FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Real-time Network Flow Control using Machine Learning and OVS
Student:	Štěpán Šimek
Supervisor:	Ing. Karel Hynek
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2022/2023

Instructions

Get acquainted with network monitoring methods based on deep packet inspection and (extended) IP flows.
Analyse the area of Real-Time communication technology and protocols, focus on their detection possibilities based on traffic characteristics.
Design an algorithm for automatic detection of real-time communication based on Machine Learning. Develop a software prototype capable of its detection. Use the information from the detector for traffic prioritisation in Open vSwitch (OVS).
Evaluate the accuracy of the machine learning-based algorithm and the proper functionality of the prototype.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 30 January 2022 in Prague.

Bachelor's thesis

REAL-TIME NETWORK FLOW CONTROL USING MACHINE LEARNING AND OVS

Štěpán Šimek

Faculty of Information Technology
Department of Computer Systems
Supervisor: Ing. Karel Hynek
May 11, 2023

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Šimek Štěpán. *Real-time Network Flow Control using Machine Learning and OVS*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my supervisor, Ing. Karel Hynek, for his guidance, sharing his expertise, and unwavering support. I'm also grateful to Ing. Tomáš Čejka, Ph.D. for sharing his expertise and technical support in the realization of the Proof-of-Concept experiment. Finally, I would like to express my gratitude to my family and friends who supported me on this journey.

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 Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on May 11, 2023 .....

Abstract

Real-time communication using online collaboration platforms plays an important role in everyday business operations. Its prioritization in our networks can help mitigate problems imposed by the network's limitations. This thesis aims to design a prioritization solution for real-time protocol. The solution utilizes machine learning for real-time traffic recognition and Open vSwitch subsystem for prioritization. The solution was designed based on a thorough study of related works. Anonymized network traffic dataset was captured on real-world ISP lines. Additionally, the prioritization software prototype was implemented into open-source flow exporter IPFIXprobe and tested using a small home-office router Turris.

Keywords traffic prioritization, real-time, network flow control, traffic recognition, machine-learning, Open vSwitch, IP flow

Abstrakt

Real-time komunikace pomocí online platforem zastává důležitou roli v každodenních aktivitách mnoha společností. Prioritizace této komunikace umožňuje zmírnit dopady zapříčiněné limitacemi sítí. Cílem této bakalářské práce je navrhnout a implementovat řešení prioritizace pro real-time protokoly. V rámci tohoto řešení bylo využito strojového učení pro rozpoznání real-time síťového provozu a technologie Open vSwitch pro zajištění prioritizace. Navržené řešení vychází z podrobné analýzy používaných real-time protokolů na reálné síti. Navíc pro vytvoření modelu strojového učení byla vytvořena nová a rozsáhlá anonymizovaná datová sada zachycená na síti reálného poskytovatele internetového připojení. Prototyp softwaru prioritizace real-time protokolů byl zakomponován do open-source exportéru síťových toků IPFIXprobe a otestován pomocí routeru Turris určeného pro domácnosti a malé podniky.

Klíčová slova prioritizace síťového provozu, real-time, řízení síťových toků, rozpoznání síťového provozu, strojové učení, Open vSwitch, IP toky

List of abbreviations

AdaBoost	Adaptive Boosting
API	Application Programming Interface
CESNET	Czech Education and Scientific NETwork
CLI	Command-line interface
CNAMEs	Canonical Name
CSRC	Contributing source
CSV	Comma-separated Values
CV	Cross-validation
DNS	Domain Name System
DPI	Deep Packet Inspection
DST	Destination
DTLS	Datagram Transport Layer Security
ETL	Extract-Transform-Load
FN	False Negative
FP	False Positive
HPE	Hewlett Packard Enterprise
ICE	Interactive Connectivity Establishment
iOS	iPhone OS
IoT	Internet of Things
IP	Internet Protocol
IPFIX	Internet Protocol Flow Information Export
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ISP	Internet Service Provider
LACP	Link Aggregation Control Protocol[edit]
ML	Machine Learning
MS	Microsoft
NAT	Network Address Translation
OS	Operating System
OVS	Open vSwitch
P2P	Peer to Peer
PCAP	Packet Capturing
PCAPNG	PCAP Next Generation
PoC	Proof-of-Concept
QoS	Quality of service
RFC	Request for Comments
RFECV	Recursive feature elimination with cross-validation
RSPAN	Remote SPAN
RTCP	Real-Time Transport Control Protocol
RTP	Real-time Transport Protocol
SDN	Software-defined Networking
SDP	Session Description Protocol
SOHO	Small-office/home-office
SPAN	Switched Port Analyzer
SRC	Source
SRTP	Secure Real-time Transport Protocol
SSRC	Synchronization source
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TN	True Negative

TP	True Positive
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
VoIP	Voice Over IP
WebRTC	Web Real-Time Communication
XGB	eXtreme Gradient Boosting
XGBoost	eXtreme Gradient Boosting

Introduction

Organizations all around the world rely on real-time communication/collaboration platforms. These platforms hold an essential role in organizations because they provide means to communicate with colleagues and customers located at different geographical locations across the world. These services are often offered via the Internet and it introduces a potential challenge for the organizations as they must be connected to the Internet via a stable and reliable connection catering to all network-traffic-related needs of the organization. Internet connection available to the organization may be limited by the environment, location, and other factors. To meet all the needs and prioritize the business needs, an organization must develop a new model of managing its network ensuring that the most critical applications and platforms are prioritized network-wise.

Conventional approaches to network prioritization rely on a pre-defined set of rules and configurations that may not be able to adapt to the dynamic and unpredictable nature of modern network traffic. Machine learning techniques can provide a more flexible and adaptive approach to network flow control by analyzing network traffic patterns and predicting optimal flow rates in real-time.

One popular tool for implementing network flow control is Open vSwitch (OVS), which is a flexible and scalable virtual switch that can be used to manage network flows in a variety of environments. By combining OVS with machine learning algorithms, it is possible to create a real-time flow control system that can adapt to changing network conditions and optimize flow rates to improve overall network performance.

We explore the use of machine learning techniques to implement real-time network flow control using OVS. We will review the challenges involved in designing such a system and discuss how machine learning algorithms can be used to overcome these challenges. We will also present experimental results demonstrating the effectiveness of our approach in a variety of network environments. Addressing potential limitations and risks of using machine learning in network flow control will be mentioned, few examples include the need for large amounts of training data and the potential for unexpected behavior.

Traffic prioritization depending on flow-based analysis has multiple advantages, it is more efficient resource-wise and less invasive to user privacy as it prioritizes the flow based on network flow characteristics rather than examining the content of the communication. Our current digital society puts more emphasis on user privacy than ever before. The data transferred over our networks contain lots of personal information that can be misused by malicious actors. Establishing trust between a user and a network operator is essential in our digital society. Shattering that trust could lead to a breakdown of mutual trust established via our networks.

Another advantage of using machine learning for network flow control is that it can be customized to suit the specific needs of a particular network environment. Different machine learning algorithms can be trained on different types of data to optimize flow control for different network

configurations and traffic patterns.

Overall, the combination of machine learning and OVS presents an opportunity to develop more advanced and effective real-time network flow control systems. As network traffic continues to grow in complexity and volume, such systems will become increasingly important in ensuring reliable and efficient network performance.

This thesis will be centered around audio/video and screen-sharing types of traffic. It focuses on the research and design part, but it will also include an implementation (Proof of Concept) that can be used for demonstration purposes.

The first chapter 1 establishes the necessary concepts needed for the following parts of the thesis, it consists mainly of an introduction to topics of network flow monitoring, protocols, used concepts, and technologies. The second chapter 2 addresses the topic of dataset creation. It emphasizes the process of creating such a dataset and the methods used for it. The third chapter 3 guides us through the process of machine learning. Its viability, advantages, and disadvantages of the proposed solutions are discussed there. The fourth chapter 4 pursues the prioritization part and prepares the foundation for our Proof-of-Concept. The next chapter 5 aims to evaluate the machine learning model and discuss the PoC. The last chapter 6 concludes the outcome of this thesis.

Chapter 1
Background

1.1 Network traffic monitoring

Network monitoring is a crucial part of network management. By monitoring and analyzing network traffic, an organization can get more insight into the performance and security of its network. Due to the increasing complexity of our networks and their role in organizations, administrators must manage the network in a way where all the critical applications are accessible at all times and with reliability in mind. Network monitoring combined with network analysis gives the administrator strong tools to manage the network.

Our emphasis on secure communications has been increasing in the last decades to the level that most of the communication happening on the Internet is encrypted. [1] Not only has the encryption of payloads been getting more common, but encryption of the metadata (protocol metadata) as well, sometimes to the degree when a regular classification system depending on reading protocol headers is not able to detect the sub-protocol, but only a transport-level protocol (such as TCP or UDP). This approach is great for security, but not very much for network management purposes.

We have multiple methods of analysis available, one of them is based on deep packet inspection (abbreviated as DPI), and another method is flow-based analysis. Each one has its own advantages and disadvantages. This section aims to explain the differences between them and their pluses and their drawbacks.

1.1.1 Packet based - Deep Packet Inspection

Deep Packet Inspection is defined by Fortinet as "*a method of examining the content of data packets as they pass by a checkpoint on the network*" [2]. DPI examines not only the headers of a packet but metadata and payload as well. This approach enables the classification algorithms to detect the type of traffic with more precision.

One of the advantages compared to packet headers inspection is more precise classification which enables network administrators to set up more specific rules rather than generic ones.

Unfortunately, this approach does not only consist of advantages but has multiple disadvantages as well. DPI requires additional processing which is usually very resource-consuming as protocol detection and parsing have to take place. Due to the nature of DPI, more processing is required to make a judgment which can result in delays in communication.

Nowadays push for security and privacy in our networks changes the structure of packets being sent and delivered over our networks. More and more metadata (including packet subheaders) is being encrypted thus making DPI less efficient to the point when DPI is not able to

say anything more than basic packet header reading method. Deep packet inspection will not be described further as the thesis focuses on a flow-based approach.

1.1.2 IP Flow based

IP flow monitoring is based on the concept of flows, which can be described as a sequence of packets that share the same key attributes [3] (in our case: destination and source IP addresses, protocol, and source and destination port numbers). The very same key attributes are called flow keys. Flows can also have additional specific characteristics – for example inter-packet delays etc.

Flows are usually delimited by a time frame starting from the first packet processed until the last one observed. There is often introduced a maximum timespan of a flow to prevent keeping stale flows from overloading the system and for logging purposes. A flow can reach its end in multiple ways: reaching the maximum time limit introduced for the flow, reaching the maximum time introduced for the inter-packet delay and detecting the end of the connection (for stateful connections), and many more.

According to Hofstede et al. [4], flows are divided into two categories: unidirectional and bidirectional. Unidirectional flows contain characteristics for traffic in one direction. On the contrary bidirectional flows merge two unidirectional flows into one in a way when direction-less key attributes are the same but direction-relevant key attributes stay the same in 'forward' direction and 'backward' flow attributes are inverted. The direction-relevant key attributes for the flow are set up based on the first packet processed in most cases however exceptions exist. [5] This thesis considers the method based on the first packet processed. Direction-relevant attributes are doubled for bidirectional flows (examples include a count of source/destination packets, a count of bytes coming in source/destination directions, etc.).

Extended IP flows are an extension to an IP flow concept, they contain additional information compared to basic IP flows. This information can be: information about packet payloads, semi-parsed protocol information, and flow statistics and many more. Flow statistics can contain information ranging from the number of packets, and the number of bytes transferred to intra-packet delay combined with approximated standard deviation of packet size.

This additional information allows network administrators to monitor and troubleshoot problems on the network with more insight.

1.2 Network monitoring architecture

In recent years flow-based network analysis has been gaining popularity. The whole process starts at an observation point where flow exporters operate. [5] A flow exporter is a network device that generates flow from the network traffic it goes through. Parsing and extracting information from passing packets and creating or updating flows is its main responsibility. The flow exporter creates or updates new flows based on key flow characteristics.

The flow exporter sends the flow information to the flow collector. The next phase takes place in a flow collector. A flow collector is a software or hardware component receiving flows from a flow exporter [5]. It aggregates flow records from multiple flow exporters, pre-processes them, and then stores them for additional analysis. A general overview of flow-based network monitoring architecture can be seen in figure 1.1.

There are many flow exporters available on the market ranging from proprietary to open-source. Some of the popular flow export protocols are NetFlow, sFlow, J-Flow, and IPFIX.

"*NetFlow is a network protocol developed by Cisco for collecting IP traffic information and monitoring network flow*" [6]. It is mostly used in Cisco devices – routers and switches. Netflow has gone through many revisions. The most widely used is Netflow version 5 which offers basic

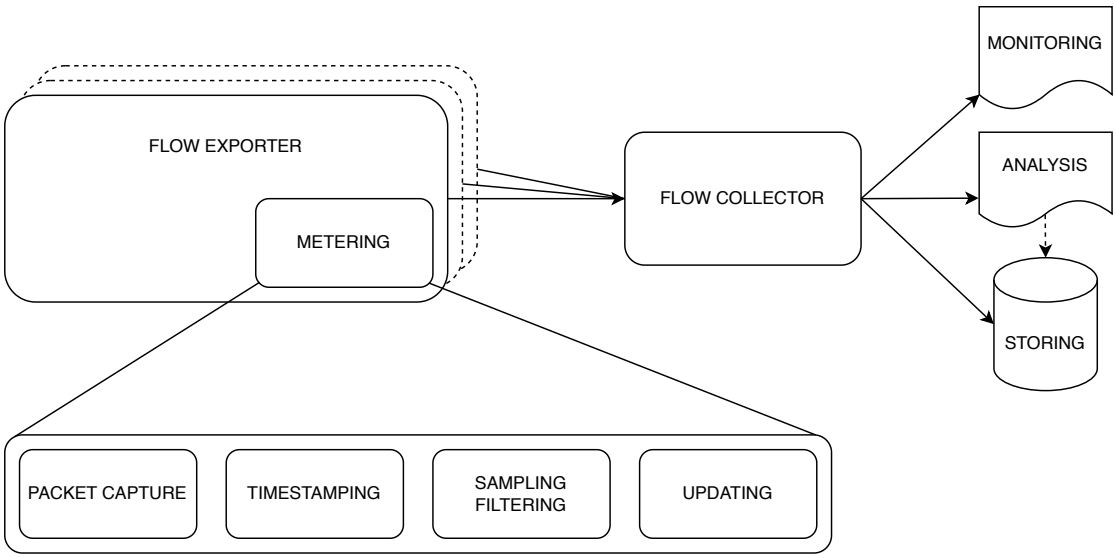


Figure 1.1 General overview of flow monitoring architecture

flow data, meanwhile newer version offer more advanced features (support for custom fields and more).

1.2.1 IPFIX

IPFIX is a newer flow export protocol based on NetFlow v9 [3]. It provides a flexible and extensible way to export flow data and supports a wide range of transport protocols. It supports IPv6 as well as IPv4. It also provides support for message authentication and encryption. IPFIX uses a template-based approach to define the structure of flow records. The structure contains the names of the fields and their type. It allows an administrator to define custom fields and incorporate organization-relevant data to be passed into the flow analysis. This dynamic approach improves scalability. IPFIX is also supported by big players in the networking market – Cisco [7], Juniper [8], HPE Aruba [9] and Fortinet [10] and many more. There are various IPFIX flow exporters on the market, such as Cisco's Joy [11], Yet Another Flowmeter [12], and IPFIXprobe [13]. The IPFIXprobe is the most relevant for this thesis, therefore it will be further described.

1.2.1.1 IPFIXprobe

IPFIXprobe [13] is an open-source flow exporter. It creates biflows (bidirectional-flows) either from network traffic captures, from a network interface, and exports them to an output interface. IPFIXprobe system is an extensible platform offering multiple plugins on their GitHub page [13]. The modules are often running on multiple threads to increase the processing throughput.

Its plugins fall into one of three categories:

- **input plugins** are plugins responsible for 'feeding' the input data in the correct format to the processing plugins
- **processing plugins** are plugins responsible for data processing (protocol recognition, statistics collection and so on)
- **output plugins** are plugins responsible for outputting the processed data (exported flows) for further processing

Due to the nature of this thesis, we will mostly focus on the processing category of plugins. The processing consists of multiple phases:

1. Metadata initialization phase
2. Metadata enriching
3. Flow export phase

The metadata initialization phase consists of memory allocation of the necessary metadata for the plugin and flows and its initialization. The metadata enriching phase has two stages: `pre_update` and `post_update`. The `pre_update` stage is invoked when a new packet is received but is not processed by the internal IPFIXprobe flow metadata enricher. The `post_update` stage is invoked after the internal metadata of flow has been enriched. The processing order of plugins is determined by the order of plugins in the initialization phase.

1.3 VoIP real-time traffic

Real-time communication such as calling, video-calling, and streaming are important part of our everyday life. The amount of data transferred and its importance over the interconnected global network rise day by day. The combination of more data transferred and at the same time offering stable, fast, low-delay connections is a very hard task. Some organizations already reached a point when their throughput comes close to the limit, in that case, they have to prioritize some type of communication. Unfortunately for the case of network management, our traffic is getting more obfuscated than before because of security and privacy reasons. [14] It is hard to distinguish the 'intention' behind the communication without proper analysis. This analysis without proper hardware and software backing introduces another complex variable into the game of network management and prioritization.

Many real-time protocols build their protocol on top of the UDP transport protocol for IP-based networks. This thesis will focus on real-time communication such as calling, video-calling, and screen-sharing. Our models can also be applicable in other cases. Lots of modern protocols rely on a network stack called WebRTC.

1.3.1 WebRTC

WebRTC is "*a technology that enables Web applications and sites to capture and optionally stream audio and/or video media, as well as to exchange arbitrary data between browsers without requiring an intermediary.*" [15] All WebRTC components must use encryption, the encryption scheme used is DTLS. It is very popular among real-time messaging app providers. It is used in applications such as Google Meet [16], Facebook Messenger [17], Discord [18] and many more. WebRTC consists of multiple frameworks: ICE (Interactive Connectivity Establishment), SDP, and many other data/signaling protocols. Data channels in WebRTC typically use SRTP, a secure derivation of the RTP protocol.

1.3.1.1 NAT and ICE

NAT (Network Address Translation) holds a very important role in our networks. Its role is to connect private networks to public ones. RFC 3022 describes Network Translations as follows:

> "*Basic Address translation would (. . .) allow hosts in a private network to transparently access the external network and enable access to selective local hosts from the outside. Organizations with a network setup predominantly for internal use, with a need for occasional external access are good candidates for this scheme.*" [19]

The need for network address translations has been constantly rising due to the limited number of IPv4 public addresses. The number of devices connecting to the intra-connected global network has been exponentially rising in the last decades and the trend with the presence of IoT is not showing any signs of a significant slowdown. Private IPv4 addresses are not and should not be routable outside of internal networks, the routability of these addresses would defy the core concept of private addresses. Theoretically, we could assign a public IPv4 address to each device in an organization so the devices can communicate with the Internet, unfortunately, this approach is unfeasible due to the vast amount of devices wanting to communicate and the scarce amount of public IPv4 addresses available. Another approach how to solve the problem could be: translating private addresses into public space and vice versa.

Symmetric NAT is a type of NAT that maps a unique combination of an internal IP address, port, and protocol to a unique combination of external IP and a port. The symmetric NAT takes into account the protocol, destination IP and destination port thus not allowing to send a packet from a different external IP address or a different port to the mapped address and port. [20] For visual explanation please refer to figure 1.4.

1.3.1.2 ICE

ICE provides a framework allowing a client to form a connection with another client without direct network visibility (f.e. being behind NAT). The framework is complemented by two protocols: STUN and TURN. SDP is often used with it and shall be mentioned as well.

STUN is a protocol allowing you to obtain your public IP address and determine any restrictions that might prevent you from forming a connection with a peer. Figure 1.2 shows the process of establishing a connection if no restrictions are found. If a restriction caused by Symmetric NAT is found, TURN will be used.

TURN is a protocol allowing to bypass the restrictions of Symmetric NAT. The client opens a connection to a TURN server and sends the information through the TURN server. The second peer applies the same principles. All the connections are conveyed over the TURN server. It causes overhead and introduces delays, this method is only used in the cases when direct connection is not available. [21] Figure 1.3 shows a process of communication between peers using TURN.

SDP is a framework for describing multimedia content of the connection, such as codecs, resolution, encryption, and such. SDP is made of at least one line of UTF-8 text, each line starts with one character type and '=' and its value or description. [21]

Due to the reasons of peer-to-peer communication happening we cannot do prioritization based on IP address ranges given by platform providers. [22]

1.3.2 RTP protocol and its derivatives

The real-time transport protocol is a data transport protocol designed for the delivery of data with real-time characteristics, for example, audio and video. RTP offers multiple features such as payload identification, sequence numbering, timestamping, and delivery monitoring. RTP is mainly used in combination with UDP, thanks to multiplexing and checksum support. It also supports multicast communications. [24]

Since RTP is only a data transport protocol, it should be accompanied with a management-plane protocol. Designers of this protocol thought about that and developed a lightweight version of the management protocol called RTCP. Its features include QoS monitoring, sharing information about participants etc. [25] Multiplexing of sessions is provided by the destination transport address (IP address and port). For example, in a VoIP scenario, audio and video should be

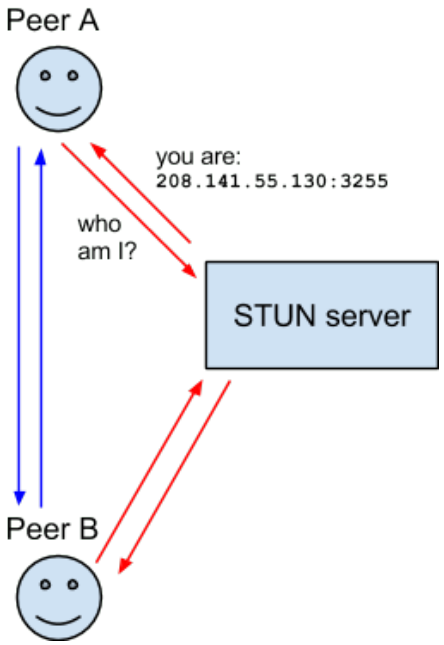


Figure 1.2 STUN connection schema, from [21]

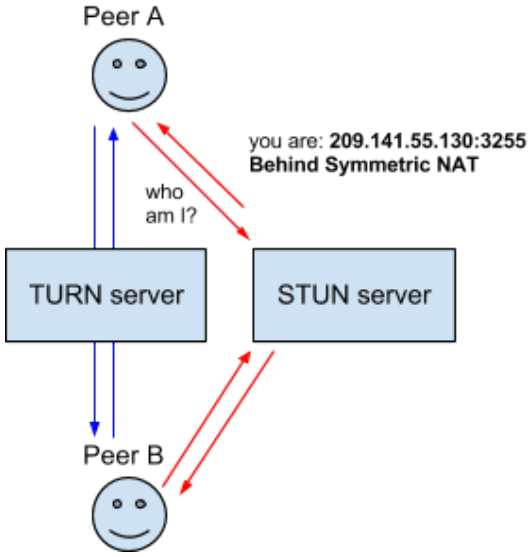


Figure 1.3 TURN connection schema, from [21]

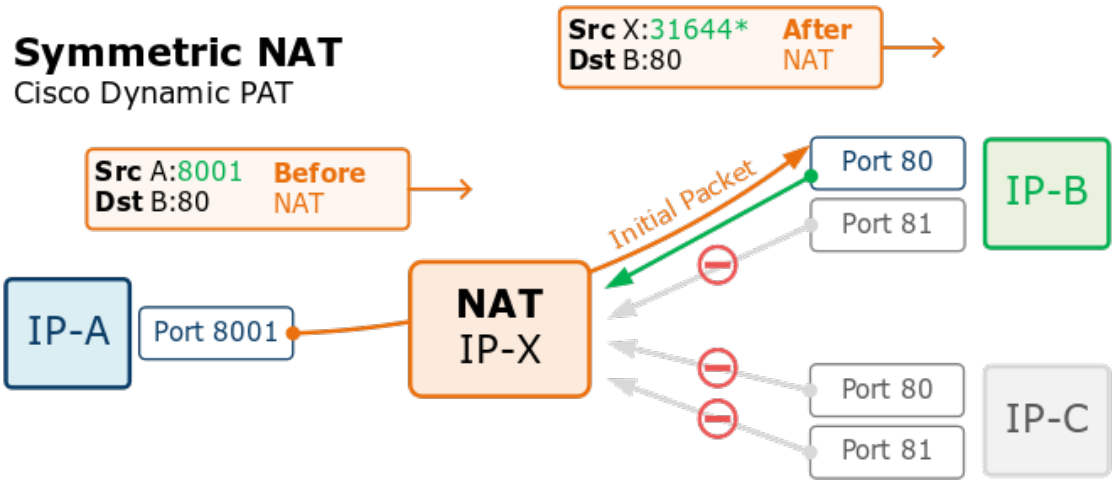


Figure 1.4 Symmetric NAT, from [23]

carried in different sessions. Thanks to this design prioritization can be done per flow (f.e. just audio in our case). Figure 1.5 shows the RTP header's structure and it comprises of:

- flags consisting of version, padding, extension, CSRC count, and marker bit
- payload type
- sequence number
- timestamp
- SSRC
- CSRC list

Version field indicates the version of the revision of the RTP protocol used. Current *version* revision is two. *Padding* flag specifies whether the packet contains one or more octets at the end that are not part of the payload. The last byte of padding specifies a count of padding bytes. RTP allows extensions, its presence is defined by bit set by *extension* flag. *CSRC count* field specifies the number of CSRC identifiers following the fixed header (if the extensions field is used CSRC identifiers come after the extension fields). *Marker bit* is a flag not having a fixed meaning, its meaning depends on a profile. The profile is a description of modifications and additional interpretations related to a certain app towards the RTP protocol. It contains a set of payload type codes and their mapping payload formats. It also defines extensions and modifications to the RTP protocol.

The type of the RTP payload can be found in *payload type* field. It determines the interpretation for the application. An application using RTP can change the payload type. The number of RTP packets send is saved to *sequence number* field. It can be used to detect packet loss. The initial value should be random. *Timestamp* representing the sampling moment of the first data contained in the RTP packet. The sampling must be derived from a clock that increments monotonically and linearly in time. If an application uses fixed-rate sequencing, then the *timestamp* clock can be incremented by 1 for each sampling period. If an application processes blocks covering f.e 173 sampling periods, the *timestamp* should be increased by 173, despite being transmitted or dropped. The initial value of *timestamp* should be random. *SSRC* field complements the header with an identification of the synchronization source. It should be chosen randomly. No two synchronization sources should have the same *SSRC* in the same RTP session. RTP implementation should take into consideration the possibility of a collision and implement a procedure to solve it. Synchronization source can change its identifier but must choose a new *SSRC* identifier to avoid ambiguity. *CSRC list* field contains contributing source identifiers. [24]

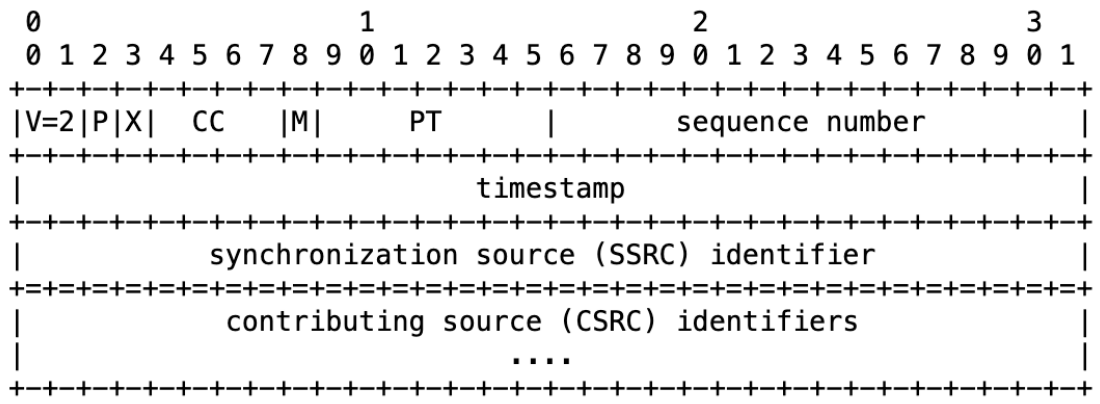


Figure 1.5 RTP header, from [24]

1.3.2.1 SRTP

SRTP (Secure RTP) is built on top of RTP and is one of the most used protocols for real-time communications, especially audio and video. As SRTP is built on top of RTP, it follows its principles and the SRTP header is a small modification/extension to the original RTP header. SRTP header adds authentication and encryption to the services provided by RTP, its scope can be seen in figure 1.6. Additionally, it contains an authentication tag. Encryption and authentication mechanisms are agreed upon via a different channel.

Wide usage of this SRTP by popular platforms such as Microsoft Teams [26], Zoom [27], Google Meet [28], Skype [29], Facetime [30], Whatsapp [31] and Signal [32] make this protocol a great target for analysis. **This thesis shall focus entirely on this protocol**.

1.3.2.2 RTCP

RTP is accompanied by another protocol providing a management plane of the connection. This role can be done by RTCP. An application using RTCP transmits regular messages to all participants in the session. [24]

RTCP has multiple roles:

- provide feedback on the quality of transmission
- provides persistent identifiers (CNAMEs) for an RTP session
- synchronization protocol between hosts
- furnishes the protocol with basic session control

The last role is optional. All other roles should be used in all environments. They should work also in multicast environments.

1.3.3 SRTP and MS Teams

MS Teams is one of the most common applications for audio and video calls in many organizations around the world. The documentation provided by Microsoft for MS Teams in terms of protocols used and its endpoint is more detailed compared to its competitors. According to [34, 35], MS Teams uses four defined ports, each for a different traffic type. The port numbers and their assigned services are written in table 1.1. The IP ranges used by MS Teams and Skype for Business are then shown in table 1.2.

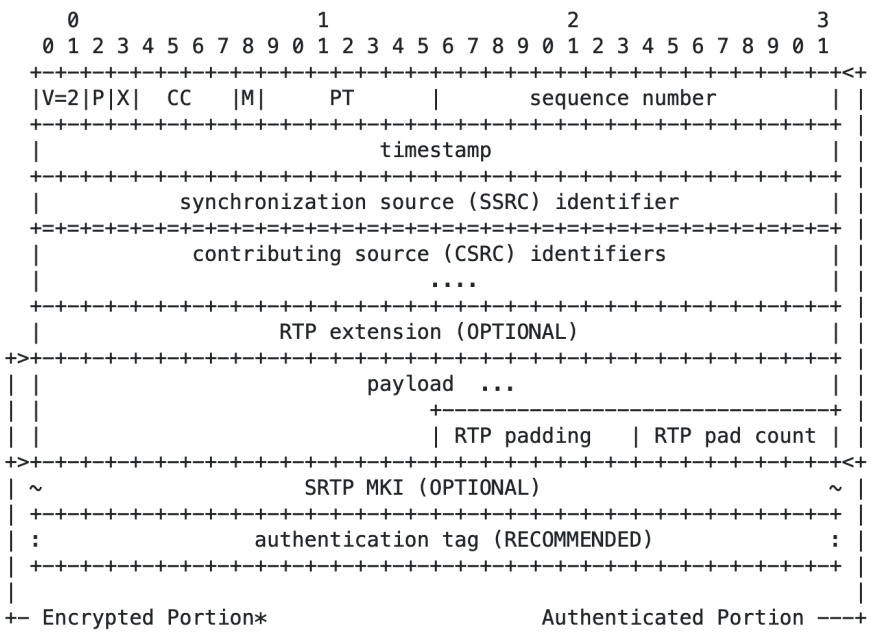


Figure 1.6 SRTP header, from [33]

Table 1.1 Ports used by MS Teams for real-time communication

Port	Purpose
3478	Relay Discovery allocation and real-time traffic (ICE - STUN and TURN)
3479	Audio
3480	Video
3481	Video Screen Sharing

Table 1.2 IP address ranges used by MS Teams for real-time communication

IP address range
13.107.64.0/18
52.112.0.0/14
52.122.0.0/15
2603:1063::/39

1.4 Machine learning methods for traffic recognition

Machine learning methods can be divided into multiple categories:

supervised "*learning is a machine learning approach that's defined by its use of labeled datasets. These datasets are designed to train or "supervise" algorithms into classifying data or predicting outcomes accurately. Using labeled inputs and outputs, the model can measure its accuracy and learn over time.*" [36]

unsupervised "*learning uses machine learning algorithms to analyze and cluster unlabeled data sets. These algorithms discover hidden patterns in data without the need for human intervention (hence, they are "unsupervised").*" [36]

The most popular algorithms for traffic recognition [37] include Decision Trees [38], k-nearest

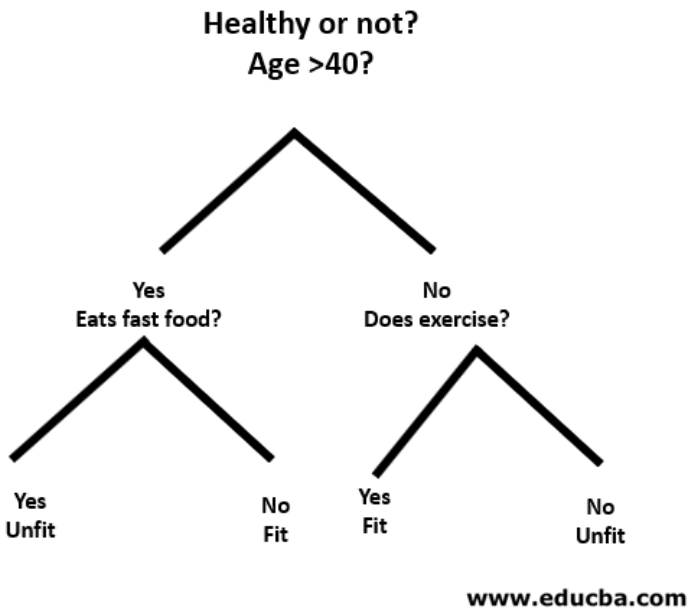


Figure 1.7 Example of a Decision Tree, from [42]

neighbours [39], Random Forests [40], boosted algorithms [41] (based on Decision Trees and Forests) and neural networks.

1.4.1 Decision Trees

Decision Trees are a supervised ML (Machine Learning) algorithm that represents a decision-making process. [38] They can be used for classification and regression problems. The algorithm is illustrated as a tree seen in figure 1.7 where every internal node represents an IF condition over a feature in a learning dataset. The internal nodes' direct children represent one of the outcomes of the condition. Each leaf node represents a decision made by the decision algorithm based on the input data. The decision can have a form of probability of binary classification.

1.4.1.1 Criterions for splitting

The splitting criterions determine how is the data divided into different branches of the tree based on the data features. The two most common ones are Gini impurity and entropy.

Gini impurity is a criterion for measuring the impurity/uncertainty of a set of data points in a group. For example: in Decision Trees, we can measure the purity of split groups – the desired result is to have higher purity (all points in a group belong to the same group). The Gini coefficient ranges from $< 0; 0.5 >$. Zero represents a pure dataset, meanwhile, 0.5 represents an impure dataset (an equal number of data points belonging to different classes). The Gini impurity measures the probability that a random data sample (data point) would be misclassified.

Entropy is similar to Gini impurity, it is also a criterion for measuring impurity/randomness of a set of data points in a group. The range for entropy is $< 0; 1 >$. Entropy represents the degree of randomness in the distribution.

1.4.1.2 Building a Decision Tree

They are built by recursively splitting the initial set of learning data into smaller subsets using a top-down approach. The building algorithm selects a feature based on certain criteria, the most common are Gini impurity and entropy, it splits the dataset into n-parts (n depends on the number of possible outcomes) and continues the process until the stopping condition is reached. Stopping conditions can be defined in several ways, such as exhaustion of available features, reaching maximum depth, etc.

Decision Trees are great at handling both categorical and numerical data without requiring any additional transformations, which makes them flexible and versatile.

They are susceptible to overfitting. Overfitting happens when the model incorporates noise in the learning data, rather than generalizing it. It can reflect in poor performance when evaluating the model on new data. They also offer limited capabilities in terms of handling continuous features, as they only do splits into n-intervals.

1.4.2 Trees ensembles

Tree ensembles is a generic term describing a combination of several base estimators (in our case Decision Trees) into a new algorithm. [43] Simplified graphic overview can be seen in figure 1.8. Random Forest is an algorithm belonging to the average-based category and boosted forests belong to the boosted-based category. The tree ensembles are usually distinguished into two categories:

average-based "*The driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced.*" [43]

boosting-based "*Base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble.*" [43]

1.4.2.1 Random Forest

Random Forest is an ML algorithm combining outputs of multiple Decision Trees into a single result. [40] The algorithm is an extension to the category of bagging algorithms. "*The algorithm relies on a pseudorandom procedure to select components of a feature vector, and Decision Trees are generated using only the selected feature components. Each tree generalizes classification to unseen points in different ways by invariances in the unselected feature dimensions. Decisions of the trees are combined by averaging the estimates of posterior probabilities at the leaves.*" [45] Random Forests' advantages compared to Decision Trees are reduced overfitting, and higher robustness related to handling missing values and noise in the data.

1.4.2.2 Boosted forest

Boosted forests is an ML algorithm combining principles of Decision Trees and gradient boosting. In boosted forests, Decision Trees are created sequentially, each additional tree is trained to correct the errors of the previous tree. Each training phase starts with assigning weights to the data points, with weights of misclassified points increased to emphasize their importance. The goal of boosted forests is to create a more robust model by adding weaker learners to the ensemble. Advantages of the ensemble include improved accuracy. It is still prone to overfitting, but there are available countermeasures. We can find many implementations, AdaBoost (Adaptive Boosting) [46] XGBoost (eXtreme Gradient Boosting) [47] are one of the most popular ones.

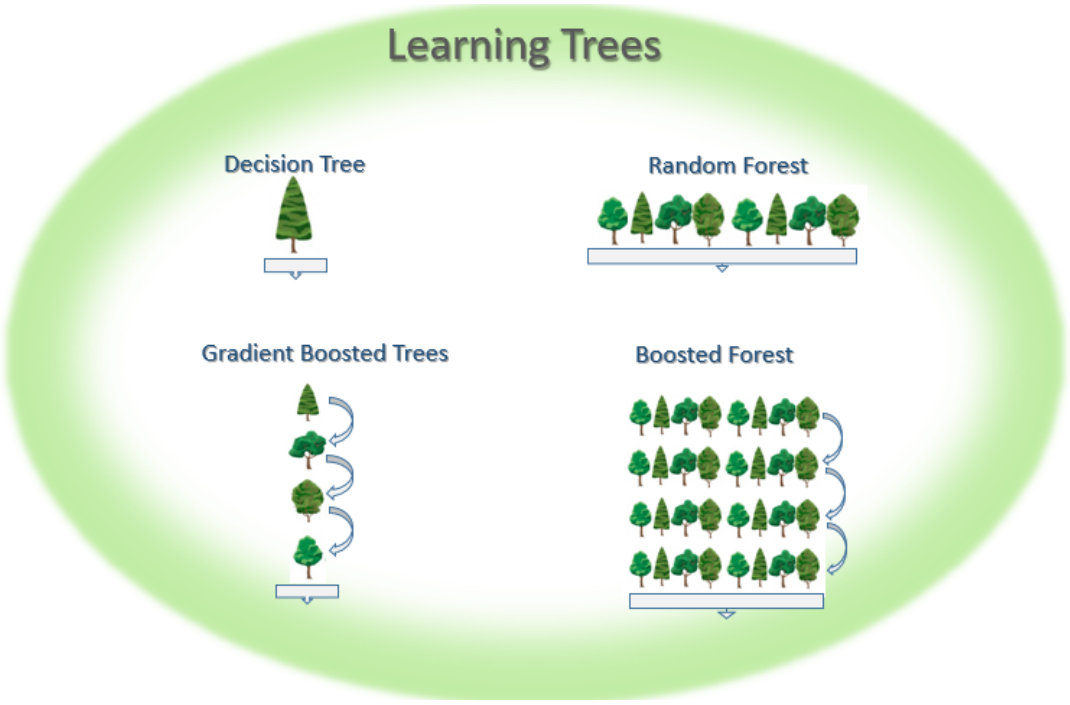


Figure 1.8 Graphical visualization of Decision Trees/Forests/Boosted forests, from [44]

1.5 Open vSwitch

"*Open vSwitch is a (...) virtual switch (...) designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols (e.g. NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, 802.1ag)*" [48] It allows network administrators to manage virtualized network infrastructure.

1.5.1 OpenFlow protocol

"*OpenFlow is a network communication protocol used between controllers and forwarders in an SDN architecture. The core idea of SDN is to separate the forwarding plane from the control plane. (...) OpenFlow introduces the concept of flow table, based on which forwarders forwards data packets. Controllers deploy flow tables on forwarders through OpenFlow interfaces, achieving control on the forwarding plane.*" [49] Figure 1.9 generalizes the architecture of SDN into 3 planes, resp. 2 planes: control plane and forwarding plane.

1.5.2 Open vSwitch and flows

OVS (Open vSwitch) stack has multiple kinds of flows:

OpenFlow flows are used by OpenFlow controllers to define a switch's policy. They support wildcards, priorities, and multiple tables. [50]

Kernel flows also called Datapath flows, are bare flows without support of priorities. They exist only in one table, which makes them suitable for caching. [50]

Further in the thesis, all flows mentioned in the context of Open vSwitch technologies will be implicitly OpenFlow flows if not defined otherwise.

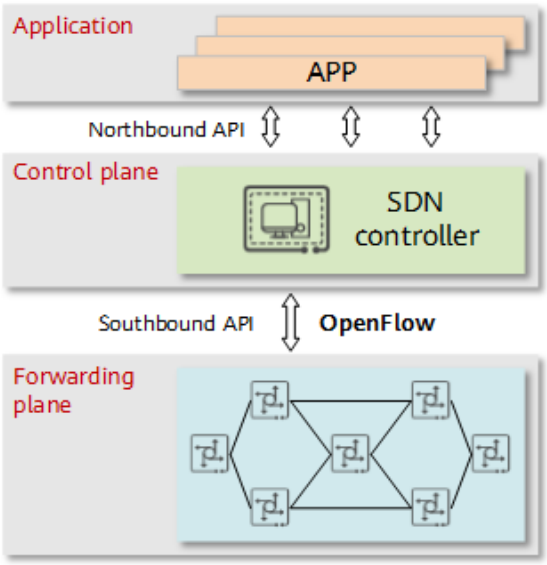


Figure 1.9 Role of OpenFlow in SDN, from [49]

1.5.2.1 Flow prioritization

Flow prioritization in OVS can be achieved by creating a QoS system. It requires having multiple queues for traffic classes and assigning flows to them. The assignment is done based on the OpenFlow rules created by a separate system (in our case RTP recognition algorithm). The queues have many interesting parameters, the most important for our case are `other-config:max-rate` and `other-config:min-rate`, values for both of the parameters are specified in bits per second. The OVS QoS subsystem first tries to fulfill the `min-rate` needs of all relevant queues (with guaranteed `min-rate`s) and after that, it fulfills other traffic needs.

The rule's priority sets the order of processing and matching the flows against the rules set by OVS. The priority values range from 0 to 65 535. Higher priorities will get matched sooner. If two rules have the same priority, then the order during evaluation is undefined. The flow's priority defaults to 32 768 if not specified explicitly [51].

Version 1.4 [51] introduces a new field called 'importance', which defines the order of flow eviction. Importance values range from 0 to 65 535. Zero is the default value and makes the flow non-evictable in terms of importance [51].

The rule defines the 'treatment' of the packets (in our case prioritization) in a flow defined by the flow key. Flows in OVS are uni-directional, therefore we need to create rules for both directions. According to OVS manual [51], the rule (or OVS flow) consists of:

- Match condition (information identifying the flow, could be source and destination IP addresses, transport protocols, etc.)
- Set of actions applied to the IP flow (normal, modify, drop, forward, etc.)

For prioritization purposes, we are interested in *set_queue* action, which is followed by the name of the queue created in the QoS phase.

Chapter 2
Dataset creation

As this thesis focuses on supervised learning algorithms for flow recognition, we need a labeled dataset. 1.4 We were not able to find a suitable dataset for our purpose; thus after a discussion with the thesis supervisor, we decided to create the dataset ourselves. The modular architecture of IPFIXprobe allows us to write custom processing plugins 1.2.1.1 that can be used for our case of creating a labeled dataset. The plugins described in this chapter are created only for the purpose of creating the dataset and shall not be used further in the prioritization mechanism.

2.1 Design

One of the challenges facing us is the classification of the data. IPFIXprobe does not currently support RTP protocol (and its derivatives) recognition. Therefore we had to design a solution to face this difficulty. The design phase consists of two parts. The first part focuses on creating a general architecture. Following the architectural design of IPFIXprobe modules and the Single Responsibility Principle in software design, two modules have to be created: RTP DPI classifier (shall be further referred as DPI classifier) and RTP DPI exporter (shall be further referred as DPI exporter). The purpose behind the DPI exporter lies in exporting the information from the classifier. Another reason for creating two modules lies in the limitations of the IPFIXprobe system, which usually exports the information once and upon the termination of the flow. This introduces a problem with UDP traffic as it does not have any state, and therefore the flow ends and is exported upon a specific timeout is reached.

2.1.1 Modules design

The two modules are closely intertwined. They run independently, and the processing order is ensured by processing the packets for the DPI classifier in the `pre_update` phase and the DPI exporter in the `post_update` phase (more information about the phases can be found in section 1.2.1.1). The DPI classifier module can be run independently. DPI exporter has a dependency on the DPI classifier module.

2.1.1.1 DPI classifier

The DPI classifier module follows the architecture of the generic IPFIXprobe module. The classification process is split into multiple parts.

The first part is general validation if a UDP packet satisfies the conditions of the RTP packet (RTP header). RTP header has a minimum length of 12 bytes. For simplification, we will

consider only IPv4 packets not having source or destination ports equal to port 53 (DNS). The requirement was set up to avoid overloading the testing system. Further validation validates the *RTP version field* and *RTP payload type*. *RTP version field* must be equal to 2 in accordance with RFC 3550 [24]. *RTP payload type* must not be 72_{10} and 73_{10} as the RFC 3550 [24] specifies. The recommendation set by the RFC 5761 [52] states that *payload type* should not be in the range 64–95. More weak validation steps are defined in RFC 3550 [24] and they relate to packet length validation.

After the validation phase, the verification phase comes. The verification is done per flow (not in bi-flow style as before). The rest of the algorithm can be represented with an extended state machine. The state machine has 3 states:

HEADER_EMPTY initial state

HEADER_MATCHING state representing having potential valid RTP header

HEADER_INITIALIZED final state representing having passed the validation and verification

The initial state is HEADER_EMPTY. After successful validation, the state is changed to HEADER_MATCHING and it saves the RTP header for further verification purposes.

When a packet is received and passes the validation, the further verification process begins. It consists of comparing the *payload type* of the received packet and the saved packet from the previous phase if they match *SSRC fields* are compared, and for *sequence numbers* and *timestamp fields* differences are calculated and then compared with constants. This approach is there to ensure recognition even if some packets are lost during the transmission. If the *payload types* in the verification do not match, then only *SSRC field* is checked. If verification is not successful, the saved RTP header is replaced with the current potential RTP header, and the state stays as HEADER_MATCHING. If the verification is successful, then the saved RTP header is only updated with relevant information (to have the most recent *timestamp* and *sequence* information saved), and the state is changed to HEADER_INITIALIZED.

When a packet is received in state HEADER_INITIALIZED validation and verification occur; if both checks are successful, then the saved RTP header is updated, otherwise, the RTP header is not updated. In both cases, the state HEADER_INITIALIZED remains unchanged.

The extended state machine (see figure 2.1) also includes counters to calculate the number of matched and unmatched possible RTP packets, but it is omitted in the explanation for simplification reasons.

2.1.1.2 DPI exporter

The DPI exporter module also follows the architecture of the generic IPFIXprobe module. The module is distinctive from regular IPFIXprobe modules, as they usually work standalone, but this module is tightly bound to the DPI classifier and won't work without it.

As previously noted, to ensure the correct processing order, the packets are processed in the `post_update` processing phase. The module follows a very simple concept: it stores metadata for the flow and once it reaches the threshold defined by a constant in the code, it flushes the packet metadata into a file. Because plugins can be run on multiple threads simultaneously, the exporter creates a file stream (opening a unique file at */tmp/* location) per thread. Once a threshold of packets is reached, it writes the metadata to the file. Metadata consists of:

- RTP counters
- Last time of communication
- Last time of communication in the direction from $SRC \rightarrow DST$
- Last time of communication in the direction from $DST \rightarrow SRC$

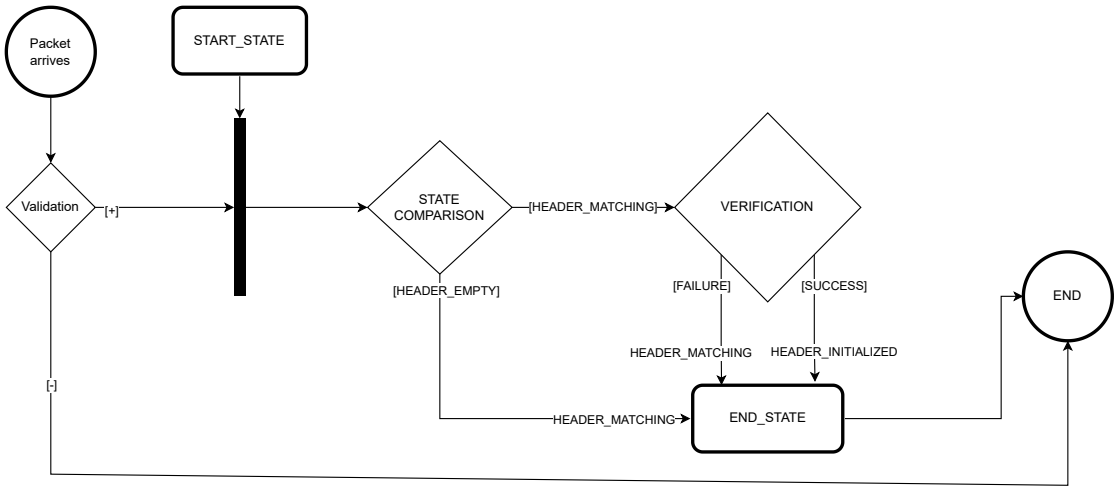


Figure 2.1 Extended state machine diagram for RTP recognition

- Source bytes
- Destination bytes
- Source packets
- Destination packets
- Packet length
- Payload length
- Direction of the current packet

2.2 Implementation

Implementation is split into two main tasks: implementation of the DPI classifier and DPI exporter. Both of the modules follow the traditional architecture of processing plugins in IPFIXprobe. Due to the plugin dependency as mentioned in the chapter above, additional steps need to be taken into consideration while implementing the plugin.

2.2.1 DPI classifier

The module was created following the official guide available on the GitHub page of IPFIXprobe project [13] using a shell script called *create_plugin.sh* located in the *process* folder of the project. Instructions given by the script were followed.

The plugin creates a record, saving all the information needed for the plugin to process the flow in the module. The information includes but is not limited to example bare RTP headers as described in section 1.3.2. The simplified implementation of RTP headers in C++ can be seen in listing 2.1. It also contains RTP counters for both directions, they can be seen in listing 2.2 and a state variable saving the state of the extended state machine described in the design section chapter 2.1.1.1.

Code listing 2.1 Simplified RTP header used for used in DPI classifier module

```
struct rtp_header {
   uint16_t csrc_count : 4;
   uint16_t extension : 1;
   uint16_t padding : 1;
   uint16_t version : 2;
   uint16_t payload_type : 7;
   uint16_t marker : 1;
   uint16_t sequence_number;
   uint32_t timestamp;
   uint32_t ssrc;
};
```

Code listing 2.2 Additional metadata stored for DPI classifier

```
struct rtp_counter {
   uint32_t total_src_after_recognition;
   uint32_t rtp_src;

   uint32_t total_dst_after_recognition;
   uint32_t rtp_dst;
};
```

Implementation closely follows the design principles described in section 2.1.1. It implements the extended state machine described there and updates the state variable in the plugin *record*. It also increments the RTP counters in case the algorithm decides that the processed packet is an RTP packet.

The DPI classifier module has additional methods for validation, verification, and helper methods. The validation and verification methods follow the design. Helper methods mostly help with ETL operations.

2.2.2 DPI exporter

The module was created following the official guide available on the GitHub page of IPFIXprobe project [13] using a shell script called *create_plugin.sh* located in the *process* folder of the project. Instructions given by the scripts were followed.

DPI exporter starts collecting the RTP record metadata called *rtp_exporter_capture_group* at soon as the first packet arrives (can be configured to skip the first n packets via changing macro `RTP_EXPORTER_EXPORT_CAPTURE_GROUP_START` in *rtp.hpp*) and exports the metadata after processing 200 packets (can be configured by changing the define macro in *rtp.hpp* that is called `RTP_EXPORTER_EXPORT_CAPTURE_GROUP_SIZE`). The to-export metadata stores states of flow metadata after each packet is processed, it creates a sequence of states to be exported. The to-export metadata contains the flow key, *rtp_exporter_capture_group*, and additional flow information (such as the number of packets transferred, number of bytes transferred, etc.). The structure of *rtp_exporter_capture_group* can be seen in listing 2.3. The labeling feature RTP_RATIO is calculated as $PACKETS_{RTP\_RECOGNIZED}/PACKETS_{ALL}$ and is exported as well with the flow metadata.

Due to the multi-threaded nature of IPFIXprobe modules, we need to create a separate file stream per thread to export the information. For creating the files we need to set up a unique identifier, we decided to go with 'std::this_thread::get_id()' as it should be unique across all the threads in the current process. The 'this_thread::get_id()' might be reused in case a thread finishes, this situation might arise in IPFIXprobe in two cases: correct shutdown of the application terminates all threads and the second case happens when an exception is thrown or a segmentation fault happens, in that case, the program is not able to continue successfully. The

direction in *rtp_exporter_capture_group* is defined in listing 2.4 — false meaning direction from $SRC \rightarrow DST$ and true otherwise.

Code listing 2.3 RTP captured information exporter additional metadata

```
struct rtp_exporter_capture_group {

   struct rtp_counter rtp_counter;
   struct timeval time_last;
   struct timeval time_last_src;
   struct timeval time_last_dst;

   uint64_t src_bytes;
   uint64_t dst_bytes;
   uint32_t src_packets;
   uint32_t dst_packets;

   uint16_t packet_len;
   uint16_t payload_len;

   bool direction;

   rtp_exporter_capture_group(): time_last_src{0}, time_last_dst{0} {}

} rtp_exporter_capture_group;
```

Code listing 2.4 DPI exporter macros

```
#define RTP_EXPORTER_SOURCE_SRC_TO_DST false
#define RTP_EXPORTER_SOURCE_DST_TO_SRC !(RTP_EXPORTER_SOURCE_SRC_TO_DST)
```

Once the n^{th} (defined by macros as described in the paragraph above) packet is processed, the export starts. The output format is CSV. The exported data is saved to the file opened during the initialization of the module (per thread).

The DPI exporter module follows the design in section 2.1.1. The validation mechanism is weaker compared to the DPI classifier and it is included in the packet handling method. It consists of checking if the flow is IPv4-based, UDP and not running on port 53.

2.3 Data collection

The data collection process is divided into two fragments. The first fragment focuses on data collection on the local network to observe the 'real' communication pattern created by MS Teams, the dataset created shall be referred as *local* traffic dataset. The second fragment focuses on anonymized filtered flow information captured in CESNET [53] network. The dataset created in the CESNET network shall be referred as *core-network* dataset. After the analysis of the *local* traffic dataset, information gained in the process can help tailor the configuration for capturing the core-network dataset. Additionally, the *local* traffic dataset can be used for testing purposes during development (as the dataset contains traffic captured in the process), allowing reproducibility. The *core-network* dataset will be used further for training the machine-learning models.

2.3.1 Local traffic capture

The activity data was captured on the local network by a tool called Wireshark (captured on an interface handling the traffic going to the default gateway) and saved as a PCAPNG file. This will

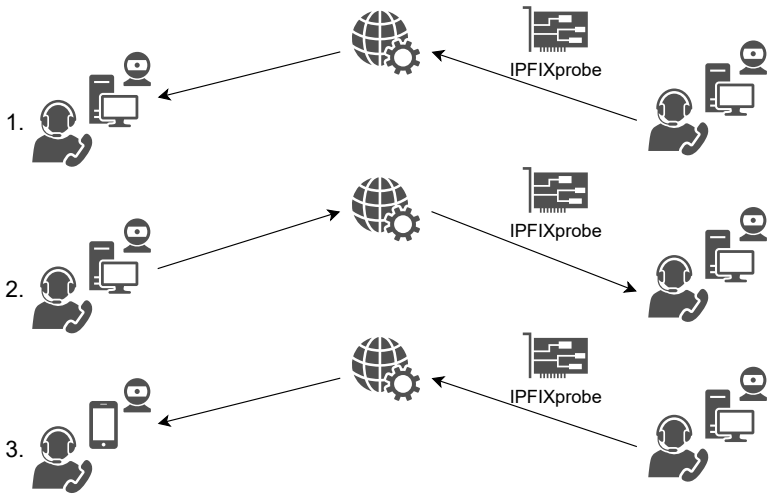


Figure 2.2 General overview of the local capture process

allow further processing either by IPFIXprobe or manual inspection of the traffic. The capture activity was a call via MS Teams. After fifteen seconds in the call, video-sharing was turned on. The whole communication lasted approximately thirty seconds. The capture was conducted three times. The first experiment was conducted on a virtualized machine connected to the main networking stack of the host via NAT, the virtualized machine was running Fedora 38 OS, and the other side receiving the call was running Fedora 38 OS as well. The same experiment was repeated, but the sides of the caller and callee were reversed. The third experiment was conducted on a virtualized machine connected to the main networking stack of the host via NAT, and the other side receiving the call was running iOS 16.0. The following analysis was focused on ports 3478–3481 (more information in section 1.1) as these ports were used by the MS Teams platform. A diagram demonstrating the local capture process can be seen in figure 2.2.

2.3.1.1 Dataset analysis

The next step was an analysis of the PCAPNG files. The very first few packets of the call excluding DNS packets were ICE (see section 1.3.1.2) packets – STUN/TURN. These few packets can theoretically influence the recognition process, therefore the RTP exporter module (see section 2.2.2) supports an option to skip the very first few packets in the export analysis.

2.3.2 Core-network traffic capture

The capture environment, as well as the capturing process, vastly differ compared to the local traffic capture. The capture activity started with IPFIXprobe processing data directly from the interface located at the outer perimeter of the CESNET2 public network. The captured data, including source and destination IP addresses, contained sensitive information (personal information) and therefore they had to be anonymized. Due to the anonymization, labeling the flows based on the IP addresses was not feasible.

Two types of capture were conducted:

- UDP traffic using ports 3478–3481, limited to known IP addresses used by MS Teams
- UDP traffic using ports 3478–3481 with no IP address limitation

The combination of IP addresses, ports, and protocols used are known for MS Teams (see section 1.3.3), these combinations were used for both of the capture types. In the first case,

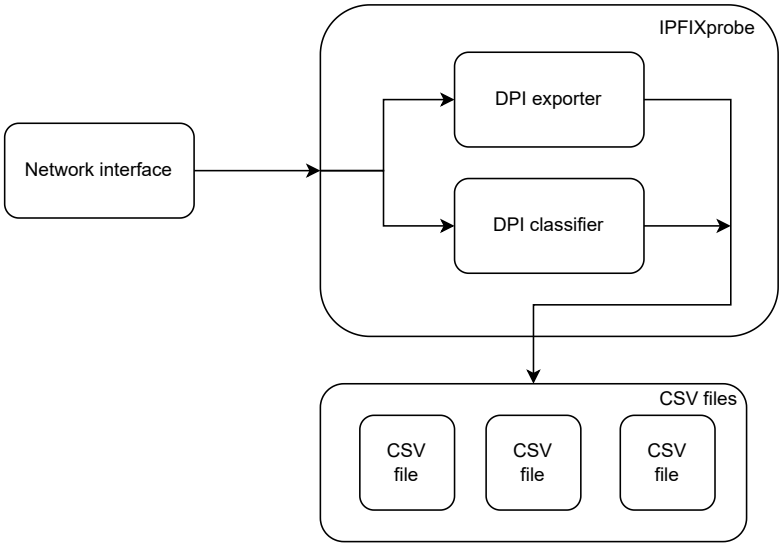


Figure 2.3 General overview of the *capturing* architecture for core-network traffic capture

filters for IP addresses and ports were applied, and in the second case, only filters for ports were applied.

The MS Teams platform prefers direct connection between 'call peers', we can expect most of the traffic directed at MS Teams servers to be relayed over TURN servers. The second capture may contain more direct connections between 'call peers', however, we cannot be sure of the true purpose of the traffic as other protocols can be used. When dealing with a direct connection established via STUN, the ports may differ and therefore can't be part of the planned captures. The second type of capture was limited to ports used by MS Teams to limit the scope of capture, even though it's not as effective as STUN may allocate different port than the ones used in the filter, some implementations of STUN use the destination port as preferred one, if free, and allocate it for the communications.

The process started with filters applied in IPFIXprobe. The data source for IPFIXprobe was a network interface attached to the CESNET public infrastructure. IPFIXprobe had two modules turned on: DPI classifier and DPI exporter. The DPI classifier classified the flow and upon a threshold was reached DPI exporter exported recorded flow data (see section 2.2.2).

2.4 Resulting dataset

The resulting dataset consists of features mentioned in table 2.2. The dataset requires additional preprocessing, which shall be discussed in section 3.2. Upon testing many combinations of thresholds, we decided to set the RTP recognition threshold to 0.3. The table 2.1 shows the total number of recognized RTP flows and the number of packets available for further analysis in the dataset.

Table 2.1 Dataset creation basic dataset analysis

Feature name	Flow count	Packets metadata count
IS_RTP	9 743	1 948 600
IS_NOT_RTP	40 257	8 051 400
Total	**50 000**	**10 000 000**

Table 2.2 Dataset creation feature list

Dataset feature name
time_first.tv_sec
time_first.tv_usec
src_ip
dst_ip
src_port
dst_port
direction
packet_len
payload_len
src_packets
dst_packets
src_bytes
dst_bytes
time_last.tv_sec
time_last.tv_usec
time_last_src.tv_sec
time_last_src.tv_usec
time_last_dst.tv_sec
time_last_dst.tv_usec
rtp_counter.rtp_src
rtp_counter.rtp_dst
rtp_counter.total_src_after_recognition
rtp_counter.total_dst_after_recognition
ratioRtp

Chapter 3
Machine learning design

This chapter describes creation of a machine learning model and creation of an *'adapter'* IPFIXprobe module for machine learning. The first part focuses on data preprocessing, the second part focuses on the machine learning model creation, and the third one on designing and implementation of IPFIXprobe module that will be used as an *'adapter'* for the machine learning model.

3.1 Requirements

The ML RTP recognition model should be able to recognize real-time traffic (RTP traffic) and run in IPFIXprobe environment. The IPFIXprobe project is programmed in C++, so there has to be a synergy between the machine learning model and the IPFIXprobe modules. After a discussion with the supervisor, we have decided to implement the recognition algorithm as a native IPFIXprobe module. This will require porting the model from Python which limits the ML models/algorithms available, porting the model will be further discussed in section 4.1. The ML model must be able to decide the recognition output, so it does not lower network flow throughput of IPFIXprobe modules significantly. The available ML algorithms were discussed with the supervisor and we have decided to choose tree-based algorithms and their boosted alternatives.

3.2 Data preprocessing

The dataset created in chapter 2 contained raw data that required further processing. The dataset containing flow metadata was explored and further processed via an interactive Jupyter Notebook [54] using the Python programming language. Following libraries were used: numpy [55], pandas [56] and matplotlib [57].

The process started with the normalization of the dataset. The dataset features can be divided into two categories:

- Direction-less
- Direction-relevant

The difference between these two categories has been described in section 1.1.2. It is beneficial to distinguish between traffic going from caller to MS Teams server and vice versa for machine learning. The flows are often recognized in the opposite direction than they started in, therefore normalization is needed. The list of all features and their categories can be found in

table 3.1. Normalization of direction requires all direction-relevant features to be switched via their direction counterparts (src with dst and vice versa). The time normalization consists of converting time-related features from format *.tv_sec*, *.tv_usec* and merging them to one variable.

Table 3.1 Features' categories of the raw dataset generated by IPFIXprobe after time normalization

Feature name	Category
time_first	direction-less
src_ip	direction-relevant
dst_ip	direction-relevant
src_port	direction-relevant
dst_port	direction-relevant
packet_len	direction-less
payload_len	direction-less
src_packets	direction-relevant
dst_packets	direction-relevant
src_bytes	direction-relevant
dst_bytes	direction-relevant
time_last	direction-less
time_last_src	direction-relevant
time_last_dst	direction-relevant
rtp_src	direction-relevant
rtp_dst	direction-relevant
total_src_after_recognition	direction-relevant
total_dst_after_recognition	direction-relevant

Upon discussion with the thesis supervisor, we decided to enrich the dataset with biproducts of the data records (flow metadata) and also add statistical features describing the flow as a whole. These statistical features can offer more insight into communication and can be used further for machine learning as additional features. The list of enriched features, including the added statistical ones can be seen in table 3.2.

The *dst* or *src* in the name of a feature represents the direction of flow, the source is set up as a caller and the destination as MS Teams server. The *avg* represents an average for the flow metadata, *std* represents a standard deviation, *time_diff* represents a difference between the two last consecutive records, *ratios* are calculated as the $feature_{src}/feature_{dst}$ and *rates* as $feature/duration$. Time duration is calculated as $time_{last} - time_{first}$. Packets and bytes as a sum of $feature_{src} + feature_{dst}$. Upon discussion with the supervisor, we decided to normalize *rtpRatio* to a boolean value called *is_rtp*. The decision boundary has been set to 30%. The only flow-key information preserved is *dst_port* as it represents the service used (more in section 1.3.3).

3.2.1 Basic dataset analysis

As the figure 3.1 shows, there is a large imbalance between RTP-classified traffic and other traffic. The ML model used must take the imbalance into consideration and work with it. As figure 3.1 states the ratio of RTP recognized traffic with non-RTP traffic is approximately 1:4. We can also look at the RTP ratios from the *dst_port* perspective for ports 3478, 3479 (see figure 3.2) and ports 3480 and 3481 (see figure 3.3). The ports usage in MS Teams can be found in section 1.3.3.

3.2.2 Feature selection

"*Feature selection (...) has been proven to be effective and efficient in preparing data (especially high-dimensional data) for various data-mining and machine-learning problems. The objectives of feature selection include building simpler and more comprehensible models, improving data-mining performance, and preparing clean, understandable data.*" [58]

Table 3.2 List of all the features created by the preprocessing and their status (enriched)

Feature name
time_duration
src_bitrate
dst_bitrate
bitrate
src_packet_rate
dst_packet_rate
packet_rate
packet_ratio
bytes_ratio
time_src_diff_avg
time_dst_diff_avg
time_diff_avg
time_src_diff_std
time_dst_diff_std
time_diff_std
src_bytes
dst_bytes
bytes
src_bytes_per_packet_avg
dst_bytes_per_packet_avg
bytes_per_packet_avg
src_bytes_per_packet_std
dst_bytes_per_packet_std
bytes_per_packet_std
src_packets
dst_packets
packets
packet_len
dst_port
is_rtp

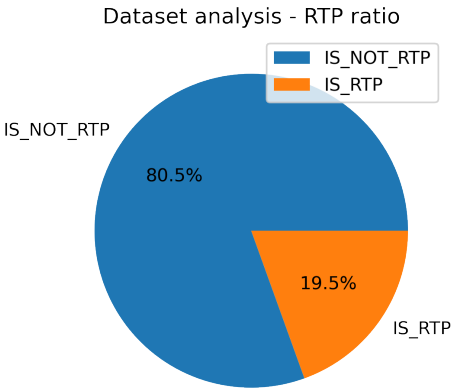
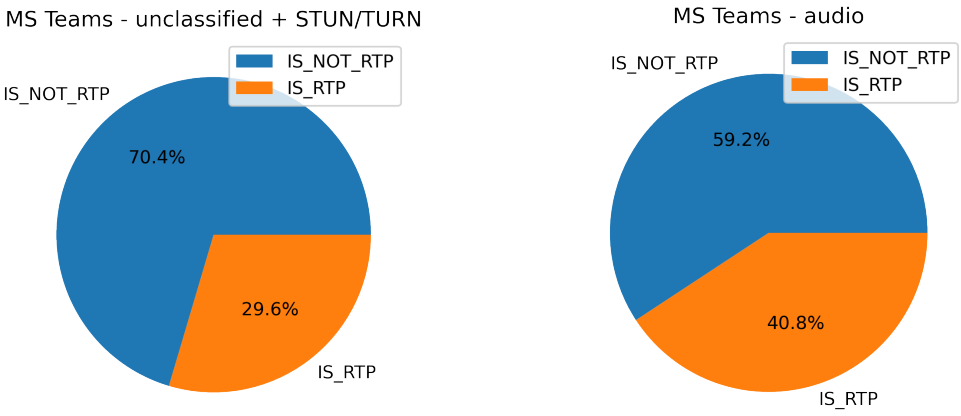


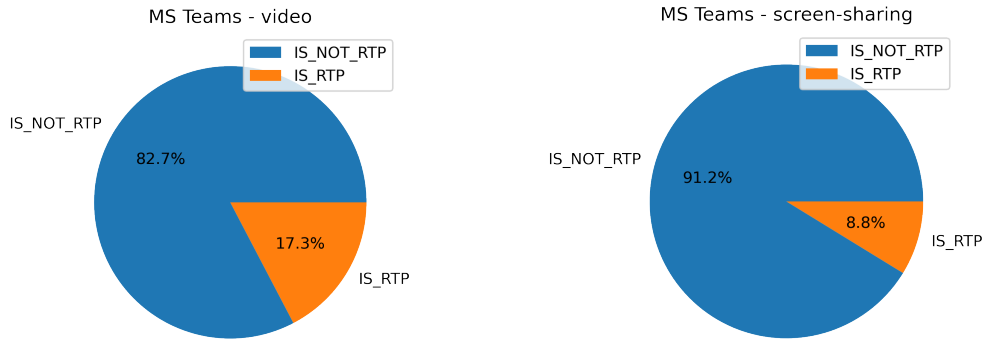
Figure 3.1 Ratio of RTP traffic to non-RTP traffic



(a) RTP ratio for port 3478

(b) RTP ratio for port 3479

Figure 3.2 Ratio of RTP traffic to non-RTP traffic for port 3478 and 3479



(a) RTP ratio for port 3480

(b) RTP ratio for port 3481

Figure 3.3 Ratio of RTP traffic to non-RTP traffic for port 3480 and 3481

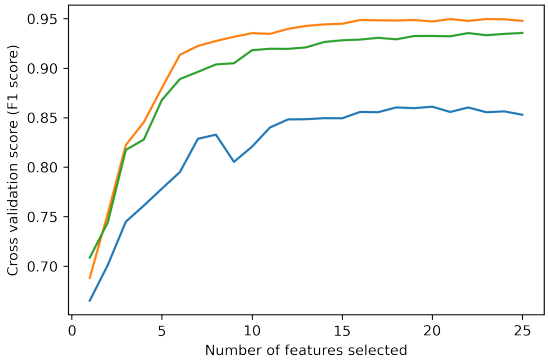


Figure 3.4 Visualisation of RFECV and min-features taken (the three lines represent cross-validation scores)

Feature selection is closely intertwined with the machine learning algorithm used. We used the '*RFECV*' with '*RandomForestClassifier*' (with these parameters class_weight='balanced', n_estimators=100, max_depth=20), the scoring mechanism chosen was F1 score and as cross-validation algorithm has been chosen '*StratifiedKFold(3)*'. The selected feature set has a recommendation character as other modifications to the dataset might be needed, especially if another classifier is used.

These features have been excluded from the training:

- dst_port
- src_packets
- packets
- dst_packets

The optimal amount of features evaluated by the '*RFECV*' was 20 features. These features were chosen to be excluded by the feature selection algorithm:

- time_src_diff_avg
- src_bytes
- time_dst_diff_avg
- time_diff_avg
- bytes

3.3 Machine learning methods

After compiling the datasets with preprocessed features, a machine learning algorithm shall be used to categorize the traffic. The ML model can be either supervised (labeled datasets) or unsupervised (more information in section 1.4). This thesis focuses on supervised forms. The compiled dataset has been already labeled. Several machine learning algorithms were chosen to achieve the classification task, consisting of Decision Tree, Random Forest, AdaBoost, and XGBoost.

To facilitate the ML process, all the machine learning models are trained and evaluated in Jupyter Notebook in Python, using these libraries: numpy [55], pandas [56], matplotlib [57], scikit-learn [59] and XGBoost library [47].

3.3.1 Approach

There are two main approaches, building the data model for each type of traffic: voice, video, and screen-sharing, or building one '*fits-all*' model. Both these options have their benefits and drawbacks.

The *one-type-one-model* is able to classify the traffic in more precise way as the traffic characteristics are more consistent, nonetheless, it requires a ML model per type of traffic. Having multiple models turned on can degrade the performance of the classification system. Another disadvantage related to our work is possible misclassification of other voice protocols as this paper focused only on RTP and other voice protocols can have different characteristics.

Contrary the *one-model-fits-all* is more robust in processing multiple types of traffic as the models were built to recognize the real-time traffic patterns rather than the type of traffic.

After evaluating the benefits and drawbacks of both models, we decided to select the *one-model-fits-all*.

3.3.2 Model design

As discussed in section 2.2.2, the created dataset consists of 'history' chunks of flow analysis enriched with additional (mainly statistical) features. We can train the machine learning model on the whole dataset or on smaller chunks (representing the state after n^{th} packet is processed). In order to emphasize the 'stable' (in terms of not changing throughout the communication) features of the communication, we decided to implement the strategy of using the whole dataset.

The dataset will be split into training and testing parts with stratify option enabled. Stratify ensures that the split dataset follows the distribution of the values in the core dataset. The ratio for training and splitting the dataset was after a few experiments set to 80%. The method used for splitting the dataset was *train_test_split* available in scikit-learn library.

In order to simplify machine learning model retraining we included an option to exclude certain columns from the model training. The training paradigm includes comparisons of ML models with multiple metrics in mind, however for training one main metric must be chosen. After proper evaluation of the metrics suitable for our cause and their availability in the machine learning framework used we chose F1 score. It is calculated as $\frac{2*TP}{2*TP+FP+FN}$, where TP, FP, and FN are part of a confusion matrix (see figure 3.5).

Each of the models offers multiple hyperparameters, refining the machine learning model. The most popular ones for tree-based ML algorithms include *maximum depth of a tree* and *number of estimators* for ensemble algorithms. The tree-based algorithms in scikit-learn (Decision Tree, Random Forest, and XGBoost) also offer *class_weight* hyperparameter, which is mainly used to balance the dataset in relation to the algorithm.

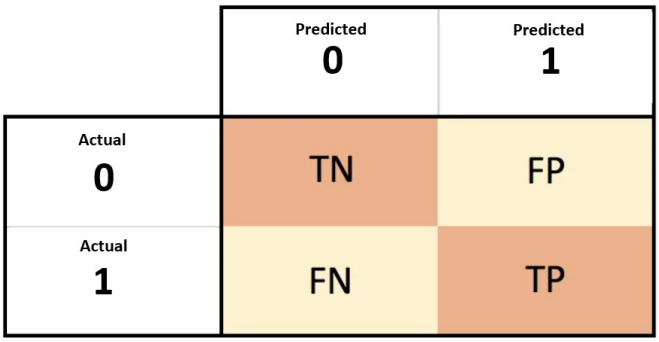


Figure 3.5 Confusion matrix explanation, from [60]

3.3.3 Model training

The ML model training process followed the design principles (see section 3.3.2). For testing various hyperparameters, we chose to use *GridSearchCV* feature with the combination of scikit-learn's *Pipelines*. It trains the ML model testing all the combinations of hyperparameters, evaluates them, and returns the best model. The combinations of hyperparameters tested can be seen in figure 3.1.

Code listing 3.1 Python dictionary showing hyperparameters used for model training

```
training_methods = {
    'decision_tree': {
        'pipeline': Pipeline([
            ("classifier", tree.DecisionTreeClassifier(
                class_weight='balanced'))
        ]),
        'params' : {
            'classifier__max_depth': range(5, 50, 5)
        },
        'jobs':max_jobs,
        'jobs_method':1,
        'metric': metric_default,
    },
    'random_forest': {
        'pipeline': Pipeline([
            ("classifier", ens.RandomForestClassifier(
                class_weight='balanced'))
        ]),
        'params': {
            'classifier__n_estimators': range(100, 225, 25),
            'classifier__max_depth': range(5, 25, 5)
        },
        'jobs': max_jobs,
        'metric': metric_default,
    },
    'adaboost': {
        'pipeline': Pipeline([
            ("classifier", ens.AdaBoostClassifier(
                tree.DecisionTreeClassifier(max_depth=20,
                class_weight='balanced',algorithm='SAMME')
            ))
        ]),
```

```
        'params' : {
            'classifier__n_estimators': range(100, 225, 25),
        },
        'jobs':max_jobs,
        'metric': 'f1',
    },
    'xgboost': {
        'pipeline': Pipeline([
            ("classifier", xgb.XGBClassifier(tree_method='hist'))
        ]),
        'params' : {
            'classifier__n_estimators': range(100, 225, 25),
            'classifier__max_depth': range(5, 25, 5)
        },
        'jobs':max_jobs,
        'metric': 'f1',
    }
```

The next part of the model evaluation was manual, it focused on comparing the models. The tree ensemble methods based on bagging (see section 1.4.2.2) heavily relied on a few features, having high feature importance. The decision made by the ML algorithm primarily depended on a small set of features, that were relevant to certain types of traffic but did not necessarily have to be relevant for all real-time traffic.

Following the design decisions (see section 3.1), the machine learning shall be run in the IPFIXprobe ecosystem. The ML model must be ported to C++ (see section 4.1.1), to be able to be integrated with IPFIXprobe. Therefore the model testing comprises of an additional step – porting the model into C++ and integrating it into the IPFIXprobe subsystem. The models trained using the hyperparameters defined in listing 3.1 were exported via m2cgen utility (further described in section 4.1.1). The models were too big to be considered (hundreds of MBs of C++ code). We tried integrating them into the IPFIXprobe subsystem, but we were met with very long compilation times and in some cases compilation failures. Another aspect of the ML model is its size, our ML recognition subsystem should not significantly increase the memory footprint of the IPFIXprobe system. Upon discussion with the thesis supervisor, we decide to limit the number of all nodes (in decision-tree-based algorithms) to 2^{15}. The internal nodes shall be translated into IF conditions (see section 1.4.1). This imposes additional limits for hyperparameters. In order to utilize the advantages of tree ensembles, the number of estimators must be higher than one. The perfect binary tree (representing the maximum possible size of a decision tree) has $2^{h+1} - 1$ nodes [61], where h is equal to the depth of the tree. Upon thorough manual testing, the most performant combination of hyperparameters was chosen – the maximum number of estimators was set to 100 and the maximum depth to 7. For the decision tree, the limits imposed are related only to the maximum depth and are equal to 14.

The training followed the same settings in terms of hyperparameters except for the ones explicitly mentioned in this section. The model's statistical information can be seen in table 3.3. The two best models according to F1 scores were Decision Tree and XGBoost, upon comparison of F1 scores and confusion matrixes, we decided to choose XGBoost as the to-go model. The comparison of the models' performance after n packets were processed can be seen in figure 3.6.

3.3.4 Final model

The chosen model upon analysis is the XGBoost-based model. The feature significance can be found in table 3.4. The model's performance analysis over the first n^{th} packets can be seen in figure 3.6.

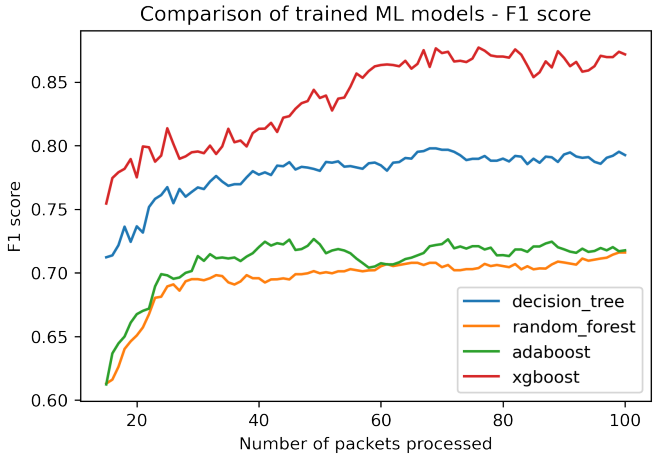


Figure 3.6 Comparison of trained ML models – limited hyperparameters

Table 3.3 Comparison of machine learning algorithms test scores – smaller models

ML model	#E.	M.D.	F1	CM	
Decision Tree	1	14	0.847	641727	50676
				7 109	160 471
Random Forest	100	7	0.775	618 533	73 870
				14 756	152 824
AdaBoost	100	7	0.793	628 975	63 428
				15 653	151 927
XGBoost	100	7	0.891	676 166	16 237
				19 876	147 704

Table 3.4 Feature significance of XGBoost selected model

Feature name	Feature significance
dst_packet_rate	0.590
packet_rate	0.058
dst_bitrate	0.058
dst_bytes_per_packet_std	0.057
src_packet_rate	0.037
bytes_per_packet_std	0.033
dst_bytes_per_packet_avg	0.033
bitrate	0.025
bytes_ratio	0.024
time_diff_std	0.024
bytes_per_packet_avg	0.019
src_bytes_per_packet_std	0.015
src_bytes_per_packet_avg	0.015
time_src_diff_std	0.011

3.4 IPFIXprobe module

3.4.1 Data preprocessing

The data generated by IPFIXprobe have been processed outside of the IPFIXprobe ecosystem, however as the ML shall have a form of an IPFIXprobe module (see section 3.1), the preprocessing has to be also moved to the module. This introduces multiple challenges, some of the calculations need to be moved to on-the-fly calculations. On top of standard mathematical calculations, such as addition, division, etc., we need to find on-the-fly algorithms for the calculation/approximation of standard deviation. For running calculation/approximation of standard deviation, we can use Knuth's or Welford's online algorithm. Upon testing the algorithms, we decided to use Welford's online algorithm.

The additional flow metadata required for calculations are mostly time-based ones: *time_first*, *time_src_last*, *time_dst_last*, *time_last*. For the listed statistical features, we need Welford's calculators: time_src_diff, time_dst_diff, time_diff, src_bytes_per_packet, dst_bytes_per_packet, and bytes_per_packet.

3.4.2 Module structure

The module shall be divided into two logical parts. The first part shall be responsible for processing the flow metadata and calculation of statistics-based features. It shall also take care of storing all necessary flow metadata required for the calculations. The second part shall be responsible for handling model evaluation and communication between IPFIXprobe and OVS.

3.4.3 Module implementation

The module consists of two parts. After IPFIXprobe recognized a new flow, the `post_create` function is called. It creates a record for storing metadata for the flow. In our case, the record contains the metadata required for calculations. After initialization of the record, we call a method `manage_packet` that performs basic validation of the packet (could be RTP or not). The validation consists of checking if the used protocol is UDP and if the IP stack used is version 4, IPv6 cannot be considered due to its unavailability in the dataset. It also filters out port 53/UDP as it is mostly used for DNS. After passing the validation, metadata are updated. The evaluation of the machine learning model is set up to happen every 25 packets (if validation is passed). The number of packets can be changed by changing a macro *ANALYSIS_STEP*. The evaluation of the ML model can be delayed by changing macro *ANALYSIS_THRESHOLD*. The default value is set to 25 for both.

The ML vector is represented by structure `rtp_ml_vector` (can be seen in listing 3.2), it also contains a function `predict` that returns true only if the machine learning model *decides* that the flow is RTP based on the features. The `rtp_ml_vector` necessarily does not need to contain all available features, as it might use only a small part of them. We included all the features in the `rtp_ml_vector` vector, as it might be easier to replace the model without needing to modify the IPFIXprobe module responsible for detection. The evaluation model is located in the *process/rtp-ml-model.hpp*.

Code listing 3.2 C++ ML vector

```
struct rtp_ml_vector {
  double time_duration;
  double src_bitrate;
  double dst_bitrate;
  double bitrate;
  double src_packet_rate;
  double dst_packet_rate;
  double packet_rate;
  double packet_ratio;
```

```
        double bytes_ratio;
        double time_src_diff_avg;
        double time_dst_diff_avg;
        double time_diff_avg;
        double time_src_diff_std;
        double time_dst_diff_std;
        double time_diff_std;
        double src_bytes;
        double dst_bytes;
        double bytes;
        double src_bytes_per_packet_avg;
        double dst_bytes_per_packet_avg;
        double bytes_per_packet_avg;
        double src_bytes_per_packet_std;
        double dst_bytes_per_packet_std;
        double bytes_per_packet_std;
        double packet_len;

        bool predict();
    };
```

The evaluation of the model is run twice; it is caused by not having the direction normalized (see section 3.2). If any of the runs are evaluated as true, the flow is marked as RTP and sends a command via the `system` function to instruct OVS to prioritize the flow.

Chapter 4

Implementation of traffic prioritization

This chapter focuses on transferring the ML model trained in chapter 3 and setting up the prioritization subsystem using OVS. It also includes a part where a PoC is designed, implemented, and tested.

4.1 Machine learning model transfer to lower-level language

4.1.1 Design

As previously discussed (see section 3.1), the model will need to run as a native IPFIXprobe module classifying the data. This imposes a challenge of porting the trained ML models. The scikit-learn library offers a list of related packages, which contains several packages focused on exporting ML models to other languages. We are most interested in porting to C/C++. The listed options include sklearn-porter [62], m2cgen [63] and treelite [64]. Sklearn-porter only supports porting of Decision Trees, whereas m2cgen supports porting Decision Trees, tree ensembles such as Random Forests, and XG-boosted trees. Treelite introduces an additional library dependency to the IPFIXprobe stack, which violates the requirements set in the design phase (see section 3.1) and therefore it cannot be considered. Upon evaluation of available options with the requirements set, the most suitable module for our use-case is m2cgen.

4.1.2 Implementation

The m2cgen library needs a *fitted* model, the export command can be found in listing 4.1. It generates a function called `score` with declaration available in listing 4.2. It must be re-adjusted to fit the interface of the program. The *input*, in our case pointer, is to an array of doubles (representing the features), and the *output* is a pointer to an array of 2 doubles. The memory must be allocated before running the function. The list of input variables follows the order set up in design, with the exception of: *dst_port, packets, src_packets, dst_packets.* The *output* array's first value is between $< 0, 1 >$ inclusive. It represents the probability of not being RTP. The *output* array's second value represents the probability of being RTP. The probability of being RTP is $1 - value_{first}$.

Code listing 4.1 Python code for model conversion to C language

```
# classifier must be supported and fitted
code = m2c.export_to_c(classifier)
```

Code listing 4.2 Generated scoring function by m2cgen

```
void score(double * input, double * output);
```

Code listing 4.3 C++ interface for ML prediction

```
bool rtp_ml_vector::predict();
```

The m2cgen generates a C/C++ code (methods). It converts all features to doubles. As shown in figure 4.2, the method signature does not match the standard IPFIXprobe code interface and therefore has to be remade. Another challenge is the usage of '*memcpy*' instead of assignment. We prepared a Python script converting the most essential features to the IPFIXprobe format. It changes the method signature to '*bool rtp_ml_vector::predict()*', and converts all *input[x]* variables to the proper feature fields. It also replaces '*memcpy*' with regular assignments. The script does not currently do all the necessary work, but rather the most repetitive. We have to insert the method into the template. The process shall be described in the README.md file located next to the script. The *rtp_ml_vector* structure contains all the features (see section 3.2) converted to double and a function 'predict' returning true if the vector has been classified as real-time traffic (as RTP) or false if not.

4.2 Traffic prioritization via OVS

Open vSwitch provides numerous methods (see section 1.5.2.1) for traffic prioritization. One of the methods for prioritization is creating a QoS subsystem in OVS and creating multiple queues with `min-rate` and `max-rate` parameters. After creating and assigning the queues to interfaces in OVS subsystem, a network administrator is required to create a set of rules assigning the network flows into the proper queues, thus realizing the prioritization. The `min-rate` parameter can ensure guaranteed minimal packet bandwidth in terms of packet scheduling. The `max-rate` parameter limits the maximum speed of a flow.

The prioritization follows the design principles described in section 4.2. First, we create an OVS bridge system. Our bridge shall be called br0. Our input interface shall be called eth0 and the output interface eth1.

Commands for creating a bridge and adding the interfaces can be seen in figure 4.4. OVS QoS system creation can be seen in figure 4.5. We introduced multiple parameters to artificially limit the speeds to be able to test it (more information about the design can be found in section 4.2 and about the parameters and their influence in section 1.5.2.1). The QoS subsystem must be assigned to the output interface.

Code listing 4.4 Initialization of OVS

```
ovs-vsctl add-br br0 # create OVS bridge
ovs-vsctl add-port br0 eth0 # add eth0 interface to the bridge

ip link set br0 up # bring up the OVS bridge interface
```

Code listing 4.5 OVS QoS system set-up

```
ovs-vsctl -- \
  --id=@newqos create qos type=linux-htb \
      queues=0=@q0,1=@q1 -- \
```

```
    --id=@q0 create queue \
      other-config:max-rate=5000000 \
      other-config:min-rate=5000000  -- \
    --id=@q1 create queue \
      other-config:max-rate=20000000 \
      other-config:min-rate=20000000 \
    -- set port br0 qos=@newqos
```

After creating the QoS system, we need to add flow rules to define how the flows should be treated. First, we remove all the flow rules in the table, as there might be some remnants of the default configuration. After that, we add the rules and we should not forget about adding the default handler. Figure 4.6 shows the process with adding one bi-flow rule (two uni-flows rules) stating that traffic transferred via UDP protocol coming from IP address 1.2.3.4 to IP address 4.3.2.1 from port 1234 to port 4321 and vice versa (direction-wise) should be prioritized. The priority of the rule has to be higher than the default priority value 32 768 (more in section 1.5.2.1).

Code listing 4.6 OVS flows assignment to queues

```
ovs-ofctl del-flows br0 # remove all previous flow rules in br0
# ...

# add flow rule
ovs-ofctl add-flow br0 \
 priority=40000,ip,nw_src=1.2.3.4,nw_dst=4.3.2.1,
 udp,udp_src=1234,udp_dst=4321,actions=set_queue:1,normal

# add reversed flow rule
ovs-ofctl add-flow br0 \
 priority=40000,ip,nw_dst=1.2.3.4,nw_src=4.3.2.1,
 udp,udp_dst=1234,udp_src=4321,actions=set_queue:1,normal

# ...
ovs-ofctl add-flow br0 \
 actions=set_queue:0,normal # add default action
```

4.3 Proof of Concept demonstration

4.3.1 Design

The Proof of Concept consists of a concrete test of the created ML model. The test shall have multiple phases, first phase is focused on testing the model on PCAPNG files and comparing the results of the ML model with the manual observation of the PCAPNG files (captured during the dataset-creation phase – see section 2.3.1). This part however cannot test the prioritization part of the OVS.

The second part consists of running the ML model on a Turris router [65] having installed OVS and IPFIXprobe with our model recognition incorporated. We shall connect our device to the network running the OVS set up for prioritization and IPFIXprobe with the ML model.

4.3.2 Implementation

The implementation followed the design principles defined in section 4.3.1. The first phase consisted of testing the ML model on PCAPNG files.

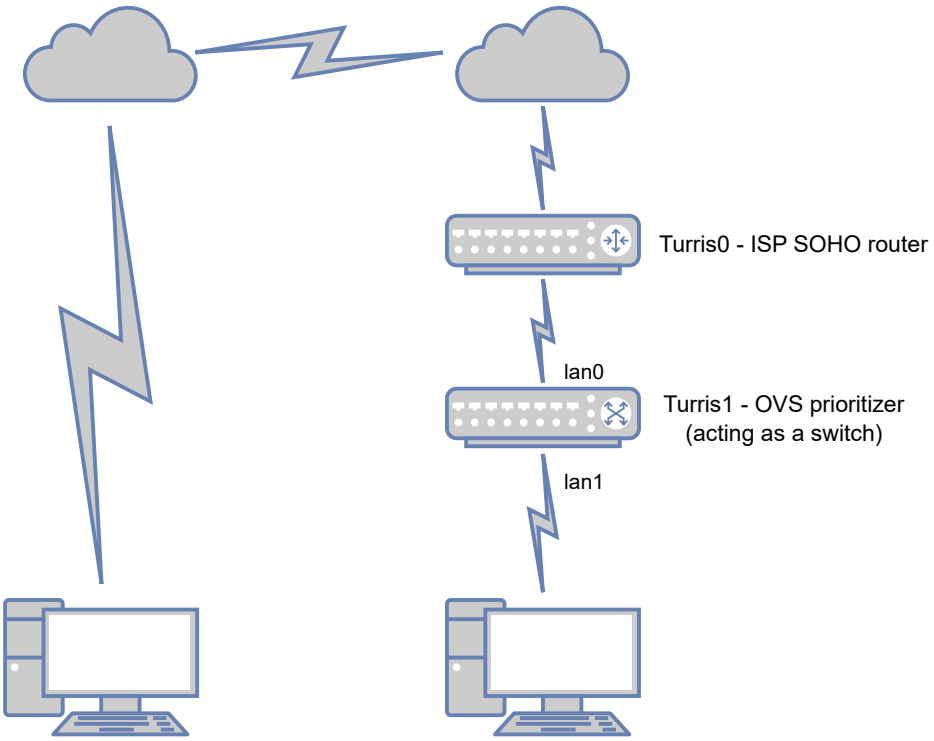


Figure 4.1 Simplified diagram of the PoC architecture

The second phase consisted of testing the ML model on physical hardware. The architecture of the network stack consisted of two Turris Omegas. First Turris was acting as a regular ISP-provided SOHO router performing NAT and providing DHCP and DNS services on the LAN interfaces. The IP range used for testing was 192.168.0.0/24. The second Turris had the default LAN bridge disassembled. The *lan0* port was connected to the first Turris. The *lan1* port was connected to a testing laptop running on Fedora 38 OS.

On the second Turris, we set up the OVS following the guide in section 4.2 with a few adjustments. The adjustments included add-port commands (adjusted to our use-case: *lan0* and *lan1*). The second adjustment was related to queues, we designed `q0` as the default one with `max-rate` limit set to 1Mbps. The `q1` queue had a limit set up to 20Mbps. The QoS subsystem was assigned to out-interface: *lan0*. The default flow rule assigned all the traffic to the limited queue (`q0`). When the IPFIXprobe ML module recognized real-time traffic (RTP traffic in our case), it triggers an invoke of `system` function and sends the prioritization command to the OVS via `ovs-ofctl`. The command structure can be seen in section 4.2. The IP address and ports have to be modified. The simplified diagram can be seen in figure 4.1.

Chapter 5
Evaluation

5.1 Model evaluation

In chapter 3 we tested multiple machine learning models, then designed a solution integrating the model into the IPFIXprobe ecosystem.

Figure 5.1 shows an improving trend for the model's performance – the more packets we process, the more precise the detection gets. The first few packets might not be RTP, but rather TURN and STUN (see section 2.3.1). The RTP exporter can be set up to skip the very first few packets.

The improving trend can be also seen in figure 5.2, which shows the number of False Positives and False Negatives (visual explanation can be seen in figure 3.5) regarding the number of packets processed in the communication.

5.2 Model transfer evaluation

As mentioned in sections 4.1.1, and 4.1.2, the exported model is not the same model as we trained in Python, all the features were converted to doubles. It can introduce a potential challenge as it can affect the model's accuracy. Based on our observations the model's performance has not changed significantly.

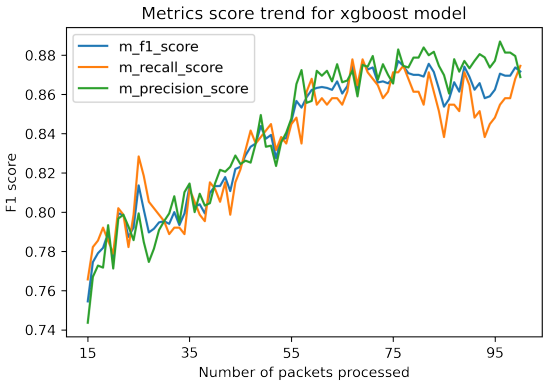


Figure 5.1 General stats of the chosen ML model

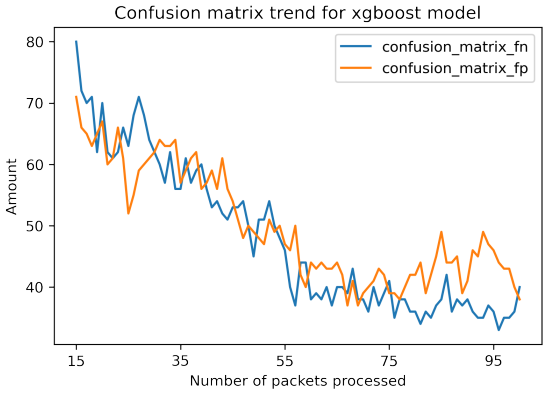


Figure 5.2 Confusion matrix - trend of FP and FN of the chosen ML model

5.3 Proof of Concept demonstration evaluation

5.3.1 Evaluation of PCAPNG files

The ML was also tested on the *local-traffic* dataset consisting of PCAPNG files. IPFIXprobe has the option to input data from PCAP-like files rather than connecting to an interface. We used that option and processed our captures, all of them were recognized as real-time traffic (RTP).

5.3.2 Model evaluation on SOHO network

The ML model was also tested on the local network. The experiment was running on Fedora 38 and ran the trained ML model. The expected outcome was to prioritize the calls. We simulated the environment of a throttled network by having two queues (see section 4.2), one very slow (simulating congested network), and one fast (prioritized). The experiment consisted of multiple calls, initiated by different parties. Video-sharing was turned on after 15s. The experiment was successful and all the calls were recognized. We also simulated an outage of connection and the ML module was able to re-detect it (prioritize it) in the middle of the connection.

5.4 Performance statistics

Based on the design decision (see section 4.3.1) and implementation choices (see section 4.3.2) made, the project is not ready to be used in production environment. It serves as a demonstration of the project and shall be run in a testing environment. The communication between IPFIXprobe and OVS is implemented as C++ `system` calls, which might affect the overall performance of the whole IPFIXprobe subsystem.

Chapter 6
Conclusion

Real-time communication is a crucial element in many organizations across the world, more emphasized by the trend of home office and globalism. The global trend in communication platforms is offering their services via the Internet. It introduces a potential challenge as the receiving organizations must ensure that their connection to the Internet is able to handle all the traffic produced by the organization. This proves to be a difficult task as a bottleneck can be introduced by the environment.

Traffic prioritization can help partially mitigate the negative effect on the organization by prioritizing the traffic crucial to the organization. The standard approach for traffic prioritization is based on rules defined by a combination of source and destination IP addresses, protocols, and ports. This proves to be inefficient as these communication platforms are often hosted in clouds, and IP addresses and ports can change from minute to minute. Cloud-based services often share the very same IP, therefore prioritization based solely on IP addresses is inefficient. All said related to prioritization stands only if the communication platform provider guarantees that the combination of IP addresses and/or ports is only used by specific services.

This thesis was focused on designing, implementing (Proof of Concept), testing, and evaluating a solution developed for the purpose of real-time traffic prioritization combined with machine learning and Open vSwitch (OVS). Traffic recognition focuses on RTP-based protocols as they are prevalent protocols used for many popular communication platforms such as Microsoft Teams, Zoom, Discord, etc.

As no relevant public datasets of real-time traffic have been found in the research part, dataset creation has been a fundamental part of this thesis. It consisted of the creation of new IPFIXprobe modules, an RTP classifier based on deep-packet-inspection, and an exporting module. The created dataset contained several types of traffic, ranging from audio, video, screen-sharing, etc. It contained information about approximately 50 000 flows (with history, it consisted of circa 10 million records). Traffic recognition focuses on RTP-based protocols, which are prevalent protocols used for many popular communication platforms such as Microsoft Teams, Zoom, Discord, etc.

Various machine learning algorithms were tested, ranging from simple machine learning algorithms to more advanced ones. The requirement of running the final machine learning model inside of IPFIXprobe introduced several challenges. They had to be taken into account during the learning phase. The results of the machine learning were promising as some of the models reached F1 scores around 90%. The most accurate machine learning models suitable for usage in the IPFIXprobe ecosystem included a Decision Tree and a XG-boosted tree ensemble. The chosen ML model was XG-boosted ensemble due to its performance. The machine learning model has been comprehensively evaluated.

Transferring the model into IPFIXprobe ecosystem proved not to be an easy task, we re-

searched several open-source projects focused on porting machine-learning models to the native language of IPFIXprobe. The most promising one was m2cgen, which was able to convert some scikit-learn models into the native language of IPFIXprobe. An 'adapter' IPFIXprobe module capable of hosting the machine learning model had to be designed and implemented.

The Proof Of Concept testing consisted of setting up a real-life network environment using Turris routers (capable of running OVS and IPFIXprobe) and integrating our Proof of Concept prioritization solution into them. The solution was tested by conducting a video call via the MS Teams platform with an artificially speed-throttled environment. The proposed solution was able to detect the video and prioritize the call-related traffic. Artificially created disruptions were introduced to the network to test the model stability in a network-unstable environment; the solution was able to handle the situation and prioritize the relevant traffic. The created solution was thoroughly tested. Overall, all points of the thesis assignment were fulfilled.

6.1 Future work

Future work could include extending the training dataset with additional protocols and retraining the models and comparing the accuracy of such models. The implementation was programmed only for demonstration purposes and is not production-ready. In making the system usable for production use, further testing is required. The prioritization subsystem can be only deployed in small networks, it is not suitable for medium and large networks. The design and implementation of the modules responsible for the prioritization were not aimed to be production ready as that would impose additional complexity into the project, f.e. communication between the module responsible for the evaluation of flow traffic and OVS was set up in a demonstrative way, in larger networks the overhead caused by the design choice could outweigh the benefits. Redesigning the communication between the machine-learning flow evaluation module and OVS would be necessary for deployment to larger networks.

Appendix A

IPFIXprobe integration guide

A.1 Compilation

1. Follow the IPFIXprobe compilation guide on GitHub [13]
   a. In GIT clone phase: run this command instead of the recommended one in the guide
      i. Run: `git clone --branch rtp_simekst --recurse-submodules https://github.com/CESNET/ipfixprobe`
      ii. Replace these files `local_path` with `remote_path`
      - `source/ipfixprobe-module/process/rtp.hpp` with `ipfixprobe/process/rtp.hpp`
      - `source/ipfixprobe-module/process/rtp.cpp` with `ipfixprobe/process/rtp.cpp`
      - `source/ipfixprobe-module/process/rtp-exporter.hpp` with `ipfixprobe/process/rtp-exporter.hpp`
      - `source/ipfixprobe-module/process/rtp-exporter.cpp` with `ipfixprobe/procertp-exporter.cpp`
      - `source/ipfixprobe-module/process/rtp-ml.hpp` with `ipfixprobe/process/rtp-ml.hpp`
      - `source/ipfixprobe-module/process/rtp-ml.cpp` with `ipfixprobe/process/rtp-ml.cpp`
      - `source/ipfixprobe-module/process/rtp-ml-model.hpp` with `ipfixprobe/process/rtp-ml-model.hpp`
      iii. Update Makefile.am in *ipfixprobe* folder
      A. Add these lines to `ipfixprobe_process_src`
      - ```
        process/rtp-ml.hpp \
        process/rtp-ml.cpp \
        process/rtp-ml-model.hpp
        ```
      - The result should look like this:
        ```
        ipfixprobe_process_src=\
            process/http.cpp \
            process/http.hpp \
            process/rtsp.cpp \
            process/rtsp.hpp \
        ```
        ```

```
    process/sip.cpp \
    ...
    process/rtp.hpp \
    process/rtp.cpp \
    process/rtp-exporter.hpp \
    process/rtp-exporter.cpp \
    process/common.hpp \
    process/rtp-ml.hpp \
    process/rtp-ml.cpp \
    process/rtp-ml-model.hpp
```

B. Add these lines to `ipfixprobe_headers_src`

```
include/ipfixprobe/stats-welford.cpp \
include/ipfixprobe/stats-ml.cpp
```

- The result should look like this:

```
ipfixprobe_headers_src=\
    include/ipfixprobe/plugin.hpp \
    include/ipfixprobe/input.hpp \
    include/ipfixprobe/storage.hpp \
    ...
    include/ipfixprobe/byte-utils.hpp \
    include/ipfixprobe/ipfix-elements.hpp \
    include/ipfixprobe/stats-welford.cpp \
    include/ipfixprobe/stats-ml.cpp
```

2. Follow the rest of the guide on the GitHub

A.2 Usage

- PCAP options enabled
  - Run on the network interface

```
./ipfixprobe -i 'pcap;ifc=<interface_name>' -p rtp-ml -o text
```

  - Run on PCAP file

```
./ipfixprobe -i 'pcap;file=<pcap_file>' -p rtp-ml -o text
```

Bibliography

1. FORTINET, Inc. *The Challenges of Inspecting Encrypted Network Traffic* [online]. [visited on 2023-02-21]. Available from: https://www.fortinet.com/blog/industry-trends/keeping-up-with-performance-demands-of-encrypted-web-traffic.
2. FORTINET, Inc. *Fortinet Deep packet inspection* [online]. [visited on 2023-02-21]. Available from: https://www.fortinet.com/resources/cyberglossary/dpi-deep-packet-inspection.
3. CLAISE, Benoît. *Cisco Systems NetFlow Services Export Version 9* [RFC 3954]. RFC Editor, 2004. Request for Comments, no. 3954. Available from DOI: 10.17487/RFC3954.
4. HOFSTEDE, Rick; ČELEDA, Pavel; TRAMMELL, Brian; DRAGO, Idilio; SADRE, Ramin; SPEROTTO, Anna; PRAS, Aiko. Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *IEEE Communications Surveys & Tutorials*. 2014, vol. 16, no. 4, pp. 2037–2064. Available from DOI: 10.1109/COMST.2014.2321898.
5. TRAMMELL, Brian; BOSCHI, Elisa. *Bidirectional Flow Export Using IP Flow Information Export (IPFIX)* [RFC 5103]. RFC Editor, 2008. Request for Comments, no. 5103. Available from DOI: 10.17487/RFC5103.
6. SOLARWINDS WORLDWIDE, LLC. *What Is NetFlow?* [Online]. Available also from: https://www.solarwinds.com/netflow-traffic-analyzer/use-cases/what-is-netflow.
7. CISCO SYSTEMS, Inc. *Flexible NetFlow Configuration Guide* [online]. [visited on 2023-02-21]. Available from: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/fnetflow/configuration/xe-16/fnf-xe-16-book/fnf-ipfix-export.html.
8. JUNIPER NETWORKS, Inc. *Configuring Inline Active Flow Monitoring to Use IPFIX Flow Templates on MX, vMX and T Series Routers, EX Series Switches, NFX Series Devices, and SRX Devices* [online]. [visited on 2023-02-21]. Available from: https://www.juniper.net/documentation/us/en/software/junos/flow-monitoring/topics/concept/services-ipfix-flow-template-flow-aggregation-configuring.html.
9. ARUBA NETWORKS, Inc. *Monitoring Network Traffic Using IPFIX* [online]. [visited on 2023-02-21]. Available from: https://www.arubanetworks.com/techdocs/ArubaOS_80_Web_Help/Content/ArubaFrameStyles/Mobility/IPFIX.htm.
10. FORTINET, Inc. *Netflow and IPFIX support* [online]. [visited on 2023-02-21]. Available from: https://docs.fortinet.com/document/fortigate/6.2.0/cookbook/728397/netflow-and-ipfix-support.
11. CISCO SYSTEMS, Inc. *GitHub - cisco/joy: A package for capturing and analyzing network flow data and intraflow data, for network research, forensics, and security monitoring.* [Online]. [visited on 2023-02-21]. Available from: https://github.com/cisco/joy.

12. CARNEGIE MELLON UNIVERSITY. *YAF* [online]. [visited on 2023-02-21]. Available from: https://tools.netsa.cert.org/yaf/.

13. CESNET, z. s. p. o. *ipfixprobe - IPFIX flow exporter* [online]. [visited on 2023-02-21]. Available from: https://github.com/CESNET/ipfixprobe.

14. WANG, Zihao; FOK, Kar Wai; THING, Vrizlynn L.L. Machine learning for encrypted malicious traffic detection: Approaches, datasets and comparative study. *Computers & Security*. 2022, vol. 113, p. 102542. ISSN 0167-4048. Available from DOI: https://doi.org/10.1016/j.cose.2021.102542.

15. MOZZILA CORPORATION. *WebRTC API* [online]. [visited on 2023-02-21]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API.

16. HANCKE, Philipp. *How does Hangouts use WebRTC? webrtc-internals analysis* [online]. [visited on 2023-02-21]. Available from: https://webrtchacks.com/hangout-analysis-philipp-hancke/.

17. HANCKE, Philipp. *Facebook Messenger likes WebRTC* [online]. [visited on 2023-02-21]. Available from: https://webrtchacks.com/facebook-webrtc/.

18. VASS, Jozsef. *How Discord Handles Two and Half Million Concurrent Voice Users using WebRTC* [online]. [visited on 2023-02-21]. Available from: https://discord.com/blog/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc.

19. EGEVANG, Kjeld Borch; SRISURESH, Pyda. *Traditional IP Network Address Translator (Traditional NAT)* [RFC 3022]. RFC Editor, 2001. Request for Comments, no. 3022. Available from DOI: 10.17487/RFC3022.

20. QIAOQIAO, Liu. *What Is Network Address Translation (NAT)?* [Online]. [visited on 2023-02-21]. Available from: https://info.support.huawei.com/info-finder/encyclopedia/en/NAT.html.

21. MOZZILA CORPORATION. *Introduction to WebRTC protocols* [online]. [visited on 2023-02-21]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols.

22. REDDY.K, Tirumaleswar; JOHNSTON, Alan; MATTHEWS, Philip; ROSENBERG, Jonathan. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)* [RFC 8656]. RFC Editor, 2020. Request for Comments, no. 8656. Available from DOI: 10.17487/RFC8656.

23. *TLOCs and NAT* [online]. [visited on 2023-02-21]. Available from: https://www.networkacademy.io/ccie-enterprise/sdwan/tlocs-and-nat.

24. SCHULZRINNE, Henning; CASNER, Stephen L.; FREDERICK, Ron; JACOBSON, Van. *RTP: A Transport Protocol for Real-Time Applications* [RFC 3550]. RFC Editor, 2003. Request for Comments, no. 3550. Available from DOI: 10.17487/RFC3550.

25. MOZZILA CORPORATION. *Introduction to the Real-time Transport Protocol (RTP)* [online]. [visited on 2023-02-21]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Intro_to_RTP.

26. MICROSOFT CORPORATION. *Microsoft Teams call flows* [online]. [visited on 2023-02-21]. Available from: https://learn.microsoft.com/en-us/MicrosoftTeams/microsoft-teams-online-call-flows.

27. ZOOM VIDEO COMMUNICATIONS, Inc. *Encryption for Zoom Phone* [online]. [visited on 2023-02-21]. Available from: https://support.zoom.us/hc/en-us/articles/360042578911-Encryption-for-Zoom-Phone.

28. GOOGLE, LLC. *Learn about call & meeting encryption in Google Meet* [online]. [visited on 2023-02-21]. Available from: https://support.google.com/meet/answer/12387251?hl=en.
29. MICROSOFT CORPORATION. *Encryption for Skype for Business Server* [online]. [visited on 2023-02-21]. Available from: https://learn.microsoft.com/en-us/skypeforbusiness/plan-your-deployment/security/encryption.
30. APPLE, Inc. *FaceTime security* [online]. [visited on 2023-02-21]. Available from: https://support.apple.com/en-gb/guide/security/seca331c55cd/web.
31. WHATSAPP, LLC. *WhatsApp Encryption Overview* [online]. [visited on 2023-02-21]. Available from: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.
32. THATCHER, Peter. *Multi-device calls with ICE forking* [online]. [visited on 2023-02-21]. Available from: https://signal.org/blog/ice-forking/.
33. CARRARA, Elisabetta; NORRMAN, Karl; MCGREW, David; NASLUND, Mats; BAUGHER, Mark. *The Secure Real-time Transport Protocol (SRTP)* [RFC 3711]. RFC Editor, 2004. Request for Comments, no. 3711. Available from DOI: 10.17487/RFC3711.
34. COLLINGE, Paul. *How to quickly optimize Office 365 traffic for remote staff & reduce the load on your infrastructure* [online]. [visited on 2023-02-21]. Available from: https://techcommunity.microsoft.com/t5/microsoft-365-blog/how-to-quickly-optimize-office-365-traffic-for-remote-staff-amp/ba-p/1214571.
35. MICROSOFT CORPORATION. *Office 365 URLs and IP address ranges* [online]. [visited on 2023-02-21]. Available from: https://learn.microsoft.com/en-us/microsoft-365/enterprise/urls-and-ip-address-ranges?view=o365-worldwide.
36. AL., Julianna Delua et. *Supervised vs. Unsupervised Learning: What's the Difference?* [Online]. [visited on 2023-02-21]. Available from: https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning.
37. BOUTABA, Raouf; SALAHUDDIN, Mohammad A.; LIMAM, Noura; AYOUBI, Sara; SHAHRIAR, Nashid; ESTRADA-SOLANO, Felipe; CAICEDO, Oscar M. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*. 2018, vol. 9, no. 1. Available from DOI: 10.1186/s13174-018-0087-2.
38. IBM, Inc. *What is a Decision Tree?* [Online]. [visited on 2023-02-21]. Available from: https://www.ibm.com/topics/decision-trees.
39. IBM, Inc. *What is the k-nearest neighbors algorithm?* [Online]. [N.d.]. [visited on 2023-02-21]. Available from: https://www.ibm.com/topics/knn#:~:text=The%5C%20k%5C%2Dnearest%5C%20neighbors%5C%20algorithm%5C%2C%5C%20also%5C%20known%5C%20as%5C%20KNN%5C%20or,of%5C%20an%5C%20individual%5C%20data%5C%20point..
40. IBM, Inc. *What is random forest?* [Online]. [visited on 2023-02-21]. Available from: https://www.ibm.com/topics/random-forest#:~:text=Random%5C%20forest%5C%20is%5C%20a%5C%20commonly,both%5C%20classification%5C%20and%5C%20regression%5C%20problems..
41. IBM, Inc. *What is boosting?* [Online]. [visited on 2023-02-21]. Available from: https://www.ibm.com/topics/boosting.
42. PEDAMKAR, Priya. *Decision Tree in Machine Learning* [online]. [visited on 2023-02-21]. Available from: https://www.educba.com/decision-tree-in-machine-learning/.
43. COURNAPEAU, David; AL., Matthieu Brucher et. *Ensemble methods* [online]. [visited on 2023-02-21]. Available from: https://scikit-learn.org/stable/modules/ensemble.html.

44. LEXISNEXIS RISK SOLUTIONS. *Learning Trees – A guide to Decision Tree based Machine Learning* [online]. [visited on 2023-02-21]. Available from: https://hpccsystems.com/resources/learning-trees-a-guide-to-decision-tree-based-machine-learning/.

45. HO, Tin Kam. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 1998, vol. 20, no. 8, pp. 832–844. Available from DOI: 10.1109/34.709601.

46. FREUND, Yoav; SCHAPIRE, Robert E. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*. 1997, vol. 55, no. 1, pp. 119–139. ISSN 0022-0000. Available from DOI: https://doi.org/10.1006/jcss.1997.1504.

47. COMMUNITY, Distributed (Deep) Machine Learning. *XGBoost Documentation* [online]. [visited on 2023-02-21]. Available from: https://xgboost.readthedocs.io/en/stable/index.html.

48. LINUX FOUNDATION. *Open vSwitch* [online]. [visited on 2023-02-21]. Available from: https://www.openvswitch.org/.

49. YANGYANG, Gao. *What Is OpenFlow?* [Online]. [visited on 2023-02-21]. Available from: https://info.support.huawei.com/info-finder/encyclopedia/en/OpenFlow.html.

50. LINUX FOUNDATION. *Implementation Details* [online]. [visited on 2023-02-21]. Available from: https://docs.openvswitch.org/en/latest/faq/design/.

51. LINUX FOUNDATION. *Open vSwitch manual - ovs-ofctl* [online]. [visited on 2023-02-21]. Available from: http://www.openvswitch.org//support/dist-docs/ovs-ofctl.8.txt.

52. PERKINS, Colin; WESTERLUND, Magnus. *Multiplexing RTP Data and Control Packets on a Single Port* [RFC 5761]. RFC Editor, 2010. Request for Comments, no. 5761. Available from DOI: 10.17487/RFC5761.

53. CESNET, z. s. p. o. *CESNET2 network* [online]. [visited on 2023-02-21]. Available from: https://www.cesnet.cz/services/ip-connectivity-ip/cesnet2-network/?lang=en.

54. JUPYTER, Project. *Project Jupyter* [online]. [visited on 2023-02-21]. Available from: https://jupyter.org/.

55. *NumPy* [online]. [visited on 2023-02-21]. Available from: https://numpy.org/.

56. *Pandas - Python Data Analysis Library* [online]. [visited on 2023-02-21]. Available from: https://pandas.pydata.org/.

57. *Matplotlib - Visualization with Python* [online]. [visited on 2023-02-21]. Available from: https://matplotlib.org/.

58. LI, Jundong; CHENG, Kewei; WANG, Suhang; MORSTATTER, Fred; TREVINO, Robert P.; TANG, Jiliang; LIU, Huan. Feature Selection: A Data Perspective. *ACM Comput. Surv.* 2017, vol. 50, no. 6. ISSN 0360-0300. Available from DOI: 10.1145/3136625.

59. COURNAPEAU, David; AL., Matthieu Brucher et. *scikit-learn: machine learning in Python* [online]. [visited on 2023-02-21]. Available from: https://scikit-learn.org/stable/.

60. GUPTA, Kwanit. *GitHub - kwanit1142/Machine-Learning-Models-on-different-scenarios: Pattern Recognition and Machine Learning based Assignments and Labs, under Prof. Richa Singh in Course CSL2050.* [Online]. [visited on 2023-02-21]. Available from: https://github.com/kwanit1142/Machine-Learning-Models-on-different-scenarios.

61. ZOU, Yuming; BLACK, Paul E. *perfect binary tree* [online]. [visited on 2023-02-21]. Available from: https://xlinux.nist.gov/dads/HTML/perfectBinaryTree.html.

62. MORAWIEC, Darius. *GitHub - nok/sklearn-porter: Transpile trained scikit-learn estimators to C, Java, JavaScript and others.* [Online]. [visited on 2023-02-21]. Available from: `https://github.com/nok/sklearn-porter`.

63. ZEIGERMAN, Iaroslav; YERSHOV, Viktor; TITOV, Nikita. *GitHub - BayesWitnesses/m2cgen: Transform ML models into a native code (Java, C, Python, Go, JavaScript, Visual Basic, C#, R, PowerShell, PHP, Dart, Haskell, Ruby, F#, Rust) with zero dependencies* [online]. [visited on 2023-02-21]. Available from: `https://github.com/BayesWitnesses/m2cgen`.

64. DISTRIBUTED (DEEP) MACHINE LEARNING COMMUNITY. *Treelite : model compiler for decision tree ensembles* [online]. [visited on 2023-02-21]. Available from: `https://treelite.readthedocs.io/en/latest/`.

65. CZ.NIC, z. s. p. o. *Turris - Overview* [online]. [visited on 2023-02-21]. Available from: `https://www.turris.com/en/omnia/overview/`.

Content of attached storage device

- `readme.txt` — description of the attached storage's content and its structure
- `source` — folder containing all source codes of the implementation
  - `helpers` — folder containing helper scripts
  - `ipfixprobe-modules` — folder containing the source codes for IPFIXprobe modules
  - `notebooks` — folder containing Jupyter notebooks used during the development
  - `models` — folder containing final machine learning model
- `thesis.pdf` — text for the bachelor thesis in PDF format
- `thesis.zip` — archive containing LaTeX source files for the thesis document