项目文档：网络嗅探器

Qingru Zhang

1. 项目概述

本项目实现了网络嗅探器（SnifferZ）的基本功能，实现了本机包抓取和解析，并实现了用户友好的图形界面。目前支持 IP，IPv6，TCP，ICMP，IGMP，ARP 协议。项目基于 python3，使用 python 第三方库 Scapy 实现包抓取和包解析的基本功能，同时，实现了包过滤，包查找，数据包重组以及数据包的保存和读取等功能。

项目 UI 基于 PyQt5 实现，设计风格类似 WireShark。UI 组件分三个部件：主控件，网卡选择控件，过滤器设置控件。网卡选择控件在程序开始运行时实行网卡选择功能；主控件控制抓包和其他功能，并以表格形式展示抓取的数据包，以选项卡的形式展示包解析的详细结果；Filter 控件用于设定包过滤的 BPF 规则。本项目尽可能实现友好的界面，拓展了右键菜单功能，实现表格自动滚屏和表头自适应调整等功能。

1.1 环境配置

项目基于 python3 实现，测试环境需要 python 3.4 及以上版本。依赖的第三方库如下：

表 1.1.1 第三方库依赖

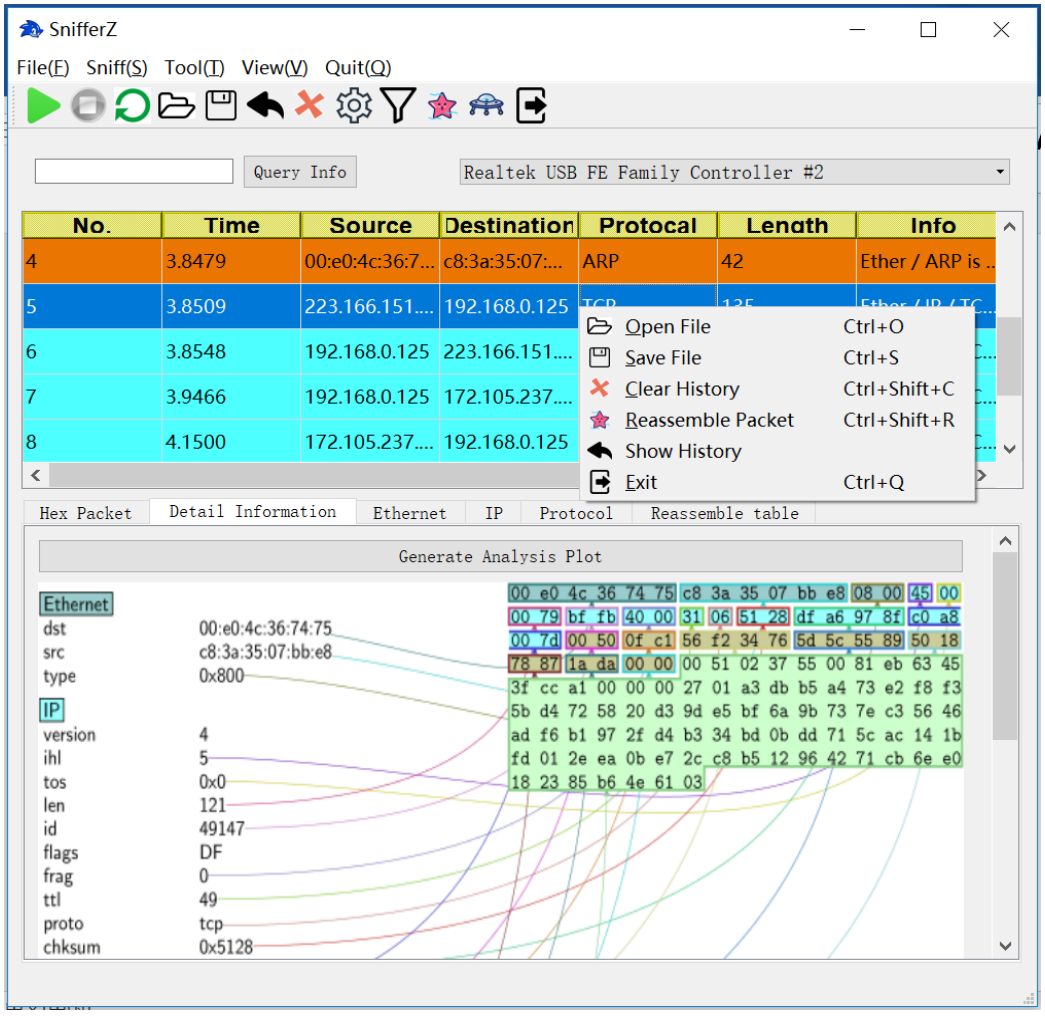
第三方库	版本信息
Scapy	Scapy 2.4.0 (python 3.6)
PyQt5	PyQt5 5.9 (python 3.6)
PyX	PyX 0.14.1 (python 3.6)

其中 PyX 用于绘制包解析的结构图，如果没有此第三方库，包解析结果的第二选项卡 Detail Information 无法正常展示，其他功能不受影响。

表 1.1.2 测试环境信息

项目	配置信息
OS	Microsoft Windows 10 x64-based PC
Processer	i7-7200u 2901 MHz
Physical Memory	DDR4 8,002 MB
Hard Disk	256G 7200r
Python	Python 3.6.1 :: Anaconda 4.4.0 (64-bit).

由于 python 的跨平台属性，该项目亦可在 Linux 环境下运行，但由于绘制包解析的结构图的绘制依赖 window 平台下的 PDF 转化工具，因此，Linux 环境下，该选项卡的内容无法正常展示。程序执行效果如下图：



1.2 程序文件列表

表 1.2.1　程序文件列表

soruce/	源代码文件夹
source/dependency/	依赖库文件夹
VirtualEnv/	虚拟环境文件夹
source/img/	UI 内使用的 icon 图标和项目 Logo
source/tool/	包解析依赖的外部工具
source/main.py	程序执行入口，执行命令：python main.py
source/GUIDesign.py	定义 UI 主控件的文件
source/InterFaceChoose.py	UI 初始执行时，网卡选择控件的定义文件
source/FilterWidget.py	设置包嗅探过滤规则的 Filter 控件
source/SnifferThread.py	定义包嗅探的多线程继承类
source/MyTabel.py	主控件中表格展示控件的定义文件
source/MyTabWidget.py	包解析选项卡控件的定义文件
source/FileThread.py	文件处理线程
README.md	说明文档

2. 数据结构与程序结构

2.1 主窗口控件

QMainWindow 的继承类 GUI_Design 是程序的主线程，也是 UI 的主窗口，用于创建展示 UI，控制其他控件，创建抓包的子线程 SnifferThread，控制文件处理子线程 FileThread。GUI_Design 的属性 packet_history 以字典的形式保存了抓取的全部数据包，

键值为抓取时的顺序号。

2.2 辅助控件

其他 UI 控件为 QWidget 的继承类，用于辅助实现主窗口的功能：

IfaceWidget 为实现程序开始运行时网卡选择的功能，当用户从下拉栏选择网卡并确认后，主窗口才会出现以执行后续功能。效果图如下

图 2.2.1　程序初始执行效果图

FilterWidget 用于设置嗅探时的包过滤规则，通过主窗口的控制按钮来显示，其过滤规则使用 BPF 语法，用户可以在跳出的规则设置窗口设置勾选并输入相应的过滤规则。

Result_Table 是表格控件，用于展示抓取的数据包条目，其实现了自动滚屏，头部宽度自适应调整等 UI 调整功能。

No.	Time	Source	Destination	Protocal	Length	Info
2	1.7194	188.172.233....	192.168.0.125	TCP	134	Ether / IP / TC...
3	1.8221	192.168.0.125	188.172.233....	TCP	54	Ether / IP / TC...
4	3.3350	192.168.0.125	59.110.168.1...	TCP	55	Ether / IP / TC...
5	3.4368	59.110.168.1...	192.168.0.125	TCP	66	Ether / IP / TC...
6	4.1489	223.166.151....	192.168.0.125	TCP	135	Ether / IP / TC...

图 2.2.2　result table 效果图

MyTableWidget 是包解析与结果展示的选项卡控件，**双击** Result_Table 中的条目将触发 MyTableWidget 解析并展示相应的数据包，其一共含有四个选项卡：Hex Packet 选项卡用于展示数据包的原始十六进制信息；Detail Information 选项卡用于展示数据包结构图；Ethernet 选项卡用于展示数据链路层以及上的解析结果；IP 选项卡用于展示 IP 层以上的解析结果；Protocol 选项卡用于展示传输层及以上的解析结果；Reassemble table 为表格控件，用于展示分段重组的结果，通过多选 result table 中的数据包条目，点击分段重组按钮，Reassemble table 将以 IP 层 id 为键值展示分段重组结果，同时，双击其中的条目，将在选项卡中展示重组数据报的解析结果。重组结果作为属性值保存在 MyTableWidget 控件内。

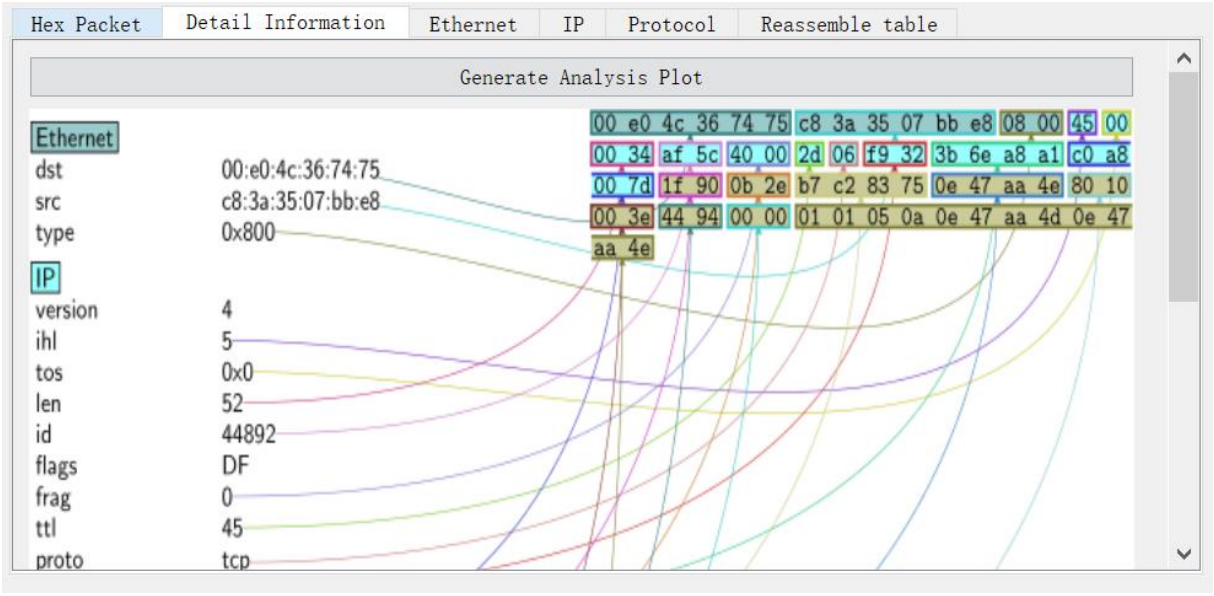


图 2.2.3 选项卡效果展示

CentralWidget 为主窗口的子控件，用于控制 result table 和 mytabwidget 间的信号通信和联动，同时，完成和主窗口的数据通信。

2.3 子线程

GUI_Design 作为主线程以列表的形式保存抓取报的原数据，其通过创建子线程

SnifferThread 来抓取数据包，每个 SnifferThead 都有结束该线程的函数槽 join，GUI_Design 中的线程结束信号与每一个子线程的结束槽相连，并在创建新的线程前，发射线程结束信号，确保每次嗅探只开启一个线程。

FileSaveThread 为 GUI_Design 创建的文件保存子线程，在执行数据包选择保存功能时使用。PDFthread 是 GUI_Design 创建的数据包结构分析线程，用户点击 Detail Information 中的 Generate Analysis Plot 时，GUI_Design 将会创建该线程来分析所选的数据包，生成结构分析图，并将生成的 pdf 转化为 png 格式展示在选项卡中。

程序总结构图如下：

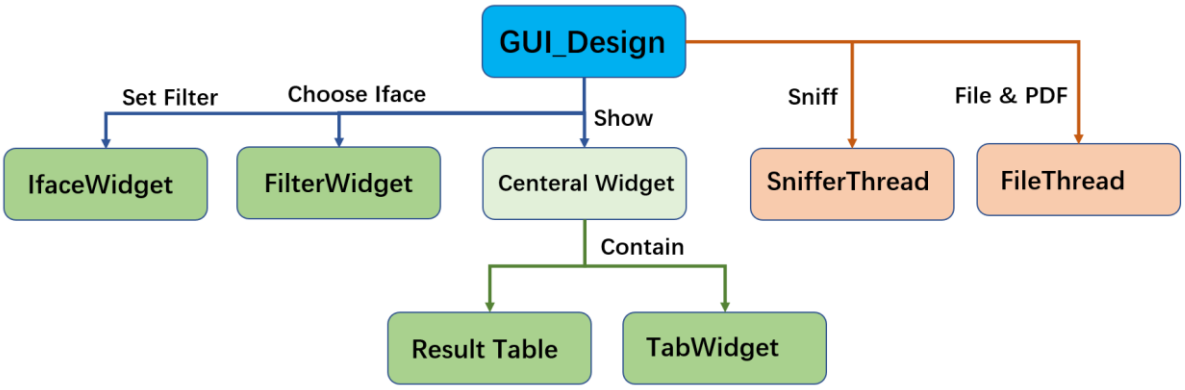


图 2.2.4 程序设计结构图

3. 主要算法

3.1 控制包抓取子线程

主线程 GUI_Design 创建子线程 SnifferThread 来抓取数据包，而在创建新线程之前，要确保之前嗅探线程已全部退出，因此，在 SnifferThread 中设置 join() 函数槽与 stop_sniff_signal 结束信号，stop_sniff_signal 结束信号均与 join 相连，而 GUI_Design 在创建信号时，将自身的结束进程信号传递给子线程，使子线程的 stop_sniff_signal 是主线程结束信号的引用，这样每当主线程发射结束信号，所有创建的 SnifferThread 子线程都会调用 join 回调函数，来结束自身。主线程创建子线程的代码如下：

```
def StartSniff(self):
    if self.pkt_num == 0:
        self.stop_sniff_signal.emit(True)
        self.clear_all()
        self.startTime = time.time()
        self.sniff_thread = SnifferThread.SnifferThread(UI_stop_signal = self.stop_sniff_signal, iface = self.iface_choose,
                    filter_UI = self.filter_UI )
        self.sniff_thread.start()
        self.sniff_thread.update_ResultTable_signal.connect(self.Update_ResultTable)
```

图 3.1.1 主线程创建子线程

子线程定义如下:

```
class SnifferThread(QtCore.QThread):

    update_ResultTable_signal = QtCore.pyqtSignal(int, Packet)

    def __init__(self, UI_stop_signal = None, iface = conf.iface, parent = None, pkt_num = 0, fi

        super(SnifferThread, self).__init__(parent)

        self.dkpt = None
        self.filter = filter_UI
        self.pkt_num = pkt_num
        self.iface = iface
        self._stop_event = threading.Event()
        self.stop_sniff_signal = UI_stop_signal

        self.stop_sniff_signal.connect(self.join)

    def run(self):
        print('**Start Sniff')
        self.dkpt = sniff(iface = self.iface, filter = self.filter, prn = self._sniff_callback,
                          stop_filter = lambda p: self._stop_event.is_set())
        print('**End Sniff')

    def _sniff_callback(self, pkt: Packet):
        self.pkt_num += 1
        self.update_ResultTable_signal.emit(self.pkt_num, pkt)

    def join(self, isStop):
        if isStop:
            self._stop_event.set()
            print('*** Set Join Event On: %d'%self.currentThreadId())
```

图 3.1.2 SnifferThread 定义

其中 sniff 函数启动 stop_filter 回调函数来检查是否需要结束嗅探。抓取一个数据包后回调 _sniff_callback 函数将抓取的包发射给主线程。

3.2 包解析函数

Sniff 函数抓取的包以 scapy.packet 对象，其部分属性值如下:

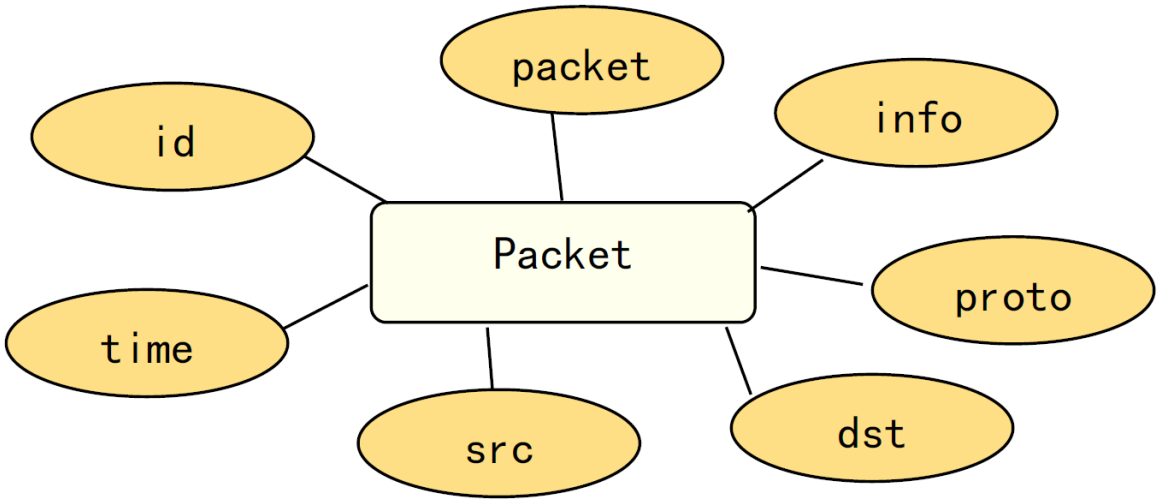


图 3.2.1　数据包属性

通过调用 scapy.packet 对象的方法 getlayer 可以直接得到各层的首部和数据字段中的信息，部分解析函数如下：

```
def get_packet_proto(pkt: Packet):
    summary_pkt = pkt.summary()
    sum_list = summary_pkt.split('/')
    if 'ARP' in sum_list[1]:
        return 'ARP'
    elif 'DARP' in sum_list[1]:
        return 'DARP'
    elif 'DHCP' in sum_list[1]:
        return 'DHCP'
    elif 'IPv6' in sum_list[1]:
        return 'IPv6' + '/' +sum_list[2].strip().split(' ')[0]
    elif 'IP' in sum_list[1]:
        if 'Raw' in sum_list[-1] or 'Padding' in sum_list[-1]:
            layer3 = sum_list[-2].strip()
        else:
            layer3 = sum_list[-1].strip()
        proto = layer3.split(' ')[0]
        return proto
    else:
        proto = sum_list[1].split(' ')[0]
        proto = proto.strip() + '/' + sum_list[2].split(' ')[1]
        return proto
```

图 3.2.2 协议信息的获取

数据包的数据区默认以 ASCII 码解码，不在可显示字符范围（32-127）内的二进制数（0-31，128-255），以十六进制形式显示。

3.3 数据包重组

当路由器收到一个数据包时，它会检查目的地址并确定出接口的使用和该接口的MTU。如果分组大小大于 MTU，则该分组将被分段，分段的分组其 flags 的 DF 位将被置 0，MF 位将被置 1 。依据 flags 标志位，判断分组是否被分段，并依据 IP 首部的 frag 信息对分段进行重组。需要注意的时，重组后需要重新设置长度，标志位和偏移量。

```
42  def Reassemble_packet(plist):
43      id_dict = {}
44      for pkt in plist:
45          if str(pkt[IP].id) not in id_dict.keys():
46              id_dict[str(pkt[IP].id)] = PacketList()
47              id_dict[str(pkt[IP].id)].append(pkt)
48          else:
49              id_dict[str(pkt[IP].id)].append(pkt)
50
51      result_dict = {}
52      for id_key in id_dict.keys():
53          tmp_dict = {}
54          for pkt in id_dict[id_key]:
55              tmp_dict[str(pkt[IP].frag)] = pkt
56          try:
57              result_dict[id_key] = tmp_dict['0']
58          except:
59              return None
60          loads = b''
61          for frag in sorted(tmp_dict.keys()):
62              loads = loads + tmp_dict[frag].getlayer(Raw).load
63
64          result_dict[id_key].len += len(loads) -  len(result_dict[id_key][Raw].load)
65          result_dict[id_key][Raw].load = loads
66          result_dict[id_key].flags = 2
67          result_dict[id_key].frag = 0
68      return result_dict
```

图 3.3.1　分段重组函数

3.4 包过滤

FilterWidget 用于设置主线程的过滤规则，过滤规则使用 BPF 语法，在 FilterWidget 的输入框前有相应的提示。主线程将设置好的过滤规则传递给 SnifferThread，在每次调用 sniff 函数时使用此规则：

```
1.  self.sniff_thread = SnifferThread.SnifferThread(UI_stop_signal = self.stop_sniff_sig
    nal, iface = self.iface_choose, filter_UI = self.filter_UI )
2.  ################################################################################
3.  self.dkpt = sniff(iface = self.iface, filter = self.filter, prn = self._sniff_callba
    ck,  stop_filter = lambda p: self._stop_event.is_set())
```

过滤规则如下：

```
'host' : '[src|dst] host <host ip>',
'ether host' : 'ether [src|dst] host <MAC>' ,
'vlan' : 'vlan <ID>' ,
'portrange' : '[src|dst] portrange <p1>-<p2> or [tcp|udp] [src|dst] portrange <p1>-<p2>' ,
'proto' : '[ip|ip6][src|dst] proto <protocol>' ,
'port' : '[ip|ip6][tcp|udp] [src|dst] port <port>' ,
'net' : '[src|dst] net <network>' ,
'tcpflags' : '[ip|ip6] tcp tcpflags & (tcp-[ack|fin|syn|rst|push|urg|)' ,
'Fragmented' : 'Fragmented IPv4 packets (ip_offset != 0)' ,
```

3.5 数据包保存与打开

可以从 result table 中选择条目，然后保存选定的数据包为 cap 文件，文件保存读取代码如下：

```
def OpenFile(self):
    fileName, filetype = QFileDialog.getOpenFileName(self, "选取文件", "./",  "WireShark Tcpdump (*.cap);;All
    pkt_from_file = Scapy.rdpcap(fileName)
    self.clear_all()
    for i,pkt in enumerate(pkt_from_file):
        self.packet_history[str(i)] = pkt
        SingelRow = self.build_pkt_dict(i+1,pkt)
        self.central_Widget.Insert_row_given_index(i, SingelRow)
    self.pkt_num = i+1

def SaveFile(self):
    indexs = self.central_Widget.table_result.tablewidget.selectedIndexes()
    row_indexs = set(index.row() for index in indexs )
    print(row_indexs)
    if row_indexs:
        Pdict = {}
        for index in row_indexs:
            Pdict[str(index)] = self.packet_history[str(index)]
    else:
        Pdict = self.packet_history
    save_path, ok2 = QFileDialog.getSaveFileName(self, "文件保存", "./", "WireShark Tcpdump (*.cap);;All Files
    save_thread = FileThread.FileSaveThread(save_path, Pdict, parent = self)
    save_thread.start()
```

3.6 数据包查询

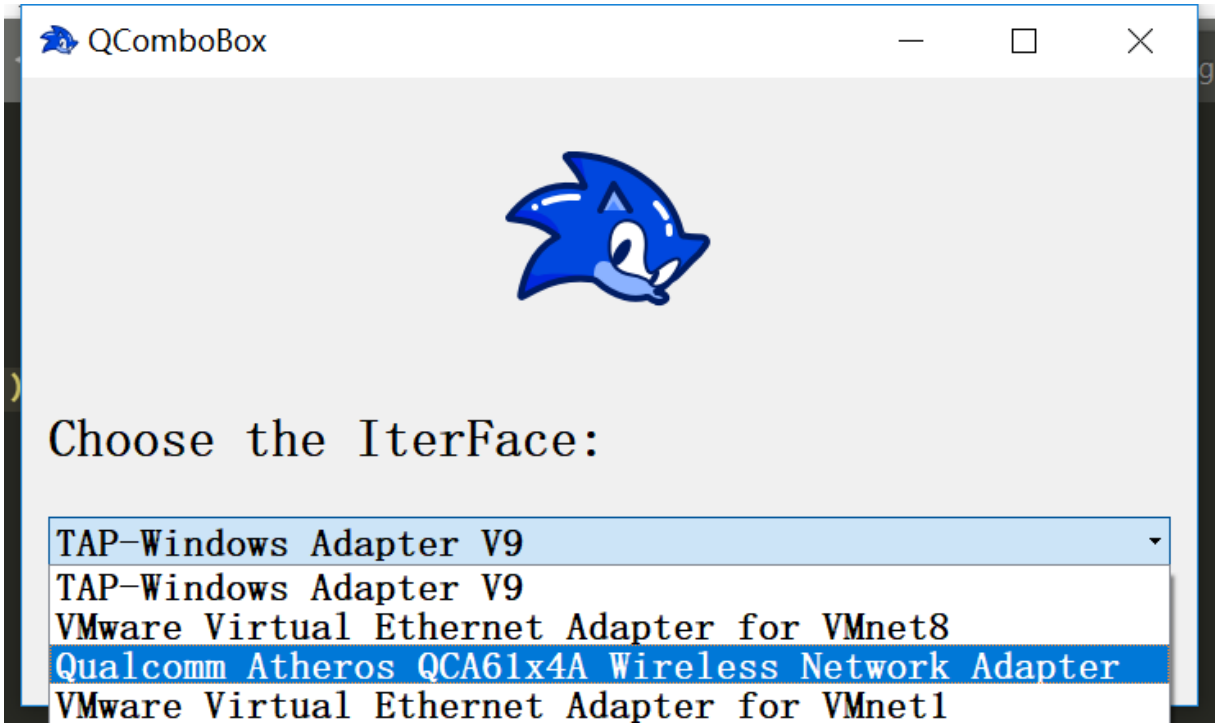
从主控件的输入行中获取查询字符串，对 packet_history 中保存的数据包查询其 repr(pkt)中内容，将查询结果显示在表格中。如果要重新显示抓包历史，可点击 return 按钮。数据包查询代码如下：

```
487   def search_info(self):
488       self.search_word = self.central_Widget.get_search_word()
489       print('Search Word:',self.search_word)
490       if self.search_word:
491           self.StopSniff()
492           self.central_Widget.clear_table_result()
493           for index_key in self.packet_history.keys():
494               if self.search_word in repr(self.packet_history[index_key]):
495                   dict_tmp =  self.build_pkt_dict(index_key ,self.packet_history[index_key])
496                   self.central_Widget.Insert_one_row_to_Result_Table(dict_tmp)
```

4. 程序测试

4.1 指定网卡侦听，并解析数据包

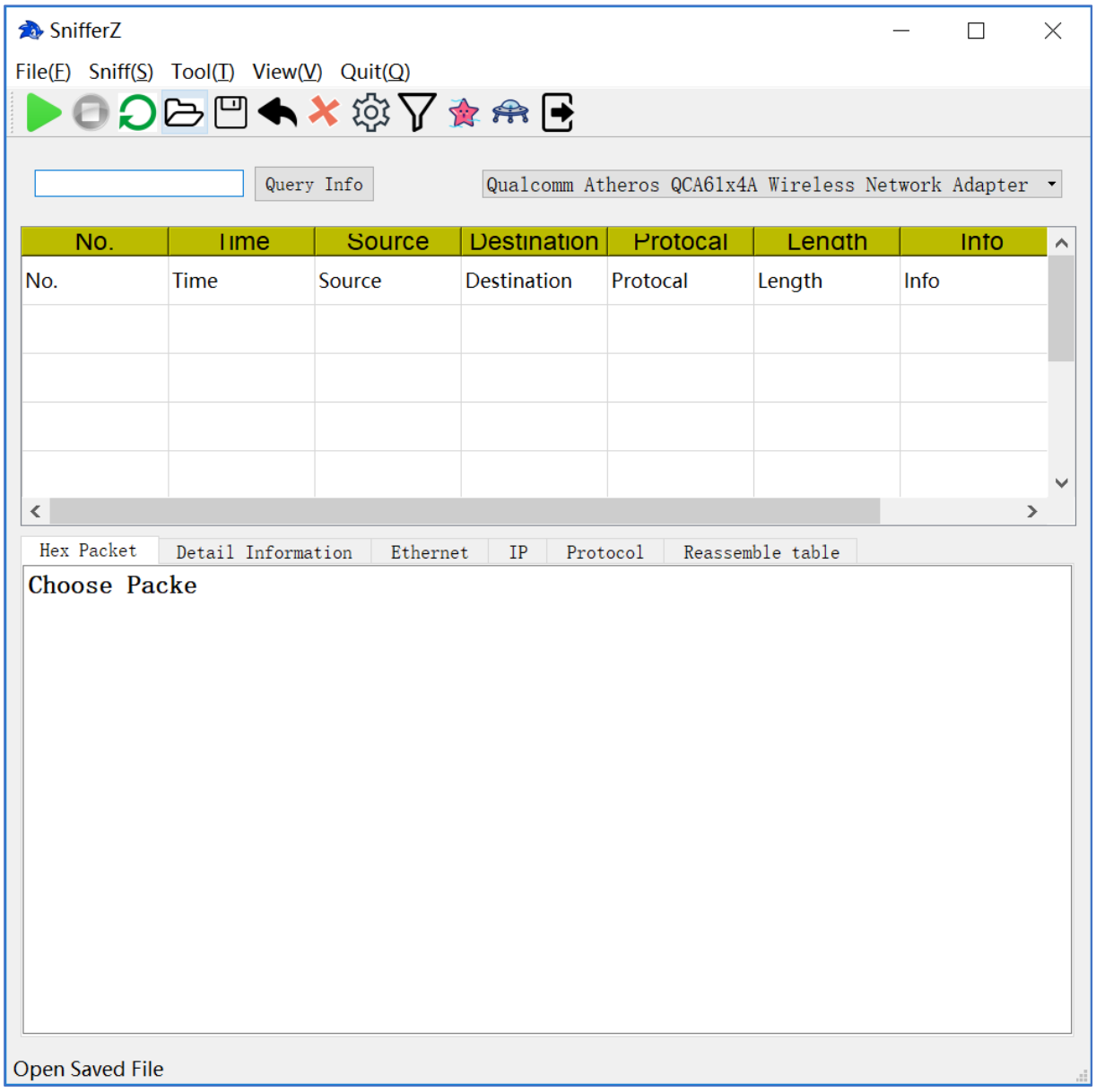
python main.py 运行主程序，初始时进入网卡选择界面，选择要侦听的网卡为无线网卡：



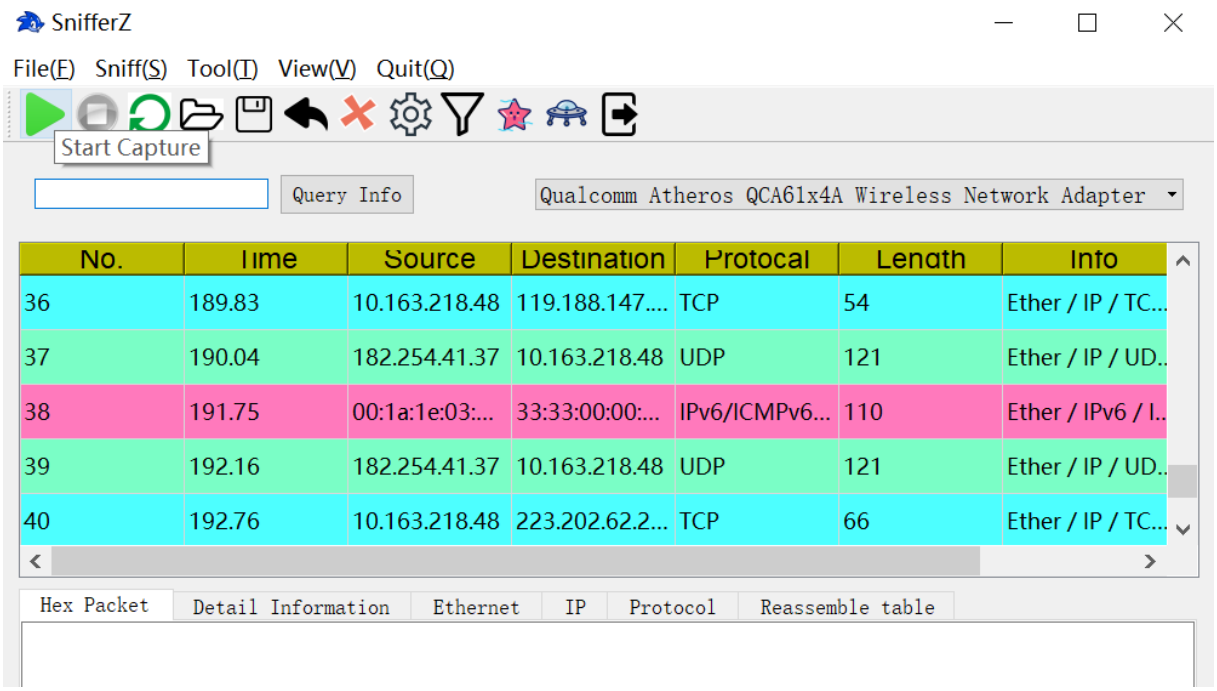
由 ipconfig 得知网卡的 ip 为 10.163.218.48

```
无线局域网适配器 WLAN:

   连接特定的 DNS 后缀 . . . . . . . :
   IPv6 地址 . . . . . . . . . . . . : 2403:d400:1001:3:4cf6:c75:eca1:fd43
   临时 IPv6 地址. . . . . . . . . . : 2403:d400:1001:3:5d50:c55d:e89b:4d3f
   本地链接 IPv6 地址. . . . . . . . : fe80::4cf6:c75:eca1:fd43%13
   IPv4 地址 . . . . . . . . . . . . : 10.163.218.48
   子网掩码 . . . . . . . . . . . . : 255.255.0.0
   默认网关. . . . . . . . . . . . . : fe80::1a:1e02:5b03:5d60%13
                                       10.163.0.1
```

程序进入主界面，其中网卡下拉选择框中显示着当前选择的网卡为无线网卡。

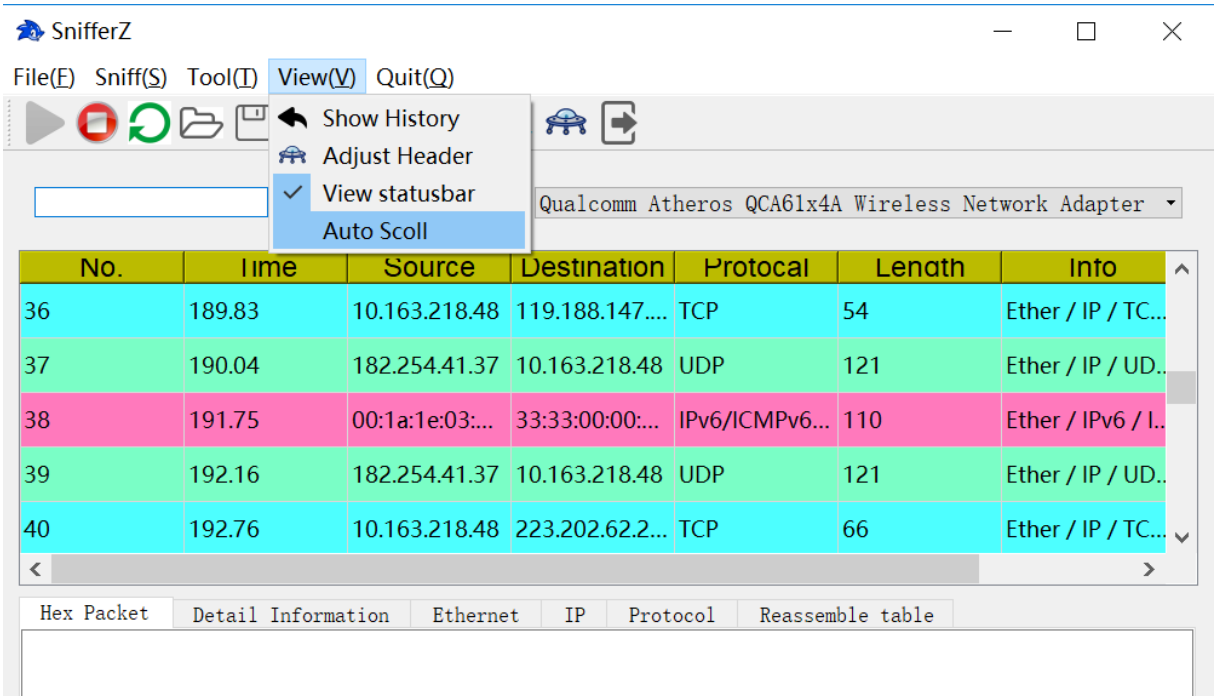


点击工具栏中的 Start Capture，开始侦听流经网卡的数据包：



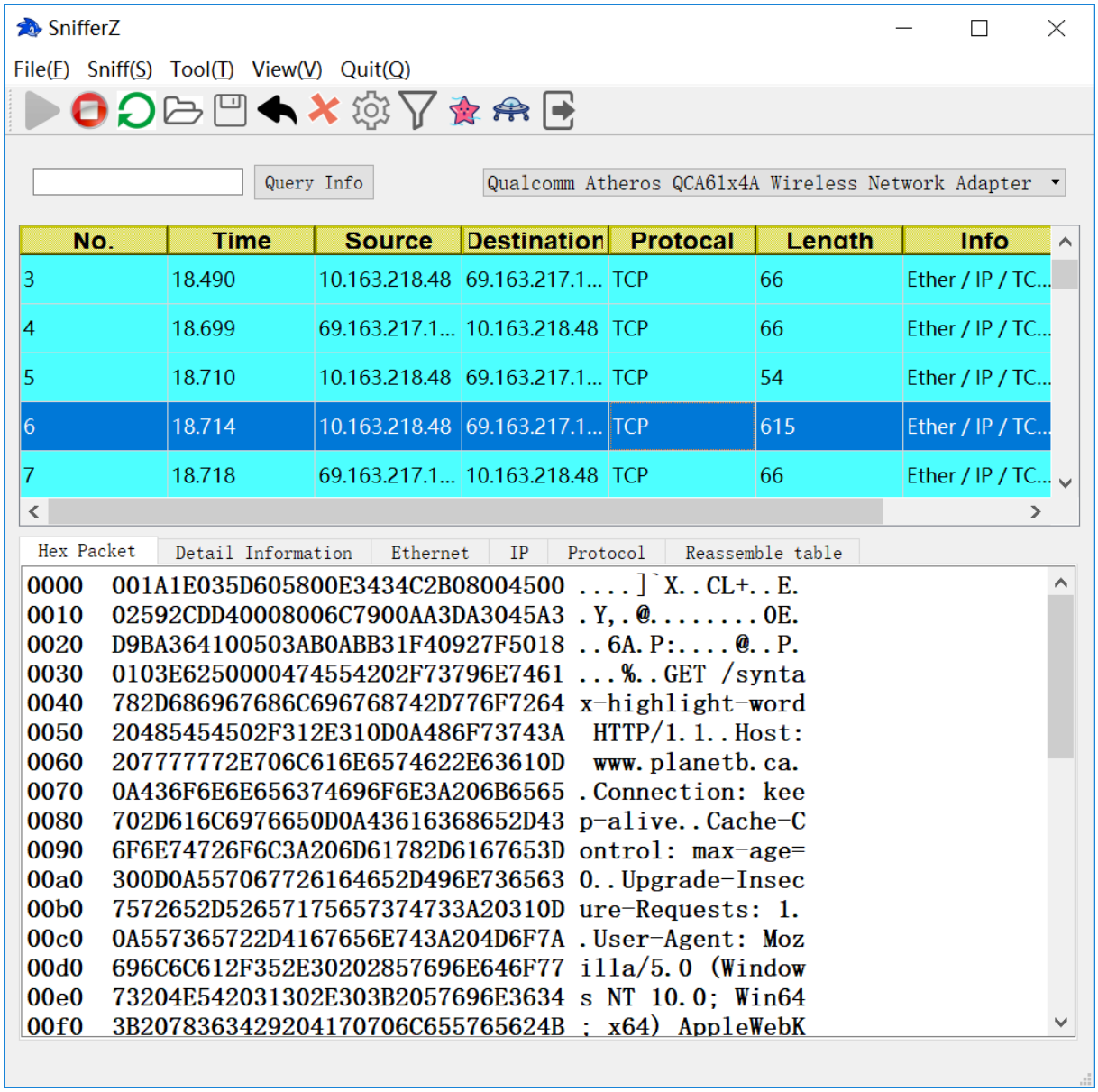
由表格中显示的目的 IP 得知，侦听目标为无线网卡。

从菜单栏中可以取消勾选自动滚屏，并可以自适应调整表头宽度：

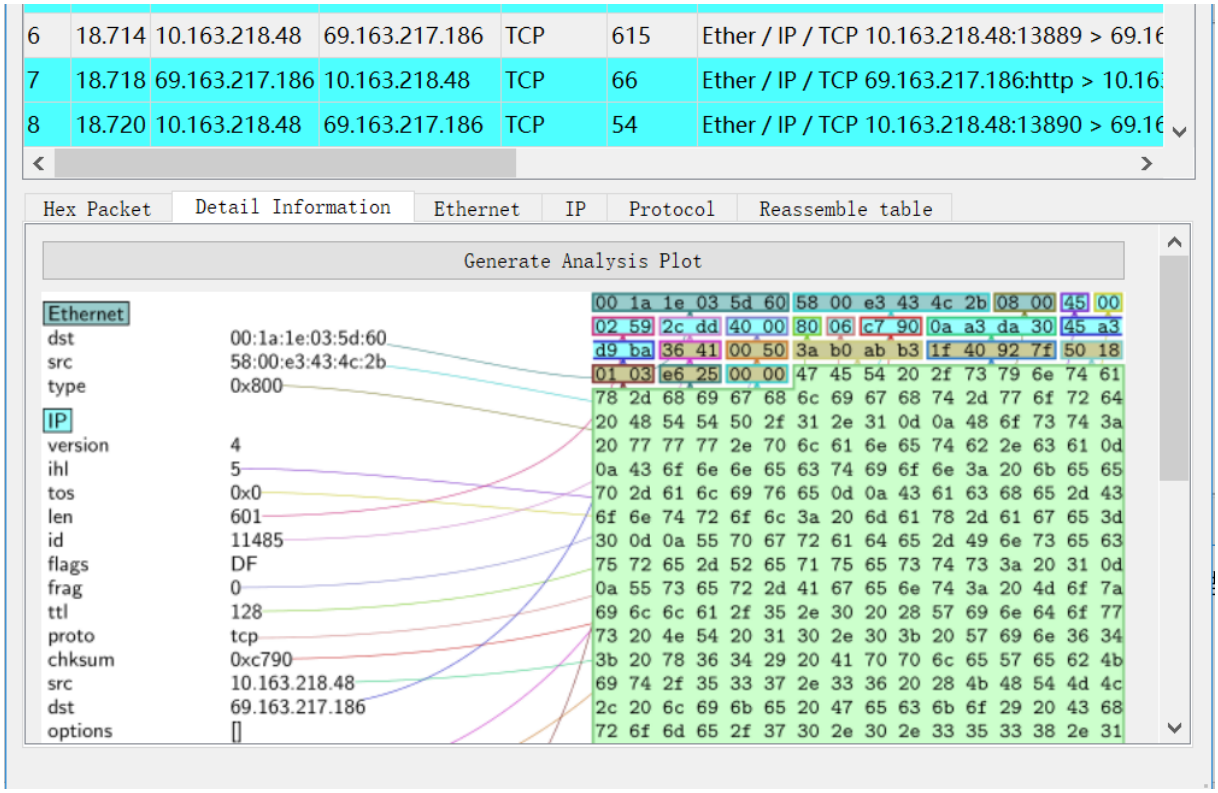


双击表格中的任一条目，将解析选中的数据包，解析结果将现实在选项卡中：

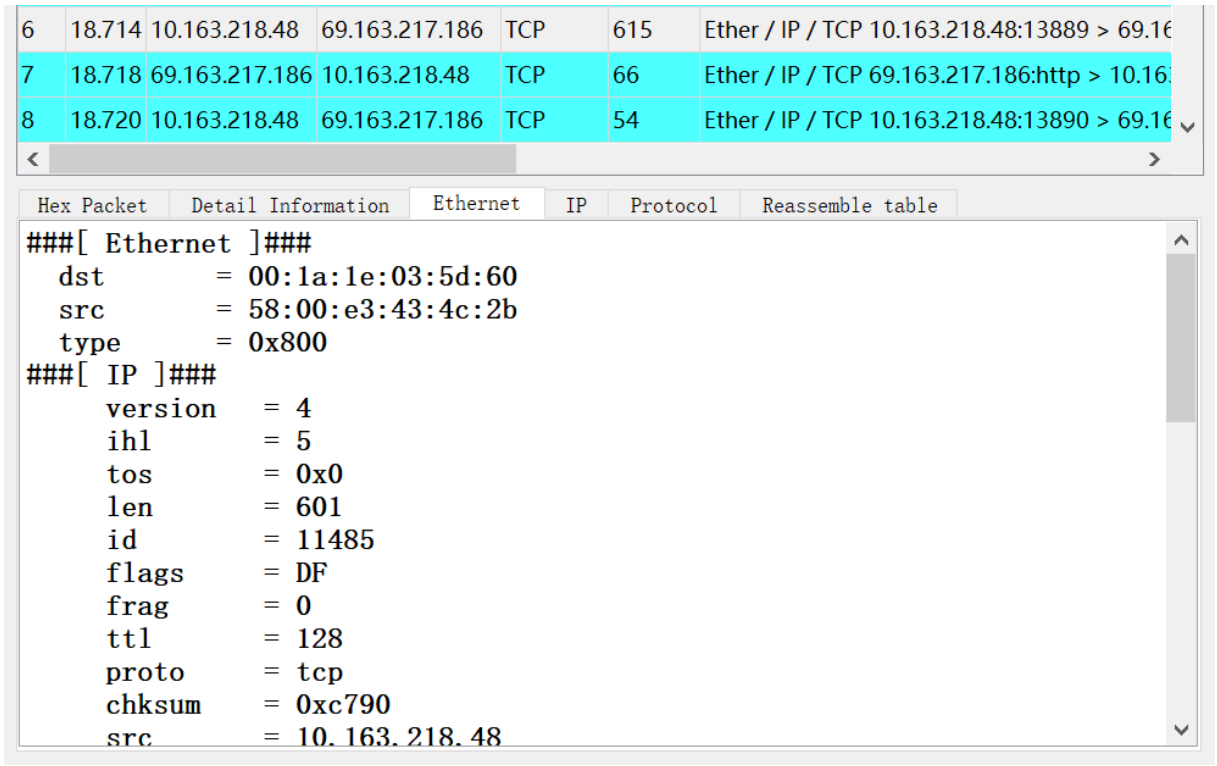
十六进制原数据：



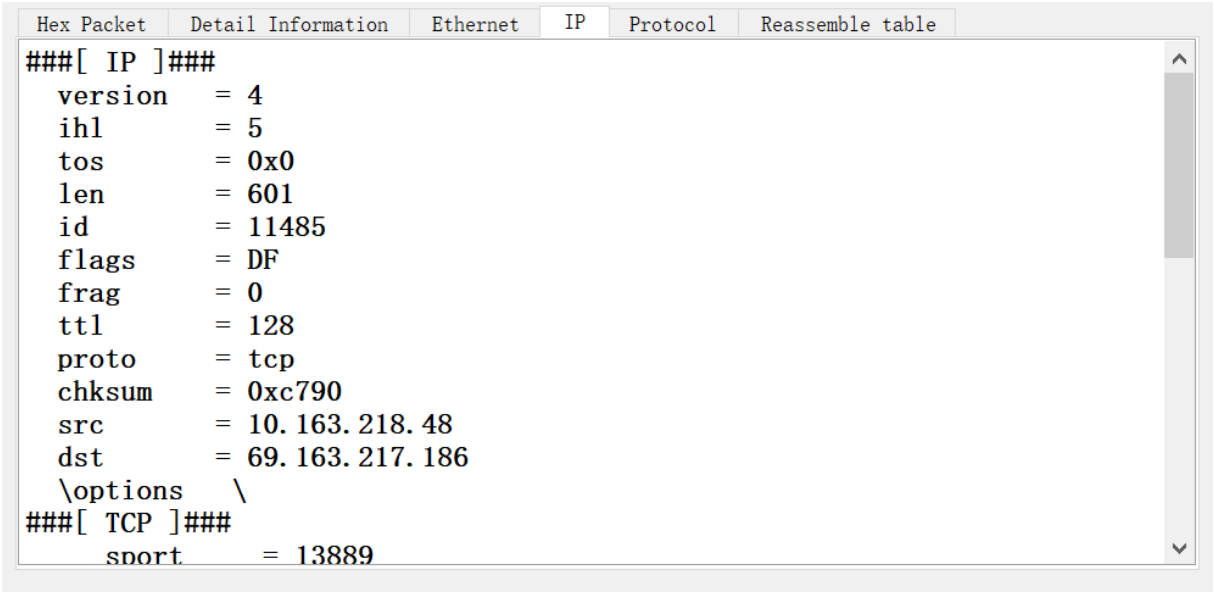
点击 Detail Information 中的 Generate Analysis Plot 按钮，生成包解析结构图：



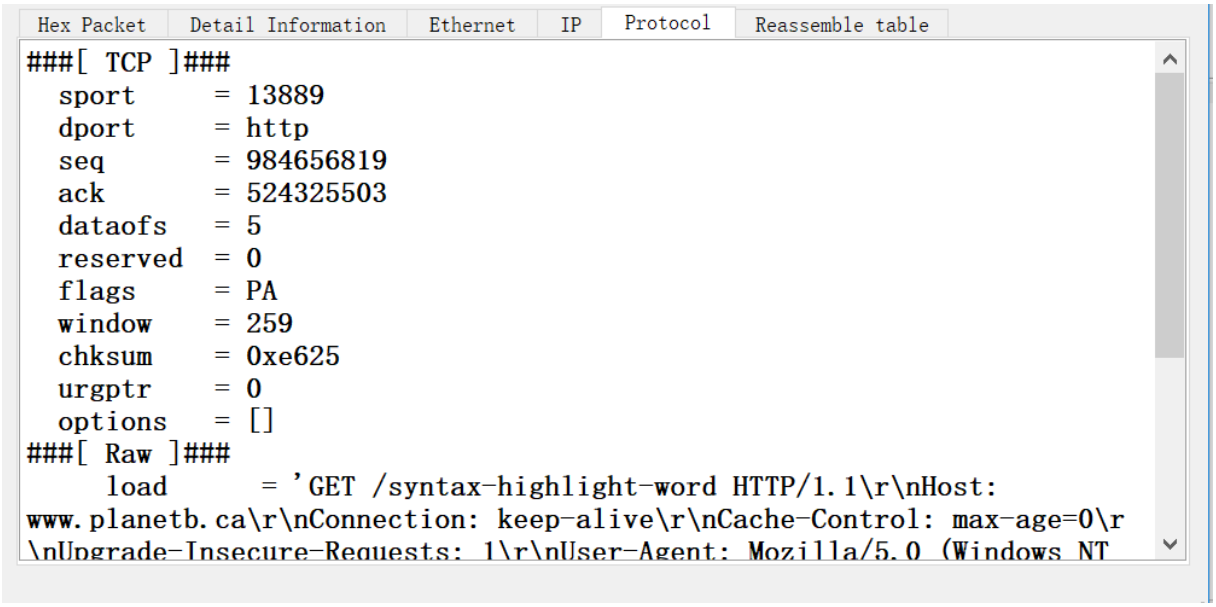
数据链路层及以上的解析结果:



IP 层以及上的解析结果:



传输层及以上的解析结果：

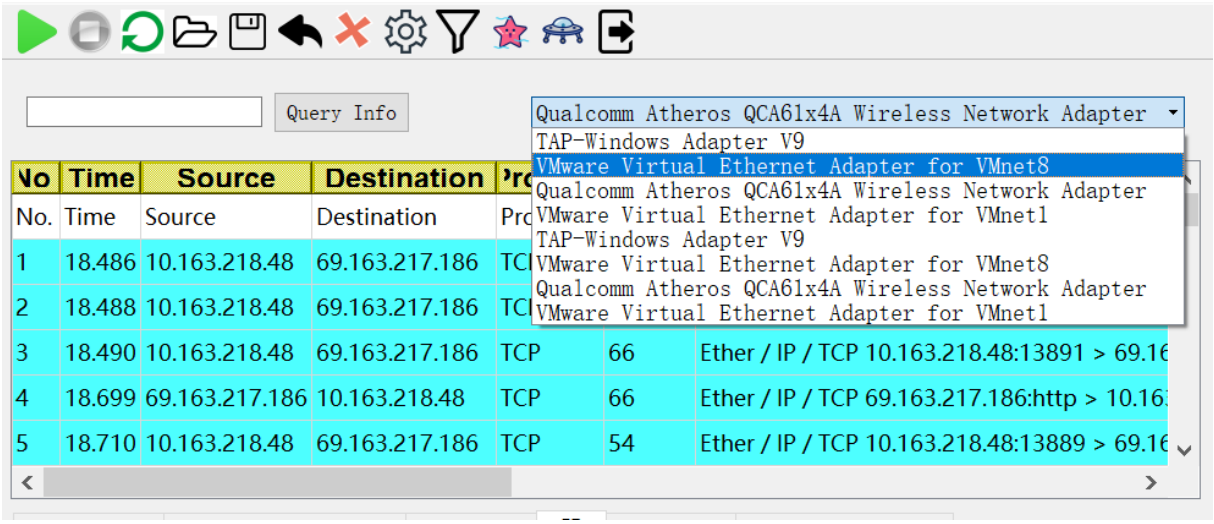


其中数据区部分按 ASCII 解码，可解码的部分具有可读性，不可解码或打印的字节（0-31，128-255）按十六进制原格式表示：

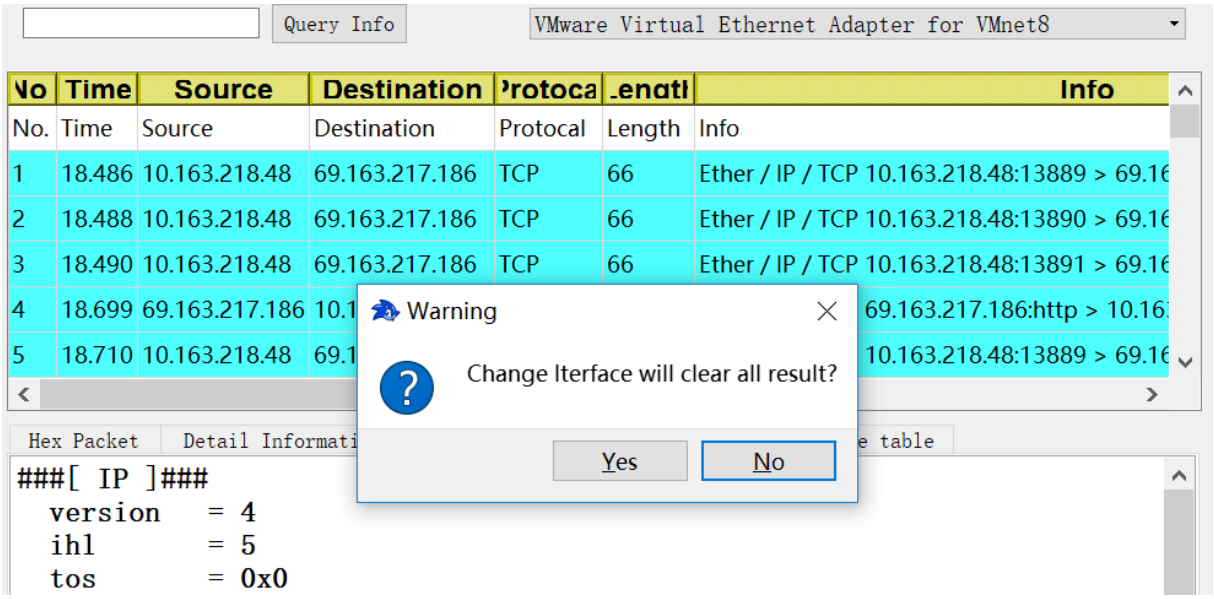
```
###[ Raw ]###
      load       = 'GET /syntax-highlight-word HTTP/1.1\r\nHost:
www.planetb.ca\r\nConnection: keep-alive\r\nCache-Control: max-age=0\r
\nUpgrade-Insecure-Requests: 1\r\nUser-Agent: Mozilla/5.0 (Windows NT
10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
70.0.3538.110 Safari/537.36\r\nDNT: 1\r\nAccept: text/html,application/
xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8\r
\nAccept-Encoding: gzip, deflate\r\nAccept-Language: zh-CN,zh;q=0.9,en-
US;q=0.8,en;q=0.7\r\nCookie:
sc_is_visitor_unique=rx5681546.1546514045.0849CA058D7D4F8ABFFB6C43794B4
90F.5.5.5.5.5.5.5.5.5\r\n\r\n'
```

4.2 网卡的选择

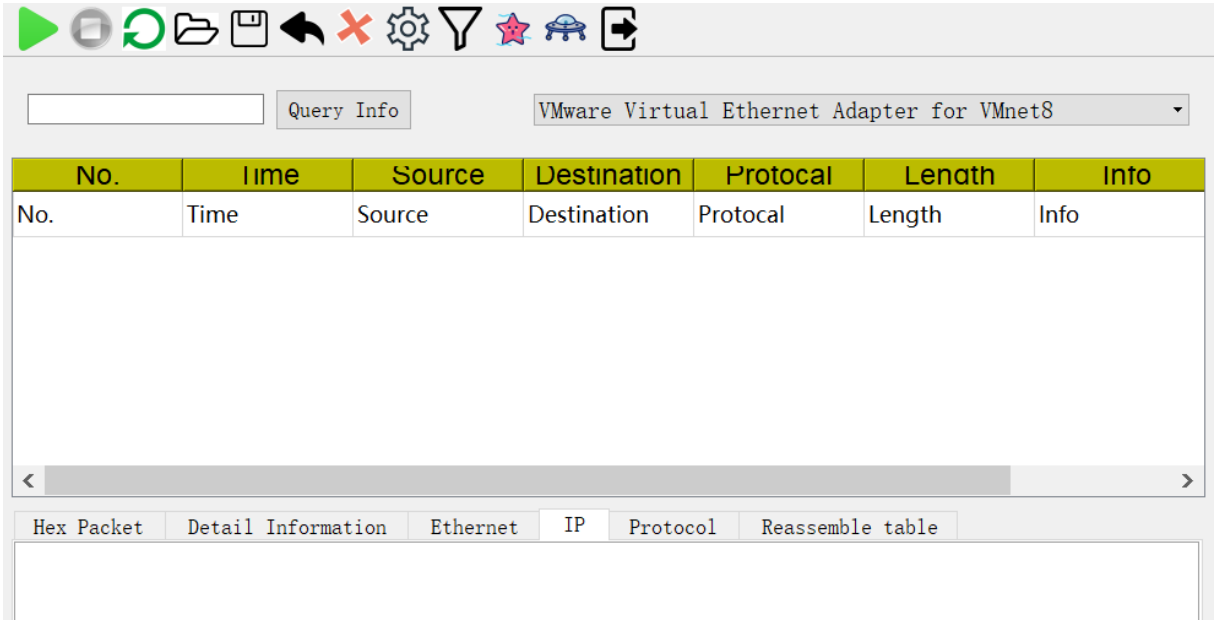
在主界面内可直接更改选择侦听的网卡，但更改后会停止当前侦听，并清空历史数据：



选择虚拟网卡 VMnet8 作为侦听对象，弹出警示窗：



点击确认。Sniffer 重新开始侦听：



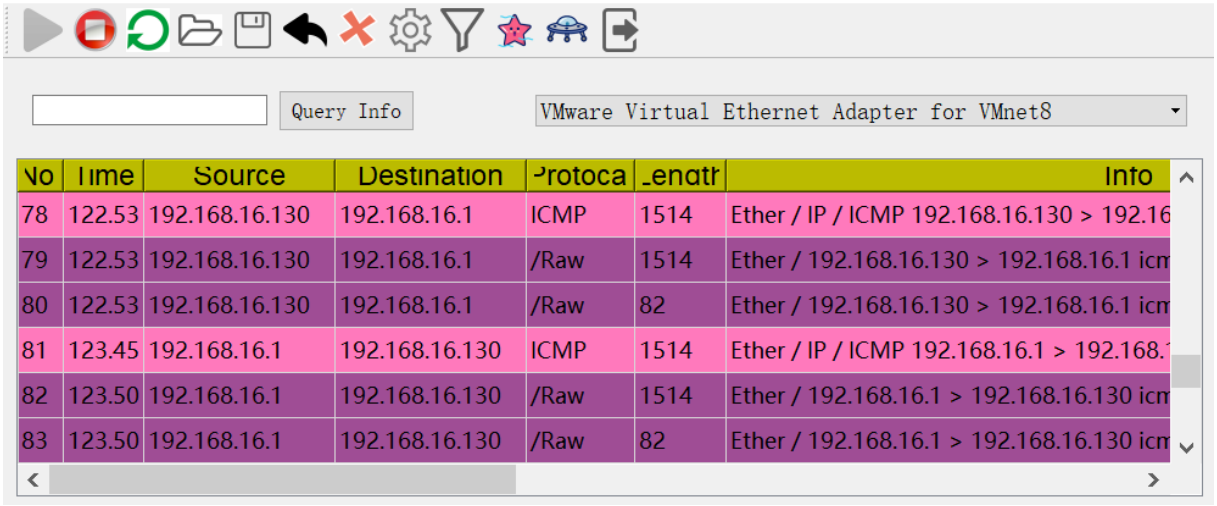
之后可按 4.1 中的操作重新开启侦听不同的网卡。

4.3 数据包重组

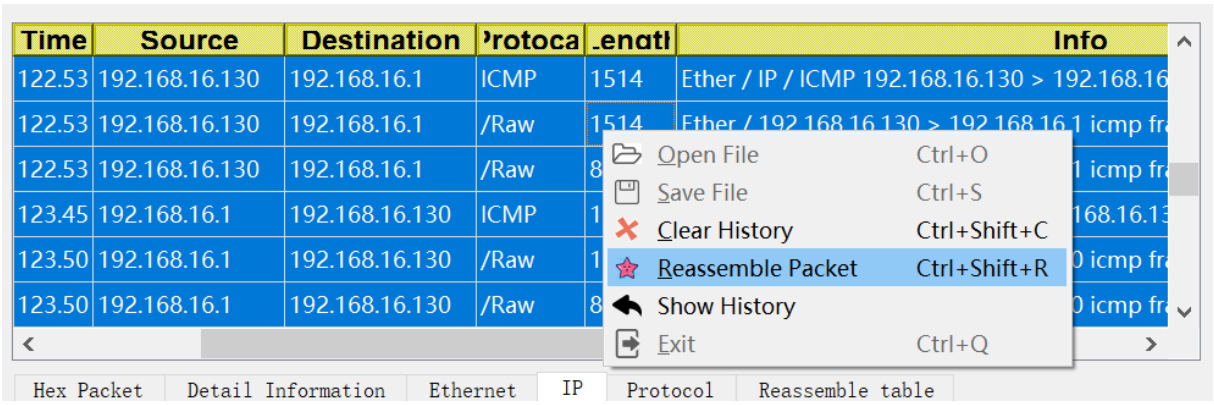
数据包重组测试，使用 ping 命令指定发送数据包的大小，从而强制执行分组。测试时，侦听虚拟网卡 VMnet8 192.168.16.1，使用该网卡去 ping 处于同一虚拟子网的虚拟机 192.168.16.130 。ping 构造如图：

```
D:\Learning\Computer Network\CourseDesign\Sniffer (master -> origin)
λ ping -l 3000 -S 192.168.16.1 192.168.16.130

正在 Ping 192.168.16.130 从 192.168.16.1 具有 3000 字节的数据:
来自 192.168.16.130 的回复: 字节=3000 时间<1ms TTL=64
来自 192.168.16.130 的回复: 字节=3000 时间=1ms TTL=64
来自 192.168.16.130 的回复: 字节=3000 时间<1ms TTL=64
来自 192.168.16.130 的回复: 字节=3000 时间=1ms TTL=64

192.168.16.130 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms, 最长 = 1ms, 平均 = 0ms
```

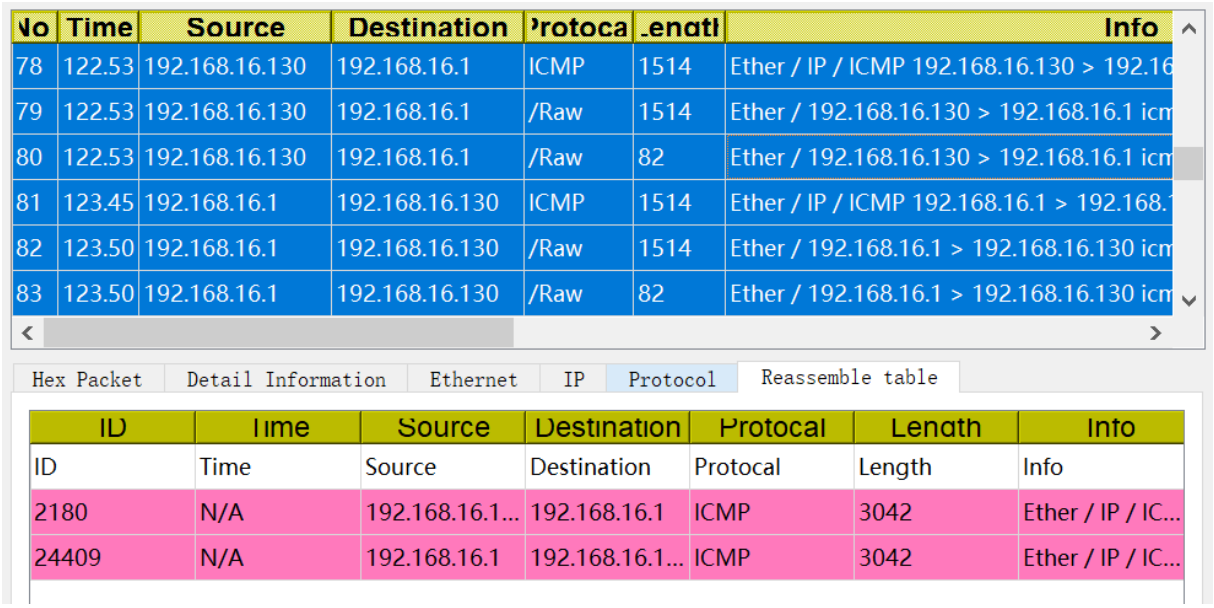
侦听结果如图：



选中需要分段重组的数据包，点击 Reassemble Packet 按钮，在 Reassemble Table 选项卡中显示重组结果：

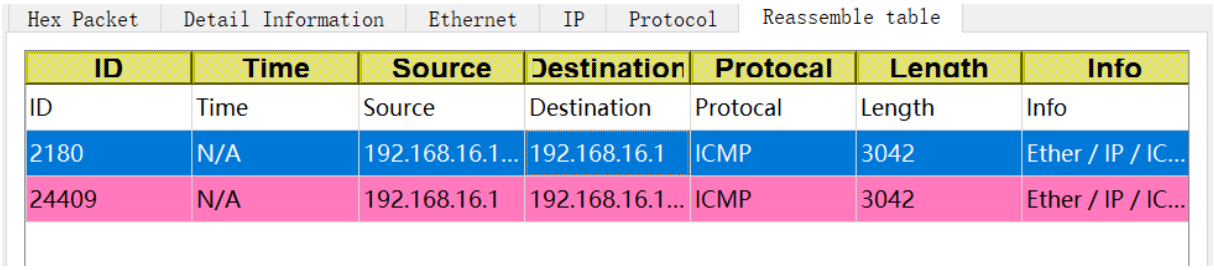


Reassemble Table 选项卡的显示如下：



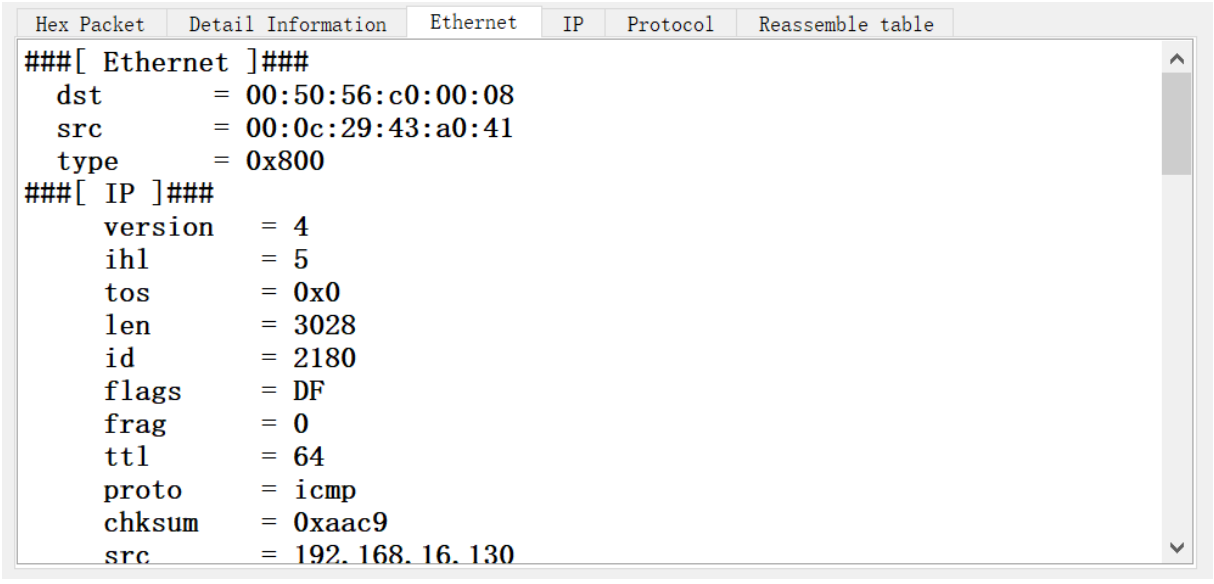
其表格中的键值 ID 标识着用于重组的分段的 IP 层序号。

双击重组结果将在选项中显示重组数据包的解析结果:

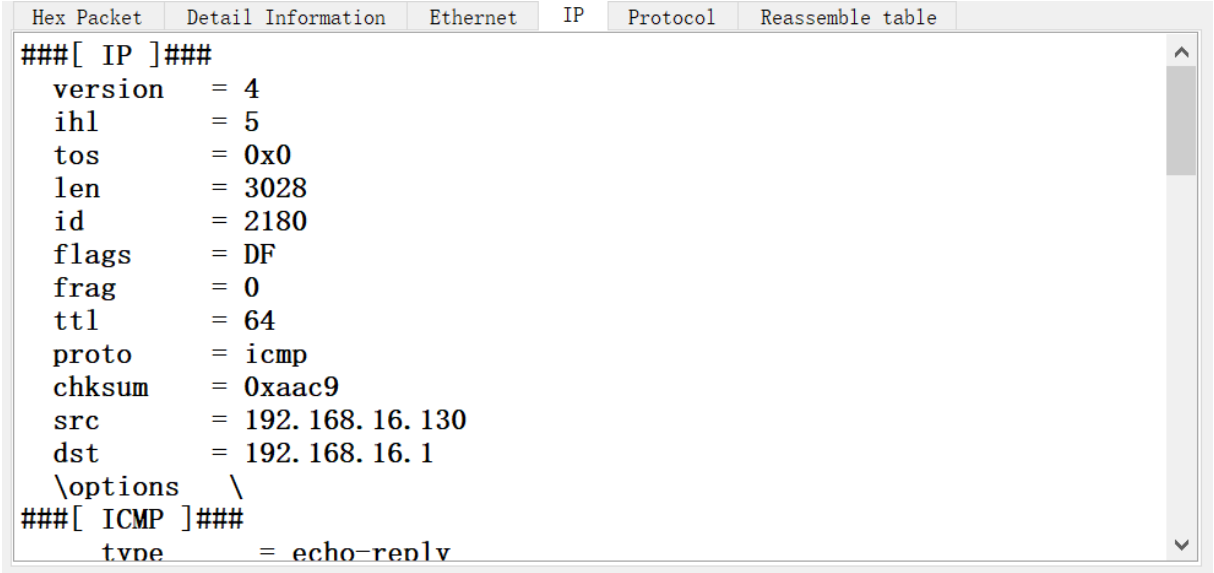


十六进制原格式:

数据链路层及以上的解析结果:

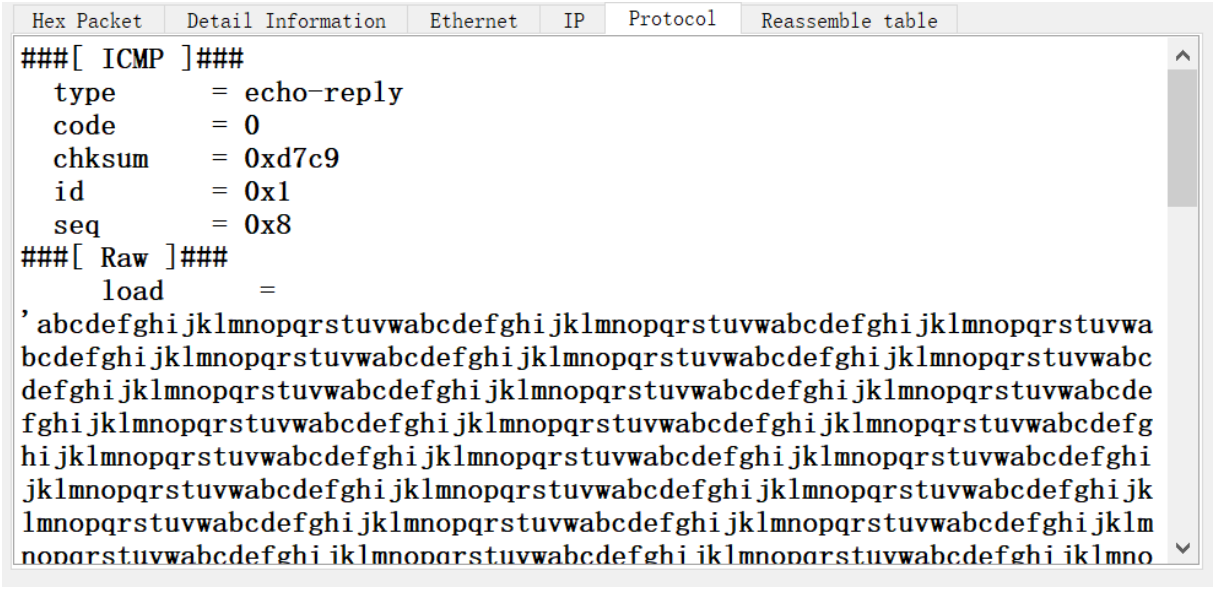


IP 层及以上的解析结果：



IP 层数据区的长度为 3028，与三个分段的长度减去重复首部后相加的结果相一致。

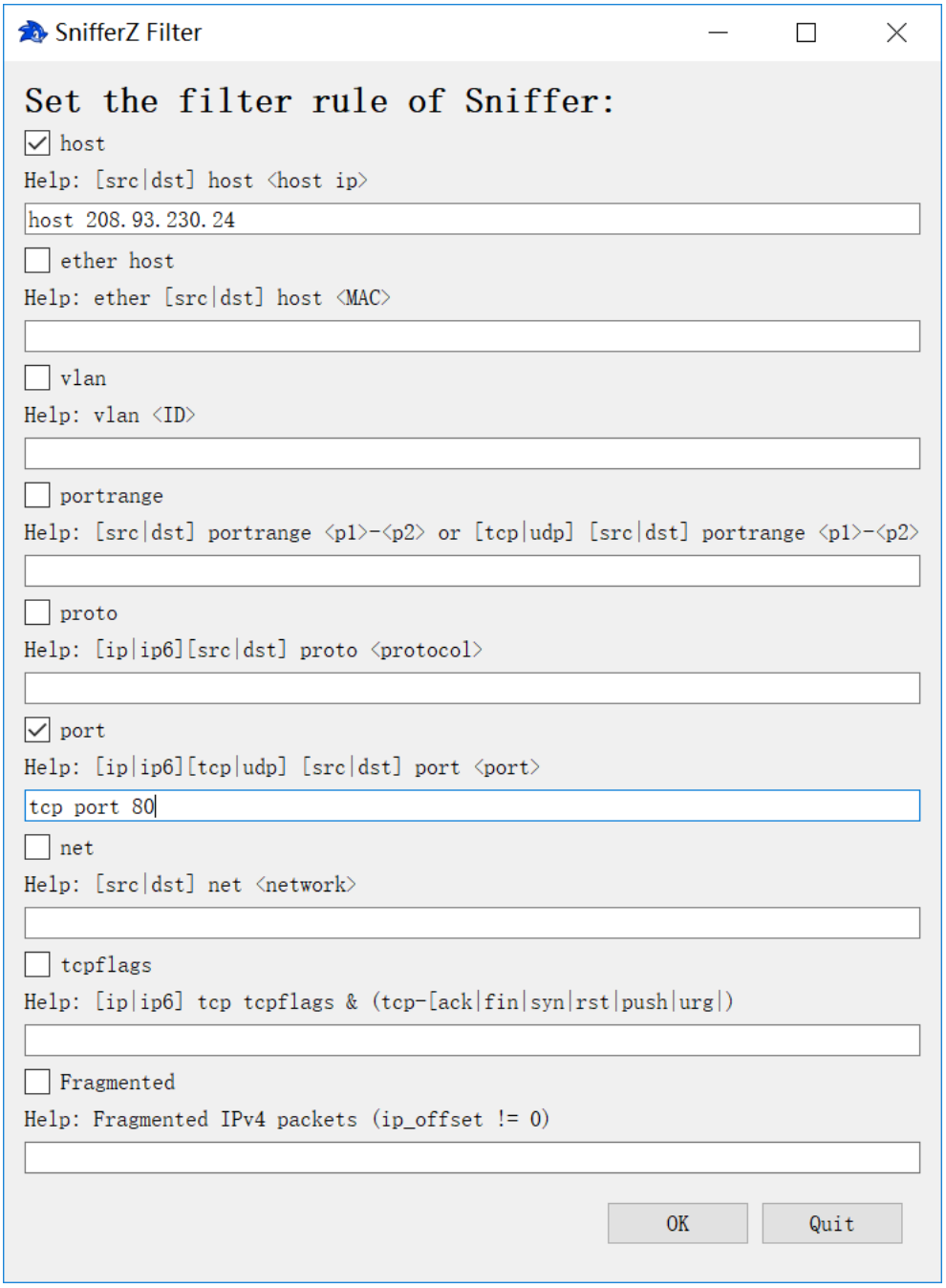
传输层及以上的解析结果：



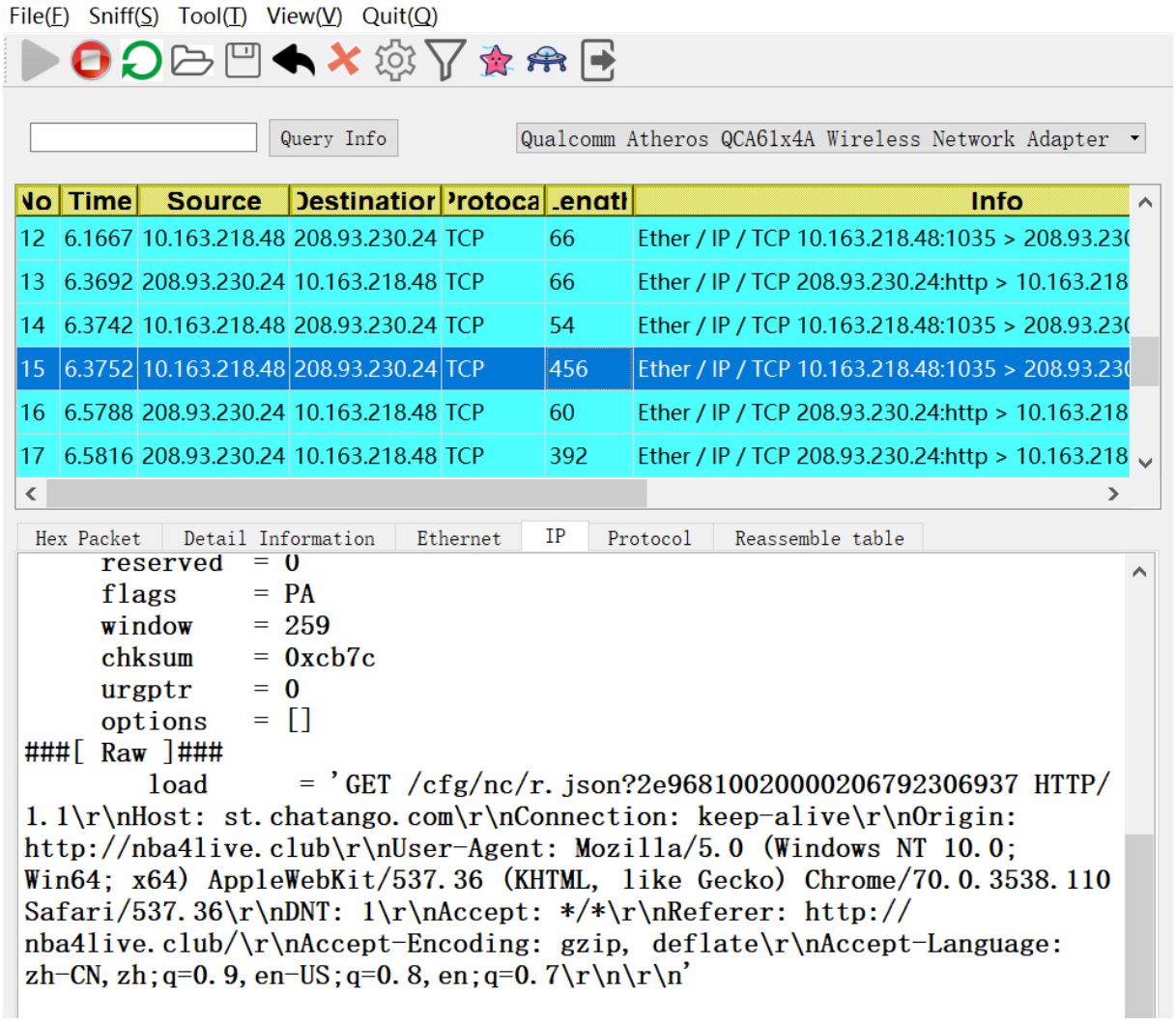
重组结果中 ICMP 的首部被完整保留，数据区的数据完整相加，并且数据区的解析结果完全具有可读性：

4.4 包过滤

点击 Set Filter 按钮，弹出 Filter 设置界面：



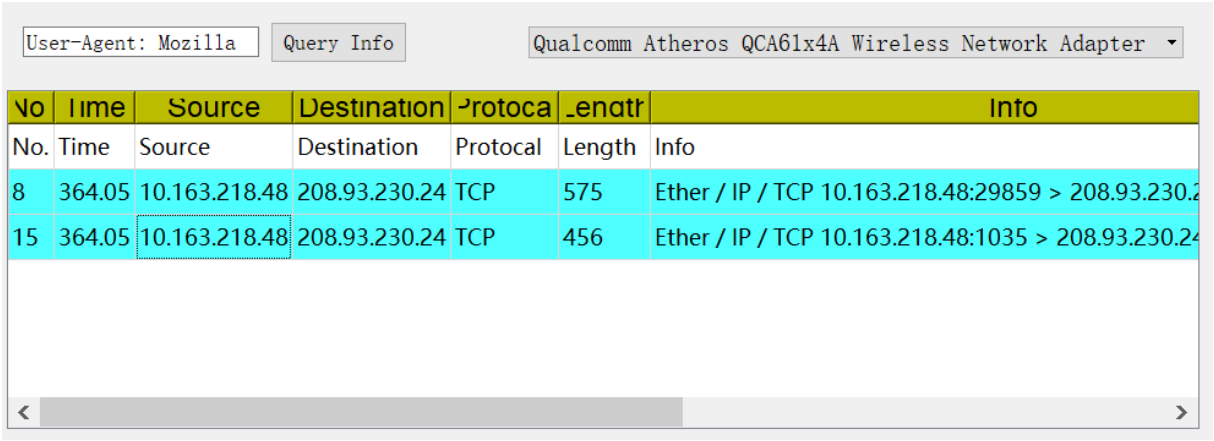
测试中，设置数据包的主机 IP（目的 IP 或源 IP）为 208.93.230.24 (www.nba4live.club)，侦听端口为 80。确认后开始侦听，浏览器访问网站 www.nba4live.club ：



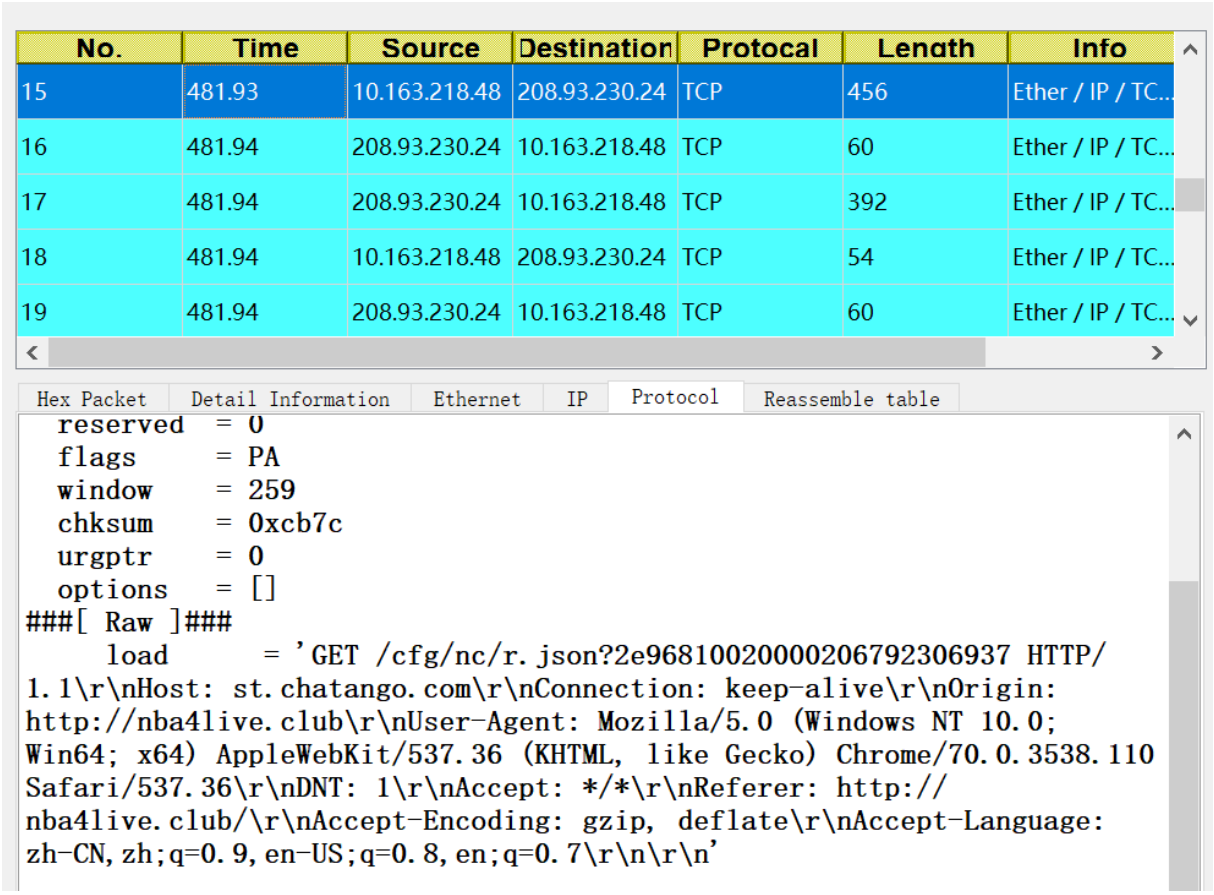
侦听的结果均为 208.93.230.24 与主机间的 http 通信。

4.5 数据包查询

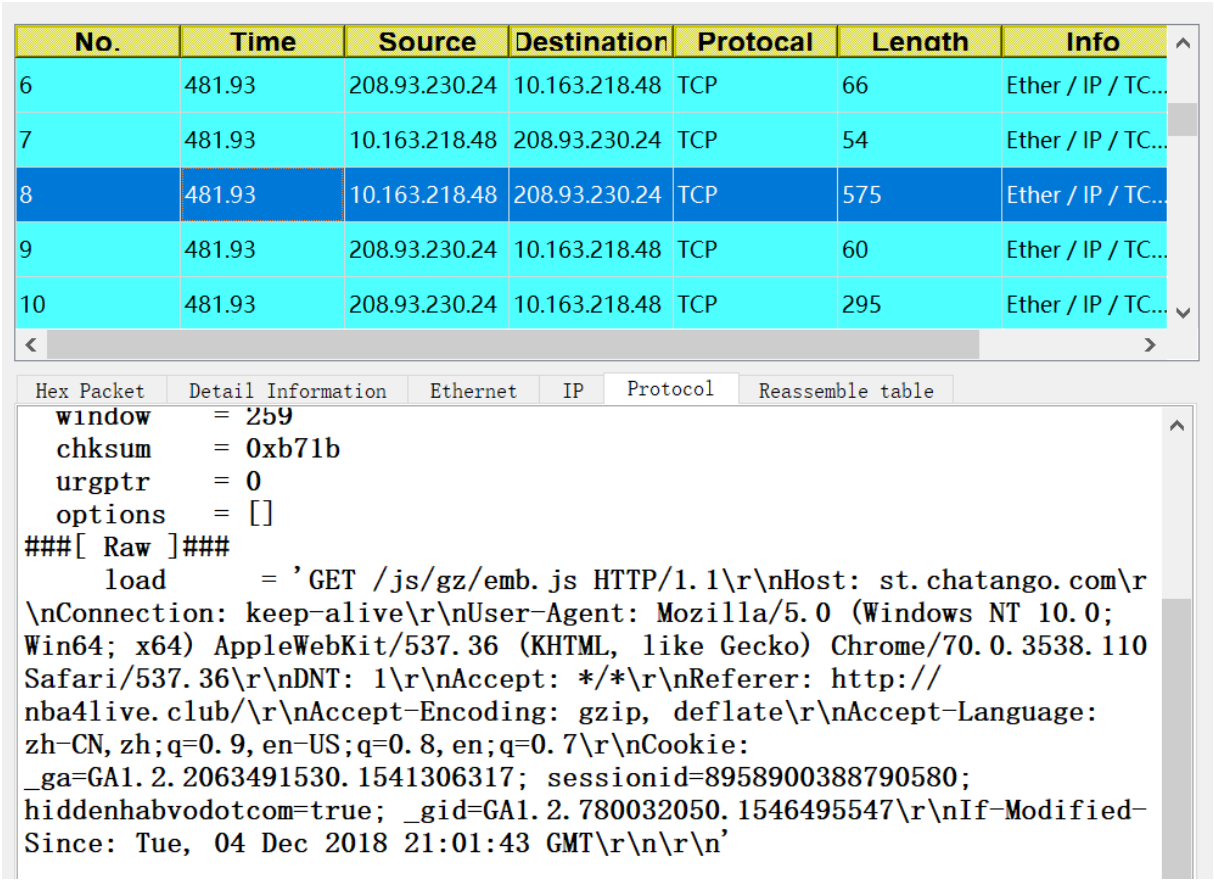
数据包查询在输入栏中输入要查询的字符串，程序将会在侦听历史中查询，头部解析结果或数据区 ASCII 解码包含查询字符串的数据包，并将查询结果显示在表格中：



查询结果显示第 8，15 条数据包的数据区包含 User-Agent: Mozilla 字符串，点击返回后查看对应内容：



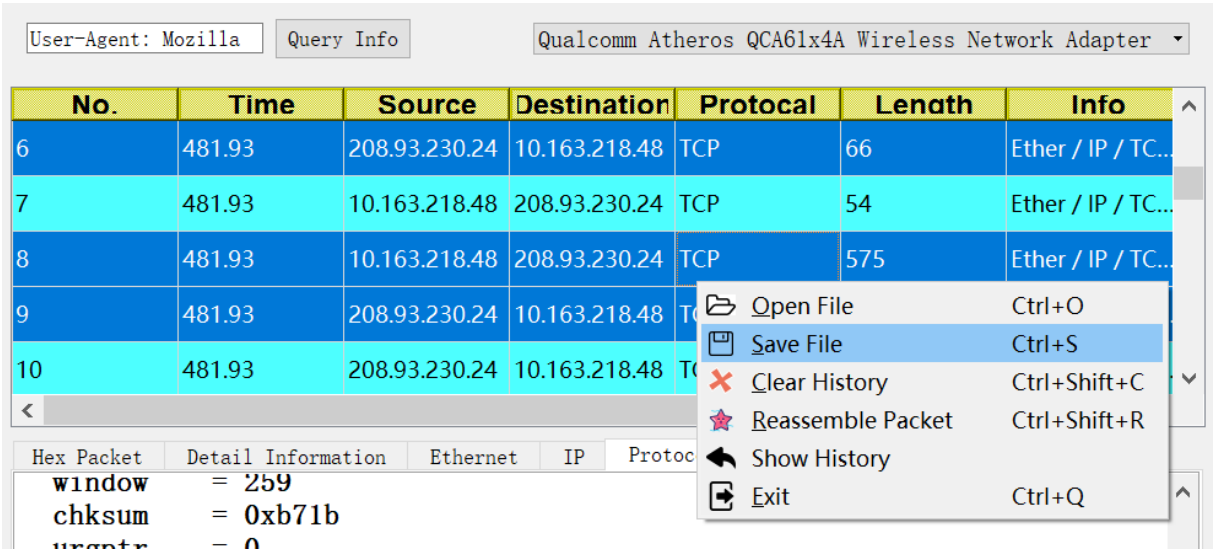
（需要注意的是，在查询结果的表格中双击条目是无效的，不会得出对应内容的解析）

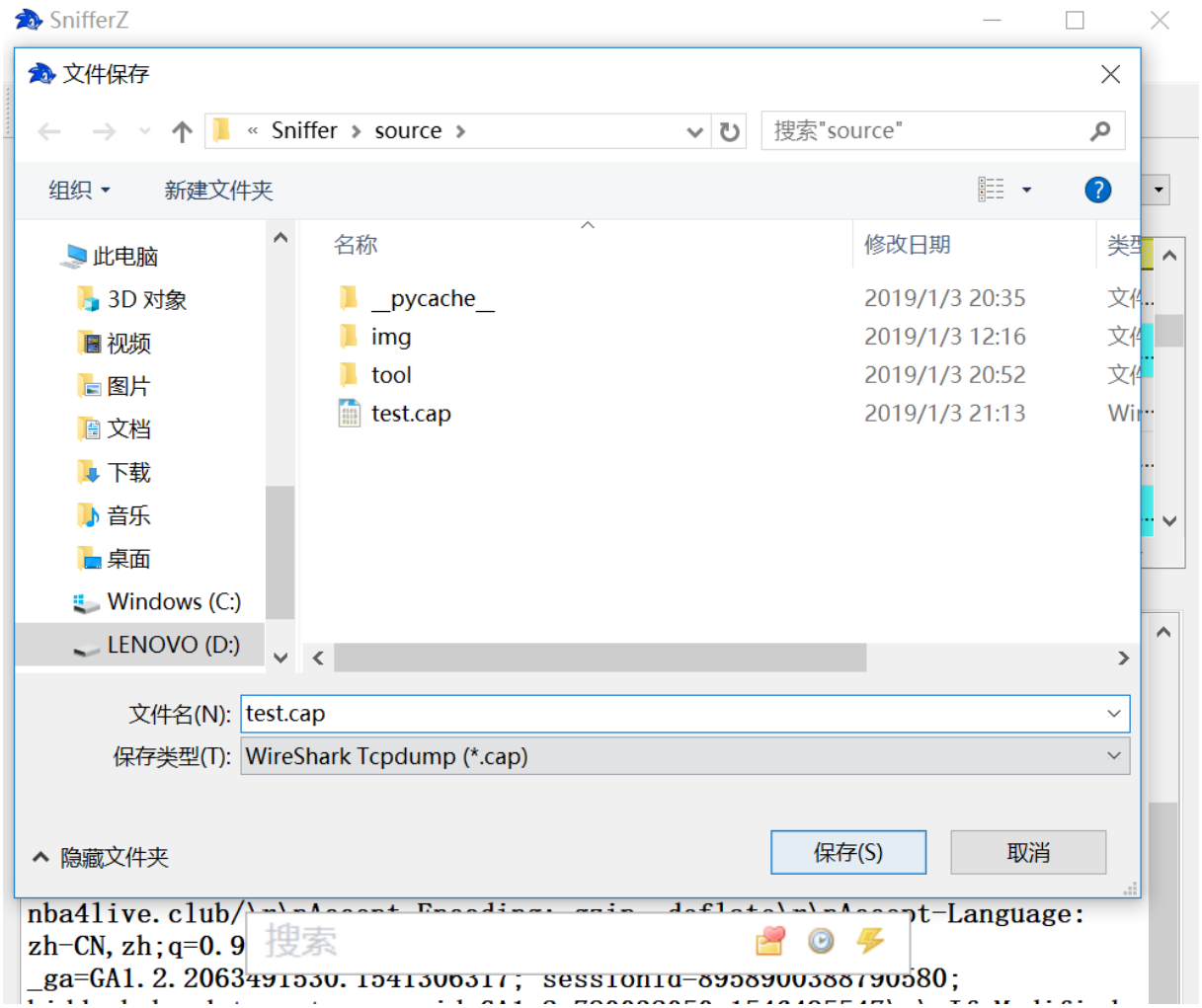


回到抓包历史中查看查询到的数据包，其数据区确实存在欲查找的字符串。

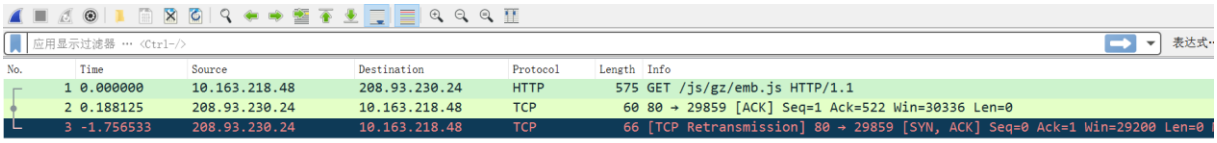
4.6 数据包保存

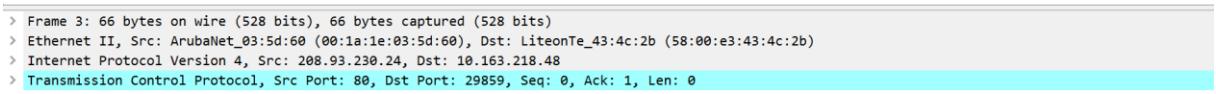
在表格中选中需要保存的数据包，点击保存，弹出文件保存窗口：

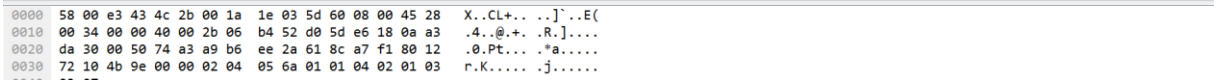




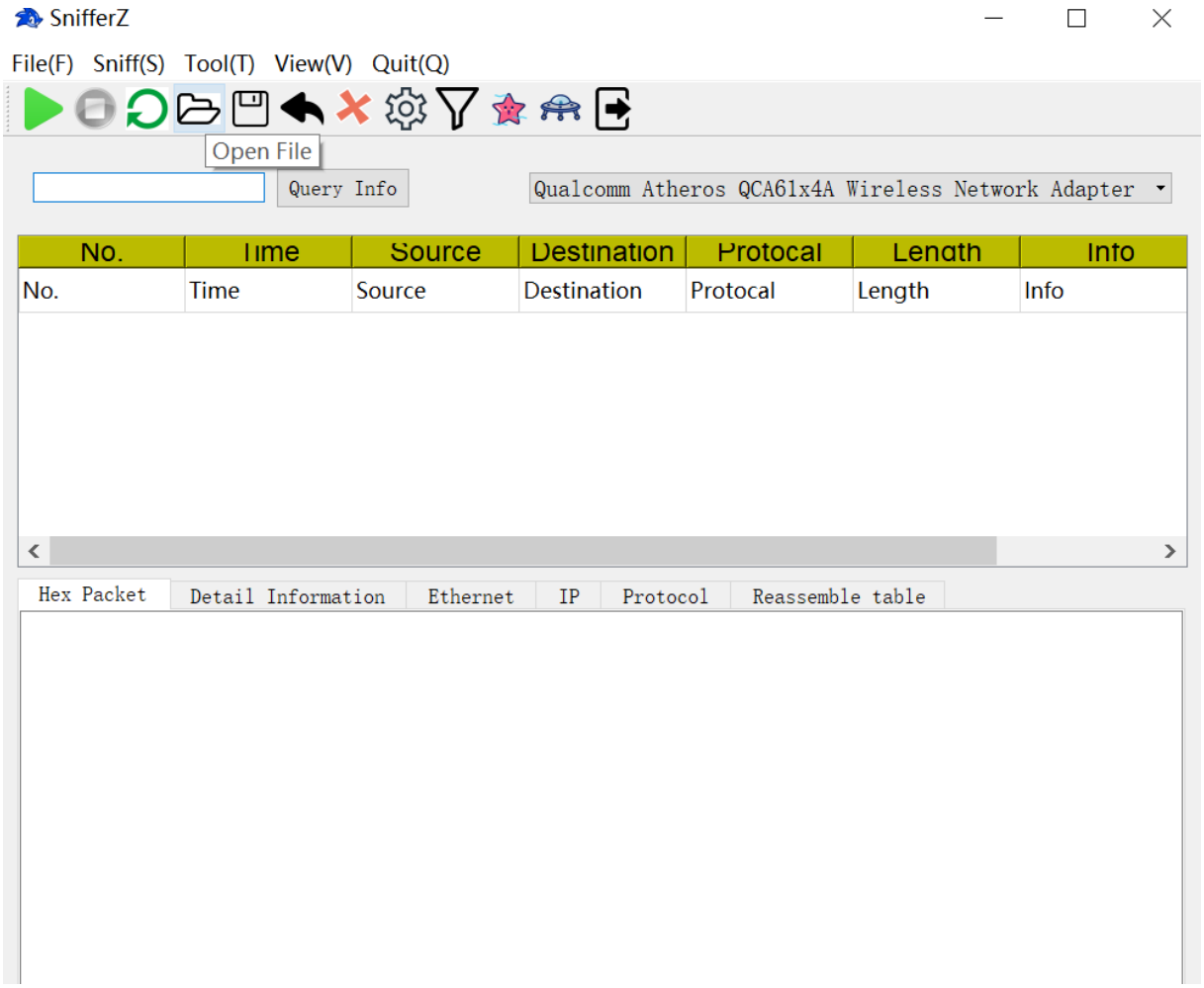
保存的文件可被 WireShark 正常打开：



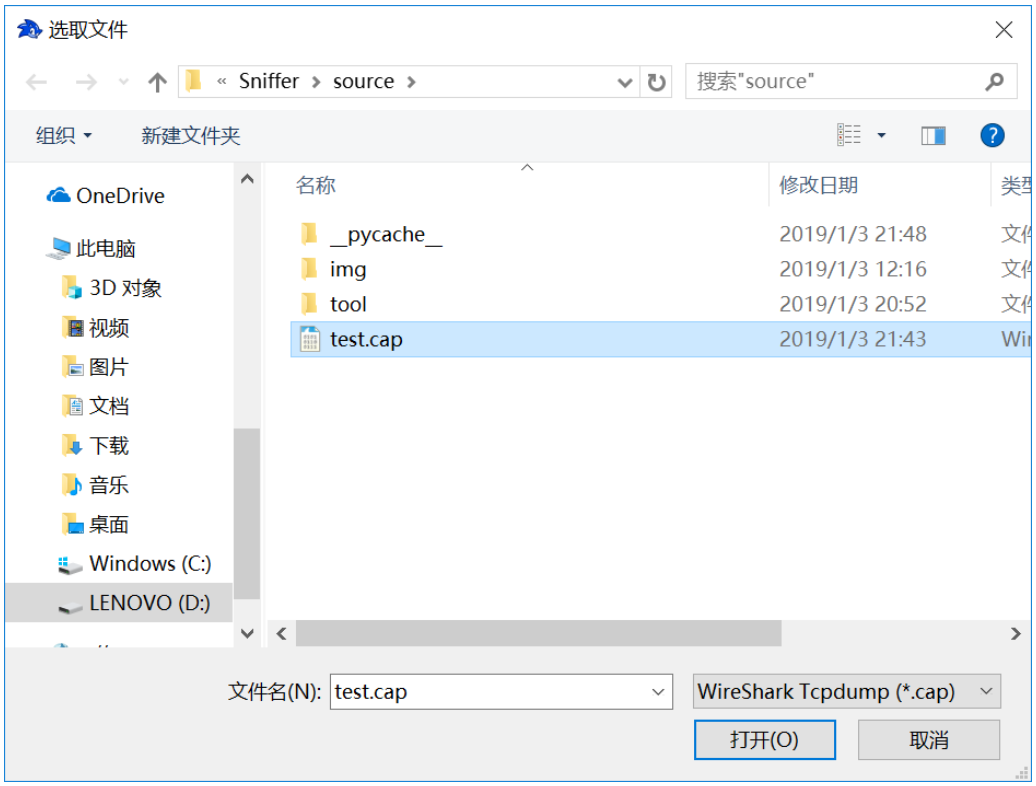




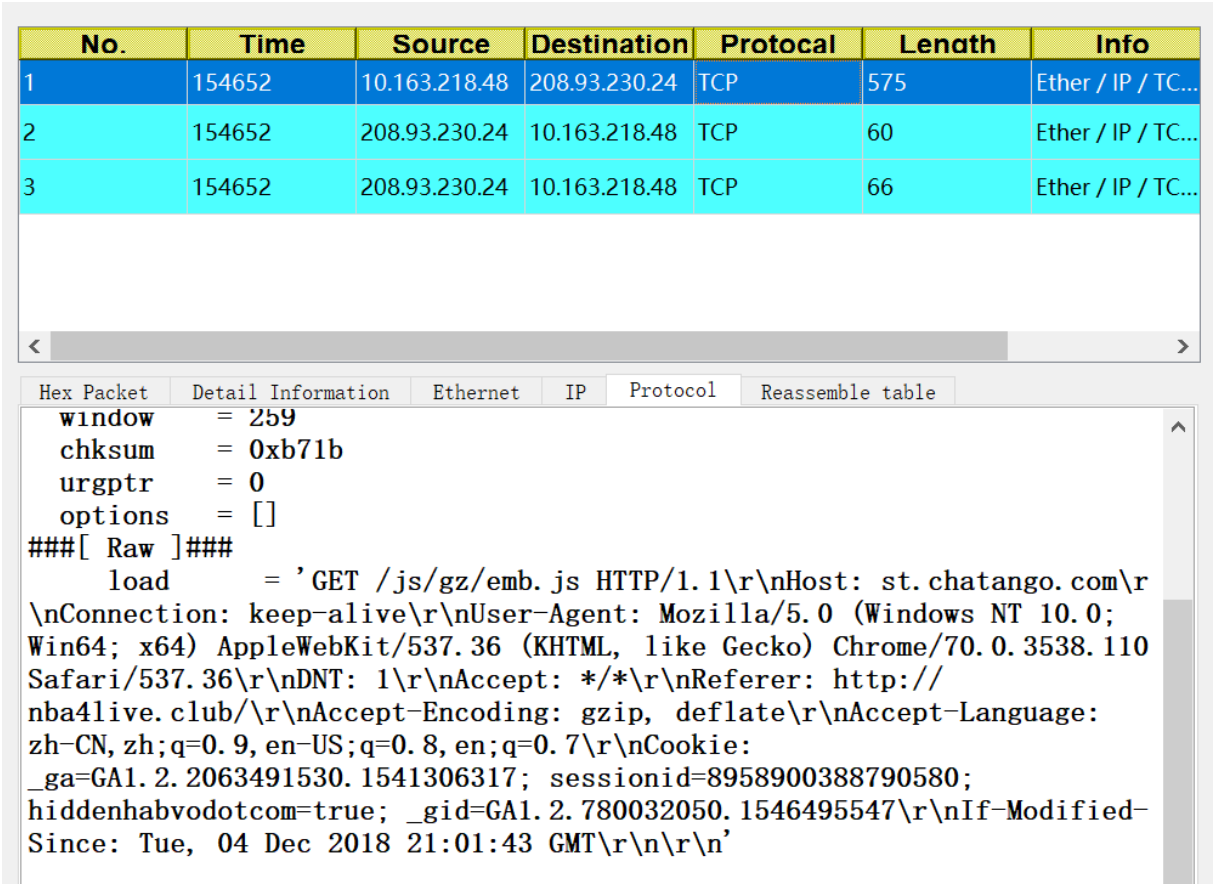
同样，SnifferZ 也可以打开保存的数据包：



点击 Open File 后弹出文段选取窗口，选择之前保存的 test.cap:



打开后显示和原先相同的数据包：



No.	Time	Source	Destination	Protocal	Length	Info
1	154652	10.163.218.48	208.93.230.24	TCP	575	Ether / IP / TC...
2	154652	208.93.230.24	10.163.218.48	TCP	60	Ether / IP / TC...
3	154652	208.93.230.24	10.163.218.48	TCP	66	Ether / IP / TC...

Hex Packet | Detail Information | Ethernet | IP | Protocol | Reassemble table

```
  window    = 259
  chksum    = 0xb71b
  urgptr    = 0
  options   = []
###[ Raw ]###
     load       = 'GET /js/gz/emb.js HTTP/1.1\r\nHost: st.chatango.com\r
\nConnection: keep-alive\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.110
Safari/537.36\r\nDNT: 1\r\nAccept: */*\r\nReferer: http://
nba4live.club/\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language:
zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7\r\nCookie:
_ga=GA1.2.2063491530.1541306317; sessionid=8958900388790580;
hiddenhabvodotcom=true; _gid=GA1.2.780032050.1546495547\r\nIf-Modified-
Since: Tue, 04 Dec 2018 21:01:43 GMT\r\n\r\n'
```

5. 遇到的问题与解决办法

5.1 UI 使用框架的更换

一开始的计划是使用 Kivy 实现 UI 界面，后台使用 Scapy 侦听，并计划利用 Kivy 的跨平台特性将实现的 Sniffer，结合 pyInstaller，python-for-andrroid 将实现的项目打包到多平台上，但无奈 Kv 抽象化的 UI 描述语言所属层次较高，很难从底层明白其运行原理，加之学习资料较少（中文入门资料更屈指可数），论坛和社区发展不够成熟，只在 YouTube 搜到了尚不完整的 Crash Cousre 入门视频，给新手的快速学习造成一定困难。因此，在 Kivy 学习到一半时，果断放弃使用其开发 UI，转向使用 PyQt5，由于其逻辑抽象和构建思维更接近 python，得以在较短的时间内入手并开发。

5.2 嗅探线程的控制问题

起初并没有意识到开启新线程前，要先 join 之前开启的线程，导致多次点击 Start 按钮后，表格中并列出现相同的数据包。查询资料后，才有了 3.1 的解决办法，主线程将自己的信号线埋入所有开启的子线程中，从而一根信号线控制所有开启子线程的 join。

5.3 Filter 的过滤规则问题

一开始并不清楚 Scapy 库中的 Filter 使用何种规则，导致 filter 的设计在开始时一筹莫展，从 StackOverflow 的回答评论中，得知 filter 的设置使用 BPF 语法，（参见：https://stackoverflow.com/questions/37453283/filter-options-for-sniff-function-in-scapy）于是，在 filter 控件中使用了其支持的全部功能，充分使用了 sniff 函数的过滤功能。

5.4 QLabel 无法展示 PDF

包结构分析图的构造结构保存在 PDF 中，而 QLable 中无法显示 PDF 文件，查找了诸多解决办法，如：使用 poppler 库，将 PDF 矢量图转化为 png 图片，再去显示，但 PyPI 上的 poppler 已经停止维护，直接 pip 安装的结果不能使用，从网站上搜寻老版本或编译后的 wheel 文件都不能成功安装。而尝试使用 PyQt5 中的浏览器组件去显示 PDF 也会报出错误，又会有新的坑去排。尝试各种办法后，突发奇想，直接使用 window 平台下的命令行工具，直接通过 os 模块，在 shell 中完成 PDF 到 PNG 的转化，搜寻了各种工具后，使用了 Mupdf 中的 Mutool 工具实现此需求。直接在 python 中使用：

```
1.  state = os.system(r'.\tool\mutool.exe convert  -
    o .\tool\tmp.png .\tool\packet_analysis.pdf')
```

完成转化任务。最终可以使构造的 PDF 分析图展示在 UI 主界面中。此项功能目前只能在 windows 平台下使用，但相信找到 Linux 下的 PDF 转化工具（容易的任务），此项功能也可拓展到 Linux 平台下。

5.5 分段的构造

在分段重组任务中，第一个困难便是如何构造分段用于测试？查看了很多嗅探结果，发现，分段现象是少之又少的情况，流经本机上的流量一般很少出现分组，这便造成了一个困难。询问同学后得知，可以使用 ping 命令主动设置数据报长度，强制使其分组来达到目的，如下为 windows 主机和 ubuntu 虚拟机的 ping 命令：

```
1. ping -l 3000 -S 192.168.16.1 192.168.16.130
2. ping -s 3000 192.168.16.1
```

6 体会与建议

本次课程设计中，我独立完成了项目架构，项目实现，项目测试，报告撰写，Demo 制作等过程。尽管项目实现过程中难免出现波折，中途更换了 UI 实现框架，但整个过程中，遇到问题，积极的去搜寻答案，冷静的解决问题，从另一个方面提升了我主动探索的能力，文献查阅的能力，以及统筹安排的能力。

在前期的框架选择中，我对 Kivy 的学习难度判断失准，导致花费了较多时间去学习，但收获和进度有限，因此中途果断放弃 Kivy，转而使用更为熟悉的 PyQt5 去开发 UI，PyQt5 使用的感觉也让人眼前一亮，使用其开发 UI 的思维和熟悉的 python 思维相当一致，因此在较短的时间内完成了学习，并在理解了其信号和槽机制后，开发速度有了极大的提升。

后期的调试和测试中，也遇到了许多未知的问题，好在对 python 较为熟悉，对查询解决办法的思路也比较熟悉，在这些问题上并没有停留太多时间。之后，我进一步完善了程序的异常处理，使项目更具鲁棒性，但由于时间有限，并接近考试周，有些功能还尚待拓展，如：Linux 环境下报结构分析图的生成，这些问题，我会在后续的工作中继续拓展，完善后将项目开源在 GitHub 上。

最后，十分感谢老师在课堂上详细的讲解，老师在课堂上讲解的内容让我明白如何去完成包解析，分段重组等功能。也十分感谢助教老师在大作业课上耐心的讲解大作业的要求，在那节大作业课上我便开始构思如何去实现嗅探器的各项功能，而助教老师的讲解过程无疑地帮助我构建起了项目实现的初步框架。