FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
SLOVENSKÁ TECHNICKÁ UNIVERZITA

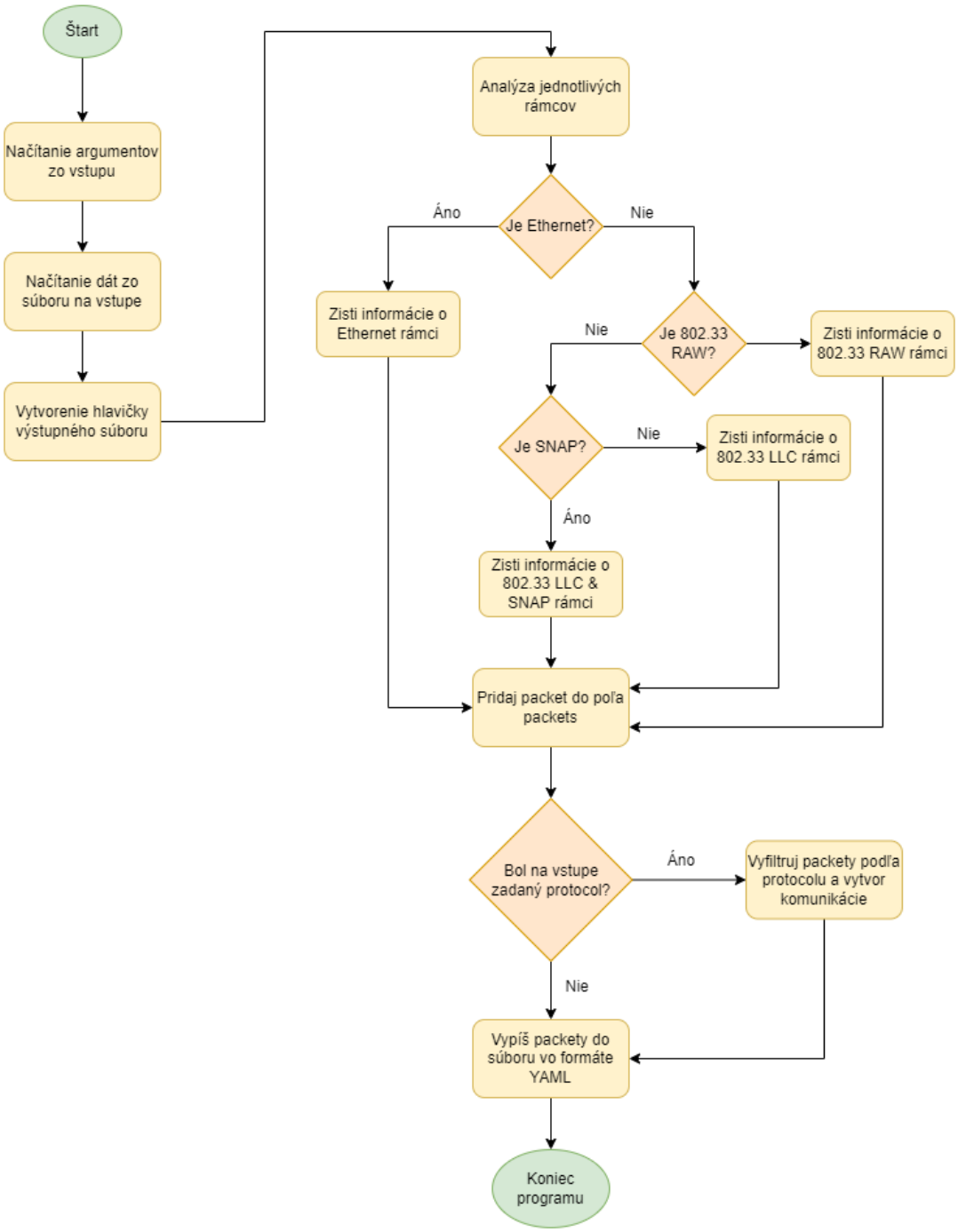
Počítačové a komunikačné siete

Zadanie č.1

Meno: René Bukovina
Cvičenie: štvrtok 16:00
Cvičiaci: Bc. Branislav Jančovič

Obsah

Flowchart


Štart
Načítanie argumentov zo vstupu
Načítanie dát zo súboru na vstupe
Vytvorenie hlavičky výstupného súboru
Analýza jednotlivých rámcov
Je Ethernet?
Áno
Nie
Zisti informácie o Ethernet rámci
Je 802.33 RAW?
Nie
Zisti informácie o 802.33 RAW rámci
Je SNAP?
Nie
Zisti informácie o 802.33 LLC rámci
Áno
Zisti informácie o 802.33 LLC & SNAP rámci
Pridaj packet do poľa packets
Bol na vstupe zadaný protocol?
Áno
Vyfiltruj packety podľa protocolu a vytvor komunikácie
Nie
Vypíš packety do súboru vo formáte YAML
Koniec programu

Mechanizmus analýzy jednotlivých vrstiev

Základné informácie o rámci

```
for idx, item in enumerate(getScapy, start=1):
    # decode every frame and make and array of every byte in frame
    hexDecoded = hexlify(raw(item), " ").decode().split()

    # check if we have ISL header
    if ''.join(hexDecoded[0:6]) == "01000c000000" or ''.join(hexDecoded[0:6]) == "03000c000000":
        hexDecoded = hexDecoded[26:-1]

    # create packet dictionary to store all information
    pkt = {
        'frame_number': idx,
        'len_frame_pcap': len(hexDecoded),
        'len_frame_medium': 64 if len(hexDecoded) < 60 else len(hexDecoded) + 4,
        'frame_type': getFrameType(hexDecoded),
        'src_mac': formatAddress(hexDecoded[6:12]),
        'dst_mac': formatAddress(hexDecoded[0:6]),
    }
```

Typ rámca

1. Skontrolujeme či hodnota na 13. a 14. bajte je väčšia ako 1500. Ak áno, tak pokračujeme na analýzu rámca typu ETHERNET II. Ak nie tak ideme na krok 2.
2. Ak máme v rámci na 15. a 16. bajte hexadecimálnu hodnotu ‚FFFF', tak sa jedná od rámec typu Novell 802.33 RAW, a následne pokračujeme na analýzu rámca tohto typu. Ináč ideme na krok 3.
3. Ak sa v rámci na 15. a 16. bajte nachádza hexadecimálna hodnota ‚AAAA', tak vieme, že sa jedná o rámec typu IEEE 802.33 LLC & SNAP, a môžem pokračovať analýzou tohto typu rámca. Ak sa ale hodnoty líšia od danej hodnoty, tak predpokladáme, že sa jedná o rámec typu IEEE 802.33 LCC, preto budeme riešiť analýzu tohto typu.

```
def getFrameType(hex):
    if (int(hex[12]+hex[13], 16) >= 1500):
        return 'ETHERNET II'

    if ((hex[14]+hex[15]) == 'ffff'):
        return 'IEEE 802.3 RAW'

    if ((hex[14]+hex[15]) == 'aaaa'):
        return 'IEEE 802.3 LLC & SNAP'
    else:
        return 'IEEE 802.3 LLC'
```

Analýza 2. vrstvy

Informácia o type protokolu sa nachádza na 13. a 14. bajte pri rámcoch typu Ethernet II, tieto dva bajty obsahujú 4-miestny hexadecimálny reťazec, ktorý označuje daný protokol na druhej vrstve. Možné typy protokolov na danej vrstve sa nachádzajú v súbore *ethernet-protocols.json*

```
# analyzing ethernet II frames
 if 'ETHERNET II' in pkt['frame_type']:
     #find ether_type on 13th and 14th byte
     pkt['ether_type'] = getEtherType(hexDecoded[12]+hexDecoded[13])
```

Pokiaľ sa hodnota nerovná žiadnej v databáze, tak použijeme hodnotu „unknown“. To zároveň platí aj pre prideľovanie protokolov, známych portov atď.

Analýza pokračuje iba pre rámce s protokolom ARP alebo IPv4.

```
# analyzing ethernet II frames
if 'ETHERNET II' in pkt['frame_type']:
    # find ether_type on 13th and 14th byte
    pkt['ether_type'] = getEtherType(hexDecoded[12]+hexDecoded[13])

    if 'IPv4' in pkt['ether_type']:
        ihl = int(hexDecoded[14][1], 16)
        pkt['src_ip'] = getIp(hexDecoded[26:30])
        pkt['dst_ip'] = getIp(hexDecoded[30:34])
        pkt['id'] = int(''.join(hexDecoded[18:20]), 16)

        # Fragments checks
        flag_value = int(hexDecoded[20], 16) >> 5
        offset = int(
            ''.join(hexDecoded[20:22]), 16) & (2**3-1)  # https://stackoverflow.com/a/74338975/19510795

        if (flag_value == 1):  # this means packet is fragment
            pkt['flag_mf'] = True
            pkt['frag_offset'] = offset
        elif (flag_value == 0 and offset > 0):
            pkt['flag_mf'] = False
            pkt['frag_offset'] = offset

        ###

        pkt['protocol'] = getIpProtocol(int(hexDecoded[23], 16))
        # find source ports with ihl to know we need to look for them
        if pkt['protocol'] == 'UDP' or pkt['protocol'] == 'TCP':
            pkt['src_port'] = int(
                hexDecoded[14+ihl*4]+hexDecoded[14+ihl*4+1], 16)
            pkt['dst_port'] = int(
                hexDecoded[14+ihl*4+2]+hexDecoded[14+ihl*4+3], 16)
            # check to see if one of found ports is known port to protocol IPv4 protocol
            if type(getAppProtocol(pkt['src_port'], pkt['protocol'])) == str or type(getAppProtocol(pkt['dst_port'], pkt['protocol'])) == str:
                pkt['app_protocol'] = getAppProtocol(pkt['src_port'], pkt['protocol']) if type(getAppProtocol(
                    pkt['src_port'], pkt['protocol'])) == str else getAppProtocol(pkt['dst_port'], pkt['protocol'])

        elif 'ICMP' == pkt['protocol']:...

        # ipv4_senders counter
        if pkt['src_ip'] in ip_send:
            ip_send[pkt['src_ip']] += 1
        else:
            ip_send[pkt['src_ip']] = 1
```

Za povšimnutie stojí fakt, že *hexDecoded* je náš vstupný reťazec rozdelený po bajtoch do poľa, kde jednotlivé indexy pre dané pole majú hodnotu *bajtu v rámci – 1*. Práve *-1*, kvôli tomu, že sa jedná o pole a to začína indexom *0*.

Sekcia **Fragments Checks** slúži pre overovanie fragmentovaných rámcov pre ICMP komunikácie. Poznámka a link na Stackoverflow riešenie je prevzaté iba na odstránenie prvých 3-bitov z

Štatistika IPv4

Ak nebol na vstupe zadaný protokol pre tvorbu komunikácií, robíme štatistiku pre každú IP adresu s počtom odoslaných paketov. Počítanie jednotlivých odoslaných paketov pre danú IP adresu robíme súbežne s analýzou 2. vrstvy.

```
# ipv4_senders counter
if pkt['src_ip'] in ip_send:
    ip_send[pkt['src_ip']] += 1
else:
    ip_send[pkt['src_ip']] = 1
```

Na konci programu si pred výpisom do súboru nájdeme, ktorá z IP adries odoslala najviac paketov. Ak je takých IP adries viac, tak vypíšeme všetky.

```
else:
    data['packets'] = packets
    data['ipv4_senders'] = []

    # format counted ips to spec
    for key, i in ip_send.items():
        data['ipv4_senders'].append({
            'node': key,
            'number_of_sent_packets': i
        })

    data['max_send_packets_by'] = []
    # get ip with the most packets sent
    data['max_send_packets_by'].append(max(ip_send, key=ip_send.get))
```

Analýza 3. vrstvy

Analýzu vnoreného protokolu na 3. vrstve vykonávame iba pre rámce s protokolom IPv4, nakoľko protokol ARP nedisponuje touto vlastnosťou. Databáza pre tieto protokoly sa nachádza v súbore *ipv4-protocols.json*

```
def getIpProtocol(dec):
    file = open("./protocols/ipv4-protocols.json", "r")
    protocols = json.load(file)
    try:
        return protocols[str(dec)]
    except:
        print("Uknown port: {}".format(dec))
        return 'unknown'
```

Analýza 4. vrstvy

Analýzu 4. vrstvy vykonávame pre protokoly TCP a UDP. Ak port nepatrí žiadnemu známemu protokolu, tak názov vo výpise neuvádzame. Databázy sa nachádzajú v súboroch *udp-protocols.json* a *tcp-protocols.json*

```
def getAppProtocol(dec, protocol):

    if 'UDP' in protocol:
        udp = open("./protocols/udp-protocols.json")
        udp_ports = json.load(udp)
        try:
            return udp_ports[str(dec)]
        except:
            return False
    elif 'TCP' in protocol:
        tcp = open("./protocols/tcp-protocols.json")
        tcp_ports = json.load(tcp)
        try:
            return tcp_ports[str(dec)]
        except:
            return False
    else:
        return False
```

Dodatočnú analýzu pre protokol ICMP vykonávame zvlášť pri analýze všetkých IPv4 rámcov. Všetky podporované typy ICMP komunikácie sa nachádzajú v súbore *icmp-types.json*

```
elif 'ICMP' == pkt['protocol']:
    pkt['icmp_type'] = getIcmpType(
        int(
            hexDecoded[14+ihl*4], 16))

    if pkt['icmp_type'] in ['ECHO REQUEST', 'ECHO REPLY', 'TIME EXCEEDED']:
        pkt['icmp_id'] = int(
            hexDecoded[14+ihl*4+4]+hexDecoded[14+ihl*4+5], 16)
        pkt['icmp_seq'] = int(
            hexDecoded[14+ihl*4+6]+hexDecoded[14+ihl*4+7], 16)

    # predefine dict keys to be filled in last fragment
    if flag_value == 0 and offset > 0:
        pkt['icmp_id'] = ''
        pkt['icmp_seq'] = ''
```

```
def getIcmpType(dec):
    file = open("./protocols/icmp-types.json", "r")
    types = json.load(file)
    try:
        return types[str(dec)]
    except:
        print("Uknown type: {}".format(dec))
        return 'unknown'
```

Mechanizmus analýzy komunikácií

V mojej implementácii sa nachádza možnosť analyzovať komunikácie typu ARP, ICMP(s fragmentáciou) a typu TFTP.

Analýza ARP

```
def analyzeArp(packets):
    complete_comms = []
    partial_comms = []
    passed_frame_numbers = []
    arp_packets = list(
        filter(lambda packet: packet['ether_type'] == 'ARP', packets))
    # get only ARP packets
    for packet1 in arp_packets:

        # if we already passed the frame, we continue to next one
        if packet1["frame_number"] in passed_frame_numbers:
            continue

        # go through all packages, except passed one's and find reply to request
        for packet2 in arp_packets:
            if packet1['frame_number'] == packet2['frame_number'] or packet2['frame_number'] in passed_frame_numbers:
                continue

            if packet1['arp_opcode'] == 'REQUEST' and packet2['arp_opcode'] == 'REPLY' and packet1['dst_ip'] ==
            packet2['src_ip'] and packet2['dst_ip'] == packet1['src_ip']:
                comm = commExists(complete_comms, packet1, packet1)

                if 'number_comm' not in comm:      You, last week • fully working ARP analysation …
                    initialize_comm(comm, complete_comms, packet1)

                comm['packets'].append(packet1)
                comm['packets'].append(packet2)
                passed_frame_numbers.append(packet1['frame_number'])
                passed_frame_numbers.append(packet2['frame_number'])
                if len(comm['packets']) == 2:
                    complete_comms.append(comm)
                break

        if packet1["frame_number"] in passed_frame_numbers:
            continue

        comm = commExists(partial_comms, packet1, packet1)
        if 'packets' not in comm:
            initialize_comm(comm, partial_comms, packet1)

        comm['packets'].append(packet1)
        passed_frame_numbers.append(packet1['frame_number'])

        if len(comm['packets']) < 2:
            partial_comms.append(comm)

    data = {
        'complete_comms': complete_comms,
        'partial_comms': partial_comms
    }
    return data
```

Analýza ICMP

Mechanizmus analýzy ICMP komunikácií je postavený na princípe analýzy ARP je však samozrejme obohatený o nachádzanie príslušných fragmentov vo funkcii *find_fragments(),* ktorá bude popísaná nižšie

```
def analyzeIcmp(packets):
    complete_comms = []
    partial_comms = []
    passed_frame_numbers = []
    icmp_packets = list(
        filter(lambda packet:  packet['protocol'] == 'ICMP', packets))

    # get only ICMP packets
    for request in icmp_packets:

        # if we already passed the frame, we continue to next one
        if request["frame_number"] in passed_frame_numbers:
            continue

        # go through all packages, except passed one's and find reply to request
        for reply in icmp_packets:
            if request['frame_number'] == reply['frame_number'] or reply['frame_number'] in passed_frame_numbers:
                continue

            if request['icmp_type'] == 'ECHO REQUEST' and ((reply['icmp_type'] == 'ECHO REPLY' and request['icmp_id'] == reply['icmp_id'] and
            request['dst_ip'] == reply['src_ip']) or reply['icmp_type'] == 'TIME EXCEEDED') and reply['dst_ip'] == request['src_ip']:
                comm = commExists(complete_comms, request, reply, is_icmp=True)
                new_comm = False

                if 'number_comm' not in comm:  # initializing new communication if it doesnt exist yet
                    new_comm = True
                    initialize_comm(comm, complete_comms, request, True)

                comm['packets'].append(request)
                # now we need to find corensponding fragments to REQUEST
                if 'flag_mf' in request:
                    find_fragments(comm, icmp_packets,
                                   request, passed_frame_numbers)

                comm['packets'].append(reply)
                if 'flag_mf' in reply:
                    find_fragments(comm, icmp_packets,
                                   reply, passed_frame_numbers)

                passed_frame_numbers.append(request['frame_number'])
                passed_frame_numbers.append(reply['frame_number'])

                if new_comm:  # appending only if the comm is new
                    complete_comms.append(comm)
                break

        if request["frame_number"] in passed_frame_numbers:
            continue

        comm = commExists(partial_comms, request, request)

        if 'packets' not in comm:
            comm['number_comm'] = 1 if len(
                partial_comms) < 0 and 'number_comm' not in comm else len(partial_comms) + 1
            comm['src_comm'] = request["src_ip"]
            comm['dst_comm'] = request["dst_ip"]
            comm['packets'] = []

        comm['packets'].append(request)
        passed_frame_numbers.append(request['frame_number'])

        if len(comm['packets']) < 2:
            partial_comms.append(comm)

    data = {
        "complete_comms": complete_comms,
        "partial_comms": partial_comms
```

Dohľadávanie fragmentov

```
def find_fragments(comm, icmp_packets, current_packet, passed_frame_numbers):
    # filter all fragments except current_packet, add packet info to last fragment and then remove
    info from current_packet, same goes for reply later
    fragments = sorted(list(filter(lambda packet: packet != current_packet and 'flag_mf' in packet
    and packet['src_ip'] == current_packet[
                        'src_ip'] and packet['dst_ip'] == current_packet['dst_ip'] and packet['id']
                       == current_packet['id'], icmp_packets)), key=lambda pkt: pkt['frame_number'])

    last_frag = fragments[-1]
    last_frag['protocol'] = current_packet['protocol']
    last_frag['icmp_type'] = current_packet['icmp_type']
    last_frag['icmp_id'] = current_packet['icmp_id']
    last_frag['icmp_seq'] = current_packet['icmp_seq']

    # remove packet info from first fragment
    for k in ['protocol', 'icmp_type', 'icmp_id', 'icmp_seq']:
        current_packet.pop(k, None)
    # append fragments to comm
    for f in fragments:
        passed_frame_numbers.append(f['frame_number'])
        comm['packets'].append(f)
```

Analýza TFTP

```
def analyzeTftp(packets):
    complete_comms = []
    # get only UDP packets
    udp_packets = list(filter(
        lambda packet: packet['ether_type'] == 'IPv4' and packet['protocol'] == 'UDP', packets))

    passed_frames = []
    for packet in udp_packets:
        comm = {}
        # find packet with port of TFTP com. start point
        if packet['frame_number'] in passed_frames:
            continue

        if packet['dst_port'] == 69:
            # format new communication header
            comm['number_comm'] = len(complete_comms) + 1
            comm['src_comm'] = packet['src_ip']
            comm['dst_comm'] = packet['dst_ip']
            comm['packets'] = []
            comm['packets'].append(packet)
            passed_frames.append(packet['frame_number'])

            for pair in udp_packets:
                if pair['frame_number'] in passed_frames:
                    continue
                # find packet responding to first packets source IP
                if pair['dst_port'] == packet['src_port'] and packet['src_ip'] == pair['dst_ip'] and packet['dst_ip'] == pair['src_ip']:
                    comm['packets'].append(pair)
                    passed_frames.append(pair['frame_number'])
                    flip = False
                    # cycle the rest of the packets to find all remaining packets of communication when we know the source and dest ports
                    for rest in udp_packets:
                        if rest['frame_number'] in passed_frames:
                            continue
                        # if statements to handle if we looking for req or respond
                        if (not flip and pair['dst_port'] == rest['src_port'] and rest['src_ip'] == pair['dst_ip'] and rest['dst_ip'] == pair['src_ip']):
                            comm['packets'].append(rest)
                            passed_frames.append(rest['frame_number'])
                            flip = True

                        if (flip and pair['src_port'] == rest['src_port'] and rest['src_ip'] == pair['src_ip'] and rest['dst_ip'] == pair['dst_ip']):
                            comm['packets'].append(rest)
                            passed_frames.append(rest['frame_number'])
                            flip = False

                        # ended comm with error
                        if (int(''.join(rest['hexa_frame'].split()[42:44]), 16) == 5):
                            if rest['frame_number'] not in passed_frames:
                                comm['packets'].append(rest)
                                passed_frames.append(rest['frame_number'])
                            break

                    if comm and len(comm['packets']) > 1:
                        hexDecoded = comm['packets'][-1]['hexa_frame'].split()
                        hexDecoded2 = comm['packets'][-1]['hexa_frame'].split()
                        op_code = int(
                            hexDecoded[42]+hexDecoded[43], 16)
                        if op_code == 5:
                            complete_comms.append(comm)
                        elif op_code == 4 and len(hexDecoded2[46:]) < 512:
                            complete_comms.append(comm)
                        break

    data = {
        'complete_comms': complete_comms
    }
    return data
```

Výstup programu

Výstup programu je naformátovaný do súboru typu YAML. Na dosiahnutie tejto funkcionality bola použitá externá knižnica *ruamel.yaml*.

```
    f = open(
        "pks-output-{}.yaml".format(args.protocol if args.protocol else 'all'), "w")
    yaml = ruamel.yaml.YAML()
    yaml.default_flow_style = False
    yaml.indent(mapping=2, sequence=4, offset=2)
    yaml.dump(data, f)
```

Štruktúra udržiavania externých dát

Ako štruktúru som použil objekty vo formáte JSON, ktoré sú v tvare:

```
{
  ...
  "hexadecimálna hodnota / decimálna hodnota" : "názov protokolu/portu k danej hodnote"
  ...
}
```

Pri načítaní viem tieto dáta preformátovať na dátový typ dictionary v jazyku Python a tak ihneď pristupovať k daným hodnotám.

Popis používateľského rozhrania

Program je ovládateľný prepínačmi **-d** (--destination) a **-p** (--protocol)

V termináli:

```
python main.py -p <názov protokolu, pre analýzu komunikácií> -d <cesta k súboru *.pcap s rámcami>
```

Ani jeden zo spomínaných prepínačov nie je nutné zadať, nakoľko ak nie je na vstupe zadný žiaden protokol cez prepínač *-p*, tak program vypíše všetky analyzované rámce, bez komunikácií a ak nezadáme argument pomocou prepínača *-d*, tak program použije defaultnú cestu a to *./vzorky/eth-1.pcap*

Implementačné prostredie

Pre riešenie svojho zadania som si vybral programovanie prostredie jazyka Python, na koľko je tento jazyk veľmi jednoduchý zo stránky syntaxe a zároveň veľmi schopný na procesovanie komplexných dát vďaka veľkému množstvu vstavaných modulov a možnosťou rýchlo a efektívne rozširovať prostredie pomocou externých modulov. Vo svojej implementácií som použil z externých knižníc knižnicu menom Scapy, z ktorej som použil funkciu *rdpcap(),* ktorá slúži na čítanie rámcov zo súborov typu *.pcap* a knižnicu Binascii, ktorej úlohou je preložiť dáta načítané zo vstupného súboru na hexadecimálny reťazec, pomocou funkcie *hexlify()*. Ďalšou výhodou prostredia je fakt, že Python nie je nízko-úrovňový programovací jazyk, ako napr. jazyk C/C++, čo by nám pri spracovaní zadania, vytváralo nadbytočnú prácu s pamäťou.

Zhodnotenie

Vytvorený program spĺňa očakávané účely a funguje správne. Možnosť rozšírenia programu je pomerne jednoduchá nakoľko je program naprogramovaný dynamickými funkciami no spolieha sa na presne definovanie označenia niektorých protokolov v podmienkach. Možnosť upravovať databázy nemení funkcionalitu programu pokiaľ budú zanechané správne označenia protokolov.