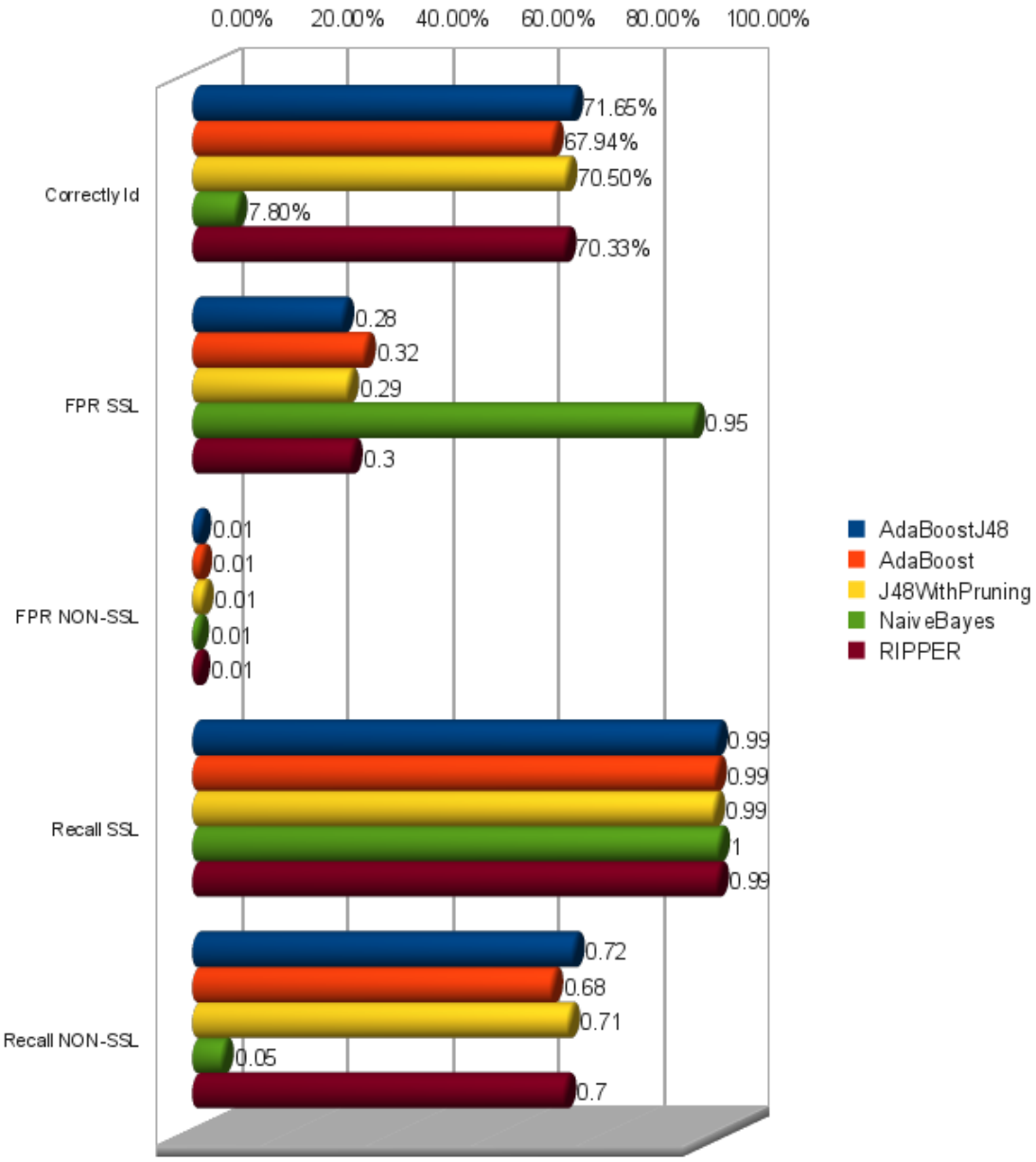

Figure A.3: Training set size 12000 results from each performance metric employed for each ML algorithm

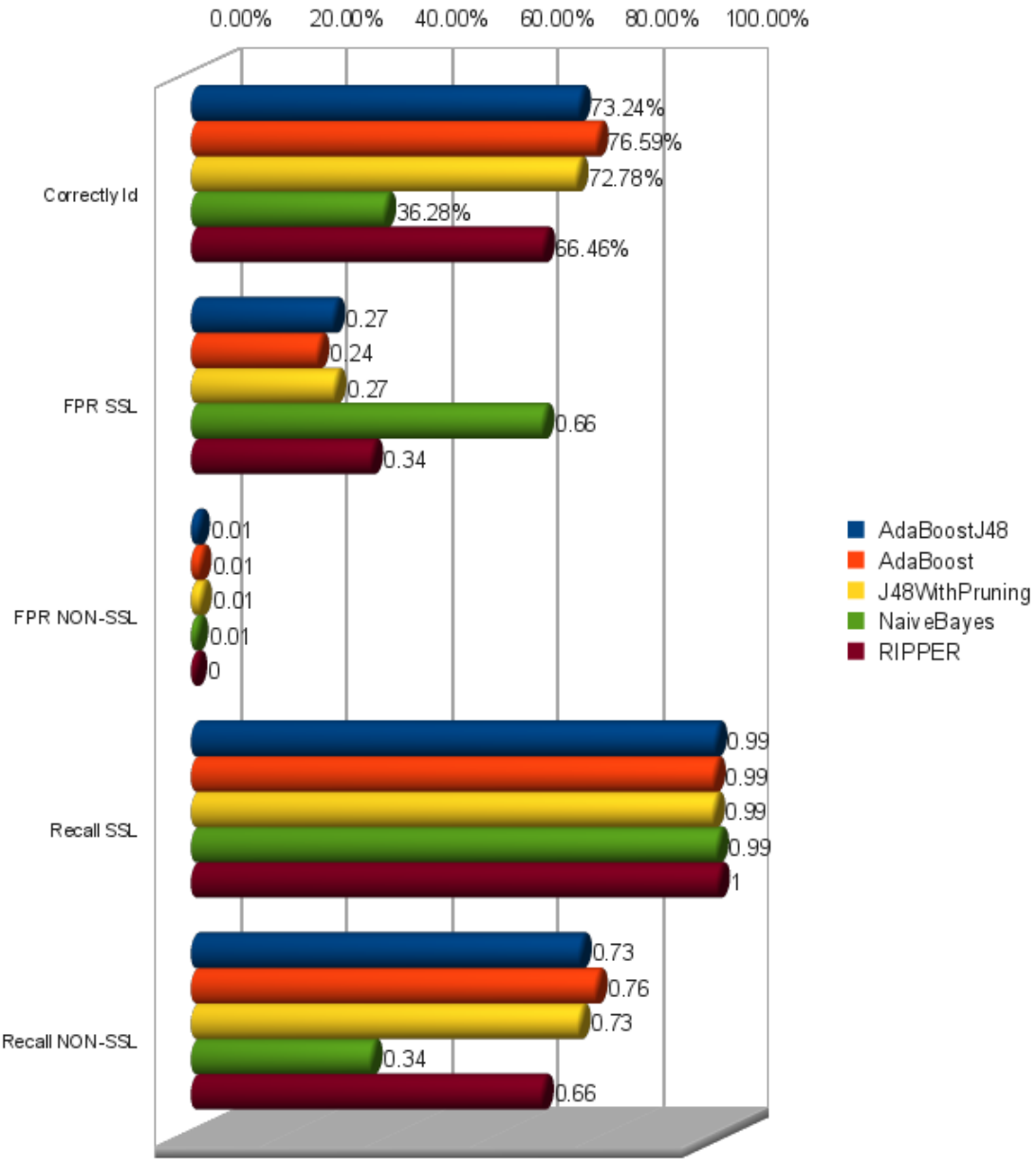

Figure A.4: Training set size 20000 results from each performance metric employed for each ML algorithm

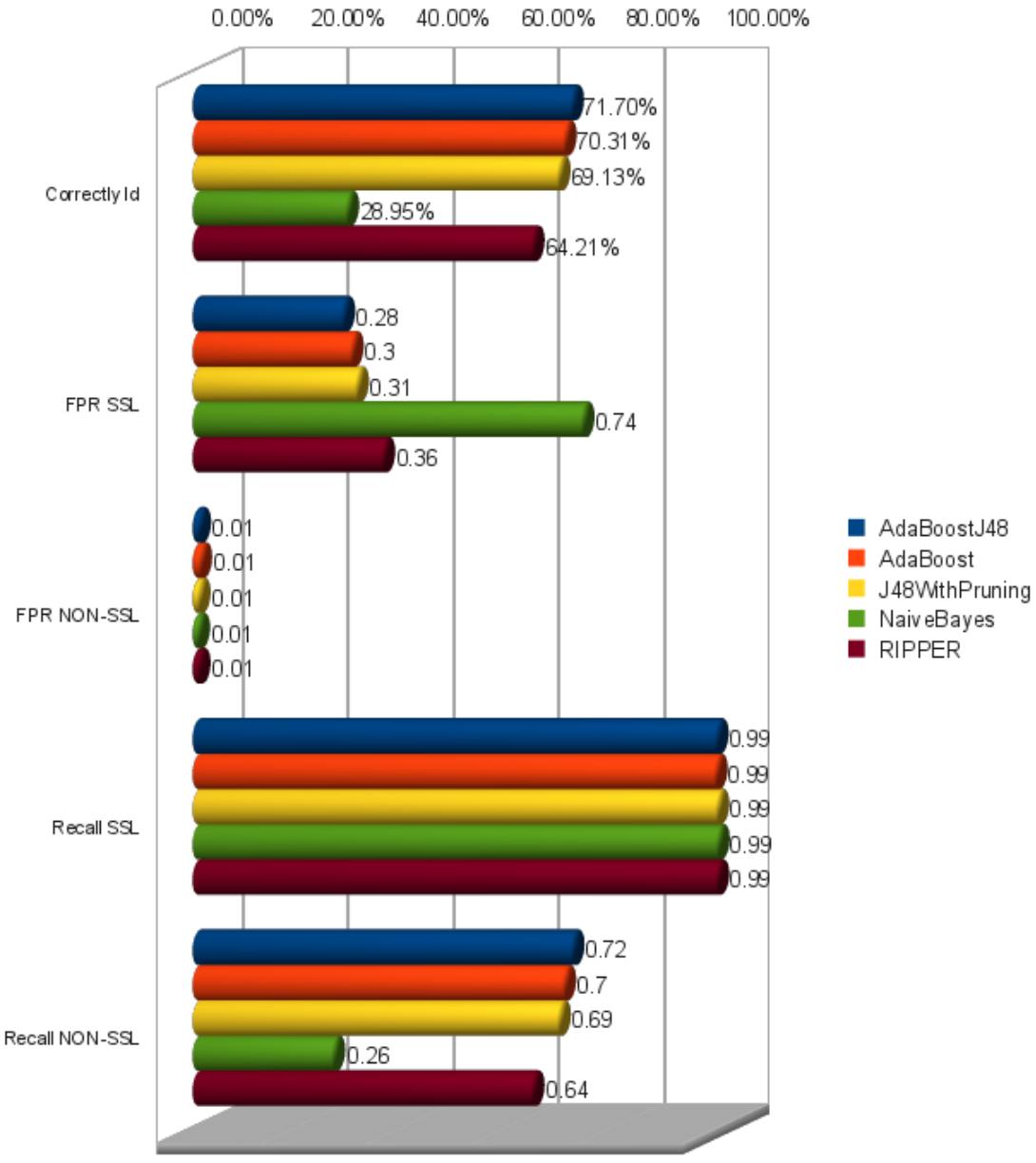

Figure A.5: Training set size 50000 results from each performance metric employed for each ML algorithm

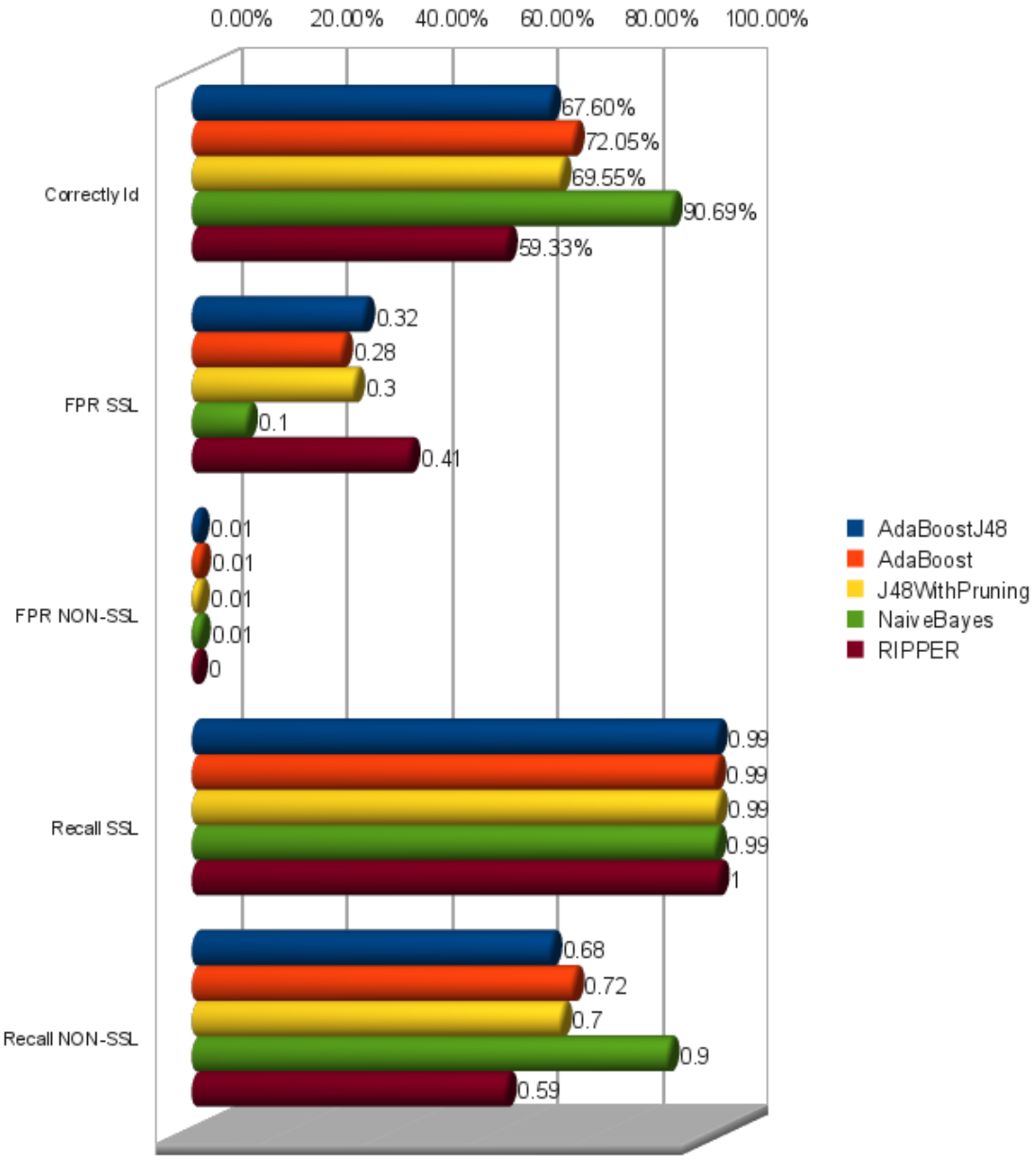

Figure A.6: Training set size 100000 results from each performance metric employed for each ML algorithm

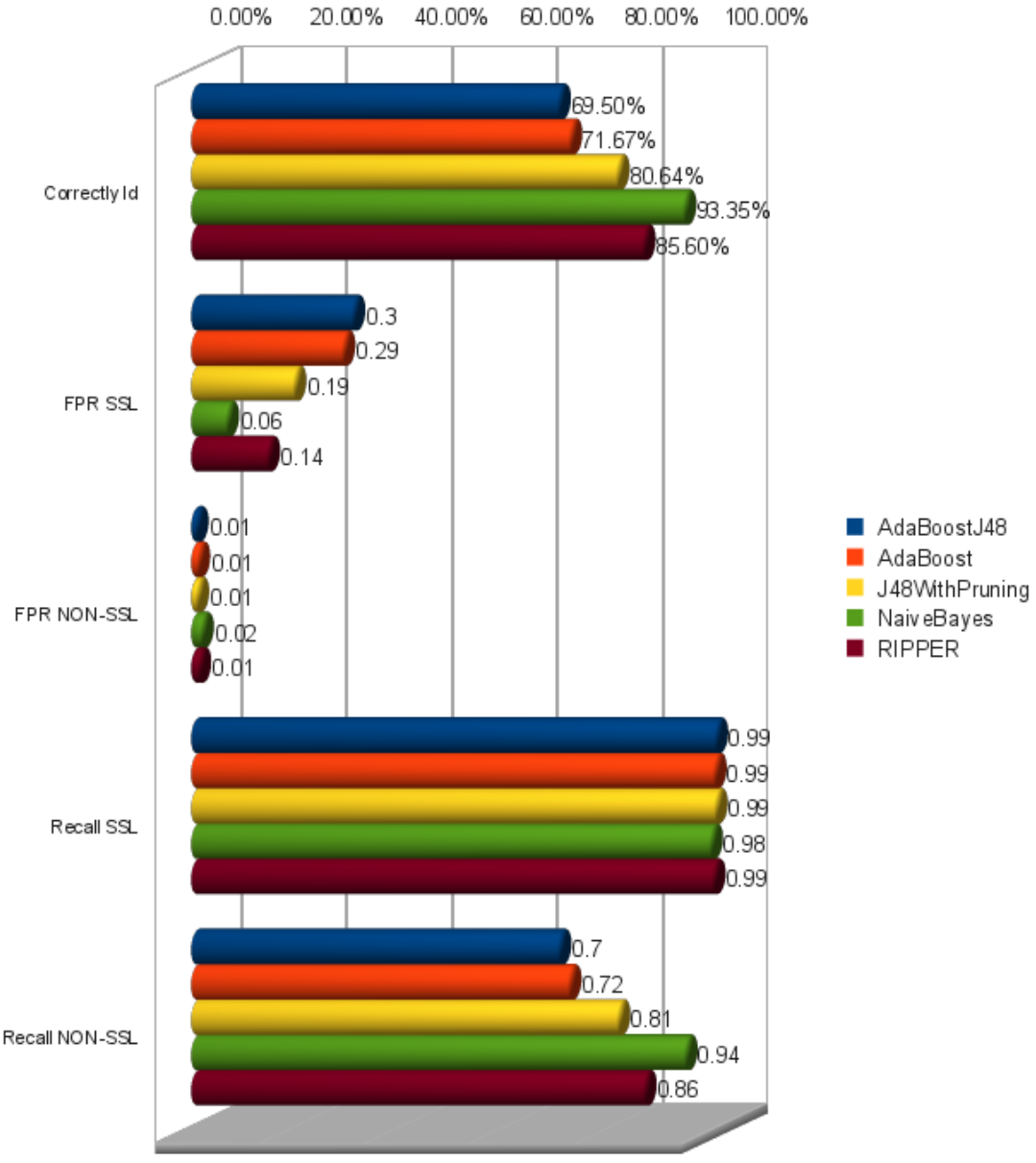

Figure A.7: Training set size 150000 results from each performance metric employed for each ML algorithm

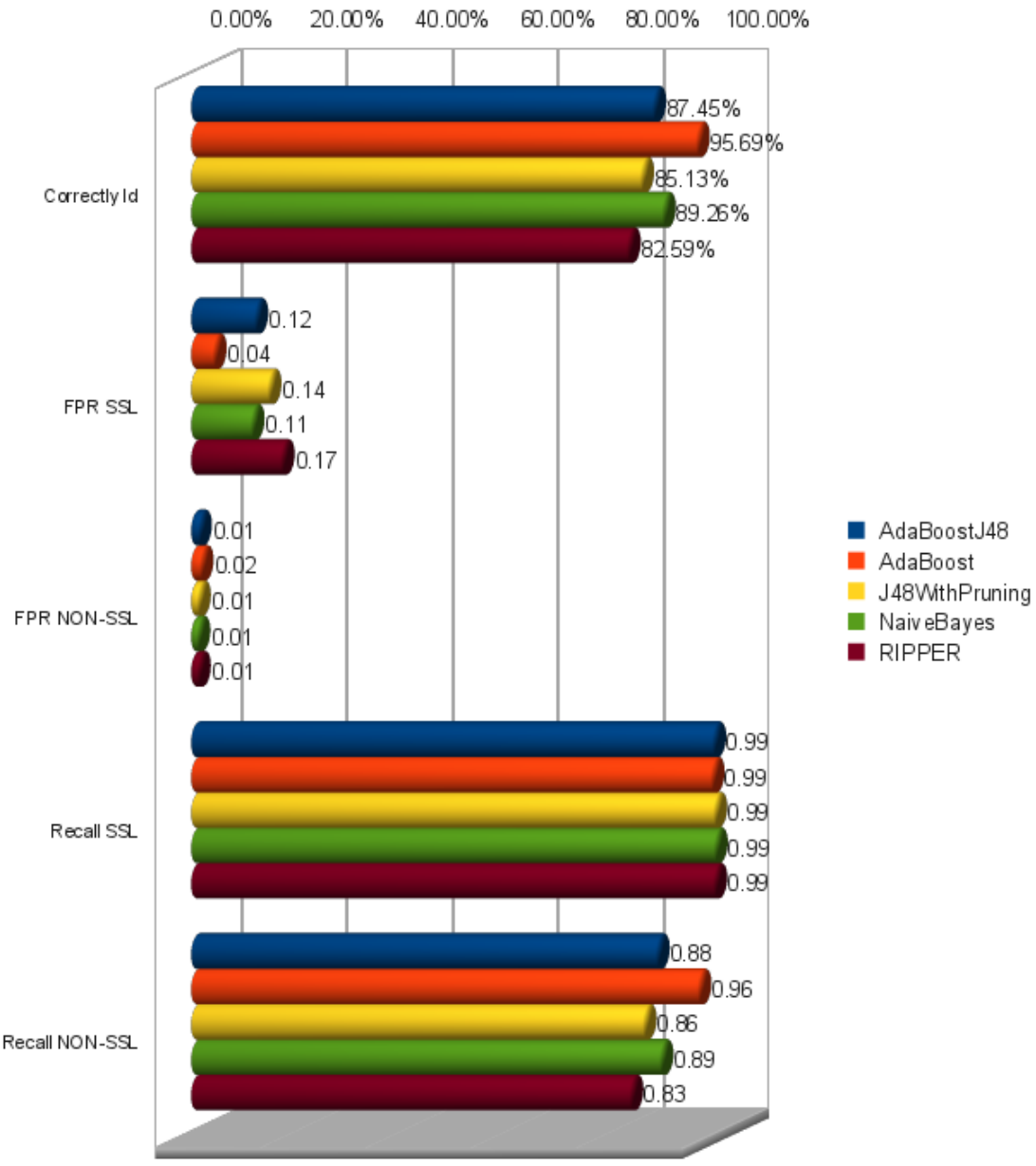

Figure A.8: Training set size 500000 results from each performance metric employed for each ML algorithm

Appendix B

SSL vs SSL-Tunnel

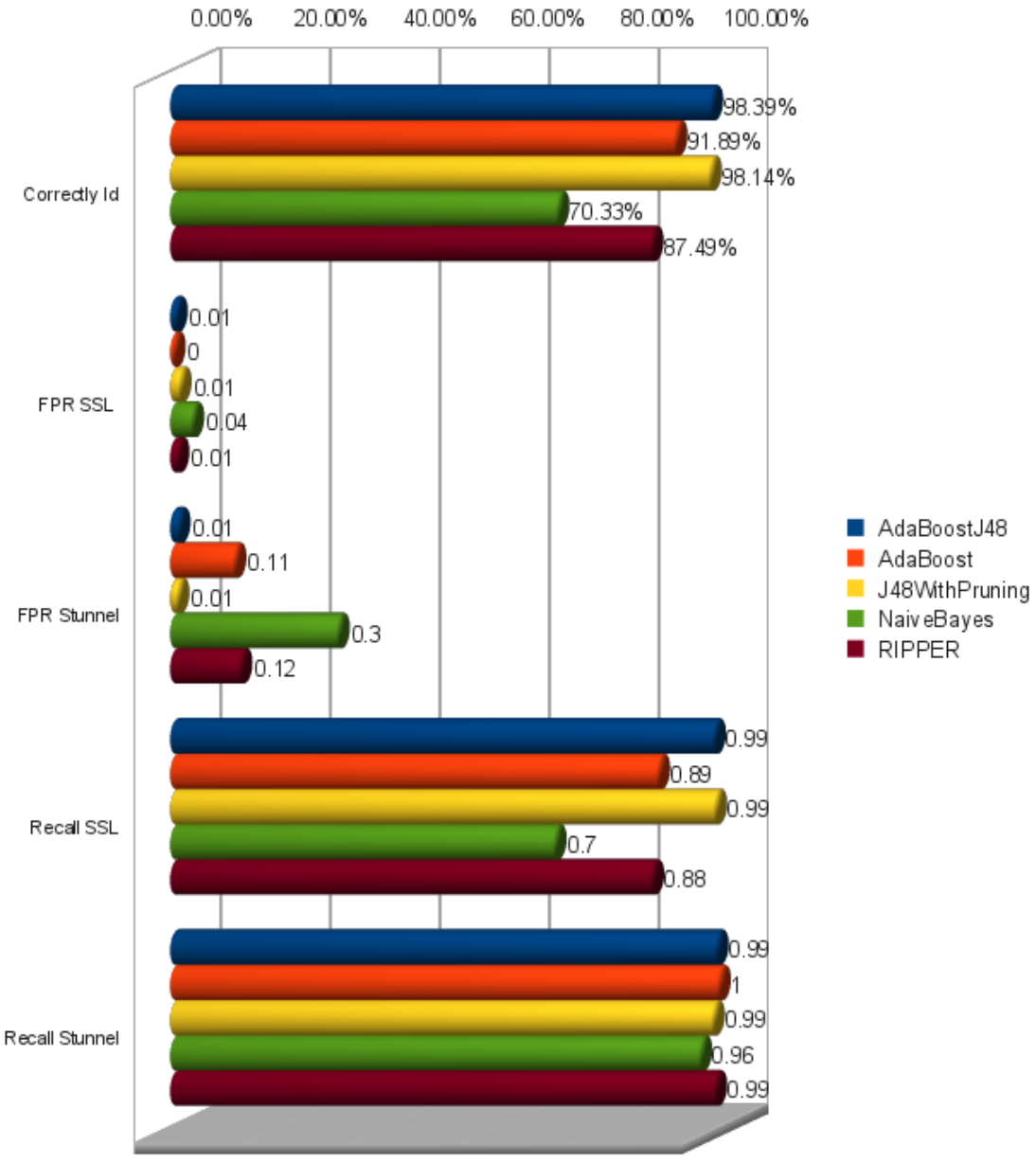

Figure B.1: Training set size 6000 results from each performance metric employed for each ML algorithm

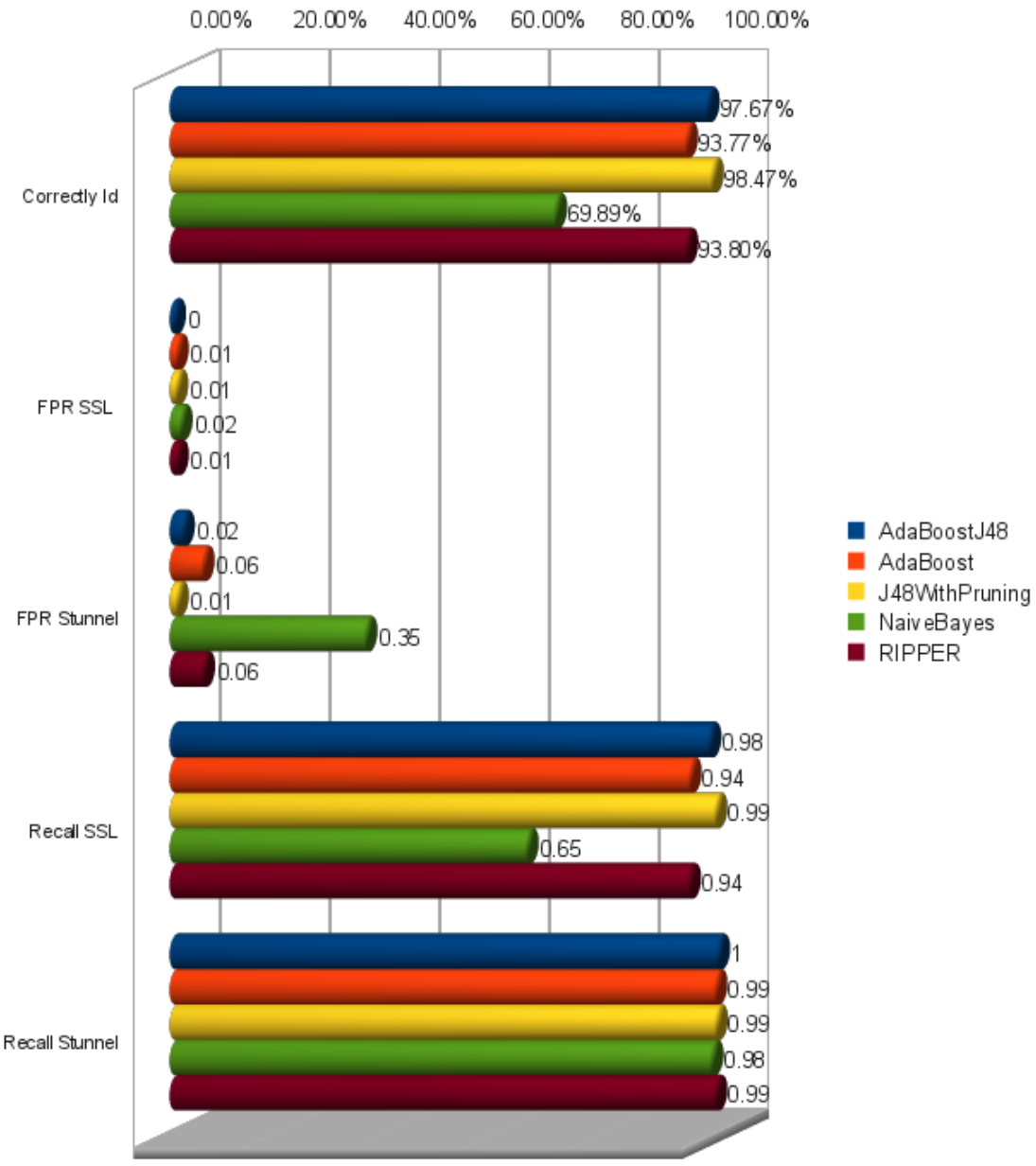

Figure B.2: Training set size 9000 results from each performance metric employed for each ML algorithm

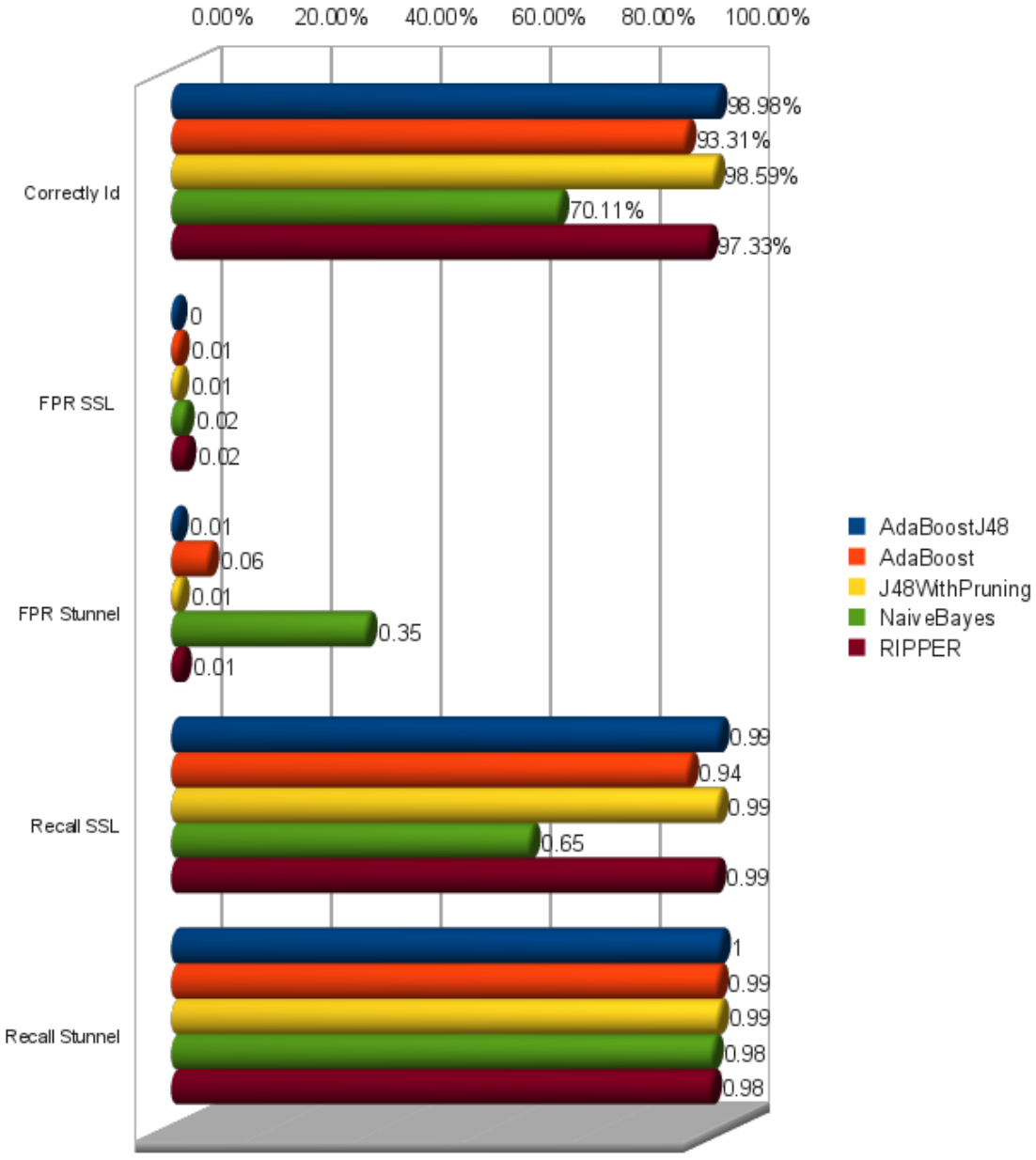

Figure B.3: Training set size 12000 results from each performance metric employed for each ML algorithm

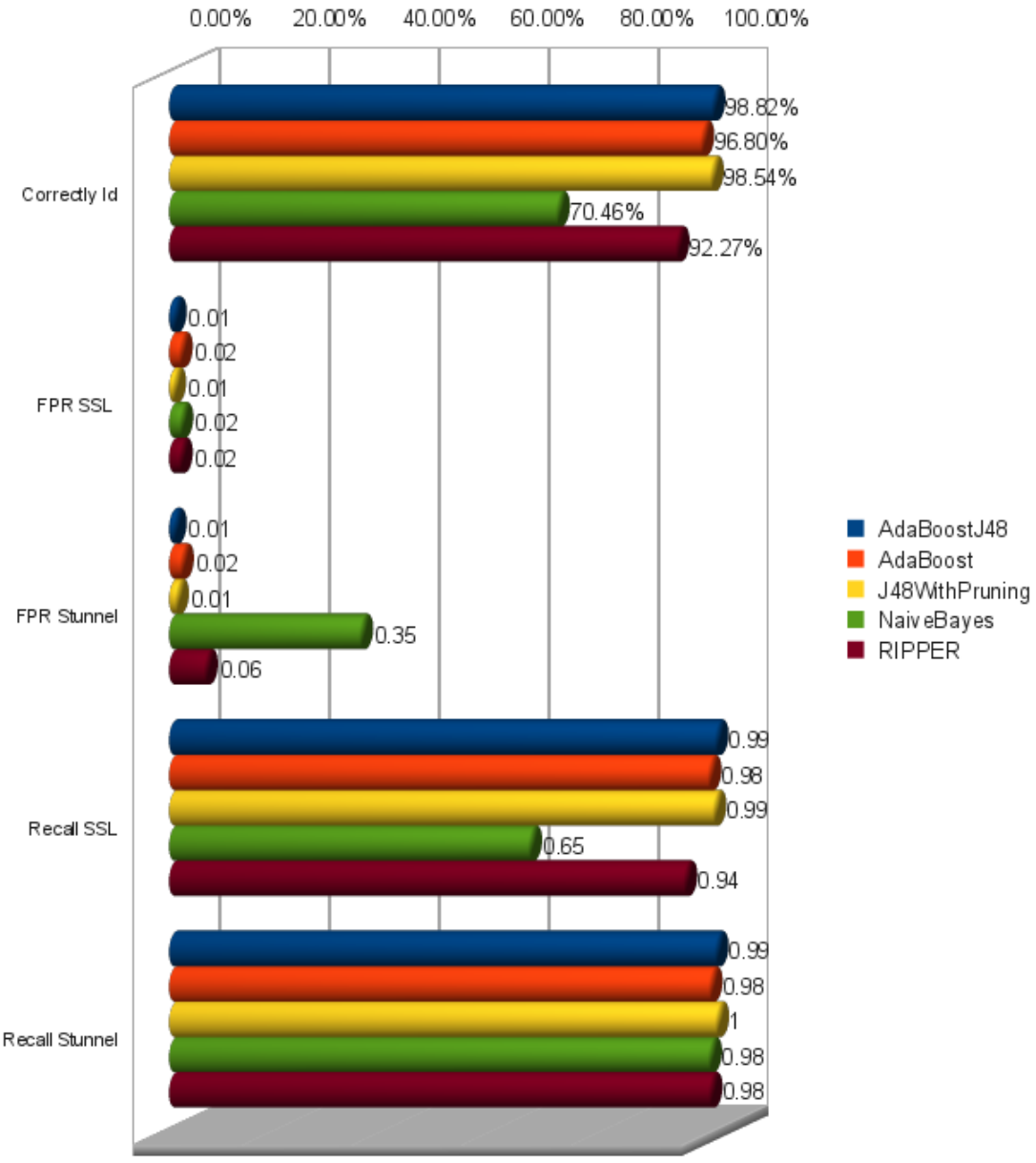

Figure B.4: Training set size 20000 results from each performance metric employed for each ML algorithm

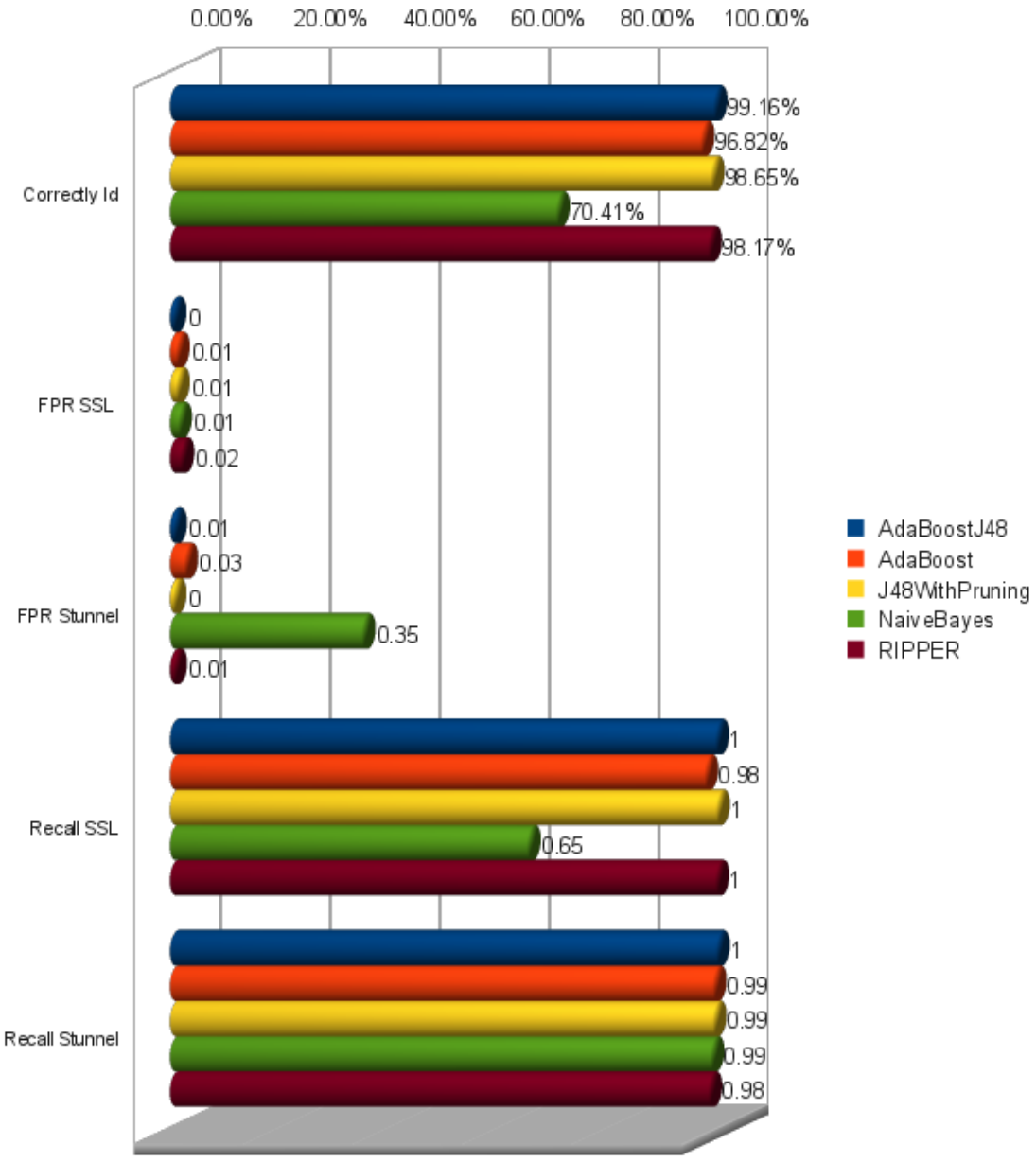

Figure B.5: Training set size 50000 results from each performance metric employed for each ML algorithm

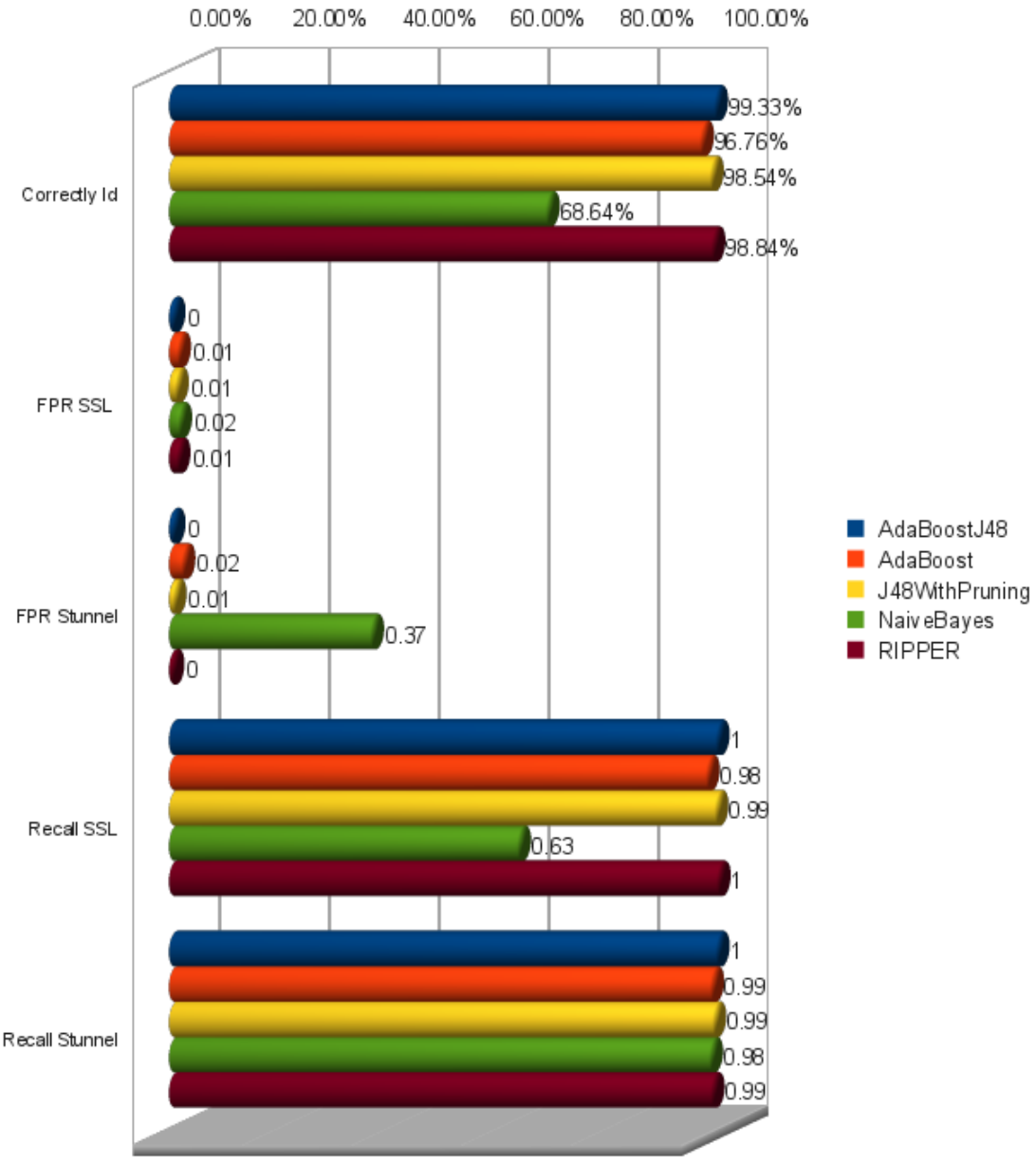

Figure B.6: Training set size 100000 results from each performance metric employed for each ML algorithm

Appendix C

Non-SSL vs SSL-Tunnel

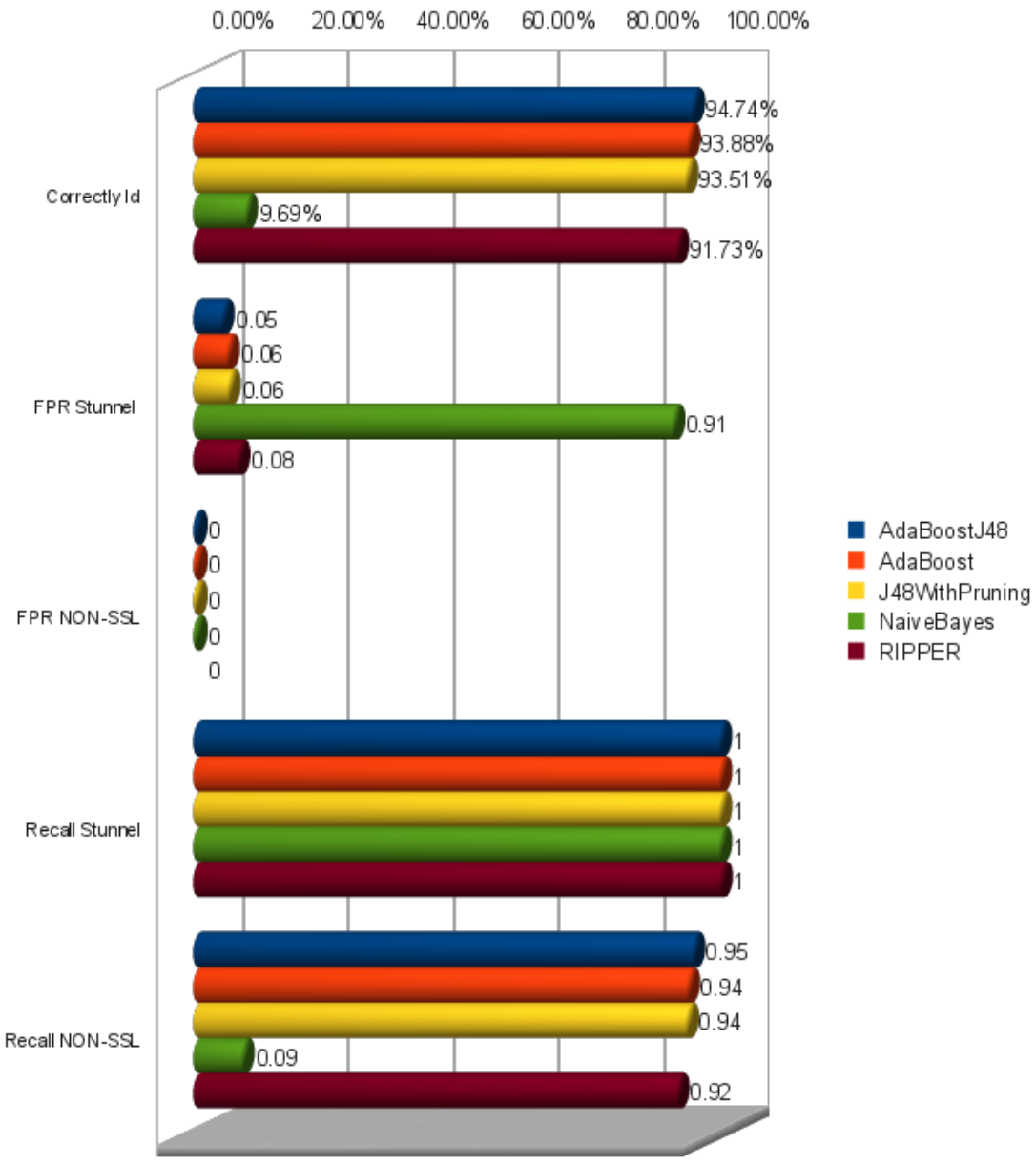

Figure C.1: Training set size 6000 results from each performance metric employed for each ML algorithm

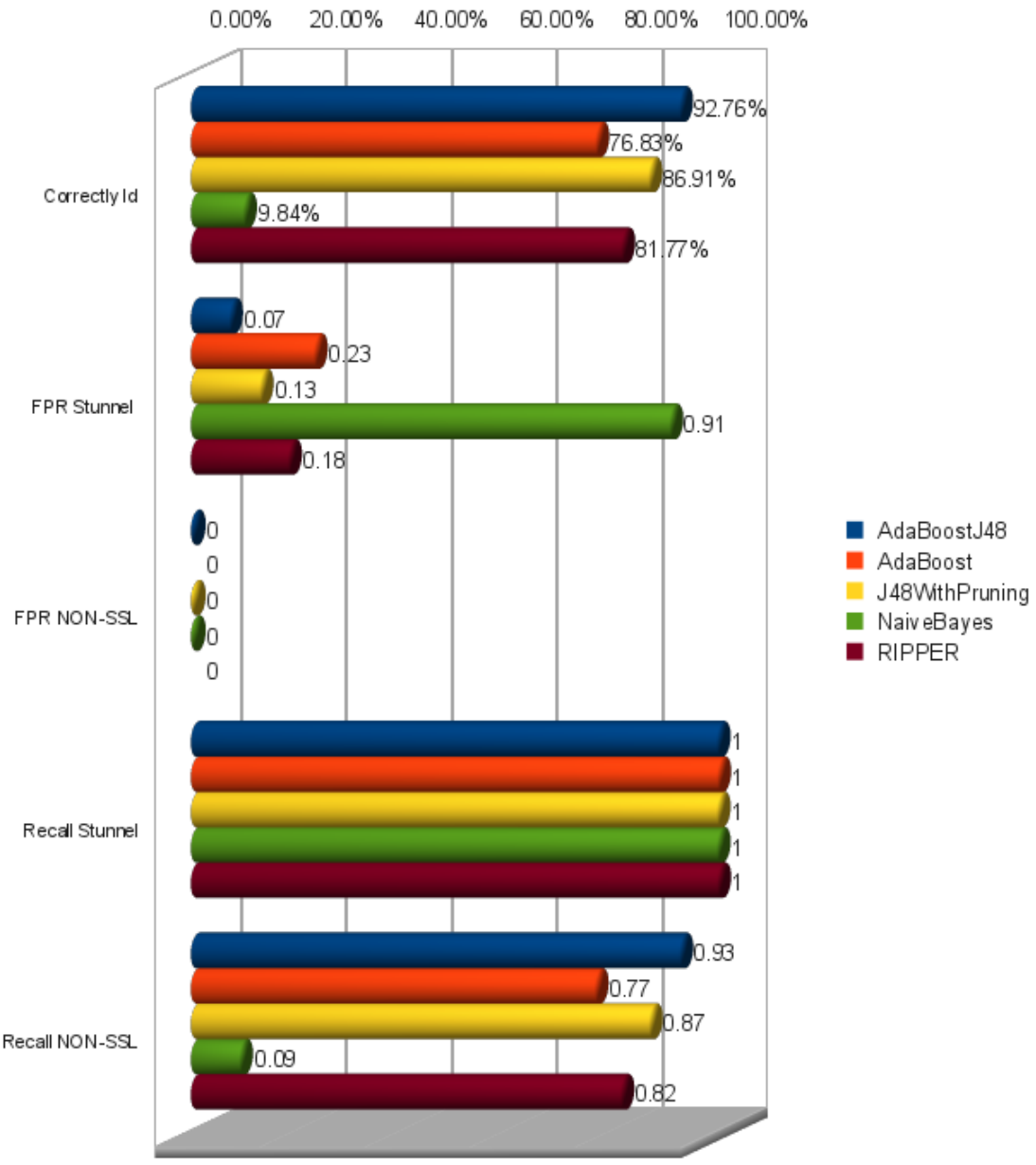

Figure C.2: Training set size 9000 results from each performance metric employed for each ML algorithm

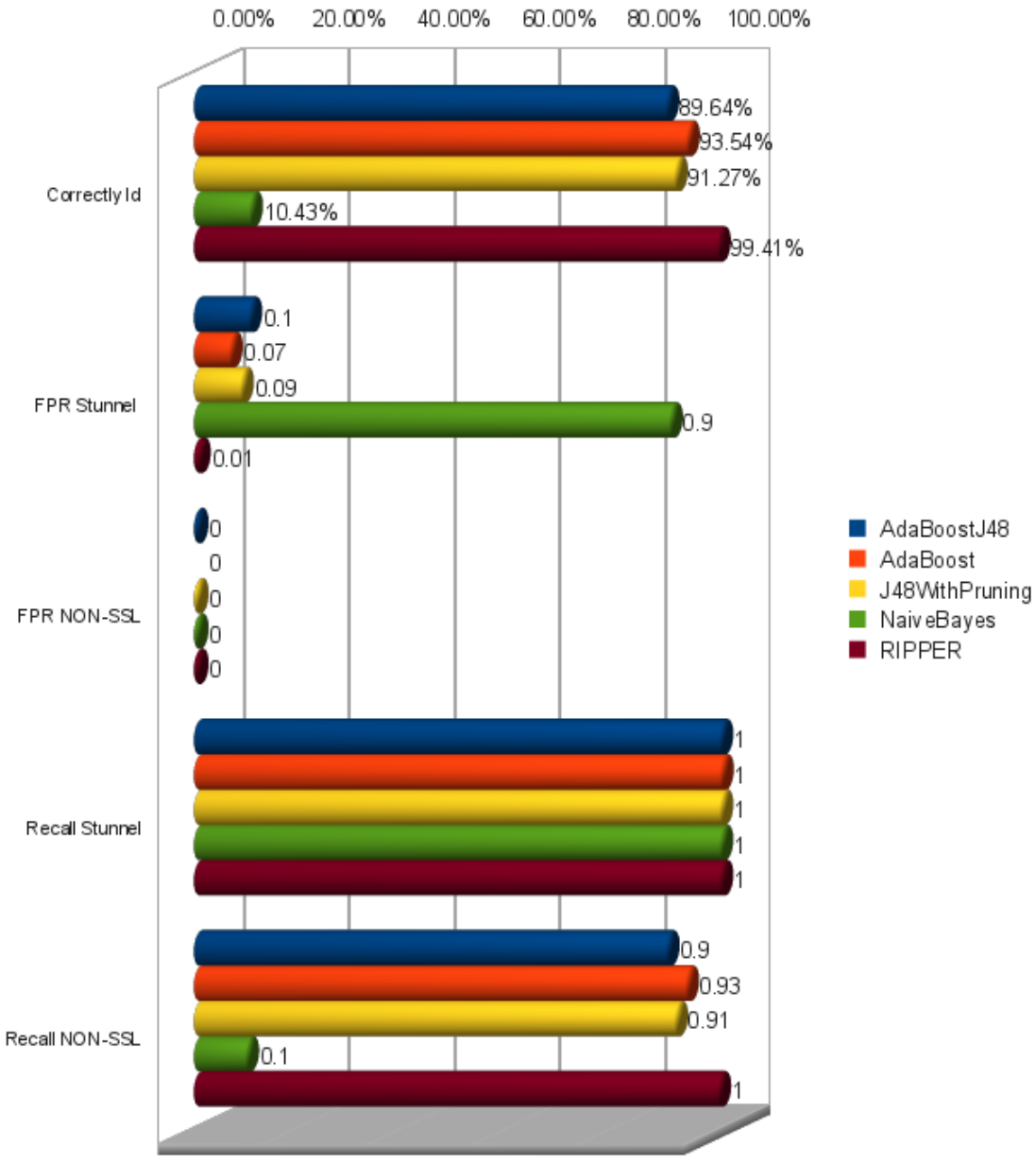

Figure C.3: Training set size 12000 results from each performance metric employed for each ML algorithm

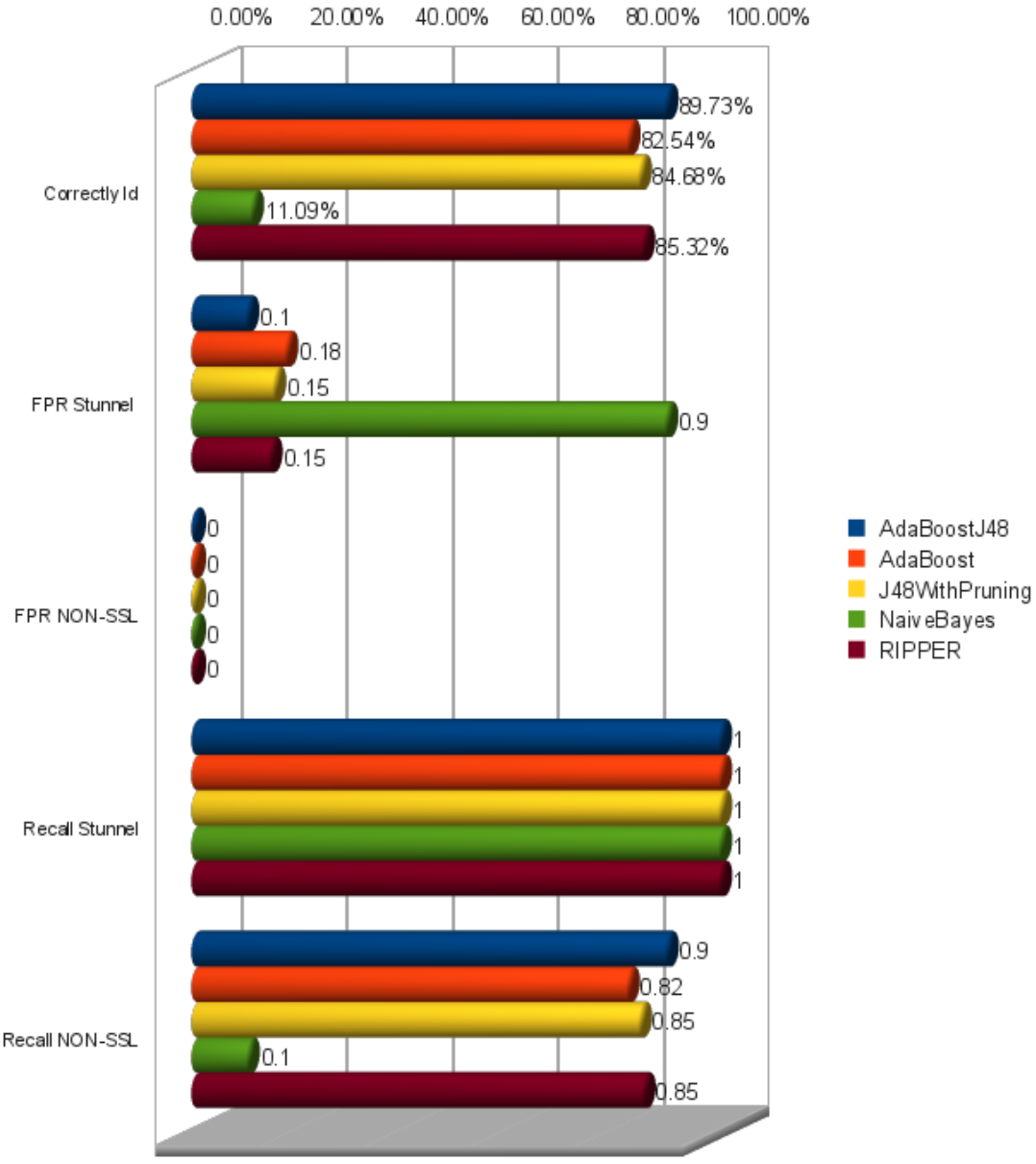

Figure C.4: Training set size 20000 results from each performance metric employed for each ML algorithm

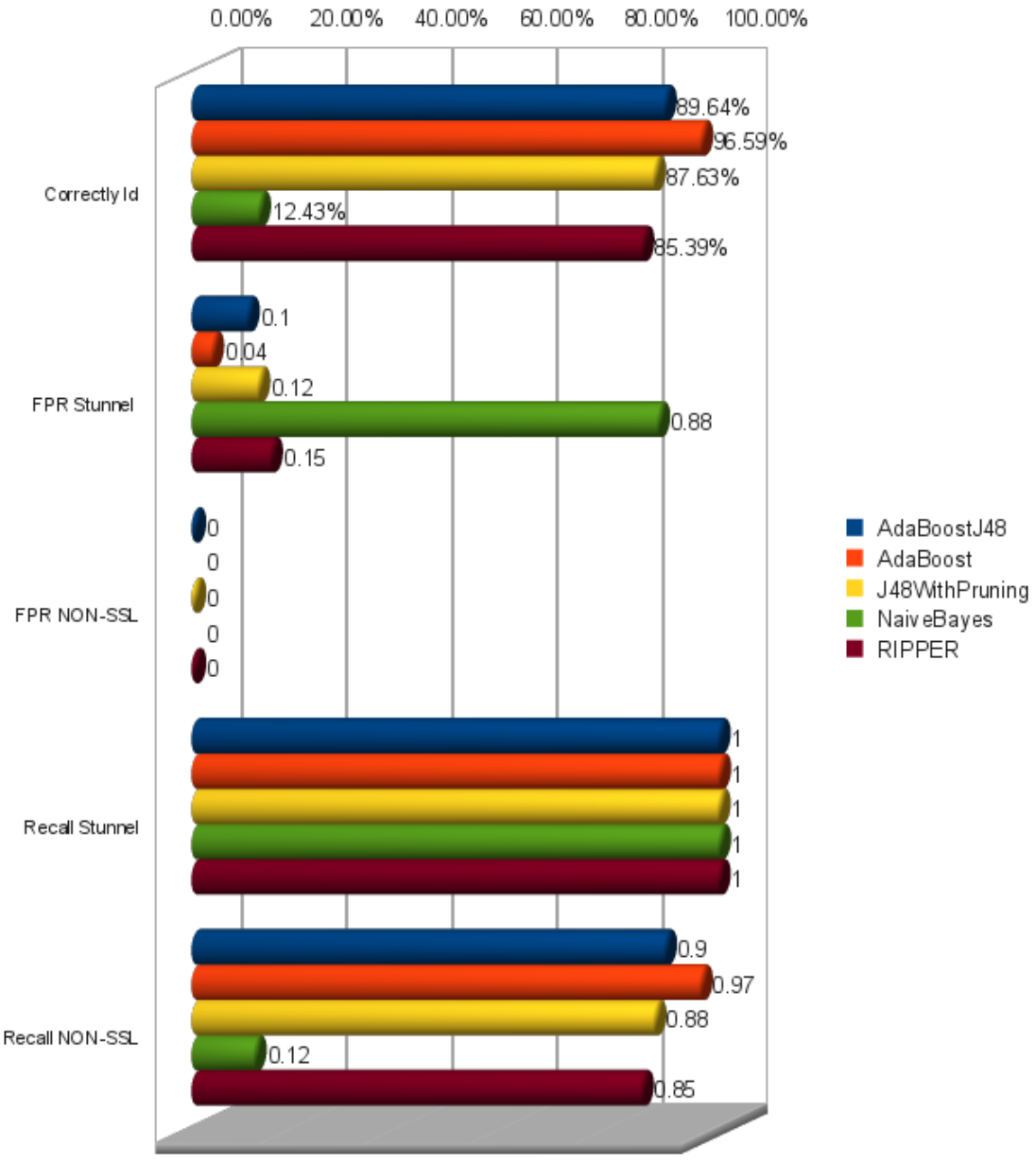

Figure C.5: Training set size 50000 results from each performance metric employed for each ML algorithm

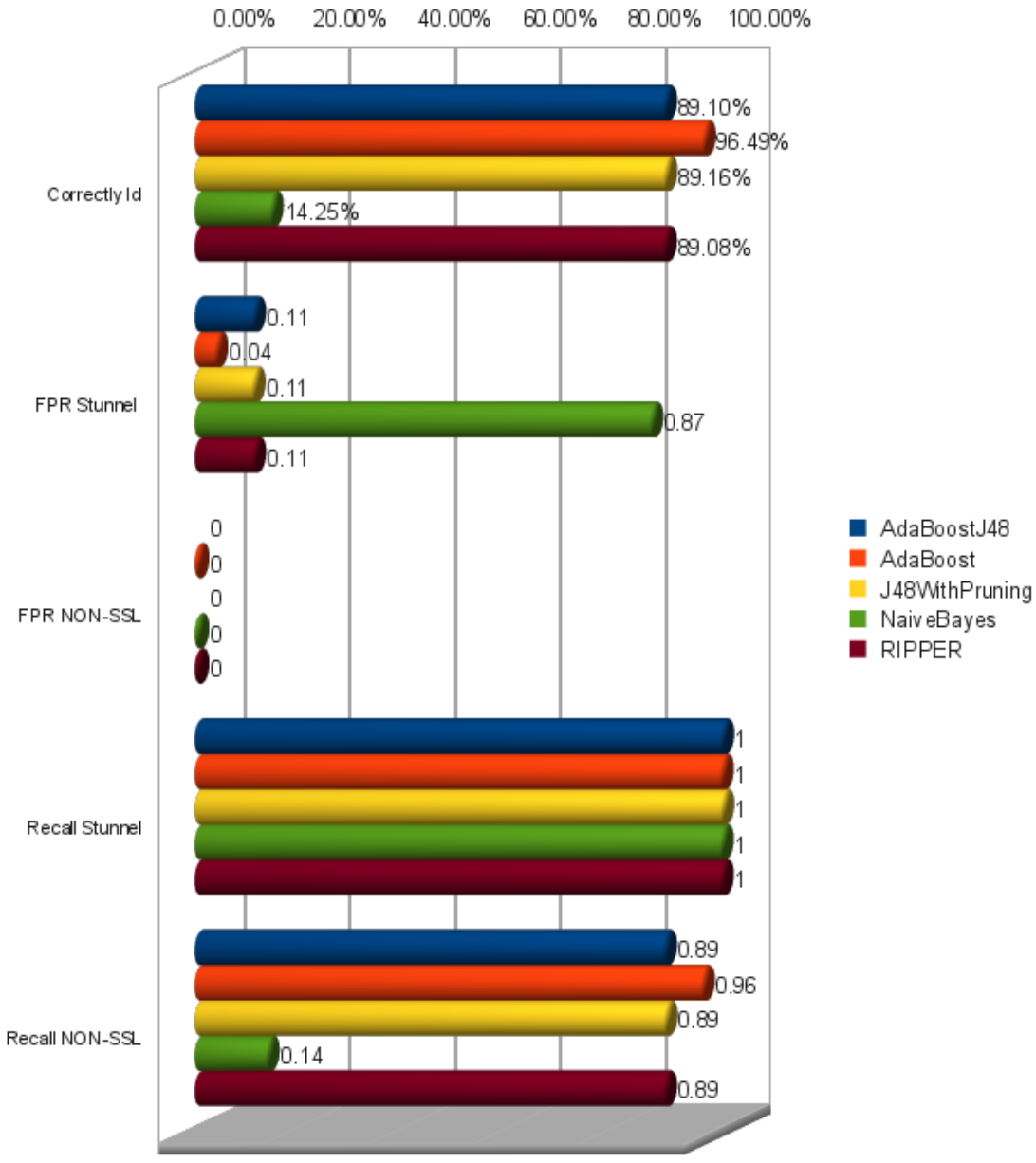

Figure C.6: Training set size 100000 results from each performance metric employed for each ML algorithm

Appendix D

Number of flows per application instance for each training set size in the different class runs

Application Instance	6000	9000	12000	20000
FTPS	115	173	230	384
http_tunnel-HTTPS	115	173	230	384
http_tunnel-POP3S	115	173	230	183
http_tunnel-SMTPS	115	173	230	384
http_tunnel-TELNET-SSL	115	173	230	384
icmp_tunnel-HTTPS	115	173	230	384
icmp_tunnel-POP3S	115	61	61	61
icmp_tunnel-SMTPS	115	173	230	384
JABBER-SSL	115	173	230	384
POP3S	115	173	230	384
SKYPE	115	173	230	384
SMTPS	115	173	230	384
SSL-BITTORRENT	115	173	230	384
stunnel-FTP	115	173	230	384
stunnel-FTPS	115	173	230	384
stunnel-HTTP	115	173	230	384
stunnel-HTTPS	115	173	230	384
stunnel-POP3	115	173	230	384
stunnel-POP3S	115	173	91	91
stunnel-SMTP	115	173	230	384
stunnel-SMTPS	115	173	230	384
stunnel-SSH	115	173	230	384
stunnel-TELNET	115	173	230	384
stunnel-TELNET-SSL	115	173	230	152
TELNET-SSL	115	61	61	61
HTTPS	125	399	727	1772
FTP	150	225	300	500
HTTP	150	225	300	500
http_tunnel-HTTP	150	225	300	500
http_tunnel-POP3	150	225	300	500
http_tunnel-SMTP	150	225	300	500
http_tunnel-SSH	150	225	300	500
http_tunnel-TELNET	150	225	300	500
icmp_tunnel-HTTP	150	225	300	500
icmp_tunnel-POP3	150	225	300	500
icmp_tunnel-SMTP	55	55	55	55
icmp_tunnel-SSH	150	225	300	500
icmp_tunnel-TELNET	150	89	89	89
JABBER	150	225	300	500
POP3	150	225	300	500
SMTP	150	225	300	500
SSH	150	225	300	500
SVN	150	225	300	500
TELNET	150	225	300	500
VUZE-BITTORENT	150	225	300	500
SPURIOUS	245	531	756	1356

Application Instance	50000	100000	150000	500000
FTPS	961	1923	2884	9615
http_tunnel-HTTPS	405	405	405	405
http_tunnel-POP3S	183	183	183	183
http_tunnel-SMTPS	961	1923	2884	9615
http_tunnel-TELNET-SSL	961	1923	2884	4670
icmp_tunnel-HTTPS	961	1923	1434	1434
icmp_tunnel-POP3S	61	61	61	61
icmp_tunnel-SMTPS	961	673	673	673
JABBER-SSL	961	1923	2884	9615
POP3S	961	1923	2884	4175
SKYPE	961	1923	2884	1744
SMTPS	961	1923	2884	9615
SSL-BITTORRENT	961	1923	2884	9615
stunnel-FTP	961	1923	2884	9615
stunnel-FTPS	961	1923	2884	9615
stunnel-HTTP	961	1923	2884	9615
stunnel-HTTPS	961	1923	2884	1761
stunnel-POP3	961	1923	2884	9615
stunnel-POP3S	91	91	91	91
stunnel-SMTP	961	1923	2884	9615
stunnel-SMTPS	961	1923	2884	9615
stunnel-SSH	961	1923	2884	9615
stunnel-TELNET	961	1923	2884	9615
stunnel-TELNET-SSL	152	152	152	152
TELNET-SSL	61	61	61	61
HTTPS	5788	13760	22912	109595
FTP	1250	2500	3750	12500
HTTP	1250	2500	3750	12500
http_tunnel-HTTP	1250	2500	3750	12500
http_tunnel-POP3	1250	2500	3750	12500
http_tunnel-SMTP	1250	2500	3750	12500
http_tunnel-SSH	1250	2500	3750	12500
http_tunnel-TELNET	1250	2500	3750	12500
icmp_tunnel-HTTP	386	386	386	386
icmp_tunnel-POP3	1250	2500	3750	2372
icmp_tunnel-SMTP	55	55	55	55
icmp_tunnel-SSH	494	494	494	494
icmp_tunnel-TELNET	89	89	89	89
JABBER	1250	2500	3750	12500
POP3	1250	2500	3750	12500
SMTP	1250	2500	3750	12500
SSH	1250	2500	3750	12500
SVN	1250	2500	3750	12500
TELNET	1250	2500	3750	12500
VUZE-BITTORENT	1250	2500	3750	12500
SPURIOUS	5226	11476	17726	71604

Application Instance	6000	9000	12000	20000
FTPS	200	300	400	666
http_tunnel-HTTPS	200	300	400	666
http_tunnel-POP3S	200	300	400	666
http_tunnel-SMTPS	200	300	400	666
http_tunnel-TELNET-SSL	200	300	400	666
icmp_tunnel-HTTPS	200	300	400	666
icmp_tunnel-POP3S	61	61	61	61
icmp_tunnel-SMTPS	200	300	400	666
JABBER-SSL	200	300	400	666
POP3S	200	300	400	666
SKYPE	200	300	400	666
SMTPS	200	300	400	666
SSL-BITTORRENT	200	300	400	666
TELNET-SSL	61	61	61	61
HTTPS	478	778	1078	1886
stunnel-FTP	272	409	545	909
stunnel-FTPS	272	409	545	909
stunnel-HTTPS	272	409	545	909
stunnel-POP3	272	409	545	909
stunnel-POP3S	102	102	102	102
stunnel-SMTP	272	409	545	909
stunnel-SMTPS	272	409	545	909
stunnel-SSH	272	409	545	909
stunnel-TELNET	272	409	545	909
stunnel-TELNET-SSL	61	61	61	61
stunnel-HTTP	661	1065	1477	2565

Table D.3: Number of application instance flows included in each training set size for **SSL vs SSL-Tunnel**.

Application Instance	50000	100000	150000	500000
FTPS	1666	3333	5000	16666
http_tunnel-HTTPS	405	405	405	405
http_tunnel-POP3S	1666	3333	5000	16666
http_tunnel-SMTPS	1666	3333	5000	16666
http_tunnel-TELNET-SSL	1666	3333	5000	4670
icmp_tunnel-HTTPS	1666	1434	1434	1434
icmp_tunnel-POP3S	61	61	61	61
icmp_tunnel-SMTPS	673	673	673	673
JABBER-SSL	1666	3333	5000	16666
POP3S	1666	3333	5000	4393
SKYPE	1666	3333	1744	1744
SMTPS	1666	3333	5000	16666
SSL-BITTORRENT	1666	3333	5000	16666
TELNET-SSL	61	61	61	61
HTTPS	7140	17369	30622	136563
stunnel-FTP	2272	4545	6818	5106
stunnel-FTPS	2272	4545	6818	22727
stunnel-HTTPS	2272	1761	1761	1761
stunnel-POP3	2272	4545	6818	22727
stunnel-POP3S	102	102	102	102
stunnel-SMTP	2272	4545	6818	22727
stunnel-SMTPS	2272	4545	6818	22727
stunnel-SSH	2272	4545	6818	22727
stunnel-TELNET	2272	4545	6818	22727
stunnel-TELNET-SSL	61	61	61	61
stunnel-HTTP	6661	16261	25350	106608

Table D.4: Number of application instance flows included in each training set size for **SSL vs SSL-Tunnel**.

Application Instance	6000	9000	12000	20000
stunnel-FTP	272	409	545	909
stunnel-FTPS	272	409	545	909
stunnel-HTTPS	272	409	545	909
stunnel-POP3	272	409	545	909
stunnel-POP3S	102	102	102	102
stunnel-SMTP	272	409	545	909
stunnel-SMTPS	272	409	545	909
stunnel-SSH	272	409	545	909
stunnel-TELNET	272	409	545	909
stunnel-TELNET-SSL	272	152	152	152
stunnel-HTTP	450	974	1386	2474
FTP	142	214	285	476
HTTP	142	214	285	476
http_tunnel-HTTP	142	214	285	476
http_tunnel-POP3	142	214	285	476
http_tunnel-POP3S	142	92	92	92
http_tunnel-SMTP	142	214	285	476
http_tunnel-SSH	142	214	285	476
http_tunnel-TELNET	142	214	285	476
icmp_tunnel-HTTP	142	214	285	476
icmp_tunnel-POP3	142	214	285	476
icmp_tunnel-SMTP	55	55	55	55
icmp_tunnel-SSH	142	214	285	476
icmp_tunnel-TELNET	142	89	89	89
JABBER	142	214	285	476
POP3	142	214	285	476
SMTP	142	214	285	476
SSH	142	214	285	476
SVN	142	214	285	476
TELNET	142	214	285	476
VUZE-BITTORENT	142	214	285	476
SPURIOUS	247	626	919	1672

Table D.5: Number of application instance flows included in each training set size for **Non-SSL vs SSL-Tunnel**.

Application Instance	50000	100000	150000	500000
stunnel-FTP	2272	4545	6818	5106
stunnel-FTPS	2272	4545	6818	22727
stunnel-HTTPS	2272	1761	1761	1761
stunnel-POP3	2272	4545	6818	22727
stunnel-POP3S	102	102	102	102
stunnel-SMTP	2272	4545	6818	22727
stunnel-SMTPS	2272	4545	6818	22727
stunnel-SSH	2272	4545	6818	22727
stunnel-TELNET	2272	4545	6818	22727
stunnel-TELNET-SSL	152	152	152	152
stunnel-HTTP	6570	16170	25259	106517
FTP	1190	2380	3571	11904
HTTP	1190	2380	3571	11904
http_tunnel-HTTP	1190	2380	3571	11904
http_tunnel-POP3	1190	2380	3571	11904
http_tunnel-POP3S	92	92	92	92
http_tunnel-SMTP	1190	2380	3571	11904
http_tunnel-SSH	1190	2380	3571	11904
http_tunnel-TELNET	1190	2380	3571	11904
icmp_tunnel-HTTP	386	386	386	386
icmp_tunnel-POP3	1190	2380	3571	2372
icmp_tunnel-SMTP	55	55	55	55
icmp_tunnel-SSH	494	494	494	494
icmp_tunnel-TELNET	89	89	89	89
JABBER	1190	2380	3571	11904
POP3	1190	2380	3571	11904
SMTP	1190	2380	3571	11904
SSH	1190	2380	3571	11904
SVN	1190	2380	3571	11904
TELNET	1190	2380	3571	11904
VUZE-BITTORENT	1190	2380	3571	11904
SPURIOUS	6034	13184	20319	79856

Table D.6: Number of application instance flows included in each training set size for **Non-SSL vs SSL-Tunnel**.

Appendix E

Fine Tuning AdaBoost

Size	Correctly Id.	FPR SSL	FPR Non-SSL	Recall SSL	Recall Non-SSL
6000	71%	0.28	0.02	0.94	0.72
9000	67%	0.32	0.01	0.99	0.68
12000	68%	0.32	0.01	0.99	0.69
20000	78%	0.21	0.01	0.99	0.79
50000	70%	0.3	0.01	0.99	0.71
100000	72%	0.28	0.01	0.99	0.73
150000	71%	0.28	0.01	0.99	0.72
500000	82%	0.18	0.01	0.99	0.82

Table E.1: Results from changing the re-sampling option for AdaBoost in the SSL vs Non-SSL run show little change.

Size	Correctly Id.	FPR SSL	FPR Non-SSL	Recall SSL	Recall Non-SSL
6000	62%	0.37	0.02	0.98	0.63
9000	67%	0.32	0.01	0.99	0.68
12000	68%	0.31	0.01	0.99	0.69
20000	78%	0.21	0.01	0.99	0.79
50000	70%	0.3	0.01	0.99	0.71
100000	72%	0.28	0.01	0.99	0.73
150000	71%	0.28	0.01	0.99	0.72
500000	78%	0.216	0.01	0.99	0.78

Table E.2: Results from selecting alternate weight thresholds for AdaBoost in the SSL vs Non-SSL run show little change.