**PKS project 1**

**Packet analyzer**

Márk Bartalos

Seminars with**: Ing. Lukáš Mastiľak**Seminars on**: Thu 14:00-15:50**

Contents

[Introduction 3](#_Toc117114987)

[Programming environment 3](#_Toc117114988)

[User interface 3](#_Toc117114989)

[Used libraries 3](#_Toc117114990)

[Implementation of task 1 3](#_Toc117114991)

[External file structure 5](#_Toc117114992)

[Flowchart of analyzing packet processing 8](#_Toc117114993)

[Implementaton of task two 8](#_Toc117114994)

[Analysis of frames 9](#_Toc117114995)

[Implementation of task three 10](#_Toc117114996)

[Task four 11](#_Toc117114997)

[Filtering 11](#_Toc117114998)

[TCP handshake and communication detection 13](#_Toc117114999)

[UDP communication pairing 13](#_Toc117115000)

[ARP 14](#_Toc117115001)

[ICMP 15](#_Toc117115002)

[Final results 16](#_Toc117115003)

[Further extensibility 16](#_Toc117115004)

[Examples of YAML exports and “interface” 16](#_Toc117115005)

# Introduction

As task we got building a packet analyzer which has similar functionality as the popular Wireshark program.

# Programming environment

As programming environment, I chose Python 3.10.8 and Visual Studio Code. I choose Python because it will be a great fit for this task with its versatility ease of use and many libraries.

Application can be started using the command: ***python main.py***

It can be also supplied extra parameters such as: ***python main.py -p PROTOCOL FILE\_NAME***(Note that file name can only be supplied if protocol filter is used)

**Note: input files have to be in the Capture directory**

# User interface

As this is a command line application user interface is only the command used to run the application. All data is written into YAML by design.

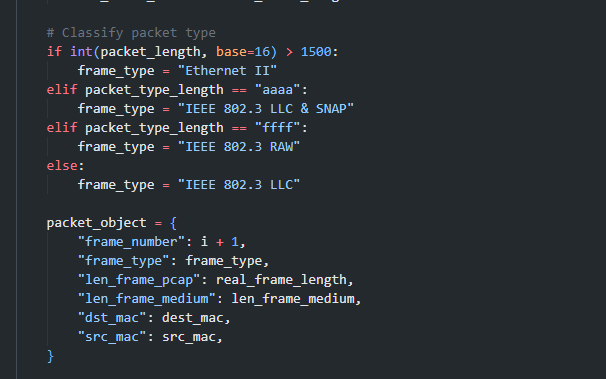
# Used libraries

* scapy.all - for reading the packet capture files
* scapy.compat – for getting the raw data from the read files
* ruamel.yaml – for formatting and writing YAML

# Implementation of task 1

For task 1 we had to implement basic analyzation of packets, we had to extract useful information from them such as their source and destination MAC address, frame type, length of the frame and so on. This extracted we had to serialize to YAML and save to a file and the end of each packet we had to append a hexa frame which is basically the full packet in hexadecimal numbers.

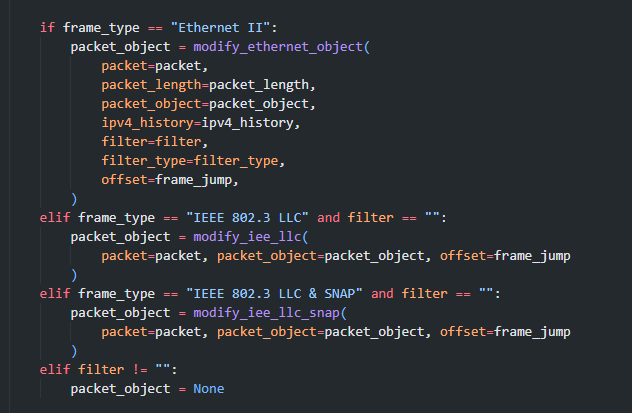
The main analysis was done in the analyze\_frames() method



Here we classify the packet based on its length and packet type which is from the packet itself.

After identifying the packet we create the packet object.

Each packet type has its modify function which fills out its type specific properties.

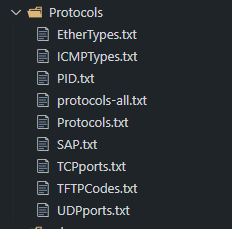


ETHERNET II specific modify function

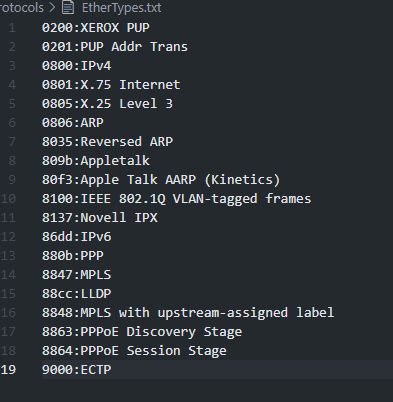


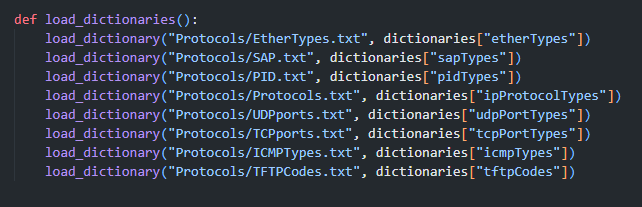
For setting properties such as ether type or app protocol I used .txt files which are loaded into a dictionary at the start of the program

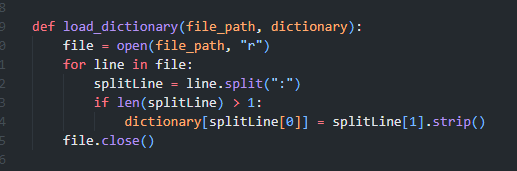
## External file structure

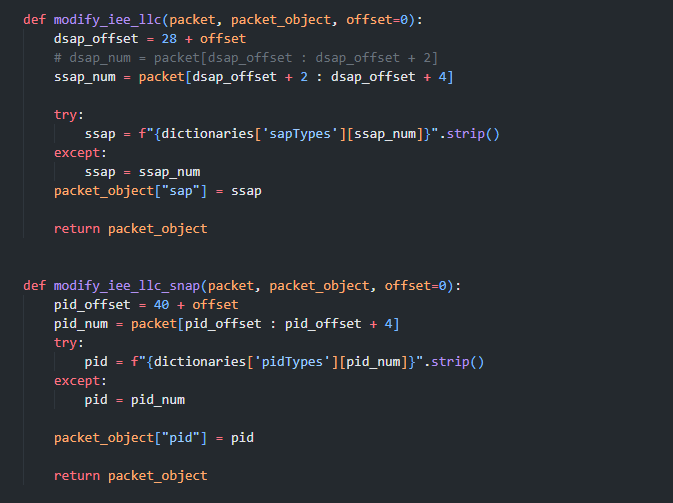


EtherTypes.txt

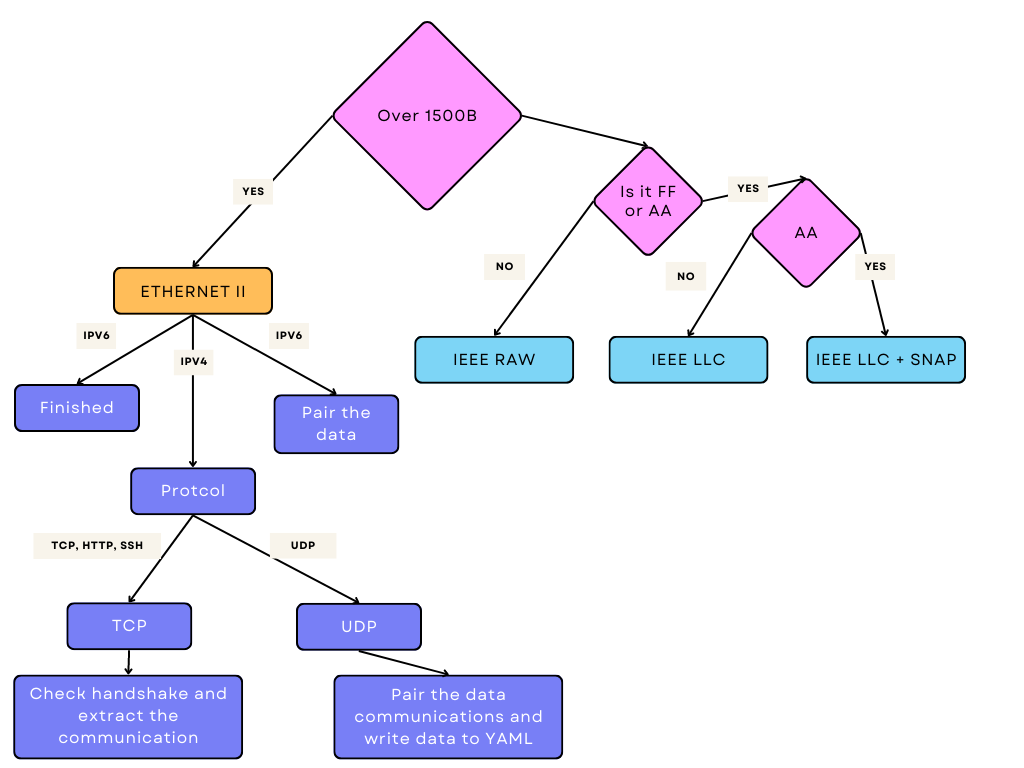






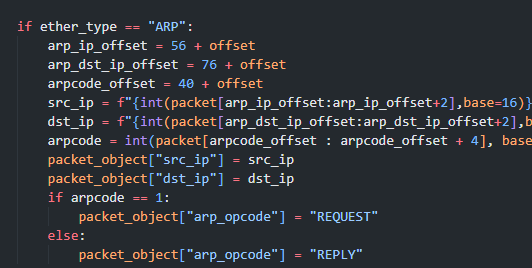


# Flowchart of analyzing packet processing

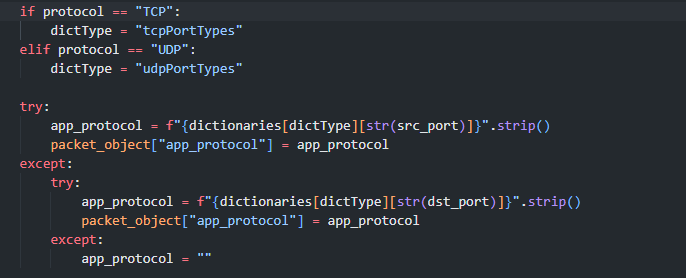


# Implementaton of task two

Here we had to fill out more and protocol specific information to each protocol in the ETHERNET II frame. Example for ARP:

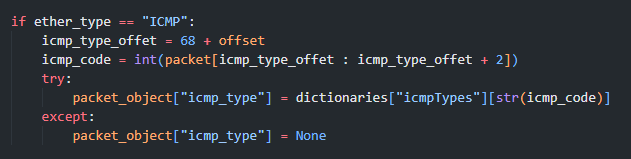


All the well know protocol names are loaded from files and retrieved from a dictionary

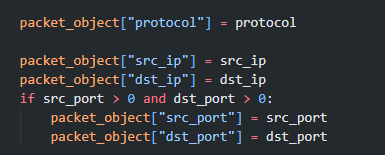


Here we first get the type of protocol to be able the retrieve the appropriate dictionary, because TCP and UDP well know ports are in different txt files.

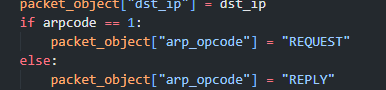
## Analysis of frames



Settings the ICMP type from the ICMP header



Setting the IP and the port for IPV4

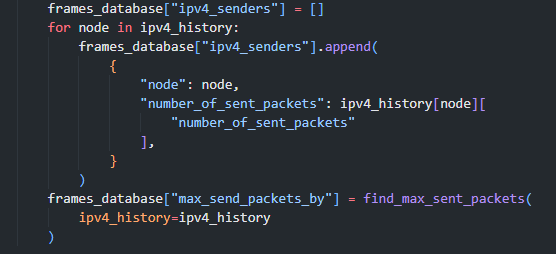


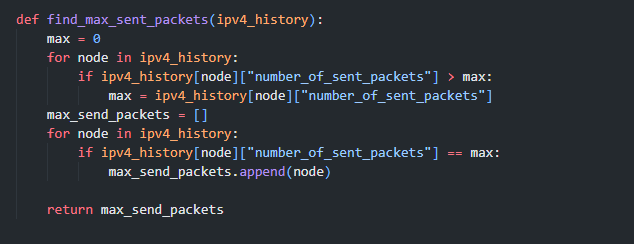
Settings arpcode for arp.

# Implementation of task three

Here we had to create statistics of IPV4 packets. I did this using a dictionary where the key was the sender’s ip address and if the same sender sent a packet the packet count was incremented.

Then at the end selected the largest sent packet number and using a for loop I search up the ip addresses which sent the same amount of packets.

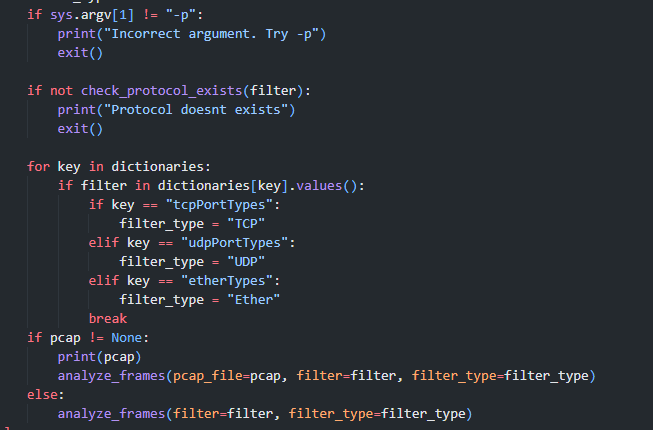




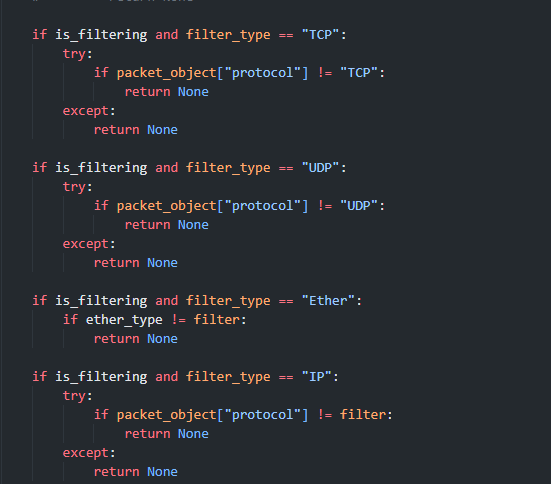
# Task four

## Filtering

Filtering is done using -p argument. When passing a protocol to the filter the filter checks if the protocol exists from external files, then gets the correct type if the protocol is a nested protocol such as HTTP. Then passes these arguments to the analyze\_frames() function. It also supports adding file name as additional argument when using -p.

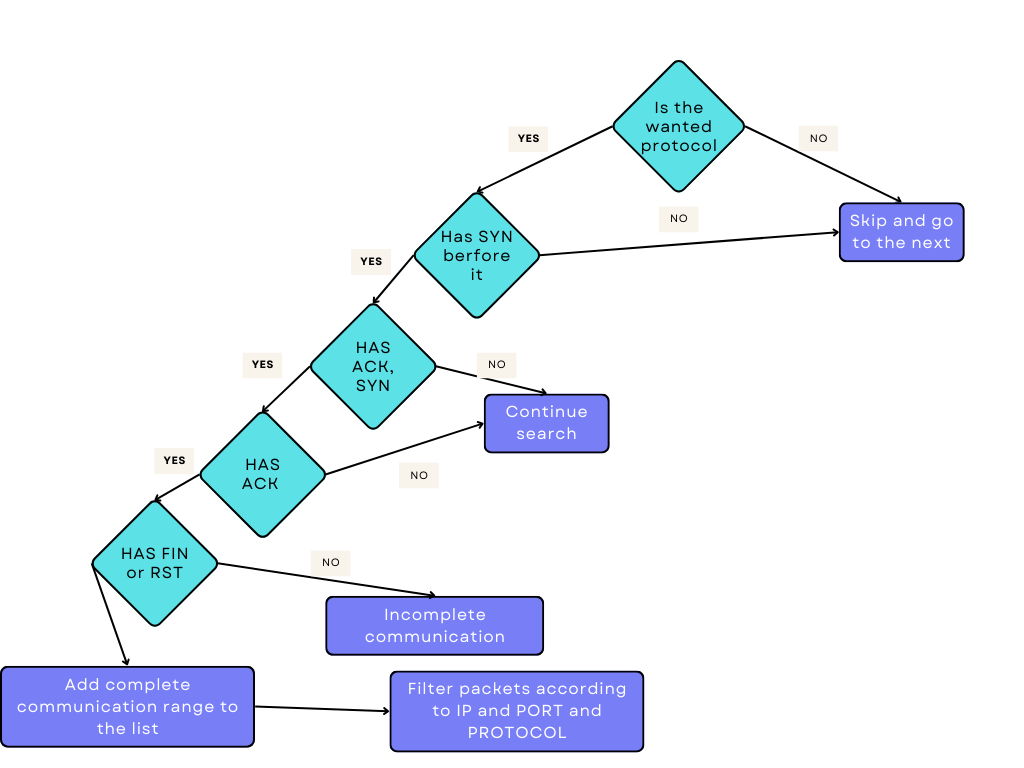


After that the packeges are filtered in the modify\_ethernet\_object () function.



This is done for every packet one by one.

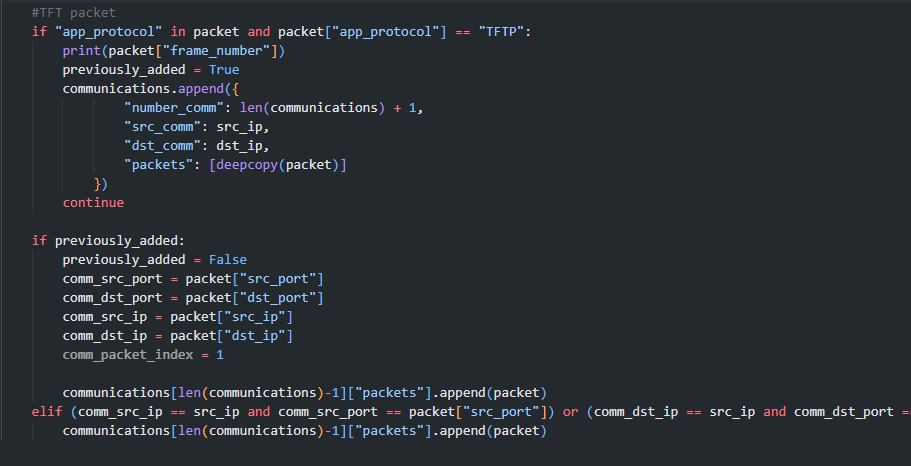
## TCP handshake and communication detection



First, I checked the handshake then if a complete connection was the result I added the range of packets inside of that connection, then in the next step I filtered all those packets according to IP, PORT and filter PROTOCOL which was passed into the function.

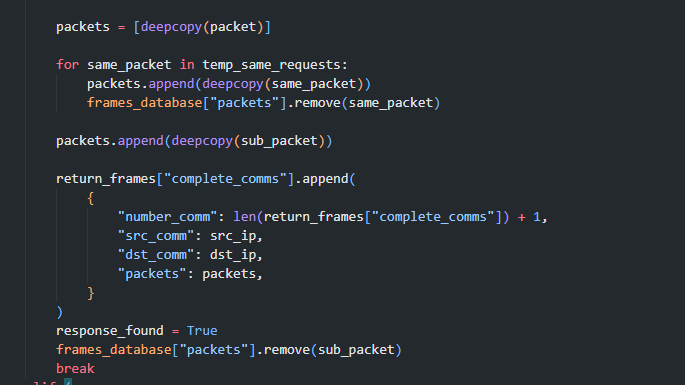
## UDP communication pairing

With each TFTP packet I added a connection object to the connection list, after the TFTP packet, I saved the ports and IP addresses for the next packet and all following packets were compared to these ports and IP addresses, if they did match than they were part of the same connection. Communications which were not preceded by TFTP packet were only paired by port and ip address. The first IP and port addresses were saved and if the next matched the IPs and the ports than it belonged to the same communication.



## ARP

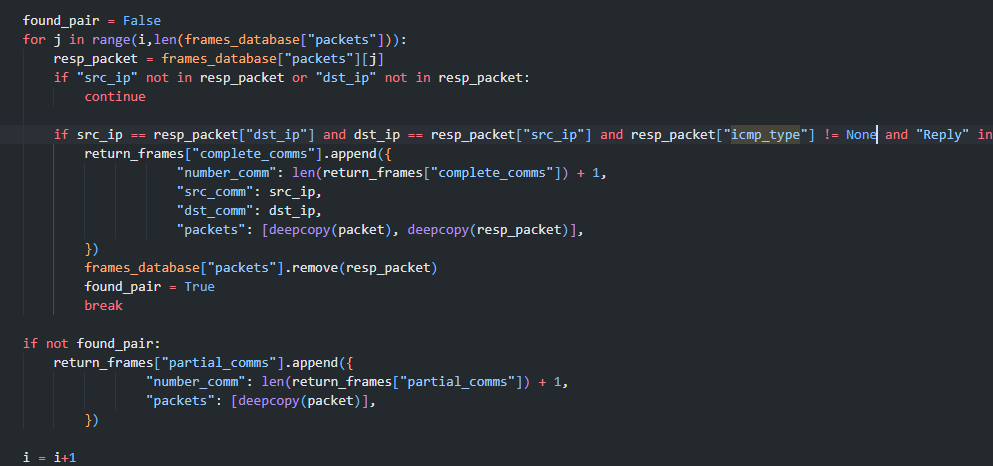
In the case of ARP I looped through the ARP packages one by one, if a REQUEST packet is found the program starts another loop from that packet and searches until it finds a REPLY for that exact Ip address. If along the way it finds a REQUEST packet with the same addresses it adds to a temporary list and these are added when a current REPLY if found in that way supporting N\*REQUEST. Added REPLY and REQUEST (after the first one) are removed from the list.  
Then these complete communications are appended to the return object, which will contain all communications and will be returned by the function.



## ICMP

For ICMP protocol I first when filtering I filled out the ICMP type where this field existed, then similar to method used for ARP with a for loop I went through the loop and for each request I tried to find a response with a nested loop, if a reply was found the communication was added to the complete communications list and the reply packet was deleted.

For every request packet it tries to find and answer packet meaning the destination and the source ips of the source are flipped and contains the keyword “Reply”.



# Final results

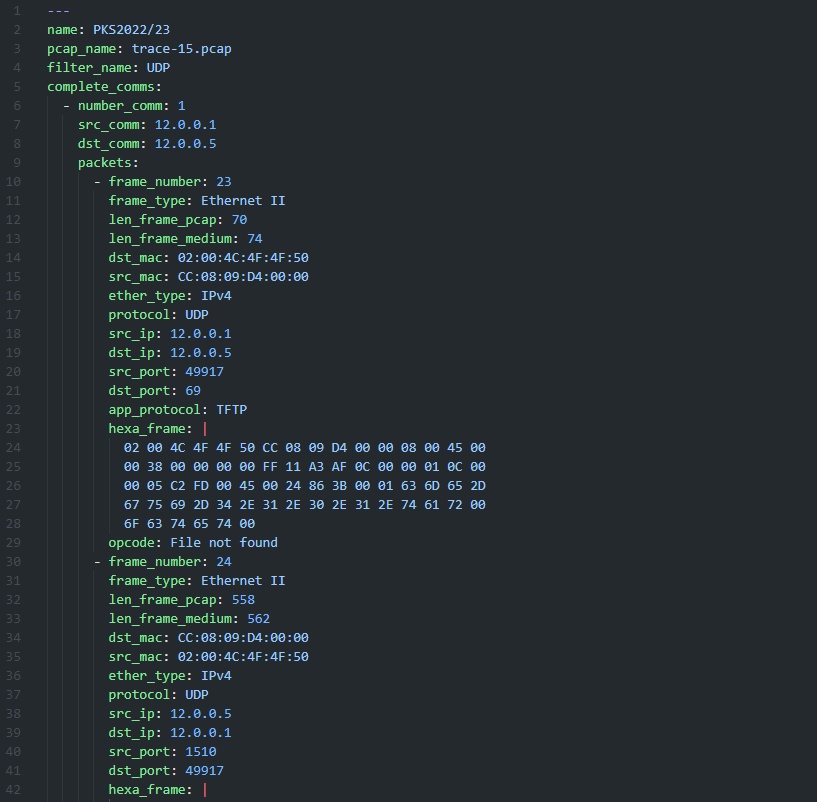
All task were successfully completed and implemented. Every output yaml file was tested with the validator and were successfully validated. For validation I used a simple **tester.py** python script which tested the different protocols and filters on each capture file and then validated the results using the validator.

# Further extensibility

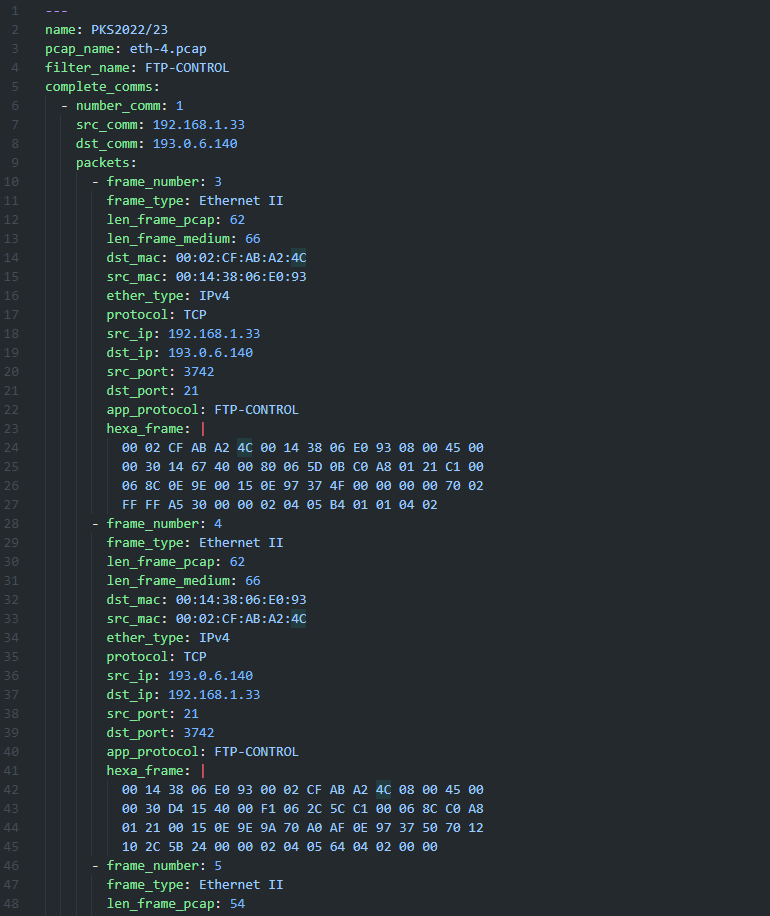
The program can be easily extended by additional scripts. Due to the codes more procedural nature in most cases the man.py file will have to be modified. New protocol support can be easily added by appending to the protocol text files. New protocol and header analyzation also can be added pretty easily by calling it in the analyze\_frames() function.

# Examples of YAML exports and “interface”

UDP TFTP communication



TCP communication using FTP-CONTROL filter



No filter – trace-26.pcap

