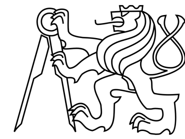
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Flow-Based Classification of Devices in Computer Networks
Student:	Zden k Kasner
Supervisor:	Ing. Tomáš ejka
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Study principles of monitoring in computer networks based on traffic flows.
Study the Nemea system [1,2] for network traffic analysis.
Study suitable semi-supervised classification methods described in [3] to recognize different types of devices that communicate over network infrastructure. A classification method should use information about network flows that are generated by observed devices.
Design a software tool for the Nemea system that can help users to classify network devices according to their network flows.
Implement the designed module.
Verify the functionality of the module with the use of a test dataset that will be provided by the supervisor.

References

[1] https://github.com/CESNET/Nemea
[2] V. Bartos, M. Zadnik, T. Cejka: "Nemea: Framework for stream-wise analysis of network traffic," CESNET technical report 6/2013.
[3] Zhang, Jun, et al. "An effective network traffic classification method with unknown flow detection." *IEEE Transactions on Network and Service Management,* 2013.

L.S.

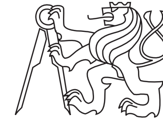
doc. Ing. Jan Janoušek, Ph.D. Head of Department	prof. Ing. Pavel Tvrdík, CSc. Dean

Prague December 1, 2015

Czech Technical University in Prague

Faculty of Information Technology

Department of theoretical computer science



Bachelor's thesis

Flow-Based Classification of Devices in Computer Networks

Zdeněk Kasner

Supervisor: Ing. Tomáš Čejka

15th May 2016

Acknowledgements

I would like to express my gratitude to my supervisor Ing. Tomáš Čejka. He continuously supported me throughout my thesis and provided me with many useful remarks and suggestions. Special thanks goes to the members of the Liberouter team for the experience I gained during the work on the project.

I would also like to thank my close friends and my girlfriend for helpful advices, motivation and patience during my work. Furthermore, I would like to sincerely thank my professors and school colleagues at FIT CTU for the enjoyable experiences during my studies.

Finally, my deep gratitude goes to my family for their continued moral support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 15th May 2016 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2016 Zdeněk Kasner. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Kasner, Zdeněk. *Flow-Based Classification of Devices in Computer Networks.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstract

This thesis deals with automatic recognition of types of devices communicating over a network. Devices in computer networks generate traffic, which can be captured as traffic flows. In my work, I have designed a method which uses traffic flows to classify types of devices. This method consists of measuring statistical properties of traffic flows and using the measured values as an input for support vector machines, an algorithm of machine learning. The main part of this work is focused on implementing this method in the form of a module for the Network Measurements Analysis (NEMEA) system, a software for network traffic analysis and anomaly detection.

Keywords classification of devices, supervised learning, support vector machines, network traffic flows, network analysis, NEMEA

Abstrakt

Tato bakalářská práce se zabývá automatickým rozpoznáváním typů zařízení komunikujících po síti. Zařízení v počítačových sítích generují provoz, který lze zachytit v podobě síťových toků. Ve své práci jsem navrhl metodu, která používá tyto síťové toky pro klasifikaci jednotlivých typů zařízení. Metoda vychází z měření statistických vlastností síťových toků a využívá naměřených hodnot jako vstup pro algoritmus strojového učení – metodu podpůrných vektorů (support vector machines). Hlavní část této práce popisuje implementaci navržené metody v podobě modulu pro systém Network Measurements Analysis (NEMEA), software pro síťovou analýzu a detekci anomálií.

Klíčová slova klasifikace zařízení, učení s učitelem, metoda podpůrných vektorů, toky v počítačových sítích, síťová analýza, NEMEA

Contents

Introduction **1**
Goals . . . 1
Structure of the Thesis . . . 2

1 Network Traffic Analysis **3**
1.1 Network Monitoring . . . 3
1.2 Traffic Flow . . . 4
1.3 NEMEA Framework . . . 5
1.4 Devices . . . 6

2 Machine Learning **9**
2.1 Introduction . . . 9
2.2 Learning Styles . . . 9
2.3 Classification Methods . . . 11
2.4 Algorithms . . . 12
2.5 Evaluation . . . 14
2.6 Classification of Network Traffic . . . 16

3 Design of the Module **17**
3.1 Basic Functionality . . . 17
3.2 Classification of Devices . . . 20
3.3 Additional Functions . . . 24
3.4 Summary . . . 25

4 Implementation **27**
4.1 Tools and Environment . . . 27
4.2 Data Structures . . . 29
4.3 Scaling . . . 32
4.4 Training Data . . . 34
4.5 Parameters . . . 35

4.6 Output 35

5 Evaluation **37**
5.1 Dynamic Program Analysis 37
5.2 Classification of Devices 39

Conclusion **41**
Future Work 41

Bibliography **43**

A Acronyms **47**

B Installation Manual **49**
B.1 Dependencies 49
B.2 Installation 49
B.3 Module Usage 50

C Figures **51**

D Contents of DVD **55**

List of Figures

1.1 NEMEA Modules Example 5
1.2 Hierarchical Structure of Devices 8

2.1 Supervised Learning 10
2.2 Unsupervised Learning 10
2.3 Semi-Supervised Learning 11
2.4 Decision Tree 13
2.5 Optimal Separating Hyperplane with a Margin 14

3.1 Example of Local Network Monitoring 18
3.2 Analyzing Stored Traffic Flows 18
3.3 Labeling Devices 23
3.4 Workflow of the Module 26

4.1 Illustration of a B+ Tree 30
4.2 Function $\tanh(x)$ Used to Scale Features 33

5.1 Performance of the Module 38

C.1 Memory Usage in a Large-Scale Network 52
C.2 Memory Usage in a Local Network 53

List of Tables

1.1 UniRec Fields of a Flow Record 6
1.2 Example of Labeled Devices 8

3.1 Features Used for Machine Learning 21
3.2 Reserved IP Addresses 25

4.1 Example of Network Traffic 30
4.2 Summary of Labels in Training Data 34
4.3 Command Line Parameters 35

5.1 Cross Validation 40

List of Listings

4.1 Feature Vector 28
4.2 Example of the File `models.list` 31
4.3 Example of a Filter 31
4.4 Example of Training Rules 32
4.5 Example of a List of Reserved IP Addresses 32
4.6 CSV Output 36
4.7 JSON Output 36
C.1 Output of Memcheck 51

Introduction

Tools for network traffic analysis are widely used in network security and administration. Analyzing anonymized network traffic helps to prevent the attacks on the network, patch up network vulnerabilities and allocate sufficient network capacity.

An important information which is missing in network monitoring is the type of monitored device and its role in a network infrastructure (for instance a personal computer (PC), a mail server, a router, etc.). This information improves network awareness – it allows to understand the structure of a network and helps to interpret ongoing changes. For instance, if a PC infected by malware starts to send spam messages, it may be no longer be detected as a client PC, but as a mail server instead.

The described background knowledge is usually not available and has to be derived from the network traffic. However, the behavior of a particular type of device may be influenced by many factors. Differences may be caused by the specific device, the size of a network, the length of network monitoring, etc. Therefore, it is not trivial to describe each type of device in the form of a few simple rules.

The solution is to measure statistical properties of network traffic. Different types of devices may produce different network traffic patterns. If an algorithm learns these patterns, it can classify unknown types of devices by observing their network traffic. The described procedure can be performed using methods of machine learning.

Goals

In this thesis, I deal with implementing a software tool which can classify different types of devices in computer networks. The tool will be used as a module in NEMEA, the system for network traffic analysis and anomaly detection. The module will extract features from the network traffic generated

by observed devices. The features will serve as training data for machine learning. Subsequently, a classifier will be used to predict types of devices.

The module can be used in various use-cases. For example, a network administrator may want to get high-level information about devices on the network that is not known to him yet. The idea is to launch the module on the network to get information about devices by passively observing the network traffic. If the administrator needs to know about ongoing changes on the network, the module can help to track changes of devices automatically.

Furthermore, the module can be used in network security. For instance, it can reveal the use of services that allow remote access for attackers (e.g. trojans, backdoors) by tracking changes in the behavior of devices.

Finally, the module may also serve as a basis for other network analysis modules in the NEMEA system.

Structure of the Thesis

The thesis is divided into five chapters. Chapter 1 provides an overview of network traffic analysis and discusses the taxonomy of devices communicating in computer networks. Chapter 2 presents machine learning styles, methods of classification and algorithms. The main part of my work is described in chapters 3 and 4. The design of the module and the method of classifying devices is described in Chapter 3. The details of implementation of the module can be found in Chapter 4. Finally, Chapter 5 concludes the results of the work and evaluates the performance of the module.

Chapter 1

Network Traffic Analysis

This chapter provides an introduction to network traffic analysis. Current practices in network monitoring are described. The text is focused on the aspects of flow-based network monitoring. At the end of the chapter, possible taxonomies of network devices are discussed in detail.

1.1 Network Monitoring

Collecting and analyzing anonymized traffic data can help network providers to deal with several issues. In the area of network security, it allows to discover sources of attacks and protect weak points of the network. In network administration, it allows to control whether users are using the permitted services only and not exceeding their data limits. All of this has to be done with respect to user privacy.

There are several approaches to monitoring of present-day networks. The approaches are further described in this section. The description is inspired by the document on best practices in network monitoring [1].

1.1.1 Packet Inspection

The first approach is based on inspecting packet contents. A *packet* is a basic unit of data in network traffic. It consists of a header with control information and payload with user data. There are tools which allow to capture and analyze the packets, e.g. the tools with a command line interface as tcpdump[2] or with a graphical interface as Wireshark[3].

However, the amount of network traffic flowing through monitoring probes in larger networks can be enormous. As reported in [4], on the backbone line Prague – Brno in the academic network CESNET2, the average packet rate varies around 600,000 packets per second. This volume of data would be difficult to process with the present-day computational power. Therefore, packet inspection is usually limited to local networks.

1.1.2 Statistical Properties

The document [1] further describes another approach which is to collect and analyze *statistics* describing network behavior. These statistics can be gathered with a various level of detail depending on which particular kind of information is needed. This mostly involves monitoring the status of key network components, e.g. the value of Simple Network Management Protocol (SNMP) counters at network interfaces. This approach is quite simple, but the data is too aggregated to perform detailed network analysis.

1.1.3 Flow-Based Monitoring

Flow-based monitoring combines the advantages and drawbacks of both of the previously mentioned approaches. In flow-based network monitoring, the payload is removed and the packet headers are aggregated in entities called *traffic flows* (or “flows”). For each flow, various statistics are measured, such as the number of packets and bytes, the time of the start (or the end) of the flow, Transmission Control Protocol (TCP) flags and more.

The traditional approach to data analysis in flow-based network monitoring involves storing flow data on a hard drive. The data is subsequently analyzed at regular intervals. The document [1] suggests the NfSen collector as an example of a typical publicly available collector for flow-based network traffic analysis.

1.2 Traffic Flow

In general, a traffic flow is a sequence of packets sent from a source to a destination. In this work, the definition by Y. Wang et al. [5] is used:

- *Unidirectional flow* (or flow) is a series of packets sharing the same 5-tuple consisting of source and destination Internet Protocol (IP) addresses, source and destination port numbers and the transport layer protocol.

- *Bidirectional flow* (or biflow) is a pair of unidirectional flows using the same transport layer protocol and going in the opposite directions between the same source and destination IP addresses and ports.

With the level of abstraction provided by flows, storing and processing large amount of data is avoided. At the same time, the information needed for further network analysis is preserved. It also means that all the payload data in the captured network traffic is discarded automatically and cannot be used to identify users.

1.3 NEMEA Framework

NEMEA is an open-source system for automated flow-based network analysis. It is developed at CESNET a.l.e. by the research team Liberouter. The system is designed to monitor networks with high traffic throughput. In contrast with traditional approaches, it uses stream-wise flow processing. This approach allows continuous and real-time traffic analysis and alert reporting without any additional data storage. Principles and inner functionality of the system are described in detail in the technical report [6].

The NEMEA system consists of separate building blocks called modules. Modules communicate with each other through interfaces. The usual workflow of a module is to receive a stream of data at its input interface, process it and send another stream of data to its output interface. Each module is launched as a separate process in the operating system. Therefore, it can be controlled independently on other modules and use system resources on its own. Modules can be assembled to create a more complex system.

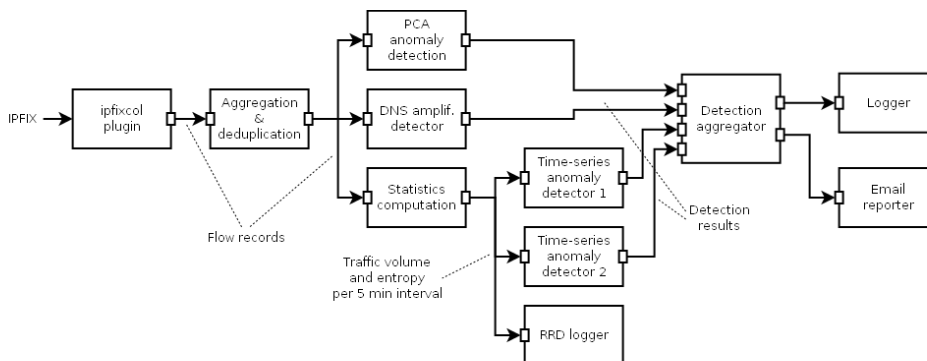


Figure 1.1: Example of a more complex assembly of NEMEA modules. Source: the Liberouter website [7].

The core of the NEMEA system is the NEMEA framework. It offers an interface for communication between the modules, a standard format of messages and an implementation of useful common functions and data structures.

1.3.1 TRAP

For communication, NEMEA modules use an Application Programming Interface (API) supplied by Traffic Analysis Platform (TRAP). TRAP is represented by a dynamic library called *libtrap*, which is linked by every NEMEA module. It helps to abstract modules from the communication layer. *libtrap* takes care of low-level objects and functions like buffering, (non-)blocking send operation and sockets. It also provides modules with standard input and output communication interfaces.

1.3.2 UniRec Data Format

UniRec (Unified Record) is a format of messages used between NEMEA modules. It is similar to a C structure, but it provides more flexibility. The UniRec format allows to create new message structures dynamically while preserving memory and access time efficiency. This allows modules to exchange a very large volume of various types of data.

Each UniRec record consists of several fields that can be static or dynamic. The set of fields used by a module is called a template. The template of the receiving module has to be equal or subset of the template of the sending module.

1.3.3 Flow Records

A NEMEA module dealing with network analysis needs to process incoming flow records. Using TRAP, this job is simplified to processing UniRec fields defined in the template.

The usual template with information about a flow is listed in Table 1.1. This set of fields directly implies which features can be utilized as the input for machine learning methods.

Table 1.1: UniRec Fields of a flow record

Data Type	**Field Name**	**Description**
`ipaddr`	`SRC_IP`	Source address of a flow
`ipaddr`	`DST_IP`	Destination address of a flow
`uint16`	`SRC_PORT`	Source transport-layer port
`uint16`	`DST_PORT`	Destination transport-layer port
`uint8`	`PROTOCOL`	L4 protocol (TCP, UDP, ICMP, etc.)
`uint32`	`PACKETS`	Number of packets in a flow or in an interval
`uint64`	`BYTES`	Number of bytes in a flow or in an interval
`time`	`TIME_FIRST`	Timestamp of the first packet of a flow
`time`	`TIME_LAST`	Timestamp of the last packet of a flow
`uint8`	`TCP_FLAGS`	TCP flags of a flow (logical OR over TCP flags field of all packets)

1.4 Devices

This section addresses the taxonomy of network devices. The text introduces the definition of a network device and discusses the ways to systematize their types.

1.4.1 Definition

The term *network device* is used as a synonym to a basic element which form the network topology (a repeater, a switch, a router, etc.) [8], [9]. This definition can be useful for designing and maintaining the network infrastructure.

However, a different kind of definition is needed for this work. In network traffic analysis, a network device is a source of network traffic. This can be an endpoint of the network, as for instance a client PC, or a network element such as a router (in special cases, e.g. using the Routing Information Protocol (RIP)). In traffic flows, a source of network traffic is represented by an IP address. To conclude, the *network device* (or a device) in this work is a source of network traffic represented by an IP address.

1.4.2 Taxonomy

To my best knowledge, there is no existing suitable taxonomy of network devices. Therefore, it is necessary to define it. The taxonomy should define the possible labels of devices and the relation between them. Choosing the appropriate taxonomy is important for selecting an appropriate method of machine learning.

Hierarchical Structure A hierarchical structure offers a simple and comprehensible way to separate devices into classes. An example is shown in Figure 1.2. For instance, if a network device is classified as *File Transfer Protocol (FTP) server*, it also belongs to the classes *file server* and *server*. If the device cannot be classified to this extent (e.g. because of the lack of training data for this particular class), it can be classified into one of the parent classes.

There are some disadvantages in using this approach. Due to the static tree structure, a device cannot belong to the *client* class and the *access point* class at the same time. However, this may be a valid and useful information in certain situations. By imposing specific requirements, the hierarchical structure also limits the range of available machine learning methods.

Flat Structure A more flexible structure can be achieved by making all the classes equal. In this structure, each class is only a label assigned to the device. A device can be assigned none, one or more labels at the same time. In this case, multi-label classification can be used to classify devices.

An example is shown in Table 1.2. The matrix can be defined as follows. Let S be a set of all labels, $|S| = N$. Each device is assigned a set of labels S' where $S' \subseteq S$ and $|S'| \in \{0, 1, \ldots, N\}$. In the matrix, "1" is used to indicate that the label in this column belongs to the subset S' of this device, "0" otherwise.

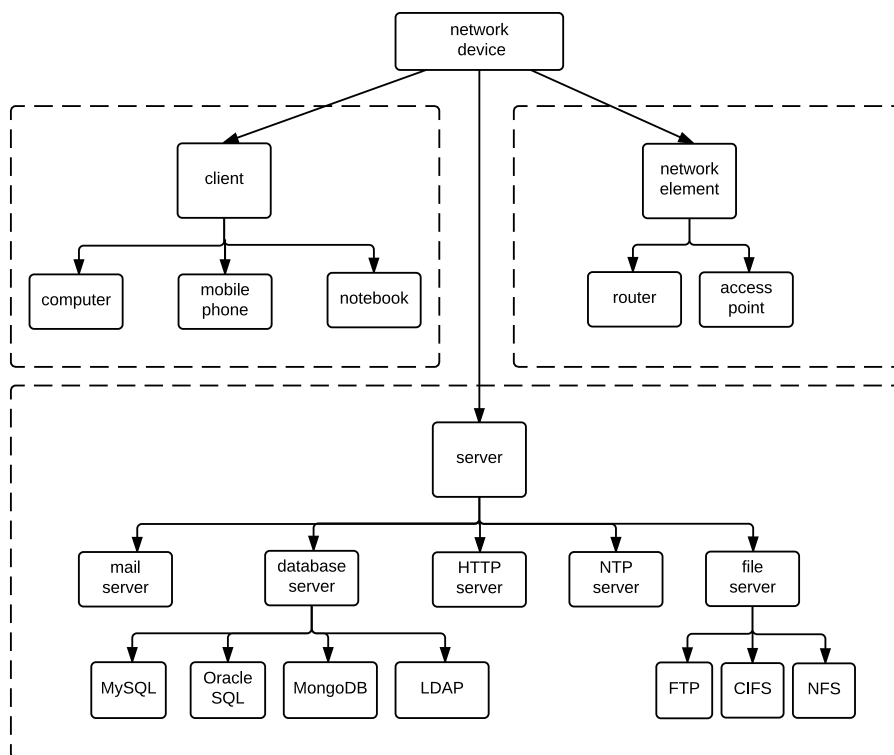


Figure 1.2: Hierarchical structure of devices

Table 1.2: Example of labeled devices. Each columns is a label. “1” signalizes the presence of a label, “0” otherwise.

	SERVER	CLIENT	MAIL	HTTP	FTP
Device A	1	0	0	1	0
Device B	0	1	1	1	0
Device C	1	0	0	0	1
Device D	1	0	1	0	0

Since current devices are usually multifunctional, this approach is a better approximation of reality. For this reason, this approach has been chosen in this work.

Chapter 2

Machine Learning

This chapter provides a summary of present-day machine learning styles and methods. Emphasis is put on the methods that can be used for classification of devices.

2.1 Introduction

A common way of making a computer to *do something* (i.e. transform data, interact with a user, compute results) involves explicit programming. In this case, the set of instructions for the computer is a step-by-step procedure, directly implying how to get the desired outcome. However, there are problems that are too complex to be dealt with in this manner. This is the point where **machine learning** takes place.

Machine learning can be viewed as "programming by example". The algorithms of machine learning are learning the patterns by observing the data (induction) and using the gained knowledge to make predictions (deduction). This process resembles closely the process of learning in human mind.

According to [10], *"the art of machine learning is to reduce a range of fairly disparate problems to a set of fairly narrow prototypes."* From the perspective of device classification, there is no sense in defining a set of rules for each type of device. Instead, the methods of machine learning can provide a model that is abstract and generally applicable.

2.2 Learning Styles

Traditionally, there are two fundamentally different styles of machine learning – *supervised* and *unsupervised* learning. They differ both in the structure of input data and the desired outcome. There is also another category called *semi-supervised* learning, which combines both previously mentioned

approaches. Although it is not fundamentally different from the others, it is mentioned by certain sources ([11], [12]) as one of the machine learning styles.

2.2.1 Supervised Learning

Supervised learning uses labeled training data made of pairs (x_i, y_i) where each pair represents an instance and its value. The goal of supervised learning is to learn the *mapping* from x to y. If values of y are taken from a finite set, the task is called **classification**.

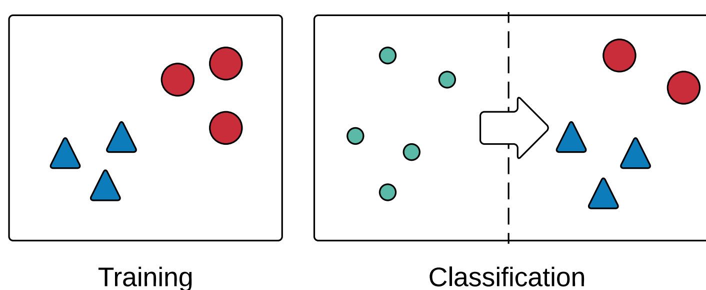


Figure 2.1: Supervised learning

2.2.2 Unsupervised Learning

The goal of unsupervised learning is to find interesting structures in the given data. In other words, the algorithms either try to *separate* the data into clusters or to infer the underlying structure (e.g. its statistical distribution). Because the structure is usually not known in advance, it is more difficult to evaluate the performance of a model.

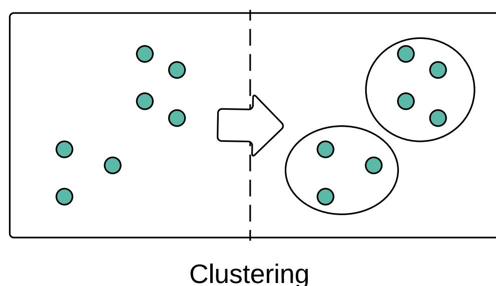


Figure 2.2: Unsupervised learning

2.2.3 Semi-Supervised Learning

Semi-supervised learning is trying to learn the mapping of x to y as well as supervised learning. The difference is that the training dataset of semi-supervised learning can also consist of **unlabeled data**. In this case, a specific assumption about the data has to be made. The article [12] suggests a smoothness assumption: *if two points in the training dataset are close, they are more likely to share a label.* With this assumption, unlabeled data can help the classification. Semi-supervised learning may be useful in case there is not enough labeled training data.

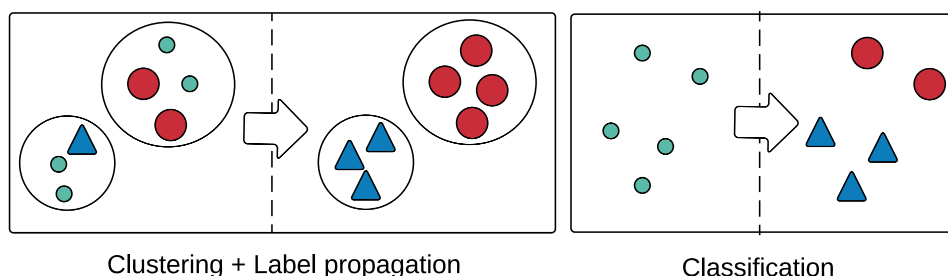


Figure 2.3: Semi-supervised learning

2.3 Classification Methods

Classification is the process of associating instances with labels. It can be considered as a specific task of supervised learning. Training data for classification consists of a set of instances, each of them associated with a class label. The training data is analyzed to construct a classifier which is able to classify unseen instances.

Classification methods can be divided into two basic categories defined in [13].

2.3.1 Single-Label Classification

Single-label classification is a common variant of the classification problem. During the classification process, each instance is assigned a single label l from the set of labels L, $|L| > 1$.

If $|L| = 2$, then the learning problem is called a *binary classification problem*. This problem can be solved e.g. by support vector machines (Section 2.4.4). If $|L| > 2$, then it is called a *multi-class classification problem*, which can be handled e.g. by decision trees (Section 2.4.3).

2.3.2 Multi-Label Classification

In multi-label classification, each example is associated with a set of labels $Y \subseteq L$. Table 1.2 can be considered as an example of a multi-label dataset.

The article [13] describes various *problem transformations* which can be used to solve this problem using the existing single-label methods:

1. **discard all** the multi-label instances from the multi-label data set
2. **select one** of the multiple labels of each multi-label instance and **discard** the rest
3. consider each different set of labels that exists in the multi-label data set as **a single label**
4. learn $|L|$ **binary classifiers** $H_l : X \to \{l, \neg l\}$, one for each different label l in L.

2.4 Algorithms

This section presents several supervised machine learning algorithms (selected from [14]) that represent different approaches to supervised machine learning. The text summarizes principles and aspects of these algorithms. All the algorithms can be used for multi-label classification after an appropriate problem transformation.

2.4.1 k-Nearest Neighbors

k-Nearest Neighbors (kNN) is an example of an **instance-based** learning algorithm. The class of an unknown instance is determined as the most frequent class of its *k* nearest neighbors in the training dataset. A *distance metric* (such as the Euclidean distance) is used to find the nearest neighbors. For each unlabeled instance, a set of the nearest neighbors has to be computed separately. That implies that the classification time is *asymptotically dependent* on the number of instances.

2.4.2 Naive Bayes

The simplest of the family of **statistical** learning algorithms is the *Naive Bayes classifier*. Naive Bayes classifier computes the probability of each class given a feature vector and selects the class with the largest probability. The term *naive* is derived from an assumption of the strong independence between the features (which is almost never satisfied in practice, therefore “naive”). Nevertheless, with appropriate preprocessing, Naive Bayes can perform well even in real-world situations, as described in [15].

2.4.3 Decision Trees

Decision trees belong to the group of **logic-based** algorithms. The algorithm uses *a decision tree* in order to classify an instance. The advantage of decision trees is a simple and human readable visual representation of the predictive model. Each node of a tree symbolizes a conditional split on the value of one of the attributes, starting with the most significant attribute in the root. Therefore, the path from the root to the leaf is exactly determined for each instance by the values of its attributes. Finally, a leaf represents the target class of the instance.

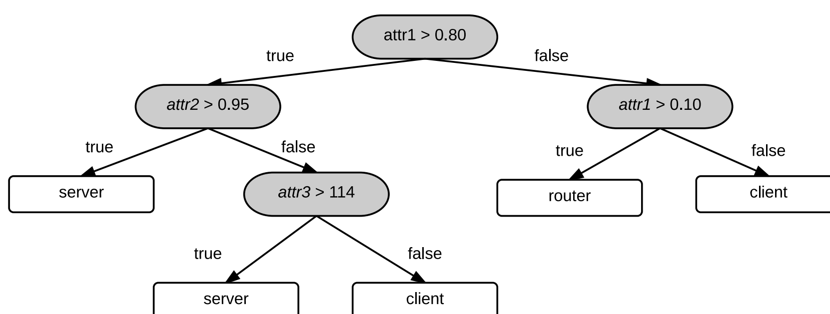


Figure 2.4: Decision tree example

2.4.4 Support Vector Machines

One of the most recent supervised learning methods is the *support vector machine* (SVM). The goal of SVM is to linearly separate the instances. Instances with n features can be seen as points in n-dimensional space. SVM is trying to find the optimal **separating hyperplane** between those points. An example of a hyperplane is a *line* in the two-dimensional space.

In general, there may be many hyperplanes separating the data. However, SVM is trying to maximize the margin of a hyperplane. The margin can be computed as twice the distance between a hyperplane and the nearest data point in the hyperspace (see Figure 2.5). This ensures that the hyperplane is optimal (i.e. with the greatest distance from all the points) and therefore generalizes well with unseen data. The SVM classifiers are always *binary*, as the hyperplane always separates the space into exactly two semi-spaces.

Linear SVM can be defined as follows. If the training data is linearly separable, then a pair $(\mathbf{w}, b)$, where $\mathbf{w}$ is termed the weight vector and b the bias (or $-b$ is termed the threshold), exists such that

$$\mathbf{w}^T\mathbf{x}_i + b \geq 1 \text{ , for all } \mathbf{x}_i \in P \tag{2.1}$$

$$\mathbf{w}^T\mathbf{x}_i + b \leq -1 \text{ , for all } \mathbf{x}_i \in N \tag{2.2}$$

with the decision rule given by

$$f_{w,b}(x) = \operatorname{sgn}(\mathbf{w}^T\mathbf{x} + b) \tag{2.3}$$

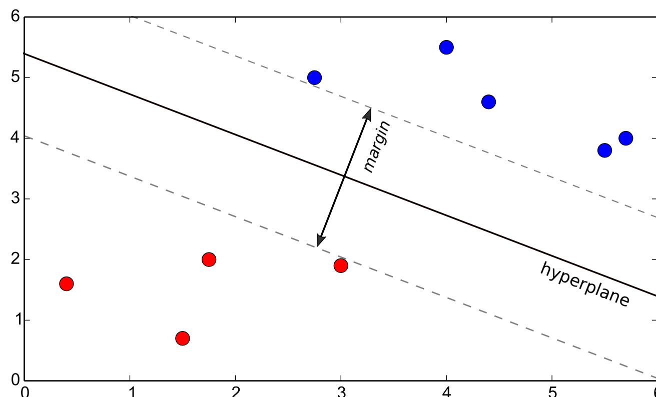


Figure 2.5: Optimal separating hyperplane with a margin in 2D space

2.5 Evaluation

Performance of a classifier can be evaluated in various ways. This section sums up the metrics which are useful for this work – metrics for binary classification, extension of these metrics into the multi-label environment and a technique for evaluating a classifier known as *cross validation*.

2.5.1 Binary Classification Metrics

Two common metrics are suggested in [16] for measuring the classification performance of binary classifiers: overall accuracy and F-measure.

- **Overall accuracy** is the ratio of the sum of all correctly classified instances to the sum of all testing instances.

  $$\text{Accuracy} = \frac{\text{number of correctly classified instances}}{\text{number of testing instances}}$$

  This metric is used to measure the accuracy of a classifier on the whole testing data.

- **F-measure** is calculated by

  $$\text{F-measure} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

where precision is the ratio of correctly classified instances over all predicted instances in a class and recall is the ratio of correctly classified instances over all instances belonging in a class. F-measure is used to evaluate the per-class performance.

2.5.2 Multi-Label Classification Metrics

Performance evaluation in multi-label classification is more complicated than in traditional single-label setting, as each example can be associated with multiple labels simultaneously. The sources [17] and [18] suggest several metrics that can be used to evaluate binary metrics used for multi-label classification:

- **Exact Match Ratio** evaluates the fraction of correctly classified examples. Example is classified correctly only if the predicted label set is identical to the ground-truth label set. It is a multi-label counterpart of the overall accuracy metric.

- **Macro Averaging** calculates the mean of a chosen binary metric for each label. Each label contributes to the result with equal weight, therefore this approach may highlight the performance of less frequent labels.

- **Micro Averaging** on the opposite calculates the chosen binary metric only once, considering the results for each pair sample-label. Micro-averaging may be preferred in multi-label settings, as the less frequent labels have lower impact on the result.

2.5.3 Cross Validation

If a single dataset is used for both training and testing, the best result would give a classifier which would learn the rules based on all the attributes of each instance in the dataset. This classifier would be overfitting and not generalizing well on unseen instances. As suggested in [19], a solution to this problem is a procedure called cross validation (CV). In the basic approach, called k-fold CV, the training set is split into k smaller sets. The following procedure is followed for each of the k folds:

1. A classifier is trained using $k - 1$ of the folds as training data.

2. The resulting classifier is validated on the remaining part of the data.

The performance measure reported by k-fold CV is the **average** of the values computed in the loop.

2.6 Classification of Network Traffic

The state-of-the-art methods for classifying network traffic focus on the level of individual *traffic flows*. These methods try to associate each flow with a service (web browser, Skype, torrent, FTP, etc.). As described in [5], the traditional methods rely on the well-known and registered port lists maintained by Internet Assigned Numbers Authority (IANA). Recently, there have also been studies focusing on statistic-based classification methods. The survey [20] summarizes some of the methods. These methods try to learn typical characteristics of different classes of applications using machine learning.

To my best knowledge, there is no publication dealing with classification of devices as the source of multiple services, as described in this thesis. The concept of classifying devices is more abstract, although to a certain degree it may overlap with classifying services. Similar methods can be used in both cases, only in case of classifying devices, the features are aggregated for a single IP address.

Chapter 3

Design of the Module

In this chapter, the design of the module is proposed. First of all, the text deals with receiving and processing traffic flows. Furthermore, classification of devices is analyzed in more detail. The focus is put on designing an appropriate technique of classification and choosing a suitable machine learning algorithm. Several additional features are also introduced, including filtering the IP addresses and training new labels. Finally, the overall workflow of the module is concluded and illustrated in the summary.

3.1 Basic Functionality

The module requires only a minimum of user interaction in order to generate the output. The basic processing is designed as automatic – a user only has to launch the module and provide input data, i.e. traffic flows. The basic functionality consists of several consecutive steps: receiving flows, processing flows, classifying devices and producing an output.

3.1.1 Input

The module is integrated into the NEMEA system, which implies two basic scenarios of the module deployment.

The first option is to process network traffic in **real time**. In this case, network traffic is being sent to the NEMEA system. This can be done for instance by duplicating the network traffic in a local network (as depicted in Figure 3.1). For capturing network traffic, a NEMEA module `flow_meter` can be used. Another possibility is to receive the data from a collector. In both cases, flows are processed online – the only difference may be the volume of incoming network traffic, which is usually much higher from the collector.

The other option is to process captured traffic flows stored in a **file**. This can be useful for analyzing network traffic retrospectively. The most simple case, in which the data is already in a comma-separated values (CSV) file

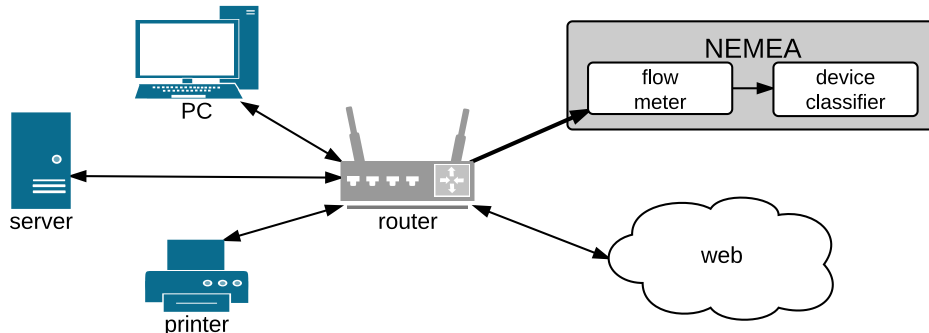


Figure 3.1: Example of local network monitoring

compatible with the UniRec format, is shown in Figure 3.2. The NEMEA module `logreplay` can be used for replaying stored traffic flows. In this case, computational power is the most limiting factor, as the flows are replayed immediately.

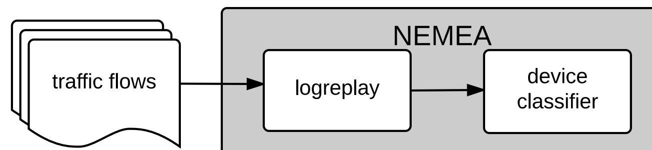


Figure 3.2: Assembly of modules used for analyzing stored traffic flows

3.1.2 Processing Flows

After a flow record is received by the module, the UniRec fields (listed in Table 1.1) are extracted and processed. In this case, processing a flow means updating values which are computed for each IP address. The flow itself is discarded immediately after processing. As a consequence, the module should be memory efficient even with a large volume of network traffic data.

An overview of values that are extracted from flows can be found below. The values are updated individually for each IP address.

- **peers** – the number of distinct IP addresses which the device has interacted with
- **flows** – the number of flows related to the device[1]

[1]The device is a source or a destination of the flow.

- **biflows** – the number of biflows (i.e. flows in opposite directions captured within a defined time period) in which the device has been the *source* of one of the flows in the pair and the *destination* of the other one

- **initiated biflows** – the number of biflows in which the device has been the source of the *first flow* in the pair

- **packets** – the number of packets in the flows related to the device

- **time** – the length of the flows related to the device

- **protocol** – the number of the flows related to the device using a TCP protocol

- **ports** – the number of the flows related to the device using a particular port (computed for each port individually)

3.1.3 Classification and Output

Once the flow processing is finished (or interrupted), output may be generated. It should be emphasized that during the processing only statistical values are computed – no labels are known yet. Therefore, generating output is a two-step process. In order to get a result, devices have to be **classified** first. The process of classification is described in more detail in Section 3.2. Following are the various cases in which the classification is performed:

- A time interval has been specified by the user. Devices are classified regularly within this interval.

- The main loop is interrupted, i.e. the process is terminated by the system. Devices are classified unless a time interval has been specified.

- The signal `SIGUSR1` is sent to the module. This allows to classify devices at any moment during runtime without interrupting the module.

- An end-of-file (`EOF`) message is received at the input interface. This occurs when the end of the file with stored flows is reached.

As soon as labels of devices are classified, results are **printed** and optionally **exported**. This allows the results to be further processed by the user or another program. An example of an output of the module is shown in Section 4.6

3.2 Classification of Devices

The main task of the module is to classify devices, i.e. to associate IP addresses with labels. A device is described by a set of statistical values. This section deals with the problem of transforming the statistical values to labels. More specifically, it presents extracting relevant features, selecting the appropriate machine learning algorithm and designing a suitable method of classification.

3.2.1 Feature Extraction

There are many statistical values that can be extracted from network traffic. However, some of them do not describe the type of the device as well as the others – they are less informative, or conversely redundant. This is why feature extraction is important for device classification.

Feature extraction is a process of deriving new values from initial data. The set of derived values (a *feature vector*) is designed to be more suitable for prediction. Feature extraction can help to reduce the dimensionality of a dataset and to reduce the noise in data. This may result in faster classification process and better performance. As discussed in [21], it is a matter of judgment to decide which values should be derived. A decision is usually based on the background knowledge of the problem.

The features selected for the feature vector are listed in Table 3.1. All the features are based on the statistical values which were collected during the flow processing. The features were designed in order to reduce the correlation between their values and the length of network monitoring (i.e. the runtime of the module).

3.2.2 Selection of Learning Style

As soon as the features are computed, the feature vector can be used as an input for machine learning. In order to design a particular method suitable for this case, one of the machine learning styles from Section 2.2 has to be selected in the first place.

The nature of *unsupervised* learning is not suitable for classification. This approach fundamentally does not use labeled data. It can only help to find structures in the data which itself is not enough to describe a behavior of devices.

Under certain circumstances, *semi-supervised* learning could be suitable for device classification. Although there is a large volume of unlabeled traffic data, there are not enough labeled instances which could be used for training. This is the problem that semi-supervised learning is trying to deal with. In the articles [5] and [16], semi-supervised learning is suggested for classification of network traffic flows.

Table 3.1: Values used as the input features for machine learning

Feature	**Description**	**Count**
Init Ratio	Number of initialized biflows relative to the number of biflows in total. Describes how often the device started the communication.	1
Data Ratio	Number of bytes transferred in source flows relative to the number of transferred bytes in total. Describes the proportion of data in source traffic.	1
Packets/Flow	Expected value and standard deviation of packets per source / destination flow.	4
Time/Flow	Expected value and standard deviation of seconds per source / destination flow.	4
TCP Ratio	Number of flows using TCP protocol relative to the total number of flows.	1
Port Usage	Relative usage of a port in proportion to the total traffic. Features represent individual well-known ports and registered ports according to [22]. Usage of dynamic ports is aggregated into a single feature.	49152
Total		49163

However, there are several major disadvantages in using semi-supervised learning for device classification:

1. Using a complex approach as semi-supervised learning could deteriorate the performance of the module in large networks.

2. The final number of device classes is not known in advance, so the best number of clusters would have to be determined at first.

3. The article [16] deals with classification of individual flows. However, in device classification, an instance is a set of multiple flows. As a consequence, the suggested techniques as *label propagation* between correlated flows cannot be used.

On the contrary, *supervised learning* offers a variety of algorithms. Summary of the supervised learning algorithms [14] suggests that they are usually

straightforward, scalable and applicable in different cases. This is why I decided to use a single supervised learning algorithm for classification of devices in my work.

3.2.3 Selection of Algorithm

Several supervised learning algorithms were listed in Section 2.4. I selected **support vector machines** (SVM) as the most suitable algorithm for my work. Following is a summary of advantages and disadvantages of this algorithm, based on [14].

Advantages of SVM

- SVM are well suited to deal with learning tasks where the number of features is large with respect to the number of training instances.
- The training optimization problem of the SVM necessarily reaches a global minimum.
- Unlike Naive Bayes, SVM makes no assumption of the underlying distribution and therefore can fit complex data more readily.
- An open-source library LIBSVM (described in Section 4.1.2) provides an implementation of support vector machines for many programming languages including C/C++.

Disadvantages of SVM

- SVM methods produce only binary classifiers. This is not a problem when using binary multi-label classification as described later.
- Results of some other algorithms, as for example the decision trees, may be better visualized.

3.2.4 Selection of Classification Method

The system of labeling devices introduced in Section 1.4.2 implies the use of multi-label classification. Multi-label classification allows to associate an instance with more labels at once. Existing single-label algorithms, including the SVM, can be used for multi-label classification after an appropriate problem transformation.

The most common problem transformations are described in Section 2.3.2. From these options, the transformation (4) is the only one that can be used in the design of the module. The previous transformations either change the dataset irreversibly (1, 2) or allow to use only the subsets of labels that were present in a training dataset (3).

The transformation (4) requires creating a **set of binary classifiers**. This is a particularly suitable task for SVM, as SVM classifiers are binary. Each of the classifiers is trained to classify a different label. During the classification process, each of the classifiers is given a feature vector as an input. The output of each classifier determines the presence or absence of the corresponding label. The set of all present labels is considered as a final output for each device. The process is depicted in Figure 3.3.

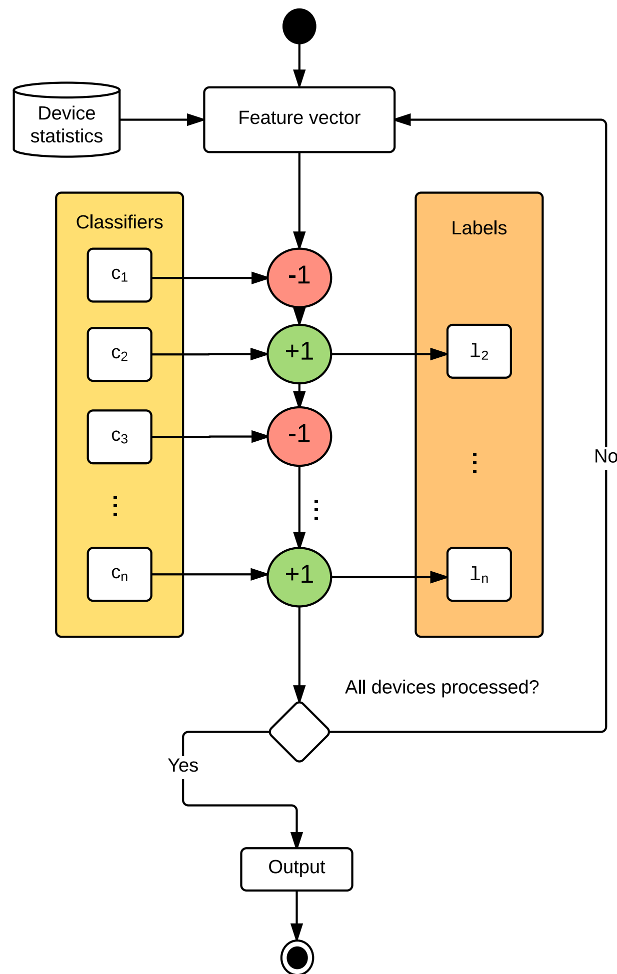


Figure 3.3: Multi-label classification method for labeling devices using a set of binary classifiers. Each binary classifier determines the presence or absence of a single label. The set of present labels is the resulting label set of an IP address.

3.3 Additional Functions

The module is equipped with several additional functions which extend its capabilities. The functions include filtering IP addresses, learning from new training data and recognizing reserved IP addresses.

3.3.1 Filtering

It is often desirable to track only a subset of devices. In a local network, for example, it may be necessary to track only devices from this network – in other words, to track IP addresses within a given subnet. Or, for checking an individual IP address, it may be a waste of computational power to process the results of all devices. These cases can be solved by *filters*.

A filter is a list of IP addresses which should be tracked. As described in Section 3.1.2, statistical values of source and destination IP address are updated after a flow is received. If a filter is specified, the rules are checked before updating the values of each IP addresses. If the IP address is not tracked, its values are not updated.

There is a module `unirecfilter` in the NEMEA system that is able to filter flows. However, in this case this module cannot be used, as it always sends a complete flow and cannot tell if one of the IP addresses (source or destination) should not be tracked. Therefore, filtering is implemented as a standalone function in the module.

3.3.2 Training

The module is equipped with a certain set of classifiers. Nevertheless, the set can never encompass the entire range of all possible labels. There is a need to train classifiers for new labels. Moreover, even the existing classifiers can be made more accurate with additional training data. These two issues can be solved by the ability of the module to learn.

In order to learn, the module can be launched in the *training mode*. In this case, the user provides a set of IP addresses associated with labels. These IP addresses are tracked in the network traffic and their features are computed in the same way as in normal mode (i.e. when the training mode is not turned on). However, in this case *generating the output* means storing the computed features of tracked IP addresses.

The training phase begins when the processing is done. The collected data is appended to the existing training data. With this ensemble, entirely new classifiers are constructed for each label. These classifiers are used the next time the module is launched in normal mode.

3.3.3 Reserved IP Addresses

There are certain IP addresses which should serve a specific purpose in a network. These IP addresses are standardized in Request For Comments (RFC) 6890 [23]. An example may be the IP address `255.255.255.255` used for broadcast.

It is trivial to label these IP addresses manually, but it would require additional effort to classify them. Due to this, the module uses the predefined list of reserved IP addresses. If any of these IP addresses is encountered, it is not classified, but automatically associated with the defined label.

Table 3.2 summarizes the IP addresses recognized by the module. These IP addresses were selected from the IPv4 Special-Purpose Address Registry by IANA [24].

Table 3.2: List of reserved IP addresses recognized by the module

0.0.0.0/8	BROADCAST-CURRENT
127.0.0.0/8	LOOPBACK
169.254.0.0/16	LINK-LOCAL
192.31.196.0/24	AS112-V4
192.52.193.0/24	AMT
192.88.99.0/24	6TO4-RELAY-ANYCAST
192.175.48.0/24	AS112-NAMESERVER
198.18.0.0/15	BENCHMARK
198.51.100.0/24	DOC-TEST-NET-2
203.0.113.0/24	DOC-TEST-NET-3
255.255.255.255/32	BROADCAST

3.4 Summary

The output of the module are IP addresses associated with labels. In order to generate this output, the module computes statistical properties of the network traffic generated by each IP address. From these statistical properties, features describing the behavior of devices are extracted and provided as the input for trained SVM classifiers. The output is the set of all positive outputs of these classifiers.

The module is also capable of filtering IP addresses, training new classifiers, improving the existing classifiers and using predefined rules for reserved IP addresses.

The workflow of the module is illustrated in Figure 3.4.

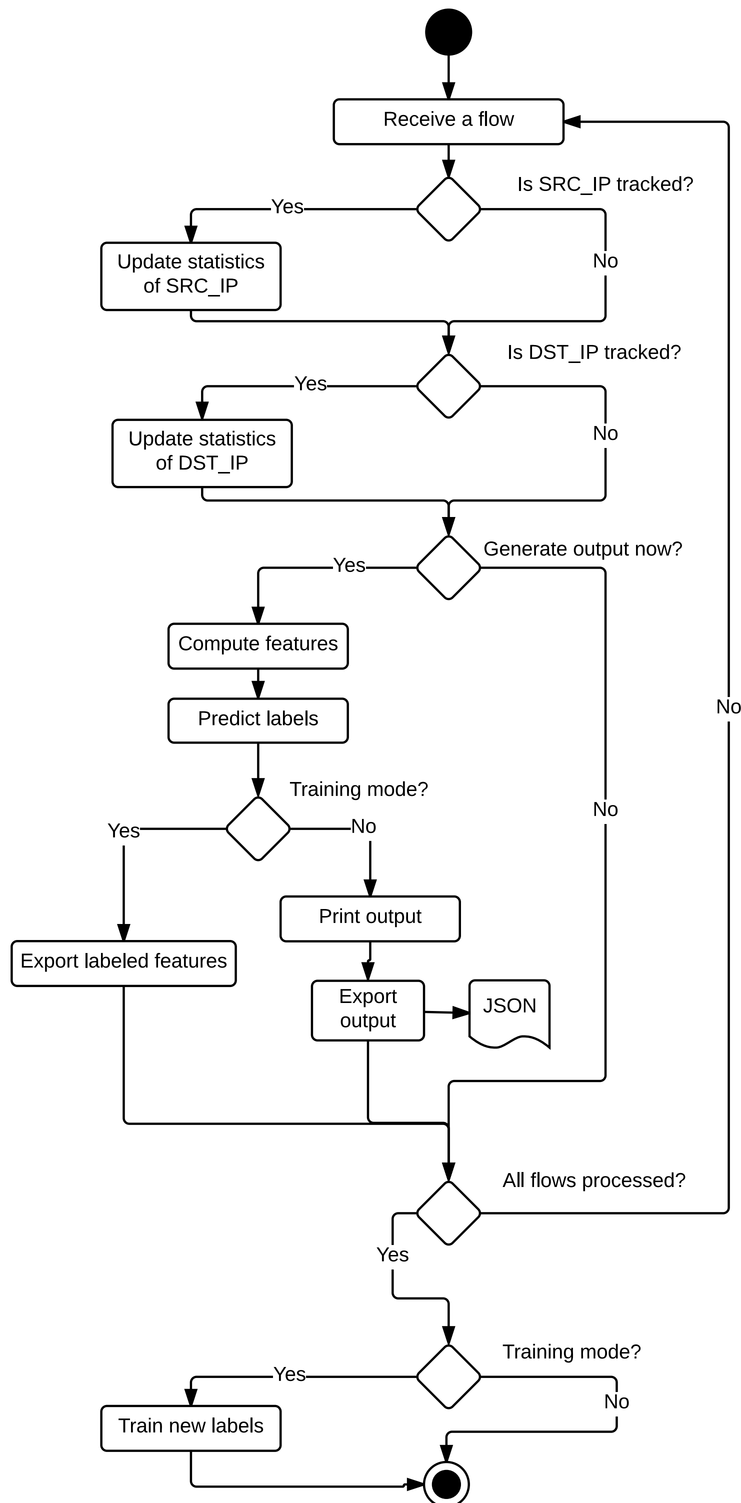


Figure 3.4: Workflow of the module

Chapter **4**

Implementation

The chapter describes the details of the implementation of the module. It contains the description of the tools, environment and data structures used by the module. Furthermore, the specific data transformations needed for correct classification are explained. The output and the parameters of the module are presented at the end of the chapter.

4.1 Tools and Environment

This section presents the environment in which the module operates and tools used during the development of the module.

4.1.1 Programming Languages

The module is implemented in C programming language. Its efficiency and performance allows to process even very large volume of network traffic in real time. In the NEMEA system, C is a standard language in the modules analyzing the network flows.

For classification, the module uses the LIBSVM library. The core of LIBSVM implemented in a C/C++, additional tools are implemented in Python. A Bourne Again Shell (Bash) script is used for the procedure of training. All described languages are supported in the NEMEA system.

4.1.2 LIBSVM

LIBSVM is an open-source implementation of support vector machines. It is an integrated software offering a wide range options for support vector training and classification. LIBSVM provides interfaces for many programming languages including C/C++. Alternatively, the binaries of LIBSVM can be run as an executable. According to [25], LIBSVM is currently one of the most widely used SVM software.

Feature Vectors Feature vectors are the input of LIBSVM. Feature vectors in LIBSVM use a **sparse form**, which means that each feature is explicitly encoded as a pair `index:value`, where `index` is a fixed index of the feature and `value` is a real number expressing its value. This allows to leave out zero-valued features. An approach like this helps to save memory in case there are many possible features, but only a small fraction of them is usually in a vector. For example, a typical device uses only a few ports over a period of time. In a sparse vector, only the usage of those ports have to be stored.

Training data requires feature vectors associated with labels. A label is a single integer which precedes the features. An example of a labeled sparse feature vector is in Listing 4.1

Listing 4.1: Feature vector with a label

```
0 1:0.363636 2:0.0150914 3:0.867685 4:0.999114 5:0.972717
   6:0.999996 7:0.0136355 8:0.0159502 9:0.0136355
   10:0.0152047 11:1 477:0.991483 599:0.266161
```

Models For classification, LIBSVM uses classifiers called *models*. A model is a file created during training. It contains the support vectors for the classification problem. Support vectors are the points describing a hyperplane which separates data in two semi-spaces – one for the presence and one for the absence of the current label. Support vectors are encoded in the sparse form in the same way as feature vectors.

Interface The C interface of LIBSVM is available directly through header file `svm.h`. The function for classification takes two arguments: an existing SVM model and an array of `svm_nodes`. The array represents a feature vector. The function returns a double which for binary classification is either $+1$ or -1 (i.e. a positive or negative prediction).

Kernel LIBSVM uses a function called *kernel* which maps training vectors into a higher dimensional space. This helps to solve the optimization problem even in cases the data are not linearly separable. There are four available kernels in LIBSVM – linear, polynomial, radial basis function (RBF) and sigmoid (all of them are further described in [26]). The recommended option is the RBF kernel, which I also utilized in my work.

There are two parameters for an RBF kernel: C and γ. It is not known what values of C and γ are best for a given problem. To find an approximation of these parameters, LIBSVM offers a Python script `tools/grid.py` which does a "grid-search" of these parameters. The implementation is described in more detail in [26]. In my work, I use this script to approximate the parameters C and γ for each model.

Multi-Label Classification LIBSVM provides a way to perform a multi-label classification using a binary approach described in Section 3.2.4. For each label, a binary-class problem is built so that feature vectors associated with the label are in one class and the rest are in another class. In the input data, multiple labels are separated by commas.

The implementation of training and evaluation for multi-label classification is provided by the Python script `tools/binary.py`. I made several modifications to the script which is used by the module. The modified script stores all the individual models and does not perform evaluation immediately after training.

Training Training is performed in order to create the models. Although the interface for training is available in the header file `svm.h`, I decided to use the scripts `tools/binary.py` (multi-label classification) and `tools/grid.py` (heuristic search for C and γ parameters of RBF kernel), which use executable files of LIBSVM.

4.1.3 Software Tools

Other software tools used in the project include the following:

- The module uses the GNU Build System [27], also known as Autotools, for building the project and managing the dependencies.
- The project is versioned using the version control system Git [28]. The source code of the project is in a private Git repository hosted on the servers of CESNET.
- The `README` file of the module is using the Markdown markup language [29].

4.2 Data Structures

The module should be able to process a large amount of data. It is thus important to choose appropriate data structures in order to achieve high performance.

4.2.1 Statistics

The statistical values for each IP address are stored in a C structure. As the number of distinct IP addresses in the network traffic can be very large, it is essential to work with the structures effectively. The module often performs the following tasks:

1. find the statistics of a given IP address
2. insert a new IP address
3. iterate through ordered IP addresses

B+ tree is a suitable data structure for these operations. The asymptotic complexity of finding or inserting the item in the B+ tree is $O(log_m n)$, where m is the maximal number of children per node and n is the total number of elements in the tree. The operation of finding next element has complexity $O(1)$ and iterating through n elements $O(n)$.

For the module, I used the implementation of B+ tree provided the NEMEA system. The implementation is described in detail in the Bachelor's thesis by Zdeněk Rosa [30].

To give an example, lets assume that traffic flows between IP addresses listed in Table 4.1 have been captured.

Table 4.1: Example of network traffic

index	source	destination
0	14.254.210.80	114.254.157.26
1	14.254.210.80	76.19.167.217
2	74.170.217.201	147.184.148.239
3	76.19.167.217	14.254.210.80
4	14.254.210.80	76.19.167.217
5	14.254.210.80	73.105.16.163

Accordingly, the B+ tree illustrated in Figure 4.1 is created. The degree m is set to 4.

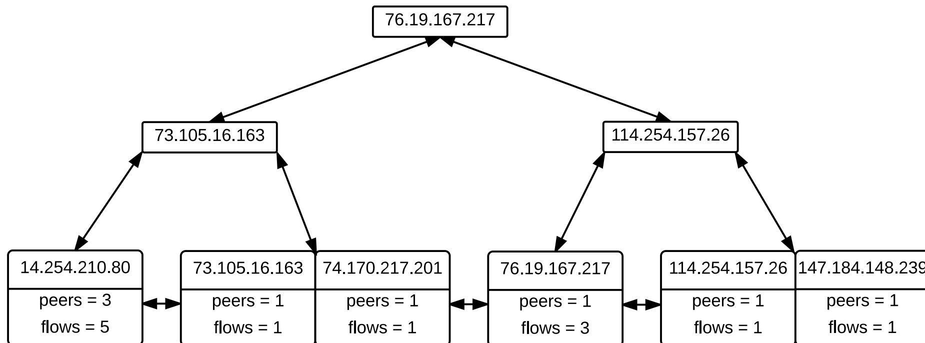


Figure 4.1: Illustration of a B+ tree which is created after the traffic between IP addresses in Table 4.1 is processed. The structures are only illustrative.

4.2.2 Ports

Usage of ports is tracked for each device. Since a device typically uses only a small fraction of the port range, the ports are stored in a **linked list**, which helps to save the memory.

The linked list is not sorted during the processing, thus the complexity of finding or inserting a port is $O(n)$. This is not a problem as the absolute number of used ports of each device is very small (according to [31], over 99 % addresses use less then 50 destination ports). After the processing is done, the values are sorted, scaled and used in a feature vector.

4.2.3 Models

Classifiers in LIBSVM are called models. A model for each label is constructed during training. All the existing models are stored in the folder `./libsvm/models/`. This folder also contains the file `models.list`, which lists all the models. In this file, each model is described by the pair `id:label`, where `id` represents its *identifier* in training data (and the name of the file with the model) and `label` describes the *type of device* associated with the model.

An example of the content of a file `models.list` is shown in Listing 4.2. The file is used by the module to load the existing models.

Listing 4.2: Example of the content of a file `models.list`

```
0:SERVER
1:CLIENT
2:HTTP
3:FILE
4:MAIL
5:NTP
6:SIP
```

4.2.4 Filter

A filter is a file containing a list of IP addresses which should be tracked. Each line contains a single IP address or a subnet in the format `IP_ADDRESS[/MASK]`, where the mask is optional. An example of a filter is illustrated in Listing 4.3.

Listing 4.3: Example of the content of a file with filter. Each line contains a tracked IP address.

```
77.75.78.48
124.1.2.83
192.168.1.0/24
```

4.2.5 Training

When the processing is finished and the training mode is turned on, training data is automatically exported. The module immediately asks for launching a Bash script `train.sh`. This script appends the training data to the existing dataset and launches the script `tools/binary.py`, which creates binary classifiers from the new dataset. The script `train.sh` can be also launched manually by the user.

Training rules describe the label(s) associated with IP addresses in training data. Each line of file with training rules is in the format `IP_ADDRESS[/MASK] LABEL_1,...,LABEL_N`. Listing 4.4 contains an example of training rules.

Listing 4.4: Example of the content of a file with training rules. Each line contains a rule for a single IP address/subnet.

```
216.58.214.238 SERVER,HTTP
77.75.79.53 SERVER,HTTP,FILE
195.113.144.245 SERVER,SIP
10.0.0.0/8 CLIENT
```

4.2.6 Reserved IP Addresses

Reserved IP addresses are listed in the file `models.list`. Each reserved IP address has the notation `@IP_ADDRESS[/MASK]:LABEL`. An example is illustrated in Listing 4.5.

Listing 4.5: Example of a list of reserved IP addresses. The IP addresses are listed in the file `models.list`

```
@0.0.0.0/8:BROADCAST-CURRENT
@127.0.0.0/8:LOOPBACK
@169.254.0.0/16:LINK-LOCAL
@255.255.255.255/32:BROADCAST
```

4.3 Scaling

According to the guide [26], data scaling is important in order to achieve good classification results with SVM. It prevents attributes in greater numeric ranges from dominating those in smaller numeric ranges. It also helps to avoid numerical difficulties during the calculation.

The guide proposes to scale each feature to the range $[-1, +1]$ or $[0, 1]$. As all the features used in this work are defined as nonnegative, the features are scaled by the module to the range $[0, 1]$.

A simple scaling method is provided by the LIBSVM library. The method is given by the following formula:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{4.1}$$

where x is an original value and x' is the normalized value.

This scaling method has been used by the module at first. However, it turned out that it has several disadvantages:

- Values of $\min(x)$ and $\max(x)$ are not known in advance for some features. For example, the range of *packets/flow* in the training dataset may differ significantly from the range of this value in real world data. Therefore, the actual range after scaling can be greater than $[0, 1]$.
- A scaling factor – $\min(x)$ and $\max(x)$ of training data – has to be calculated and stored for each feature during training.
- This linear scaling does not reflect the nonlinear nature of certain features.

These disadvantages can be solved by using a function whose range of values is between $[0, 1]$. An example is a hyperbolic tangent function $\tanh(x)$. This function reflects the fact that lower values have more impact on a result than higher values.

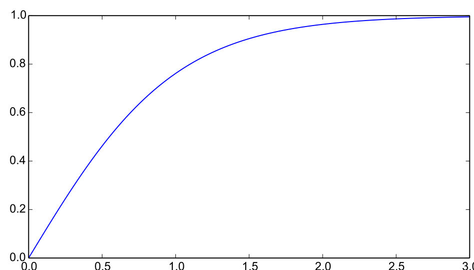


Figure 4.2: Function $\tanh(x)$ used to scale features

I concluded that the best approach is to split the designed features (listed in Table 3.1) into two groups:

1. *Init Ratio*, *Data Ratio*

   No scaling is needed for these features. Their scale is linear and their range is already between $[0, 1]$.

2. *Packets/Flow*, *Time/Flow*, *TCP Ratio*, *Port Usage*

   Scaled by the function $\tanh(ax)$, where a is a constant and x is the value. The constant a is used to increase the variance of scaled values (a has been empirically selected as 3 for Port Usage and $\frac{1}{100}$ for the rest of the features).

4.4 Training Data

In order to create some models, the module has been trained with samples of network traffic. The network traffic of specific types of devices has been acquired in two ways:

- I used the service *Shodan.io* [32], a search engine for Internet-connected devices. This engine is capable of finding the IP addresses of specific types of devices. Subsequently, my supervisor extracted network traffic of these IP addresses which has been captured by the monitoring probes in the network CESNET2 on 18/02/2016. I processed the network traffic to extract features which has been used for training data. The network traffic has been discarded after processing.

- The network traffic of clients connected to a local network has been used for training data. The network is described later in Section 5.1.2.

Table 4.2 summarizes the labels retrieved from the training data. The traffic has been sampled to get more training instances (an instance is calculated from the network traffic of a single device over a time period). There are total 443 instances in training data. Some devices and instances account for multiple labels.

Table 4.2: Summary of labels in training data. *Id* is the index of the model in training data.

Id	**Label**	**Devices**	**Instances**
0	SERVER	17	254
1	CLIENT	22	187
2	HTTP	12	156
3	FTP	2	35
4	MAIL	4	57
5	NTP	4	55
6	SIP	1	25
7	DNS	1	15
8	RDP	1	2
9	SSH	2	12

4.5 Parameters

The module takes parameters listed in Table 4.3. These parameters allow users to use all the functions of the module and to set additional options.

Table 4.3: Command line parameters of the module

Short	Long	Description
`-a`	`--accumulate`	Do not discard statistics after each output.
`-f`	`--filter`	Process only IP addresses (or subnets) specified in file given as parameter.
`-F`	`--file`	Save results to file in JSON format.
`-l`	`--list`	List existing models and exit.
`-m`	`--minutes`	Period after which the results are computed (default is 0 = at the end of stream).
`-p`	`--peers`	Classify only devices with equal or greater number of peers.
`-t`	`--train`	Run in the training mode. Training rules are in file given as parameter.

4.6 Output

The module provides only a basic output − pairs of IP addresses and their labels. Although there are other statistics that can be displayed for each device, they are out of the scope of the module.

The output can be generated in two forms:

- **CSV format**

  An output in CSV format is printed by default to standard output. The CSV format is two-column and tab-separated. An example of this output is shown in Listing 4.6.

- **JSON format**

  A output can be optionally exported into a file in JSON format. A device is represented as a JSON object, a single output is represented as a JSON array, all the outputs are aggregated in a parent JSON array. An example of this output is shown in Listing 4.7.

Listing 4.6: Example of default output of the module in CSV format

```
ip_address       labels
66.189.174.146   CLIENT
76.241.17.159    SERVER
92.155.236.218   SERVER,MAIL
107.41.42.63     SERVER,MAIL
128.135.183.34   SERVER,HTTP
140.231.116.196  SERVER,HTTP,FTP
145.22.147.241   SERVER,HTTP
```

Listing 4.7: Example of output of the module in JSON format

```
[[{
    "ip" : "66.189.174.146",
    "labels" : ["CLIENT"]
},
{
    "ip" : "76.241.17.159",
    "labels" : ["SERVER"]
},
{
    "ip" : "92.155.236.218",
    "labels" : ["SERVER","MAIL"]
},
{
    "ip" : "107.41.42.63",
    "labels" : ["SERVER","MAIL"]
},
{
    "ip" : "128.135.183.34",
    "labels" : ["SERVER","HTTP"]
},
{
    "ip" : "140.231.116.196",
    "labels" : ["SERVER","HTTP","FTP"]
},
{
    "ip" : "145.22.147.241",
    "labels" : ["SERVER","HTTP"]
}]]
```

Chapter 5

Evaluation

This chapter evaluates the results of the implemented module. It includes evaluation of the performance and the memory efficiency of the module, followed by the accuracy of classification of devices. All the tests in this chapter have been done on GNU/Linux distribution using a computer with 2.3 GHz Intel Core i3-2348M processor and 4 GB of system memory.

5.1 Dynamic Program Analysis

The module has been tested by the Valgrind [33] framework. The Valgrind package includes several debugging and profiling tools for dynamic analysis. From those tools, Memcheck has been used for memory management and Massif for heap profiling. For visualization of the results, I used the graphing utility *Gnuplot* [34] and the application for graphing Massif output *Massif-Visualizer* [35].

5.1.1 Memory Management

Possible memory leaks in the module have been checked using Memcheck with parameters `--leak-check=full` and `--show-leak-kinds=all`. The output of Valgrind is listed in Listing C.1. The testing involved running the module with different parameters and input data. All the leaks in the module have been fixed immediately during the development.

The testing have also revealed a memory leak which originates from the used LIBSVM library. It accounts for 72 kB of reachable memory once allocated after the module is started. I have reported this minor fault to the authors of the LIBSVM library.

5.1.2 Performance and Heap Usage

Large-Scale Network

The performance of module should be sufficient even for a large volume of network traffic. For this reason, I have tested the performance and memory usage of the module using the dataset `collected-records-tcp-nix2-card1.ipfix`. This dataset of size 383.9 MB contains 3,687,313 flows collected on 15/02/2016 by a monitoring probe in the network CESNET2. The file has been converted from Internet Protocol Flow Information Export (IPFIX) to CSV format.

Figure 5.1 shows dependence of performance of the module on the number of stored IP addresses. The performance is measured in processed flows per second. The module has been launched with default parameters, therefore, it has been processing all the network traffic without any filtering. In this case, the performance depends mainly on the number of elements in the of the B+ tree used for storing the information about individual IP addresses.

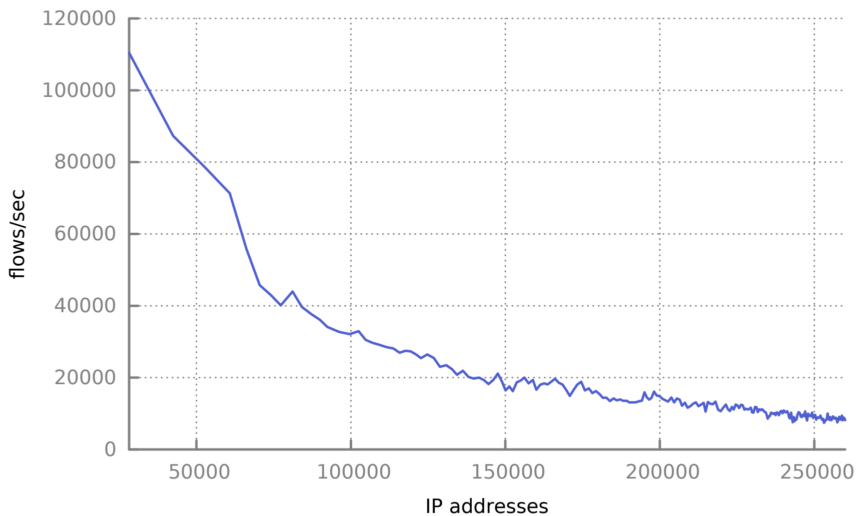


Figure 5.1: Dependence of performance of the module on number of stored IP addresses

The heap memory used by the module also depends primarily on the number of stored IP addresses. The memory usage is illustrated in Figure C.1. During the testing, the final number of 265,834 distinct IP addresses accounted for 528.5 MB of used heap memory. For the processing part, this memory is allocated mostly for the statistical values of IP addresses, additional memory is required for computed feature vectors in the classification part. After the classification is done, the memory is freed.

It is worth noting that the case above serves primarily as a benchmark. In the more practical use-case, a user would set a filter for specific IP addresses

or subnets. If the amount of the tracked IP addresses in the network traffic is far lower than the total number of IP addresses, the performance of the module is approximately constant, as only the statistical values of *tracked* IP addresses are stored in the memory.

Local Network

The module has also been tested with local network traffic. The dataset for this purpose has been captured at the university in a local network created for the experiment. The network has been set up according to Figure 3.1. From 18/01/2016 to 23/02/2016, all the traffic flows in the network have been captured in a file. The file of size 214.7 MB contains 1,711,848 flows generated by local devices. The file has been used as the dataset for the testing. All the subjects which have connected to the network agreed with processing their network traffic.

The experiment traced the heap consumption with the local network dataset. The module has been launched with parameter `-m 360`, which caused generating the output every 6 hours. The results are illustrated in Figure C.2. Most of the memory used up by the module is allocated by underlying TRAP library. The peaks of memory allocated by the module are not significant and the memory is freed regularly after generating the output.

5.2 Classification of Devices

Classification accuracy of the classifier has been tested on the **training dataset** described in Section 4.4. A custom version of CV has been used for the evaluation. The process below is described in Algorithm 1.

Training dataset has been split randomly into 10 equally-sized chunks for 10-fold CV. For each iteration of CV, a set of binary classifiers have been created. Each of these binary classifiers has been evaluated using Exact match ratio and F-measure. All the values of Exact match ratio have been averaged, all the values of F-measure have been microaveraged and macroaveraged. Finally, all the 10 results have been averaged to get the final results.

The results of the CV are listed in Table 5.1. The average Exact match ratio is 97.50 %, microaverage F-measure 99.04 % and macroaverage F-measure 78.95 %.

In real-world environment, usability of the module does not depend only on the classifier, but on the training dataset, too. At this moment, the dataset of the module contains only a basic set of labels. However, even this dataset may be sufficient for intended use-cases. It should also be highlighted that the dataset can be extended with more data samples.

Algorithm 1 Evaluation of classification accuracy

```
 1: function Cross_validation
 2:     n ← 10                                              ▷ number of folds
 3:     T ← n equally-sized chunks of training dataset
 4:     L ← labels in T
 5:     for i ← 1, n do
 6:         D_train ← (⋃_{j=1}^{n} T[j]) \ T[i]             ▷ training dataset
 7:         D_test ← T[i]                                   ▷ testing dataset
 8:         for each l ∈ L do
 9:             find optimal parameters C and γ
10:             M ← train_model(D_train, l, C, γ)
11:             p ← test_model(D_test, l)                   ▷ results of prediction
12:             em_ratio ← Exact match ratio of p
13:             f_measure ← F-measure of p
14:         end for
15:         E ← average of em_ratio for all labels
16:         A_micro ← micro average f_measure for all labels
17:         A_macro ← macro average f_measure for all labels
18:     end for
19:     Res ← average of all E, A_micro, A_macro
20: end function
```

Table 5.1: Results of CV for the training dataset. *Index* is the number of iteration of CV. The last line contains the average of results from all the 10 iterations.

Index	Exact Match R.	Microavg. F-m.	Macroavg. F-m.
1	100.00 %	100.00 %	80.00 %
2	100.00 %	100.00 %	80.00 %
3	95.45 %	98.64 %	76.00 %
4	97.72 %	99.35 %	80.00 %
5	97.72 %	99.35 %	99.23 %
6	97.72 %	98.76 %	84.66 %
7	95.45 %	98.85 %	87.46 %
8	97.72 %	98.59 %	59.52 %
9	93.18 %	96.89 %	72.65 %
10	100.00 %	100.00 %	70.00 %
Avg.	97.50 %	99.04 %	78.95 %

Conclusion

My task was to design and implement a method to recognize different types of network devices. This text has described the process of development of the module for classification of devices.

My research consisted of two parts. In the first part, I have dealt with the principles of network monitoring. During my collaboration with CESNET, I have contributed to the NEMEA system. This allowed me to gain a lot of experience in the field of network traffic analysis. In the second part, I have studied methods and algorithms of machine learning. Furthermore, I have become familiar with the state-of-the-art in classification of traffic flows.

In the analytical part, I have designed a method which can associate IP addresses with labels. The method is based on computing features from the traffic flows and using them as an input for machine learning. I selected SVM as a suitable machine learning algorithm. I have also described how to use SVM for multi-label classification.

Subsequently, I have created a module for the NEMEA system which implements the designed method. I used the service *Shodan.io* to find network traffic of real devices for the training dataset. I have evaluated the performance of the module on network traffic captured at both large-scale and local networks. I have also tested the classification accuracy of the module on the training dataset using the 10-fold CV.

The module is ready to be used in various use-cases. Network administrators can get an overview about devices on the network by passively observing network traffic. The module can track the ongoing changes on the network. Finally, the output of the module may be also valuable for detecting suspicious behavior of devices.

Future Work

All the tasks in my assignment have been fulfilled. I will now focus on deploying the module to the public Github repository of the NEMEA system.

There are other opportunities to improve the current state of the module. The module should be tested with real-time network traffic. The number of labels recognized by the module and the classification accuracy can be increased using more network traffic samples. There is a possibility to experiment with a different set of features measured from the network traffic. Furthermore, the module should be extended to support IPv6 addresses or to aggregate information about devices with multiple IP addresses.

Bibliography

[1] Žádník, M. Network Monitoring Based on IP Data Flows. Best Practice Document GN3-NA3-T4-CBPD131, March 2010.

[2] The Tcpdump Group. Tcpdump. http://www.tcpdump.org/, accessed: 2016-04-02.

[3] The Wireshark team. Wireshark. https://www.wireshark.org/, accessed: 2016-04-02.

[4] CESNET2: IP/MPLS backbone utilization. http://netreport.cesnet.cz/netreport/CESNET2_IP_MPLS_backbone_utilization/Praha%20-%3E%20Brno/index.html, accessed: 2016-04-02.

[5] Wang, Y.; Xiang, Y.; et al. A novel semi-supervised approach for network traffic clustering. In *Network and System Security (NSS), 2011 5th International Conference on*, IEEE, 2011, pp. 169–175.

[6] Bartos, V.; Zadnik, M.; et al. Nemea: Framework for stream-wise analysis of network traffic. *CESNET, ale, Tech. Rep*, 2013.

[7] The Nemea project. https://www.liberouter.org/technologies/nemea/, accessed: 2016-05-12.

[8] McQuerry, S.; Thomas, M. *Interconnecting Cisco Network Devices*. Cisco Press, 2000.

[9] Network Devices. http://computernetworkingnotes.com/comptia-n-plus-study-guide/network-devices-hub-switch-router.html, accessed: 2016-05-13.

[10] Smola, A.; Vishwanathan, S. Introduction to machine learning. *Cambridge University*, 2008: pp. 32–34.

[11] Jason Brownlee. A Tour of Machine Learning Algorithms. http://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/, November 2013, accessed: 2016-04-07.

[12] Chapelle, O.; Schölkopf, B.; et al. Semi-supervised learning. *MIT press Cambridge*, 2006.

[13] Tsoumakas, G.; Katakis, I. Multi-label classification: An overview. *Dept. of Informatics, Aristotle University of Thessaloniki, Greece*, 2006.

[14] Kotsiantis, S. B.; Zaharakis, I.; et al. Supervised machine learning: A review of classification techniques. In *Informatica 31 (2007)*, 2007, pp. 249–268.

[15] Rennie, J. D.; Shih, L.; et al. Tackling the poor assumptions of naive bayes text classifiers. In *ICML*, volume 3, 2003, pp. 616–623.

[16] Zhang, J.; Chen, C.; et al. An effective network traffic classification method with unknown flow detection. *Network and Service Management, IEEE Transactions on*, volume 10, no. 2, 2013: pp. 133–147.

[17] Zhang, M.-L.; Zhou, Z.-H. A review on multi-label learning algorithms. *Knowledge and Data Engineering, IEEE Transactions on*, volume 26, no. 8, 2014: pp. 1819–1837.

[18] Model evaluation: quantifying the quality of predictions. http://scikit-learn.org/stable/modules/model_evaluation.html#from-binary-to-multiclass-and-multilabel, accessed: 2016-05-10.

[19] Cross-validation: evaluating estimator performance. http://scikit-learn.org/stable/modules/cross_validation.html, accessed: 2016-05-10.

[20] Nguyen, T. T.; Armitage, G. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys & Tutorials, IEEE*, volume 10, no. 4, 2008: pp. 56–76.

[21] Pyle, D. *Data preparation for data mining*, volume 1. Morgan Kaufmann, 1999.

[22] M. Cotton et. al. Service Name and Port Number Procedures. RFC 6335, August 2011. Available from: https://tools.ietf.org/html/rfc6335

[23] M. Cotton et. al. Special-Purpose IP Address Registries. RFC 6890, April 2013. Available from: http://tools.ietf.org/html/rfc6890

[24] IANA. IPv4 Special-Purpose Address Registry. http://www.iana.org/assignments/iana-ipv4-special-registry/iana-ipv4-special-registry.xhtml, accessed: 2016-04-27.

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

[26] Hsu, C.-W.; Chang, C.-C.; et al. A practical guide to support vector classification. 2003.

[27] An Introduction to the Autotools. `http://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html`, accessed: 2016-05-05.

[28] Git. `https://git-scm.com/`, accessed: 2016-05-05.

[29] Gruber, J. Markdown. `https://daringfireball.net/projects/markdown/`, accessed: 2016-05-05.

[30] Rosa, Z. *Detekce síťových tunelů v počítačových sítích*. Bachelor's thesis, České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.

[31] Cejka, T.; Svepes, M. Analysis of Vertical Scans Discovered by Naive Detection. In *Autonomous Infrastructure, Management and Security (AIMS2016)*, 2016.

[32] Shodan. `http://shodan.io/`, accessed: 2016-05-09.

[33] Valgrind. `http://valgrind.org/`, accessed: 2016-05-08.

[34] Gnuplot. `http://www.gnuplot.info/`, accessed: 2016-05-09.

[35] Wolff, M. Massif-Visualizer. `http://milianw.de/tag/massif-visualizer`, accessed: 2016-05-09.

Appendix A

Acronyms

AMT Automatic Multicast Tunneling

API Application Programming Interface

Bash Bourne Again Shell

CIFS Common Internet File System

CSV Comma-Separated Values

CTU Czech Technical University

CV Cross Validation

DNS Domain Name System

EOF End-Of-File

FIT Faculty of Information Technology

FTP File Transfer Protocol

GNU GNU's Not Unix!

HTTP Hypertext Transfer Protocol

IANA Internet Assigned Numbers Authority

ICMP Internet Control Message Protocol

IP Internet Protocol

IPFIX IP Flow Information Export

JSON JavaScript Object Notation

kNN k-Nearest Neighbors

LDAP Lightweight Directory Access Protocol

NEMEA Network Measurements Analysis

NFS Network File System

NTP Network Time Protocol

PC Personal Computer

RBF Radial Basis Function

RDP Remote Desktop Protocol

RFC Request For Comments

RIP Routing Information Protocol

SIP Session Initiation Protocol

SNMP Simple Network Management Protocol

SQL Structured Query Language

SSH Secure Shell

SVM Support Vector Machines

TCP Transmission Control Protocol

TRAP Traffic Analysis Platform

UDP User Datagram Protocol

UniRec Unified Record

Appendix B

Installation Manual

The recommended system for installing the module for classification of devices and the NEMEA framework is *CentOS/7*.

B.1 Dependencies

For installation of the module for classification of devices and the NEMEA framework, following libraries are needed:

- gcc
- gcc-c++
- libtool
- libxml2-devel
- libxml2-utils
- make
- pkg-config

This command can be used for installing the libraries:

```
sudo yum install -y gcc gcc-c++ libtool libxml2-devel make pkg-config
```

B.2 Installation

Source codes can be found on the attached DVD. To install the module from the attached source codes, copy the `src/` directory from the DVD to your home directory.

1. **Install the NEMEA Framework**

   Use the following instructions to install the NEMEA framework from the source codes:

```
tar -xvf src/dist/nemea-framework-2.0.0.tar.gz
cd nemea-framework-2.0.0/
./configure --prefix=/usr/ --libdir=/usr/lib64 \
--bindir=/usr/bin/nemea
make
sudo make install
```

2. **Install the module**

   After you have successfully installed the NEMEA framework, move back to your home directory. Then use the following instructions to configure and build the module:

```
tar -xvf src/dist/device_classifier-1.0.0.tar.gz
cd device_classifier-1.0.0/
./configure --prefix=/usr/ --libdir=/usr/lib64 \
--bindir=/usr/bin/nemea
make
```

B.3 Module Usage

The file `README.md` in the module root directory contains the instructions for using the module.

Example

This is an example of using the module to analyze captured network traffic.

1. Build the module `logreplay` using the instructions above. The module is attached in the tarball `src/dist/logreplay-1.0.0.tar.gz`

2. Extract the sample dataset `data/local-network-fit-nemea.tar.gz`.

3. Run `logreplay` to replay first 100,000 flows from the sample dataset.

```
./logreplay -i "u:test" -f local-network-fit-nemea.csv \
-c 100000 &
```

4. Use `device_classifier` to label IP addresses in the dataset.

```
./device_classifier -i "u:localhost:test"
```

Appendix C

Figures

Listing C.1: Output of Memcheck for the module. All the memory allocated by the module is freed properly. The only reachable block is allocated by the LIBSVM library.

```
==14379== HEAP SUMMARY:
==14379==     in use at exit: 72,704 bytes in 1 blocks
==14379==   total heap usage: 27,257 allocs, 27,256 frees, 9,370,827
    bytes allocated
==14379==
==14379== 72,704 bytes in 1 blocks are still reachable in loss record 1
    of 1
==14379==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/
    vgpreload_memcheck-amd64-linux.so)
==14379==    by 0x55E72AF: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.
    so.6.0.21)
==14379==    by 0x4010139: call_init.part.0 (dl-init.c:78)
==14379==    by 0x4010222: call_init (dl-init.c:36)
==14379==    by 0x4010222: _dl_init (dl-init.c:126)
==14379==    by 0x4001309: ??? (in /lib/x86_64-linux-gnu/ld-2.19.so)
==14379==    by 0x2: ???
==14379==    by 0xFFF000216: ???
==14379==    by 0xFFF000233: ???
==14379==    by 0xFFF000236: ???
==14379==
==14379== LEAK SUMMARY:
==14379==    definitely lost: 0 bytes in 0 blocks
==14379==    indirectly lost: 0 bytes in 0 blocks
==14379==      possibly lost: 0 bytes in 0 blocks
==14379==    still reachable: 72,704 bytes in 1 blocks
==14379==         suppressed: 0 bytes in 0 blocks
==14379==
==14379== For counts of detected and suppressed errors, rerun with: -v
==14379== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

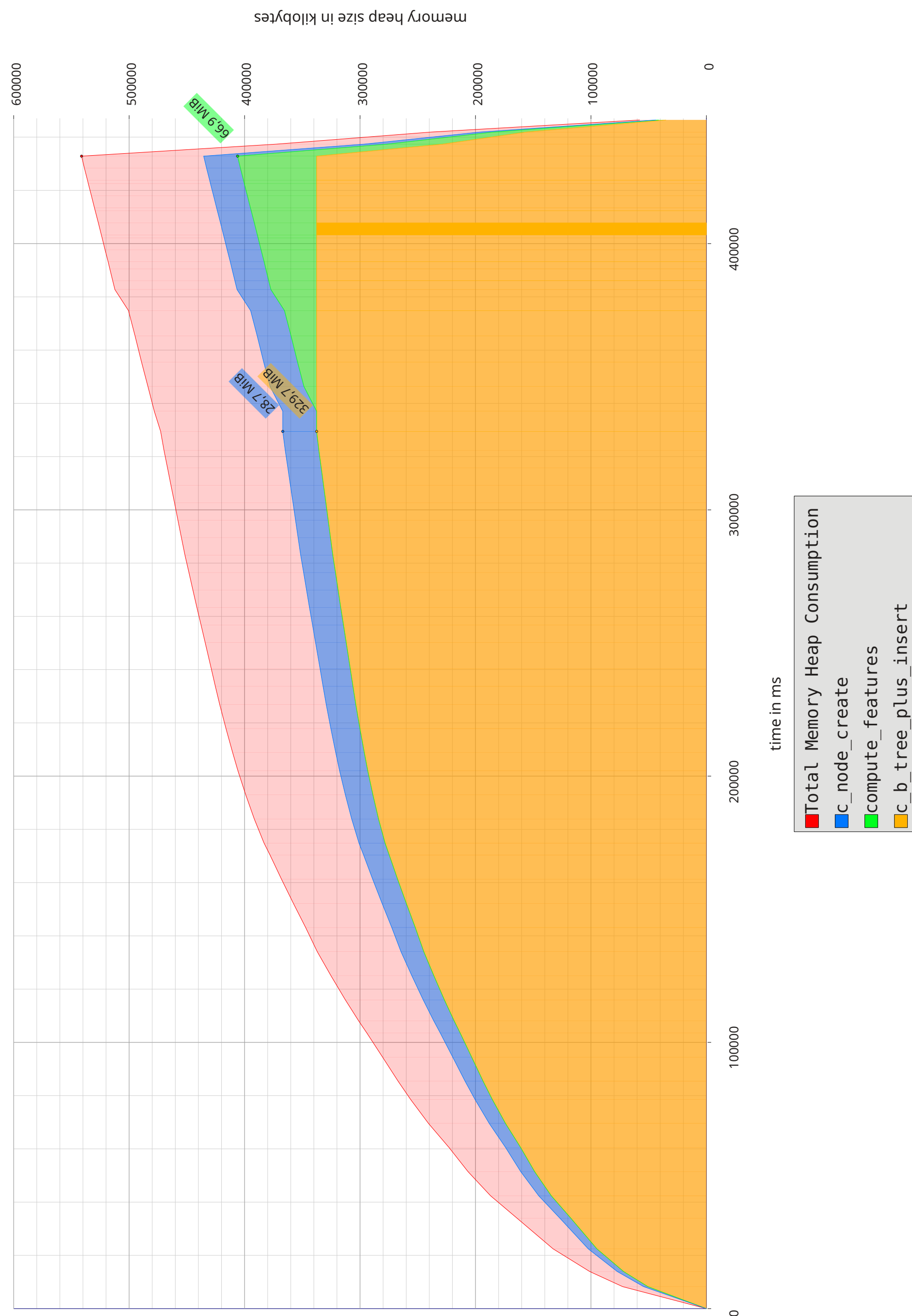


Figure C.1: Output of the Massif for the large-scale network dataset. All the IP addresses in the dataset are tracked. The memory is allocated primarily by the function `c_b_tree_plus_insert` from the implementation of B+ tree. B+ tree is used to store information about IP addresses.

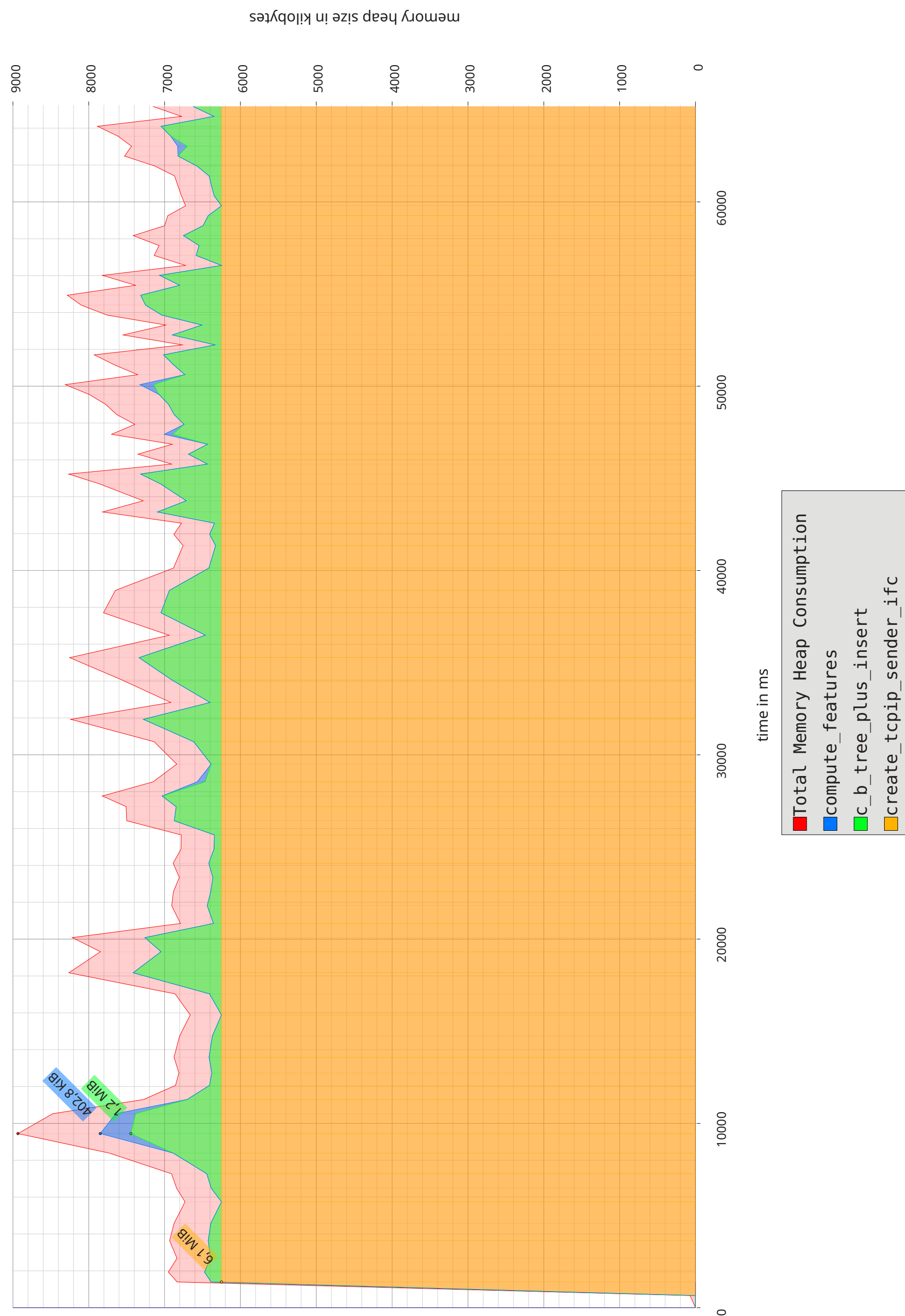


Figure C.2: Output of the Massif for the local network dataset. The memory is freed every 360 minutes. Most of the memory is used by the function `create_tcpip_sender_ifc` from the underlying interface of TRAP library.

Appendix D

Contents of DVD

```
data
└─ local-network-fit.tar.gz ........ sample dataset
doc
└─ doc.tar.gz ........ archive with generated documentation
readme.txt ........ file with DVD contents description
src
├─ dist ........ tarballs with source codes
└─ thesis ........ directory of LaTeX source codes of the thesis
   └─ img ........ thesis figures directory
text
├─ thesis.pdf ........ thesis in PDF format
└─ thesis.ps ........ thesis in PS format
```