MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Self-optimizing traffic classification framework

MASTER THESIS

Ján Rusnačko

Brno, spring 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: RNDr. Marián Novotný, Ph.D.

Acknowledgement

I would like to thank my advisor Dr. Marián Novotný for his guidance, endless patience and valuable feedback. I am thankful to my brother for his thorough reviews. I am most grateful to my fiancée and family for their never-ending encouragement and support. Without them this work would not have been possible.

Abstract

Traffic classification is an important part of network management with respect to quality of service and security monitoring. We introduce a novel framework for designing traffic classification algorithms based on statistical flow features. We build on the results achieved by SPID and use multilevel clustering with custom classifiers to avoid peaking effect. Furthermore, we introduce several novel flow features and optimize the structure and parameters of the classifier with a genetic algorithm. The effectiveness of the proposed solution is evaluated on traces of real traffic and compared with SPID. Preliminary results show improved precision and recall with substantial increase of speed of classification.

Keywords

Traffic Classification, Machine Learning, Clustering, SPID

Contents

Introduction

Network traffic classification has been one of the main research areas in networking and security for at least a decade. During this time, several main approaches to this problem have been established, yet none of them proved superior. This is mainly due to the nature of traffic classification, with inherent limitations imposed by protocols, environment, performance requirements, technology, laws and policies etc. With such restrictions, suitable solution often depends heavily on the problem statement. Different approaches usually cover different use cases and therefore we can see concurrent research in several directions instead of one being dominant. Moreover, we see strong influences between them that allow to mitigate weaknesses, thus for example it is not uncommon to see methods of deep packet inspection applied in statistical protocol classification.

Motivation for network traffic classification is also at least twofold. Firstly, we can view it as a core tool that enables efficient network management. Network administrators and Internet service providers must rely on packet classification in most aspects of their work. With growing implementation of converged networks, quality of service and traffic engineering are key aspects of network management. The inability to differentiate notoriously performance consuming peer to peer file sharing protocols and VoIP traffic would threaten even basic functionality of a converged network. The key requirement in this use case is performance - classification must be done on the fly with minimum latency and very low resource consumption.

The second motivation comes from network threats and increasing importance of security. While we can view it as an integral part of network management, it has substantially different requirements. In this use case, the classification algorithm needs to be precise and robust against countermeasures, since attackers usually actively use obfuscation. Most obvious example of this is full packet encryption, since it completely disqualifies methods based on deep packet inspection and pattern matching. Moreover, we need the identification as early as possible - while classification at the end of the flow is completely acceptable when billing a customer, it is crucial in security applications. On the other hand, scalability and performance

might not be of concern when securing a small LAN, whereas it is a priority when implementing quality of service on high-throughput backbone link. Since many requirements are naturally contradictory, algorithms usually either focus only on a few of them or seek acceptable balance. In this thesis, we will try to prove that there is still room for better algorithms that would push the boundaries rather than balance the disadvantages.

The goal of this thesis is to introduce novel classification framework, that would achieve high performance in both speed and memory consumption while retaining high precision. To accommodate different requirements, our framework is designed to be multilevel and modular - this will allow using different metrics and features for different categories of protocols. It should be easily extensible with new features, classification metrics, evaluation metrics and decision rules to be able to take advantage from other approaches. To get optimal results, it is also self-optimizing - presented with training data, evolutionary algorithm is used to select one solution from potentially many configurations, which is then used for the classification. Last but not least, it should be simple enough to be both robust and useful in practice.

The first chapter of this thesis describes the past and the current approaches to traffic classification, methodologies, open problems and achieved results. The second chapter is dedicated to SPID - Statistical Protocol Identification Algorithm designed by Erik Hjelmvik. It will serve as an inspiration and also as a benchmark for performance evaluation. In the third chapter we will describe general structure of the classification framework and reason about its fundamental features. The fourth chapter includes description of concrete design choices regarding metrics and features. Afterwards, the fifth chapter will describe practical implementation and empirical findings. Last chapter contains performance measurements and evaluation.

Chapter 1

Related work

The approaches to traffic classification can be divided into four broad types. Historically the oldest, and by far the most used is port-based classification of IANA assigned well-known and registered ports. This approach is very fast, because the port numbers act as labels and so the packets themselves carry information about application they belong to. However, applications that use non-registered ports cannot be classified and if two endpoints of communication are in agreement, they can exchange any application traffic even on a well-known or registered port. Indeed, research by Moore and Papagiannaki [43] suggests, that port-based classification achieves around 70 % byte accuracy. Madhukar and Williamson [41] showed, that 30-70 % of collected Internet traffic uses a non-registered port and later survey by Maier et al. [42] confirms this result.

1.1 Deep Packet Inspection

The most natural and straightforward approach to solve this problem is Deep Packet Inspection (DPI). Methods based on DPI usually inspect whole packet payload and try to match it with known signatures (patterns). This results in the best precision overall - in fact, DPI methods are often used to establish ground truth about testing data and assign correct protocol labels, later used to evaluate performance or for training of machine learning algorithms [30][8][9][42][2][4][17][55]. As signature definition language, regular expressions are the most prevalent solution - they are more expressive than exact-match strings, but their implementations also use more memory.

The heart of pattern matching algorithm is usually a fast and optimized version of Finite State Automaton. Two main broad categories of these algorithms correspond to Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA) [1]. DFA based pattern matching algorithms [60]

1. Note that in automata theory, DFA recognizes exactly set of regular languages, which correspond to regular expressions.

[36][37][7] have nice memory properties, since every processed character corresponds to exactly one change of state and therefore predictable number of memory accesses. However, for complex regular expression explosion of number of states of DFA is too large to remain practical. NFA based algorithms [19][27] solve this problem - the number of states is in the order of the number of characters in the regular expression. On the other hand, many states can be active concurrently and one character can trigger multiple state updates resulting in unpredictable number of memory operations [6]. To mitigate these fundamental problems, parallel processing [60][57], hybrid state automata [6] and other [26][33] algorithms are studied.

DPI classification is widely used in practice and is implemented even in network devices. Therefore, a lot of research is dedicated to optimizing DPI algorithms on dedicated hardware and for lower memory and CPU usage [14][49][54][11][15]. Major hardware vendors such as Cisco, Juniper, Alcatel-Lucent and Huawei offer DPI capabilities in their routers and firewalls.

There are also many software applications for traffic classification which make use of DPI. L7-filter is open-source Linux de-facto classifier based on Netfilter subsystem, which allows QoS and resource redistribution based on the priority of application. It is based on manually created patterns (current number of classified protocols is around 100), which contain regular expressions.Its clean integration into Linux kernel is an advantage, but it also suffers from two problems common for many DPI implementations:

- manual creation and maintenance of pattern files is time consuming and does not allow L7-filter to scale well with too many applications
- software implemented DPI is very resource heavy, even though research shows performance can be further optimized [22][21]

Another type of applications which build on DPI are network intrusion detection systems (NIDS). Two most popular are Bro [48] and Snort [50], both of which are open source.

Despite many advantages of DPI and its widespread usage in practice, it also comes with several fundamental shortcomings. The first is inherently high resource usage, which comes with inspection and pattern matching of all bytes in the packet payload. The second disadvantage is the complete inability to classify encrypted or obfuscated traffic. If encryption is done properly, DPI methods have no information about the content of packets and fail to perform correctly. This gives an advantage to protocol designers

and users, who might want to hide the traffic. Another issue is privacy law and policies, which set strict boundaries for DPI deployment.

1.2 Host based classification

Host based classification is an innovative approach, which tries to overcome the disadvantages of DPI. In a sense, it is the complete opposite to DPI - instead of looking deep into the packets and looking for specific patterns, it rather focuses on a high level view on interactions on network and behavior between the hosts. The base information that algorithm collects are network-wide flow data and social profiles of hosts.

One of the first algorithms which pioneered this approach was BLINC [30]. It builds host profile at the social (e.g. popularity of host, membership in node communities), functional (role of provider or consumer) and application (transport layer interactions) level. The classification itself is done by comparing the created profile with known signatures. Work on Traffic Dispersion Graphs also has a potential to classify hosts based on their interactions [31] [28].

The data for these algorithms may come from netflow collectors, which are usually already deployed on networks. Another advantages of this approach are offline computation resulting in overall good performance, good robustness and resistance against countermeasures. However, disadvantage is inability to classify particular flows, and is therefore more suitable for intrusion detection and finding anomalies.

1.3 Flow-features based classification and machine learning

Flow-features based classification is a middle ground between a very low level view of DPI and a very high level view of host based methods. Network flow is usually defined as a set of packets, that share common five tuple ⟨source IP address, destination IP address, source port, destination port, protocol⟩, but this definition holds for a time snapshot. In practice, flows also have a time dimension. They have a start and an end - either explicit, or forced by timeout - and two sets of packets with common five tuple will be actually treated as different flows, when separated by some time window.

The flow acts as a central object of interest - it carries attributes, which are updated as new packets that belong to the flow arrive. The goal of the classification is to assign a label to each flow. The choice of attributes is a

decision that designer of such algorithm is bound to make. Values of attributes for different protocols are usually not inherent, which requires one preprocessing step to measure the values on a testing data. For example, if our attribute of choice is packet size, we need to measure it for all protocols we want to be able to classify, since the values are usually not a result of a design choice of protocols and are not obvious. However, there may still be a correlation between packet size and protocol, which can be used for the classification. Indeed, it has been shown that flow features based on packet sizes and inter-arrival times are capable of preserving a lot of information about flow content. For example, a study by Bissias et al. [10] introduces algorithm, that is able to identify destination of SSL encrypted traffic. It is based on packet sizes and packet inter-arrival times and achieves 23 % accuracy. However, subsequent study by Liberatore and Levine [38] shows up to 90 % accuracy on larger dataset. This shows that flow features can preserve a lot of information about content and are useful also outside of protocol classification.

The correlation between multiple features and protocols is not known in advance, and in order to solve this, the researchers usually rely on known and tested machine learning algorithms and techniques. This process can be split into three distinct phases:

1. obtaining representative data
2. selection of flow features
3. selection of machine learning algorithm

The first problem relates to a broader topic of methodology in traffic classification discussed in greater detail in 1.4.

Two most used features have historically been packet size and packet inter-arrival time. However, their implementation may vary in different algorithms. Maximum packet size and average packet size are features with most discriminative power [39], but whole flow needs to be captured before they are determined. Apart from these, many other features can be used - a list of 248 flow discriminators can be found in [45].

Design of classification algorithm in many research papers comes down to selection of one of the many existing machine learning (ML) algorithms. Researchers use off-the-shelf algorithm and focus on applying and tuning specific technique rather than looking inside the classification process and understanding where the descriptive power comes from (see Related Work in [39]). This approach allows reuse of knowledge of machine learning and

allows us to compare multiple algorithms and choose the best - for papers comparing machine learning algorithms see [31] [58] [12] [46]. However, a few problems arise with this. One of them is that existing ML algorithms usually do not support early identification and all packets of flow are required to be captured and measured before the classification is possible or precise enough.

Based on the output, we can divide algorithms into two distinct classes - deterministic and probabilistic. The output of a deterministic algorithm is usually exactly one label. On the other hand, the output of probabilistic algorithm contains probabilities, with which the flow contains a certain protocol. Using these the flow is labeled with the protocol that was assigned the highest probability following maximum likelihood strategy. This approach is highly favored, because a confidence threshold can be set and the output can be labeled *Unknown*, if the confidence is not high enough, which makes for a more robust solution.

Machine learning algorithms used in traffic classification are examples of either supervised learning or unsupervised learning. Supervised learning solves problem of classification [2], whereas clustering algorithms are examples of unsupervised learning.

1.3.1 Supervised learning techniques

Supervised learning algorithm builds a structure, that assigns predefined labels to new observations. Input consists of labeled training data, which is used to build this knowledge structure in a learning phase. Term *supervised* is used due to the fact that training data is pre-classified. Most used supervised learning algorithms include Naïve Bayes classifier, C4.5 Decision Tree and Support Vector Machines (SVM).

Moore and Zuev [44] use bayesian analysis techniques and show, that even Naïve Bayes classifier can achieve, with few modifications, very high accuracy. First enhancement is use of kernel density estimation. Whereas Naïve Bayes classifier estimates each discriminator by fitting Gaussian distribution over the data, kernel estimation uses kernel methods to estimate real density (section 4.3 in [44]) - see figure 1.1. The second enhancement is the use of Fast Correlation-Based Filter described in [61]. With both enhancements, Naïve Bayes classifier achieved over 96 % accuracy.

2. As defined in terminology of ML, i.e. problem of identification to which class a new observation belongs, given the training set of labeled observations. This is stricter usage of the term *classification*, since *network traffic classification* - as used throughout this thesis - usually covers also clustering techniques.

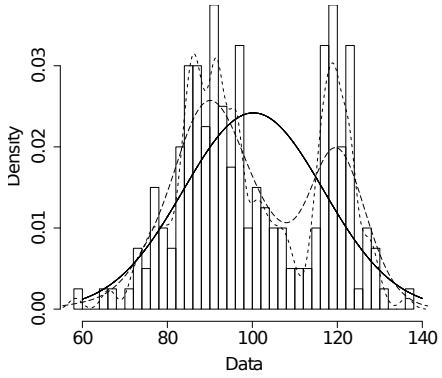


Figure 1.1: Different kernel estimations depending on smoothing parameter (bandwidth) [44]

C4.5 algorithm is used for traffic classification in [40] together with correlation feature selection (CFS) and genetic algorithm to select features, to get 88.89% accuracy. In [58] C4.5 algorithm is compared with Bayes Network, Naïve Bayes and Naïve Bayes Tree algorithms and is found faster in classification speed . Rather surprisingly, feature reduction using Correlation-based Feature Selection and Consistency-based Feature selection proved to have little impact on precision, yet significant on speed of classification.

Kim et al. [31] compare nine machine learning and traffic classification algorithms and conclude that SVM classifier consistently outperformed others. To increase robustness and mitigate effect of what is in ML known as problem of overfitting, they randomly sample flows from each dataset for training. Indeed, overall accuracy doubled compared to the case when only trace from one network was used for training. The achieved accuracy of SVM tested on different trace than the trace it was trained on was 49.8%. Using random sampling across traces, overall accuracy increased to 94.2%. This also shows that powerful ML algorithms can be sensitive to which network flow originates from and reminds us of robustness and care during evaluation.

1.3.2 Unsupervised learning techniques

Unsupervised machine learning algorithms also work in two phases - training and classification. During the training phase, the data is split into groups called clusters based on some similarity metric. Data does not need to be labeled for an algorithm to find clusters - hence unsupervised. However, to use unsupervised ML algorithms on classification problem, additional step

is necessary. After data is partitioned into clusters, the algorithm needs to label each cluster (or add some other labeling algorithm), so that during classification phase, the flow can be labeled with label of cluster it belongs to. The simplest and commonly used solution is to label the cluster with class of the majority of flows.

Bernaille et al. [9] compare three clustering algorithms: K-Means, Gaussian Mixture Models (GMM) and spectral clustering. One of the problems of clustering is the choice of number of clusters. Normalized Mutual Information (NMI) metric was used to optimize this parameter. Clustering was done with many values of parameter, and NMI was used to choose the optimal. Using the same logic, NMI was used to choose optimal number of packets to be used - it has been shown that too many packets actually bring more noise than discriminative power. Multiple labeling heuristics are compared and the most favored combination is GMM with TCP port numbers, which naturally outperforms flow majority heuristic. However, the improvement over classical port based classification is questionable, since ultimately labeling still depends on them.

Bacquet et al. [4] compare five clustering algorithms: Basic K-Means, Semi-supervised K-Means, DBSCAN, EM and Multi-objective clustering approach using Genetic Algorithm (MOGA). Results show that MOGA outperforms other clustering algorithms in both speed and precision. It uses basic K-Means algorithm for clustering and genetic algorithm for selection of features and number of clusters. Fitness function takes into account cohesiveness and separateness of clusters, number of cluster and features - lower is better. The versatility of genetic algorithm for optimization is apparent - the set of solutions converges to Pareto front and final solution may be chosen as optimal combination of several criteria.

1.3.3 Hybrid algorithms and multiple classifier approach

Researchers quickly hit the limits of such approach despite using state-of-the-art machine learning algorithms and techniques. To push the boundary further, attention has shifted from simple ML algorithms to hybrid algorithms and multi-classifiers in the last years.

Bar-Yanai et al. [5] proposed hybrid algorithm to combine advantages of two ML algorithms. K-nearest neighbors is a very simple classification algorithm, which achieves high accuracy - above 99 %, as claimed in the article - even for $k = 1$. However, complexity grows linearly with training set size [31] and so this approach does not scale well. To overcome this issue, K-Means clustering algorithm is used to segment data into several classes.

During training phase, flows are assigned to the clusters . During verification, flow is first assigned to cluster and then K-nearest neighbors algorithm labels it only using neighbors from within the cluster. This minimizes running time of algorithm and brings it close to the K-Means, yet preserves accuracy of K-nearest neighbors. To prove the efficiency of such scheme, this hybrid algorithm was implemented and deployed on realtime embedded environment of Cisco's classification platform. Results were more than promising - accuracy of hybrid algorithm was practically the same as K-nearest neighbors, while running time was very close to faster K-means. The importance of this paper lies in demonstrating that even though clustering algorithms lack accuracy, they are robust and can be used to split classification problem into several subproblems to decrease complexity of the task.

A study by Szabó et al. [55] provides additional insight into classification vs. clustering problem. Firstly, they evaluate robustness of clustering algorithms by cross-checking their performance on samples from other network. The performance drop was smaller in case of clustering algorithms compared to other classification algorithms, so they are less sensitive to network changes and other environmental noise. This result is consistent with findings of Bar-Yanai et al. [5]. Next, combination of clustering with classification methods is studied. There are two ways to combine them: (i) classification with clustering information, where result of clustering are fed as additional feature to classification algorithm, and (ii) model refinement with per cluster based classification, where each cluster has its own classifier trained to classify the traffic, that falls within the cluster. Evaluation shows that both methods show improvement over simple case, but out of these two approaches, the per cluster based classification performed better. Another contribution is the introduction of granularity levels. At the lowest level, there are features that can be measured on per-packet basis. More coarse are features measured on flow-slices, e.g. intervals containing of certain number of packets.The most coarse-grained features are flow-level features, that can be measured only after the flow is finished, e.g. number of transmitted packets. The classification then runs on all three levels in parallel and results from higher level are fed to lower level. Two possible implementations are compared: either the results of classification or the results of clustering are passed from higher level, and latter is found preferable.

Another way to improve known algorithms is multiple classifier approach. Idea behind it is that optimal combination of results from multiple classifiers can yield better results than any single one of them. This is again a well studied problem in machine learning [32][59] applied to traffic

classification. Ideal algorithm should compensate for weaknesses of each of classifiers, but preserve their strengths. Centerpiece is called a combiner, which collects results from each of the classifiers. Based on the input, combiners can be divided into three types:

Type 1 collects the most likely results from each classifier
Type 2 accepts results sorted by their likelihood from each classifier
Type 3 accepts vector of probabilities for each class from each classifier

Probably the simplest combiners are Random Selection, where output of combiner is randomly selected from outputs of classifiers, and Majority Voting, where each classifier has one "vote". Other combiners are based on bayesian probability theory, Dempster-Shafer theory of evidence etc. Good comparison of combiners including Behavior-Knowledge space and Wernecke's method can be found in [17] and in [13]. Both studies confirm that resulting algorithm has better accuracy than the best classifier, however, practical implementation and evaluation of performance are not considered. Even though combiners themselves do not require a lot of computational resources, multiple classifier running in parallel may ultimately prove impractical.

1.4 Open problems in methodology of traffic classification

Unfortunately, methodology in area of traffic classification is one of the main problems that researchers struggle with. Ideally, research should be reproducible and results from different papers should be comparable. However, this is not the case. As Szabó et al. state in [56]: "This situation results in such anarchy that papers can state nearly anything about their introduced method as there is no chance to check it by others or verify with a commonly known and accepted reference test." and Salgarelli at el. also urge research community to "find an objective and scientific way of comparing results coming out of different groups " [51].

This situation is caused by two fundamental problems:

Lack of shareable datasets (with full payload) is mostly caused by laws and policies, that prohibit such sharing. Analysis by Zhang et al. [62] shows that in 64 research papers more than 80 different datasets were used. Note that this is not the case for payload-stripped data - numerous anonymized datasets without payload are available. A very innovative approach to lever-

age this was proposed by Géza Szabó in cooperation with Ericsson Hungary Ltd. and High Speed Networks Laboratory at the Budapest University of Technology and Economics, when he developed User Behavior Based Traffic Emulator. This emulator takes data with potentially anonymized packets stripped from payload and generates pcap with full payload by emulating input traffic. A link to the online version of this tool is provided on http://www.crysys.hu/~szabog/.

No set methodology in data pre-classification and lack of agreement on metrics to be used for performance evaluation [51]. Pre-classification is a chicken-egg problem: if we want to train our classifiers and benchmark their performance, we need to establish a ground truth about our data - we need to classify it. Ideally, we would like to avoid using another specific classifier, but this is usually the case and DPI methods are the most popular as already mentioned in section 1.1. However, there is no guarantee that a benchmark classifier will never make a mistake, and this may be a source of deteriorated precision or biased evaluation. One way to cut this problem is to mark packets with an identifier of generating application [56], although this requires access to the source of traffic. Generally, this restricts data gathering to closed LANs with potentially specific traffic and this method can hardly be used on backbone links with better sample of real-world traffic.

Several solutions to the mentioned problems with methodology were proposed in [56]. First, we may try to compare classifiers on the same dataset. However, this requires publicly available implementation of methods being proposed by research groups and this is typically not the case. While several studies present comparison of algorithms [3][55][31][39][58], these still have little value, due to the fact that they are comparing machine learning algorithms applied on network traffic, rather than proposed traffic classification algorithms proposed by others. Therefore, the performance of SVM algorithm differs in published papers depending on used features, preprocessing steps etc., and this is not taken into account due to lacking implementation of proposed methods. A problem with lack of available implementations is acknowledged by Dainotti et al. in [18] who implement community classification platform traffic Identification Engine (TIE) to allow easier implementation and fair evaluation and comparison of proposed classification algorithms. TIE is written C, targeted to UNIX operating systems and supports multi-classification systems and online classification.

Another proposed solution is an introduction of new traffic capture library, which would provide interface to recorded traces and control which

features of traffic are made available to users [51]. This would potentially mean that service providers might be able to provide researchers with data measured from real-world traffic while retaining control over them. Lastly, better anonymization methods of traffic traces could be developed, that would preserve application protocol specifics but scramble user data - a soft-anonymization.

Chapter 2

SPID - Statistical Protocol Identification Algorithm

Statistical Protocol Identification Algorithm (SPID) is a novel traffic classification algorithm developed by independent network security researcher Erik Hjelmvik and introduced in [25]. It is built to use statistical attributes on both packet and flow level. Attributes range from traditional flow features based on packet sizes to attributes inspired by DPI methods. During classification attributes of flow are measured and build *protocol models*. These are compared to the protocol models from database using *Kullback-Leibler divergence* - see figure 2.1. Proof-of-concept implementation written in C# with UI in .Net can be found on Sourceforge [24]. Both binary and source code is available, though source code proved difficult to compile due to missing assemblies. However alternative implementation is available in C++ [1].

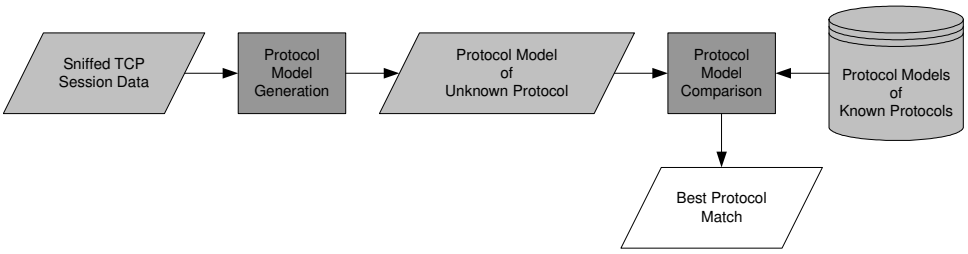


Figure 2.1: Overview of SPID classification [25]

Design of SPID is somewhat different from other algorithms based on statistical flow features. Instead of application of known ML algorithm, it has been designed from ground up with following goals:

1. Small protocol database size
2. Low time complexity
3. Early identification of the protocol in a session

4. Reliable and accurate protocol identification

This shows that unlike majority of research it has been designed with practicality in mind. It addresses major deficiencies of other algorithms: motivation for small database and small time complexity is ability to run SPID even on embedded network devices with limited hardware resources. Early identification is also necessary requirement for an algorithm to be deployed in real network and not all ML-inspired techniques allow this. For these reasons will SPID be an inspiration and benchmark for our algorithm and we will therefore dedicate to it the rest of this chapter.

2.1 Overview of SPID

Identification of flow in SPID is done by comparing flow model with protocol models built in training phase. Model of flow or protocol is a collection of several attribute fingerprints, which store values of attributes in a predefined format. Because SPID is a statistical algorithm, so are the fingerprints - each fingerprint stores a probability distribution of an attribute, rather than a single value. Measured data for each attribute is stored in fingerprint in a counter vector, from which the probability distribution is calculated afterwards. Fingerprints are created by so called attribute meters by frequency analysis of various features of payload and flow - see figure 2.2.

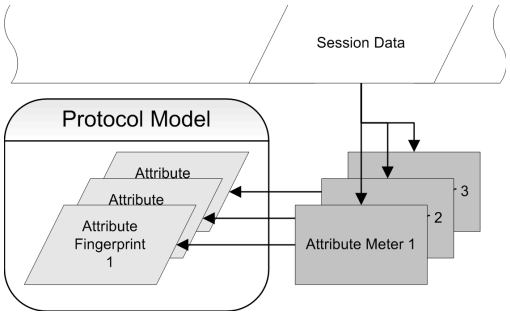


Figure 2.2: Generation of protocol models [25]

An example of attribute meter is ByteFrequencyMeter, which simply counts occurrences of bytes in payload, that are stored in fingerprint counter vector afterwards - see figure 2.3. This vector stores count for all 256 bytes that could occur in payload of flow or protocol. Probability distribution of bytes should be evenly spread out for encrypted traffic, while plaintext protocols will show different patterns.

Index	0	...	80 ('P')	81 ('Q')	82 ('R')	...	255
Counter vector	7689	...	1422	502	1001	...	3276
Probability vector	0.026	...	0.004	0.002	0.003	...	0.011

Figure 2.3: Example of attribute fingerprint: ByteFrequencyMeter [25]

Counter vector is normalized to a probability distribution after measurement and used for a comparison. Values of counter vector store the number of times that packet of flow has caused corresponding attribute meter to return that particular index of vector. Probability distribution is a normalized counter vector with values that sum up to 1.0. Length of fingerprints was chosen to be 256 for all attributes, and even though this is the most natural choice considering output of some attributes, some other length could be used, perhaps even different lengths for different attributes.

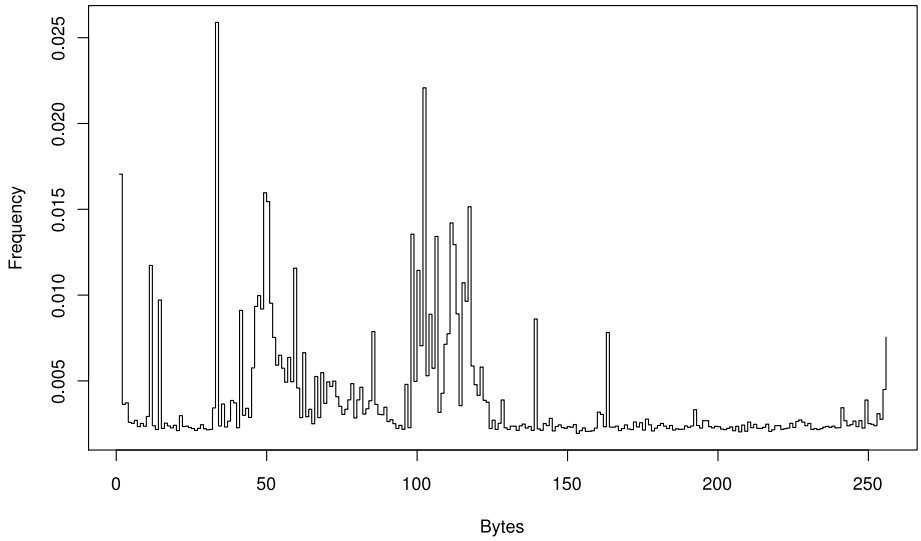


Figure 2.4: Byte frequency histogram for HTTP

A proof-of-concept application contains 33 different attributes, that measure both statistical properties of flow and payload. A comprehensive list can be found on project's wiki [23]. Some of them are simple, others cap-

ture more advanced properties that include direction and request-response behavior of protocols. Few additional attributes proposed for SPID can be found in [34].

2.1.1 Generation of models in training phase

Given the nature of SPID, there is no difference in implementation between models representing trained protocol, eg. HTTP, and model representing currently observed flow. Due to this, original paper uses term *protocol model* for both of them, even though first aggregates measurements from all packets from all Flows that contain given protocol, and second just stores measurements of to-be-classified flow.

In training phase, SPID expects pre-classified data, that is used to create protocol models for each application layer protocol. During learning, algorithm needs to know in advance to which protocol each packet belongs, so it can aggregate results from all of them regardless of source and destination of packets. We can say that during training phase, packets are aggregated on per protocol basis. For example fingerprint corresponding to ByteFrequencyMeter in protocol model of HTTP would store counts of all bytes of all HTTP sessions in training data.

Each packet is an observation. When a new observation is received, it is measured by all attribute meters and each of them update its fingerprint in protocol model by incrementing values of counter vector. In original implementation each attribute meter returns only set of indices which values should be incremented after each observation.

2.1.2 Classification

In classification phase, packets are grouped together into flows and each flow has its own protocol model. When new packet arrives, its corresponding flow is determined and attribute meters update fingerprints in flow‘s protocol model. Actual classification is performed by comparing probability distributions of measured attributes of observed flow with attributes of known protocol models. Metric used to compared them is Kullback-Leibler divergence - also known as relative entropy - introduced by S. Kullback and R.A.Leibler in [35]. KL divergence is a metric defined on probability distributions of the same length and its output ranges from 0 (identical distributions) to ∞. Also note, that it is not symmetric and hence not a proper metric.

Value of KL divergence specifies how much information is needed to

$$D_{KL}(P||Q) = \sum_i P(i) log_2 \frac{P(i)}{Q(i)}$$

describe probability distribution P given probability distribution Q - or in other words, how much information is lost if P is approximated by Q. In case of SPID, using KL divergence as a metric on fingerprints will tell us how much information is lost if currently observed flow is approximated by a model of known protocol.

During classification, fingerprints of corresponding attributes of both protocol models (one of the current flow and the other of a known protocol) are compared with KL divergence metric and the average value across all attributes is used to determine results. Matching protocol is the one with lowest average KL divergence, unless it does not pass a threshold, in which case the result is *Unknown*. This prevents incorrect classification of flows that cannot be classified due to missing information from training phase. Threshold is set empirically, original study mentions that value 2.25 was found to be a good compromise. Lowering this value will force algorithm to insist on better match, while more flows will be classified as *Unknown* (and the opposite for higher values). However, we believe SPID should not be optimized with regard to this parameter, since different users may have different goals with regard to false positives and classification ratio.

2.2 Analysis of SPID

Preliminary results published in the original paper [25] suggest that SPID is a very powerful classification algorithm, meaning it has a tendency to fit training data very precisely. Evaluation of performance was done on five protocols: BitTorrent, eDonkey, HTTP, SSL and SSH. In all cases, precision was 100 %. Time complexity of SPID has not been evaluated or compared to other machine learning algorithms.

High precision is a result of careful choice of over 30 attributes - each attribute was chosen to capture a different aspect of protocols. Using so many attributes allows SPID to fit data very closely and achieve high precision. However, this has also several drawbacks. Even though classification metric and structure of algorithm is simple, sheer number of attributes means performance overhead for both memory and CPU usage. Each attribute has a counter vector with 256 integers - if each integer allocates 4 bytes of memory, 33 attribute fingerprints for a single flow will take more than 33 kilobytes. For classification, another vector with 256 floats representing

probability distribution is needed - if this is also stored in fingerprint (as is the case for reference implementation), it increases allocated memory size for one flow at least twice. This probability vector can be calculated from counter vector each time algorithms needs it, but this would stress CPU. In our implementation, normalization of counter vector to probability distribution was the biggest bottleneck.

With the number of attributes CPU usage increases, too - KL divergence has to be calculated as many times as there are attributes. This is not a problem, if we first capture packets of the observed flow and classify them only after we have enough data. However, if we demand on-the-fly classification and result for each flow as early as possible, we may want to calculate KL divergence after every packet (or after n packets for small n to get better performance).

2.2.1 Peaking effect

There is a more fundamental problem with high number of features, in machine learning known as a peaking effect. If we plot a graph of classification accuracy with respect to the number of features, we will observe that as the number of features increases, accuracy increases too - but only until it reaches a peak and starts decreasing, as shown on figure 2.5. Key to this counter-intuitive phenomenon is a lack of error monotonicity of classifier trained on multiple samples. Given the feature set and knowledge of feature's distributions, Bayes error (error of optimal classifier for a given feature set) is monotone for a single sample: if A and B are feature sets and $A \subset B$, then $\varepsilon_B \leq \varepsilon_A$, where ε_A and ε_B are error rates for A and B. However, if $\varepsilon_{A,n}$ and $\varepsilon_{B,n}$ are error rates for classifier trained on n samples, $\varepsilon_B \leq \varepsilon_A$ may no longer be true. In fact, if $E[\varepsilon_{A,n}]$ and $E[\varepsilon_{B,n}]$ are expected error rates, it may be the case that $E[\varepsilon_{B,n}] > E[\varepsilon_{A,n}]$ [53]. Intuition behind this is simple: when first features are added, we give more information to the classifier and thus increase its accuracy. However, with more features, not all of them are useful and some may not even add any additional information - in such case they act as sources of noise, which decrease the accuracy of the classifier.

Due to this effect designer of classification algorithm may want to search for a subset of features which results in optimal accuracy. However, this is a hard problem, where the best algorithms only try to improve the exhaustive search. Specifically, greedy search cannot be used to find optimal subset of features. If $F = \{F_1, \ldots, F_k\}$ is s set of features and $\{S_1, \ldots, S_n\}, S_i \subseteq F$ are all possible subsets of F, there exists an ordering on $S_1, \ldots, S_n$ (with

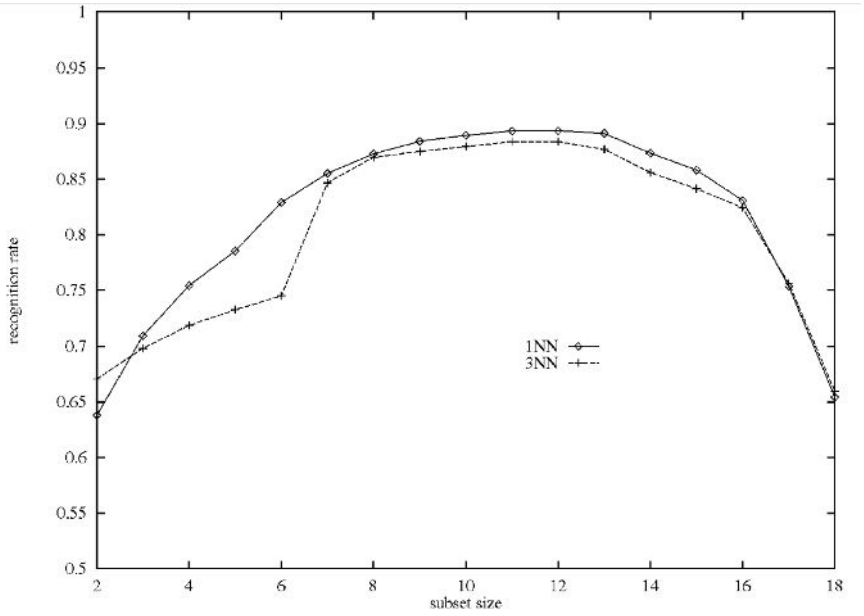


Figure 2.5: Peaking effect of k-NN classifier and SFFS feature selection algorithm for finding optimal subsets of k features [29]

respect to classification accuracy), on which algorithms searching for the best k-element subset by successively enlarging the best j-element subset for $j = 1, 2 \ldots k - 1$ will not find the optimal solution [16].

Given the high number of features in SPID, all of which are manually picked by the author, we may suspect that few of them are actually spoiling accuracy. Indeed Köhnen et al. conclude that "some attribute meters are not very accurate and adultered results" [34].

Chapter 3

Self-optimizing traffic classification framework

We introduce a novel approach to traffic classification in this chapter. We propose a classification framework, that represents a whole class of classification algorithms. We let the framework generate optimal algorithm for a given training data, instead of having algorithm with fixed parameters. We aim to propose a simple solution which could be easily extended, while generating simple and efficient algorithms.

The framework represents a different approach in design of classification algorithms. Classification procedure is always considered in context of chosen features, pre-classification steps and other implicit parameters, and algorithm is trained and optimized as a whole. This is in contrast with usual approach, where algorithm is designed to be fixed and does not change during the training phase.

3.1 Requirements for classification framework

We devised several requirements for classification framework prior to the design phase. These requirements should guarantee practicality and usability of developed solution. Following these we hope to avoid a common situation, when usability of algorithm is negatively influenced by decisions made in the design phase. Requirements are as follows:

High precision. Classification algorithm should achieve at least the same precision as SPID while improving other parameters.

Low memory footprint. A goal is to decrease the amount of information that is stored for each flow during classification. This is one of SPID's disadvantages as mentioned in chapter 2. As stated in the original paper, one of the key design requirements was a small database size. However, each flow has to maintain the same amount of information as the whole database due

to the nature of SPID. Memory consumption will therefore get prohibitively high in high-speed networks with this setup.

On-the-fly classification. The algorithm should classify the flow as early as possible, rather than at the end of the flow. This requirement is also highly motivated by real-world usage of classification algorithms. In order to provide Quality of Service, shape traffic and build security mechanisms on top of flow classification, early identification of flow is crucial.

Simple design and modularity. We should design the framework the be simple to understand and modular to promote extensibility and customizability. This should help administrators, who often have a legitimate requirements that are specific to their network and usage. A good example of such effort is TIE - community oriented traffic classification platform [18], that allows and promotes extensibility of algorithm with plugins.

Automated parameter tuning. The framework should expose all parameters to optimization engine, which would choose the optimal values of parameters in the training phase. Resulting algorithm would be one instance of all algorithms that our framework represents. This is the opposite of usual method of parameter tuning. Each algorithm has many hidden parameters, that are almost always manually chosen. The designer usually picks empirically tested and proved values of parameters, that leverage some internal knowledge about the traffic. These can include minimum and maximum number of packets read for each flow, choice of metric, preprocessing steps, thresholds etc. Interactions between these parameters may become complex and the choices may become time consuming. Therefore our requirement is to make this process automatic and more efficient.

3.2 Proposed solution

The framework consists of two main components: *Optimizer* and *Classifier*. Classifier represents the classification algorithm - its internal structure is configurable and its interface provides methods for training and classification. Role of the Optimizer is to find the optimal structure of the Classifier using methods for configuration and classification - see figure 3.1. To achieve this it repeatedly reconfigures Classifier and computes fitness of such configuration. Fitness is computed by training the Classifier on training data and classifying validation data. Multiple outputs from classifica-

tion can be included in fitness, such as precision, memory consumption and overall time.

Internal structure of the Classifier consists of tree hierarchy, that splits the problem of classification into several subproblems. Internal nodes of hierarchy make decisions based on clustering and forward flows to a different subtrees. Leaf nodes perform actual classification based on given metric and features. Each leaf node is able to classify a certain set of protocols and can use different metric and features than others. Each internal node - a clusterer - uses a different feature to split the traffic.

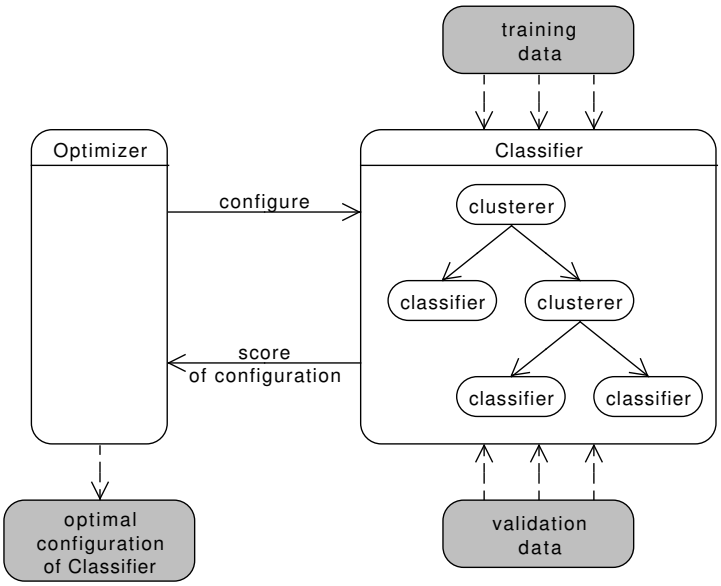


Figure 3.1: Overall structure during training phase

Note the ambiguity of term classifier - we use it to describe the whole algorithm as well as to refer to leaf nodes, that compute the actual output. To mitigate this we will use word Classifier with capital C for overall algorithm and lowercase variant shall refer to a leaf node in the tree hierarchy.

3.2.1 Training phase

The framework runs in two phases - training and classification. In the training phase Optimizer controls the execution flow, as it is searching for the optimal configuration. It repeats following steps:

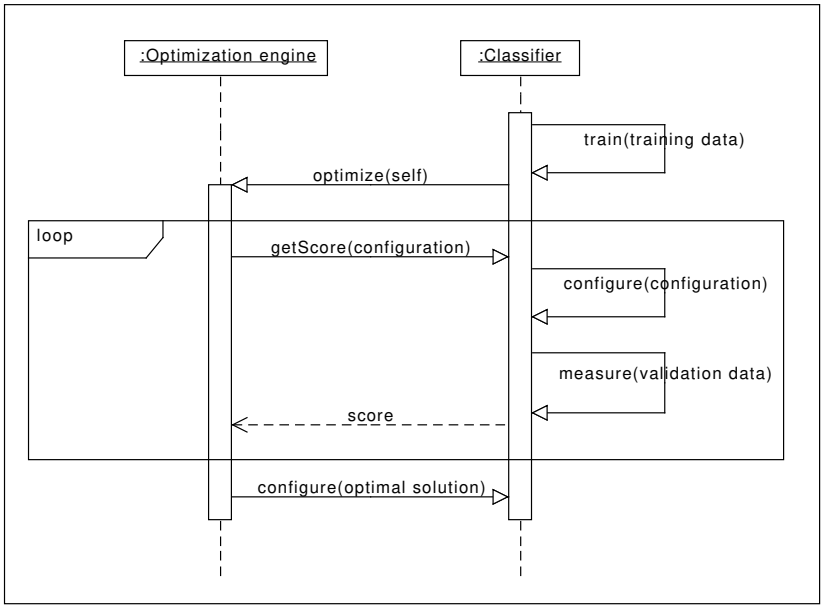


Figure 3.2: Interaction between Optimizer and Classifier

1. Generate a possible configuration of Classifier
2. Reconfigure Classifier with generated configuration
3. Train Classifier on training traffic
4. Classify validation traffic
5. Use results of classification to compute the fitness of configuration

Both training and validation traffic need to be pre-classified to provide a ground truth for the algorithm. Using this information Optimizer finds an optimal solution.

3.2.2 Classifier optimization

Optimization engine is introduced into the classification framework to address the issue of parameter tuning. Following modular design it is a standalone component that interacts with Classifier trough defined interface. Classifier provides Optimizer with methods for its reconfiguration and classification.

Interaction between Classifier and optimization engine is described in figure 3.2. Classifier calls optimization engine on itself after it gathers in-

formation from training data. The engine may then repeatedly try different configurations of classifier and query it for the score of each configuration. Classifier measures the score by configuring itself according to the configuration passed from the Optimizer and classifying validation data. Score can be based on several aspects including precision, speed, memory efficiency, weighted scores of used features etc.

Advantage of this approach is that the scope of optimization depends only on our representation of classifier configuration. This allows us to optimize the classifier with respect to its structure.

Optimization engine is based on genetic algorithm in our proof-of-concept implementation. Different implementations of framework may have more sophisticated optimization engines based on different optimization algorithms and heuristics and with tuned parameters. Further investigation of the efficiency of genetic algorithm as optimizer, comparison of different solutions or design of problem-specific optimization heuristics is left for future work.

3.2.3 Design of classifiers

Each classifier is essentially a SPID-like algorithm with its own features, metrics and protocols it is able to classify. Each feature is represented as a probability distribution. Metric computes similarity score between two probability distributions and final decision is the protocol most similar to the classified flow.

We decided to build on results achieved by SPID, since its core principles proved to be both simple and effective. Representation of features as vectors with probability distributions is what we believe to be the primary reason of its precision. Reason for this is twofold:

- it is based on discretization. In their study [39] Lim et al. investigated effects of discretization of features on precision of classification algorithms. They state that even unsupervised equal-interval-width discretization is essential for traffic classification.
- representation as probability distribution exhibits high degree of robustness. Specifically, it is likely that measured values for similar traffic will be close together. Probability of this happening is given by the probability distribution of a given feature.

However, in contrast with SPID we do not enforce that the length of vectors of all features must be the same.

Choice of Kullback-Leibler divergence as similarity metric between the probability distributions is not further discussed in the original paper. For this reason, the framework should not depend on Kullback-Leibler divergence, but allow other metrics to be used.

3.2.4 Feature reduction and memory efficiency

In order to use fewer features, we divide problem of classification into several smaller subproblems using clustering. Each subset of protocols has its own classifier trained to distinguish between them.

When a new packet arrives, only feature which we use for clustering is measured. Packet can go through multiple clusterers depending on the depth and structure of the tree hierarchy. At the end it is classified by a classifier specific to the final cluster.

As example, consider a set of protocols that contains both encrypted and plaintext protocols. Features, that are most useful in classification of plaintext traffic are usually useless in case of encrypted traffic, and vice versa. However, we can measure entropy of packet payload and assign packets into two clusters - one for encrypted traffic and the other for plaintext. Each cluster has its own optimized classifier. The one for plaintext traffic could use first few bytes of the first packet to classify a flow, since application protocols usually have some header. On the other hand, classifier for encrypted traffic would use different attributes that do not rely on information about the payload, such as byte frequencies or packet inter-arrival timings. Notice, that if we measured payload of encrypted packet, we would be just adding unnecessary noise to the classifier. When classifier is trained specifically on plaintext traffic, it can be more efficient and not depend on heuristics useful for encrypted traffic.

Partitioning of protocols into groups allows us to change the objectives of classifier. Its original goal is to evaluate similarity of flow with known protocols. Resulting label is then the protocol most similar to the flow with respect to some metric. This favors such a choice of set of features, that best describe given protocol. In case of many protocols this results in many features, some of which may not be relevant.

However, when we partition classification into several smaller subproblems, objective of a classifier starts to change. Given a small set of protocols, it does not have to evaluate similarity of flow with each of them. Rather, it should decide which one is most probably the correct one. This simplifies the problem for classifier, because it can now focus more on differences between the protocols. Given two protocols, designer can come up with dozen

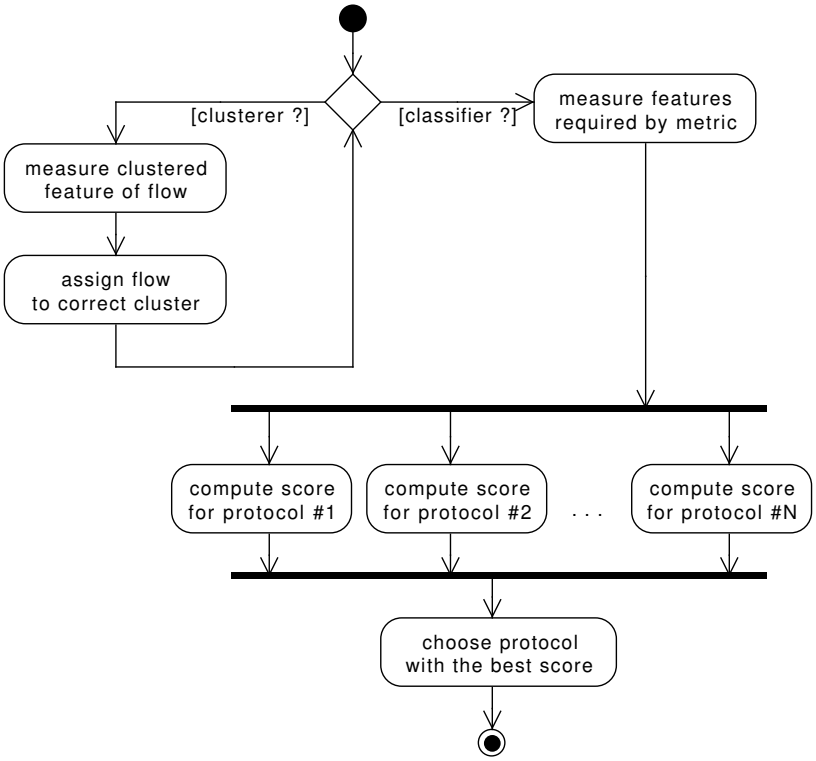


Figure 3.3: Activity diagram of classification

of features that best describe them. But if we only need to distinguish between the two of them, one feature in which they differ most suffices. To generalize, a combination of optimal features for each protocol is not optimal combination of features for classification of these protocols.

Choice of clustering in favor of other method to partition traffic is motivated by the need for robustness. Assigning a packet to a group of protocols is a discrete classification, that has a very high impact on resulting decision of overall classifier. If the packet is assigned to the group of protocols it does not belong to, the best that the classifier for a given subproblem can do is to label it as unknown. It is therefore more useful to use less precise, but more robust clustering.

This approach has advantage over multi-classifier approach with respect to both memory consumption and CPU usage. Parallel evaluation of multiple classifiers is more resource intensive than choice of one of them

based on clustering. Compared to the multilevel approach, we do not feed the results of clustering into classification algorithm, since the decision is implicit.

From implementation standpoint, there are two ways to divide data into multiple clusters. Our proposed framework uses a hierarchy of clusterers, each of which uses only one feature. However, we could cluster the data in a single step by merging multiple features into multidimensional data. Although this reduces the number of clustering steps to one, it has an impact on the precision and the speed of clustering. Difficulty of clustering for high-dimensional data rapidly increases due to large amount of outliers, different densities in different dimensions and lack of defined shape [52]. For these reasons our framework is built on the concept of hierarchy of clusterers.

3.2.5 On-the-fly classification with feature extractors

Each feature has a specific procedure for updating the value, computing the final value, different requirements on the amount of captured data for feature to be valid etc. We introduce feature extractors to our framework to encapsulate all the parameters and procedures for each feature.

Features used for classification must support continuous updates of measured values as new packets arrive to allow on-the-fly classification. Requirements for features used for clustering are different - they must not be high-dimensional and the final value must be calculated using the least amount of packets possible. Each cluster defines a set of required classification features, which is unknown before the flow is assigned to the cluster. If a clustering feature required N packets before assigning the flow to the correct cluster for classification, Classifier would have to buffer the first N packets before making a decision. After the cluster is known, only the classification features are measured and continuously updated without caching any data. Feasibility of splitting the traffic using only the first packet was studied by Dorfinger et al. in [20], where the traffic is split to encrypted and unencrypted based on entropy with 99 % precision.

Chapter 4

Framework realization

The framework proposed in the previous chapter represents a general approach towards classification algorithm design. We use the framework to propose a concrete algorithm, which we implement and evaluate. We also take liberty to use the implemented algorithm to investigate some aspects of classification and to get a better understanding. Therefore we make design choices that can be either omitted or optimized in other implementations. This chapter describes parts of algorithm not specified by the framework - choice of features and classification metrics.

4.1 Classification metrics

Our implementation contains two metrics: Kullback-Leibler divergence and cosine similarity. As mentioned in 3.2.3, our framework should not depend on KL divergence only. To test the optimization engine and investigate effects of using multiple metric, we also add cosine similarity metric defined as

$$\cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

Value is a cosine of angle of two vectors of an inner product space and is independent of their magnitude. This does not affect our classifier, since vectors representing probability distributions of features are normalized.

Both metrics can be used for classification, which allows Optimizer to choose the one that yields better results. Goal of our implementation is to show that the sOptimizer can choose and favor one, more optimal, metric. Investigation of other metrics and their optimality is left for future work.

4.2 Clustering features

Choice of features used for clustering directly affects how protocols are divided into groups. We implement entropy and N-truncated-entropy features to divide the traffic to encrypted and unencrypted. Newline-equality feature has been designed to distinguish between plaintext and binary protocols. None of them are represented as probability distributions, but rather as a float number, since data with high dimensionality is not suitable for clustering.

4.2.1 Entropy

We calculate sample entropy of payload of length N defined over alphabet `{0x00, ..., 0xFF}` of size 256 with n_i being the number of occurrences of byte i in the payload. Frequency f_i of i-th byte is defined as $f_i = n_i/N$. Sample entropy of a payload is given by formula

$$H_N^{MLE} = -\sum_{i=0}^{256} f_i log_2(f_i)$$

where MLE stands for maximum likelihood estimator. Note the difference between sample entropy and entropy

$$H(p) = -\sum_{i=1}^{m} p_i log_2(p_i)$$

of random variable X with probability distribution p. Due to asymptotic equipartition property , H_N^{MLE} converges to $H(p)$ when N tends to infinity and elements of sample are drawn independently according to p [47]. This means that H_N^{MLE} is a good estimator for $H(p)$ especially when $N >> m$. However, it may be far from $H(p)$ when $N \sim m$ or $N < m$.

Our motivation for implementation of this feature is to compare it with N-truncated entropy.

4.2.2 N-truncated entropy

This feature is used in [20] to divide the traffic to encrypted and unencrypted. As stated in the paper, it is hard to estimate the entropy on a sample, especially when number of observed values N is lower than number of possible values m. N-truncated entropy is defined as follows:

1. Generate all words of length N according to p

2. Estimate the entropy for all words based on maximum likelihood
3. Calculate N-truncated entropy as the average of estimates

This construction solves the problem of sample entropy in case when $N < m$. Recall that a sample entropy could be arbitrarily far from $H(p)$. By construction of H_N, sample entropy H_N^{MLE} is an unbiased estimator for H_N, i.e. value of H_N^{MLE} is close to H_N up to a small tolerance ϵ.

For a uniform distribution $\mathcal{U}$, when $p_i = 1/m$ for all i, value of N-truncated entropy is

$$H_n(\mathcal{U}) = \frac{1}{m^N} \sum_{n_1+\ldots+n_m=N} \left[\binom{N}{n_1 + \ldots + n_m} \times \left(-\sum_{i=1}^{m} \frac{n_i}{N} log \frac{n_i}{N} \right) \right]$$

However, computation using this formula is not feasible for high N and m. Instead, we approximate the value using first method from [47]. For each constant c, when N and m tend to infinity and N/m tends to c, N-truncated entropy can be estimated as

$$H_n(\mathcal{U}) = log(m) + log(c) - e^{-c} \sum_{j=1}^{\infty} \frac{c^{j-1}}{(j-1)!} log(j) + o(1)$$

where $o(1)$ is error term that quickly decreases as N increases. The series converges quickly and to avoid infinite loop, we stop calculating the sum when $\frac{c^{j-1}}{(j-1)!} log(j) < 10^{-12}$ for current value j, i.e. when the next term in sum is negligible.

In our implementation we calculate H_N^{MLE} for packet payload and calculate the difference from $H_N(U)$. If the difference is small, frequencies of bytes in packet payload are close to the uniform distribution (U). This would mean that the payload is encrypted.

4.2.3 Newline equality

This is a novel feature designed for the framework. The output is boolean equal to 1 if and only if the number of occurences of byte `0x0D` (carriage return CR) is equal to the number of occurences of byte `0x0A` (line feed LF). A logic behind is simple: in plaintext protocols bytes CR and LF used to denote new line are always together, so their frequencies are also the same. We introduce this metric as potentially useful for dividing traffic protocols into plaintext and binary. However, it could also be used for classification, so we implement a version of this feature that returns array suitable for

classification metric. When byte counts match, result is `[1,0]`, otherwise `[0,1]` is returned.

4.3 Classification features

We chose to implement these 6 features from SPID: AccumulatedDirectionBytes, ActionReactionFirst3ByteHash, ByteFrequency, DirectionPacketLengthDistribution, First2OrderedFirst4CharWords and FirstPacketPerDirectionFirstNByteNibbles. They were chosen according to the information from SPID's wiki [23] regarding their speed and precision. We also add two novel features NullFrequency and DirectionEntropy.

4.3.1 NullFrequency

This feature measures the relative frequency of *Null* byte in payload. *Null* byte is the most outstanding byte in several protocols, and this features leverages that. It is stored in vector as

```
[NullFrequency, 1-NullFrequency]
```

where `NullFrequency` is ratio of byte `0x00` to all bytes.

We propose this feature since many protocols have byte frequencies of *Null* byte biased. The frequency is either much higher compared to other bytes - as is the case for HTTPS or SMB 2, or close to zero as in SMTP or IMAP. Graphs of byte frequencies and values of this feature for protocols used in this work are included in Appendix A.

4.3.2 AccumulatedDirectionBytes

This feature captures the amount of consecutive data sent in the same direction. Only first three direction changes are measured, so the vector is split into four parts. Therefore sizes of the first two application layer request-response pairs are measured. Measured sizes are mapped linearly to fingerprint vector using function

$$\text{offset} = \frac{\text{accumulatedBytesCount}}{\text{byteChunkSize}}$$

where $\text{accumulatedBytesCount}$ is cumulative size of consecutive packets sent in one direction and byteChunkSize is a constant set to 64. This means

the maximum size that can be mapped is $\mathrm{byteChunkSize} * \mathrm{fingerprintLength}/4 = 64 * 256/4 = 4096$ bytes.

Other similar features in SPID use logarithmic function to map values onto fingerprint vector to get higher granularity for lower values. Decision to use linear function is not discussed by author of SPID, so we also implemented a logarithmic version of this feature with function

$$\mathrm{offset} = \mathrm{accumulatedBytesCount}^{\mathrm{exp}}$$

$$\mathrm{exp} = \frac{\log(\mathrm{fingerprintLength}/4)}{\log(\mathrm{maxPacketLength})}$$

where $\mathrm{maxPacketLength}$ is equal to 4096 and compare it to the linear version.

Chapter 5

Implementation

To evaluate proposed solution, we created a proof-of-concept implementation of proposed algorithm. It is written in Ruby, a high-level object-oriented programming language. Reason for this is practical - thanks to its high level nature and programmer-friendliness, it allowed us to quickly rewrite substantial blocks of code and we were able to test and try numerous ideas and improvements in the process. Disadvantage of this choice is the speed of processing, which makes this implementation not suitable for high-speed environments or commercial use. To mitigate some inefficiency and increase performance, we identified bottlenecks and functions that interpret spent the most time in and reimplemented them in C. These chucks of C code were embedded inline directly into Ruby code using RubyInline library.

5.1 FingerprintContainer

Classifier consists of hierarchy of clusterers, that split the set of protocols into groups. Each group has its own classifier with different features and metric. Node in hierarchy is implemented by the class FingerprintContainer, that represents a general node - see figure 5.1.

The type of instance of FingerprintContainer can be either inner node or leaf depending on its position. Inner node is configured to perform clustering and forward data to the correct cluster. Leaf node contains a list of protocols to be classified and metric, with respect to which it performs final classification of flow. Class Classifier acts as a wrapper around the hierarchy and provides a high-level interface for classification.

5.2 Classifier and Optimizer

Interaction between instances of these classes is described in figure 3.2. We optimize the work in training phase by processing the training traffic only once. This is possible because classifiers for clusters require only measured

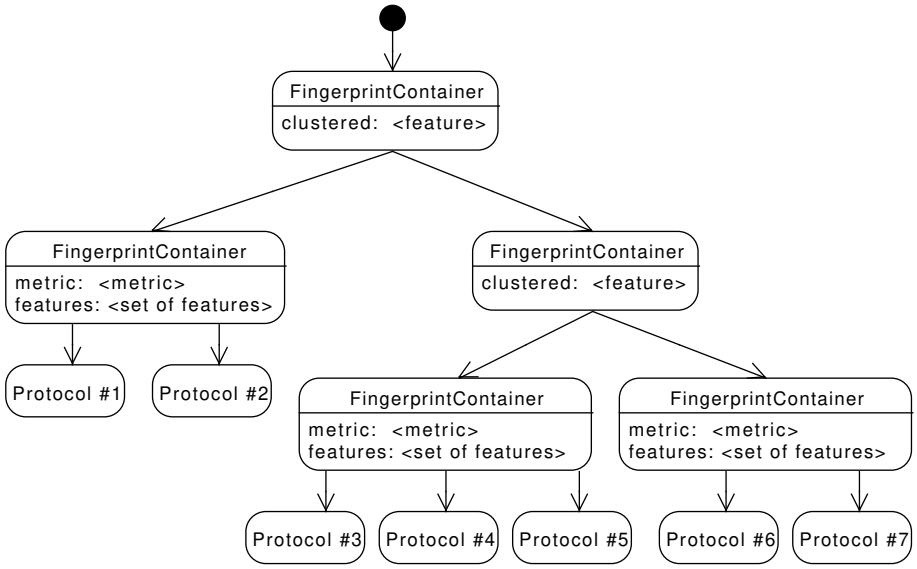


Figure 5.1: Example hierarchy of FingerprintContainer instances

features of protocols to compare them to classified flows. Therefore, we measure the values of features of protocols from training traffic once before optimization. During optimization phase, we copy measured data instead of training each configuration of Classifier on training traffic.

We do not implement separate methods to measure features of protocols. Instead, we make use of existing methods to measure features of flows. We combine values of all flows with the same protocol to get the representative value for each protocol.

5.3 Feature extractors

Each feature is implemented by corresponding feature extractor. It provides methods for:

- updating a value of feature for a given flow when a packet arrives
- calculating the final value
- merging values of flow to get a representative value for a set of flows.

This method is used to obtain values for protocols during the training phase.

- checking validity of value. Each feature may define thresholds and a method, that checks whether the value can be used for classification against the thresholds.

In our implementation we usually use thresholds to specify minimum amount of data or packets required to get a representative value. Specification of checking method allows us to express more complex conditions. An example is ActionReactionFirst3ByteHash feature, that is valid only after packets in both directions were captured.

5.4 Classification metrics

We implement metrics as subclasses of class ClassificationMetric, which defines the interface used by classifiers. This allows us to implement different metrics, as long as they provide methods specified by the interface. Three required methods are:

get_score	returns score of similarity between classified flow and protocol
is_valid?	returns true if score is above/below defined threshold and is therefore valid. Result `UNKNOWN` is returned otherwise.
choose_best_protocol	given an array of protocols and scores, returns protocol most similar to classified flow

5.5 Configuration representation

Low-level configuration of Classifier is represented by the class Gene. Optimizer based on genetic algorithm uses instance of Gene to perform mutation and reproduction. It is therefore desirable that the Gene has a fixed length and structure.

It consists of three parts:

1. feature, that is used for clustering
2. set of features used for classification
3. classification metrics

All three are unique and defined for each FingerprintContainer in hierarchy. However, not all are used. Internal nodes that perform clustering ignore classification features and metric, while leaf nodes ignore feature for clustering. To decide whether a node will be a clusterer or classifier, we add a special *Nil* clustering feature. Presence of this feature switches FingerprintContainer into classification mode.

node	Node #1					...	Node #N				
binary vector	1	0	...	1	0	...	0	1	...	1	1

Figure 5.2: Sets of features used for classification are encoded in binary vector. Chunks of this vector represent nodes in tree hierarchy, which are stored in breadth-first order.

To further simplify representation, we assume that depth of hierarchy of FingerprintContainers is no more than 3 levels high. Subset of used classification features is determined by the binary vector, where 1 represents used feature and 0 ignored - see figure 5.2. Feature used for clustering is represented as integer with 0 being the *Nil* clustering feature. Classification metric is also encoded as integer. Mutation and reproduction functions required by genetic algorithm are defined on all three parts separately.

A Dna class is added to simplify the interface of the class Gene. It is effectively a wrapper around Gene and provides methods for querying and decoding information stored in Gene. Class Chromosome is used by genetic algorithm and represents one solution. It is a wrapper around the Dna and provides methods to get score and compute fitness function of solution encoded in the Dna.

5.6 Clusterer and GeneticSearch

We do not implement these two classes in our framework, but rather use implementations from library AI4R :: Artificial Intelligence for Ruby, available at `ai4r.org`. We choose K-Means algorithm as clusterer without further investigation - it is left for future work to determine which clustering algorithm performs best. Perhaps even choice of clustering algorithm could be determined by optimization, as it can be added to the Gene as another information specific to a FingerprintContainer with no issues. However, feasibility and usefulness of this approach remains untested.

GeneticSearch class from AI4R library implements fitness proportionate selection operator, also known as roulette wheel selection. Selection works as follows:

1. Population is sorted by fitness
2. Accumulated normalized values are calculated. Accumulated value is a sum of fitness values of worse or equal solutions. After normalization the best individual has normalized fitness 1.
3. Random number R from interval $\langle 0, 1 \rangle$ is chosen. First individual, whose accumulated normalized fitness is higher than R is chosen for the next generation.

Chance of individual with fitness f_i to get to next generation is $\frac{f_i}{\sum_{j=1}^{N} f_j}$, where N is the size of population. This selection algorithm assures that better individuals will have proportionally higher chance to get to the next generation, while it may still choose some weaker individuals to prevent quick convergence to the local optimum.

Fitness and mutation function along with generation of new individuals in population are problem specific and have to be implemented before using GeneticSearch class. We implement fitness function to return average F measure of a given configuration and ignore memory usage or speed of classification. Mutation function can flip each bit of configuration with probability

$$(1 - F_n) * 0.03$$

where F_n is fitness of configuration normalized across the population, so that the highest is set to 1 and the lowest set to 0. Inverted value of fitness will result in better solutions having less mutations.

Chapter 6

Evaluation

In this chapter we present experimental results together with description of test setup and interpretation of results.

6.1 Methodology

We chose 9 protocols to test our framework: HTTP, HTTPS, IMAP, IMAPS, SMTP, SMTPS, SSH, BITTORRENT and SMB version 2. This collection includes plaintext, encrypted and binary protocols in order to test the clustering and classification performance.

Obtaining existing traffic traces with full payload proved to be difficult - to protect the privacy of users, they are not generally available. Several Internet archives provide pcap traces with full payload, but samples usually contain just one flow and are very specific, e.g. capture network traffic of a virus or network attack.

To get unbiased and representative dataset with enough flows, we captured traffic on a personal computer. A small script was running in the background, that periodically captured running applications and logged them, so we could correlate the data with captured flows. This allowed us to label the traffic with certainty and provide a ground truth without reliance on any other classification tool.

To evaluate the classification algorithm, we use *Precision* (or accuracy), *Recall*, *F-measure* and *Runtime* of classification. First three are defined as

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

where TP stands for true positives, FP false positives, FN false negatives, and Runtime is measured as overall time of classification.

We split the captured traffic of each protocol into three sets: training, validation and testing. Traffic in each set is split into files by protocol, so

Protocol	Number of flows in training set	Number of flows in validation set	Number of flows in testing set
BITTORRENT	319	6	100
HTTP	1188	6	311
HTTPS	1776	6	828
IMAP	24	5	13
IMAPS	164	6	75
SMB2	5	5	5
SMTP	27	6	14
SMTPS	31	6	9
SSH	15	6	11

Table 6.1: Structure of used dataset

we know which flow contains which protocol in each phase. Flows for each set were chosen randomly. Training set is used by classifier for training itself - measure average values of features for each protocol. Validation set is used in optimization phase to evaluate the score of generated configuration. Configured classifier classifies traffic in the validation set and the calculated score is returned to the optimizer. This is repeated as many times as new configuration is generated by the genetic algorithm, so we included only about six flows from each protocol into the validation set. After the optimal solution is found, we evaluate performance on the testing set. Structure of each set is summarized in Table 6.1

6.2 Measurements of SPID algorithm

In order to have a good benchmark for comparison and evaluation, we first measured performance of our implementation in SPID configuration - without clustering or optimization, with Kullback-Leibler divergence used as metric. Comparison of our framework with this configuration has several advantages over comparison with original implementation:

1. comparison of speed of the two approaches is possible, since no difference in performance comes from implementation

2. differences in implementation do not affect results. For example, our framework implements on-the-fly classification logic differently than proof-of-concept SPID implementation.

Feature set	Average F-measure	Runtime [s]
base	0.72658	455.05
base + lin	0.77329	598.60
base + log	0.75846	580.81
base + null	0.80397	510.12
base + lin + log	0.80259	701.26
base + lin + null	0.82099	589.67
base + log + null	0.81682	609.85
base + lin + log + null	0.81406	691.03

Table 6.2: Scores for different feature sets

We run the experiments for several sets of features. First set contains only features proposed and implemented in SPID, namely ActionReaction-First3ByteHash, ByteFrequency, DirectionPacketLengthDistribution, First2OrderedFirst4CharWords and FirstPacketPerDirectionFirstNByteNibbles - we refer to this as a *base* set. To test the effect of new features, we add AccumulatedDirectionBytes (linear version), AccumulatedDirectionBytes (logarithmic version) and NullFrequency features to the base set. Results are summarized in table 6.2 and detailed reports are included in Appendix B. When we evaluate our algorithm, we will compare it with *base+lin* feature set, since it contains only original features of SPID.

6.2.1 Evaluation of new classification features

Comparison of linear and logarithmic version of AccumulatedDirectionBytes feature shows the latter produces better results. Results are the same for all protocols except for BitTorrent. Detailed results, included in Appendix B, show that the linear version is consistently better than logarithmic. However, this is not affected by other protocols. All false negatives were classified as unknown traffic, so the difference is not produced by similarity of BitTorrent to any other protocol. As we can see on figure 6.1, values of logarithmic version are more spread, because logarithmic function is more fine grained for lower values. However, more spread out distribution results in higher divergence when compared to the value of measured flow. Therefore, a score for some BitTorrent flows exceeds the threshold and they are classified as unknown.

The score of base feature set with both versions of AccumulatedDirectionBytes included is higher than the scores for each of them. The difference is only in classification of BitTorrent. This suggests that when Accumu-

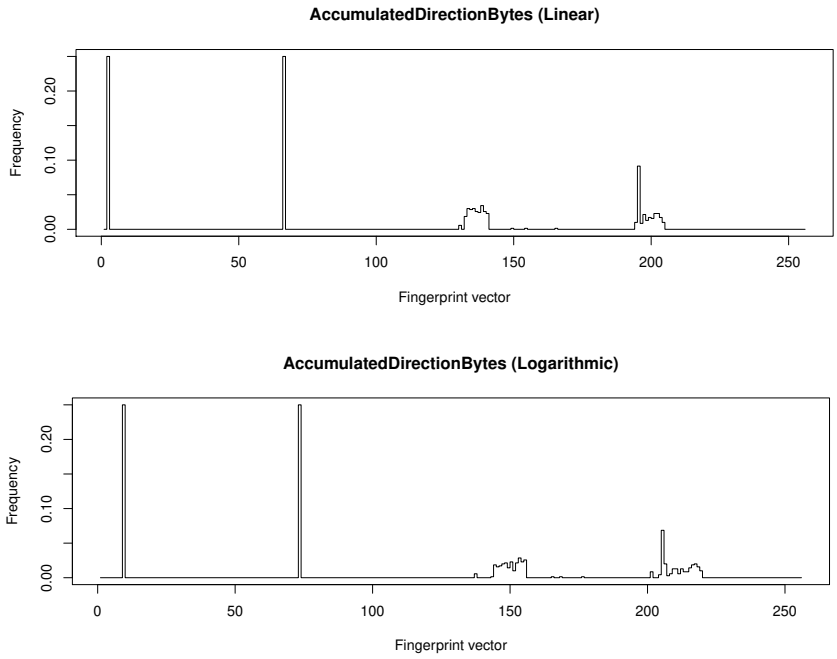


Figure 6.1: Comparison of two versions of AccumulatedDirectionBytes on BitTorrent traffic

latedDirectionBytes feature has more weight in the final score, BitTorrent can be classified more precisely. It would be an interesting topic for future research to study how different weights of features contribute to overall performance.

NullFrequency feature proved to have a significant impact on classification of BitTorrent. F-measure of base set for BitTorrent protocol increased from 0 to 0.72 for base set with NullFrequency. As discussed in section 4.3.1, protocols tend to have a frequency of Null byte biased in some way. However, thorough obfuscation of BitTorrent results in a stable frequency and provides a good discriminator.

6.3 Evaluation of proposed algorithm

To allow Optimizer search a large portion of possible solutions, we have set the size of population in each generation to 50 and the number of generations to 100. Classifier accepted also a configuration with clustering feature set to Nil, in which case no clustering was used. We allow such configura-

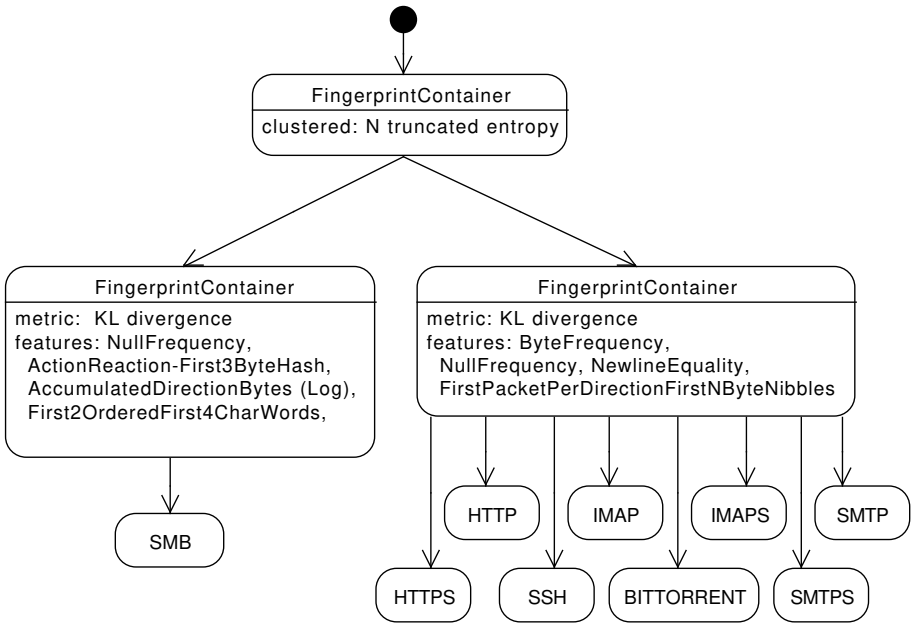


Figure 6.2: Generated solution #1

tions to test the hypothesis that splitting protocols into groups and training a separate classifiers for them results in higher precision. Fitness function was based only on classification F-measure and ignored number of features. Results generated by the Optimizer were stored in external database to allow further analysis.

Generated solution is shown on figure 6.2. Optimizer favored clustering and N truncated entropy was chosen as the clustering feature. Unfortunately, due to the low entropy of SMB 2 protocol it is the only protocol in its cluster. Optimizer reduced number of features in the second cluster, but the cluster with SMB 2 has also four features. This is unnecessary and no classification is needed - when clusterer assigns flow to SMB 2 cluster, it can be immediately labeled as SMB 2 without measuring any additional feature. However, this is not a fault in Optimizer - since the fitness function does not take a number of features into account, it found set of features that describes SMB 2 the best.

Average F-measure of generated solution is slightly better than SPID's F-measure - see Table 6.3. Training phase took more than 6 hours and 2160

Average Precision	0.8175
Average Recall	0.8518
Average F measure	0.80277
Optimization runtime	6h 20m 42.493 s
Classificaiton runtime	164.93 s

Table 6.3: Measurement of solution #1

unique solutions were generated during optimization. However, classification phase took only 164 seconds, which is a substantial improvement compared to SPID's 598 seconds. Investigation of the speedup showed that it was not achieved only by reduction of the number of features. Since we implement on-the-fly classification, optimized classifier will label the flows earlier. On the other hand, SPID needed to measure on average more packets until the resulting score dropped below the threshold and the flow was classified.

6.3.1 Evaluation with SMB 2 excluded

SMB 2 protocol has the lowest entropy out of the protocols in our dataset and clusterer correctly creates a separate cluster for it, since the gap between entropy of SMB 2 and second lowest entropy is the widest - see Table A.1. To evaluate behavior of the framework in a more natural setting without such extreme, we excluded SMB 2 from our dataset and generated second solution using the framework.

Average Precision	0.8546
Average Recall	0.8833
Average F measure	0.8185
Optimization runtime	1h 3m 42.803 s
Classification runtime	290.09 s

Table 6.4: Measurement of solution #2

As shown in Figure 6.3, Optimizer again chose N truncated entropy as the feature for clustering. Protocols are split into two even clusters - one contains protocols SSH, HTTP and SMTP that have a higher values of N truncated entropy, the other contains BITTORRENT, IMAPS, SMTPS and HTTPS. It is also interesting that both clusters use AccumulatedDirectionBytes in both variants for classification - this suggests that it is not only helpful in classification of BITTORRENT traffic as discussed in section 6.2.1.

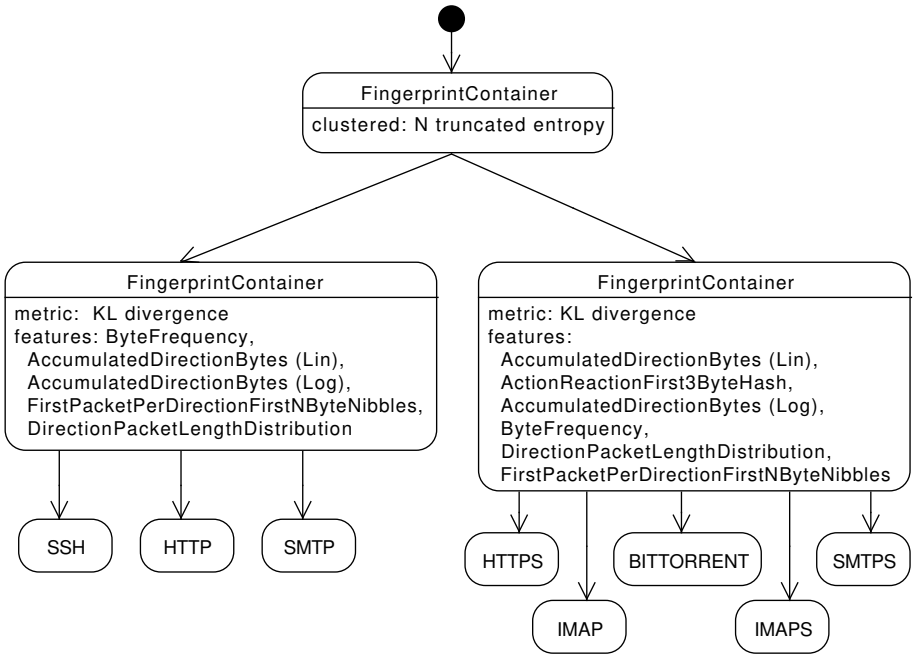


Figure 6.3: Generated solution #2

Results of classification are summarized in Table 6.4 and detailed report is included in Appendix C. Overall performance is better than for the first solution and also optimization phase took considerably less time. However, classification runtime almost doubled. This was caused by HTTPS, classification of which took 3 minutes 22 second compared to 1 minute 36 seconds for the first solution.

F-measure of classification of HTTP improved - number of false negatives dropped from 118 to 41. One cause is a better feature set for classifier - low recall of the first solution was caused by high count of HTTP flows classified as unknown. Since the feature set of second solution matches HTTP better, more flows are successfully classified. Another source of imprecision was confusion of HTTP and BITTORRENT - they fell into the same cluster and 14 flows were incorrectly classified as BITTORRENT. This was not possible in second solution since they are separated into different clusters.

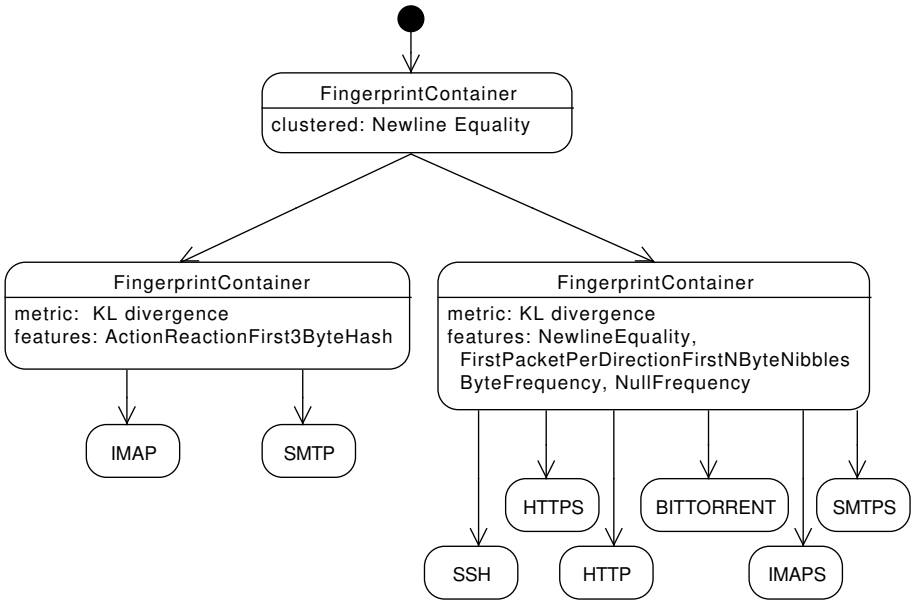


Figure 6.4: Generated solution #3

6.3.2 Clustering with newline equality

To test the clustering with newline equality and investigate, why it was not chosen by optimizer, we manually configure classifier as shown on Figure 6.4. As expected, clusterer puts SMTP and IMAP into the same cluster, since only for these two protocols NewlineEquality returns true. Since both IMAP and SMTP are plaintext protocols, only the value of ActionReactionFirst3ByteHash should suffice to classify them. For the second cluster we chose feature set from the first generated solution.

Average Precision	0.8263
Average Recall	0.7741
Average F measure	0.7921
Classification runtime	97.02 s

Table 6.5: Measurement of solution #3

Overall F-measure decreased and the main cause is low recall of classification of HTTP. High number of false negatives was caused by clustering

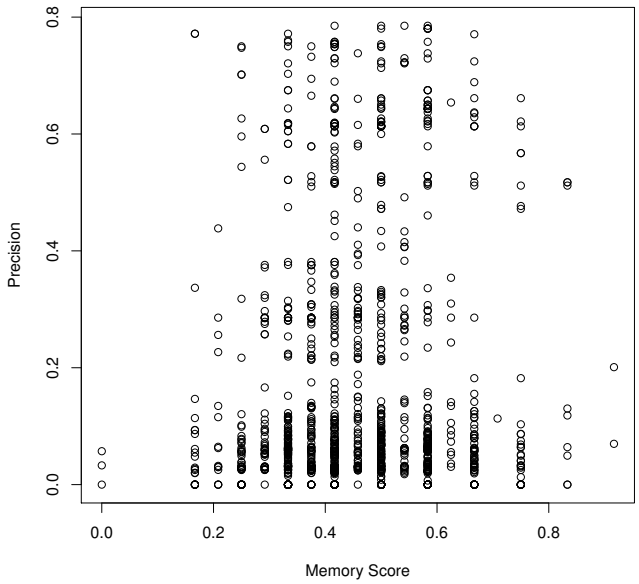


Figure 6.5: Solutions generated randomly

by NewlineEquality. Even though HTTP flows are often compressed, there is still considerable amount of HTTP traffic, which contains the same count of CR and LF bytes and thus falls within the first cluster. After the flow falls within incorrect cluster, the best thing classifier can to is to label it unknown, and that is the source of high false negatives for HTTP. For this reason clustering with NewlineEquality is not robust.

6.3.3 Solution space structure and the peaking effect

To better understand the structure of solution space and the correlation between the number of features and precision, we generated random solutions and measured their performance. The results are plotted in Figure 6.5, with memory score on x-axis and precision on y-axis. Memory score was computed as a ratio of number of used features to the total number of features, with 0 meaning no features were used and 1 that all features were used.

The majority of solutions have a low score. There are almost no solutions with one or two features and a high precision, however, the same holds for

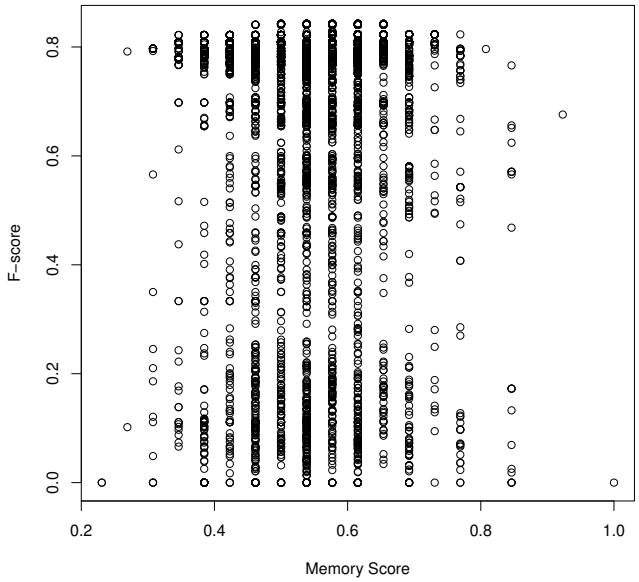


Figure 6.6: Solutions generated by Optimizer

solutions with many features. It is clear that solutions with high precision usually use from 3 to 7 features for classification.

Figure 6.6 contains configurations generated by Optimizer during generation of the first solution. We can see the peaking effect - solutions with the highest precision have 4 to 7 features and genetic algorithm preferred them. Hourglass-like structure is the result of genetic algorithm. Solutions with low score are generated randomly, whereas solutions with high scores are results of mutation of individuals from the previous population.

6.4 Discussion

Evaluation of the framework shows the optimal configuration of Classifier highly depends on the set of protocols to be classified. Moreover, comparison of the first and second generated solution shows that two sets of features can have a similar classification performance without having much in common. Such structure of the problem highly increases the difficulty of finding the optimal classification algorithm. Therefore, it is advantageous to include Optimizer that automates this task.

Optimizer based on generic genetic algorithm could be more stable and predictable. During our work with framework we found that during multiple runs with the same setup the number of solutions generated by the Optimizer was varying a lot. During optimization phase of the first solution, 2160 unique solutions were generated compared to 468 for the second solution. This suggests there is a room for improvement of the Optimizer, either by optimizing genetic algorithm, or by choosing a different algorithm for optimization. In this work we used the simplest to prove the concept of classifier optimization and the enhancement of the Optimizer can be a topic for future research.

Splitting the protocols into different groups with their own classifiers proved to be more effective and the Optimizer favored it in both solutions. Resulting algorithm was considerably faster than SPID with slightly higher F-measure and less measured features. However, this was caused by more optimal selection of features, which can classify BITTORRENT better. As shown in section 6.2, feature set of SPID can be further optimized, too. Therefore, more important observation comes from the comparison of two generated solutions - splitting the protocols decreases amount of false negatives. The reason behind this is simple - since classifiers have less possible outcomes, they are less likely to make an incorrect decision.

To leverage the fact that each classifiers has a smaller set of protocols to classify, a different classification method should be proposed. Kullback-Leibler divergence proved to be a good metric, but choosing the outcome based on maximum likelihood with threshold does not differ with the number of possible outcomes. A classifier that would take differences in Kullback-Leibler divergences into account could achieve better results. As example, we might have three protocols with high Kullback-Leibler divergence and thus unlikely to be the correct outcome, and one with rather low score. If this score is still above the threshold, 2.25 for SPID, it will be marked unknown. However, since one protocol is much more likely than others, we could output it as a result despite being slightly above the threshold. This will affect classification of traffic unknown to classifier, but whether the amount of false positives would significantly increase remains to be tested. It may be the case that unknown traffic would result in more even distribution of scores across known protocols and none of them would dominate. Design of new classification method is a promising area for improvement of the framework, since other classification methods and heuristics could utilize the advantage of split traffic better.

Clustering performance met expectations, however, it proved difficult to optimize clustering feature. A novel feature - NewlineEquality - was

not suitable and N truncated entropy was sensitive to the amount of bytes and packets measured. In case of HTTP, when the whole payload was measured, value was likely to be higher due to the compression of application data. Choosing different range of payload bytes to be measured, designing a novel feature or proposing a heuristic could significantly improve robustness of clustering.

A novel classification features NullFrequency and NewlineEquality proved to be rather effective, as they were included in the first generated solution. However, logarithmic version of AccumulatedDirectionBytes proved less effective than linear version. Explanation of this rather surprising result suggests that even optimal representation of features is difficult to identify.

NewlineEquality was originally designed as a feature for clustering. Its binary output is not suitable for classification that uses Kullback-Leibler divergence. We designed a version for classification with binary output `[1,0]` or `[0,1]` that can be used as input to the KL divergence function. However, the output of metric is still binary, as the two values of feature can be either the same or opposite. It was rather surprising that this feature was chosen into feature set of classifier in the first solution. However, it fits into idea that each classifier has to distinguish between small set of protocols, rather than match the correct label out of all possible outcomes with the highest probability.

Chapter 7

Conclusion

We presented a novel framework for designing new traffic classification algorithms. It is based on the study of other approaches, SPID in particular, and designed to address a common issues with precision, memory efficiency, speed of classification and the tradeoff between them. The framework represents a general approach, which we used to propose a new algorithm. Implementation of the algorithm was considerably faster and more precise than SPID.

We also use the algorithm to study certain aspects of traffic classification to get a better understanding of the problem. We confirmed peaking effect in feature selection, which we avoid by splitting the traffic using clustering and optimization of feature set. Adding the Optimizer and creating configurable Classifier proved to be both feasible and more practical approach towards classification. Despite this we identify potential for improvement in Optimizer and selection method of classifier. Feature representation from SPID was extended to support variable length vectors and this improvement allowed us to design two novel features.

We proposed a new feature NewlineEquality to be used for clustering, but surprisingly, it was more suitable for classification. A NullFrequency feature was designed to capture a simple observation, that some protocols have high ratio of Null bytes in payload, and it proved useful. We studied the effect of a slight change in representation of AccumulatedDirectionBytes feature and observed its high impact on precision of classification. Using entropy to split the traffic was difficult, since it heavily depends on all details of implementation. Despite our findings, study of features, their representation, measurement and parameter optimization is often overlooked.

Current state of methodology has several open problems, that do not allow effective comparison of proposed approaches. Evaluation of different machine learning algorithms applied on problem of classifying network traffic often omits important implementation details. In such situation, we believe that equally important as design of new algorithms is asking Why? and searching for deeper understanding of the results.

Bibliography

[1] Florian Adamsky. SPID. https://github.com/cit/Spid. [Online; accessed 11-May-2013].

[2] R. Alshammari, PT Lichodzijewski, M. Heywood, and A.N. Zincir-Heywood. Classifying ssh encrypted traffic with minimum packet header features using genetic programming. In *GECCO'09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2539–2546, 2009.

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

[4] C. Bacquet, K. Gumus, D. Tizer, A.N. Zincir-Heywood, and M.I. Heywood. A comparison of unsupervised learning techniques for encrypted traffic identification. *Journal of Information Assurance and Security*, 5:464–472, 2010.

[5] Roni Bar-Yanai, Michael Langberg, David Peleg, and Liam Roditty. Realtime classification for encrypted traffic. *Experimental Algorithms*, pages 373–385, 2010.

[6] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, page 1. ACM, 2007.

[7] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 145–154. ACM, 2007.

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

[9] L. Bernaille, R. Teixeira, and K. Salamatian. Early application identification. In *Proceedings of the 2006 ACM CoNEXT conference*, page 6. ACM, 2006.

[10] George Bissias, Marc Liberatore, David Jensen, and Brian Levine. Privacy vulnerabilities in encrypted http streams. In *Privacy Enhancing Technologies*, pages 1–11. Springer, 2006.

[11] Benjamin C Brodie, David E Taylor, and Ron K Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 191–202. IEEE Computer Society, 2006.

[12] Arthur Callado, Carlos Kamienski, Géza Szabó, B Gero, Judith Kelner, Stênio Fernandes, and Djamel Sadok. A survey on internet traffic identification. *Communications Surveys & Tutorials, IEEE*, 11(3):37–52, 2009.

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

[14] Young H Cho and William H Mangione-Smith. Deep packet filter with dedicated logic and read only memories. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 125–134. IEEE, 2004.

[15] Young H Cho and William H Mangione-Smith. Fast reconfiguring deep packet filter for 1+ gigabit network. In *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 215–224. IEEE, 2005.

[16] Thomas M Cover and Jan M Van Campenhout. On the possible orderings in the measurement selection problem. *Systems, Man and Cybernetics, IEEE Transactions on*, 7(9):657–661, 1977.

[17] A. Dainotti, A. Pescapé, and C. Sansone. Early classification of network traffic through multi-classification. *Traffic Monitoring and Analysis*, pages 122–135, 2011.

[18] Alberto Dainotti, Walter de Donato, and Antonio Pescapé. Tie: A community-oriented traffic classification platform. *Traffic Monitoring and Analysis*, pages 64–74, 2009.

[19] Sarang Dharmapurikar and John Lockwood. Fast and scalable pattern matching for content filtering. In *Architecture for networking and communications systems, 2005. ANCS 2005. Symposium on*, pages 183–192. IEEE, 2005.

[20] Peter Dorfinger, Georg Panholzer, and Wolfgang John. Entropy estimation for real-time encrypted traffic identification (short paper). In *Traffic Monitoring and Analysis*, pages 164–171. Springer, 2011.

[21] Danhua Guo, Guangdeng Liao, Laxmi N Bhuyan, and Bin Liu. An adaptive hash-based multilayer scheduler for l7-filter on a highly threaded hierarchical multi-core server. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 50–59. ACM, 2009.

[22] Danhua Guo, Guangdeng Liao, Laxmi N Bhuyan, Bin Liu, and Jianxun Jason Ding. A scalable multithreaded l7-filter design for multi-core servers. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 60–68. ACM, 2008.

[23] Erik Hjelmvik. Statistical Protocol Identification Algorithm - Attributes. http://sourceforge.net/apps/mediawiki/spid/index.php?title=AttributeMeters. [Online; accessed 11-May-2013].

[24] Erik Hjelmvik. Statistical Protocol Identification Algorithm homepage. http://sourceforge.net/projects/spid/. [Online; accessed 11-May-2013].

[25] Erik Hjelmvik and Wolfgang John. Statistical protocol identification with spid: Preliminary results. In *Swedish National Computer Networking Workshop*, 2009.

[26] Nan Hua, Haoyu Song, and TV Lakshman. Variable-stride multi-pattern matching for scalable deep packet inspection. In *INFOCOM 2009, IEEE*, pages 415–423. IEEE, 2009.

[27] Brad L Hutchings, R Franklin, and D Carver. Assisting network intrusion detection with reconfigurable hardware. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 111–120. IEEE, 2002.

[28] Marios Iliofotou, Prashanth Pappu, Michalis Faloutsos, Michael Mitzenmacher, Sumeet Singh, and George Varghese. Network monitoring using traffic dispersion graphs (tdgs). In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 315–320. ACM, 2007.

[29] Anil Jain and Douglas Zongker. Feature selection: Evaluation, application, and small sample performance. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(2):153–158, 1997.

[30] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: multilevel traffic classification in the dark. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 229–240. ACM, 2005.

[31] Hyunchul Kim, Kimberly C Claffy, Marina Fomenkov, Dhiman Barman, Michalis Faloutsos, and KiYoung Lee. Internet traffic classification demystified: myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT conference*, page 11. ACM, 2008.

[32] Josef Kittler, Mohamad Hatef, Robert P. W. Duin, and Jiri Matas. On combining classifiers. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(3):226–239, 1998.

[33] Taskin Kocak and Ilhan Kaya. Low-power bloom filter architecture for deep packet inspection. *Communications Letters, IEEE*, 10(3):210–212, 2006.

[34] C Köhnen, Christian Uberall, Florian Adamsky, Veselin Rakocevic, Muttukrishnan Rajarajan, and R Jäger. Enhancements to statistical protocol identification (spid) for self-organised qos in lans. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–6. IEEE, 2010.

[35] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[36] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 339–350. ACM, 2006.

[37] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Architec-*

ture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on, pages 81–92. IEEE, 2006.

[38] Marc Liberatore and Brian Neil Levine. Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 255–263. ACM, 2006.

[39] Yeon-sup Lim, Hyun-chul Kim, Jiwoong Jeong, Chong-kwon Kim, Ted Taekyoung Kwon, and Yanghee Choi. Internet traffic classification demystified: on the sources of the discriminative power. In *Proceedings of the 6th International COnference*, page 9. ACM, 2010.

[40] Yongli Ma, Zongjue Qian, Guochu Shou, and Yihong Hu. Study of information network traffic identification based on c4. 5 algorithm. In *Wireless Communications, Networking and Mobile Computing, 2008. WiCOM'08. 4th International Conference on*, pages 1–5. IEEE, 2008.

[41] A. Madhukar and C. Williamson. A longitudinal study of p2p traffic classification. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2006. MASCOTS 2006. 14th IEEE International Symposium on*, pages 179–188. IEEE, 2006.

[42] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On dominant characteristics of residential broadband internet traffic. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 90–102. ACM, 2009.

[43] A. Moore and K. Papagiannaki. Toward the accurate identification of network applications. *Passive and Active Network Measurement*, pages 41–54, 2005.

[44] Andrew W Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 50–60. ACM, 2005.

[45] Andrew W Moore, Denis Zuev, and Michael Crogan. Discriminators for use in flow-based classification. Technical report, Technical report, Intel Research, Cambridge, 2005.

[46] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys & Tutorials, IEEE*, 10(4):56–76, 2008.

[47] J Olivain and J Goubault-Larrecq. Detecting subverted cryptographic protocols by entropy checking. *Laboratoire Spécification et Vérification, ENS Cachan, France, Research Report LSV-06-13*, 2006.

[48] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[49] Piti Piyachon and Yan Luo. Efficient memory utilization on network processors for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 71–80. ACM, 2006.

[50] Martin Roesch et al. Snort-lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX conference on System administration*, pages 229–238. Seattle, Washington, 1999.

[51] Luca Salgarelli, Francesco Gringoli, and Thomas Karagiannis. Comparing traffic classifiers. *ACM SIGCOMM Computer Communication Review*, 37(3):65–68, 2007.

[52] Yong Shi, Yuqing Song, and Aidong Zhang. A shrinking-based clustering approach for multidimensional data. *Knowledge and Data Engineering, IEEE Transactions on*, 17(10):1389–1403, 2005.

[53] Chao Sima and Edward R Dougherty. The peaking phenomenon in the presence of feature-selection. *Pattern Recognition Letters*, 29(11):1667–1674, 2008.

[54] Jung-Sik Sung, Seok-Min Kang, Youngseok Lee, Taeck-Geun Kwon, and Bong-Tae Kim. A multi-gigabit rate deep packet inspection algorithm using tcam. In *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE*, volume 1, pages 5–pp. IEEE, 2005.

[55] G. Szabó, J. Szüle, Z. Turányi, and G. Pongrácz. Multi-level machine learning traffic classification system. In *ICN 2012, The Eleventh International Conference on Networks*, pages 69–77, 2012.

[56] Géza Szabó, Dániel Orincsay, Szabolcs Malomsoky, and István Szabó. On the validation of traffic classification algorithms. *Passive and Active Network Measurement*, pages 72–81, 2008.

[57] Lin Tan and Timothy Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ACM*

SIGARCH Computer Architecture News, volume 33, pages 112–122. IEEE Computer Society, 2005.

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

[59] Lei Xu, Adam Krzyzak, and Ching Y Suen. Methods of combining multiple classifiers and their applications to handwriting recognition. *Systems, Man and Cybernetics, IEEE Transactions on*, 22(3):418–435, 1992.

[60] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, pages 93–102. IEEE, 2006.

[61] Lei Yu and Huan Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, volume 20, page 856, 2003.

[62] Min Zhang, Wolfgang John, K Claffy, and Nevil Brownlee. State of the art in traffic classification: A research review. In *PAM Student Workshop*, 2009.

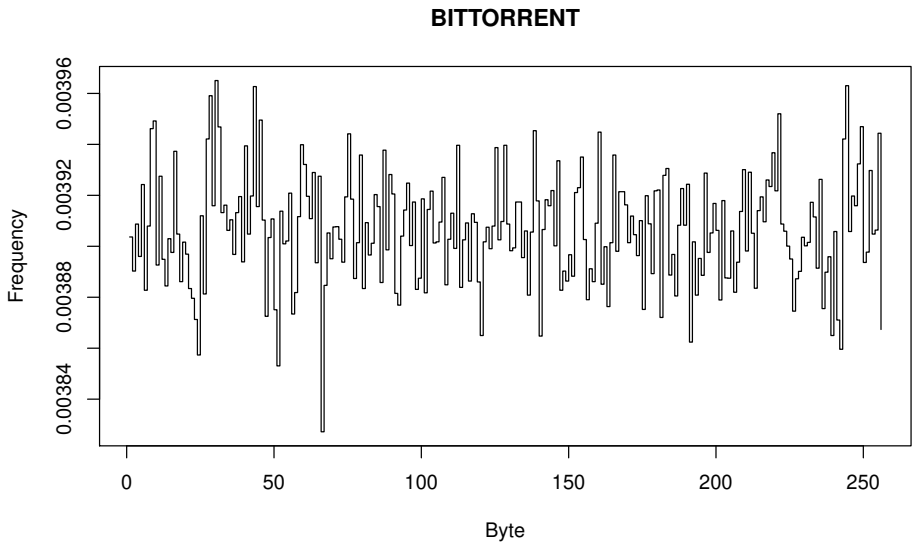
Appendix A

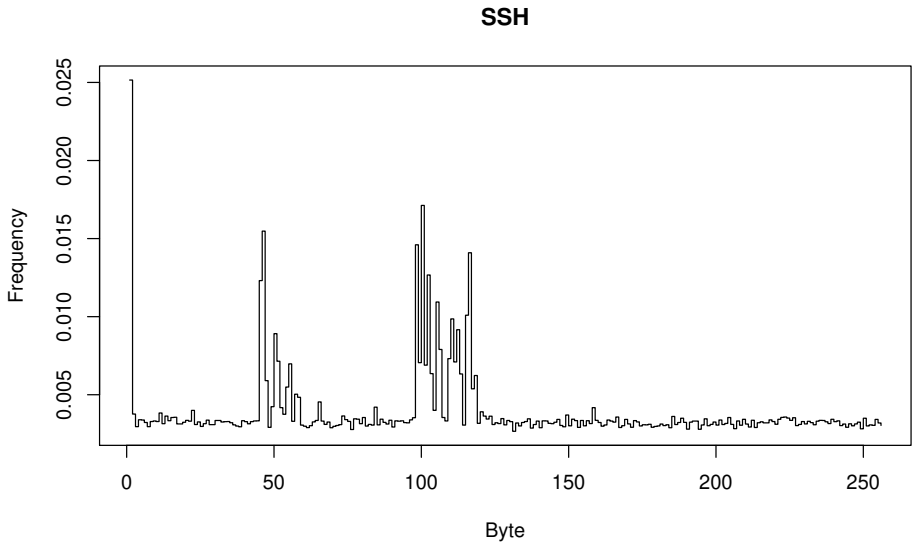
Values of features and byte frequencies of protocols

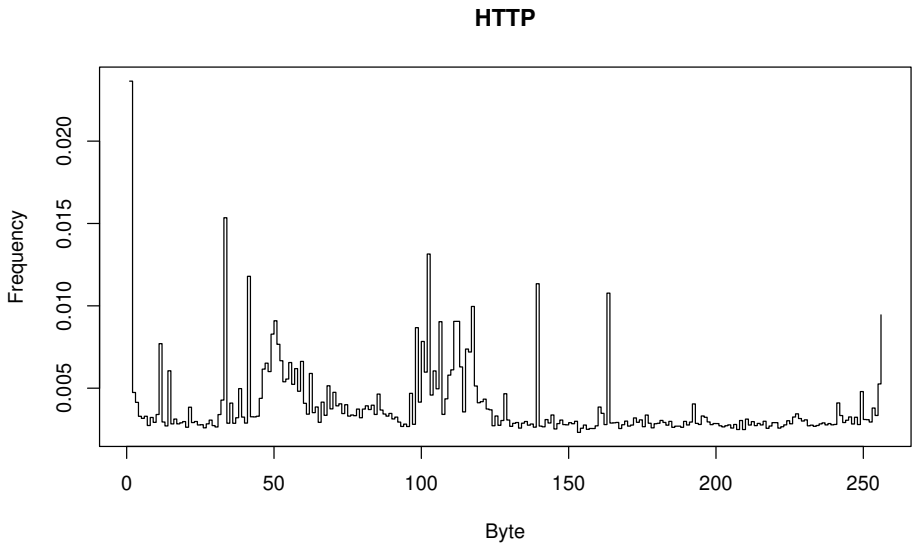
Investigation of byte frequencies of protocol proved to be a good way to understand a protocol better from the classification standpoint. Table A.1 shows values of four features for each protocol as measured and used by our implementation.

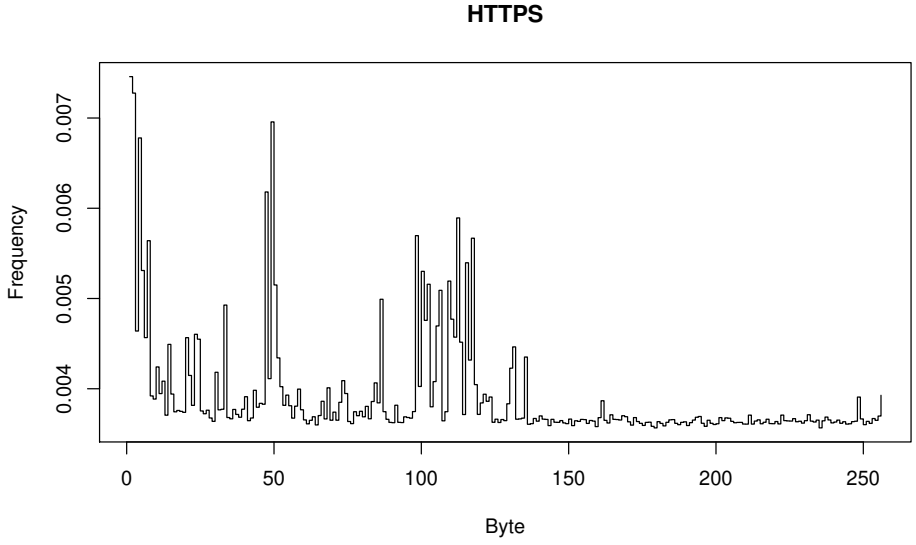
Protocol	Entropy	N truncated entropy	NullFrequency [%]	Newline Equality
BITTORRENT	7.99	0.06	0.39	False
HTTP	7.65	1.18	2.36	False
HTTPS	7.96	0.68	0.75	False
IMAP	6.08	0.81	0.00	True
IMAPS	7.92	0.51	1.22	False
SMB2	3.34	2.68	62.18	False
SMTP	6.10	0.95	0.00	True
SMTPS	7.98	0.43	0.41	False
SSH	7.93	1.51	2.52	False

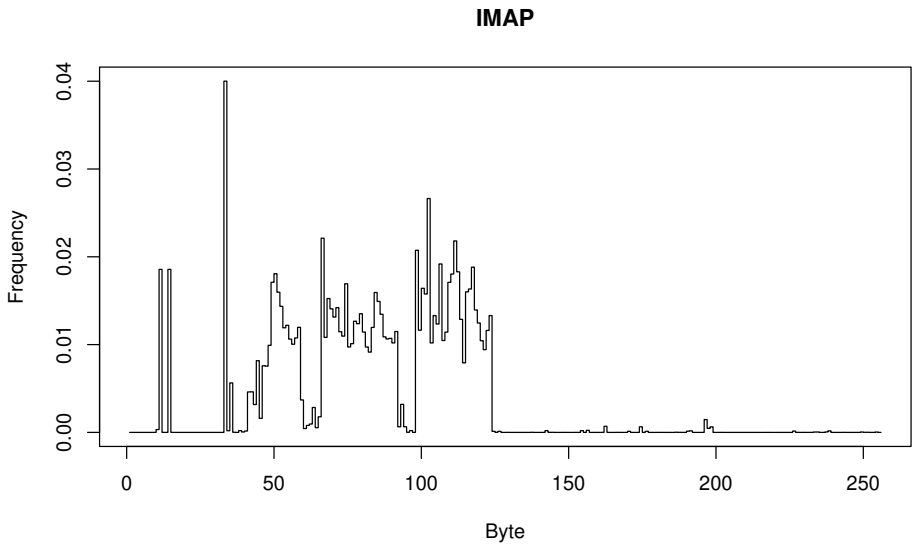
Table A.1: Values of chosen features of protocols

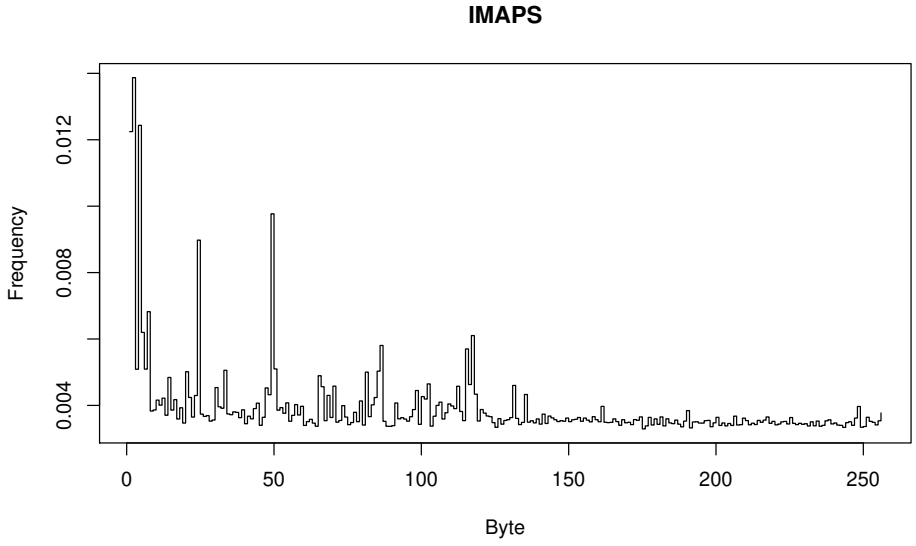

BITTORRENT
Frequency
0.00384
0.00388
0.00392
0.00396
0
50
100
150
200
250
Byte

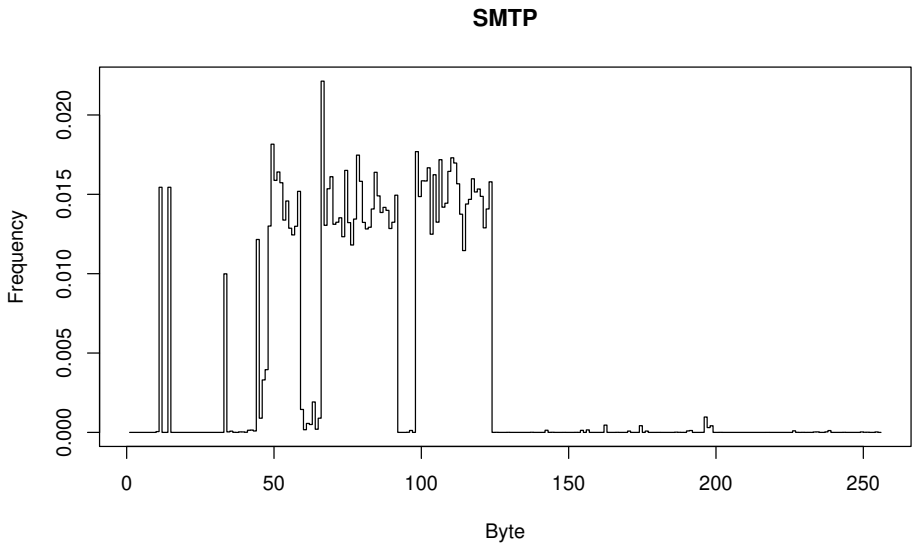

SSH
Frequency
0.005
0.010
0.015
0.020
0.025
0
50
100
150
200
250
Byte

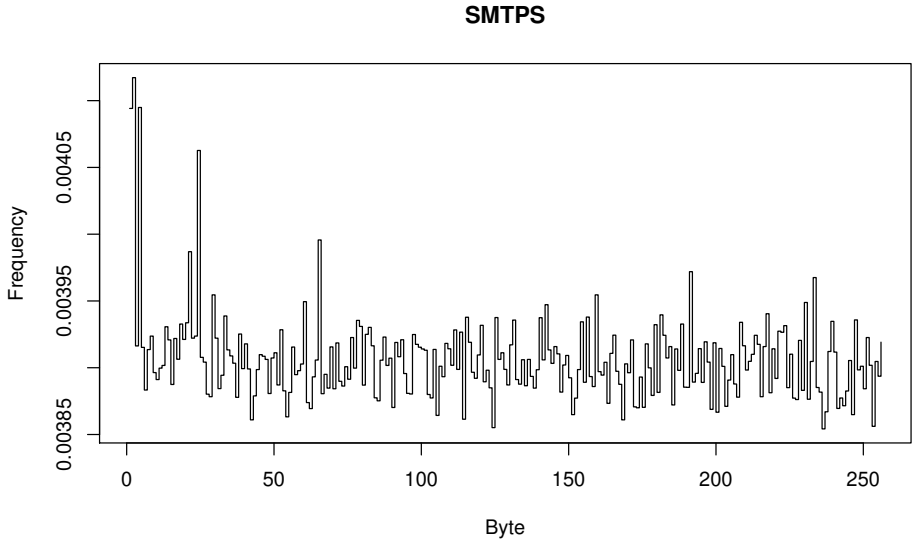

HTTP
Frequency
0.005
0.010
0.015
0.020
0
50
100
150
200
250
Byte

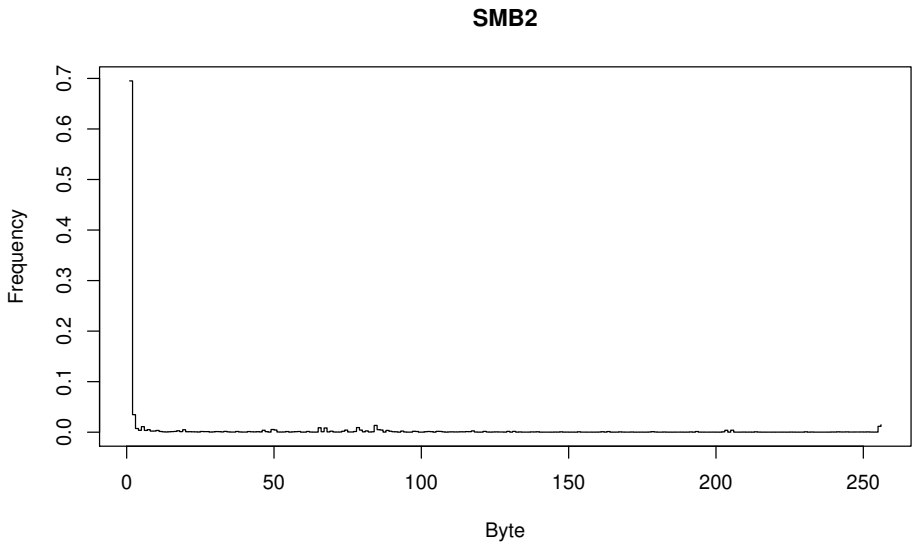

HTTPS
Frequency
0.004
0.005
0.006
0.007
0
50
100
150
200
250
Byte


IMAP
Frequency
0.00
0.01
0.02
0.03
0.04
0
50
100
150
200
250
Byte


IMAPS
Frequency
0.004
0.008
0.012
0
50
100
150
200
250
Byte


SMTP
Frequency
0.000
0.005
0.010
0.015
0.020
0
50
100
150
200
250
Byte


SMTPS
Frequency
0.00385
0.00395
0.00405
0
50
100
150
200
250
Byte


SMB2
Frequency
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
50
100
150
200
250
Byte

Appendix B

Detailed results of SPID evaluation

Protocol	base					
	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	0	0	36	-	0.0	-
HTTP	253	0	53	1.0	0.8267	0.9051
HTTPS	660	13	14	0.9806	0.9792	0.9799
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	35	0	40	1.0	0.4666	0.6363
SMB2	4	0	1	1.0	0.8	0.8888
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	3	22	6	0.12	0.3333	0.1764
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	116	0	0.0	-	-
average				0.78895	0.70164	0.72658

Protocol	base+lin					
	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	13	0	31	1.0	0.2954	0.4561
HTTP	252	0	54	1.0	0.8235	0.9032
HTTPS	654	14	19	0.9790	0.9717	0.9753
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	34	5	40	0.8717	0.4594	0.6017
SMB2	4	0	1	1.0	0.8	0.8888
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	3	21	6	0.125	0.3333	0.1818
SSH	10	0	1	1.0	0.9090	0.9523
unknown		112	0	0.0	-	-
average				0.88618	0.73248	0.77329

Protocol	**base+log**					
	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	8	0	34	1.0	0.1904	0.32
HTTP	253	0	53	1.0	0.8267	0.9051
HTTPS	654	13	19	0.9805	0.9717	0.9761
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	34	5	40	0.8717	0.4594	0.6017
SMB2	4	0	1	1.0	0.8	0.8888
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	3	21	6	0.125	0.3333	0.1818
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	115	0	0.0	-	-
average				0.88636	0.72117	0.75846

Protocol	**base+null**					
	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	34	0	26	1.0	0.5666	0.7234
HTTP	257	0	51	1.0	0.8344	0.9097
HTTPS	655	13	19	0.9805	0.9718	0.9761
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	35	5	40	0.875	0.4666	0.6086
SMB2	4	0	1	1.0	0.8	0.8888
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	3	22	6	0.12	0.3333	0.1764
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	104	0	0.0	-	-
average				0.88617	0.76463	0.80397

Protocol	**base+lin+log**					
	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	32	0	25	1.0	0.5614	0.7191
HTTP	252	0	54	1.0	0.8235	0.9032
HTTPS	654	13	19	0.9805	0.9717	0.9761
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	34	5	40	0.8717	0.4594	0.6017
SMB2	4	0	1	1.0	0.8	0.8888
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	3	21	6	0.125	0.3333	0.1818
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	107	0	0.0	-	-
average				0.88636	0.76203	0.80259

Protocol	**base+lin+null**					
	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	54	1	14	0.9818	0.7941	0.8780
HTTP	257	0	50	1.0	0.8371	0.9113
HTTPS	654	14	20	0.9790	0.9703	0.9746
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	34	5	40	0.8717	0.4594	0.6017
SMB2	4	0	1	1.0	0.8	0.8888
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	3	21	6	0.125	0.3333	0.1818
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	91	0	0.0	-	-
average				0.88417	0.78924	0.82099

Protocol	**base+log+null**					
	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	47	0	17	1.0	0.7343	0.8468
HTTP	253	0	54	1.0	0.8241	0.9035
HTTPS	654	13	19	0.9805	0.9717	0.9761
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	34	5	40	0.8717	0.4594	0.6017
SMB2	4	0	1	1.0	0.8	0.8888
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	3	21	6	0.125	0.3333	0.1818
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	99	0	0.0	-	-
average				0.88636	0.78131	0.81682

Protocol	**base+lin+log+null**					
	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	54	8	15	0.8709	0.7826	0.8244
HTTP	254	0	53	1.0	0.8273	0.9055
HTTPS	654	13	25	0.9805	0.9631	0.9717
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	34	5	40	0.8717	0.4594	0.6017
SMB2	4	0	1	1.0	0.8	0.8888
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	3	21	6	0.125	0.3333	0.1818
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	94	0	0.0	-	-
average				0.87201	0.78608	0.81406

Appendix C

Detailed results of generated solutions

Protocol	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	98	24	2	0.8032	0.98	0.8828
HTTP	172	0	118	1.0	0.5931	0.7445
HTTPS	793	13	31	0.9838	0.9623	0.9730
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	56	12	19	0.8235	0.7466	0.7832
SMB2	4	0	1	1.0	0.8	0.8888
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	0	1	9	0.0	0.0	-
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	131	0	0.0	-	-
average				0.8456	0.7768	0.8027

Actual	bittorrent	http	https	imap	imaps	smb2	smtp	smtps	ssh	unknown
BITTORRENT	98	0	0	0	0	0	0	0	0	2
HTTP	14	172	0	0	0	0	0	0	0	104
HTTPS	10	0	793	0	9	0	0	0	0	12
IMAP	0	0	0	13	0	0	0	0	0	0
IMAPS	0	0	7	0	56	0	0	1	0	11
SMB2	0	0	0	0	0	4	0	0	0	1
SMTP	0	0	0	0	0	0	14	0	0	0
SMTPS	0	0	6	0	3	0	0	0	0	0
SSH	0	0	0	0	0	0	0	0	10	1
unknown	0	0	0	0	0	0	0	0	0	0

Table C.1: Detailed reports for the first solution

Protocol	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	84	16	8	0.84	0.9130	0.8749
HTTP	268	0	41	1.0	0.8673	0.9289
HTTPS	650	0	91	1.0	0.8771	0.9345
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	34	5	34	0.8717	0.5	0.6355
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	3	21	0	0.125	1.0	0.2222
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	133	0	0.0	-	-
average				0.8546	0.8833	0.8185

Actual	bittorrent	http	https	imap	imaps	smtp	smtps	ssh	unknown
BITTORRENT	84	0	0	0	0	0	0	0	8
HTTP	0	268	0	0	0	0	0	0	41
HTTPS	16	0	650	0	5	0	0	0	70
IMAP	0	0	0	13	0	0	0	0	0
IMAPS	0	0	0	0	34	0	21	0	13
SMTP	0	0	0	0	0	14	0	0	0
SMTPS	0	0	0	0	0	0	3	0	0
SSH	0	0	0	0	0	0	0	10	1
unknown	0	0	0	0	0	0	0	0	0

Table C.2: Detailed reports for the second solution

Protocol	TP	FP	FN	Precision	Recall	F-measure
BITTORRENT	98	24	2	0.8032	0.98	0.8828
HTTP	173	0	118	1.0	0.5945	0.7456
HTTPS	793	13	31	0.9838	0.9623	0.9730
IMAP	13	0	0	1.0	1.0	1.0
IMAPS	56	12	19	0.8235	0.7466	0.7832
SMTP	14	0	0	1.0	1.0	1.0
SMTPS	0	1	9	0.0	0.0	-
SSH	10	0	1	1.0	0.9090	0.9523
unknown	0	130	0	0.0	-	-
average				0.8263	0.7741	0.7921

Actual	bittorrent	http	https	imap	imaps	smtp	smtps	ssh	unknown
BITTORRENT	98	0	0	0	0	0	0	0	2
HTTP	14	173	0	0	0	0	0	0	104
HTTPS	10	0	793	0	9	0	0	0	12
IMAP	0	0	0	13	0	0	0	0	0
IMAPS	0	0	7	0	56	0	1	0	11
SMTP	0	0	0	0	0	14	0	0	0
SMTPS	0	0	6	0	3	0	0	0	0
SSH	0	0	0	0	0	0	0	10	1
unknown	0	0	0	0	0	0	0	0	0

Table C.3: Detailed reports for the third solution