Politechnika Śląska
Wydział Automatyki, Elektroniki i Informatyki

Paweł Foremski

ZASTOSOWANIE KLASYFIKACJI KASKADOWEJ DO ROZPOZNAWANIA RUCHU W SIECI INTERNET

INTERNET TRAFFIC IDENTIFICATION USING CASCADE CLASSIFICATION

Rozprawa doktorska napisana pod kierunkiem prof. dr hab. inż. Tadeusza Czachórskiego

Gliwice
31.08.2018

Abstract

The Internet is the largest telecommunication network in the world, being used by a few billion people everyday. It became the most important communication medium that deeply changed our lives: nowadays, we rely on Internet services like the World-Wide Web, e-mail messages, and video conferencing. These services need decent infrastructure for transmitting the data between myriads of servers, personal computers, smartphones, and other devices. Moreover, the Internet is growing at enormous pace—in terms of traffic volume, connected devices, and number of communication protocols—which means we need adequate management tools to keep it robust and secure.

One of the basic Internet management tools is *Traffic Classification*: that is, matching Internet transmissions to names of applications that generated them. This tool has several applications, e.g. traffic measurement and visualization, policy routing, traffic shaping, and trend analysis. Contemporary traffic classification algorithms employ artificial intelligence to automatically learn the features that discriminate various Internet protocols—that is—no rules are set *a priori* by a human operator. However, although highly effective, none of them is appropriate for all Internet protocols—instead, classifiers are often tailored only at a specific application. Thus, we need to integrate these algorithms together in order to manage all Internet protocols in a single system.

In this thesis, we introduce a new method for integrating traffic classifiers: the *Waterfall* architecture. It builds upon the *Cascade Classification* technique, which connects several modules in a series of classifiers. By means of experimental validation on real Internet traffic, we evaluate the method and demonstrate its robustness: for instance, Waterfall is able to classify more than 50% of evaluated traffic using just 3 simple classifiers. We also introduce a cascade optimization technique that lets for tuning a Waterfall system for desired performance goals by means of simulation.

Acknowledgements

I would like to thank my PhD advisor, Professor Tadeusz Czachórski, and my grant supervisor, Professor Michele Pagano (University of Pisa), for their guidance and help. Their friendly attitude and support during the start of my scientific career was very important to me. I also express my gratitude to colleagues with whom I authored my first academic papers: Mateusz Nowak, Sławomir Nowak, and Christian Callegari (University of Pisa).

I owe special thanks to Dave Plonka (Akamai) and Arthur Berger (Akamai/MIT) for the outstanding summer of 2015, when I had the privilege of being their PhD intern in Cambridge, MA, and for the later successful collaboration. The time I spent with them was pivotal to my career. I also thank my colleagues—Michał Gorawski, Krzysztof Grochla, Grzegorz Karch, Michał Romaszewski, Konrad Połys, and Mariusz Słabicki—for entertaining coffee break discussions and for reviewing early drafts of my papers.

I thank God for the gift of life and the talent for computers I discovered at the age of 7, which since then has been one of my greatest sources of joy. I always owe gratitude (and a long vacation) to my wonderful wife Dominika, for her patience and for our beautiful daughter Antonina.

I dedicate this work to my deceased mother Grażyna, who devoted her life to education of me and my elder brother, and to my father Artur, who bought us the first computer back in the years when it costed half a car.

Paweł Foremski
Gliwice, December 2017

Work on some of the publications reproduced in this thesis was funded by the Polish National Science Centre, under research grant nr 2011/01/N/ST6/07202, project "Multilevel Traffic Classification" [59].

Contents

List of Abbreviations

GUI Graphical User Interface
DPI Deep Packet Inspection
P2P Peer-to-Peer
VoIP Voice over IP
PPPoE Point-to-Point over Ethernet
TTL Time To Live
LAN Local Area Network
QoS Quality of Service
QoE Quality of Experience
DSCP Differentiated services Code Point
CDN Content Delivery Network
ISP Internet Service Provider
DNS Domain Name System
DNSSEC Domain Name System Security Extensions
IoT Internet of Things
SaaS Software as a Service
AI Artificial Intelligence
PR Pattern Recognition
ML Machine Learning
SVM Support Vector Machine
BKS Behavior Knowledge Space
NFL No Free Lunch
MCS Multiple Classifier System
NLP Natural Language Processing
VSM Vector Space Model
TC Traffic Classification
CC Cascade Classification
CF Classifier Fusion
ABI Application Binary Interface
ARFF Attribute-Relation File Format

Chapter 1

Introduction

1.1 Background

Traffic Classification (TC)—or Traffic *Identification*—is the act of matching IP packets to names of computer programs that generated them. It resembles an "Internet microscope", a tool that lets us to look at an Internet link, see the data transmissions, and identify various types of IP traffic. Another useful metaphor to TC is listening to foreigners and recognizing their language. Quite often, we are able to identify an unfamiliar language or dialect even if we cannot fully understand it. Similarly, the TC problem is recognizing Internet protocols given their IP traffic, without interest in their full content.

Having IP packets attributed to specific protocols or applications makes the Internet easier to manage. For example, TC is important for traffic monitoring: if we want to visualize the traffic flowing through a router, it is useful to know the protocols inside. TC also helps network security officers to reveal and track suspicious Internet activity. It is often used for implementing Quality of Service (QoS) schemes, via traffic shaping, policy routing, and packet filtering. Scientific and government agencies employ TC for identifying global Internet trends [27, 159]. However, TC does not imply traffic surveillance of any kind. Historically, the original motivation for work on this problem was Internet management: intrusion detection [67] (year 1994) and traffic prioritization [127] (year 1996), among the others. Nevertheless, nowadays TC is also used for different purposes, e.g. profiling Internet users [115].

Classifying an IP packet alone is not trivial, as the packet headers do not have the application or protocol name stored in them. In the past, port numbers were used for discriminating the traffic class [84] (e.g. linking port 80/TCP to HTTP), but this became ineffective due to the raise of Peer-to-Peer (P2P) networking in the early 2000s [81], which uses random port numbers. Nowadays, a popular and *de facto* standard classification method is Deep Packet Inspection (DPI): pattern matching on full packet contents [128]. However, although being more accurate than port-based classification, it requires more computing power and brings privacy concerns. Moreover, pervasive encryption and other issues make DPI increasingly irrelevant [48, 82].

Instead, modern classifiers investigate groups of packets to find distinguishing features of IP flows, rather than of single IP packets. Usually, a sequence of

packets is statistically summarized—for example, using the average packet size and inter-packet arrival time—and the resultant *feature vector* is classified using a Machine Learning (ML) algorithm, e.g. Support Vector Machine (SVM) [86]. Such methods are largely resistant to misuse of the port number and to encryption: the overall behavior of a particular protocol or host is examined instead of seeking for a strict match in a single packet. Moreover, the essence of ML classification is training an algorithm to discern between two groups of abstract objects without programming it explicitly to do so. This means that ML-based traffic classifiers learn Internet protocols by example, without the need for *a priori* rules set by a human operator.

However, future TC methods will have to deal with an increasing adoption of encryption, encapsulation, multi-channel techniques, and with the tremendous growth of the Internet [40]. TC becomes more complex and needs breaking into smaller parts to keep it tractable. Recent publications follow this path by proposing various novel techniques—e.g. counting packets [14], intercepting DNS traffic [15, 63], analyzing payload frequencies [55]—that address only a portion of the Internet traffic. However, we lack methods that would combine these proposals to effectively work together. We need to apply Multiple Classifier System (MCS) techniques to the future of TC [87].

1.2 Thesis Statement

In this thesis, we show how to apply cascade classification to solve the TC problem by integrating many classifiers to work together. We will show that:

> Cascade classification is an effective method for identifying Internet traffic. It allows for connecting different traffic classifiers together using the "divide and conquer" paradigm, and in comparison with classifier fusion, it inherently requires less computing power.

The thesis contributes to the state of the art in computer science by introducing a new method for TC, which is an important computer networks problem. Our goal is to present an original method *per se*, thus we present a qualitative comparison with classifier fusion as a commentary.

1.3 Motivation

Let us analyze the main factors that influence the future of TC. We confront this issue from two viewpoints: computer networks and artificial intelligence.

According to a recent Cisco forecast [34], the annual IP traffic transmitted in 2021 will reach 3.3 zettabytes (3.3 trillion gigabytes), growing at 24% yearly from 2016. By that time, there will be roughly 27 billion devices connected to the Internet, and more than 63% of the traffic volume will come from wireless and mobile devices. Another challenge is the Internet of Things (IoT) network: according to Gartner, by 2020 there will be 20 billion of IoT endpoint devices installed worldwide [69]. Taking another perspective, in 2014 IDC estimated the number of software engineers to be roughly 18.5 million [77]. Even if only some of them develop Internet applications, the number of application protocols in the world can easily reach millions. For instance, there were 3.5 million Android

applications in the Google Play Store as of 2017 [10], and each of them could possibly implement its own protocol on the top of the HTTPS protocol.

The Internet is growing at an enormous pace, in terms of traffic volume, connected devices, and new use cases. Inferring from the history of the Internet and from various forecasts, we can expect completely new protocols to appear, making the task of traffic identification more challenging. For instance, future TC algorithms will have to deal with huge bit rates, enormous size of the IPv6 addressing space, and protocols built atop of various overlay techniques (e.g. Tor anonymity network or QUIC). Even today, we need different TC methods for different parts of the Internet traffic: long-established protocols remain easy to identify (e.g. HTTP [84]), while newer protocols require detailed examination and more computing power (e.g. P2P-TV [14]). We argue that classifying all of the Internet traffic with a single technique—at full speed, versatility, and granularity—will be increasingly troublesome. Instead, we propose to split the TC task into smaller parts, solve them separately, and combine the outcomes so the whole system can handle a relatively complete mix of Internet traffic.

On another hand, looking from the artificial intelligence perspective, there is the well-known No Free Lunch (NFL) theorem that shows no superiority of any classification algorithm over the rest [153]. It states that in principle it is impossible that a particular classifier performs better for some specific problems without worse performance for the rest. Indeed, it is only possible to trade performance on rare problems with those one expects to encounter more often [45]. Thus, we can take advantage of the NFL theorem by isolating many subproblems in TC and solving them separately using different classifiers. This brings the idea of MCS, or *ensemble learning*, in which one uses many classifiers for better performance than could be obtained by using any of them alone [87]. The basic idea behind MCS is to use a pool of classifiers on the same problem—either all of them (*classifier fusion*) or a some of them (*classifier selection*)—and to apply a combining strategy (e.g. voting) on their outcomes to obtain the final result. Such design can improve TC systems: for example, A. Dainotti et al. showed this for classifier fusion, using Behavior Knowledge Space (BKS) [41].

In the thesis we apply *cascade classification*, which is inherently different than classifier fusion, as it is a variant of classifier selection. In principle, cascade classifiers work by querying the base classifiers *sequentially*: if a particular classifier provides the answer to given problem, it wins—otherwise the process advances to the next module. Although being relatively neglected, cascade classifiers could be of primary importance for real-life applications [87]. They naturally break complex tasks into subproblems. In [64], we introduced cascade classification to TC. By experimental evaluation, we found for example that by ordering the classifiers properly, over 50% of IP flows were identified in the first 3 classifiers out of 5 total. Thus, our proposal requires less computing power than classifier fusion, which always uses all of its modules. Our method also has many other advantages that we will show in the thesis.

As motivated above, future TC systems will have to break the complex traffic identification task into many subproblems. The MCS architecture, developed for several years in the field of machine learning, seems an adequate tool for this purpose. We introduce cascade classification—an MCS variant—to traffic identification, because it is a novel method and we believe it has important advantages over solutions already presented in the literature.

1.4 Contributions

Below we enumerate the central contributions of the thesis:

1. **We present *Waterfall*, a novel method for traffic identification.** To the best of our knowledge, we are the first to propose this novel idea in TC. (see Chapter 7)
2. **Waterfall integrates independent classifiers together.** It implements the *divide and conquer* design paradigm that enables iterative development of TC systems. We demonstrate an illustrative system consisting of 5 different classifiers. (see Section 7.4)
3. **Comparing with classifier fusion, Waterfall is inherently faster.** It does not require all modules in the pool to be run on all IP flows. By experiments on real traffic we show that it is even possible to reduce the total computation time by adding new classifiers in a proper order. (see Section 7.5)
4. **We show how to tune a Waterfall system automatically.** We propose a new method for optimizing classification cascades, tailored to traffic identification. Our solution finds the choice and order of classifiers that minimizes the error count, required CPU time, and number of unlabeled flows. (see Chapter 8)

These novel ideas were presented in publications [65] and [64], both of which were recently summarized and extended in [66]. Besides the main contributions, we also highlight a few of the supplementary contributions below:

5. **We introduce *DNS-Class*, a novel TC algorithm based on DNS**. The algorithm immediately classifies a significant portion of Internet traffic by running text classification on domain names assigned to IP flows. (see Chapter 6)
6. **We review recent works in traffic classification.** By surveying 13 publications we show diversity in recent methods and we further motivate the need for integrating traffic classifiers together. (see Chapter 5)
7. **We present *tracedump*, a novel single application traffic sniffer.** We show how to intercept all Internet traffic of a single Linux process. (see Section 4.2)
8. **We release open source implementations of our proposals.** We publish source code for all methods and tools presented in the thesis. We also present *flowcalc*, a software toolkit for analyzing IP traffic and calculating flow features. (see Section 4.3)

These contributions were published in [63], [62], [61], and on the project website [59]. They are complementary to the main ideas presented in the thesis. For example, DNS-Class is used as one of the classifiers for building a Waterfall cascade, and `flowcalc` is used for extracting traffic features before the IP flows enter the classification system. We also highlight that our cascade optimization algorithm [65] contributes to machine learning by proposing a new method for optimizing an MCS structure.

We believe the thesis opens a new broad avenue for future research. Waterfall offers the following benefits to the field of traffic identification:

- **Dedicated methods for different parts of Internet traffic.** For example, separate classifiers for traditional protocols, P2P, and tunneled traffic—so that the system as a whole handles 100% of traffic.
- **Re-use of existing algorithms.** One can use established TC methods as "black box" modules for identifying well-known traffic and focus only on new problems in TC.
- **Simpler management of software projects.** Project team members can work independently on their modules. The project can easily embrace the *iterative and incremental development* model.
- **Automatic choice of the best system.** Waterfall selects the best sequence of classifiers from a possibly large pool, so that the performance for specific network is maximized.

Waterfall addresses the issues described in Section 1.3 that motivate our work: anticipated growth of the Internet and consequences of the NFL theorem. Thus, it is an adequate method for building modern TC systems. We hope our work will contribute the society with a tool for managing the future Internet.

1.5 Outline and methodology

We divide this dissertation into two parts: Part I, which introduces the reader to traffic identification and machine learning, and Part II, which presents advanced topics and describes major contributions of the thesis. The first and the last chapters are independent, as they discuss the whole work.

In Part I, we begin with a formal definition of the traffic classification problem, which explains the object of our study. We describe the context, history, state of the art, and speculate on the future of TC (Chapter 2). Then we introduce the reader to machine learning, which explains the basic method of contemporary TC solutions. We show how classification works on abstract examples, we describe supervised ML, and finally we introduce MCS (Chapter 3). Next, we give a detailed discussion on obtaining and processing real Internet traffic for training and evaluating TC systems. We present two dedicated software tools: `tracedump` and `flowcalc` (Chapter 4). We finish Part I by describing in more detail 13 real-world algorithms related to TC. We present a survey of papers, demonstrating their diversity and thus motivating our work (Chapter 5). In summary, Part I gives the thesis background; our methodology is basically referencing literature, describing software, and defining procedures.

Part II presents main thesis contributions. We begin by presenting DNS-Class, a TC algorithm tracking DNS transactions, which classifies one-third of Internet traffic, and thus needs augmentation with other algorithms (Chapter 6). Then, we describe Waterfall, a novel TC algorithm that integrates many classifiers in a cascade (Chapter 7). We finish by presenting an algorithm for optimizing classification cascades, tailored at the Waterfall architecture (Chapter 8). In general, Part II adopts an experimental methodology: we state hypotheses and validate them on real Internet traffic, applying the evaluation procedures described in Part I. Finally, we confer the thesis statement in Chapter 9.

Last but not least, as we state in Section 1.2 and explain in Section 3.5, our goal is to present an original method *per se*. We provide a qualitative comparison between cascade classification and classifier fusion, where a quantitative

comparison would also be possible. We believe that confronting a Waterfall system directly with an equivalent BKS system built off the same base classifiers would be biased, because these two MCS techniques were designed with different assumptions on the constituent classifiers in mind. Typically, we would build a Waterfall system off dedicated classifiers (i.e., targeting a portion of Internet traffic), whereas for BKS, we would rather use universal classifiers (i.e., not limited to specific protocols). Thus, we leave a quantitative comparison as out of scope of the thesis: we only compare the features of Waterfall with BKS as a commentary to our novel method.

Part I

Traffic Classification using Machine Learning

Chapter 2

Traffic Classification

2.1 General approach

Suppose we want to analyze the IP traffic flowing through an Internet link, i.e. all IP packets sent or received on a network interface. By using a decent *network sniffer*, we can easily collect the required data, along with basic interpretations of the packet contents.

Fig. 2.1 presents an illustrative output of the popular Wireshark network traffic sniffer. In the upper part of the figure we see a list of captured packets ordered by time, in the lower-right part we see the raw data transmitted on the wire, and finally in the lower-left part we see the selected packet interpreted according to Internet standards: IPv4 and TCP protocols [118, 119]. As visible on the example, the selected packet is a TCP datagram sent from IPv4 address `173.194.35.191` to `91.200.174.93`, from TCP port number `80` to `29324`.

Note that IP packets generally do not carry the name of the protocol or application in the headers or payload. Thus, in order to classify IP traffic it is impossible to just read the application name from the packet contents, even if we know how to interpret the data according to Internet standards.

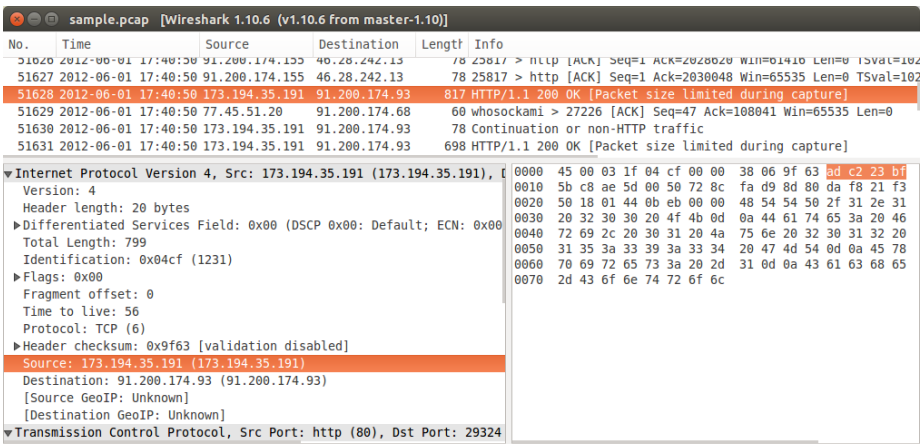

Figure 2.1: Capturing IP traffic using the "Wireshark" network traffic sniffer.

The only universal solution is to guess the protocol by smartly analyzing the traffic and gaining evidence that support our particular hypothesis. For the example presented in Fig. 2.1, we could use 3 pieces of information to guess the application: 1) source port `80` is traditionally linked with the HTTP protocol, 2) source IP address belongs to Google, and 3) ASCII interpretation of the payload reads `HTTP/1.1 200 OK`, which is a valid message of the HTTP/1.1 protocol. Thus, we may speculate that the given packet is a response to a WWW request issued to Google from a modern web browser like Chrome or Firefox. We used external sources of information to make our inference: the IANA port assignments [76], the ARIN Whois database [11], and the IETF database of RFC documents [53]. Note that the presented approach to TC requires timely knowledge on the Internet and is prone to errors, both of which will supposedly always be the central problem of TC.

Is our ad-hoc method from the example above reliable for *all* kinds of Internet traffic? Unfortunately not, because modern applications often use random port numbers, Peer-to-Peer (P2P) communication, and encryption: e.g. the Skype and BitTorrent protocols are much more difficult to identify. Thus, the traffic characteristics used in our example above—the port number, IP address, and packet payload—are not universal. Nevertheless, as the thesis will show, there still exists a large portion of the Internet traffic that is identifiable using simple methods. For the rest, we need to employ more sophisticated methods.

Let us reconsider analyzing the captured traffic presented in Fig. 2.1. Instead of looking at individual packets, we will consider *IP flows*. A flow is the set of all packets belonging to particular connection, i.e. all packets having the same destination and source IP addresses, port numbers, and the transport protocol. The notion of IP flow is useful because it allows us to find more information on the traffic than using just IP packets alone. For example, Table 2.1 presents illustrative *flow features* calculated for a few flows in the captured traffic. The column "Payloads up and down" gives the payload lengths for the first 5 data packets in the upload and download directions; the column "Payload statistics" gives the average payload length and its standard deviation for both directions; finally, the column "Inter-arrival statistics" gives the same statistics for the time gaps between consecutive data packets. The "DNS" column is supplemental and presents the domain name inferred from DNS traffic of the source host

#	Prot	Source and destination	DNS	Payloads up and down	Payload statistics [B]	Inter-arrival statistics [ms]
1	UDP	91.200.174.60:27005 178.150.21.167:27013	maso.com.ua	23, 286, 0, 0, 0 23, 138, 0, 0, 0	154, 186 80, 81	38, 38 40, 40
2	UDP	91.200.174.134:21816 109.163.231.236:80	tracker.1337x.org	16, 16, 100, 0, 0 0, 0, 0, 0, 0	44, 48 0, 0	3371, 3268 0, 0
3	TCP	91.200.174.234:34070 173.194.70.188:5228	mtalk.google.com	80, 182, 22, 488, 47 1348, 312, 1348, 312, 43	164, 191 495, 595	109, 38 69, 46
4	TCP	91.200.174.75:60674 69.171.227.27:8883	orcart.facebook.com	80, 182, 819, 22, 310 1438, 1438, 1366, 1438, 1438	118, 195 643, 401	106197, 35709 38697, 3834
5	UDP	91.200.174.20:22906 95.215.62.26:80	tracker.openbittorrent.com	16, 100, 100, 100, 100 0, 0, 0, 0, 0	152, 144 0, 0	1265, 1249 0, 0

Table 2.1: Illustrative IP flow features for the captured traffic.

(see Chapter 6): it is another example of traffic inference that surpasses the boundary of single IP packet analysis.

How these information could help us in classification? The basic observation is that similar values of flow features will often imply the same application behind the IP flows, whereas dissimilar feature values suggest different applications. Thus, it is possible to discover various types of traffic by comparing the values of the flow features. Moreover, by establishing reference values of the flow features for a particular application, we can identify its IP flows in whole traffic by comparing the observed flow features with the reference. The simplest method in such a case is to manually prepare a set of *rules*, e.g.

```
if DNS matches "tracker" AND AveragePayload ∈ [16, 250] then
    classify(BitTorrent)
endif
```

Nowadays, the most popular approach to feature-based TC is Machine Learning (ML): using an algorithm to learn the rules from a labeled dataset without explicit programming. We will introduce the ML idea and describe its most popular algorithms in Chapter 3. In general, ML automatically discovers patterns in flow features and exploits them for TC.

Classifying the traffic using flow features is more universal than matching particular values in IP packets, because it is difficult to completely eliminate statistical patterns in the Internet traffic. The state-of-the-art TC methods for classifying P2P and encrypted applications employ statistical analysis of flow-based and host-based features, as we will discuss in Chapter 5. However, ML methods heavily depend on *ground-truth*—that is, the true labels of IP flows in the training dataset—which we will cover in Chapter 4.

2.2 The TC problem

The task of TC is identifying the names of Internet protocols or applications in given IP packets. More formally, if X is the set of n packets one wants to classify and L is the set of the target traffic classes, then the TC task is defined by the following:

$$TC(X) = \{y_1, \ldots, y_m\} \quad m \leq n, \tag{2.1}$$

$$y_i = (X_i, l_i) \quad X_i \subseteq X,\ l_i \in L, \tag{2.2}$$

where y_i is an *outcome*, $X_i \neq \emptyset$ is an *object*, and l_i is a *label*: name of the protocol or application behind IP packets X_i. See Fig. 2.2 for an illustration. Note that the task of TC involves not only assigning proper labels, but also grouping packets in objects and extracting traffic features. Each packet must belong to at least one object:

$$\bigcup_{i=1}^{m} X_i = X. \tag{2.3}$$

In an extreme case of classifying single packets we have $m = n$, but in the typical case of IP flows being the classification objects we may expect $m \ll n$.

For completeness, let us note that instead of processing raw IP packets in X, it is possible to use abstract forms of traffic information in X that directly

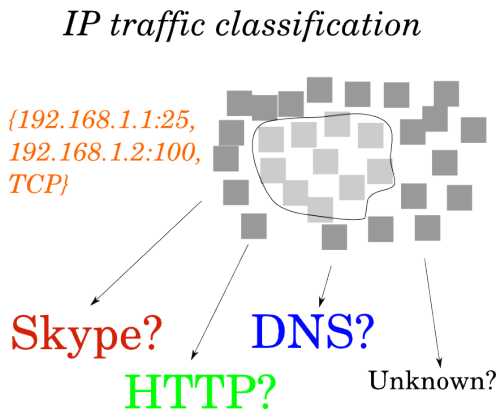

Figure 2.2: Illustration of the Traffic Classification problem.

correspond to real IP packets, e.g. NetFlow aggregates [35]. Moreover, we may choose arbitrary labels if that would better suit the network management goal. In general the labels are not required to resemble application or protocol names.

That said, let the *problem of Traffic Classification* be solving Eqs. (2.1), (2.2), and (2.3) under a given set of requirements and constraints. In other words, let the TC problem be answering the question on how to solve the TC task in practice, with minimum resources and maximum performance.

Thus, the actual solution to the TC problem will heavily depend on the network and on the practical goal one wants to achieve. A TC system designed for a branch office firewall will likely be much different than a TC system designed for visualizing traffic on a core Internet router.

2.3 Design and taxonomy of TC systems

There are several categories of requirements that one should consider when tackling the TC problem in practice, for instance:

- **Accuracy and scope:** e.g. minimum precision and recall, protocols and applications of interest, resilience to background traffic;
- **Available data:** e.g. access to packet payloads and traffic samples, support for anonymized addressing, packet sampling, and asymmetric routing;
- **Computing resources:** e.g. CPU and energy limits, memory capacity;
- **Timing constraints:** e.g. support for immediate classification and wire-speed processing for high traffic volumes;
- **Flexibility:** e.g. temporal stability of traffic models, reliable performance across different networks;
- **Usability:** e.g. overall simplicity, costs of updates, debugging and self-improvement facilities, source code access.

Obviously, it is often difficult to completely satisfy requirements in one category without compromising in the others. One should clearly state the overall network goal and the target operating environment prior to choosing an existing TC method or developing a new algorithm.

If a particular requirement cannot be satisfied for the whole traffic, the Waterfall architecture described in the thesis can be used to implement different sets of requirements for different portions of the Internet traffic. Every module in the cascade can be designed according to its own set of performance constraints. In Chapter 8, we will discuss how to automatically optimize such a cascading TC system for the overall network management goal.

In order to facilitate the process of creating a TC system, let us propose the following set of design questions that help in specifying the requirements:

1. What is the general context of the TC system being designed? What is the purpose of the planned TC system and how will it be used?
2. What can be said about the target network link? Should one expect packet loss, asymmetric routing, or DNS / HTTP traffic being handled through an internal caching server?
3. What are the constraints on the data one can extract from the traffic? Is there full access to the packet contents, or only to the packet headers? What are the privacy restrictions? Is it possible to read the traffic at wire speed or should sampling be employed instead?
4. What is the expected result granularity? Is it sufficient to know just the network protocol or the specific software name is required? Should the system identify the content transported inside protocols and tunnels?
5. What are the timing constraints? Should the system identify each flow instantly, after the very first few packets, or can the connection last a few minutes before being classified?
6. What are the expected traffic volume and available hardware resources? Should the system process monitored traffic in real-time (on-line operation), or is delayed processing acceptable (off-line operation)?
7. What is the set of protocols of interest? What is the set of the background applications to be ignored? Is it enough to provide information on whole traffic, just the 95% of bytes or packets, or just some selected applications?
8. What are the constraints for training the system? Will the system be trained in a laboratory, or rather in a specific production environment? Should the model be transferable across different networks?
9. For how long should the system provide accurate results? How the ground-truth labels will be acquired and updated? How the updated classification model will be transferred to the production environment?
10. What is the desired accuracy of the system in terms of bytes, packets, and flows?

Note that the question on expected accuracy is listed last. Although the TC literature often focuses exclusively on the metrics that measure the classification accuracy, these metrics in practice heavily depend on the issues considered in the questions 1-9 above. Thus, one should not compare the classification performance of various TC methods before specifying the context of the system. For an alternative set of questions for choosing a TC method, see [114].

Let us also introduce the notion of *taxonomy* for TC. Several papers suggested various ways to categorize TC algorithms depending on their characteristics: [28, 40, 70, 85, 86, 105, 114, 145]. However, the literature lacks a common taxonomy to describe the state of the art in TC. The most important works describe the existing TC algorithms in terms of the technique, accuracy, and

traffic features [28,105], but such a simple taxonomy is often too coarse-grained.

Instead, in [85] Khalife et al. present a multi-level taxonomy, which was used in a survey of methods for encrypted TC [147]. The authors propose to characterize each classification method at three different levels—classification input, technique, and output—in a hierarchical taxonomy:

- **Classification Input:**
  - traffic payload
  - traffic attributes: packet-level attributes (e.g. packet headers, sizes, inter-arrival times), flow-level attributes (e.g. flow size, duration), host-level attributes (e.g. number of connections and opened ports), and host community-level attributes (e.g. graph metrics: connection degree, graph diameter)
  - hybrid & miscellaneous
- **Classification Technique:**
  - payload inspection
  - simple statistical: basic statistical, heuristics, profiles
  - statistical ML: unsupervised, supervised, semi-supervised, reinforcement learning
  - graphical: graphlets, motifs, social networks
  - hybrid & miscellaneous: content-aware, distributed, multi-classifiers
- **Classification Output (Outcome):**
  - traffic classes: traffic cluster (e.g. bulk, small transactions), application type (e.g. game, browsing, chat), application protocol (e.g. HTTP, HTTPS, FTP), application software (e.g. specific BitTorrent client), fine-grained class (e.g. Facebook chat, Google search, Skype call), anomaly
  - traffic objects: packet, flow, host, host community

Let us conclude that solving the TC problem is complex and depends on the overall goal, but the design process can be broken into several simpler issues. Similarly, the Waterfall architecture presented in the thesis allows dividing the complex TC implementation task into several subproblems.

2.4 Practical applications

Historically, the first practical applications of TC were intrusion detection [67] and traffic prioritization [127]. However, nowadays TC—and in general, various traffic analysis methods—are used for many different purposes related to network management and the Internet, for example:

- Internet measurements, e.g. tracking usage of Internet applications [81];
- Traffic visualization, e.g. plotting main traffic components in time [54];
- Multipath routing, e.g. using cheaper links for bulk download traffic;
- Traffic shaping, e.g. implementing QoS [24, 132];
- Data security, e.g. disallowing tunnels and cloud storage applications;
- Firewalls, e.g. blocking malware and spam [95, 113];
- Traffic modeling, e.g. for collecting traffic samples;
- Marketing, e.g. profiling Internet users [115];

- Mass surveillance, e.g. the NSA XKeyscore system [71, 107].
- Internet censorship [49]
- Society manipulation [94]

We highlight that the last few applications, arguably unfortunate, were not the original motivation for TC. In fact, TC *per se* does not imply surveillance of any kind, and can be used for objectively good purposes, e.g. improving Internet reliability and security. However, it is the decision and responsibility of the TC designer why a particular system is created.

Apart of the government agencies, TC is used by various scientific and engineering organizations, mainly as the fundamental tool for Internet research [25, 96, 102, 140, 142]. Some ISPs use TC for classifying users into various groups depending on their interests and browsing habits, e.g. Orange Poland asks for permission on monitoring the IP traffic of an individual mobile Internet user for marketing purposes [126].

Various commercial companies and open source projects offer products implementing TC. Either in form of dedicated hardware—e.g. from Cisco [33], Juniper Networks [80], PaloAlto Networks [110], SolarWinds [131], Plixer [116], Allot Communications [7]—but also as specialized software—e.g. from ipoque [78], nTop [109], Bro IDS [20], Suricata IDS [133], Tstat [138]). Recently, TC has been offered as Software as a Service (SaaS) by Talaia [136].

Chapter 3

Machine Learning

In the previous Chapter, we introduced TC, giving examples of its applications, presenting intuitions and definition of the TC problem, and proposing some design methodologies. In this Chapter, we introduce foundations that underlie the recent developments in TC: the field of Machine Learning (ML).

3.1 Introduction

In general, ML allows for programming computers with data instead of programs. It aims to develop techniques by which experience leads to improved performance on computing tasks [43]. T. Mitchell defined ML as consisting of algorithms that improve their performance P on some task T through the experience E; a well-defined ML problem is thus given by $< P, T, E >$ [98]. That said, applying ML for TC means using algorithms that improve their performance P on solving the TC problem T using labeled samples of traffic E (see Section 2.2 for details). TC uses a branch of ML called Pattern Recognition (PR), which aims at automatic recognition of complex patterns in data [17].

Let us consider the plot given in Fig. 3.1: it presents real data extracted from IP packets of Skype (red points) and Google Hangouts (blue points), both of which are popular computer programs for videoconferencing. The plot visualizes the dependency between inter-packet time gaps (horizontal axis) and payload sizes (vertical axis), which exposes apparent differences in the IP traffic generated by these programs. Now, in order to gain some intuition, let us state that the goal of ML is coloring the plot presented in Fig. 3.1, but with all of the points initially black. In general, the points can represent any object, take arbitrary values, and lay in a d-dimensional space, $d \in \mathbb{N}$.

One of the main approaches to solving such a problem, called *supervised learning*, requires prior samples of how the plot should be colored. In this case, the algorithm classifies a particular point by comparing its location to the knowledge extracted from the *training data*, e.g. using the typical areas for blue and red. The other prominent approach, called *unsupervised learning*, operates without prior knowledge. Instead, in such a case the computer discovers intrinsic structures in the data, e.g. clusters with different densities.

For an example of supervised learning, consider Fig. 3.2, which presents the outcome of using our dataset for training the k-Nearest Neighbors (k-NN) al-

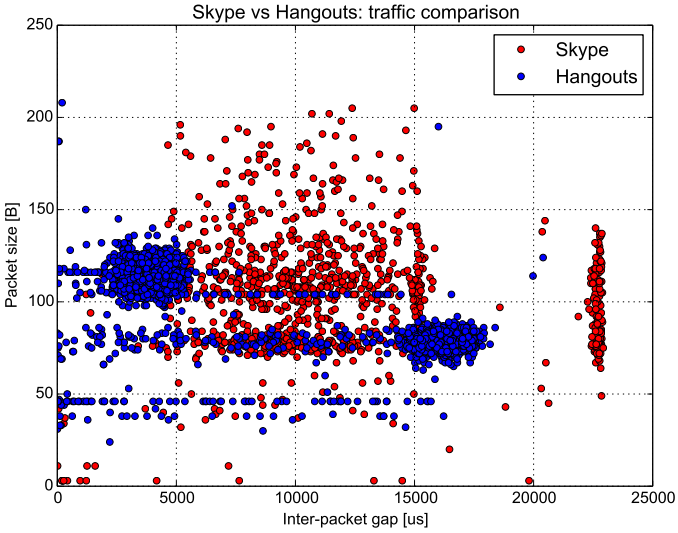

Figure 3.1: Comparison of voice call traffic in Skype vs. Google Hangouts.

gorithm, a popular ML method that exploits the closest data points. Again, the points correspond to IP packets of Skype and Hangouts, but now the background color visualizes *decision boundary*, i.e. rules for classification of future, yet unseen data. Note that the decision boundary is not perfect in a few places, e.g. for the area close to the center, it is impossible to completely separate the blue points from the red ones in the training data. However, the simple k-NN algorithm is often good enough and was successfully applied in TC, e.g. [30].

For an example of unsupervised learning, consider Fig. 3.3, which presents 4 intrinsic structures found in the data, without any prior knowledge. Note that now there is no direct correspondence between the color and the traffic type. Instead, the color indicates that a given point belongs to a cluster that is relatively coherent and separated from the rest of the data. Note that clustering algorithms do not provide final answers, but their results help in exploring unlabeled datasets and developing more sophisticated tools, e.g. [72].

3.2 Supervised learning

Supervised learning infers functions from labeled training datasets [100]. Let $\mathbf{x} \in X$ denote an input vector and $y \in Y$ denote its true output label (target). X and Y are arbitrary, but if Y is discrete and finite, we solve a *classification* task, whereas if Y is continuous, we solve a *regression* task. A supervised learning algorithm finds a function $f : X \to Y$ such that

$$f(\mathbf{x}) = y, \tag{3.1}$$

which usually is difficult to satisfy for arbitrary $\mathbf{x}$ outside the training dataset. Thus, let $g : X \times Y \to \mathbb{R}$ be a *scoring function* that quantifies how well the

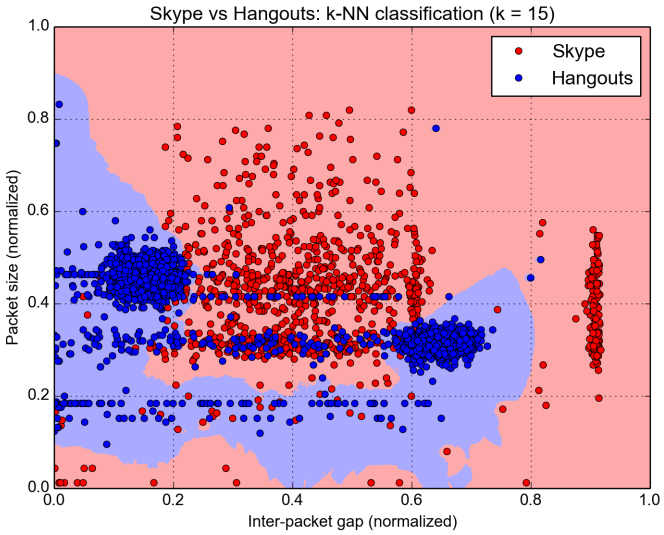

Figure 3.2: k-NN supervised learning algorithm: the original training data (points) vs. the decision boundary learned from the data (background).

output y matches the input $\mathbf{x}$. In such a case, $f(\mathbf{x})$ is given by

$$f(\mathbf{x}) = \arg\max_{y} \; g(\mathbf{x}, y), \tag{3.2}$$

which selects the y that maximizes the scoring function. As an example, consider a probabilistic scoring function $g(\mathbf{x}, y) = P(y|\mathbf{x})$, which makes f select the most plausible cause y given the evidence $\mathbf{x}$. However, in general g can be arbitrary.

Note that if we consider $\mathbf{x}$ and y as random variables, then supervised learning assumes $\mathbf{x}$ and y to be statistically dependent. Otherwise, any experience learned from the training dataset would be useless: the system would learn coincidences instead of fundamental rules. In other words, it is impossible to find patterns in chaos.

In real-world ML problems, one rarely starts with data in the abstract form of $(\mathbf{x}, y)$ pairs. Thus, a typical design process of an ML system starts with adequate sensing of real-world phenomena and representing them with numbers. For example, if one wants to build a face recognition system, then the first step would be to capture human faces with a digital camera, normalize for different lightning and postures, convert to grayscale, and rescale the pictures to common resolution of $w \times h$ pixels. Next, each real face F would be described with a (wh)-dimensional *feature vector* of numbers x_i representing the light intensities at consecutive pixels in the face image:

$$F \to \mathbf{x} = \{x_1, \cdots, x_i, \cdots, x_{(wh)}\}, \quad 0 \leq x_i \leq 1. \tag{3.3}$$

In ML, the process of converting real-world phenomena to raw data is known as *sensing* (e.g., capturing a digital picture of F); the process of detecting and normalizing objects in the raw data is known as *segmentation and grouping* (e.g., face detection); finally, the process of converting objects to feature vectors is

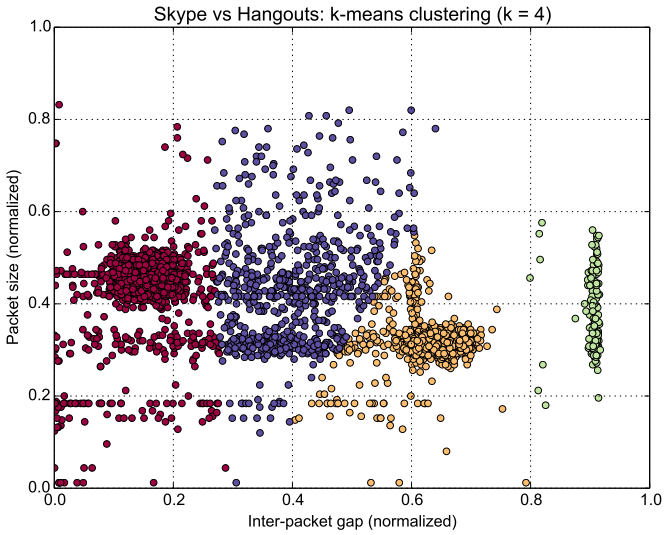

Figure 3.3: Clustering using the k-means unsupervised learning algorithm: four clusters found in the input data without using any prior knowledge.

known as *feature extraction* (e.g., producing the vector $\mathbf{x}$) [17]. All these phases have a fundamental impact on the performance of the whole ML system, but are highly domain-specific. In TC, sensing and segmentation deal with capturing and grouping IP packets, whereas feature extraction deals with reducing great amounts of traffic data down to just the essential features. We will further discuss these topics and show examples in Chapters 4 and 5.

The second step in the design process is choosing the method for making decisions (the g function in Eq. 3.2). In a well-known survey on statistical PR, A. Jain et al. list four best known approaches [79]: template matching, statistical classification, syntactic or structural matching, and neural networks. The statistical approach is widely used in TC: we depict its branches in Fig. 3.4. However, as the scoring function can be arbitrary, let us mention *decision trees* as an example of a non-metric approach that goes beyond statistical models [17].

In statistical PR, we consider $\mathbf{x}, y$ as random variables. In Fig. 3.4, we show various approaches to statistical PR (which also covers unsupervised learning for completeness). In general, the choice of the PR algorithm depends on how much we know about $\mathbf{x}, y$. If the Conditional Probability Densities (CPDs) are known, then we apply the "optimal" Bayes decision rule. However, usually the CPDs are not known *a priori* and we have to estimate them from the training dataset. In such a case, if the *form* of the CPDs is known (e.g. Gaussian)—but the parameters are missing (e.g. mean and variance)—then we only deal with a parametric decision problem: we replace the unknown parameters with their prior distributions. On the other hand, if the form of the CPDs is unknown, we either estimate the density functions from data (e.g. using kernels), or we directly construct the decision boundary, as in Fig. 3.2 for the k-NN algorithm [79]. The process of building the model from data is called *training*.

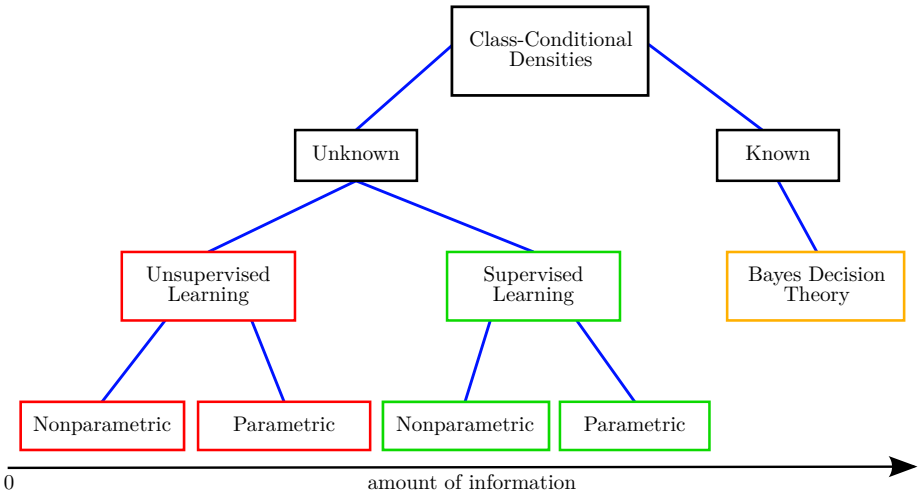

Figure 3.4: Approaches in statistical Pattern Recognition (based on [79]).

3.3 Training and testing

The common element of every supervised learning algorithm is that it needs quality training and testing data, which often is a scarce resource. For example, a common problem for research in TC is that ISP companies rarely share IP traffic data due to customer privacy and law regulations. Another important problem is reliable *ground-truth* data, i.e. knowledge on the true label for each feature vector. In TC, the common technique for obtaining ground-truth information is using an existing, state of the art method.

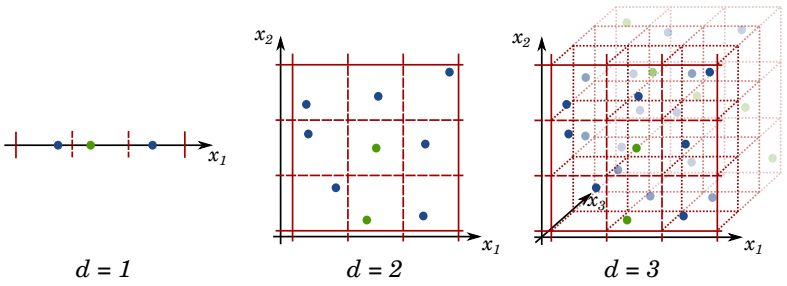

Figure 3.5: The curse of dimensionality. Adding dimensions increases the need for training data exponentially. Colors represent the target class (based on [17]).

The amount of data needed for training grows rapidly with the number of features one wants to use. In general, the number of training instances grows exponentially with the number of dimensions in the feature vectors [17]. Thus, using more features with the same training dataset may degrade the performance of an ML system. This paradoxical phenomenon is known as the *curse of dimensionality* or the *peaking phenomenon*. In order to demonstrate it, let us consider a simple look-up classifier—that is, let f in Eq. 3.1 represent fetching the cell $\mathbf{x}$ from a d-dimensional table X. If we divide each dimension in X into n regular cells, we end up with n^d cells total. Thus, in order to fill

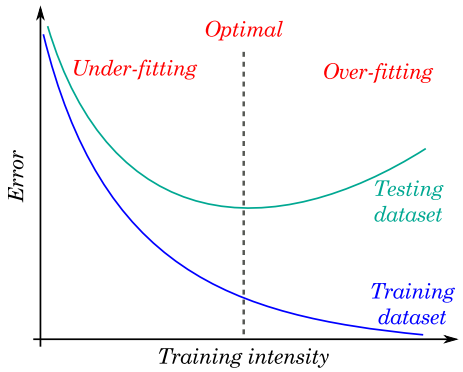

Figure 3.6: Over-fitting: too intense training can cause the ML system to memorize training dataset instead of learning the general rules behind it.

each cell in X, we would need at least the same number of training instances. See Fig. 3.5 for an illustration. In practice, a common rule of thumb is to use at least ten times more training instances per class than the number of features. The more complex the learning model, the higher should this ratio be to avoid the peaking phenomenon [79].

On the other hand, too much training can sometimes harm the performance of an ML system. In Fig. 3.6, we illustrate the dependence of performance on "training intensity", i.e. on a general concept that represents model complexity or the number of training cycles. The picture shows that by more intense training we make the training error smaller, but after some threshold the system starts to memorize the training dataset instead of learning the rules that underlie it. That is, the system becomes unable to generalize and to give proper outputs for the testing set. We call this phenomenon *over-fitting* and usually recognize it by good performance on the training data, but bad performance on the testing data. On the other hand, if both error rates are high, then the system likely *under-fits* the data and a more complex model is needed. For optimal operation, we should limit the training to the point in which both error rates are low.

A common technique for evaluating ML systems is *cross-validation* [17], which improves reliability of the performance estimates. In this approach, the whole dataset is randomly split into N subsets (often $N = 10$), and then $N-1$ of the subsets are used for training, while the remaining subset is used for testing. The procedure is repeated N times for every possible choice of the testing subset, and the performance metrics are then averaged. This reduces the chances for possible over-fitting, as the system is evaluated N times on different data.

3.4 Performance metrics

There are many popular metrics that measure the performance of a classification system [86]. The simplest is *accuracy*, which gives the number of properly classified instances, usually given as a ratio vs. the number of all instances. The more likely the system is to give the proper answer, the higher is its accuracy. However, this simple metric neglects the types of errors made during evaluation, so in practice more sophisticated measures are used: either the pair of *True*

Message type (ground-truth)	Classification result	
	Spam	Legitimate
Spam	*True Positive (Correct)*	*False Negative (Type II error)*
Legitimate	*False Positive (Type I error)*	*True Negative (Correct)*

Table 3.1: Possible types of outcome when evaluating a spam detector.

Positives and *False Positives*, or the pair of *Precision* and *Recall*.

In order to define these metrics, consider an example of a spam filter, which is a binary classification problem. Tab. 3.1 presents possible types of outcome of the system, depending on the kind of input message. The table defines names for two different types of correct answers—True Positives (TPs) and True Negatives (TNs)—and the names for two different types of incorrect answers—False Negatives (FNs) and False Positives (FPs). The percentages of these types of answers are often used as performance metrics, e.g.:

$$\%TP = \frac{TP}{TP + FN} \cdot 100\%, \quad \%FP = \frac{FP}{FP + TN} \cdot 100\% \,. \tag{3.4}$$

Intuitively, a high False Negative rate for a spam filter means the user would receive many unsolicited e-mails, while high False Positive rate means the user would loose many legitimate e-mails (due to invalid classifications as spam).

Similarly, precision and recall are defined as:

$$\mathit{Precision} = \frac{TP}{TP + FP} \,, \quad \mathit{Recall} = \frac{TP}{TP + FN} \,. \tag{3.5}$$

High Precision of a spam detector means that for all messages classified as spam, high proportion of them are truly unsolicited (which is known *a priori* from the ground-truth). However, achieving high Precision is easy if we neglect Recall, that is, we just skip the difficult cases. For this reason, Recall measures how many of the unsolicited messages known *a priori* were classified as spam. Thus, one should always consider Precision together with Recall, and optimize a classification system accordingly. For some applications, like intrusion detection, an overly cautious system is acceptable (trading Precision for Recall); for other applications, like traffic sampling, missing some IP flows might be tolerable (trading Recall for Precision).

Precision and Recall have an accompanying metric of *F-measure* (or F_1-*score*) that combines both of them using a harmonic mean:

$$\mathit{F\text{-}score} = 2 \cdot \frac{\mathit{Precision} \cdot \mathit{Recall}}{\mathit{Precision} + \mathit{Recall}} \,. \tag{3.6}$$

The *F-measure* is high if and only if both Precision and Recall are relatively high. Thus, it is a convenient way to express the performance of a classifier using a single quantity.

3.5 Multiple Classifier Systems

It is possible to combine many different classifiers to make them work together on solving classification tasks [88]: such systems are called Multiple Classifier Systems (MCS), combining classifiers, or classifier ensembles. Furthermore, MCS can be divided into *Classifier Fusion*, where all of the constituent classifiers are actively used and their final outputs are combined, and *Classifier Selection*, where the output of only one of the classifiers is selected as the ensemble output. Figure 3.7 illustrates the MCS idea.

For example, in Classifier Fusion—similarly to the democratic system—one can collect the outputs of a pool of classifiers, and choose the most popular class label as the final decision (the *Majority Vote* technique). On another hand, in Classifier Selection—similarly to a council of ministers—the classifiers can have their areas of competence, and be used only for the subset of tasks that match their capabilities (the *Classifier Selection* technique).

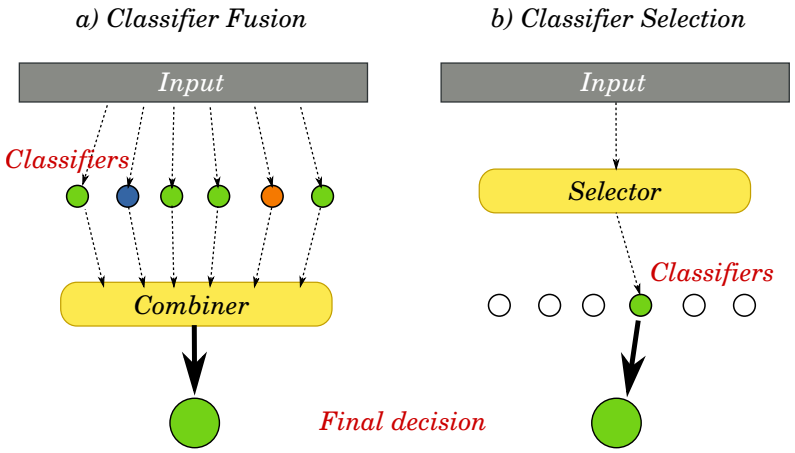

Figure 3.7: Multiple Classifier Systems: Fusion (a) and Selection (b).

Note that one might argue that, in an MCS system, the constituent classifiers are just fancy features used by an ordinary classifier built on top of them. For instance, neural networks can in theory approximate any function with arbitrary precision with a finite set of model parameters [148]—thus, we could consider the neurons as simple base classifiers, and the whole network as an ensemble. However, in practice we need a simpler tool than a huge neural network with multitude of parameters, i.e. we need tools that are easier to manage and interpret. Thus, MCS helps us by breaking complex tasks into smaller, manageable blocks: it is often easier to combine many acceptable solutions of a classification problem than to find the optimal solution that exploits all of the data features at once.

There are three fundamental reasons for why classifier ensembles can perform better than any of their constituent base classifiers [42, 88]:

- **Statistical Reasons.** In case there is little training data for a problem, some algorithms may produce suboptimal models in each run (e.g., k-NN)

or for each subsample of the input (e.g., decision trees). By averaging the classification outputs of these models, we are minimizing the bias introduced by poor training data.

- **Computational Reasons.** Some training algorithms, e.g. error back-propagation in neural networks, are only guaranteed to converge to a local minimum. Thus, averaging the models trained from different starting points is more likely to approach the global minimum than using any of the base classifiers alone. Moreover, some problems require data from quite different sources, e.g. activity recognition of a smart-phone user may require analyzing data from an accelerometer, a GPS chip, and a microphone. It is easier to process these data using three separate classifiers and then combine their outcomes rather than digesting multi-modal data in a single algorithm.
- **Representational Reasons.** In many cases it is impossible to describe a complex decision boundary for a classification problem with a simple model, e.g. represent a polynomial function with a single linear function. By averaging several simple models in an MCS system, one can represent more complex decision boundaries.

In order to design and describe an ensemble classifier, one needs a taxonomy. In [122], Rokach proposes an MCS taxonomy that considers 5 dimensions: Combiner Usage, Classifier Dependency, Ensemble Diversity, Ensemble Size, and Cross-Inducer. This taxonomy was interpreted by Kuncheva in [88], which we reproduce below with minor adaptations:

- **Output Combiner (Selector)**:
  - Not specified
  - Specified: Non-trainable, Trainable, or Meta-classifier
- **Base Classifier Dependency**
  - Independent training
  - Dependent training (incremental)
- **Ensemble Diversity**
  - Training base classifiers on different parameters
  - Resampling the training data
  - Partitioning the training data (horizontal / vertical)
  - Different target class labels
  - Different base classifiers
- **Ensemble Size**
  - Fixed in advance
  - Selected while training
  - Pruning (overproduce and select)
- **Universality**
  - Specified base classifiers
  - Any base classifiers

In this thesis, we present a cascading classifier for Internet traffic, which belongs to the Classifier Selection branch of MCS systems. In terms of the presented taxonomy, our classifier has the following features: non-trainable selector (but programmable), independent training of base classifiers, diversity through different classifiers (operating on vertically partitioned data), ensemble pruned through optimization, and applicability to any base classifiers. Below, we give illustrative examples for Classifier Fusion and Classifier Selection, describing the methods in terms of the presented taxonomy.

3.5.1 Behavior Knowledge Space

One of the prominent examples of Classifier Fusion is Behavior Knowledge Space (BKS), introduced by Huang and Suen in [75]. It belongs to a larger group called *multinomial methods*, where we look for the class label y with the highest posterior probability P given ensemble answers $\mathbf{s}$. Let $\mathbf{D} = (D_1, \ldots, D_L)$ be an ensemble of L base classifiers that classify an input feature vector $\mathbf{x}$, giving L answers $\mathbf{s} = (s_1, \ldots, s_i, \ldots, s_L)$, $s_i \in Y$. Then, in multinomial methods, we look for

$$f(\mathbf{x}) = \arg\max_{y} \; P(y|\mathbf{s}), \tag{3.7}$$

which is similar in spirit to Eq. 3.2, but introduces an intermediate layer of ensemble answers $\mathbf{s}$. In BKS, the optimization is realized using a look-up table (the BKS table), which is created during a training phase, prior to ensemble operation. Fig. 3.8 illustrates the classification algorithm.

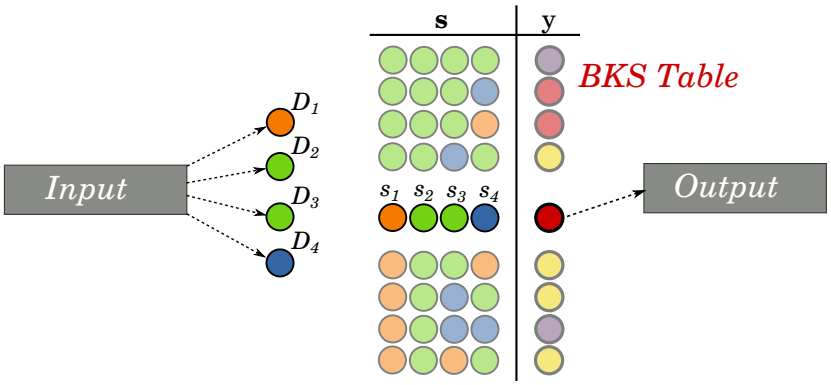

Figure 3.8: Behavior Knowledge Space: the answers $\mathbf{s}$ from an ensemble $\mathbf{D}$ are translated using the BKS look-up table into the final class label y.

The BKS table is learned using a labeled dataset: each sample is classified using the base classifiers, their answers $\mathbf{s}$ are collected, and the ground-truth label is stored under the table cell $\mathbf{s}$. The most frequent label in a cell wins; in case of a tie, any of the most frequent labels can be chosen. If an empty cell is found in the BKS table during later operation, the class is chosen using a majority vote or randomly.

In terms of the MCS taxonomy, BKS has the following characteristics: trainable combiner, independent base classifiers, ensemble diversity through different classifiers (possibly with different target classes), fixed ensemble size, and applicability to any base classifiers. A notable feature of BKS is easy translation between various sets of class labels: as visible in Fig. 3.8, the ensemble output y uses a different set of labels (colors) than any of the base classifiers. For example of vehicle classification, each base classifier could analyze only a certain feature of a vehicle (e.g., color, height, number of wheels), while the BKS combiner could give the final class (e.g., car, bike, truck).

3.5.2 Cascade Classifiers

Cascade Classifiers (CC) belong to the family of Classifier Selection and—by their very nature—are radically different than BKS. Instead of querying all base classifiers $D_i \in \mathbf{D}$ for input $\mathbf{x}$, CC queries the classifiers one by one—i.e., D_1, then D_2, then D_3, etc.—and the first that answers with a class label for $\mathbf{x}$ wins. If a classifier D_i is unable to make a reliable decision, for instance due to a tie between two classes, the input $\mathbf{x}$ is conveyed onward to the next classifier D_{i+1}. In case of no more classifiers, $\mathbf{x}$ leaves the cascade without a label.

The original publication by E. Alpaydin and C. Kaynak [8] that introduced CC describes a two-stage system, where the first classifier learns the general rule, and the second classifier learns the exceptions. Nowadays, the literature treats CC as a more general MCS concept that belongs to Classifier Selection, where a cascade can consist of many base classifiers [1,32,88]. Fig. 3.9 illustrates CC in this context. In Classifier Selection (a), the problem space is divided *a priori* into many competence areas, and each input $\mathbf{x}$ is assigned to exactly one classifier D_i according to the matching area. In Cascade Classification (b), each input $\mathbf{x}$ can enter many classifiers, but only the one that is certain about the label will make the final decision.

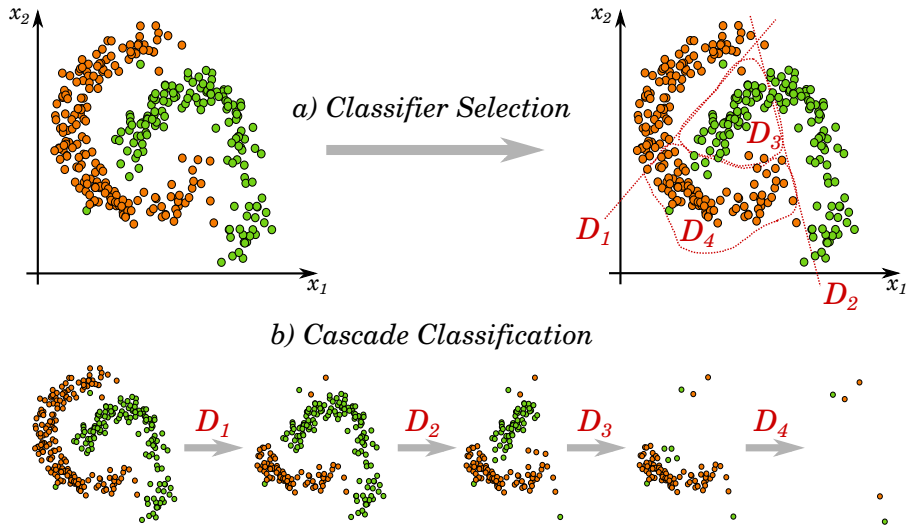

Figure 3.9: Cascade Classification (b) in the context of Classifier Selection (a). The spots represent inputs $\mathbf{x}$, their color shows the target class.

The differences between CC and BKS have deep consequences in the design of their base classifiers: in CC, they need to support the "reject" option (the "unknown" label)—in order to pass $\mathbf{x}$ from D_i to D_{i+1}—and they need to have a minimum rate of False Positives—in order to avoid mistakes at all costs. Thus, any quantitative comparison between BKS and CC systems comprising of same base classifiers seems unreasonable, as these MCS techniques have different principles of operation. In BKS, a base classifier that is accurate 75% of time can positively contribute to the ensemble, while in CC such an element would ruin the performance. On the other hand, CC is computationally faster than BKS by design, as it runs all classifiers only for a fraction of inputs.

In terms of the MCS taxonomy, CC in general has non-trainable selectors, independent or dependent training (similar to boosting [88]), diversity through different base classifiers (parameters, samples, partitioning, etc.), flexible ensemble size (with various techniques for optimization), and applicability to any base classifiers that support the reject option.

We will revisit the topic of Cascade Classification again in Chapters 7 and 8 of the thesis, where we describe the Waterfall Traffic Classifier and give a method for optimizing its performance.

Chapter 4

Datasets and Tools

So far, we discussed the task of Traffic Classification (TC) and showed that Machine Learning (ML) is a suitable tool for solving the TC problem. However, in practice one needs *datasets* and *software tools* in order to develop and properly evaluate TC methods. Thus, in this chapter, we discuss how to obtain datasets of raw IP packet traces and we present relevant software tools that convert these raw datasets into collections of classification objects.

4.1 Introduction

The process of acquiring and processing datasets heavily depends on the design of a particular TC system, as discussed in Chapter 2. Again, a TC system designed for a branch office firewall will likely require different data than a TC system designed for visualizing traffic of a core router. Whereas raw IP packet capture files may be suitable in the former case, less detailed information may be more efficient for the latter case, e.g. NetFlow objects.

Other dataset requirements come from the ML field. In Chapter 3, we introduced the concept of the curse of dimensionality, which causes the need for large amounts of data if one wants to evaluate many features in IP traffic. Moreover, supervised learning requires accurate ground-truth labels on each classification object, which is not readily available in the IP traffic itself.

Thus, each TC system requires large datasets of adequate IP traffic objects labeled with ground-truth information. In practice, such datasets are difficult to obtain, creating one of the largest barriers in TC research [40, 124]. Large Internet carriers rarely share IP traffic data outside of their organization due to privacy and security of their customers. Moreover, sharing data generates operational costs of additional engineering work and data storage. Still, some Internet carriers make their IP traffic available to affiliated researchers under a Non-Disclosure Agreement (NDA). This allows select scientists to work on high-quality data, but limits credibility of their findings—as they cannot be independently reproduced—and divides the TC research community by excluding non-privileged scientists.

Some researchers emphasized the need for public datasets that could be used as a common reference for evaluating TC methods [40,124]. Unfortunately, as of the time of this writing, TC still lacks even a single *canonical* dataset, contrary

Dataset	Year	Payload?	Anon.?	GT?	Diverse?	Large?
MAWI [96]	2017		✓		✓	✓
CAIDA [26]	2016		✓		✓	✓
UPC-CBA [23]	2015	✓		✓		
WAND [143]	2011	4 bytes	✓		✓	✓
UNIBS [50]	2009		✓	✓	✓	
Tstat [141]	2008	✓	✓	✓		

Table 4.1: Publicly available TC datasets. Column "Year" gives the last update, "Payload?" marks availability of packet payloads, "Anon.?" marks IP address anonymization, "GT?" marks presence of ground-truth labels, "Diverse?" marks collection of diverse applications, and "Large?" marks large dataset size.

to the other computer science areas that depend on ML, e.g. Optical Character Recognition (OCR), which has the MNIST database of handwritten digits [89]. However, a few public TC datasets exist, but are of limited applicability and thus do not sufficiently solve the problem of no common reference.

We list some of the publicly available TC datasets in Table 4.1. Their usability is limited mainly due to: (1) cropped packet payloads, (2) anonymized IP addresses, (3) no ground-truth information, (4) poor protocol diversity, (5) small dataset size, (6) old age, or any combination thereof. The limitations (1)-(3) may cause a dataset to be inadequate for supervised learning, which requires the target labels for training. Having neither the packet payloads (1) nor the real IP addresses (2) makes it impossible to obtain the ground-truth information if it is missing (3). Still, such datasets may be useful for unsupervised traffic analysis. The limitations (4)-(6) may cause a classification model trained on such a dataset to be inadequate for operation in modern, real-world networks.

However, datasets representing just a few protocols, or even a single application, may be applied to independently develop a specialized classifier that will later be integrated in a larger Waterfall TC system. Thus, in the next section we present a software tool for capturing packet traces of a single computer program running under the Linux operating system, which was originally published in [61]. Our overall goal here is to allow the researcher to automatically generate and collect packet traces of select network applications using Graphical User Interface (GUI) automation tools, such as Sikuli Script [60, 130, 156].

4.2 Tracedump: single application sniffer

One of the most popular network monitoring tools are *packet sniffers*, i.e. computer programs that intercept IP packets sent or received on a network host. The output of a sniffer gives insight into how exactly the network operates, and thus can be useful for solving connectivity issues. For example, in order to diagnose a routing problem, one would observe IP packets on the ingress and egress interfaces of an Internet router: if the traffic is visible on the ingress interface, but not on the egress one, then the router is not forwarding IP packets that enter the network.

While packet sniffers can monitor network interfaces, it is particularly difficult to capture only the packets belonging to a given application, i.e. the packets sent and received by a single process running on a particular host. This

is because packet sniffers were designed to be deployed on Internet routers, i.e. hosts that rarely generate their own IP traffic. Thus, a typical sniffer can easily monitor a network interface, but lacks sufficient granularity to inspect only a local process. Unfortunately, such a design influenced the operating systems: for instance, the Linux kernel lacks apparatus necessary for straightforward implementation of such a sniffer.

The ability to monitor just a single application is important for training TC systems, which need traffic datasets annotated with ground-truth labels. By employing a single application packet sniffer, the problem of ground-truth is mitigated, as the name of the application is known by design. In order to obtain an adequate quantity of training data, a single application sniffer may be combined with automation tools and run the analyzed application for a sufficiently long time. We acknowledge that synthetic traffic traces are of limited versatility, but may be important for development of specialized TC methods, e.g. Waterfall modules.

In this section, we present *tracedump*: a novel IP packet sniffer that intercepts packets belonging to a single application process. It employs several techniques in order to mitigate the lack of necessary mechanisms in the Linux kernel, particularly the `ptrace(2)` [92] system call and the BPF socket filter [97]. The sniffer attaches to a given process and monitors its system calls related to communication with the Internet. A list of local TCP and UDP ports is constructed on the fly and used for filtering out all the traffic not belonging to the application under interest.

4.2.1 Related works

The most prominent packet sniffer is `tcpdump`, which is based on the accompanying `libpcap` library [137]. Originally written in 1987 at the Lawrence Berkeley National Laboratory, it was published a few years later and quickly gained user attention. It runs on most UNIX-like operating systems—e.g. Linux, BSD, Solaris—and on Windows. Since its inception, `tcpdump` was cited by numerous scientific papers in the field of computer networks and is indeed the standard utility for capturing IP traffic. It established the PCAP output format, which is the most popular file format for storing IP packets, still being extended to support new functionality [151]. The `tcpdump` sniffer features a filtering mechanism, which allows the user to easily capture only the packets satisfying given criteria, e.g. TCP packets with the destination port equal 80. Unfortunately, the filtering mechanism does not support selecting the packets of a single application. For example, it is impossible to capture all traffic of a P2P application using port-based filtering, because P2P software uses dynamic port numbers.

`Wireshark` is also a very popular packet sniffer featuring a full-fledged GUI with lots of advanced features [152]. However, similarly to `tcpdump`, it does not allow tracing IP traffic of a specific process. Another packet capture software, `libtrace` [5], aims at addressing the weaknesses of `libpcap`. It supports many input methods and formats—and provides a very good performance at the same time—but none of them can capture the traffic of a single application.

In the field of TC, there are two notable software utilities for capturing traffic. F. Gringoli et al. in [73] present a system for collecting traffic traces that annotates IP flows with ground-truth labels. It works as the following: first, each host in the network uploads full list of its connections and their

application names to the border router; second, the border router captures the IP traffic flowing in and out of the network. Finally, the resultant traffic trace is annotated with appropriate ground-truth data using the connection lists uploaded by the hosts. Szabó et al. proposed a similar approach for Windows machines in [134]. However, these approaches do not solve the problem of single application diagnosis in the strict sense. First, they require a separate post-processing stage, which blocks real-time streaming operation; second, they may loose short-lived IP flows due to the non-zero time needed for uploading the list of active connections. Thus, these methods generally do not record DNS query-response traffic made by the monitored applications.

4.2.2 Problem analysis

Let us analyze—in a simplified manner—how a Linux application initiates a TCP connection and sends data to a distant host. The operating system provides an Application Binary Interface (ABI) for Internet communication by means of the system calls `socket()`, `connect()`, and `send()`. Thus, the application first calls the `socket()` function in order to get a unique handle for the connection. Then, the `connect()` function is called with the address of the remote peer, and finally the `send()` system call may be used to send the data.

There are two crucial issues one needs to realize when constructing a packet sniffer of a single application. First (A), the application does not handle construction of IP and transport protocol headers—it is the task of the operating system. Hence, it is not enough to monitor the data passed as arguments to `send()` in order to collect IP packets. Second (B), a call to `connect()` may generate packets before the call returns. Thus, a packet sniffer must react to `connect()` before it is executed in the kernel.

Unfortunately, it is quite hard to mitigate these problems using existing mechanisms present in the Linux kernel. One of the possible ways to write such a sniffer would be to extend the Linux `struct sk_buff` structure with a `pid` member holding the process ID number. For outgoing packets this would be trivial, but for incoming packets it could be quite troublesome. However, such approach would constrain the scope of our solution, due to the necessity to patch and recompile the operating system kernel.

It is possible to take care of (A) and (B) in user space, without modifying the kernel. A straightforward procedure would be to exploit the dynamic linker `ld.so` [90] in order to provide wrapper functions for system calls responsible for communication with the Internet. However, this would fail for statically compiled program binaries. Instead, our proposal makes use of the `ptrace()` process tracing facility.

4.2.3 Proposed solution

Tracedump is divided into three functional modules, implemented as threads: ptrace, pcap, and garbage collector (GC), depicted in Fig. 4.1. The ptrace module attaches to all threads of a given process, and using the Linux `ptrace()` function it maintains a list of all local TCP and UDP ports that the application is using. The pcap module operates like an ordinary packet sniffer, intercepting all IP packets on all network interfaces, at the kernel level (recall (A) from the previous section). Whenever the port list changes, a BPF filter is immediately

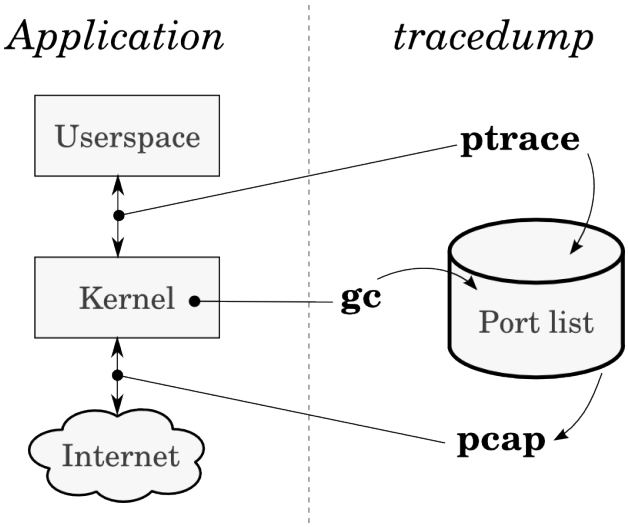

Figure 4.1: Architecture of tracedump. The port list is constructed by observing the kernel-userspace communication and is used for raw IP packet capture. The garbage collector (gc) thread periodically cleans up the list.

applied on the pcap sniffing socket, so that the packets not belonging to the monitored application are ignored. The BPF filter is updated before the kernel executes the original system call (recall (B)). The task of the garbage collector module is to detect ports that are no longer used. Each minute it reads the list of all active system connections, and it cleans up the list constructed by the ptrace thread.

The ptrace thread traces just three system calls: `bind()`, `connect()`, and `sendto()`. For the proposed architecture, we verified this is enough not to loose any IP packet—by means of Linux kernel analysis and examination of the usual path a user-space program takes to setup an Internet connection. For UDP and TCP servers, the application needs to call `bind()` to setup the local port number. For client programs, it will either call `connect()` or `sendto()`. In such a case, it may happen that the local port number is unspecified, thus the kernel will perform an "autobind" operation and allocate an ephemeral port automatically. However, due to (B), this is an undesirable situation, so tracedump splits the system call in such a case. First, it forces the process to call `bind()` with the port argument set to 0, i.e. it requests the automatic allocation to be executed. Then, the BPF filter is updated with the assigned port number, and finally the original call—either `connect()` or `sendto()`—is re-started. This is realized using machine code injection into the stack area of the monitored process.

The pcap thread attaches to the kernel using a `PF_PACKET` [91] socket, and writes captured packets to disk in the PCAP [151] file format. Whenever the list of local ports is changed, the BPF filter code is immediately rewritten and sent to the kernel using the `setsockopt()` system call.

A naïve solution to tracking the local port numbers that the application no longer uses would be to intercept the `close()` system calls. Unfortunately, using this technique alone is not sufficient to distinguish a `close()` call that effectively

```
1 root@pjf:~# tracedump ctorrent ubuntu-11.10.iso.torrent
2 pcap_init(): Writing packets to dump.pcap
3 (...)
4 Total: 673 MB
5 Creating file "ubuntu-11.10-alternate-i386.iso"
6 Press 'h' or '?' for help (display/control client options).
7 | 3/0/754 [1346/1347/1347] 672MB,0MB | 2157,0K/s | 1724,0K E:0,1
8 Download complete.
```

Listing 4.1: Using tracedump to capture BitTorrent traffic. A BitTorrent client ctorrent [39] is used for downloading a CD disk ISO image.

		Packets	Bytes	Ports
TCP	*Outbound*	249,575	17,698,668	77
	Inbound	503,730	718,005,933	
UDP	*Outbound*	2	160	1
	Inbound	2	192	
Total		753,309	735,704,953	78

Figure 4.2: Characteristics of BitTorrent IP traffic. Last column presents number of transport protocol ports as a sum for inbound and outbound traffic.

ends a connection from a `close()` call that only dissolves an association between a file descriptor and the socket. The latter may happen in case of multi-threaded applications, which may—or may not—share the file descriptor table amongst its threads. This depends on the detailed configuration of a particular thread, which is difficult to discover on a Linux machine. Thus, tracedump utilizes the conventional `procfs` network diagnosis interface, i.e. the `/proc/net/tcp` and `/proc/net/udp` special files. This interface is quite slow, hence a separate garbage collector thread is required in order to continuously re-read these files in an asynchronous manner.

4.2.4 Practical application

Tracedump has a simple command-line interface. Either a process ID or a command line is accepted as the program argument. Provided that the user has root privileges, it is possible to attach to any process in the system. Hence, tracedump can be used by system administrators for inspection of any user activity on a Linux server. The output of tracedump is in PCAP format, which may be further processed with traffic analysis tools like Wireshark or flowcalc. It is also possible to visually examine IP packets in real-time, by adopting the UNIX pipe mechanism and e.g. the tcpdump program.

Listing 4.1 presents an illustrative application of tracedump. In this example, an installation CD ISO disk image of a popular Linux distribution is downloaded using the BitTorrent [36] protocol. Our aim here is to roughly estimate the overhead of the BitTorrent protocol, in order to demonstrate tracedump.

In line 1, tracedump is started so it monitors a BitTorrent client application downloading a file. In line 2, the resultant PCAP file name is reported: `dump.pcap`. The download process completes in 6 minutes, attaining an average throughput of about 2 MB/s.

Tab. 4.2 presents brief characteristics of the generated IP traffic, obtained using utility programs of the libtrace library [5]. The resultant ISO file is 705, 998, 848 bytes long, hence the overhead of the BitTorrent protocol—including the network and transport protocols—is roughly 4.2% of the downloaded file size. This also includes all of the DNS query-reply traffic induced during the download process. Note that the output file is already an elementary dataset for developing a traffic classifier for the BitTorrent protocol.

4.2.5 Summary

Tracedump is a packet sniffer that lets for capturing traces of IP packets generated by a single application. This eliminates the problem of ground-truth, as the name of the application is known by design. However, in order to capture larger datasets of IP traffic, automation tools should be used in concert with tracedump. The obtained traces can be applied to develop specialized TC algorithms designed for the Waterfall architecture.

4.3 Flowcalc: flow analysis toolkit

4.3.1 Introduction

Flowcalc [58] is a software toolkit for converting raw IP traffic traces into more abstract data format, Attribute-Relation File Format (ARFF), which is more suitable for flow-based traffic analysis. In essence, Flowcalc calculates IP flow statistics from traffic datasets, e.g. PCAP files. From the perspective of ML, as described in Chapter 3, Flowcalc is responsible for segmentation, grouping, and feature extraction in a classification system: its ultimate result is a set of feature vectors.

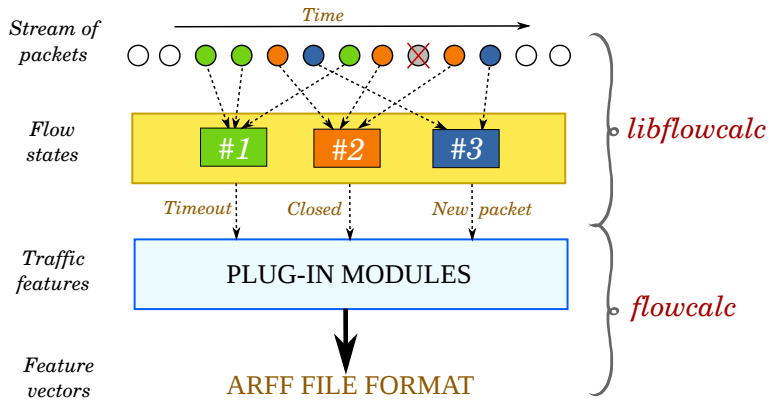

Figure 4.3: Architecture of Flowcalc.

The general architecture of Flowcalc is shown in Fig. 4.3. Internally, it consists of two software modules: *libflowcalc*, which is the engine of the whole system, and *flowcalc*, which integrates various traffic analysis plug-in modules and provides a command-line user interface.

In more detail, libflowcalc reads traffic data in various formats—e.g. PCAP, ERF and DAG—using the libtrace [5] library. Next, each IP packet is parsed, de-duplicated, and matched with its IP flow. After this step, several packet metadata is extracted, like network and transport protocol details, packet direction, timestamp, size, payload, and parts of the TCP session state. Next, the packet is passed to several plug-in modules, all of which maintain own state information and compute various features of the traffic, e.g. average packet sizes. Finally, when the last packet of a given flow is observed, libflowcalc will query each plug-in for a flow summary, which is provided as a partial feature vector. The results collected from all plug-ins are concatenated and a full feature vector that represents the flow is printed to the standard output.

The user-facing part of the system, flowcalc, is responsible for loading the plug-ins and generating the first features of each flow, e.g. flow id, source and destination IP addresses, and transport protocol port numbers. Also, users may modify run-time options through command-line arguments, e.g. desired list of plug-in modules, TCP session tracking options, and IP flow limits.

4.3.2 IP flow tracking

One of the crucial operations of Flowcalc is tracking IP flows, i.e. grouping IP packets into time-ordered sequences of all messages exchanged between two Internet peers in a single connection, e.g. a TCP stream. In order to accomplish this task, Flowcalc employs the libflowmanager software library [4] with a few customizations to the original flow tracking algorithm.

First, when a packet arrives, it is inspected for the information needed to re-construct the flow 5-tuple, i.e. a sequence of the following information:

- T_p, **transport protocol**: TCP or UDP;
- IP_{min}, **smaller IP address**: the address that evaluates as a smaller decimal number compared with the other peer, e.g. `173.194.223.101`;
- P_{min}, **port number of** IP_{min}: the transport protocol port number on the side of IP_{min}, e.g. `443`;
- IP_{max}, **larger IP address**: the larger of the two IP addresses, e.g. `212.106.181.99`;
- P_{max}, **port number of** IP_{max}: the port on the IP_{max} side, e.g. `53152`.

Next, the 5-tuple is used as a *key* that uniquely identifies the IP flow that matches the IP packet. The key is used to search through a hashing table that keeps track of all flows. If the lookup succeeds, the flow state is updated and the packet is passed onwards. If the lookup fails, it means that the packet is the first observation of a new IP flow. In such a situation, a new entry is created in the hashing table, with some additional information: flow identifier, packet timestamp, source and destination IP addresses, and the port numbers.

Then, we infer the packet direction: if the packet source IP address and port match the values of the *first* packet observed in given flow, we define the direction as *upload*; otherwise, we define it as *download*. This information is important, as traffic characteristics usually differ for the client (upload) and server (download) messages of the same protocol.

Next, we remove immediate duplicates of the same packet for given direction of the flow. This feature is realized by keeping track of the last values seen in specific packet headers, e.g. IPv4 Identification, UDP Checksum, etc. Also, if

the flow is TCP, we detect permanent packet loss in order to skip the flows with incomplete stream data.

Finally, we decide if the flow should be closed and the state information should be discarded. For UDP flows, we expire the flows after 2 minutes of inactivity [12]. For TCP, we expire the flows after 2 hours and 4 minutes [74], unless the connection is actively closed by both peers—in such a case, we expire the flow immediately. For half-closed and unestablished TCP connections, we expire the IP flows in 4 minutes [19]. As already discussed, when the last packet of a given flow is observed, i.e. when the flow is expired, Flowcalc will call all plug-in modules and collect their traffic features.

Apart from the expiration algorithm described above, Flowcalc can optionally expire flows based on an unconditional, constant timeout (e.g. after 10 seconds since the flow has started), or based on a packet counter (e.g. after 5th packet is observed for the given flow). This functionality resembles real-time traffic analysis, when certain requirements on the classification timeliness have to be met.

4.3.3 Available modules

Flowcalc is an extensible software toolkit, which allows for writing custom plug-ins that support any IP traffic classification algorithm. However, Flowcalc comes with several modules already available for computing popular traffic features:

- **stats**: computes the minimum, maximum, average, and standard deviation for the packet payload sizes and their inter-arrival times, separately for both flow directions;
- **counters**: counts number of packets and the total bytes of payload, for both directions;
- **pktsize**: records the payload size of the first 5 packets, for both directions;
- **payload2**: records up to 32 bytes of packet payloads, for both directions;
- **dns**: listens to DNS transactions and labels IP flows with their corresponding domain names;
- **coral**: classifies flows using CAIDA CoralReef port number classifier [84];
- **lpi**: classifies flows using libprotoident [6], a lightweight variant of DPI;
- **ndpi**: classifies flows using nDPI [108], a DPI software library;
- **web**: computes histograms of web request sizes and timing;
- **websize**: records payload sizes of the first 5 packets of web requests and responses, skipping handshake negotiations;

An illustrative example of Flowcalc output is presented in the Listing 4.2. Flowcalc was already used by other researchers, e.g. [57].

```
1  %% flowcalc run at Wed Dec 5 13:34:56 2016
2  % modules: ndpi
3  @relation '/trace/trace-2012.05.26-17:40:39.pcap.gz'
4  %% flowcalc 0.1
5  % fc_id: flow id
6  % fc_tstamp: timestamp of first packet in the flow
7  % fc_duration: flow duration
8  % fc_proto: transport protocol
9  % fc_src_addr: IP address of connection initiator
10 % fc_src_port: TP port number of connection initiator
11 % fc_dst_addr: IP address of remote peer
12 % fc_dst_port: TP port number of remote peer
13 @attribute fc_id numeric
14 @attribute fc_tstamp numeric
15 @attribute fc_duration numeric
16 @attribute fc_proto {TCP,UDP}
17 @attribute fc_src_addr string
18 @attribute fc_src_port numeric
19 @attribute fc_dst_addr string
20 @attribute fc_dst_port numeric
21 %% ndpi 0.1 - nDPI
22 @attribute ndpi_proto string
23 @data
24 1,0.011,0.0653,UDP,212.14.174.102,2102,199.165.76.11,123,ntp
25 2,0.439,0.0081,UDP,212.14.174.234,9173,209.51.161.238,123,ntp
26 3,0.675,0.1146,UDP,89.224.252.57,53334,212.14.174.9,10115,ukn
27 4,0.999,0.1158,UDP,212.14.174.202,55735,8.8.8.8,53,Services,DNS,dns
28 5,1.438,0.0091,UDP,212.14.174.234,9173,216.218.254.202,123,ntp
29 6,1.653,0.0093,UDP,212.14.174.105,65476,8.8.8.8,53,Services,DNS,dns
30 7,2.439,0.0077,UDP,212.14.174.234,9173,209.81.9.7,123,ntp
31 8,2.913,0.0211,UDP,154.20.231.103,12420,212.14.174.96,19170,bittorrent
32 9,3.393,0.0090,UDP,61.57.112.210,20392,212.14.174.75,12618,bittorrent
33 ...
```

Listing 4.2: Illustrative Flowcalc output, with the `ndpi` module enabled.

Chapter 5

Literature Survey

We conclude Part I of the thesis with a literature survey of traffic classification and related methods. We reinforce the ideas presented in the previous chapters with real-world examples of classifiers, application detectors, ground-truth techniques, and other interesting works. The survey motivates the thesis by showing the need for integrating many independent methods into one system.

The aim of the survey—published in 2013 in [62]—is to discuss diversity in classification methods. We selected publications that present differentiated methods, were published in 2009-2012, and are relevant to the thesis topic. Comparing with other similar works—namely [105], [28], and [159]—our work focuses on different time span. We review newer works that were not mentioned in these studies, e.g. [14, 15, 41, 56]. Moreover, the chapter gives the reader a quick insight into various methods for extracting traffic features (summarized in Table 5.3), which can be combined together in a Waterfall TC system.

5.1 Related works

In an widely cited and comprehensive survey of traffic classification using ML [105], Nguyen et al. review works published during 2004-2007. The authors claim that ML was used for the first time for classifying traffic in 1994 [67], and that it was the starting point for much of the further work. However, many works fundamental to the state of the art appeared about a decade later, e.g. [16, 82, 101, 123, 154, 158].

A survey by Callado et al. [28] divides traffic analysis into *packet-* and *flow-based*, and references several traffic classification papers published during 2004-2007. Four algorithms are compared in terms of completeness and accuracy: BLINC [82], Bayesian [101], "On The Fly" [16], and Payload Analysis [135]. The authors conclude with recommendations for traffic classification and pose eight research questions.

A paper by M. Zhang et al. [159] and its accompanying website [25] present a list of 68 traffic classification papers published during 1994-2009 together with a catalog of 86 datasets used in these works. The authors propose a structured taxonomy of traffic classification and use it to answer the question on the global share of P2P traffic—basing on the results found in the reviewed papers.

Kim et al. in [86] give an insightful comparison of three general approaches

to traffic classification: *ports-based*, *host-behavior-based*, and *flow-features-based*. The authors evaluate these methods on a strong, few-terabyte dataset collected at diverse geographical locations. Their five key findings were: 1) port number can still constitute a relevant feature; 2) behavior-based classification can be ineffective on backbone links and 3) it may exhibit low byte accuracy; 4) backbone traffic classification needs unidirectional TCP flow features; 5) their classifier based on SVM outperformed other ML algorithms and produced robust results once it was trained with a representative, unbiased training set.

In a recent study, Dainotti et al. [40] anticipate future directions in traffic classification. The authors show the evolution and current state of the field, and draw attention to the taxonomy of *flow objects* and *traffic classes*. Four challenges are discussed: 1) lack of common, representative traffic datasets labelled with ground-truth; 2) inadequacy of current methods to the three trends in network protocols: encapsulation, encryption, and multi-channel communication; 3) poor scalability of algorithms to high-bandwidth links; 4) lack of standard procedures and benchmarks for method evaluation. The authors argue for further research on multi-classifier systems and for development of open-source traffic classification tools.

5.2 Traffic classification

In this category, we collect papers that describe algorithms for identifying any network protocol, or at least a few protocols (e.g. group of P2P-TV protocols). For instance, such algorithms can be deployed on a router to provide statistics on the traffic passing through it.

1) KISS: Stochastic Packet Inspection Classifier for UDP Traffic: The work by A. Finamore et al. [56] published in 2010 (extends the original 2009 paper [55]) presents a payload inspection classifier for UDP traffic. The authors exploit the fact that protocols running over UDP must implement an application-specific header at the beginning of the packet payload, due to stateless nature of UDP communication.

For each 80-packet window in a given flow, the KISS algorithm counts occurrences of distinct 4-bit *groups* in the first 12 bytes of the packet payload; see Fig. 5.1 for an illustration. For each of 24 groups, a χ^2-like test is used in order to measure the distance between distribution of observed values and the uniform distribution, according to Equation 5.1:

$$X_i = \sum_{v=0000_2}^{1111_2} \frac{(O_v^i - E)^2}{E}, \tag{5.1}$$

where: X_i is the distance for group offset i, v is the value, O_v^i is the number of observed occurrences for value v on offset i, and E is the expected value ($E = \frac{80}{2^4} = 5$). The symbols 0000_2 and 1111_2 represent binary numbers: 0 and 15 in decimal system, respectively.

Thus, a characterization of randomness in the application header is obtained, in form of a 24-element feature vector. This vector is used in an SVM decision process, i.e. it is used for training and classification in a typical manner.

The authors evaluated the algorithm on a ca. 100GB dataset of real and testbed network traffic, obtaining respectively 99.6% and <1% of True Positives

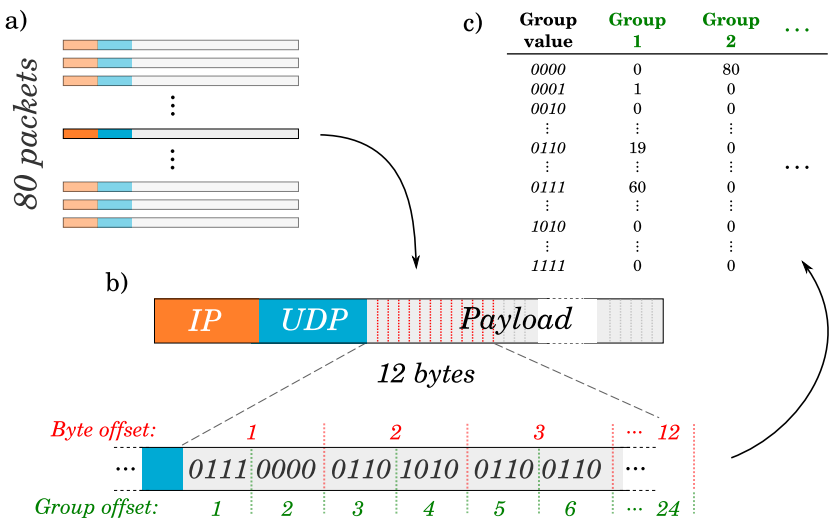

Figure 5.1: Feature extraction in the KISS algorithm: for each packet in an 80-packet window (a), the first 12 bytes of UDP payload are divided into 24 groups of 4 bits each (b). Number of occurrences of distinct values in given group is counted for the whole packet window (c).

and False Positives, on average.

2) K-Dimensional Trees for Continuous Traffic Classification: In an interesting work published in 2010 by V. Carela-Español et al. [30], the authors revisit the idea by L. Bernaille et al. [16] of early traffic classification by analyzing the size and direction of the first few packets of a TCP connection.

However, in this new work the authors apply the K-dimensional trees algorithm [68] instead, which resulted in relatively small times for training and classification. The proposed system operates in real-time and can be continuously retrained. A preliminary evaluation was performed, using a ca. 1TB dataset of 12 types of real network traffic.

3) Abacus: Accurate behavioral classification of P2P-TV traffic: In 2011, P. Bermolen et al. [14] published an exhaustive work on a classifying P2P-TV traffic, preliminarily introduced in [146].

The authors present a method that counts the number of packets received by a given host from each of its peers. Histogram of packet counts received in a 5-second window is used as a feature vector for an SVM classification algorithm.

Bermolen et al. present an excellent experimental analysis of performance, portability, and parameter sensitivity. The authors evaluated the system on a ca. 26GB dataset of testbed P2P-TV traffic (SopCast, TVAnts, PPLive, and Joost) and on a ca. 4GB of real "background" traffic: they report 95% of True Positives and less than 0.1% of False Positives in the worst case—for packets, bytes, and peers.

4) TCP Traffic Classification Using Markov Models: In a work published in 2010 by G. Münz et al. [103], a lightweight method for classification of TCP flows using observable Markov chains [120] is presented. The discretized packet length, direction, and position within the flow are mapped to a state. For

Label	Classifier (see [9, 150])	Overall performance	Selected?
J48	J48 Decision Tree	97.2%	✓
K-NN	K-Nearest Neighbor	95.9%	✓
R-TR	Random Tree	96.3%	✓
RIP	Ripper	97.0%	✓
MLP	Multi Layer Perceptron	82.3%	✓
NBAY	Naive Bayes	43.7%	-
PL	PortLoad [2]	83.7%	✓
PORT	Port number	15.6%	-

Table 5.1: Stand-alone classifiers used in [41]. The "Overall performance" column presents the overall classification accuracy, as reported by the authors; the "Selected?" column indicates which classifiers were used in the final system.

Label	Combiner	Reference in [87]	Best performance
NB	Naive Bayes [44]	pp. 126	93.5%
MV	Majority Voting [13]	pp. 112	90.8%
WMV	Weighted Majority Voting [129]	pp. 123	91.0%
D-S	Dempster-Shafer [121]	pp. 175	97.0%
BKS	Behavior Knowledge Space [75]	pp. 128	97.9%
WER	Wernecke [149]	pp. 129	97.9%

Table 5.2: Algorithms for combining pattern classifiers, as applied in [41]. The "Best performance" column gives classification accuracy for the best selection of stand-alone classifiers working in an ensemble, as reported by the authors (see Table 5.1).

each application of interest, a Markovian model is generated in the training stage. During classification, the a-posteriori probability of observed packets is calculated for each model, and the maximum value is chosen.

The authors performed experimental validation on a small dataset and compared the results to the well-established work by L. Bernaille et al. [16]; however, these two methods are inherently different. The Markov chain method yielded better stability of the results, with similar average precision and recall values. The authors extended their method in [104] by introducing a special "end of connection" Markov state, which improved the accuracy (validated on a larger dataset).

5) Early Classification of Network Traffic through Multi-classification: The work by A. Dainotti et al. [41] published in 2011 presents an innovative approach of multi-classification: the traffic is simultaneously processed by an ensemble of several stand-alone classifiers, and the final result is obtained using a decision combiner algorithm [87].

The authors connect eight stand-alone classifiers (see Table 5.1) using six state-of-the-art combiners (see Table 5.2). Experimental validation on a 59GB dataset of real traffic yielded the best accuracy for the BKS combiner and an ensemble of 6 classifiers: J48, K-NN, R-TR, RIP, MLP, and PL.

The authors highlight that in case we limit feature extraction to just the first few packets in a flow, their method brings significant performance improve-

ments, comparing to the best results of stand-alone classifiers working alone: for example, in case of just the first packet being used, a 20.8% improvement. The authors chose to use the first 4 packets, obtaining the final accuracy of 98.4%; supplementary metrics were not reported.

6) CUTE: Traffic Classification Using TErms: In 2012, S.H. Yeganeh et al. published a paper [155] in which they propose a payload inspection classifier that automatically finds protocol signatures.

For the training, the algorithm extracts common terms shared by flows of a given protocol: it aligns the flows and finds all common substrings of at least b bytes. Next, for each protocol, it assigns *weights* to terms, according to Eq. 5.2:

$$W_t^p = \begin{cases} (\frac{f_p^t}{\sum_{p \in P} f_p^t})^\rho & f_t^p \geq T \\ 0 & f_t^p < T \end{cases}, \tag{5.2}$$

where f_t^p is the frequency of term t in protocol p, P is the set of all protocols, and W_t^p is the term weight; ρ and T are the algorithm parameters. Terms that are unique to protocol have weights close to 1, whereas common terms have weights close to 0.

During classification, for each protocol, the algorithm searches the packet payload for the learned terms, and computes the average weight. The protocol with the maximum value is chosen as the target class.

Yeganeh et al. show by means of theoretical analysis and experimental validation, that in case of pattern matching for traffic classification, occurrences of terms in network flows are more important than their relative order. In practice, this means that it is enough to use term *sets* instead of *lists*: one can identify a certain protocol by checking for occurrence of terms in any order. This makes CUTE inherently simpler and faster than similar algorithms that employ term lists, e.g. LASER [112].

The authors used two traffic traces from Tier-1 ISPs for experimental analysis, i.e. tuning the classification system and validating its accuracy. They report precision and recall metrics above 90% for almost all protocols considered.

5.3 Single application detection

In this section, we describe algorithms that aim at single application or certain traffic kind. For instance, such algorithms can be deployed on a network firewall in order to block access to given service. We maintain the numbering of papers for easy referring in Table 5.3.

7) Tunnel Hunter: Detecting application-layer tunnels with statistical fingerprinting: In a paper published in 2009 [46], M. Dusi et al. present a reliable method for detecting HTTP and SSH tunnels.

The algorithm is trained with legitimate (non-tunneled) HTTP and SSH traffic. Each flow is characterized by a signature consisting of packet size, inter-arrival time, and arrival order. During classification, a flow "anomaly score" is computed by comparing the flow signature to fingerprints of legitimate traffic. If the value is above a certain level, the flow is considered as carrying tunneled traffic. The authors claim nearly 100% completeness and accuracy (verified experimentally).

8) Skype-Hunter: A real-time system for the detection and classification of Skype traffic: The paper by D. Adami et al. published in 2012 [3] introduces a novel method for identification of the Skype protocol.

The authors present a detailed, packet-level analysis of the Skype traffic and propose a relevant detection algorithm that combines signature-based and statistical procedures. The method is experimentally validated on several datasets. Compared with standard statistical classifiers and to a state-of-the-art Skype classifier [18], it yielded better performance results.

5.4 Obtaining ground-truth

In this section, we describe papers on obtaining datasets for verifying the performance of classification methods. In a typical scenario, the author of a new method will work on a trace of network traffic while developing the algorithm. The traffic composing the trace needs to be representative for the scope of interest of a particular research effort. The dataset should also indicate the real application that generated each flow in the dataset, so the researcher is able to compare the results of the algorithm with the right answer: this information is called *ground-truth*. We refer to Chapter 4 for more details.

9) GT: picking up the truth from the ground for Internet traffic: In a 2009 paper published by F. Gringoli et al. [73], the authors present a distributed system for capturing Internet traffic in a computer network. The system keeps the names of applications that generated the traffic.

A special software agent "gt" is installed on each machine taking part in the experiment. The agent periodically queries the operating system for a list of opened network sockets and the names of applications that own them. For each socket, it stores a piece of information with current time-stamp, local and remote IP address and port number, transport protocol, and application name. At the same time, a standard packet sniffer is run on the gateway router, so that all the traffic coming from and into the local network is captured.

Finally, a post-processing tool "ipclass" is run. The tool connects the socket information collected by gt with the traffic captured on the router. As the result, a traffic trace file annotated with ground-truth is produced. The authors validated the method on a 218GB dataset. For the completeness metric, they report more than 99% of bytes and 95% of flows.

10) Quantifying the accuracy of the ground-truth associated with Internet traffic traces: In 2011 M. Dusi et al. [48] published a paper that compares their gt tool [73] to traditional port- and DPI-based methods for establishing ground-truth.

Basing on evaluation on a ca. 200GB dataset, the authors claim that—depending on the protocols composing a trace—ground-truth information can be incorrect for up to 91% bytes for port-based and 26% for DPI-based methods. The authors speculate that the error one might commit while applying these well-established methods to publicly available anonymized traces is significant, especially for modern traffic like Streaming, Skype, or P2P.

11) Tracedump: A Novel Single Application IP Packet Sniffer: The paper published in 2012 [61] (see Chapter 4) introduces a packet sniffer that captures traffic of a single Linux process only. This solves the problem of ground-truth, as the application name is immediately known.

The papers explains implementation of a single-process packet sniffer and provides an architectural view on the proposed solution. The "tracedump" utility captures all application traffic in real-time, including DNS traffic. A short evaluation on BitTorrent traffic is presented. The "tracedump" tool can run a computer program in a fully controlled manner—for instance, GUI testing tools can be applied to create a kind of specialized traffic generator (preliminary results available at [60]).

5.5 Traffic analysis

12) Taking a Peek at Bandwidth Usage on Encrypted Links: In a 2011 paper [47], M. Dusi et al. present a simple regression-tree-based algorithm that monitors the amount of data that protocols transmit over encrypted tunnels, e.g. IPSec.

During the training phase, both the cipher- and plain-text transmissions are visible to the algorithm; the plain-text is used for ground-truth information. As traffic features, the authors employed probability mass function of packet sizes, and statistics related to changes in packet direction. During the operation phase, the algorithm extracts flow features each few seconds, and applies a regression tree algorithm in order to give estimates on the traffic carried within the tunnel.

The authors evaluated their method on a ca. 50GB dataset and reported an acceptable accuracy: the performance depends on the differences in the networks used for training and testing.

13) DNS to the Rescue: Discerning Content and Services in a Tangled Web: In 2012, I. Bermudez et al. published a paper on inferring Internet traffic by analyzing its DNS context [15]. The work introduces "DN-Hunter", a system that tags traffic flows with their associated domain name, based on the fact that each new flow is anticipated by a DNS query.

The system consists of two modules: a flow sniffer, which reconstructs traffic flows, and a DNS resolver, which maintains mapping between clients, domains, and servers. The authors verified that flow tagging can be accomplished in most cases and could not be replaced by making a reverse DNS lookup or inspecting TLS certificates—this would fail in 94% or 86%, respectively. The key property of this novel method is that it can identify traffic before the actual flow starts.

Using capabilities of DN-Hunter, the authors provide a detailed analysis of Content Delivery Networks (CDNs) in 5 datasets of total 64 million flows, covering thousands of ISP customers in US and Europe. Analysis of real traffic revealed domains handled by hundreds of servers that change with time. The authors discovered a diurnal pattern of more machines during late evenings; a similar phenomenon was noticed for CDNs and their domains. For an 18-day observation period, about 100,000 new domains emerged each day.

DN-Hunter can map distribution of particular content across CDNs—the authors found that LinkedIn was hosted by Edgecast (59% of flows), Akamai (17%), CDNetworks (3%), and on own servers (22%). The system can also reveal the domains of a specific CDN: top three domains provided by Amazon EC2 in Europe were cloudfront.net (20%), playfish.com (16%), and sharethis.com (5%). Finally, DN-Hunter can tell the most popular services delivered on a given IP port number—for port 25 the authors observed service tags of "smtp", "mail", "mxN", and several others. Interestingly, they also identified several BitTorrent trackers running on the Google Appspot service.

5.6 Discussion

1. **There are many ways to classify the traffic.** Each work reviewed in Sections 5.2 and 5.3 presents a different approach to classification: analysis of packet count, length, payload, etc.—see Table 5.3 for a summary. We speculate that each modern Internet protocol exhibits so many phenomena that it has plenty of observable traffic characteristics that can reveal its generating application. Moreover, A. Dainotti et al. in [41] proved that it is possible to combine multiple different classifiers into one system that unveils high performance. Thus, we argue that:
   (a) there are many traffic features yet to be found (anticipated e.g. by [14, 15, 56]);
   (b) traffic classification algorithms can be combined so they complement each other (e.g. [56] for UDP and [104] for TCP traffic);
   (c) there is much room for improvement in the design of traffic classifiers that analyze several kinds of traffic features at the same time, i.e. multi-level traffic classifiers (e.g. [41, 82]).
2. **Classification methods need thorough validation.** New services appear rapidly on the Internet, and the application protocols get more sophisticated [40], hence modeling new kinds of traffic gets harder. Consequently, robust traffic classification methods need thorough *experimental* validation, as purely theoretical approach is insufficient. A certain sign of a high-quality paper is a detailed section on validation, employing an up-to-date traffic trace. We give our recommendations for validating TC:
   (a) usage of large, representative, and geographically diverse datasets with relevant amounts of background traffic (e.g. [56, 86]);
   (b) presentation of the results in terms of well-established and *complementary* performance metrics—e.g. recall with precision, or True Positives with False Positives (e.g. [56, 104]);
   (c) analysis of parameter sensitivity of the algorithm (e.g. [14, 155]).
3. **The problem of common traffic datasets is still unsolved.** Several respected scientists demanded publication of common, packet-level traffic datasets labeled with ground-truth: e.g. [124] in 2007 and [40] more recently. This would enable systematic and fair comparison of classification methods, but the problem still remains largely unsolved. Some authors published their datasets, but none of them satisfies all of the postulated requirements (see Chapter 4). However, authors of the studies referenced in section 5.4 made ground-truth data collection simpler and more comprehensible. Particularly, the "gt" [73] software agent seems to be a candidate for the standard ground-truth tool for current and future research on Internet traffic.

5.7 Conclusions

In this Chapter, we reviewed 13 significant papers on traffic classification and related matters, published during 2009-2012. We presented the review in 4 categories: general traffic classification in Section 5.2), single protocol detection in Section 5.3, the ground-truth problem in Section 5.4, and related works in Section 5.5. We showed diversity in methods for analyzing IP traffic and dis-

cussed a few important issues, giving our recommendations. We also presented a succinct "review of reviews" in traffic classification in Section 5.1.

A decade passed since the first major publications on traffic classification appeared [105], but the authors of the reviewed papers proved that it is still possible to find new algorithms [15, 56], or significantly improve the existing ones [30]. In order to classify an IP flow, one can choose to either focus on a specific traffic feature (packet counts [14], lengths [30], payload characteristics [56,155], etc.), use many features at once [3,104], or combine several approaches in a multi-classifier system [41].

Classification methods need to be verified on real IP traffic. The problem of obtaining adequate traces labeled with ground-truth is still largely unsolved. This limits systematic and fair comparison of existing methods: there are no "reference benchmarks" in traffic classification. Besides, the authors of [48] suggest that there may be a significant error in self-made traffic traces anyway. Two utilities—"gt" [73] and "tracedump" [61]—can be applied to assure the accuracy of ground-truth data.

Let us conclude with an observation that we are able to tell things apart if we can see the differences among them. Our paper showed diversity in methods for classifying IP traffic—in our opinion, an interesting direction for TC research.

Paper	Traffic features	Experimental dataset
1) Finamore et al. [56]	For 80-packet windows: amount of randomness in the first 12 bytes of payload	100GB of real and testbed traffic (P2P-TV, Skype)
2) Carela-Español et al. [30]	Size of the first few packets; port numbers	<1TB of real traffic from CoMo-UPC [144]; ground-truth set with DPI
3) Bermolen et al. [14]	Histogram of packet counts received from each peer, in a time window (5s)	26GB of testbed traffic from 30 peers; <4GB of real traffic without P2P-TV
4) Münz et al. [103]	For the first few TCP packets: payload size, packet direction, position in stream	Self-made traces: 300 connections for training, 500 for testing
5) Dainotti et al. [41]	Various	Self-made 59GB trace of real traffic (Oct 2009); ground-truth set with DPI
6) Yeganeh et al. [155]	Existence of precomputed terms in packet payload	Two 30-minute traces from tier-1 ISPs on different continents; no encrypted flows
7) Dusi et al. [46]	Packet size and logarithm of inter-arrival time (quantized values)	Self-made HTTP and SSH traffic (legitimate and tunneled)
8) Adami et al. [3]	Packet size, packet payload (signatures), inter-arrival times	Self-made dataset, Tstat Skype traces [139], and DARPA dataset
12) Dusi et al. [47]	For time-windows: histogram of packet sizes; vector of packet counts and sizes until change in transmission direction occurs	Self-made, real traffic: 36GB captured with "gt" [73] (Oct 2009), 10GB with ground-truth set using DPI (Jul 2010)
13) Bermudez et al. [15]	DNS response received within a time-window preceding the IP flow	5 diverse sets of real traffic from EU and US; 64 million TCP flows, almost 2 days of traffic

Table 5.3: Summary of the reviewed papers: traffic features and datasets used for experimental validation

Part II

Cascade Classifiers of Internet Traffic

Chapter 6

The DNS-Class algorithm

In this chapter, we present a novel TC method, DNS-Class, which was originally published in [63]. The method targets roughly 30% of evaluated real-world Internet traffic, and is an illustrative example of a building block for a Waterfall system. Thus, below we motivate the need for cascading traffic classifiers, as otherwise using DNS-Class alone would be impractical.

6.1 Introduction

No previous work presented complete description and evaluation of a TC method that uses Domain Name System (DNS). We believe that the information carried in DNS packets can provide a basis for new traffic classifiers and contribute to MCS. The authors of [117] and [15] already described some important ideas on this topic, but we present the first work that follows the state of the art in traffic classification. For instance, comparing with these works, we adopt the traditional notion of traffic classes that correspond to network protocols, and we evaluate the performance of our system using well-known metrics. Moreover, we motivate traffic classification using DNS by showing its application to multi-classifier systems—i.e. to modular, cascade classification systems.

In this Chapter, we present DNS-Class: a novel ML method for classifying IP flows by inspecting DNS traffic transmitted in a computer network. First, our algorithm passively collects the information in DNS packets to find the domain names behind connection peers (e.g. `www.facebook.com`). Second, by running text classification on these names, DNS-Class reveals the generating application (e.g. HTTP). Combining this information with port numbers further improves the classification performance, even though port numbers alone are unreliable.

The novelty of our method is explained in the fact we believe this is the first traffic classifier employing DNS domain names as traffic features. One of the main strengths of DNS-Class is its ability to correctly classify an IP flow by only inspecting its first packet. We argue that this represents a significant step ahead with respect to the state of the art methods, which require several packets before being able to perform any classification.

The contributions of DNS-Class are the following:

1. We present the first complete description and evaluation of a TC algorithm that employs DNS, while conforming to the state of the art (Section 6.6).

2. We use DNS traffic analysis and an ML technique to propose our novel traffic classification method, DNS-Class (Sections 6.2.1 and 6.2.2).
3. We indicate the interesting characteristics and potential applications of traffic classification using DNS, with an example of cascade classification (Section 6.2.3).
4. We give a short analysis on how different network protocols depend on DNS, in terms of flows, packets, and bytes (Section 6.3.2).
5. We evaluate DNS-Class on real traffic traces, using well-known performance metrics and traffic classes that correspond to popular network protocols (Section 6.4).
6. We show that the domain name alone is not enough to reliably classify IP flows, and needs augmentation with the port number and transport protocol name (Section 6.5).
7. We release the complete source code implementing the presented system as open source [59].

6.2 The DNS-Class algorithm

The DNS-Class algorithm operates on traffic flows and consists of two stages: DNS Search and Flow Classification. In essence, the first stage assigns domain names to flows, while the second runs text classification on domain names extended with port numbers and transport protocol names. Below we give the taxonomy that will be used throughout this Chapter.

We define as *flow* the set of packets belonging to the same connection neglecting their direction, i.e. packets having the same 5-tuple of $\langle IP_{src}, Port_{src}, IP_{dst}, Port_{dst}, Proto \rangle$, with the source *src* and destination *dst* optionally swapped. The IP, $Port$, and $Proto$ terms stand for IPv4 address, transport protocol port number, and transport protocol name, respectively.

We also introduce an important notion of *named flows*: the flows for which the peer sending the first packet queried the DNS system. In such a case, the queried domain name is the *flow name*. On the contrary, if querying was not needed, the flow is an *anonymous flow*. Note that querying DNS does not necessarily involve communication with a DNS server, due to local caching of DNS information.

In the context of TC, we employ the term *application* (with synonyms of *protocol* and *class*) for referring to computer programs that generate Internet traffic, or network protocols that transport it. The information on the real application that generated a particular flow is called *the ground-truth*.

6.2.1 DNS Search

The DNS Search stage employs the idea of assigning domain names to IP flows, which was introduced in [117] and [15], as detailed in Section 6.6.

The architectural view of DNS Search is depicted in Figure 6.1. Its first element, DNS PACKET INSPECTOR, examines packets in DNS flows and updates the RESOLVER database with information on domain names for client-server pairs. The FLOW TAGGER queries this database each time a new flow starts and assigns DNS names, if possible. As the result of DNS Search, a set of flows is obtained, in which some elements have a domain name assigned.

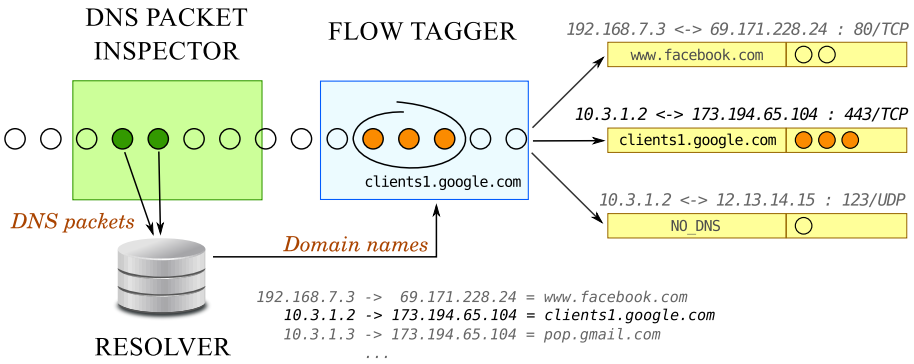

Figure 6.1: The DNS Search algorithm. Incoming packets are inspected for DNS information, grouped into flows, and flows are tagged with names. The algorithm discovered that the client `10.3.1.2` received earlier a DNS reply for the server `173.194.65.104` with the name of `clients1.google.com`, so it tagged the matching TCP connection with this domain name. The RESOLVER database holds names for client-server pairs; the same server address can appear under different names, depending on the client.

The DNS Search algorithm exploits the fact that Internet connections are often anticipated by DNS query and response packets (see Section 6.3). For example, a web browser will resolve `clients1.google.com` to `173.194.65.104` before attempting a TCP connection to the corresponding web server. By intercepting the DNS response packet, an intermediary router can uncover the original domain name entered by the user into the web browser location bar.

In greater detail, DNS PACKET INSPECTOR dissects packets according to RFC1035 [99]. At its input, it takes successful replies to queries of `A` and `MX` types. The destination address in the IP header becomes *client address* ($Client_{addr}$: the host that asked for the domain name), and each address found in the DNS Answer becomes *server address* ($Server_{addr}$: the host that the client can connect to). The first element transmitted in the DNS Questions list is the domain name; it is stored in the RESOLVER database under an index of $\langle Client_{addr}, Server_{addr} \rangle$, for each $Server_{addr}$. This information will be cached for the next 10 hours of traffic—unless updated earlier by another DNS packet with the same index—which roughly imitates the caching mechanism usually found in the DNS resolvers running on the client machines. We chose the value of 10 hours arbitrarily; smaller values should work as well (see Section 6 in [15]).

The FLOW TAGGER is run for every flow on the arrival of the first packet. It queries the RESOLVER database for the domain name under $\langle IP_{src}, IP_{dst} \rangle$ index and assigns it to the flow (for the entire flow lifetime). If this fails, it tries the opposite direction, i.e. $\langle IP_{dst}, IP_{src} \rangle$. If this fails too, no name is attached to the flow and it is left anonymous, depicted as `NO_DNS` in Figure 6.1. Note that implementing a similar mechanism using reverse DNS lookup would fail: in [15], the authors proved experimentally that issuing a reverse lookup on the server IP of a randomly chosen named flow either returns a different domain name (62% of flows), or yields no answer at all (29% of flows).

The DNS Search algorithm is designed to be implemented on a single machine, e.g. on a gateway router. However, the DNS PACKET INSPECTOR and

HTTP	NTP	Jabber	BitTorrent
nk.pl:80/TCP	pool.ntp.org:123/UDP	talk.google.com:5222/TCP	exodus.desync.com:6969/TCP
photos.nasza-klasa.pl:80/TCP	clock.fmt.he.net:123/UDP	talkx.l.google.com:5222/TCP	tracker.openbittorrent.com:80/UDP
www.playmobile.pl:80/TCP	ntp1.dlink.com:123/UDP	chat.facebook.com:5222/TCP	router.utorrent.com:6881/UDP
gg.hit.gemius.pl:80/TCP	time.nist.gov:123/UDP	xmpp.nktalk.pl:5222/TCP	tracker.publicbt.com:80/UDP
www.facebook.com:80/TCP	time-a.netgear.com:123/UDP	talk.l.google.com:5222/TCP	fr33dom.h33t.com:3310/TCP

Table 6.1: Examples of input to Flow Classification stage: the top-5 flow names, with port numbers and transport protocol names. The BitTorrent protocol uses dynamic port numbers, but its domain names can reveal the generating application. Polish domains are specific to the dataset.

Flow tagger can be decoupled and implemented on two separate machines—e.g. intercepting DNS information on a local recursive DNS server, and tagging flows on a router—with small modifications to the algorithm.

6.2.2 Flow Classification

As the input to the classification stage of DNS-Class, we take the set of named flows. Table 6.1 presents examples of input data, whereas Figure 6.2 shows classification of a particular TCP connection between `10.3.1.2` and `173.194.65.104` on port 443, having a domain name of `clients1.google.com` assigned.

First, flow information is transformed into a textual form by concatenating the flow name, port number, and transport protocol name (using separating characters). DNS-Class can also operate without flow features—i.e. can classify solely by the domain name—which is demonstrated in Section 6.4.2 pt 2. For the port number, we use the destination port of the first packet in a flow.

The text form is split by dots and dashes into a tuple of tokens T (T is limited to the last 6 tokens by default). Digits are replaced with the capital "N" character, except for the port number. The goal of this step is extracting keywords from domain names. We also tried word segmentation in long domain names as described in [106], but without a meaningful effect on the overall classification performance.

The tuple of tokens T is converted into a set of text features F, by extracting all word unigrams and bigrams. The algorithm has a Converter database, which is used as an invertible function C that maps these unigrams and bigrams to integers, as shown in Equations 6.1 and 6.2:

$$C(v_w) = v_k, \tag{6.1}$$

$$C^{-1}(v_k) = v_w, \tag{6.2}$$

where $v_w \in W$, the set of all known unigrams and bigrams, and $v_k \in K$, $K = \{1, 2, \ldots, n\}$. Note that $|W| = |K| = n$. The set W is constructed during the training of DNS-Class and held constant during classification, and so is K. In case a given unigram or bigram cannot be found in the database, its corresponding integer value is 0, i.e.

$$C(v_x) = 0 \quad v_x \notin W, \tag{6.3}$$

which means that v_x is dropped and not considered in next algorithm steps.

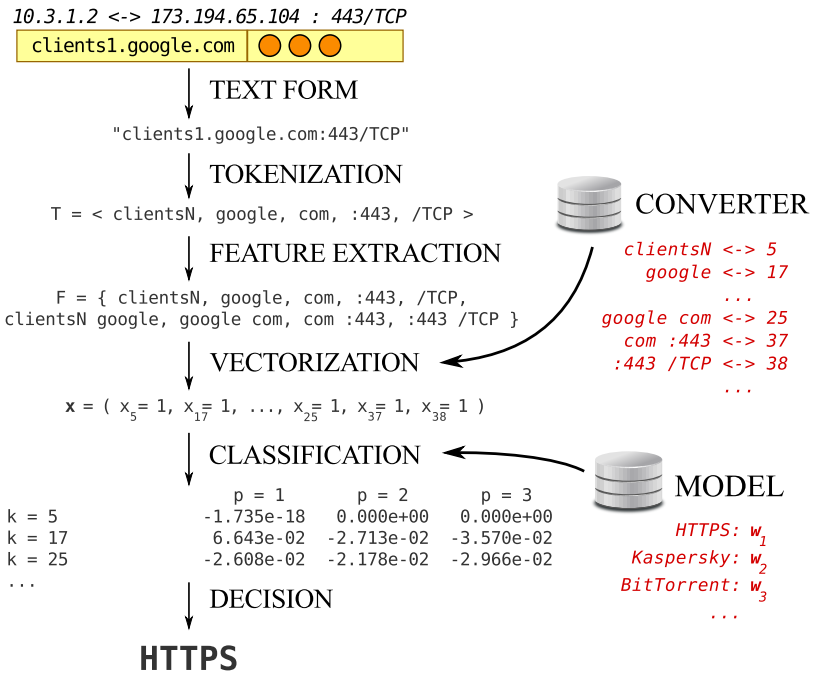

Figure 6.2: The DNS-Class algorithm. The input flow data is rewritten, converted into Vector Space Model $\mathbf{x}$, and classified using support vector classification. The CONVERTER holds 1-1 associations between textual features v_w and their integer counterparts v_k. The MODEL is constructed during system training and keeps weight vectors for each protocol.

In the VECTORIZATION step, the set of text features F is converted into a sparse feature vector $\mathbf{x}$ of size n, in which each dimension corresponds to a text feature. In other words, a Vector Space Model (VSM) representation of the text input is constructed [125]: $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, where

$$x_j = \begin{cases} 1 & j \in F' \\ 0 & j \notin F' \end{cases} \qquad \forall j \in K, \tag{6.4}$$

and F' is the set of text features F converted to integers using function C. We also tried other functions for x_j—including the popular tf-idf metric [125]—but we experienced no major impact on the overall system performance.

For the CLASSIFICATION step, we employ direct multi-class support vector classification by Crammer and Singer [38] with a linear kernel, as in LIBLINEAR [51,83] and LIBSHORTTEXT [157]. In this step, the MODEL database is queried for weight vectors $\mathbf{w}_p$, for each target class p:

$$\mathbf{w}_p = (w_p^{(1)}, w_p^{(2)}, \ldots, w_p^{(n)}), \tag{6.5}$$

where each $w_p^{(j)}$ term corresponds to the weight of text feature $j \in K$, and $p \in P$, $P = \{1, 2, \ldots, m\}$: the set of all network protocols to be recognized by the classification system. Let

$$\alpha(\mathbf{x}, p) = \mathbf{w}_p^T \mathbf{x}, \tag{6.6}$$

be the *decision value* for protocol p given feature vector $\mathbf{x}$, then the *decision function* is

$$\arg\max_{p} \ \alpha(\mathbf{x}, p), \tag{6.7}$$

which predicts the protocol behind $\mathbf{x}$. In other words, in the CLASSIFICATION step, we search for the protocol p that maximizes Equation 6.6 for a given feature vector $\mathbf{x}$. Note that $\mathbf{x}$ is a sparse vector with few non-zero elements, thus obtaining the product described by Equation 6.6 can be optimized for fast computation.

During training of DNS-Class, weight vectors $\mathbf{w}_p$ are initialized using training instances of text, according to Equations 6.8 and 6.9:

$$\begin{array}{ll} \min\limits_{\{\mathbf{w}_p\},\{\xi_i\}} & \frac{1}{2}\sum_p \|\mathbf{w}_p\|^2 + C\sum_i \xi_i \\ \text{s.t.} & \alpha(\mathbf{x}_i, y_i) - \alpha(\mathbf{x}_i, p) \geq \gamma_i(p) - \xi_i \quad \forall p, i \end{array} \tag{6.8}$$

$$\gamma_i(p) = \begin{cases} 0 & y_i = p \\ 1 & y_i \neq p \end{cases}, \tag{6.9}$$

where i is the number of text instance, $C \in \mathbb{R}_{>0}$ is the regularization parameter, $\xi_i \in \mathbb{R}_{\geq 0}$ are slack variables, and $y_i \in P$ is the true protocol behind feature vector $\mathbf{x}_i$—that is, the ground-truth label. Roughly speaking, the goal of the optimization described by Equation 6.8 is to have high $w_p^{(j)}$ values for features that are specific to protocol p, and low values for features that are common for all protocols. See [51] and [157] for a more detailed description.

In the last step (DECISION), the protocol p selected according to Equation 6.7 is translated into corresponding textual representation. For example, in Figure 6.2, $p = 1$ stands for HTTPS.

6.2.3 Rationale

DNS-Class is a specialized traffic classifier that targets named flows and DNS traffic passing through Internet gateways. This usually corresponds to a significant portion of the whole traffic, as specified in [15], where Bermudez et al. claim that for HTTP and TLS flows the portion of named flows usually exceeds 90% in most of their highly representative datasets. Given that HTTP is nowadays considered to be the dominant protocol in residential customer traffic [93], the actual portion of named flows transmitted through Internet gateways can be much higher than what our study shows in Section 6.3.2 for a particular Internet Service Provider (ISP) network.

In Figure 6.3, we present one of the possible scenarios for DNS-Class: *cascade classification*, which is described in more detail in the next Chapters of this thesis. Traffic traveling through a gateway is classified in a modular system. Each module is responsible for handling only one part of the traffic, according to several *selection criteria*. If an input flow cannot be classified, it is handed over to the next module in the "cascade" of classifiers. In such a scenario, the goal of DNS-Class is classifying only the named flows and DNS flows, leaving the anonymous flows for other modules. For example, DNS-Class can be augmented with statistical classifiers, fine-grained methods [111], or even with other specialized classifiers like Skype-Hunter [3]. Note that a reliable selection criterion for our algorithm is simply the presence of a domain name attached to the flow.

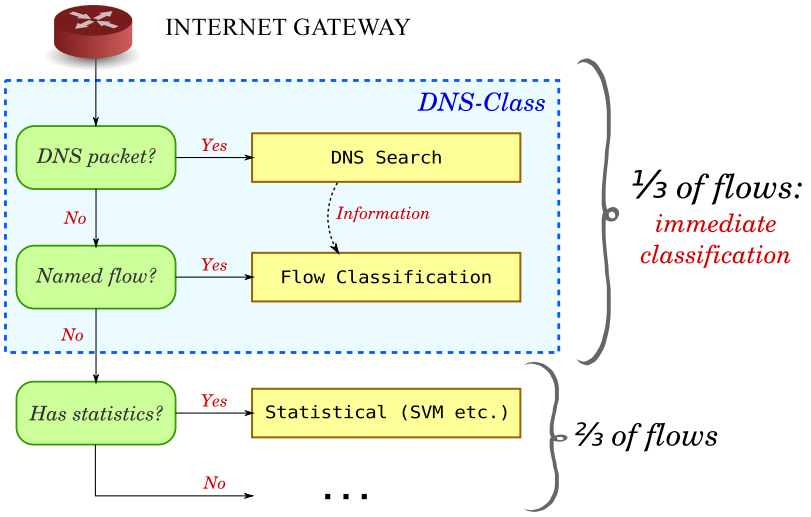

Figure 6.3: Modular traffic classifier. We propose an algorithm that targets one-third of network traffic in the investigated ISP network. DNS-Class immediately classifies named flows, leaving anonymous flows for other classifiers.

Comparing with existing traffic classification methods, DNS-Class has interesting properties. First, it immediately classifies network flows, requiring just the IP header of the first packet and the information extracted from DNS query-response conversations. Second, DNS-Class does not inspect the payload of the packets (except for DNS traffic), which makes it resistant to TLS encryption. We thus believe it represents an important development in TC, and can be applied to improve the performance of existing systems.

6.3 Datasets and traffic analysis

In this section, we analyze the traffic datasets that we used for validating DNS-Class, and which will be used in the next section for presenting the practice of applying DNS-Class to real network traffic. We also share our findings on how Internet protocols depend on DNS.

6.3.1 Traffic traces

We collected the traffic during May-June 2012 and January 2013 at a Polish ISP company that serves residential customers. In both cases, packet capture was run for around one week on the same Point-to-Point over Ethernet (PPPoE) server that handled a few hundred users. The maximum amount of captured packet data was limited due to storage constraints; the Ethernet and PPPoE headers were removed too. Table 6.2 summarizes the datasets, and Figure 6.4 presents one illustrative day of traffic in the ASNET1 dataset.

We established the ground-truth using lightweight packet inspection, as implemented in the LIBPROTOIDENT library[1], published by the University of

[1]Subversion revision number 154

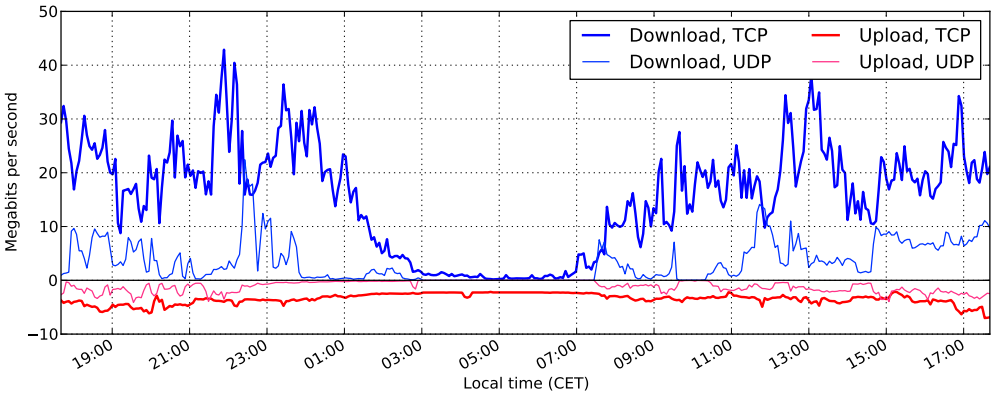

Figure 6.4: One day of traffic in ASNET1. Total network bandwidth usage of client downloads and uploads, for TCP and UDP. The data was collected June 1-2, 2012 and represents 4-minute averages.

Dataset	Start	Duration	Src. IP	Dst. IP	Packets	Bytes	Avg. Util	Avg. Flows (/5 min.)	Payload
Asnet1	2012-05-26 17:40	216h	1,828 K	1,530 K	2,525 M	1,633 G	18.0 Mbps	7.7 K	92 B
Asnet2	2013-01-24 16:26	168h	2,503 K	2,846 K	2,766 M	1,812 G	25.7 Mbps	12.0 K	84 B

Table 6.2: Datasets used for experimental validation. The "Payload" column gives the amount of data past the IP header. Both datasets contain real traffic of the same population of domestic users.

Waikato [6]. We embedded this library in FLOWCALC (see Chapter 4) that converts PCAP files into flow-level summaries in the ARFF file format. We made a few minor corrections to the results of LIBPROTOIDENT by analyzing the traffic traces manually, according to our knowledge. We noticed several flows on ports 6969, 2710, and 3310 being erroneously classified as HTTP_NONSTANDARD instead of BITTORRENT; another problem was an over-matching rule for the TEREDO protocol.

We adopted definitions of traffic classes from LIBPROTOIDENT, ignoring the transport protocol name, e.g. KASPERSKY instead of KASPERSKY_TCP and KASPERSKY_UDP. For brevity, we used the MAIL class as an aggregate for POP3, SMTP, and IMAP. For our experiments with the CAIDA port-based classifier in Section 6.4.2 pt 1, we translated the names of traffic classes.

Note that our definitions of traffic classes correspond to network protocols—which is motivated in Section 6.6—but domain names allow for a more detailed visibility of e.g. web traffic. However, we leave this as out of scope of this chapter, and refer the reader to [52] and [15] for works strictly devoted to HTTP/HTTPS traffic. On the other hand, established traffic classification methods already offer similar level of granularity in traffic classes as DNS-Class.

We sanitized the datasets by removing incomplete TCP sessions, and by dropping the traffic that is specific for Local Area Network (LAN) environments—e.g. DHCP, NETBIOS, and SSDP. As our last step, we ran our DNS Search algorithm described in Section 6.2.1 to discover the domain names of the network flows in our datasets, obtaining the results presented in Table 6.3.

6.3.2 Traffic characteristics

In order to show how different applications depend on DNS, we divide the set of all network protocols into three groups: 1) traditional client-server protocols (e.g. browsing, e-mail, streaming), 2) P2P and Gaming traffic, and 3) other. The last group consists of DNS traffic and the flows for which our ground-truth method failed. Table 6.3 (pp. 57) presents results of traffic analysis, and is the basis for this section. For the sake of brevity, we report only on the ASNET1 dataset, leaving the ASNET2 dataset for temporal stability evaluation in Section 6.4.2 pt 5. For examples of flow names and port numbers, see Table 6.1.

For validating DNS-Class, we need flows with both the ground-truth label and the domain name (i.e. 26% of flows in ASNET1). However, because DNS-Class identifies DNS packets directly during DNS Search, in total our algorithm targets 38.7% of all flows in ASNET1. As can be seen in Table 6.3a, network protocols differ in how much they depend on DNS. In next sections we will only consider the protocols for which at least 10% of flows have a domain name (in order to have enough training data), except for BITTORRENT and SKYPE, which were included for their popularity in the traffic classification literature.

Below we present our findings for the ASNET1 dataset. Note that other authors already reported similar results using different datasets, e.g. in Section 3.1.2 of [15] and Sections I and V-A of [117]. Comparing with our work, we more deeply analyze the dependence on DNS for many network protocols, and give the results in terms of flows, packets, and bytes. See Table 6.3 for details.

For the ASNET1 dataset, we found that:

1. *27% of flows have a domain name.* We believe this is the lower bound, as the ASNET1 dataset has a limited amount of the packet payload. Note that for networks with higher impact of HTTP traffic than in ASNET1, the portion of named flows would also be higher.
   (a) Flows of traditional client-server protocols vary in their dependence on DNS, but generally this class of flows often incurs DNS queries: 78% of traditional flows have a domain name.
   (b) P2P applications and computer games almost never employ DNS for communication: on average, only 0.2% of their flows have a domain name. These protocols do not need DNS for communication between peers. For example, BitTorrent trackers point to seeders and leechers by their IP addresses; similarly, game servers also list the players by IP addresses, and the exchange of game information occurs directly between the peers.
2. *50% of bytes and 44% of packets travel in named flows.* The average size of named flows is two times higher than the size of anonymous flows (200 KB vs. 110 KB).
   (a) For traditional protocols, 74% of bytes and 73% of packets are transmitted in named flows.
   (b) For P2P and Gaming traffic, this is 0.0018% and 0.02% for bytes and packets, respectively.
3. *If a flow has a domain name, it is almost certainly not P2P nor Gaming.* Only 0.4% of named flows are P2P or games.
   (a) This phenomenon can be practically applied as a quick method for ensuring that a flow does not belong to a P2P application or a computer game.

a)

	Protocol	All traffic			Traffic with domain name			Named flows	Selected?
		Flows	Packets (K)	Bytes (M)	Flows	Packets (K)	Bytes (M)		
	HTTP	4,415,380	1,325,934	1,078,082	3,603,032	985,796	800,752	81.60%	yes
	HTTPS	540,650	58,940	29,219	340,897	41,941	20,571	63.05%	yes
	Kaspersky	43,340	485	16	16,391	193	7.3	37.82%	yes
	NTP	34,160	102	4.7	10,241	47	2.1	29.98%	yes
	STUN	20,786	540	276	4,646	35	1.8	22.35%	yes
	Mail	20,293	8,590	5,558	14,391	7,950	5,455	70.92%	yes
	SIP	18,498	374	127	78	69	22	0.42%	-
	Teredo	12,504	923	59	0	0	0	0.00%	-
1)	SSH	9,424	1,348	837	973	1,017	799	10.32%	yes
	Jabber	4,752	558	62	4,530	539	61	95.33%	yes
	SQL	4,752	3,196	1,086	9	62	33	0.19%	-
	Teamviewer	1,185	1,683	231	72	13	4.3	6.08%	-
	FlashPlayer	1,039	142	48	329	10	2.0	31.67%	yes
	RTMP	821	18,073	13,330	373	3,283	2,193	45.43%	yes
	FTP	209	294	243	182	254	211	87.08%	yes
	Shoutcast	160	4,335	2,481	142	4,085	2,304	88.75%	yes
	BitTorrent	5,034,169	434,230	294,422	14,720	111	5.3	0.29%	yes
	Kademlia	1,222,167	4,273	284	0	0	0	0.00%	-
	eMule	290,101	98,558	71,112	0	0	0	0.00%	-
	Steam	179,910	2,872	955	88	10	0.9	0.05%	-
2)	Skype	172,171	66,861	26,221	236	1.5	0.10	0.14%	yes
	Ares	10,990	2,407	271	0	0	0	0.00%	-
	XboxLive	6,564	2,148	288	7	2.7	0.08	0.11%	-
	HalfLife	3,832	57	2.7	2	0.01	0.001	0.05%	-
	Roblox	364	25,595	5,297	0	0	0	0.00%	-
	DNS	1,817,572	9,620	581	0	0	0	0.00%	yes
3)	Unknown	1,717,714	374,586	136,876	209,127	28,867	8,651	12.17%	-
	Total:	15,583,507	2,446,723	1,667,969	4,220,466	1,074,286	841,076		

b)

Protocol group	Traffic with domain name		
	Flows	Packets	Bytes
Traditional (1)	77.93%	73.33%	73.56%
P2P & Games (2)	0.22%	0.02%	<0.01%
Other (3)	5.92%	7.51%	6.29%
All traffic:	27.08%	43.91%	50.42%

c)

Result of DNS search	Distribution of flows		
	Traditional (1)	P2P & G. (2)	Other (3)
Success	94.69%	0.36%	4.96%
Failure	9.96%	60.77%	29.27%
All traffic:	32.91%	43.18%	22.69%

Table 6.3: Results of traffic analysis in the ASNET1 dataset. Table 6.3a lists all significant protocols in each group—traditional client-server (1), P2P & Games (2), and Other (3)—and gives numerical values on the traffic and its dependency on DNS. The last column indicates which protocols were chosen for experimental validation. Table 6.3b analyses how many flows, packets, and bytes in each group have a domain name. Table 6.3c shows the distribution of flows among the groups, depending on the result of the DNS Search algorithm.

(b) Conversely, if a flow does *not* have a domain name—and at the same time it is not a DNS flow—then it is either P2P or Gaming with a probability of >77%.

6.4 Experimental evaluation

In this section, we present the practice of using DNS-Class in a real network, in different setups. We also compare DNS-Class to an established traffic classification method. The experiments were designed to evaluate the robustness of DNS-Class and to assert that combining domain names with port numbers is meaningful. The results are summarized in Table 6.4 (pp. 59).

6.4.1 Methodology

For each experiment, we start with tuning the algorithm parameters. Then, we evaluate the classification performance and robustness: in experiments 2-4 we employ 10-fold cross-validation on ASNET1 [45], and in experiment 5 we train on whole ASNET1 and test on whole ASNET2.

For given protocol p, we measure the classification performance using two complementary metrics of True Positives ($\%TP_p$) and False Positives ($\%FP_p$):

$$\%TP_p = \frac{|TP_p|}{|F_p|} \cdot 100\%, \quad \%FP_p = \frac{|FP_p|}{|F'_p|} \cdot 100\%, \tag{6.10}$$

where TP_p is the set of true positives for protocol p, F_p is the set of testing flows that belong to p, FP_p is the set of false positives for p, and F'_p is the set of all testing flows that do not belong to p. For measuring the overall DNS-Class performance, we simply adopt the average for all protocols:

$$\%TP = \frac{\sum_p \%TP_p}{|P|}, \quad \%FP = \frac{\sum_p \%FP_p}{|P|}. \tag{6.11}$$

where P is the set of all traffic classes p. For more background, see Chapter 3 of the thesis, where we presented a more generic definition of $\%TP$ and $\%FP$.

Note that we could compute the weighted averages according to the number of flows in each class. However, the result would be heavily biased towards the HTTP class, which holds the vast majority of flows. In most of our experiments, the results for HTTP were close to perfect values, whereas the results for other classes were worse. Thus, by adopting an average instead of weighted average we make our evaluation scheme more demanding. That said, for completeness we also present the averages weighted by the number of flows in each class: the $\%TP_w$ and $\%FP_w$ metrics.

6.4.2 Experiments

1) Traditional port number classifier (Figures 6.5 and 6.6): We began our experiments by evaluating a port number classifier on our datasets. We chose the CAIDA Coral Reef suite version 3.9.1 as our reference implementation [84]. Note that port-based classification has limitations, as discussed in several papers [81]. However, it can classify both encrypted and unencrypted flows using

a)

	FPlayer	FTP	HTTP	HTTPS	Jabber	Kasp.	Mail	NTP	RTMP	SSH	STUN	Shcast	Skype	Torrent	%TP	%FP
FPlayer	49.0%		35.5%	15.4%											49.0%	<0.1%
FTP		46.2%	53.8%												**46.2%**	<0.1%
HTTP	<0.1%	<0.1%	99.0%	1.0%			<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	99.0%	**31.0%**
HTTPS	<0.1%		35.4%	63.3%	0.4%	0.3%	0.6%		<0.1%	<0.1%			<0.1%		63.3%	1.0%
Jabber			0.1%	30.7%	69.2%										69.2%	<0.1%
Kasp.						100.0%									100.0%	<0.1%
Mail			5.8%	0.1%			94.1%								94.1%	0.1%
NTP			0.1%					99.9%							99.9%	<0.1%
RTMP			27.6%						72.4%						72.4%	<0.1%
SSH			1.4%							98.6%					98.6%	<0.1%
STUN			0.1%	1.9%							98.0%				98.0%	<0.1%
Shcast			7.3%									92.7%			92.7%	<0.1%
Skype				1.0%									99.0%		99.0%	<0.1%
Torrent			31.4%	0.1%										68.6%	68.6%	<0.1%
														Average:	82.1%	2.3%

b)

	FPlayer	FTP	HTTP	HTTPS	Jabber	Kasp.	Mail	NTP	RTMP	SSH	STUN	Shcast	Skype	Torrent	%TP	%FP
FPlayer	96.3%	<0.1%		3.7%											96.3%	<0.1%
FTP		99.9%	<0.1%												99.9%	<0.1%
HTTP			100.0%												100.0%	<0.1%
HTTPS				100.0%											100.0%	**0.3%**
Jabber					100.0%										100.0%	0.0%
Kasp.				74.9%		25.1%									**25.1%**	0.0%
Mail		<0.1%					99.9%								99.9%	0.0%
NTP								100.0%							100.0%	0.0%
RTMP	<0.1%		0.4%	1.4%					98.1%						98.1%	0.0%
SSH										100.0%					100.0%	0.0%
STUN		0.5%									99.5%				99.5%	<0.1%
Shcast		0.9%	6.8%									92.2%			92.2%	0.0%
Skype											1.7%		98.3%		98.3%	0.0%
Torrent		<0.1%	0.6%	<0.1%							0.1%			99.3%	99.3%	0.0%
														Average:	93.5%	<0.1%

c)

	FPlayer	FTP	HTTP	HTTPS	Jabber	Kasp.	Mail	NTP	RTMP	SSH	STUN	Shcast	Skype	Torrent	%TP	%FP
FPlayer	99.9%			<0.1%								0.1%			99.9%	<0.1%
FTP		99.9%	0.1%												99.9%	0.0%
HTTP			99.9%									<0.1%			99.9%	<0.1%
HTTPS	<0.1%			99.8%		0.2%			<0.1%						99.8%	<0.1%
Jabber					100.0%										100.0%	0.0%
Kasp.						100.0%									100.0%	<0.1%
Mail							100.0%								100.0%	0.0%
NTP								100.0%							100.0%	0.0%
RTMP			0.2%	0.3%					99.4%						99.4%	<0.1%
SSH										100.0%					100.0%	0.0%
STUN											100.0%				100.0%	<0.1%
Shcast	<0.1%		0.9%									99.1%			99.1%	<0.1%
Skype													99.9%	<0.1%	99.9%	0.0%
Torrent			0.9%	<0.1%							<0.1%			99.1%	**99.1%**	**<0.1%**
														Average:	99.8%	<0.1%

d)

	FPlayer	FTP	HTTP	HTTPS	Jabber	Kasp.	Mail	NTP	RTMP	SSH	STUN	Shcast	Skype	Torrent	%TP	%FP
FPlayer	90.8%		0.3%	5.3%	3.6%										**90.8%**	<0.1%
FTP		98.7%										1.3%			98.7%	<0.1%
HTTP	<0.1%	<0.1%	99.8%				<0.1%		0.2%			<0.1%			99.8%	<0.1%
HTTPS				99.9%		0.1%									99.9%	<0.1%
Jabber					100.0%										100.0%	<0.1%
Kasp.						100.0%									100.0%	<0.1%
Mail							100.0%								100.0%	<0.1%
NTP								100.0%							100.0%	0.0%
RTMP	0.4%		0.8%	0.4%					98.4%						98.4%	**0.1%**
SSH										100.0%					100.0%	0.0%
STUN											100.0%				100.0%	<0.1%
Shcast	1.6%	1.0%										97.5%			97.5%	<0.1%
Skype													100.0%		100.0%	0.0%
Torrent			0.2%	<0.1%							<0.1%			99.8%	99.8%	0.0%
														Average:	98.9%	<0.1%

Table 6.4: Results of experiments: 10-fold cross-validation The tables present confusion matrices and performance metrics for the experiments 2-5 described in Section 6.4.2: a) domain name classification, b) port number classification, c) full DNS-Class, and d) performance after 8 months since training.

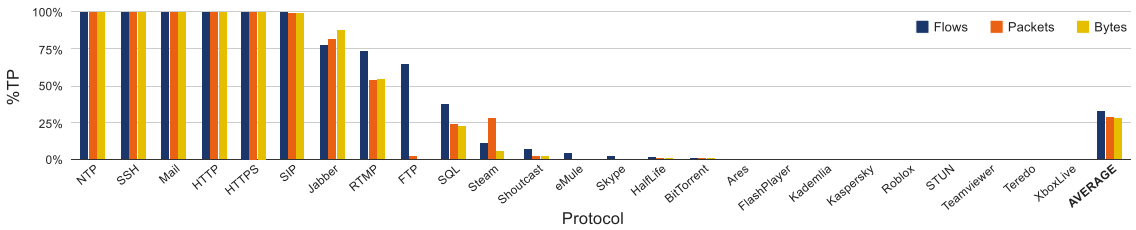

Figure 6.5: Performance of the standard Coral Reef port number classifier, for all flows in groups (1) and (2) (see Table 6.3). The %TP metric is 33%, 29%, and 28%—for flows, packets, and bytes, respectively.

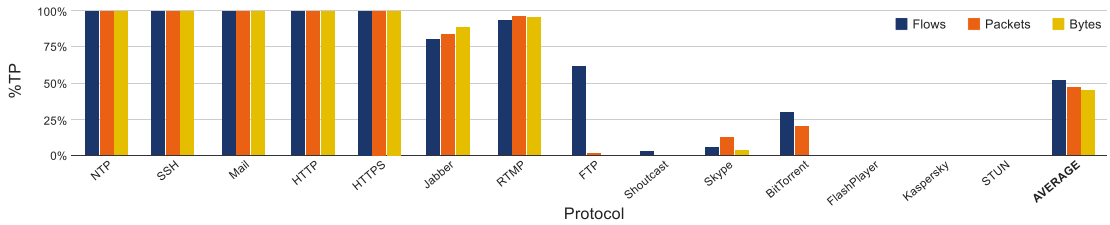

Figure 6.6: Performance of the same classifier as in Figure 6.5, but evaluated only on the named flows of selected protocols. %TP is 52%, 47%, and 45%—for flows, packets, and bytes, respectively.

just 1 packet. Thus, although not perfect, port-based classification has similar characteristics to DNS-Class, which enables a qualitative comparison.

First, we evaluated the port-based classifier on all protocols in groups (1) and (2) defined in Section 6.3. Because the classifier does not need training, we used all flows in ASNET1 for testing. As expected, the method presented good results for some classes, but in most cases it failed: the %TP metric was 33%, as visible in Figure 6.5 (the figure also presents %TP for packets and bytes, computed similarly). For comparison with DNS-Class, we evaluated the port-based classifier on our target traffic, i.e. on named flows. The performance improved, but still was quite low: %TP of 52%, as visible in Figure 6.6.

2) DNS-Class: domain names (Table 6.4a): In experiment 2, we run our algorithm on sole DNS features, i.e. we classified the traffic using only the domain name (neglecting the transport protocol and the port number). Using software options, we forced our algorithm to ignore the last two tokens in the input data (see Section 6.2.2). We trained and tested the system using the ASNET1 dataset. Results are presented in Table 6.4a, with %TP and %FP of 82.1% and 2.3%, respectively ($\%TP_w$ and $\%FP_w$ of 95.8% and 27.9%).

The three most common errors made by the algorithm were classifying `www.facebook.com:443/TCP`, `www.google.com:443/TCP`, and `interia.hit.gemius.pl:443/TCP` as HTTP instead of HTTPS.

3) DNS-Class: port numbers (Table 6.4b): Conversely to the previous experiment, here we ignored the domain name and used only the port number and the transport protocol (see Section 6.2.2). Using the same dataset, we obtained the results presented in Table 6.4b, with %TP and %FP of 93.5% and <0.1%, respectively ($\%TP_w$ and $\%FP_w$ of 99.7% and <0.1%, respectively).

For this experiment, the most common errors were connected with the KASPERSKY protocol, e.g. erroneously classifying `ksn2-12.kaspersky-labs.com:443/TCP` as HTTPS.

4) DNS-Class: domain names and port numbers (Table 6.4c): Finally, in experiment 4 we evaluated the full DNS-Class algorithm, i.e. classification by domain name, port number, and transport protocol name. This is the main experiment that presents full capabilities of our algorithm. Again, we employed the ASNET1 dataset and 10-fold cross-validation. On average, we obtained %TP of 99.8% and %FP of <0.1% ($\%TP_w$ and $\%FP_w$ of 99.9% and <0.1%), which is displayed in greater detail in Table 6.4c.

The most common source of errors were due to the HTTPS, BITTORRENT, and SHOUTCAST protocols. For example, `petelo.streams.bassdrive.com-:80/TCP` was incorrectly classified as HTTP instead of SHOUTCAST. However, we also found a few errors in the ground-truth, e.g. textual inputs like `www.facebook.com:80/TCP` and `plus.google.com:80/TCP` were incorrectly attributed to the BITTORRENT protocol instead of HTTP.

5) Temporal stability (Table 6.4d): In experiment 5, we evaluated the temporal stability of DNS-Class, i.e. whether a classification model can be used in the same network after a longer period of time. We created the model using ASNET1, but for testing we took the ASNET2 dataset, collected after 8 months. We obtained %TP and %FP of 98.9% and <0.1%, respectively ($\%TP_w$ and $\%FP_w$ of 99.8% and <0.1%). Detailed results are given in Table 6.4d.

Many errors were due to the FLASHPLAYER protocol: DNS-Class failed for inputs like `telegraph.justin.tv:443/TCP` incorrectly classifying them as `HTTPS`. Again, we noticed a few errors in the ground-truth labels, e.g. `www.-facebook.com:443/TCP` attributed to BITTORRENT instead of HTTPS.

6.5 Discussion

Practical evaluation of DNS-Class on real Internet traffic showed that:

1. *Traditional port number classifier is unreliable for the traffic targeted by DNS-Class.* We evaluated CoralReef port number classifier on named flows, and the experiment showed that, as expected, it could not replace DNS-Class due to poor performance (%TP of 52%).
   (a) Classifying only by port number is unreliable also in the general case of all flows, i.e. named and anonymous flows. This was demonstrated in several papers (e.g. [81]) and in our experiment 1 (see Figure 6.5). The classifier worked properly for traditional protocols (e.g. NTP), but failed for newer P2P protocols (e.g. BITTORRENT).
   (b) The %TP metric for port number classification was better in the case of narrowing the traffic to named flows only (Figure 6.6). For example, %TP improved for RTMP and BITTORRENT, which shows an increase in performance at the cost of smaller scope.
2. *Domains alone are insufficient for successful traffic classification.* Experiment 2 (Table 6.4a) demonstrated that in 7 out of 14 traffic classes it was not possible to reliably identify the traffic using just the domain name.
   (a) The HTTP class collected majority of wrong classifications (%FP of 31%). This is due to popularity of this class, and because the set of words in website domains stands for a huge portion of tokens that DNS-Class can find in domain names of other protocols.
   (b) We noticed poor performance for FTP and FLASHPLAYER classes (%TP below 50%), and moderate results for HTTPS, JABBER, RTMP,

and BitTorrent (%TP below 90%). Probably, domain names of these protocols contain small number of tokens that could distinguish them from the dominant class.

(c) There exist domains that are used for delivering more than one service at the same time—for example, `poczta.o2.pl`: an e-mail service delivered through traditional SMTP/POP3 protocols, and through a web interface over HTTPS.

3. *Port number is an important traffic feature for named flows.* Experiment 3 (Table 6.4b) proved that in DNS-Class, port numbers can be used for successful classification in many cases, with %TP and %FP of 93.5% and <0.1%, respectively.
   (a) DNS-Class employs an ML algorithm (instead of relying on a static database), hence it learns the port numbers from the dataset.
   (b) The system was able to properly classify named flows of P2P protocols: BitTorrent (%TP of 99.3%) and Skype (%TP of 98.3%), with no false positives.
   (c) The HTTP and HTTPS classes collected majority of errors, because several other protocols use the port numbers 80 and 443, for example Kaspersky, FlashPlayer, and Shoucast.
4. *Full DNS-Class algorithm is reliable.* In experiment 4 (Table 6.4c), DNS-Class obtained %TP of 99.8% and %FP of <0.1%. This result demonstrates that it is possible to classify a significant portion of Internet flows using just the first packet and the flow name.
   (a) Combining port numbers with domain names generally improved the classification performance for each evaluated protocol.
   (b) The HTTP class collected majority of errors due to the reasons already given in points 2a and 3c above, but with a much lower %FP of <0.1%.
   (c) DNS-Class proved to be better than our ground-truth method in a few cases, as highlighted in Section 6.4.2, experiments 4 and 5.
5. *DNS-Class is stable over time.* In experiment 5 (Table 6.4d) we demonstrated that one can use the same model for classifying traffic after 8 months since training. Our algorithm achieved %TP of 98.9%, still with %FP below 0.1%.
   (a) The overall system performance was worse than for a fresh model, but still acceptable. The performance was lower because several new services—and several new domain names—appeared on the Internet.
6. *DNS-Class is effective for encrypted flows.* The traffic datasets Asnet1 and Asnet2 used for experiments 4 and 5 (Tables 6.4c and 6.4d) contained encrypted flows, e.g. HTTPS, SSH, and Skype. The DNS-Class algorithm was able to classify this traffic with %TP close to 100% and %FP close to 0%. Our method is thus effective for TLS flows.
   (a) DNS do not provide confidentiality of data, and so is its extension, the Domain Name System Security Extensions (DNSSEC). Thus, it is unlikely that in near future the development of the Internet infrastructure will prevent traffic classification using DNS.
   (b) In terms of performance metrics, DNS-Class performed equally well for encrypted and unencrypted traffic, e.g. for HTTPS and HTTP traffic.

6.6 Related works

In this chapter, we refer to *traffic classification* as described in related survey papers of [28, 70, 105], in which *traffic classes* (or *applications*) usually correspond to network protocols (e.g. FTP): see Tables I-III in [105], Table I in [28], and Tables II-V in [70]. All of the major works on traffic classification evaluate the performance of presented methods using various metrics, described in Sections II-B in [105], IV-A in [28], and 4.6 in [70]. That said, we believe DNS-Class is the first work that applies DNS to traffic classification while conforming to the state of the art. Below we reference previous works that laid the necessary groundwork, but which do not follow the standard practices in traffic classification, for various reasons (e.g. different goals). However, these works introduced many interesting and important ideas, foremost the idea of labeling IP flows with domain names.

In a 2011 paper [117], D. Plonka and P. Barford present a system for flexible traffic and host profiling using DNS. The system labels IP flows with domain names and allows for two kinds of analysis: according to the presence of flow name (named/unnamed) and according to the domain name (hierarchical analysis). The system also uses the domain names to attribute hosts to three P2P profiles of "Torrent", "Talk", and "Game". Then, the host profiles are used to label traffic classes, in a scheme that the authors call "indirect DNS rendezvous classification" (see Section IV-A2 in [117]). Comparing with DNS-Class, the paper adopts traffic classes that do not reflect particular applications or protocols (see Section V-B in [117]). The traffic traces used for experiments do not have ground-truth labels. The authors do not present essential metrics on the performance of their indirect classification method. Finally, the system employs a static set of textual patterns instead of an ML approach, which is simpler, but requires human work to match domain names with the traffic classes.

In Section 6.2.1 of this chapter, we describe an algorithm for labeling IP flows with domain names that is similar to the one already used in [117]. However, our algorithm also considers DNS records of `MX` type (SMTP servers), but ignores `AAAA` records (IPv6 addresses). Moreover, in Section IV of [117], the authors claim that "it is sufficient for the DNS pcap records to be observed before the application traffic pcap records". We find this questionable for the presented algorithm. For example, if two different websites are hosted at the same IP address, and the user connects to both of them in a short time, the off-line database described in [117] would register only the latter query. Consequently, labeling IP flows using such analysis would produce invalid results. In this paper, we propose a flow labeling scheme without such deficiency.

In a 2012 paper [15], I. Bermudez et al. present "DN-Hunter", a system that leverages the information carried in DNS traffic to analyze Content Delivery Networks (CDN). DN-Hunter labels flows with domain names, provides fine-grained traffic visibility, tracks and analyzes CDNs and their content, and determines popular network services running on given port number. The authors describe the architecture of DN-Hunter (which consists of a real-time DNS sniffer and an off-line analyzer), present DNS traffic characteristics, and apply the system to real traffic traces. Comparing with DNS-Class, the work [15] is devoted to a different goal of uncovering the global CDN structure and describing related network phenomena. Neither the DNS sniffer nor the off-line analyzer is used for traffic classification. However, Sections 4.3 and 5.5 in [15]

present a method for automatic discovery of network applications that run on any given port number, which resemble traffic classification, but targets port numbers instead of IP flows and thus is different. Consequently, the authors do not evaluate the performance of this method using classification metrics.

In Section 6.2.1 we describe DNS Search, which is equivalent to the DNS sniffer introduced in Sections 3.1 and 6 of [15]. Comparing with our work, the authors of [15] provide much deeper and more comprehensive analysis on the practical deployment and dimensioning of the DNS sniffer. However, we release our implementation as open source code instead of pseudo-code, making it readily available to other researchers. We also use hash tables instead of C++ maps, which is computationally faster. Another difference is in the procedure of tagging IP flows using the Resolver database: in case there is no match for given pair of client and server addresses, we additionally try the opposite direction, which can improve the hit ratio in some cases—for example, if the Flow Tagger misses some packets and the flow direction is accidentally alternated.

Finally, in a 2013 paper [52], P. Fiadino et al. present "HTTPTag": an online HTTP classification system, able to identify web-based applications and services. HTTPTag runs pattern matching on domain names extracted from HTTP headers, using a set of hand-made regular expressions. The authors claim that, using 380 regular expressions corresponding to 280 services, they are able to "classify more than 70% of the overall HTTP traffic volume caused by more than 88% of the web users in an operational 3G network". While their work has an alternative source of features and different goals than DNS-Class, it introduces the significance of domain names in classifying web traffic. Our work proposes a system that is more general (i.e. applies to many network protocols instead of just HTTP) and learns from data using ML (i.e. without the need of manual training).

6.7 Conclusions

In this chapter, we presented a novel practical system that immediately classifies a considerable portion of Internet traffic using DNS information and first packets of IP flows.

In Section 6.2, we described an algorithm that tags flows with domain names and classifies them using ML. In Section 6.3, we analyzed DNS traffic in the Asnet1 dataset, showing that e.g. 1) network protocols differ in how much they depend on DNS, and 2) named flows are on average twice bigger than anonymous flows, in terms of bytes. In Section 6.4, we demonstrated the robustness of DNS-Class by evaluating it on two traces of real traffic, obtaining very good performance results, which was discussed in Section 6.5. In Section 6.6, we justified our contribution to the state of the art in traffic classification and we commented on related works that analyze domain names.

We conclude that traffic classification using DNS gives very good results for named flows. Our work can be a motivation to employ DNS information as a traffic feature and to enrich flow exporting formats like Netflow or IPFIX with flow domain names. In Section 6.2.3, we gave our original vision for DNS-Class that explains its importance for TC. In next chapter, we build upon this vision by introducing the Waterfall cascade classifier.

Chapter 7

The Waterfall architecture

In this chapter, we present the capstone of the thesis, which connects all of the already presented concepts: an ML method for TC (see Chapters 3 and 2) that adopts our software tools (see Chapter 4) to connect different TC methods (see Chapters 5 and 6), with the goal of integrating many classifiers into one effective TC system (see Chapter 1).

7.1 Introduction

As already motivated in the thesis introduction, TC needs methods for integrating results of various research activities. Many new papers describe methods that in principle propose a set of traffic features optimized for another set of network protocols [3,14,46,52,56,62,63]. Researchers promote their methods for classifying network traffic, which are usually quite effective, but none of them is able to exploit all observable phenomena in the Internet traffic and identify all kinds of protocols.

The question arises: could we integrate these approaches into one system, so that we move forward, building on the achievements of our colleagues? How would this improve TC systems, in terms of accuracy, functionality, completeness, and speed? Answering these questions can open new perspectives. A robust method for combining classifiers can promote research that is more focused on new phenomena in the Internet, rather than addressing the same old issues. We need a way to complement and develop our existing methods further.

This work describes a new, modular architecture for TC systems: the Waterfall architecture. In a nutshell, it connects several classification modules in a chain and queries them sequentially, as long as none of them replies with a positive answer—i.e. the first module that identifies a flow wins. Typically, each module is a dedicated and very accurate classifier that targets a subset of network protocols, i.e. supports the *rejection option* (the "Unknown" class) [45]. The modules are ordered from the most reliable and specific to the most general and CPU-intensive. Waterfall follows the scheme of *cascade classification*, which is a type of *classifier selection* MCS technique (see Chapter 3).

The proposed architecture solves the integration problem. Each module can exploit different traffic features and address different kinds of network protocols, for example traditional client-server traffic, P2P, or tunneled traffic. The system

can be iteratively extended and updated as new network protocols emerge or new functionality requirements arise. Surprisingly, adding more classifiers can significantly reduce the total computation time (assuming proper ordering of the modules), which is the main advantage of Waterfall over popular *classifier fusion* approaches, e.g. BKS [41, 87].

This chapter describes a novel method with the following contributions:

1. It is the first application of cascade classification to the field of traffic classification (to the best of the knowledge of the authors). It represents an alternative to the BKS method (see Sections 7.2 and 7.3).
2. Waterfall lets for integration of independent algorithms and for iterative development of traffic identification systems, in a way similar to the *divide and conquer* algorithm design paradigm (see Sections 7.3 and 7.4).
3. It has an open source implementation in Python that shows excellent performance on real traffic and classifies flows in under 10 seconds of their lifetime (see Section 7.4 and Experiment 1 in Section 7.5).
4. Practical operation shows reduction in computation time with the increase in the number of modules, and that majority of traffic can be successfully classified using simple methods (see Experiments 1 and 2 in Section 7.5).
5. Proposes a new avenue for the future directions in the field of traffic classification (see Section 7.6, which concludes the chapter).

7.2 Background

A naïve approach to the integration problem would be to survey recent papers for traffic features and apply them as long feature vectors classified with a decent ML algorithm. Even with adequate techniques employed, this could quickly lead us to the *curse of dimensionality*: an exponential growth in the demand for training data as the feature space dimensionality increases (see Chapter 3). Besides, network flows differ in the set of available features, e.g. only a part of Internet flows evoke DNS queries (see Chapter 6). Some features need more packets to be computed, e.g. port number is available after 1 packet, whereas payload statistics need 80 packets in [56] (see Chapter 5). This means that different tools are needed for different protocols: some flows can be classified immediately using simple methods, while others need more sophisticated analysis. Finally, from the software engineering point of view, a big, monolithic system could be difficult to develop and maintain.

Instead, somes researchers adopted MCS—in particular, the BKS combination method that fuses outputs of many classifiers into one final decision. In principle, the idea behind BKS is to ask all classifiers for their answers on a particular problem $\mathbf{x}$ and then query a look-up table $\mathbf{T}$ for the the final decision. The table $\mathbf{T}$ is constructed during training of the system, by observing the behavior of classifiers on a labeled dataset. For example, if an ensemble of 3 classifiers replies (A, B, A) for a sample with a ground-truth label of B, then the cell in $\mathbf{T}$ under index (A, B, A) is B (see Chapter 3). This powerful technique can increase the performance of TC systems—as shown by Dainotti et al. in [41]—but comparing with Waterfall, it inherently requires *all* modules to be run on each traffic flow, with the drawback that the more modules are used, the more processing power is required.

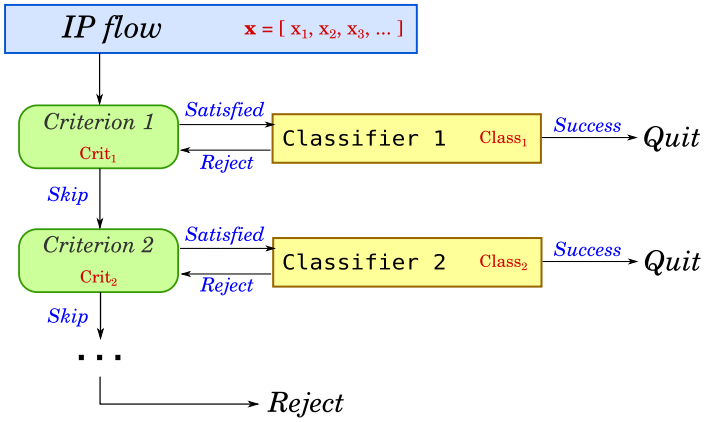

Figure 7.1: The Waterfall architecture. An IP flow is sequentially examined by the modules. In case of no successful classification, it is rejected.

7.3 The Waterfall architecture

The Waterfall idea is presented in Figure 7.1. The input to the system is an IP flow in form of a feature vector $\mathbf{x}$, which contains all the features required by all the modules, but a particular module will usually use only a subset of $\mathbf{x}$.

The system sequentially evaluates *selection criteria* that decide which *classification modules* to use for the problem $\mathbf{x}$. If a particular criterion is fulfilled, the associated module is run. If it succeeds, the algorithm finishes. Otherwise, or if the criterion was not satisfied, the process advances to the next step. When there are no more modules to try, the flow gets rejected and is labeled as "Unknown". More precisely,

$$Dec_i(\mathbf{x}) = \begin{cases} Class_i(\mathbf{x}) & Crit_i(\mathbf{x}) \text{ satisfied } \wedge\ Class_i(\mathbf{x}) \text{ successful} \\ Dec_{i+1}(\mathbf{x}) & \text{otherwise} \end{cases}, \tag{7.1}$$

$$Dec_{n+1}(\mathbf{x}) = Reject\,, \tag{7.2}$$

where Dec_i is the decision taken at step $i = \{1, 2, \ldots, n\}$, n is the number of modules, $Class_i(\mathbf{x})$ is the protocol identified by the module i, and $Crit_i(\mathbf{x})$ is the associated criterion.

The selection criteria are designed to skip ineligible classifiers quickly. For example, in order to implement a module that classifies traffic by analyzing the payload size of the first 5 packets in a flow, the criterion could check if at least 5 data packets were already sent in each direction. If this condition is true, a CPU-intensive ML algorithm could be run to try to identify the network protocol. In practice, a considerable amount of IP flows would be skipped, saving computing resources and avoiding classification with an inadequate method. On the other hand, if a flow satisfies this criterion, it would be classified with a method that does not need to support corner cases. The selection criteria are optional, i.e. if a module does not have an associated criterion, it is always run.

7.4 Practical implementation

A reference implementation of the Waterfall architecture is available as open source at [59]. It is implemented in C and Python in two parts: FLOWCALC modules, which prepare the flow feature vectors in form of ARFF files (see Chapter 4), and MUTRICS[1], which classifies the flows.

The reference MUTRICS classifier has several modules, described below. We highlight that these modules are merely an example, whereas much more advanced modules are possible (see Chapter 5).

1) ***dstip***: classification by destination IP address. During training, the module observes which remote destinations uniquely identify network protocols. If such particular IP address is popular enough, it is used as a rule for quick protocol identification by single lookup in a hash table.

2) ***dnsclass***: classification by DNS domain name of the remote host. In Chapter 6, we described how to obtain the textual host names associated with network flows and how to use this information for TC. This module implements the DNS-Class algorithm and extends it with a mechanism for detecting unknown protocols. The selection criterion checks if a particular flow has an associated DNS name, or whether it is a DNS query-response flow.

3) ***portsize***: classification by the port number and packet size. In a way similar to `dstip`, the module observes which tuples of transport protocol, port number, and payload size of the first packets in both directions uniquely identify network protocols. Popular tuples are stored in a hash table. The selection criterion checks if the flow feature vector contains packet payload sizes.

4) ***npkts***: classification by packet sizes. Uses payload sizes of the first 4 packets in both directions, plus the transport protocol and the port number. Employs the random forest ML algorithm, which is a multi-classifier that combines decision trees [45, 87]. The selection criterion is the same as in `portsize`.

5) ***port***: classification by the port number. The module uses the classic pair of transport protocol and port number to find the pairs that uniquely and reliably identify network protocols, similarly to `dstip` and `portsize`. Classification requires a single lookup in a hash table.

6) ***stats***: classification by flow statistics. The module uses the same ML algorithm as `npkts`. As features, it uses the following statistics of packet sizes and inter-arrival times in both directions: the minimum, the maximum, the average, and the standard deviation—i.e. a total of 16 statistics.

The FLOWCALC part of the system, responsible for computing the feature vectors, was limited to only consider the first 10 seconds of traffic in each flow. This simulates a real-time scenario in which the network protocol must be identified under a given time limit. All experiments presented in the next section were run with such constraints to demonstrate real-time traffic classification.

7.5 Experiments

In this section, the results of two experiments are presented: 1) classification performance on real network traffic, and 2) effect of adding new modules.

[1] The name comes from "Multilevel Traffic Classification"

7.5.1 Methodology

Four traffic datasets were used for experimental validation: a) ASNET1, collected at a Polish ISP company serving <500 residential customers, b) ASNET2, collected at the same network, c) IITIS1, collected from the network of the IITiS institute serving <50 academic users, and d) UNIBS, collected from the campus network of the University of Brescia serving 20 workstations[2]. The ASNET1 and ASNET2 datasets were collected at the same gateway router, but with a time gap of 8 months. The ASNET1 and IITIS1 datasets were collected at different networks, but at the same time. The UNIBS dataset was collected a few years earlier than the other datasets, contains no packet payload, and has the IP addresses anonimized. Details are presented in Table 7.1e (pp. 72).

DPI was employed for establishing the ground-truth labels on the datasets (a)-(c). Note that DPI is not perfect—as shown by Dusi et al. [48]—but it is the most popular method used in the literature, and often the only practically available. The LIBPROTOIDENT v. 2.0.7 was used as the DPI software (reported to offer very good accuracy in [21]). The UNIBS dataset already contained ground-truth and was not suitable for DPI because of no payload data. Finally, the datasets were sanitized by dropping flows that had no data transmitted in both directions, e.g. incomplete TCP sessions and empty UDP flows. The datasets contain different subsets of network protocols (see Table 7.1).

For measuring the classification accuracy for a given protocol p, the popular $\%TP_p$ and $\%FP_p$ metrics were employed:

$$\%TP_p = \frac{|TP_p|}{|F_p|} \cdot 100\%, \quad \%FP_p = \frac{|FP_p|}{|F'_p|} \cdot 100\%, \tag{7.3}$$

where TP_p is the set of true positives for protocol p, F_p is the set of all testing flows for protocol p that were classified, FP_p is the set of false positives for p, and F'_p is the set of all testing flows for all protocols except p that were classified. For evaluating the overall accuracy, the average values of these metrics were used—the $\%TP$ and $\%FP$ metrics—which were complemented with the $\%Unk$ metric that measures the amount of rejected flows:

$$\%Unk = \frac{|U|}{|F|} \cdot 100\%, \tag{7.4}$$

where U is the set of rejected flows, and F is the set of all testing flows.

For dividing the data into training and testing parts, a 60%/40% split was used on ASNET1 and on UNIBS, i.e. 60% of their flows were randomly selected for training, and 40% for testing. The classifier trained on ASNET1 was validated on the rest of ASNET1 (to evaluate the "classical" classification performance), and on the whole ASNET2 and IITIS1 datasets (so as to demonstrate stability in time and space). The classifier trained on UNIBS was validated only on the rest of the UNIBS dataset: this tested operation on a trace without packet payloads.

7.5.2 Results

In the Experiment 1, the system was evaluated for classification performance on the datasets (a)-(d), with 5 modules enabled: `dstip`, `dnsclass`, `portsize`,

[2]Downloaded from `http://www.ing.unibs.it/ntw/tools/traces/`

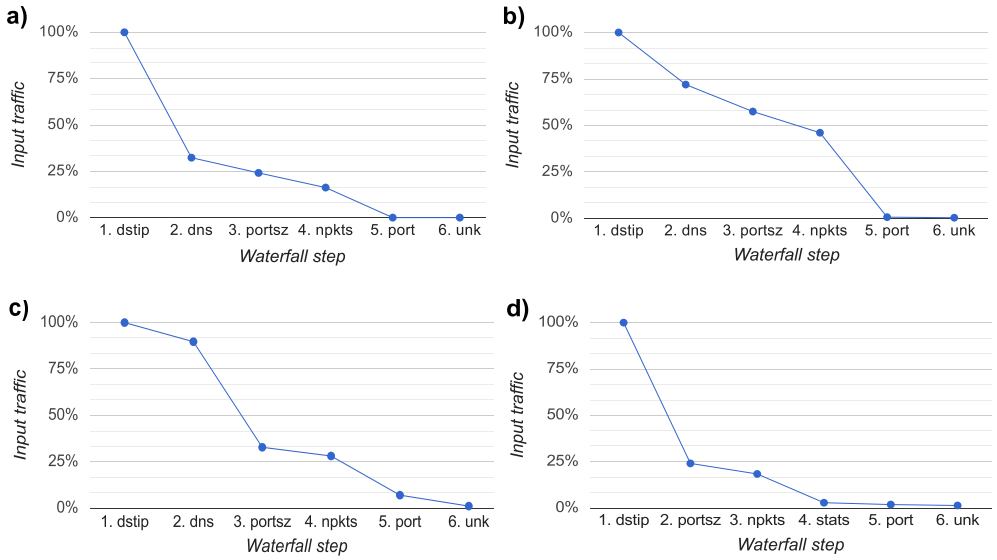

Figure 7.2: Experiment 1: amount of traffic passing through successive waterfall steps, for datasets: a) ASNET1, b) ASNET2, c) IITIS1, and d) UNIBS. 5 modules were enabled; "dns" means `dnsclass` and "portsz" means `portsize`.

`npkts`, and `port`. For the UNIBS dataset, `stats` was used instead of `dnsclass`, because this dataset had no DNS payload packets.

The results in form of confusion matrices and performance metrics are presented in Table 7.1, parts (a)-(d). For all datasets, the $\%TP$ and $\%FP$ metrics were close to 100% and 0% respectively, which indicates high classification performance of the system. The classifier successfully identified all protocols, including: BITTORRENT, SKYPE, KADEMLIA, SSH, STUN, WWW, and more. For the IITIS1 dataset, the system made no errors in classifying over 1.5 million flows; for other datasets, the number of errors was well below 0.1%. The $\%Unk$ metric was 0.1%, 0.4%, 1.1%, and 1.4%, for ASNET1, ASNET2, IITIS1, and UNIBS, respectively—i.e. almost all flows were classified.

Figure 7.2 shows traffic progress through the system: the figure presents the percentage of IP flows at the input of successive modules. An IP flow leaves the system as soon as it gets classified, so the figure visualizes how many flows get through the end of the waterfall. It is apparent that the amount of traffic that a particular module can classify depends on the dataset. For all datasets, more than half of the flows were classified using simple methods—namely the `dstip`, `dnsclass`, and `portsize` modules—without the need to run the `npkts` module, which employs a sophisticated ML algorithm.

In the Experiment 2, the effect of increasing the number of modules was studied. The system started in configuration with only one module enabled: the `npkts` module, which is CPU-intensive. In each iteration, one new module was added—usually at the front of the waterfall—and the whole system was given the task of classifying the same dataset. The experiments were run in separation of each other, on a single core of an Intel Core i7 machine[3] (i.e.

[3]Intel Core i7-930 2.80GHz, 8GB RAM, 128GB SSD

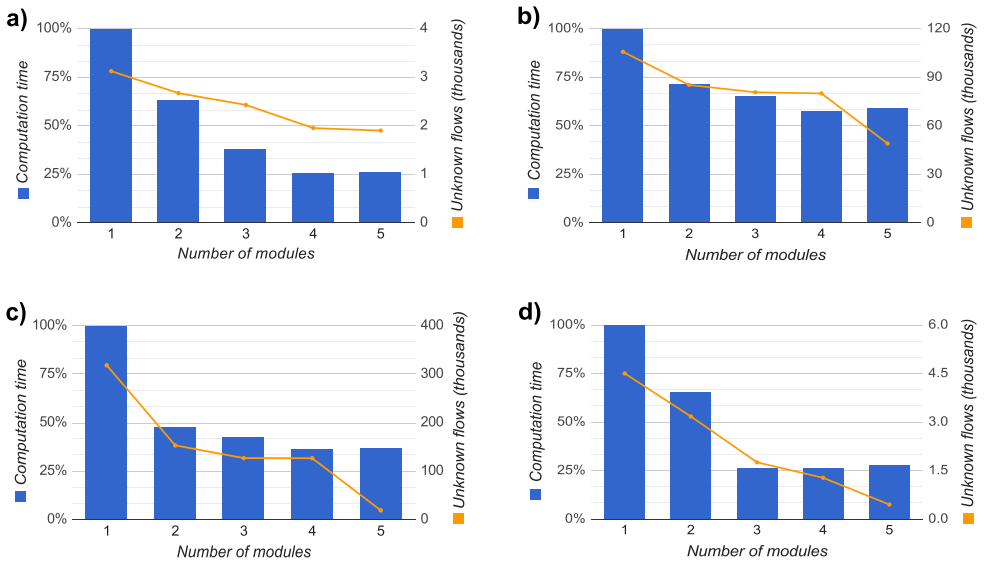

Figure 7.3: Experiment 2: effect of adding modules on computation time (bars) and on the number of unknown flows (lines), for datasets a) ASNET1, b) ASNET2, c) IITIS1, and d) UNIBS.

single-threaded implementation). The time needed to finish the computations was measured relatively to the first run. The amount of unclassified flows was measured in absolute numbers.

The results are displayed in Figure 7.3. The computation time generally decreased with new modules being enabled. The modules were added in the following order: `npkts`, `dnsclass`, `portsize`, `dstip`, and `port`—that is, from most to least CPU demanding. Motivation for this was to resemble a scenario in which a generic TC algorithm is iteratively augmented with dedicated classifiers. In each iteration, the number of unclassified flows dropped. The $\%TP$ and $\%FP$ metrics were stable and close to perfect values.

7.6 Conclusions

This chapter presented Waterfall—a novel, modular architecture for TC that lets for integration of many algorithms and exhibits a decrease in the total computation time with new modules being added. An illustrative, open source implementation of the system—the MUTRICS classifier—showed very good performance results on 4 real traffic datasets, in a scenario that resembles real-time traffic classification. The paper experimentally proved that the majority of IP flows can be immediately classified using simple methods, which exploit basic traffic features like: destination IP address, DNS domain name, and packet size.

The chapter concludes with a positive answer to the question whether many independent traffic classification algorithms could be integrated, giving good results in terms of many metrics.

a)

Protocol	Flows	A	B	C	D	E	F	G	H	I	J	K	L	M	%TP	%FP
A = BitTorrent	799,234	99.9%								<0.1%				<0.1%	99.9%	<0.1%
B = DNS	723,478		100%												100%	0%
C = eMule	75,555			99.9%										<0.1%	99.9%	<0.1%
D = Jabber	1,388				100%										100%	0%
E = Kademlia	148,824					100%									100%	<0.1%
F = Kaspersky	17,479						100%								100%	0%
G = Mail	7,982							100%							100%	0%
H = NTP	3,506								100%						100%	0%
I = Skype	55,070	<0.1%		<0.1%		<0.1%				99.9%					99.9%	<0.1%
J = SSH	3,620										100%				100%	0%
K = Steam	50,963											100%			100%	0%
L = STUN	6,828												100%		100%	0%
M = WWW	1,695,282													100%	100%	<0.1%
	3,589,209													*Avg.*	99.9%	<0.1%

b)

Protocol	Flows	A	B	C	D	E	F	G	H	I	J	K	L	M	%TP	%FP
A = BitTorrent	4,788,510	99.9%		<0.1%						<0.1%	<0.1%		<0.1%	<0.1%	99.9%	<0.1%
B = DNS	1,810,622		100%												100%	0%
C = eMule	172,676	<0.1%		99.9%						<0.1%					99.9%	<0.1%
D = Jabber	3,110				100%										100%	0%
E = Kademlia	308,926	<0.1%				99.9%				<0.1%			<0.1%		99.9%	<0.1%
F = Kaspersky	46,475						96%							4%	96%	<0.1%
G = Mail	42,055							100%							100%	0%
H = NTP	10,940								100%						100%	0%
I = Skype	112,748	0.1%				0.5%				99.4%					99.4%	<0.1%
J = SSH	6,223										100%				100%	<0.1%
K = Steam	78,167	<0.1%										99.9%			99.9%	0%
L = STUN	12,009	<0.1%											99.9%		99.9%	<0.1%
M = WWW	3,828,118						<0.1%							99.9%	99.9%	<0.1%
	11,220,579													*Avg.*	99.6%	<0.1%

c)

Protocol	Flows	A	B	C	D	E	F	G	H	%TP	%FP
A = BitTorrent	8,811	100%								100%	0%
B = DNS	1,018,193		100%							100%	0%
C = Jabber	137			100%						100%	0%
D = Mail	108,296				100%					100%	0%
E = NTP	4,827					100%				100%	0%
F = Skype	21						100%			100%	0%
G = SSH	17,764							100%		100%	0%
H = WWW	624,511								100%	100%	0%
	1,782,560								*Avg.*	100%	0%

d)

Protocol	Flows	A	B	C	D	E	%TP	%FP
A = BitTorrent	2,928	99.8%	0.1%			<0.1%	99.8%	<0.1%
B = eMule	5,600		99.9%			<0.1%	99.9%	<0.1%
C = Mail	1,971			99.3%		0.7%	99.3%	<0.1%
D = Skype	1,484	0.1%			99.8%	0.1%	99.8%	0%
E = WWW	18,798			<0.1%		99.9%	99.9%	0.2%
	30,781					*Avg.*	99.7%	0.1%

e)

Dataset	Start	Duration	Src. IP	Dst. IP	Packets	Bytes	Avg. Util	Avg. Flows (/5 min.)	Payload
Asnet1	2012-05-26 17:40	216 h	1,828 K	1,530 K	2,525 M	1,633 G	18.0 Mbps	7.7 K	92 B
Asnet2	2013-01-24 16:26	168 h	2,503 K	2,846 K	2,766 M	1,812 G	25.7 Mbps	12.0 K	84 B
Iitis1	2012-05-26 11:19	220 h	32 K	46 K	150 M	95 G	1.0 Mbps	753.7	180 B
Unibs	2009-09-30 11:45	58 h	27	1 K	33 M	26 G	0.9 Mbps	111.7	0 B

Table 7.1: Experiment 1: classification performance and the datasets. Metrics for validation on 4 real traffic datasets: a) ASNET1, b) ASNET2, c) IITIS1, and d) UNIBS. Part e) presents details on the datasets used in the paper.

Chapter 8

Optimizing cascade classifiers

Building upon the ideas presented in the previous Chapter 7, below we show how to optimize a Waterfall TC system for desired performance. We describe an algorithm that quickly simulates all cascade configurations for a pool of classifiers, which lets for minimizing CPU time, number of errors, and situations in which IP flows are left unrecognized. Besides the content presented below, we guide the reader again to Chapter 2 for background on designing TC systems, which puts TC performance in context of specific networks and their needs.

8.1 Introduction

The Waterfall architecture integrates many different classifiers in a cascade. The system sequentially evaluates module selection criteria and decides which modules to use for a given classification problem $\mathbf{x}$. If a particular module is selected and provides a label for $\mathbf{x}$, the algorithm finishes. Otherwise, the process advances to the next module. If there are no more modules, the flow is labeled as "Unknown".

We described cascade classifiers and other MCS techniques in Chapter 3. Interestingly, although cascade classifiers were first introduced in 1998 by E. Alpaydin and C. Kaynak [8], so far few authors considered the puzzle of *optimal cascade configuration* that would match Waterfall. In a 2006 paper [32], K. Chellapilla et al. propose a cascade optimization algorithm that updates the rejection thresholds of the constituent classifiers. The authors apply an optimized depth first search to find the cascade that satisfies given constraints on time and accuracy. However, comparing with this work, their system does not optimize the ordering of modules. In another paper on this topic, published in 2008 by A. Sherif [1], the author proposes a greedy approach for building cascades: i.e., start with a generic solution and sequentially prepend a module that reduces CPU time. Comparing with this work, that algorithm does not evaluate all possible cascade configurations and thus can lead to suboptimal results. We will demonstrate this in Section 8.5 using a myopic optimizer.

Thus, we propose a new solution to the cascade classification problem, which is better suited for traffic classification than existing methods. Note that com-

paring with [32] we do not consider rejection thresholds as input values to the optimization problem. Instead, in case of classifiers with tunable parameters, one could consider the same module parametrized with different values as separate modules, and apply our technique as well. For instance, a Bayes classifier with rejection thresholds on the posterior probability of $0.5, 0.75, 0.90$ would be considered as 3 different classifiers.

8.2 Problem statement

Let us consider the problem of optimizing a cascade of classifiers. Let F be a set of IP flows, and E be a set of n classification modules,

$$E = \{1, \ldots, n\} \tag{8.1}$$

that we want to use for cascade classification of flows in F in an optimal way. In other words, we need to find a sequence of modules X,

$$X = (x_1, \ldots, x_m) \quad m \leq n,\ x_i \in E,\ x_i \neq x_j \ \textit{for}\ i \neq j \tag{8.2}$$

that minimizes a cost function C,

$$C(X) = f(T_X) + g(E_X) + h(U_X) \tag{8.3}$$

where the terms T_X, E_X, and U_X respectively represent the total amount of CPU time used, the number of errors made, and the number of flows left unlabeled while classifying F with X. The terms f, g, and h denote arbitrary real-valued functions. Because $m \leq n$, some modules may be skipped in the optimal cascade. Note that U_X does not depend on the order of modules, because unrecognized flows always traverse till the end of the cascade.

8.3 Proposed solution

To find the optimal cascade, we propose to simulate the performance of every possible X by measuring the performance of each module separately, and then smartly combining the results.

Note that for an accurate solution one would basically need to run the full classification process for all permutations of all combinations in E. This would take S experiments, where

$$S = \sum_{i=1}^{n} \frac{n!}{(n-i)!} \approx e \cdot n! \tag{8.4}$$

which is impractical even for small n. On another hand, fully theoretical models of the cost function seem infeasible too, due to the complex nature of the cascade and module inter-dependencies.

Thus, we propose an approximate solution to the cascade optimization problem. The algorithm has two evaluation stages described below:

A. **Static:** classify all flows in F using each module in E, and
B. **Dynamic:** find the X sequence that minimizes $C(X)$.

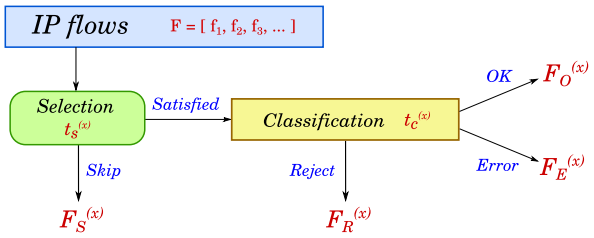

Figure 8.1: Measuring performance of module $x \in E$.

A. Static Evaluation

In every step of stage A, we classify all flows in F using each single module $x \in E$. We measure the average CPU time used for flow selection and classification: $t_s^{(x)}$ and $t_c^{(x)}$. We store each output flow identifier in one of the three outcome sets, depending on the result: $F_S^{(x)}$, $F_O^{(x)}$, or $F_E^{(x)}$. These sets hold respectively the flows that were skipped, properly classified, and improperly classified. Let us also introduce $F_R^{(x)}$,

$$F_R^{(x)} = F \setminus (F_S^{(x)} \cup F_O^{(x)} \cup F_E^{(x)}) \tag{8.5}$$

that is, the set of rejected flows. See Fig. 8.1 for an illustration of the module measurement procedure. As the result of every step, the performance of module x on F is fully characterized by a tuple $P^{(x)}$,

$$P^{(x)} = (\ F,\ t_s^{(x)}, t_c^{(x)}, F_S^{(x)}, F_O^{(x)}, F_E^{(x)}\ ). \tag{8.6}$$

Finally, after n steps of stage A, we obtain n tuples: a model of our classification system, which is the input to stage B.

B. Dynamic Evaluation

Having all of the required experimental data, we can quickly estimate $C(X)$ for arbitrary X. Because f, g, h, are used only for adjusting the cost function—and can be modified by the network administrator according to the needs—we focus only on their arguments, i.e. the cost factors T_X, E_X, and U_X.

Let $X = (x_1, \ldots, x_i, \ldots, x_m)$ represent certain order and choice of modules, and G_i represent the set of flows entering the module number i, so that:

$$G_1 = F \tag{8.7}$$

$$G_{i+1} = G_i \setminus (F_O^{(x_i)} \cup F_E^{(x_i)}) \quad 1 \le i \le m \tag{8.8}$$

Then, we estimate the cost factors using the following procedure:

$$T_X \approx \sum_{i=1}^{m} |G_i| \cdot t_s^{(x_i)} + |G_i \setminus F_S^{(x_i)}| \cdot t_c^{(x_i)} \tag{8.9}$$

$$E_X = \sum_{i=1}^{m} |G_i \cap F_E^{(x_i)}| \tag{8.10}$$

$$U_X = |G_{m+1}| \tag{8.11}$$

where $|G|$ denotes the number of flows in set G.

Note that the difference operator in Eq. 8.8 connects the static cost factors with the dynamic effects of a cascade. In stage A, our algorithm evaluates static performance of every module on the entire dataset F, but in stage B we want to simulate cascade operation, so we need to remove the flows that were classified in the previous steps. Thus, the operation in Eq. 8.8 is crucial.

Module performance depends on its position in the cascade, because preceding modules alter the distribution of traffic classes in the flows conveyed onward. For example, we can improve accuracy of a port-based classifier by putting a module designed for P2P in front of it, which should handle the flows that misuse the traditional port assignments.

8.4 Discussion

In our solution, we reduced the number of experiments from $e \cdot n!$ (see Eq. 8.4) down to n, which is an extreme improvement that makes cascade optimization practical. Another optimization comes from reducing the number of computations: when a new module x_j is added to an already simulated cascade X, we can re-use previous computations for X as follows:

$$G_j = U_X \tag{8.12}$$

$$T_{X+x_j} \approx T_X + |G_j| \cdot t_s^{(x_j)} + |G_j \setminus F_S^{(x_j)}| \cdot t_c^{(x_j)} \tag{8.13}$$

$$E_{X+x_j} = E_X + |G_j \cap F_E^{(x_j)}| \tag{8.14}$$

$$U_{X+x_j} = G_j \setminus (F_O^{(x_j)} \cup F_E^{(x_j)}) \tag{8.15}$$

Thus, we suggest searching for the minimum $C(X)$ in a recursive algorithm.

However, note that although simulation is orders of magnitude faster than experimentation, we still check every possible cascade. This makes the time complexity of our algorithm factorial, considering set computations as the elementary operations. This leaves space for further improvements by introducing another set of heuristics tuned to a specific cost function.

Moreover, note that the results depend on F: the optimal cascade depends on the protocols present in the traffic, and on the ground-truth labels. The presented method cannot provide the ultimate solution that would match every network, but it can optimize a specific cascade system for a specific network. We discuss this issue in Section 8.5 and in Chapter 2.

We assume that the flows are independent of each other, i.e. labeling a particular flow does not require information on any other flow. If such information is needed, e.g. flow DNS names, it should be extracted before the classification process starts. Thus, traffic analysis and flow classification must be separated to uphold this assumption. We successfully implemented such systems for our DNS-Class and Mutrics classifiers (see Chapters 6 and 7, respectively).

In the next Section, we experimentally validate our method and show that it perfectly predicts E_X and U_X, and approximates T_X properly. The simulated cost follows the real cost, so we claim our proposal is valid and can be used in practice. We also analyze the trade-offs between speed, accuracy, and ratio of unlabeled flows, to stress out that the final choice of the cost function should depend on the purpose of the system.

Dataset	Start	Duration	Src. IP	Dst. IP	Packets	Bytes	Avg. Util	Avg. Flows (/5 min.)	Payload
Asnet1	2012-05-26	216 h	1,800 K	1,500 K	2,500 M	1,600 G	18 Mbps	7.7 K	92 B
Asnet2	2013-01-24	168 h	2,500 K	2,800 K	2,800 M	1,800 G	26 Mbps	12 K	84 B
IITiS1	2012-05-26	216 h	32 K	46 K	150 M	95 G	1.0 Mbps	750	180 B
Unibs1	2009-09-30	58 h	27	1 K	30 M	26 G	0.9 Mbps	110	0 B
UPC1	2013-02-25	65 d	90 K	18 K	37 M	33 G	51 Kbps	68	full
	2013-11-18	35 d	7.5 K	54 K	43 M	31 G	88 Kbps	49	full

Table 8.1: Datasets used for experimental validation.

Module	ML algorithm	Traffic features
`dnsclass`	linear SVM	DNS name
`dstip`	lookup table	destination IP address
`npkts`	random forest	payload sizes: first 4 packets in+out
`port`	lookup table	destination port number
`portsize`	lookup table	payload sizes: first packet in+out
`portname`	lookup table	DNS name
`stats`	random forest	4 basic statistics of packet sizes and inter-arrival times

Table 8.2: Waterfall modules used for experimental validation.

8.5 Experimental validation

Below we present the outcome of using real traffic datasets for experimental evaluation of our proposal. We ran 4 experiments:

1. comparing simulation with reality, which proves validity of Eqs. 8.9-8.11;
2. analyzing the effect of cost function parameters on the result, which demonstrates optimization for different goals;
3. optimizing on one dataset and using the cascade on another dataset, which evaluates stability;
4. comparing our optimization method with myopic and random cascade optimization, which shows that our work is meaningful.

For the experiments, in general we used 5 datasets, summarized in Table 8.1. Datasets ASNET1 and ASNET2 were collected at the same ISP serving <500 domestic users, with an 8-month time gap. Dataset IITIS1 was collected at an academic network serving <50 researchers, at the same time as ASNET1. Dataset UNIBS1 was also collected at an academic network—at the University of Brescia [50]—but a few years earlier and using a reliable ground-truth information [73] (this dataset was anonymized). Finally, the UPC1 dataset was artificially generated—with manual simulation of different human behaviors—hence it contains full packet payloads and the names of applications that generated the traffic flows [22, 29, 31].

For the first 3 datasets, we established ground-truth using light DPI [6]. For UNIBS1 and UPC1, we used the supplied ground-truth information, which sometimes was challenging: for example, a `skype` process generates some HTTP traffic apart of the SKYPE protocol. For each dataset, we trained the modules using 60% random sample of all flows, and used the remainder for testing. We considered only the first 10 seconds of each flow to resemble a near-immediate traffic identification.

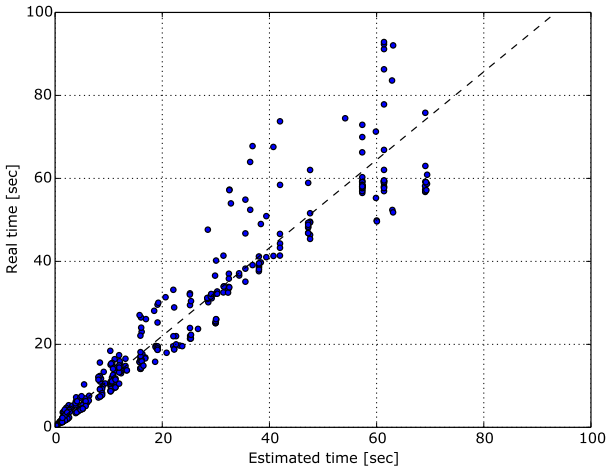

Figure 8.2: Experiment 1. Estimated classification time vs real classification time. Dashed line shows least-squares approximation. The Pearson product-moment correlation is 0.95.

Finally, in total we evaluated 7 classification modules, summarized in Table 8.2 (see Chapters 4, 6, and 7 for more details). As additional traffic features, we used the transport protocol and destination port number for each module. Although we consider port numbers as an unreliable feature, they still can provide valuable hint for more sophisticated classification mechanisms. Note that the modules support the *reject option*, so each module can drop any flow if its not certain about the outcome.

8.5.1 Experiment 1

In the first experiment, we compare simulated cost factors with real values for arbitrary cascade configurations. We randomly selected 100,000 flows from each of the first 4 datasets and ran static evaluation on them. Next, we generated 100 random cascades, and for each cascade we ran both real and simulated classification. As a result, we obtained corresponding pairs of real and estimated values of T_X, E_X, and U_X.

The results for T_X are presented in Fig. 8.2. For E_X and U_X, we did not observe a single error, i.e. our method perfectly predicted the real values. For CPU time estimations, we see a high correlation of 0.95, with little underestimation of the real value. For all datasets, the estimation error was below 20% for majority of evaluated cascades (with respect to the real value). The error was above 50% only for 5% of evaluated cascades.

We conclude that in general our method properly estimates the cost factors and we can use it to simulate different cascade configurations. Note that accurate prediction of the CPU time is not necessary for optimization: it is enough for the simulated time to be roughly proportional to the real value. Moreover, even the real values will vary depending e.g. on the CPU load due to other tasks executed in the background, which is difficult to predict.

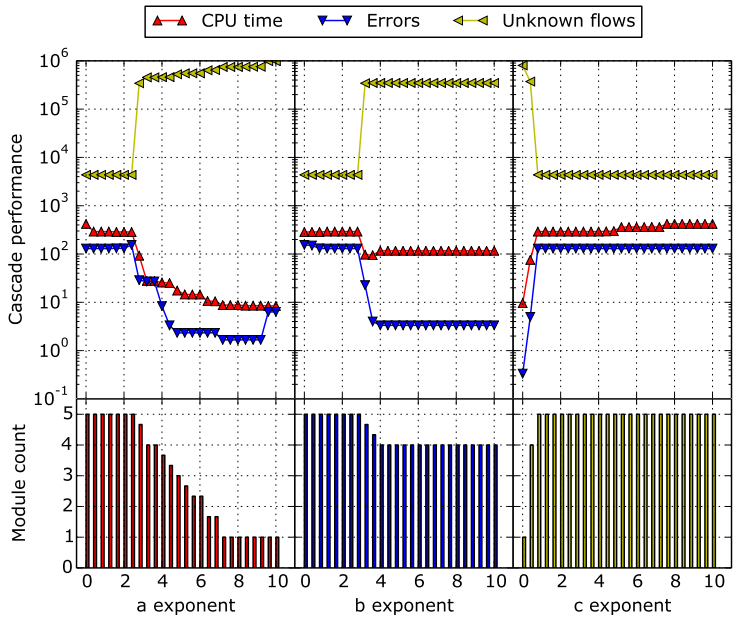

Figure 8.3: Experiment 2. Optimizing the cascade for different goals: best classification time (a exponent), minimal number of errors (b exponent), and the lowest number of unlabeled flows (c exponent): the plot shows the averages for 3 datasets.

8.5.2 Experiment 2

In our second experiment we show the effect of tuning the system for 3 different goals: a) minimizing the computation time, b) minimizing errors, and c) labeling as many flows as possible. We chose the following cost function:

$$C(X) = f(T_X) + g(E_X) + h(U_X) = (T_X)^a + (E_X)^b + (U_X)^c \tag{8.16}$$

with the default values of a, b, c equal to $0.95, 1.75, 1.20$, respectively. We separately varied these values in range of 0-10, and observed the performance of the resultant cascades. For the sake of brevity, we ran the experiment for datasets ASNET1, ASNET2, and IITIS1 and for modules `dnsclass`, `dstip`, `npkts`, `port`, and `portsize`.

In Fig. 8.3, we present the results: dependence of cascade performance and module count on the cost function parameters. As expected, higher a exponent leads to faster classification and usually less errors, but with fewer modules in the cascade, and more unclassified flows as a consequence. Optimizing for accuracy—higher b exponent—leads to reduction of errors at the cost of higher number of flows left without a label. Finally, if we choose to classify as much traffic as possible (increasing the c exponent), the system will use all available modules, at the cost of higher CPU time and error rate.

In more detail, for time optimization, the *optimal* cascades are: `port` for ASNET1, `portsize` for ASNET2, and `dnsclass` for IITIS1. In the last case, `dnsclass` is preferred due to high percentage of DNS traffic in IITIS1. Instead, in case of accuracy optimization, the *optimal* cascades are: `portsize`,

Reference	Test dataset			
	Asnet1	***Asnet2***	***IITiS1***	***UPC1***
Asnet1		1.01%	5.31%	48.96%
Asnet2	2.67%		7.29%	23.34%
IITiS1	33.37%	34.19%		192.91%
UPC1	14.51%	11.11%	31.77%	

Table 8.3: Experiment 3. Result stability: relative increase in the cost $C(X)$, depending on the reference dataset used for determining the optimal cascade.

`dnsclass, npkts, port` for ASNET1, `dstip, dnsclass, portsize` for ASNET2, and `dnsclass, port, dstip, portsize, npkts` for IITIS1. Finally, optimizing for minimum percentage of unrecognized flows yields a common result for all datasets: `dnsclass, dstip, npkts, port, portsize`.

Note that the results depend on the cost function. We used a power function for presentation purposes, in order to easily show contrasting scenarios by small adjustments to the exponents. For specific purposes, a multi-linear function may be more appropriate, as it is often found in the literature, e.g. linear scalarization of multi-objective optimization problems. Moreover, more complex expressions—including thresholds on some parameters—can be used to find a classification system capable of real-time operation: given an expected amount of flows per second, one could find a cascade that is fast enough to handle the traffic while keeping the other cost factors at possible minimum.

We conclude that our proposal works and is adaptable, i.e. by varying the parameters we optimized the classification system for different goals.

8.5.3 Experiment 3

In the third experiment, we verify if the result of optimization is stable in time and space, i.e. if an optimal cascade stays optimal with time and changes of the network. We ran the optimization for 4 datasets, obtaining different configuration for each dataset. Next, we evaluated these configurations on all datasets and measured the increase in the cost function $C(X)$ compared with the original value. Note that we did not use the UNIBS1 dataset for this experiment, as it lacks packet payloads and hence needs different set of available modules.

Table 8.3 presents the results. We see that our proposal yielded results that are stable in time for the same network: the cascades found for ASNET1 and ASNET2, which are 8 months apart, are similar and can be exchanged with little decrease in performance. However, the cascades found for ASNET1 and ASNET2 gave 5-7% worse performance compared with IITIS1, and 23-49% worse performance on UPC1. We observed extreme decrease in performance when we varied both the network and time, especially when classifying UPC1 with cascade optimized for IITIS1.

We conclude that cascade optimization is specific to the network, but on the other hand our results suggest that an optimal cascade does not change significantly with time for given network. Thus, the network administrator does not need to repeat the optimization procedure frequently.

Dataset	Algorithm	Cascade configuration	Time [s]	Errors	Unknowns
Asnet1	myopic	portname,portsize,port,dstip,dnsclass,stats,npkts	89	40	886
	optimal	portsize,portname,dstip,dnsclass,npkts,port,stats	87	30	886
			+2%	**+26%**	**0%**
Asnet2	myopic	portname,portsize,port,dstip,dnsclass,stats,npkts	141	49	817
	optimal	portsize,portname,dstip,dnsclass,npkts,port	139	22	1,224
			+2%	**+55%**	**-50%**
IITiS1	myopic	dnsclass,port,portname,portsize,dstip,stats,npkts	5.7	2.4	80
	optimal	port,portsize,npkts,stats	5.1	2.4	80
			+11%	**+0%**	**0%**
Unibs1	myopic	portsize,port,dstip,stats,npkts	102	2,017	13,892
	optimal	dstip,portsize,port,npkts,stats	91	1,985	13,892
			+10%	**+2%**	**0%**
UPC1	myopic	portname,port,portsize,dstip,dnsclass,stats,npkts	110	686	1,746
	optimal	port,portname,dstip,portsize,dnsclass,npkts,stats	92	604	1,746
			+16%	**+12%**	**0%**
		Average improvement:	**+8%**	**+19%**	**-10%**

Table 8.4: Experiment 4. Average improvements compared with myopic optimization.

8.5.4 Experiment 4

In the last experiment, we compare our proposal with a greedy optimizer, i.e. a situation in which we arrange modules by increasing CPU time. This resembles the basic approach used in Chapter 7: start with a generic sophisticated classifier and prepend faster modules in front of it. Thus, for each module we calculated the sum of t_s and t_c for each dataset separately, and ordered the modules from the fastest to the slowest. We used the results as cascade configurations, i.e. Waterfall systems configured with a conservative algorithm: "myopic" optimization.

On the other hand, we also optimized the system using our proposal, with the cost function given in Eq. 8.16, for a, b, c equal to $3.00, 1.75, 1.50$, respectively. We chose these exponent values arbitrarily to show an example of time optimization: note that the a exponent (influencing the time cost factor) is the highest. Then, we used the results as cascade configurations, but optimized with an "optimal" algorithm.

Table 8.4 compares the results: in every case, our algorithm optimized the classification system to work faster and with less errors, usually with the same amount of unclassified flows. This demonstrates the point of cascade optimization: it brings performance improvements. Recall that UNIBS1 lacks packet payloads, hence we used 5 modules in general for this dataset instead of 7.

On average, the system worked 8% faster compared with myopic time optimization, and reduced the error rate by 19%. For ASNET2, it also resulted in higher number of unrecognized flows, but the increase is insignificant given the dataset size, and this cost factor was not the goal of optimization. For instance, if one wants a real-time traffic visualization system, then some small portion of flows might remain unrecognized without negative effect on the whole system. Thus, we conclude that our work is meaningful and can help network administrators to tune cascade TC systems better than ad-hoc tools.

For completeness, in Table 8.5 we show the results of comparing our algo-

Dataset	Algorithm	CPU time	Errors	Unknown flows
Asnet1	random	23.7	0	2,750
	optimal	7.6	0	26
		+68%	**0%**	**+99%**
Asnet2	random	34.5	26.4	10,350
	optimal	28.4	15.0	363
		+18%	**+43%**	**+96%**
IITiS1	random	28.5	0	9,203
	optimal	12.9	0	1,327
		+55%	**0%**	**+86%**
Unibs1	random	6.7	25.8	356
	optimal	1.6	20.0	267
		+77%	**+23%**	**+25%**
Average improvement:		**+55%**	**+17%**	**+77%**

Table 8.5: Experiment 4. Comparison with random cascades.

rithm with the average performance of random cascades generated in Experiment 1 (for samples of 100,000 flows). In every case, our proposal yielded better results, significantly reducing all cost factors.

8.6 Conclusions

In this chapter, we showed that the Waterfall architecture, together with an optimization technique, lets for effective combining of traffic classifiers.

We described a new optimization method that separately evaluates classification modules and quickly simulates their cascade operation in every possible configuration. By searching for the cascade that minimizes a cost function, the method finds the best configuration for given parameters: thus, one can optimize for minimum CPU time, number of errors, number of unclassified IP flows, or any combination of thereof. We experimentally validated our proposal on real-world Internet traffic datasets, demonstrating method validity, effectiveness, stability, and improvements with respect to myopic and random optimizations.

Chapter 9

Thesis Conclusions

This work presented a new method for solving the problem of Traffic Classification (TC) using cascades of classifiers—that is, the Waterfall architecture. The first part of the thesis gave background necessary to understand the context of our contribution: we described the field of TC in Chapter 2, followed by chapters presenting the fundamentals of Machine Learning (ML) (Chapter 3), datasets and tools (Chapter 4), and a survey of literature (Chapter 5). The second part opened with a presentation of an original method for classifying IP flows using the DNS system (Chapter 6). The method shows decent performance on real IP traffic, but requires augmentation with other methods in order to target all kinds of Internet traffic, and thus is an illustrative example of why the thesis is important for the field of TC. Finally, we presented the Waterfall architecture in Chapter 7, and gave a method for optimizing a classifier cascade in Chapter 8.

Thus, the thesis gave a complete discourse on applying cascade classification to the TC problem, showing that it is effective and can be automatically tuned for the desired performance. We used real Internet traffic datasets to evaluate our proposals. An illustrative cascade integrating 5 different classifiers yielded the result of $\%TP > 99.6\%$ with $\%FP < 0.1\%$ and $\%Unk < 1.4\%$ for each of 4 different datasets. Moreover, more than 50% of all evaluated IP flows were successfully classified by one of the first 3 cascade modules out of 5 total, which is faster than classifier fusion that always runs all modules.

9.1 Discussion

The goal of this work was to support the following statement:

> Cascade classification is an effective method for identifying Internet traffic. It allows for connecting different traffic classifiers together using the "divide and conquer" paradigm, and in comparison with classifier fusion, it inherently requires less computing power.

Below, we show how that statement was explained and justified in the thesis.

First, we defined and discussed the following terms in Part I: cascade classification (Chapter 3), traffic identification (Chapter 2), different traffic classifiers (Chapter 5), classifier fusion (Chapter 3). In order to accomplish that, we referenced elementary literature, reviewed state of the art papers, defined basic

concepts and procedures, and described relevant datasets and tools. Such approach was used to present the thesis contributions in a clear and concise way, with relevant context and with an appropriate level of detail.

In Section 2.2, we gave a basic formalization of the concept "identifying Internet traffic" in Equations 2.1, 2.2, and 2.3. In Chapter 7, we described a cascade classifier that is able to satisfy these equations, which was validated experimentally. Thus, we demonstrated that cascade classification is an effective method for identifying Internet traffic.

In more detail, we showed the above using 4 different datasets of 5.5 billion IP packets X (comprising over 16 million IP flows X_i total), representing a dozen applications $l \in L$. For each dataset, the evaluated cascade yielded proper outcomes $y_i = (X_i, l_i)$, with excellent performance metrics (for instance, comparing to works surveyed in Chapter 5). Moreover, in Chapter 8 we introduced an original method for optimizing a classifier cascade for desired performance goals—in terms of CPU time, number of errors, and number of unrecognized IP flows. Figure 8.3 presents the results of tuning a cascade for efficiency in 3 different scenarios, all of which gave a positive outcome.

Thus, we proved the first sentence of the thesis statement in Part II, using experimental evaluation of our proposals, conforming to established procedures for performance measurement (which were given in Part I). Yet, each contribution was presented in the context of existing literature to show method novelty and our motivation. We chose these methods as they comprise the standard research process adopted in state of the art papers (see Chapter 5).

In order to discuss the second sentence of the thesis statement, note that T. Cormen et al. describe "divide and conquer" in Chapter 4 of [37] as a three-step procedure consisting of the following steps: "divide", "conquer" (solve), and "combine". Waterfall divides Internet traffic by applications, i.e. a subproblem in TC is a subset of IP flows belonging to one or few protocols, or to several protocols, but with a common traffic feature. This was demonstrated in detail in Chapter 6, where we utilized the DNS domain names—a feature of just 30% of evaluated IP flows.

Next, the "conquer" step is a natural consequence of the previous: we deal with simpler ML tasks that require shorter feature vectors, and thus are less prone to the peaking phenomenon (see Chapter 3). Moreover, each Waterfall module can skip an unknown flow and pass it for inspection in the next module, which reduces the need to learn exceptions to a general classification rule. We gave an example for this in Chapter 7, where the `dstip` module of our illustrative cascade just checked the destination IP address in a look-up table, which was trivial yet effective.

For the last "combine" step, we adopted—and *adapted*—an MCS technique of cascade classification (see Chapter 3), which was presented in Chapter 7 and extended in Chapter 8. Because each flow X_i gets only one classification outcome y_i, we combine solutions to subproblems just by adding y_i to the set of all outcomes, defined in Equation 2.1. This is simpler compared with BKS (see Chapter 3), where one needs to wait for all modules to finish, and then to query a look-up table to combine the outcomes.

Finally, we claim cascade classification of Internet traffic is inherently faster in terms of CPU time than classifier fusion. As stated in the thesis introduction (Chapter 1), we refrained from showing this experimentally (quantitatively), as these two MCS techniques have different assumptions on their base classifiers.

Instead, in Chapter 3 we described and compared classifier fusion (on the example of BKS) with cascade classification in terms of their *qualities*. For given IP flow X_i and a fixed number of classifiers L, classifier fusion will always make L queries to all modules. On the other hand, it is apparent that a classifier cascade has the possibility to query less than L modules for the X_i label, which was illustrated in Fig. 7.2: for 50% of IP flows, the algorithm was done after 3 out of 5 modules. Moreover, the BKS technique has to wait for all classifiers to finish, and then make a final look-up in the BKS table to combine their outcomes. So, comparing just the architecture of cascade classifiers with classifier fusion systems, there is physically less work to do in the former case.

Last but not least, we do not claim superiority of any MCS technique for identifying Internet traffic over the rest, as these techniques adopt different base classifiers, so predicting their net effect in a specific TC context seems impossible.

9.2 Summary

TC is crucial for managing the Internet. It lets for on-line inspection of IP traffic without human intervention, which enables traffic shaping, intelligent routing, traffic visualization, intrusion prevention, and much more. TC systems are in the product portfolio of leading manufacturers of networking hardware, like Cisco, Juniper Networks, and PaloAlto Networks. Because of the anticipated growth of the Internet—in terms of traffic volume, application diversity, and number of connected devices—the TC problem is going to be increasingly complex.

The thesis gave a viable solution for breaking this complex task into subproblems that can be solved independently. Thus, we believe our contribution opened a new avenue for future TC systems by proposing a decent alternative to classifier fusion systems. However, we acknowledge certain limitations of this work. First, we did not investigate the issue of optimal cascade training, i.e. whether each module should be trained using the full dataset, or just the subset of flows it will see when put in a specific cascade. Second, we did not compare Waterfall with a BKS or another fusion system, comprised of suitable base classifiers. These two topics are open issues for future research.

In summary, this thesis combined and put in context a few works published in the years of 2012-2017, which for the first time brought the idea of cascading classifiers to the field of traffic classification. Practical implementation and evaluation of the proposals yielded excellent results. We presented relevant context and background knowledge, and showed how to build and tune cascade classifiers of Internet traffic in practice. The Waterfall architecture enables researchers to effectively utilize dedicated classifiers that target subproblems in TC as elements of a larger system comprised of many different classifiers.

Bibliography

[1] S. Abdelazeem. A greedy approach for building classification cascades. In *Machine Learning and Applications, 2008. ICMLA'08. Seventh International Conference on*, pages 115–120. IEEE, 2008.

[2] G. Aceto, A. Dainotti, W. de Donato, and A. Pescapé. PortLoad: Taking the best of two worlds in traffic classification. In *INFOCOM IEEE Conference on Computer Communications Workshops, 2010*, pages 1 – 5. IEEE, 2010.

[3] D. Adami, C. Callegari, S. Giordano, M. Pagano, and T. Pepe. Skype-Hunter: A real-time system for the detection and classification of Skype traffic. *International Journal of Communication Systems*, 25(3):386–403, 2012.

[4] S. Alcock. libflowmanager. Available from: `https://research.wand.net.nz/software/libflowmanager.php` [26 May 2017].

[5] S. Alcock, P. Lorier, and R. Nelson. Libtrace: a packet capture and analysis library. *ACM SIGCOMM Computer Communication Review*, 42(2):42–48, 2012.

[6] S. Alcock and R. Nelson. Libprotoident: Traffic Classification Using Lightweight Packet Inspection. Technical report, University of Waikato, 2013. `http://www.wand.net.nz/publications/lpireport`.

[7] Allot Communications. Traffic Classification products. Available from: `http://www.allot.com/products/traffic-management-and-optimization/traffic-management/` [20 January 2016].

[8] E. Alpaydin and C. Kaynak. Cascading classifiers. *Kybernetika*, 34(4):369–374, 1998.

[9] R. Alshammari and A. N. Zincir-Heywood. Machine learning based encrypted traffic classification: identifying ssh and skype. In *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, pages 1–8. IEEE, 2009.

[10] AppBrain. Number of Android applications. Available from: `https://www.appbrain.com/stats/number-of-android-apps` [15 Nov 2017].

[11] ARIN. Whois Database. Available from: `https://whois.arin.net/` [9 November 2015].

[12] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787 (Best Current Practice), Jan. 2007.

[13] R. Battiti and A. M. Colla. Democracy in neural nets: Voting schemes for classification. *Neural Networks*, 7(4):691–707, 1994.

[14] P. Bermolen, M. Mellia, M. Meo, D. Rossi, and S. Valenti. Abacus: Accurate behavioral classification of P2P-TV traffic. *Computer Networks*, 55(6):1394 – 1411, 2011.

[15] I. Bermudez, M. Mellia, M. M. Munafò, R. Keralapura, and A. Nucci. DNS to the Rescue: Discerning Content and Services in a Tangled Web. In *Proceedings of the 12th ACM SIGCOMM Conference on Internet Measurement, IMC'12*, volume 1101, page 12, 2012.

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

[17] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[18] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli. Revealing skype traffic: When randomness plays with you. *ACM SIGCOMM Computer Communication Review*, 37(4):37 – 48, 2007.

[19] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Internet Standard), Oct. 1989.

[20] Bro. Traffic Classification products. Available from: `https://www.bro.org/` [20 January 2016].

[21] T. Bujlow and V. Carela-Espanol. Comparison of Deep Packet Inspection (DPI) Tools for Traffic Classification. Technical report, Polytechnic University of Catalonia, 2013.

[22] T. Bujlow, V. Carela-Español, and P. Barlet-Ros. Independent comparison of popular DPI tools for traffic classification. *Computer Networks*, 76:75–89, 2015.

[23] T. Bujlow, V. Carela-Español, and P. Barlet-Ros. Dataset: Independent Comparison of Popular DPI Tools for Traffic Classification. Available from: `http://www.cba.upc.edu/monitoring/traffic-classification` [13 January 2017].

[24] J. But, T. Nguyen, L. Stewart, N. Williams, and G. Armitage. Performance analysis of the angel system for automated control of game traffic prioritisation. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 123–128. ACM, 2007.

[25] CAIDA. Internet Traffic Classification. Available from: `http://www.caida.org/research/traffic-analysis/classification-overview/` [27 March 2013].

[26] CAIDA. Overview of Datasets, Monitors, and Reports. Available from: http://www.caida.org/data/overview/ [13 January 2017].

[27] CAIDA. The Cooperative Association for Internet Data Analysis. Available from: http://www.caida.org/ [20 January 2016].

[28] A. Callado, C. Kamienski, G. Szabó, B. Gero, J. Kelner, S. Fernandes, and D. Sadok. A survey on internet traffic identification. *Communications Surveys & Tutorials, IEEE*, 11(3):37–52, 2009.

[29] V. Carela-Español, P. Barlet-Ros, A. Cabellos-Aparicio, and J. Solé-Pareta. Analysis of the impact of sampling on NetFlow traffic classification. *Computer Networks*, 55(5):1083–1099, 2011.

[30] V. Carela-Español, P. Barlet-Ros, M. Solé-Simó, A. Dainotti, W. de Donato, and A. Pescapé. K-dimensional trees for continuous traffic classification. *Traffic Monitoring and Analysis*, pages 141 – 154, 2010.

[31] V. Carela-Español, T. Bujlow, and P. Barlet-Ros. Is our ground-truth for traffic classification reliable? In *Passive and Active Measurement*, pages 98–108. Springer, 2014.

[32] K. Chellapilla, M. Shilman, and P. Simard. Optimally combining a cascade of classifiers. In *Proceedings of SPIE*, volume 6067, pages 207–214, 2006.

[33] Cisco. Traffic Classification products. Available from: http://www.cisco.com/c/en/us/products/security/ [20 January 2016].

[34] Cisco. Visual Networking Index: Forecast and Methodology, 2016–2021. Available from: https://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf [15 Nov 2017].

[35] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), Oct. 2004.

[36] B. Cohen. The BitTorrent Protocol Specification. Available from: http://bittorrent.org/beps/bep_0003.html [18 May 2015].

[37] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

[38] K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *The Journal of Machine Learning Research*, 2:265–292, 2002.

[39] CTorrent project. CTorrent website. Available from: http://sourceforge.net/projects/ctorrent/ [18 May 2015].

[40] A. Dainotti, A. Pescape, and K. C. Claffy. Issues and future directions in traffic classification. *Network, IEEE*, 26(1):35–40, 2012.

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

[42] T. G. Dietterich et al. Ensemble methods in machine learning. *Multiple classifier systems*, 1857:1–15, 2000.

[43] T. G. Dietterich and P. Langley. Machine learning for cognitive networks: Technology assessment and research challenges. *Cognitive Networks: Towards Self-Aware Networks*, page 97, 2007.

[44] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine learning*, 29(2):103–130, 1997.

[45] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. John Wiley & Sons, 2012.

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

[47] M. Dusi, A. Este, F. Gringoli, and L. Salgarelli. Taking a Peek at Bandwidth Usage on Encrypted Links. In *Communications (ICC), 2011 IEEE International Conference on*, pages 1 – 6. IEEE, 2011.

[48] M. Dusi, F. Gringoli, and L. Salgarelli. Quantifying the accuracy of the ground truth associated with Internet traffic traces. *Computer Networks*, 55(5):1158 – 1167, 2011.

[49] R. Ensafi, P. Winter, A. Mueen, and J. R. Crandall. Analyzing the Great Firewall of China over space and time. *Proceedings on privacy enhancing technologies*, 2015(1):61–76, 2015.

[50] F. Gringoli et al. UNIBS: Data sharing. Available from: `http://netweb.ing.unibs.it/~ntw/tools/traces/index.php` [13 January 2017].

[51] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

[52] P. Fiadino, A. Bär, and P. Casas. HTTPTag: A Flexible On-line HTTP Classification System for Operational 3G Networks. In *International Conference on Computer Communications, 2013. INFOCOM'13*. IEEE, 2013.

[53] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[54] A. Finamore, M. Mellia, M. Meo, M. M. Munafo, and D. Rossi. Experiences of internet traffic monitoring with tstat. *Network, IEEE*, 25(3):8–14, 2011.

[55] A. Finamore, M. Mellia, M. Meo, and D. Rossi. KISS: Stochastic packet inspection. *Traffic Monitoring and Analysis*, pages 117 – 125, 2009.

[56] A. Finamore, M. Mellia, M. Meo, and D. Rossi. KISS: Stochastic packet inspection classifier for udp traffic. *Networking, IEEE/ACM Transactions on*, 18(5):1505 – 1515, 2010.

[57] G. Folino, F. S. Pisani, and P. Sabatino. A distributed intrusion detection framework based on evolved specialized ensembles of classifiers. In *European Conference on the Applications of Evolutionary Computation*, pages 315–331. Springer, 2016.

[58] P. Foremski. Flowcalc: software toolkit for calculating IP flow statistics. `https://mutrics.iitis.pl/flowcalc`.

[59] P. Foremski. Multilevel Traffic Classification - project website. Available from: `http://mutrics.iitis.pl/` [17 Nov 2017].

[60] P. Foremski. MuTriCs: Automatic trace generation. Available from: `http://mutrics.iitis.pl/automatic-traffic-trace-generation` [9 February 2017].

[61] P. Foremski. Tracedump: A Novel Single Application IP Packet Sniffer. *Theoretical and Applied Informatics*, 24(1):23–31, 2012.

[62] P. Foremski. On different ways to classify Internet traffic: a short review of selected publications. *Theoretical and Applied Informatics*, 25(2):119–136, 2013.

[63] P. Foremski, C. Callegari, and M. Pagano. DNS-Class: Immediate classification of IP flows using DNS. *International Journal of Network Management*, 24(4):272–288, 2014.

[64] P. Foremski, C. Callegari, and M. Pagano. Waterfall: Rapid identification of IP flows using cascade classification. In *Communications in Computer and Information Science. Proceedings of the 21st International Conference on Computer Networks, CN2014*, volume 431, pages 14–23. Springer-Verlag, 2014.

[65] P. Foremski, C. Callegari, and M. Pagano. Waterfall traffic identification: optimizing classification cascades. In *Communications in Computer and Information Science. Proceedings of the 22nd International Conference on Computer Networks, CN2015*, volume 522. Springer-Verlag, 2015.

[66] P. Foremski, C. Callegari, and M. Pagano. Waterfall traffic classification: A quick approach to optimizing cascade classifiers. *Wireless Personal Communications*, 96(4):5467–5482, 2017.

[67] J. Frank. Artificial intelligence and intrusion detection: Current and future directions. In *Proceedings of the 17th national computer security conference*, volume 10, pages 1–12. Baltimore, USA, 1994.

[68] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[69] Gartner. Forecast: Internet of things - endpoints and associated services, worldwide, 2016. Available from: `https://www.gartner.com/doc/3558917/forecast-internet-things--endpoints` [15 Nov 2017].

[70] J. V. Gomes, P. R. Inácio, M. Pereira, M. M. Freire, and P. P. Monteiro. Detection and classification of peer-to-peer traffic: A survey. *ACM Computing Surveys (CSUR)*, 45(3):30, 2013.

[71] G. Greenwald. XKeyscore: NSA tool collects nearly everything a user does on the Internet. Available from: `http://www.theguardian.com/world/2013/jul/31/nsa-top-secret-program-online-data` [19 January 2016].

[72] L. Grimaudo, M. Mellia, E. Baralis, and R. Keralapura. SeLeCT: self-learning classifier for internet traffic. *Network and Service Management, IEEE Transactions on*, 11(2):144–157, 2014.

[73] F. Gringoli, L. Salgarelli, M. Dusi, N. Cascarano, F. Risso, and K. Claffy. Gt: Picking up the truth from the ground for internet traffic. *ACM SIGCOMM Computer Communication Review*, 39(5):13 – 18, 2009.

[74] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT Behavioral Requirements for TCP. RFC 5382 (Best Current Practice), Oct. 2008.

[75] Y. S. Huang and C. Y. Suen. A method of combining multiple experts for the recognition of unconstrained handwritten numerals. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(1):90–94, 1995.

[76] IANA. Service Name and Transport Protocol Port Number Registry. Available from: `http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml` [9 November 2015].

[77] IDC. Worldwide 2014 Software Developer and ICT-Skilled Worker Estimates. Available from: `http://www.idc.com/getdoc.jsp?containerId=244709` [14 April 2015].

[78] ipoque. Traffic Classification products. Available from: `https://ipoque.com/products/dpi-engine-rsrpace-2` [11 December 2017].

[79] A. K. Jain, R. P. Duin, and J. Mao. Statistical pattern recognition: A review. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(1):4–37, 2000.

[80] Juniper Networks. Traffic Classification products. Available from: `https://www.juniper.net/us/en/products-services/security/` [20 January 2016].

[81] T. Karagiannis, A. Broido, N. Brownlee, K. C. Claffy, and M. Faloutsos. Is P2P dying or just hiding? In *Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE*, volume 3, pages 1532–1538. IEEE, 2004.

[82] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: multilevel traffic classification in the dark. *ACM SIGCOMM Computer Communication Review*, 35(4):229–240, 2005.

[83] S. S. Keerthi, S. Sundararajan, K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. A sequential dual method for large scale multi-class linear SVMs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 408–416. ACM, 2008.

[84] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and K. Claffy. The architecture of CoralReef: an Internet traffic monitoring software suite. In *PAM2001, Workshop on Passive and Active Measurements, RIPE*. Citeseer, 2001.

[85] J. Khalife, A. Hajjar, and J. Diaz-Verdejo. A multilevel taxonomy and requirements for an optimal traffic-classification model. *International Journal of Network Management*, 24(2):101–120, 2014.

[86] H. Kim, K. C. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet traffic classification demystified: Myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT conference*, page 11. ACM, 2008.

[87] L. I. Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2004.

[88] L. I. Kuncheva. *Combining pattern classifiers: Methods and Algorithms, Second Edition*. John Wiley & Sons, 2014.

[89] Y. Lecun and C. Cortes. The MNIST database of handwritten digits. Available from: `http://yann.lecun.com/exdb/mnist/` [13 January 2017].

[90] Linux kernel. ld.so(8) manual page. Available from: `http://www.kernel.org/doc/man-pages/online/pages/man8/ld.so.8.html` [18 May 2015].

[91] Linux kernel. packet(7) manual page. Available from: `http://www.kernel.org/doc/man-pages/online/pages/man7/packet.7.html` [18 May 2015].

[92] Linux kernel. ptrace(2) manual page. Available from: `http://www.kernel.org/doc/man-pages/online/pages/man2/ptrace.2.html` [18 May 2015].

[93] 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, IMC'09*, pages 90–102. ACM, 2009.

[94] B. Marczak, J. Dalek, S. McKune, A. Senft, J. Scott-Railton, and R. Deibert. BAD TRAFFIC: Sandvine's PacketLogic Devices Used to Deploy Government Spyware in Turkey and Redirect Egyptian Users to Affiliate Ads? Available from: `https://goo.gl/zFGXMg` [9 March 2018].

[95] M. N. Marsono, M. W. El-Kharashi, and F. Gebali. Targeting spam control on middleboxes: Spam detection based on layer-3 e-mail content classification. *Computer Networks*, 53(6):835–848, 2009.

[96] MAWI. Traffic Archive. Available from: `http://mawi.wide.ad.jp/mawi/` [13 January 2017].

[97] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2. USENIX Association, 1993.

[98] T. Mitchell. The Discipline of Machine Learning. Technical report, 2006.

[99] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Internet Standard), Nov. 1987.

[100] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[101] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):50 – 60, 2005.

[102] mPlane Project. mPlane Publications. Available from: `http://www.ict-mplane.eu/public/publications` [20 January 2016].

[103] G. Münz, H. Dai, L. Braun, and G. Carle. TCP traffic classification using Markov models. *Traffic Monitoring and Analysis*, pages 127 – 140, 2010.

[104] G. Münz, S. Heckmüller, L. Braun, and G. Carle. Improving Markov-based TCP Traffic Classification. In *KiVS*, pages 61–72, 2011.

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

[106] P. Norvig. Word segmentation. In T. Segaran and J. Hammerbacher, editors, *Beautiful data: the stories behind elegant data solutions*, pages 221–227. O'Reilly Media, Incorporated, 2009. `http://norvig.com/ngrams/ch14.pdf`.

[107] M. Nottingham. Edward Snowden at IETF 93. Available from: `https://gist.github.com/mnot/382aca0b23b6bf082116` [19 January 2016].

[108] ntop. nDPI Deep Packet Inspection library. Available from: `http://www.ntop.org/products/deep-packet-inspection/ndpi/` [12 June 2017].

[109] nTop. Traffic Classification products. Available from: `http://www.ntop.org/` [11 December 2017].

[110] PaloAlto Networks. Traffic Classification products. Available from: `https://www.paloaltonetworks.com/technologies/app-id` [11 December 2017].

[111] B. Park, Y. Won, J. Chung, M.-s. Kim, and J. W.-K. Hong. Fine-grained traffic classification based on functional separation. *International Journal of Network Management*, 2013.

[112] B.-C. Park, Y. J. Won, M.-S. Kim, and J. W. Hong. Towards automated application signature generation for traffic identification. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 160–167. IEEE, 2008.

[113] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee. Mcpad: A multiple classifier system for accurate payload-based anomaly detection. *Computer networks*, 53(6):864–881, 2009.

[114] M. Pietrzyk, L. Janowski, and G. Urvoy-Keller. Toward systematic methods comparison in traffic classification. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, pages 1022–1027. IEEE, 2011.

[115] M. Pietrzyk, L. Plissonneau, G. Urvoy-Keller, and T. En-Najjary. On profiling residential customers. In *Traffic Monitoring and Analysis*, pages 1–14. Springer, 2011.

[116] Plixer. Traffic Classification products. Available from: `https://www.plixer.com/` [20 January 2016].

[117] D. Plonka and P. Barford. Flexible Traffic and Host Profiling via DNS Rendezvous. In *Proceedings of the Workshop on Securing and Trusting Internet Names, SATIN 2011*, 2011.

[118] J. Postel. Internet Protocol. RFC 791 (Internet Standard), Sept. 1981.

[119] J. Postel. Transmission Control Protocol. RFC 793 (Internet Standard), Sept. 1981.

[120] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257 – 286, 1989.

[121] G. Rogova. Combining the results of several neural network classifiers. *Neural networks*, 7(5):777–781, 1994.

[122] L. Rokach. Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography. *Computational Statistics & Data Analysis*, 53(12):4046–4072, 2009.

[123] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-service mapping for QoS: A statistical signature-based approach to IP traffic classification. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 135 – 148. ACM, 2004.

[124] L. Salgarelli, F. Gringoli, and T. Karagiannis. Comparing traffic classifiers. *ACM SIGCOMM Computer Communication Review*, 37(3):65 – 68, 2007.

[125] G. Salton, A. Wong, and C.-S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[126] M. Samcik. Ile warte sa dane o tym co lubisz? Telekom zaplaci ci za to 6 zl miesiecznie [in Polish]. Available from: `http://samcik.blox.pl/2015/04/Ile-warte-sa-dane-o-tym-co-lubisz-Telekom-zaplaci.html` [20 January 2016].

[127] P. Schneider. Tcp/ip traffic classification based on port numbers. *Division Of Applied Sciences, Cambridge, MA*, 2138, 1996.

[128] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proceedings of the 13th international conference on World Wide Web*, pages 512–521. ACM, 2004.

[129] L. Shapley and B. Grofman. Optimizing group judgmental accuracy in the presence of interdependencies. *Public Choice*, 43(3):329–343, 1984.

[130] Sikuli Project. Sikuli Website. Available from: `http://www.sikuli.org/` [9 February 2017].

[131] SolarWinds. Traffic Classification products. Available from: `https://www.solarwinds.com/netflow-traffic-analyzer` [11 December 2017].

[132] L. Stewart, G. Armitage, P. Branch, and S. Zander. An architecture for automated network control of qos over consumer broadband links. In *TENCON 2005 2005 IEEE Region 10*, pages 1–6. IEEE, 2005.

[133] Suricata IDS. Traffic Classification products. Available from: `http://suricata-ids.org/` [20 January 2016].

[134] G. Szabó, D. Orincsay, S. Malomsoky, and I. Szabó. On the validation of traffic classification algorithms. In *Passive and Active Network Measurement*, pages 72–81. Springer, 2008.

[135] G. Szabó, I. Szabó, and D. Orincsay. Accurate traffic classification. In *World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a*, pages 1–8. IEEE, 2007.

[136] Talaia. Network Visibility. Available from: `https://www.talaia.io/overview/` [13 January 2017].

[137] tcpdump project. tcpdump/libpcap webpage. Available from: `http://www.tcpdump.org/` [18 May 2015].

[138] Tstat. Traffic Classification products. Available from: `http://tstat.polito.it/` [20 January 2016].

[139] Tstat Project. Skype Traces. Available from: `http://tstat.tlc.polito.it/traces-skype.shtml` [27 March 2013].

[140] Tstat Project. Tstat Publications. Available from: `http://tstat.polito.it/publications.php` [20 January 2016].

[141] Tstat Project. Tstat Traces. Available from: `http://tstat.tlc.polito.it/traces.shtml` [9 February 2017].

[142] University of Waikato. WAND Network Research Group. Available from: `http://wand.net.nz/` [20 January 2016].

[143] University of Waikato. WAND Waikato Internet Traffic Storage. Available from: `http://wand.net.nz/wits/` [9 February 2017].

[144] UPC. CoMo-UPC: TMA evaluation service @ UPC. Available from: `http://monitoring.ccaba.upc.edu/como-upc/` [27 March 2013].

[145] S. Valenti, D. Rossi, A. Dainotti, A. Pescapè, A. Finamore, and M. Mellia. Reviewing traffic classification. In *Data Traffic Monitoring and Analysis*, pages 123–147. Springer, 2013.

[146] S. Valenti, D. Rossi, M. Meo, M. Mellia, and P. Bermolen. Accurate, fine-grained classification of P2P-TV applications by simply counting packets. *Traffic Monitoring and Analysis*, pages 84 – 92, 2009.

[147] P. Velan, M. Čermák, P. Čeleda, and M. Drašar. A survey of methods for encrypted traffic classification and analysis. *International Journal of Network Management*, 25(5):355–374, 2015.

[148] K. Velten. *Mathematical modeling and simulation: introduction for scientists and engineers*. John Wiley & Sons, 2009.

[149] K.-D. Wernecke. A coupling procedure for the discrimination of mixed data. *Biometrics*, pages 497–506, 1992.

[150] N. Williams, S. Zander, and G. Armitage. Evaluating machine learning algorithms for automated network application identification. *Center for Advanced Internet Architectures, CAIA, Technical Report B*, 60410:2006, 2006.

[151] Wireshark project. PCAP file format. Available from: `http://wiki.wireshark.org/Development/LibpcapFileFormat` [18 May 2015].

[152] Wireshark project. Wireshark website. Available from: `https://www.wireshark.org/` [18 May 2015].

[153] D. H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.

[154] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling internet backbone traffic: Behavior models and applications. *ACM SIGCOMM Computer Communication Review*, 35(4):169 – 180, 2005.

[155] S. H. Yeganeh, M. Eftekhar, Y. Ganjali, R. Keralapura, and A. Nucci. CUTE: Traffic Classification Using TErms. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, pages 1–9. IEEE, 2012.

[156] T. Yeh, T.-H. Chang, and R. C. Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192. ACM, 2009.

[157] H.-F. Yu, C.-H. Ho, Y.-C. Juan, and C.-J. Lin. Libshorttext: A library for short-text classification and analysis. Technical report, National Taiwan University, 2013. `http://www.csie.ntu.edu.tw/~cjlin/papers/libshorttext.pdf`.

[158] S. Zander, T. Nguyen, and G. Armitage. Automated traffic classification and application identification using machine learning. In *Local Computer Networks, 2005. 30Th Anniversary. The IEEE Conference on*, pages 250 – 257. IEEE, 2005.

[159] M. Zhang, W. John, K. Claffy, and N. Brownlee. State of the art in traffic classification: A research review. In *PAM Student Workshop*, 2009.