项目文档：网络嗅探器

1. 项目概述

本项目实现了网络嗅探器（JKSniffer）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用户界面。目前支持 ARP、IP、IPv6、ICMP、UDP、TCP 协议。项目基于 Python 编程语言，在项目的前两轮迭代中，分别使用原生 socket 库和 scapy 第三方库实现了两个版本的包解析、IP 重组等功能。除此之外，本项目还支持包文件保存（pcap）、读取、数据包可视化存储（pdf）、包过滤器、简易发包器(支持 IP、TCP、UDP 发包)、ARP 欺骗等功能。

项目 GUI 采用 PyQt4 实现，项目尽力提供用户友好的界面，美观的构图，简易的操作。当前 GUI 支持视图框选（即选择快捷图标栏显示的内容）、背景着色等功能。

项目开发周期为一个月，共分四轮迭代。使用 git 和 github 完成版本控制管理。第一轮迭代为 socket 包解析和简易 GUI 实现，第二轮迭代为 scapy 包解析、分片重组和 GUI 的完善优化，第三轮迭代为 JKSniffer 高级功能实现，第四轮迭代为 GUI 优化和项目文档撰写。

注：该项目已开源: https://github.com/wangjksjtu/JKSniffer

1.1 环境配置

表 1-1　第三方依赖环境配置

第三方库	配置参数值
Scapy	Scapy 2.3.3 (Python 2.7) (http://scapy.readthedocs.io)
Numpy	Numpy (http://www.numpy.org/)
GnuPlot	Gnuplot 5.2.2 (http://www.gnuplot.info/)
PyX	PyX — Python graphics package (0.14.1) (http://pyx.sourceforge.net/)
PyQt4	PyQt4 v4.11 (http://pyqt.sourceforge.net/Docs/PyQt4/)

表 1-2　测试环境配置

项目	配置参数值
操作系统	Ubuntu 16.04
CPU	Intel(R) Core(TM) i7-3630QM CPU @2.40GHz
内存	8.00GB DDR_3
硬盘	500G 7200r
IP 地址	192.168.98.131
网卡接口	ens33

1.2 程序文件列表

表 1-3 项目关键目录文件功能说明

docs/	项目文档、帮助手册、README.md 等说明文档
imgs/	项目展示图片,效果截图等
src/	项目源代码目录
test/	单元测试模块代码
src/icon/	JKSniffer logo 图标
src/imgs/	JKSniffer GUI 内用到的图片集合
src/utils/	项目辅助函数实现
src/scapy_http/	http 协议扩展内容支持（未在 GUI 版本中支持）
main.py	项目可执行文件入口，执行命令（sudo python main.py）
mainWindow.py	主窗口 GUI 框架实现（mainWindow.ui 为 pyqt4 辅助生成文件）
packet.py	各协议包解析类（支持 ARP、IP、IPv6、ICMP、UDP、TCP 协议）

```
wangjk@asus-wjk:~/programs/JKSniffer$ tree -L 3 -P *.py
.
├── docs
├── imgs
├── src
│   ├── arp_spoofing.py
│   ├── Filter.py
│   ├── icon
│   ├── imgs
│   ├── Interface.py
│   ├── JKInterface.py
│   ├── main.py
│   ├── mainWindow.py
│   ├── packet.py
│   ├── pcap.py
│   ├── scapy_http
│   │   ├── http.py
│   │   └── __init__.py
│   ├── sendpkt.py
│   ├── Settings.py
│   ├── Spoofing.py
│   └── utils
│       ├── hexdump.py
│       └── __init__.py
└── test
    ├── test2.py
    ├── test3.py
    └── test.py

8 directories, 18 files
```

图 1-1 JKSniffer 工程文件结构示意

2. 数据结构

如图 2-1，JKSnifferGUI 为程序主要函数，包含打开.pcap 文件（open）、保存 PDF 格式（convert PDF）、开始抓包（start）、数据包内容显示等功能实现，主页面 GUI 模块为 Ui_MainWindow。JKSniffer 调用 FilterGUI 实现包过滤（Filter）功能，调用 JKInterface 实现侦听网卡选择功能（Interface），调用 SendpktGUI 实现简单发包功能（Packet）,调用 ARP_Spoofing 实现 ARP 欺骗功能（ARP Spoofing），调用 SendpktGUI 实现简单发包功能（Send）。侦听网卡选择功能（Interface）GUI 模块为 InterfaceGUI，ARP 欺骗功能（ARP Spoofing）GUI 模块为 SpoofingGUI。

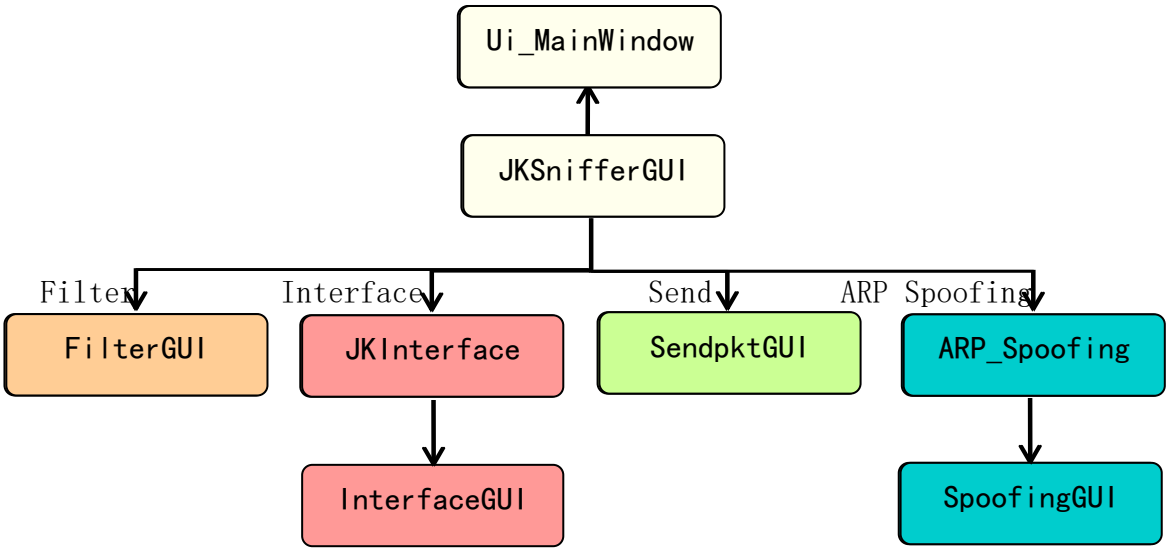


图 2-1　主要函数调用关系示意图

如图 2-2，程序中对于嗅探到的包，用数组 self.packets 保存。解析之后的包，其类型为定义的 class Packet 类型，用数组 self.pkts 保存。

解析后的包有以下几个属性，在主页面进行显示：本轮抓包抓到的该数据包次序（self.id）、本轮抓包抓到的该数据包相对时间（self.time）、本轮抓包抓到的该数据包源地址（self.src）、本轮抓包抓到的该数据包目的地址（self.dst）、本轮抓包抓到的该数据包内容信息（self.info）、本轮抓包抓到的未解析的数据包（self.packet）。

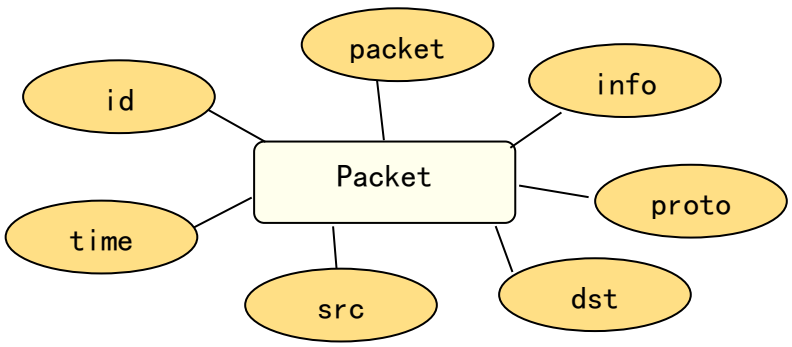


图 2-2　解析后的数据包属性示意图

3. 算法实现

3.1 数据包抓取

3.1.1 socket 编程实现

```
s=socket.socket(socket.AF_PACKET,socket.SOCK_RAW,socket.ntohs(0x0003))
```

参数一：地址簇；

参数二：类型，socket.SOCK_RAW 原始套接字，普通的套接字无法处理 ICMP、IGMP 等网络报文，而 SOCK_RAW 可以；其次，SOCK_RAW 也可以处理特殊的 IPv4 报文；此外，利用原始套接字，可以通过 IP_HDRINCL 套接字选项由用户构造 IP 头。SOCK_RAM 用来提供对原始协议的低级访问，在需要执行某些特殊操作时使用，如发送 ICMP 报文。SOCK_RAM 通常仅限于高级用户或管理员运行的程序使用；

参数三：协议，socket.ntohs()把 32 位正整数从网络序转换成主机字节序。

3.1.2 scapy 编程实现

```
sniff(iface=self.interface, prn=self.pkt_callback, \
    filter=self.filter, stop_filter=self.stop_filter)
```

参数一：iface 用来指定要在哪个网络接口上进行抓包；

参数二：prn 指定回调函数 self.pkt_callback，每当一个符合 filter 的报文被探测到时，就会执行回调函数;

参数三：包过滤规则 filter 的使用 Berkeley Packet Filter (BPF)语法，

参数四：抓取数据包停止标志位；

3.2 数据包解析

3.2.1 Socket 编程解析数据包

（1）以太网报头，如图 3-1，

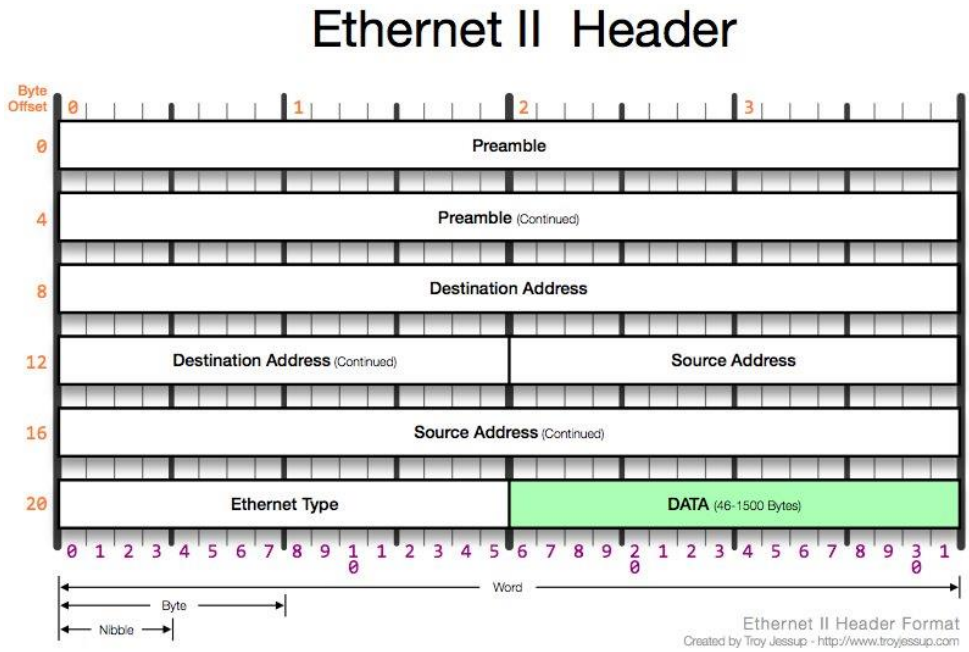


图 3-1　以太网头部解析示意图

Ethernet 头部，总长度 14 字节；目标 man 地址 6 字节，源 mac 地址 6 字节，以太网类型 2 字节，解析部分代码如下。

```
7     # Ethernet Header
8     def eth_header(self, data):
9         storeobj=data
10        storeobj=struct.unpack("!6s6sH",storeobj)
11        destination_mac=binascii.hexlify(storeobj[0])
12        source_mac=binascii.hexlify(storeobj[1])
13        eth_protocol=storeobj[2]
14        data={"Destination Mac":destination_mac,
15        "Source Mac":source_mac,
16        "Protocol":eth_protocol}
17        return data
```

（2）ARP 报头，如图 3-2，

Internet Protocol (IPv4) over Ethernet ARP packet		
octet offset	0	1
0	Hardware type (HTYPE)	
2	Protocol type (PTYPE)	
4	Hardware address length (HLEN)	Protocol address length (PLEN)
6	Operation (OPER)	
8	Sender hardware address (SHA) (first 2 bytes)	
10	(next 2 bytes)	
12	(last 2 bytes)	
14	Sender protocol address (SPA) (first 2 bytes)	
16	(last 2 bytes)	
18	Target hardware address (THA) (first 2 bytes)	
20	(next 2 bytes)	
22	(last 2 bytes)	
24	Target protocol address (TPA) (first 2 bytes)	
26	(last 2 bytes)	

图 3-2　ARP 报头解析示意图

硬件类型 2 字节，上层协议类型 2 字节，MAC 地址长度 1 字节 ，IP 地址长度 1 字节，操作码 2 字节，发送方 MAC 地址 6 字节，发送方 IP 地址 4 字节，接收方 MAC 地址 6 字节，接收方 IP 地址 4 字节，

从 ARP 报头中获取源主机 MAC 地址、IP 地址，目标主机 MAC 地址、IP 地址，解析部分代码如下，

```
def get_src_mac():
    mac_dec = hex(getnode())[2:-1]
    while (len(mac_dec) != 12):
        mac_dec = "0" + mac_dec
    return unhexlify(mac_dec)

def get_src_ip_addr():
    src_ip_addr = ''
    ip_src_dec = argv[3].split(".")
    for i in range(len(ip_src_dec)):
        src_ip_addr += chr(int(ip_src_dec[i]))
    return src_ip_addr

def get_dst_mac_addr():
    p = subprocess.Popen(["arping", argv[2], "-c", "1", "-i", argv[1]],
                         shell=False, stdout=subprocess.PIPE)
    sleep(2)
    remote_mac = search('(([0-9a-f]{2}:){5}[0-9a-f]{2})', p.communicate()[0])
    return unhexlify(remote_mac.group(0).replace(':', ''))
```

（3）IP 报头，如图 3-3（a）（b），

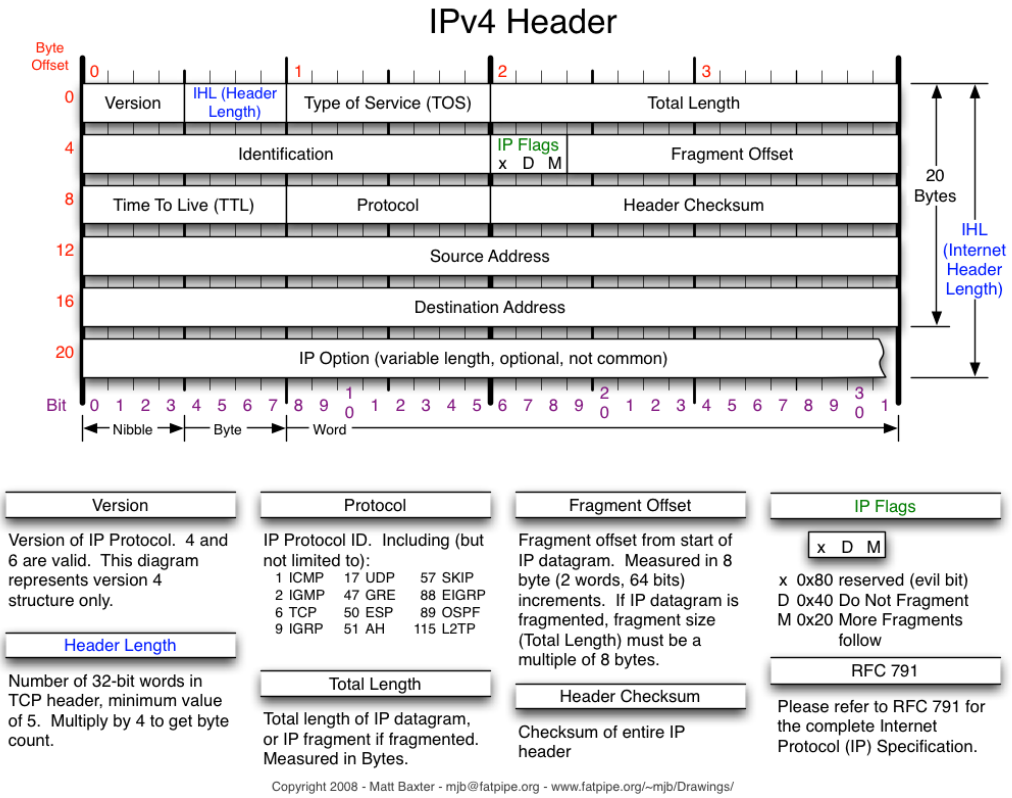


图 3-3（a） IPv4 报头解析示意图

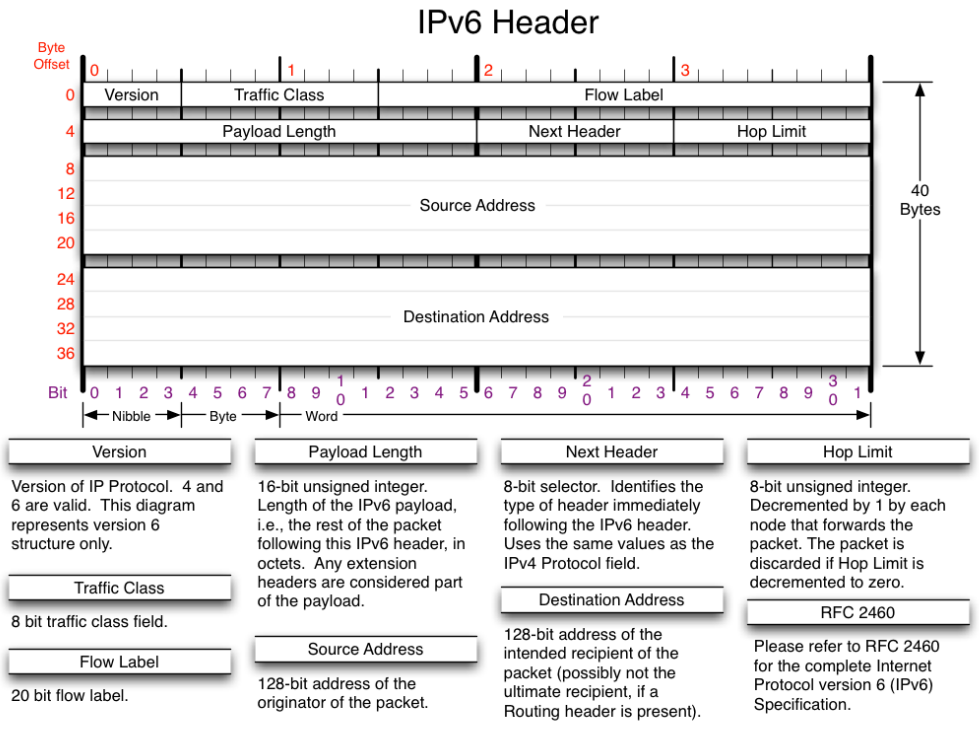


图 3-3（b） IPv6 报头解析示意图

IPv4 报头除可选部分，总长 20 字节。从 IPv4 报头长度 ihl 开始解析，计数器 ttl，下层协议类型 protocol，源主机 IP 地址 s_addr，目标主机 IP 地址 d_addr，解析部分代码如下，

```
43      # IP Header Extraction
44      def ip_header(self, data):
45          storeobj=struct.unpack("!BBHHHBBH4s4s", data)
46          _version=storeobj[0]
47          _tos=storeobj[1]
48          _total_length =storeobj[2]
49          _identification =storeobj[3]
50          _fragment_Offset =storeobj[4]
51          _ttl =storeobj[5]
52          _protocol =storeobj[6]
53          _header_checksum =storeobj[7]
54          _source_address =socket.inet_ntoa(storeobj[8])
55          _destination_address =socket.inet_ntoa(storeobj[9])
56
57          data={'Version':_version,
58          "Tos":_tos,
59          "Total Length":_total_length,
60          "Identification":_identification,
61          "Fragment":_fragment_Offset,
62          "TTL":_ttl,
63          "Protocol":_protocol,
64          "Header CheckSum":_header_checksum,
65          "Source Address":_source_address,
66          "Destination Address":_destination_address}
67          return data
```

（4）TCP 报头，如图 3-4，

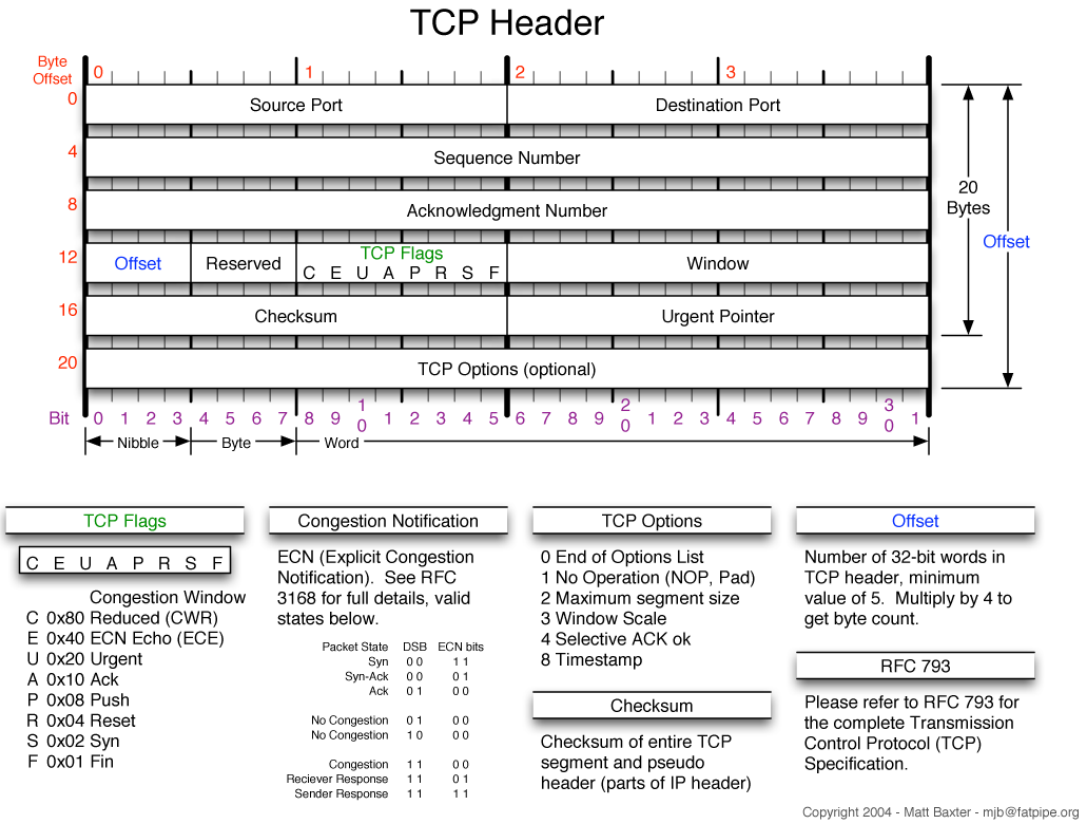


图 3-4　TCP 报头解析示意图

IP 报头中 protocol 协议号为 6，TCP 协议。

source_port 源端口 2 字节，dest_port 目的端口 2 字节，sequence 发送数据包中第一个字节序列号 4 字节，acknowledgement 确认序列号 4 字节，doff_reserved 数据偏移四位，TCP 头部长度除以 4 ，解析部分代码如下，

```
54          #TCP protocol
55          if protocol == 6 :
56              t = iph_length + eth_length
57              tcp_header = packet[t:t+20]
58
59              #now unpack them :)
60              tcph = unpack('!HHLLBBHHH' , tcp_header)
61
62              source_port = tcph[0]
63              dest_port = tcph[1]
64              sequence = tcph[2]
65              acknowledgement = tcph[3]
66              doff_reserved = tcph[4]
67              tcph_length = doff_reserved >> 4
68
69              print 'Source Port : ' + str(source_port) + ' Dest Port : ' + str(dest_port) + \
70                    ' Sequence Number : ' + str(sequence) + ' Acknowledgement : ' + str(acknowledgement) \
71                   + ' TCP header length : ' + str(tcph_length)
72
73              h_size = eth_length + iph_length + tcph_length * 4
74              data_size = len(packet) - h_size
75
76              #get data from the packet
77              data = packet[h_size:]
78
79              print 'Data : ' + data
```

（5）UDP 报头，如图 3-5，

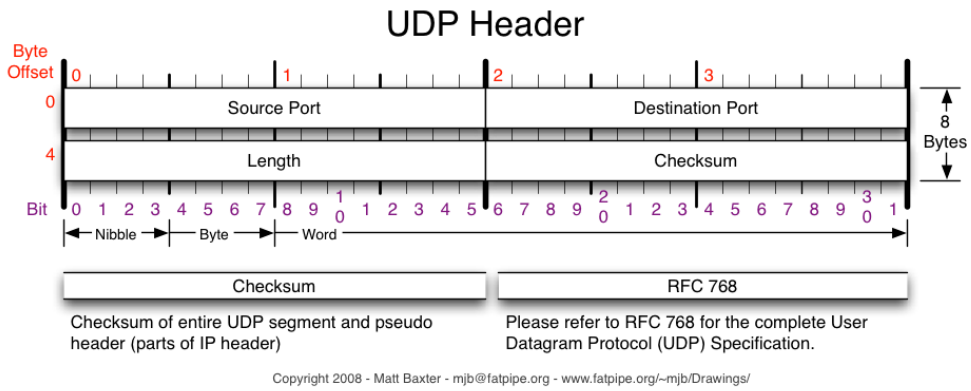


图 3-5　UDP 报头解析示意图

IP 报头中 protocol 协议号为 17，UDP 协议。

每一个数据包前八字节为头部信息，source_port 源端口号，dest_port 目标端口号，length 数据报长度，checksum 校验值。解析部分代码如下，

```
105         elif protocol == 17 :
106             u = iph_length + eth_length
107             udph_length = 8
108             udp_header = packet[u:u+8]
109 
110             #now unpack them :)
111             udph = unpack('!HHHH' , udp_header)
112 
113             source_port = udph[0]
114             dest_port = udph[1]
115             length = udph[2]
116             checksum = udph[3]
117 
118             print 'Source Port : ' + str(source_port) + ' Dest Port : ' + str(dest_port) + \
119                   ' Length : ' + str(length) + ' Checksum : ' + str(checksum)
120 
121             h_size = eth_length + iph_length + udph_length
122             data_size = len(packet) - h_size
123 
124             #get data from the packet
125             data = packet[h_size:]
126             print 'Data : ' + data
```

（6）ICMP 报头，如图 3-6，

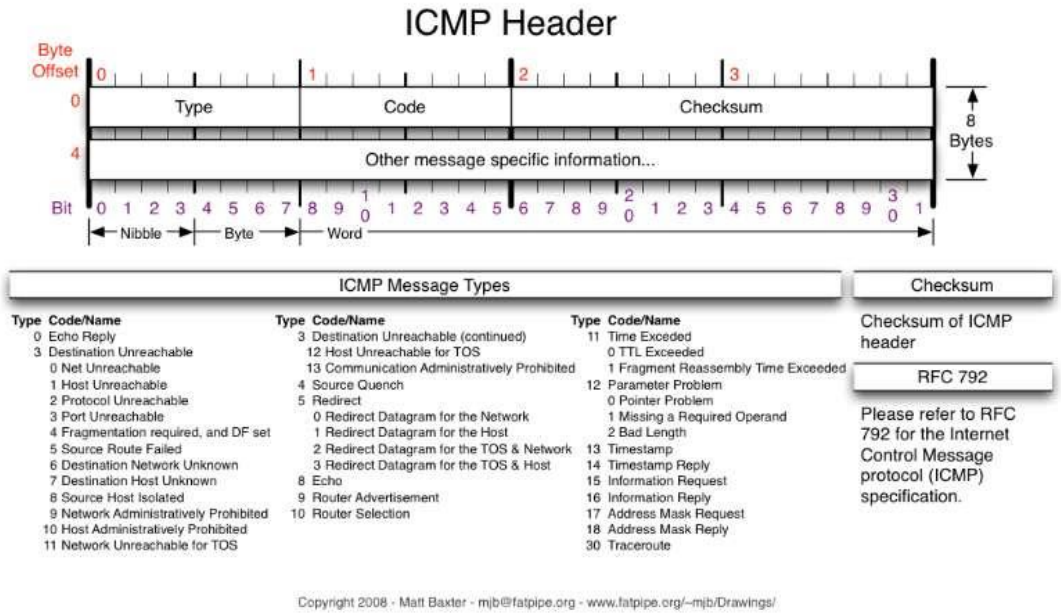


图 3-6　ICMP 报头解析示意图

IP 报头中 protocol 协议号为 17，UDP 协议。

ICMP 报头 8 字节长的，前 4 字节固定的格式，后 4 个字节根据 ICMP 包的类型而取不同的值。icmp_type 类型 1 字节，code 代码 1 字节，checksum 校验和 2 字节，解析部分代码如下，

```
81          #ICMP Packets
82          elif protocol == 1 :
83              u = iph_length + eth_length
84              icmph_length = 4
85              icmp_header = packet[u:u+4]
86
87              #now unpack them :)
88              icmph = unpack('!BBH' , icmp_header)
89
90              icmp_type = icmph[0]
91              code = icmph[1]
92              checksum = icmph[2]
93
94              print 'Type : ' + str(icmp_type) + ' Code : ' + str(code) + ' Checksum : ' + str(checksum)
95
96              h_size = eth_length + iph_length + icmph_length
97              data_size = len(packet) - h_size
98
99              #get data from the packet
100             data = packet[h_size:]
101
102             print 'Data : ' + data
```

3.2.2 Scapy 解析数据包

Scapy 库对于 sniff（）函数抓取到的数据可以执行 packet.getlayer()操作，直接获取各层协议的头部信息，具体实现代码如下，

```
237     def pkt_callback(self, packet):
238         #pkt.show()
239         if self.counter == 0:
240             self.init_time = packet.time
241         self.counter += 1
242         pkt = Packet(id = self.counter)
243         if int(packet.getlayer(Ether).type) == 34525:
244             pkt.proto = 'IPv6'
245             pkt.time = packet.time - self.init_time
246             pkt.src = str(packet.getlayer(IPv6).src)
247             pkt.dst = str(packet.getlayer(IPv6).dst)
248             pkt.info = str(packet.summary())
249             #self.packet_table.row_append(src, dst, proto, info)
250             self.packets.append(packet)
251             self.pkts.append(pkt)
252         elif int(packet.getlayer(Ether).type) == 2048:
253             if int(packet.getlayer(IP).proto) == 6:
254                 pkt.proto = 'TCP'
255             elif int(packet.getlayer(IP).proto) == 17:
256                 pkt.proto = 'UDP'
257             elif int(packet.getlayer(IP).proto) == 1:
258                 pkt.proto = 'ICMP'
259             pkt.time = packet.time - self.init_time
260             pkt.src = str(packet.getlayer(IP).src)
261             pkt.dst = str(packet.getlayer(IP).dst)
262             pkt.info = str(packet.summary())
263             #self.packet_table.row_append(src, dst, proto, info)
264
265
266             self.packets.append(packet)
267             self.pkts.append(pkt)
268         elif int(packet.getlayer(Ether).type) == 2054:
269             pkt.proto = 'ARP'
270             pkt.time = packet.time - self.init_time
271             pkt.src = str(packet.getlayer(ARP).psrc)
272             pkt.dst = str(packet.getlayer(ARP).pdst)
273             pkt.info = str(packet.summary())
274             #self.packet_table.row_append(src, dst, proto, info)
275             self.packets.append(packet)
276             self.pkts.append(pkt)
277         else:
278             print "Unresolved Protocol!"
```

3.3 IP 分片重组

当路由器收到一个数据包时，它会检查目的地址，并确定出接口使用，并且该接口的 MTU。如果分组的大小是比 MTU 大，并且在该分组的头中的不分段（DF）位被设置为 0，则路由器可分片传送数据包。

3.3.1 确定需要进行 IP 重组的数据包：

根据 IP 报头各字段内容进行解析。

对于每个解析后的数据包，保存其 IP 报头中各字段信息到 values，若 flags 三位数值为 0x01（010），代表分片后的数据包；若 flags 三位数值为 0x00（000）且 frag>0，代表分片后的最后一个数据包。这两种情况下，将数据包信息保存到字典 self.reassembling_fragdata 中，键为 id，值也为字典类型（键为偏移量，值为未解析的数据包类型）。

3.3.2 重组后数据包相应字段做出变化：

字典 self.reassembling_resultdata 保存分片重组后的显示内容，键为 id，值为重组后以字符串形式保存的数据包内容。

重组后的数据包 IP 头部中的 flags、frag 应变为 0x00，总长度也应作出相应变化，

将每个确定是分片后的数据包的总长度位转换为整数类型，进行计算，再去掉重复的 IP 报头部分，转换为字符类型添加到 self.reassembling_resultdata 字典的对应值中。

具体实现代码如下，

```
282         pkt.packet = packet
283         pkt.parser()
284         if pkt.proto=='ICMP' or pkt.proto=='UDP' or pkt.proto=='TCP':  #values[5]=flags values[6]=frag
285             keys,values = pkt.getInfo_IP()
286             if values[5]==0x01 or (values[5]==0x00 and values[6]>0):
287                 self.reassembling_fragdata.setdefault(values[4],{})
288                 self.reassembling_fragdata[values[4]][values[6]]=pkt.packet
289                 for frag_tmp in sorted(self.reassembling_fragdata[values[4]].keys()):
290                     if (frag_tmp==sorted(self.reassembling_fragdata[values[4]].keys())[0]):
291                         self.reassembling_resultdata[values[4]]=str(self.reassembling_fragdata[values[4]][frag_tmp])
292                         self.reassembling_resultdata[values[4]]=self.reassembling_resultdata[values[4]][0:20]+\
293                                                                 '\x00\x00'+self.reassembling_resultdata[values[4]][22:]
294                     else:
295                         self.reassembling_resultdata[values[4]]=self.reassembling_resultdata[values[4]]+\
296                                                                 str(self.reassembling_fragdata[values[4]][frag_tmp])[34:]
297                         tos_tmp11,=struct.unpack('B',self.reassembling_resultdata[values[4]][16])
298                         tos_tmp12,=struct.unpack('B',self.reassembling_resultdata[values[4]][17])
299                         tos_tmp21,=struct.unpack('B',str(self.reassembling_fragdata[values[4]][frag_tmp])[16])
300                         tos_tmp22,=struct.unpack('B',str(self.reassembling_fragdata[values[4]][frag_tmp])[17])
301                         tos_add=tos_tmp11*256+tos_tmp12+tos_tmp21*256+tos_tmp22-20
302                         Barray=bytearray(self.reassembling_resultdata[values[4]])
303                         Barray[16]=tos_add/256
304                         Barray[17]=tos_add%256
305                         self.reassembling_resultdata[values[4]]=str(Barray)
```

3.4 包过滤

从 Filter 模块的 QInputBox 函数获取用户想要进行的有效的包过滤规则，传递给变量 self.filter，从而改变抓取数据包函数的参数三，即过滤规则。

```
sniff(iface=self.interface, prn=self.pkt_callback, \
        filter=self.filter, stop_filter=self.stop_filter)
```

过滤规则表达式：

```
[src/dst] host host
[src/dst] port port
ip/ip6 protochain protocol
```

3.5 数据包保存

3.5.1 保存为.pcap 类型文件

```
filename = str(filename)
pcap_writer = PcapWriter(filename+'.pcap')
pcap_writer.write(self.packets)
```

3.5.2 保存为 PDF 类型文件

```
packet.pdfdump(filename)
```

3.6 数据包查询

从主页面 GUI 下方输入框中获取用户输入的搜索规则（'**=**'），

```
rulestr = str(self.srchInput.text()).strip()
rule=rulestr.split('=')
```

确认规则后，停止抓取数据包，对已保存的解析包数组 pkts 进行遍历，按照对应属性值，输出符合搜索规则的数据包，

```
self.add_packet(pkt)
```

搜索查看完毕后，返回。清空已显示数据包内容，对未解析数据包数组进行遍历显示，并继续抓包

```
self.packetsList.clear()
for pkt in self.pkts:
    self.add_packet(pkt)
    self.start()
```

3.7 简单数据包发送

从 SendpktGUI 中获取想要发送的包的头部信息、数据部分（以 TCP 为例），发包间隔时间、发包数量，

```
RValues=IP(src=tmp['IPsrc'],dst=tmp['IPdst'],ttl=int(tmp['IPttl']))/
\TCP(sport=int(tmp['TCPsport']),dport=int(tmp['TCPdport']))/data

        self.inter=int(tmp['Inter'])
        self.count=int(tmp['Count'])
```

RValues 传递给 self.Sendpkt.pktrule，

self.Sendpkt.pktrule，self.Sendpkt.inter，self.Sendpkt.count 作为参数，传递给发包函数，

```
send(self.Sendpkt.pktrule,inter=self.Sendpkt.inter,count=self.Sendpkt.c
ount)
```

3.8 ARP 欺骗

3.8.1 ARP 欺骗（目标主机）

向目标主机发送 ARP 数据包，源 IP 地址为网关 IP 地址，源 MAC 地址为本机地址（默认），目的 IP 地址为目标主机 IP 地址，目的 MAC 地址为目标主机 MAC 地址，此时对于目标主机来说，ARP 缓存内本机为网关。

3.8.2 ARP 欺骗（网关）

向网关发送 ARP 数据包，源 IP 地址为目标主机 IP 地址，源 MAC 地址为本机地址（默认），目的 IP 地址为网关 IP 地址，目的 MAC 地址为网关 MAC 地址，此时对于网关来说，ARP 缓存内本机为目标主机。

由此目标主机与网关之间传送的数据包都将经过本机，ARP 欺骗达成。

```
def poison_target(gateway_ip, gateway_mac, target_ip, target_mac):
    poison_target = ARP()
    poison_target.op = 2
    poison_target.psrc = gateway_ip
    poison_target.pdst = target_ip
```

```
poison_target.hwdst = target_mac
poison_gateway = ARP()
poison_gateway.op = 2
poison_gateway.psrc = target_ip
poison_gateway.pdst = gateway_ip
poison_gateway.hwdst = gateway_mac
```

3.8.3 ARP 欺骗恢复

向网关和目标主机分别发送 ARP 数据包，源、目的 IP 地址与 MAC 地址正确对应，此时网关和目标主机的 ARP 缓存均恢复正确。

```
send(ARP(op=2,psrc=gateway_ip,pdst=target_ip,hwdst='ff:ff:ff:ff:ff:ff',\hwsrc=gateway_mac), count=5)
send(ARP(op=2, psrc=target_ip, pdst=gateway_ip, hwdst="ff:ff:ff:ff:ff:ff",\hwsrc=target_mac), count=5)
```

4. 程序测试截图以及说明：

4.1 程序功能列表：

①指定侦听网卡，侦听数据包并解析内容；
②数据包全部数据显示；
③IP 分片重组；
④包过滤；
⑤数据包查询；
⑥数据包保存；
⑦简单数据包发送；
⑧ARP 欺骗；

4.2 功能测试：

4.2.1 指定侦听网卡，侦听数据包并解析内容：

A.运行程序，打开网卡选择页面（Interface），如图 4-1；

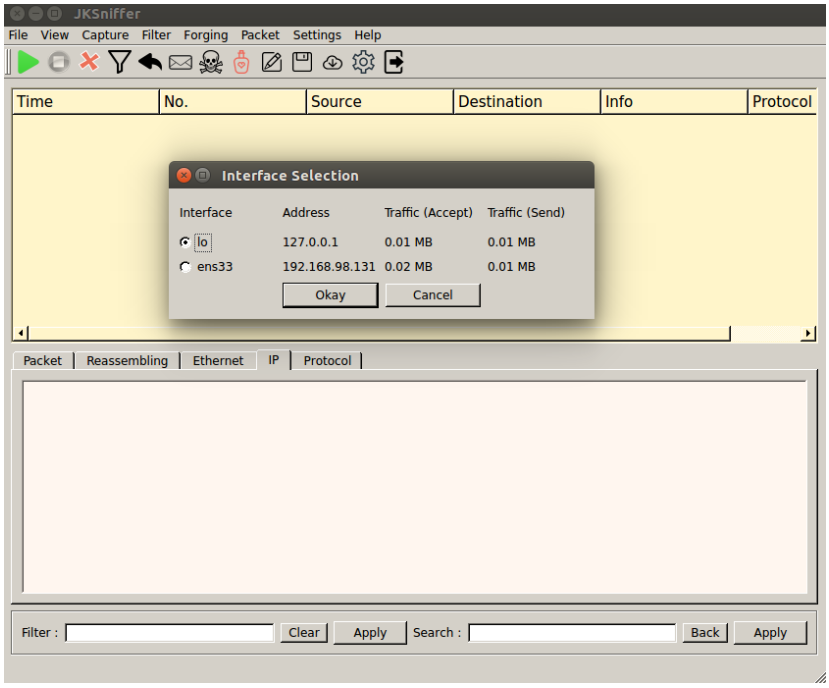


图 4-1 网卡选择页面 1

B.选择想要侦听的网卡，如图 4-2；

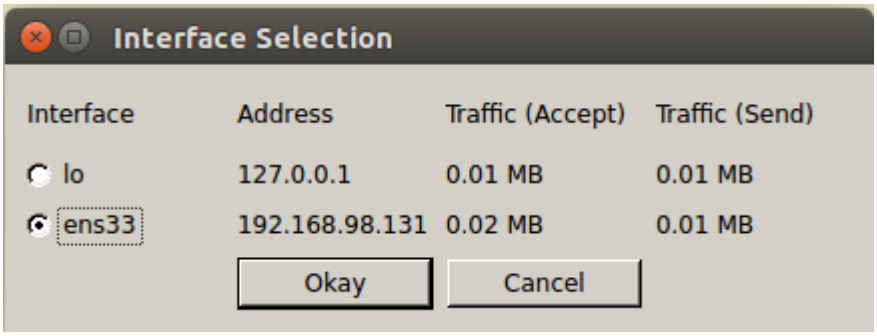


图 4-2 网卡选择页面 2

C.开始侦听；

D.通过此网卡进出本主机的数据包被嗅探到，信息被显示出来，如图 4-3；

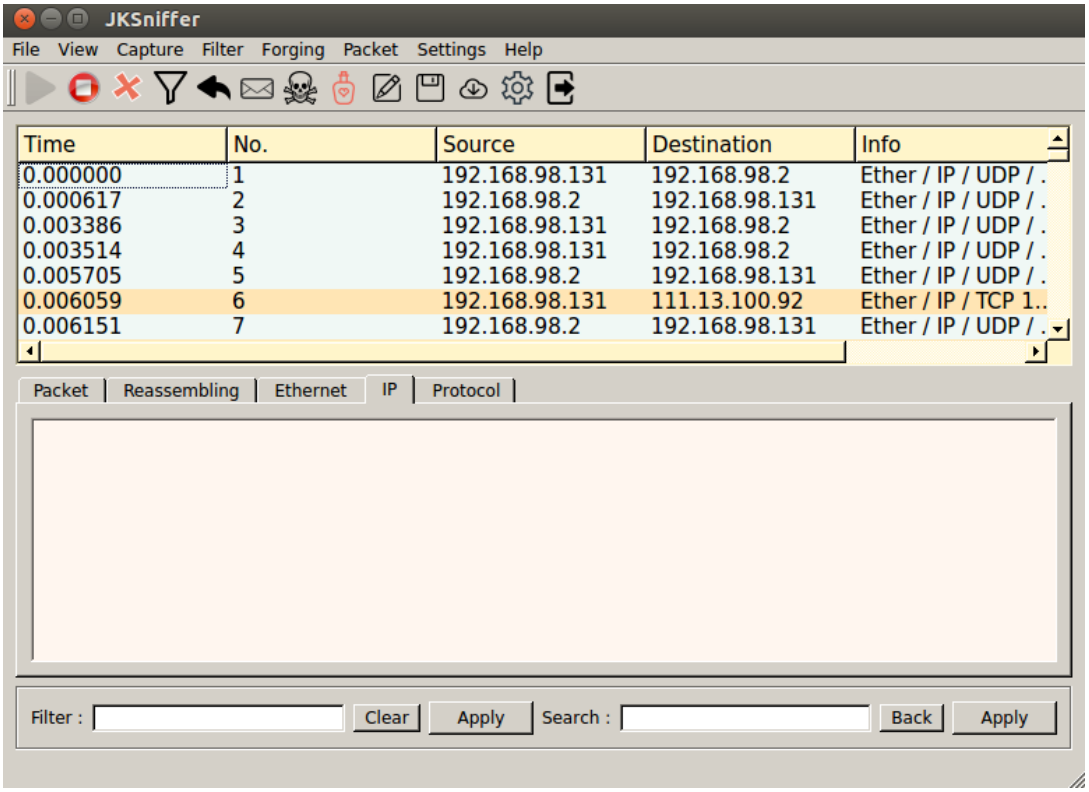


图 4-3 嗅探到数据包并显示

说明：

A.默认网卡选择是 lo；

B.改变侦听的网卡时，需要停止抓包，再进行切换；

4.2.2 数据包全部数据显示：

A.运行程序，开始侦听，嗅探到数据包；

B.选择想要查看全部数据的数据包；

C.数据包数据和解析得到的信息被显示在下面的标签中，如图 4-3（a）（b）；

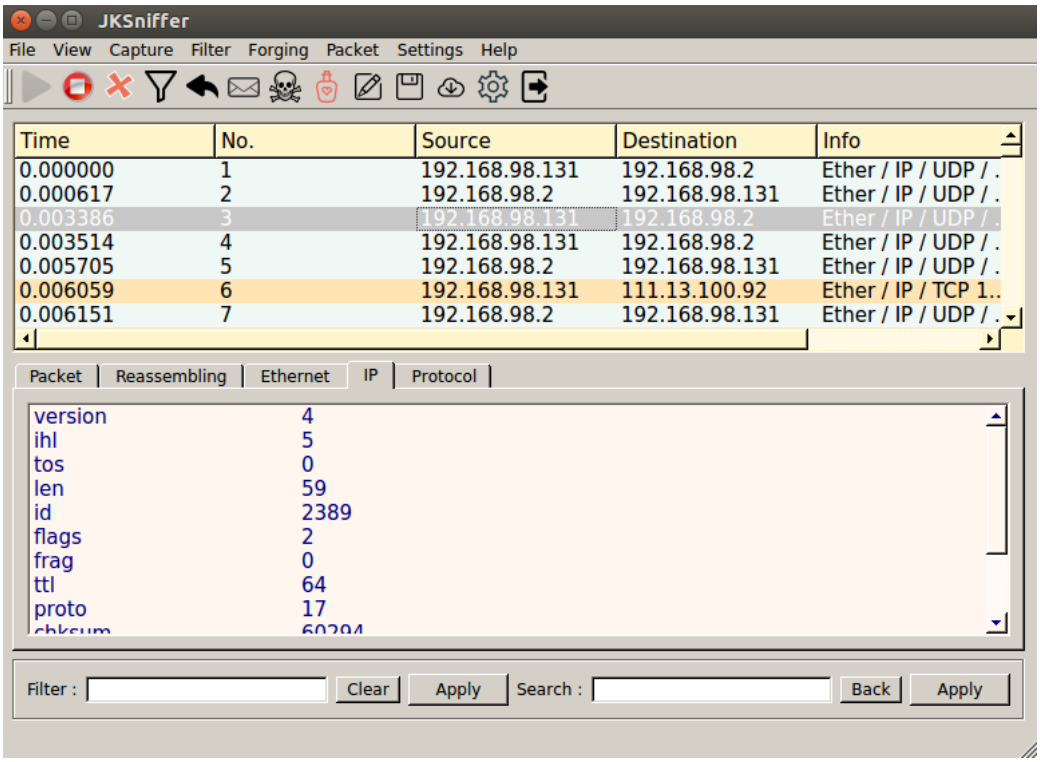


图 4-3（a）选定数据包显示 IP 层信息

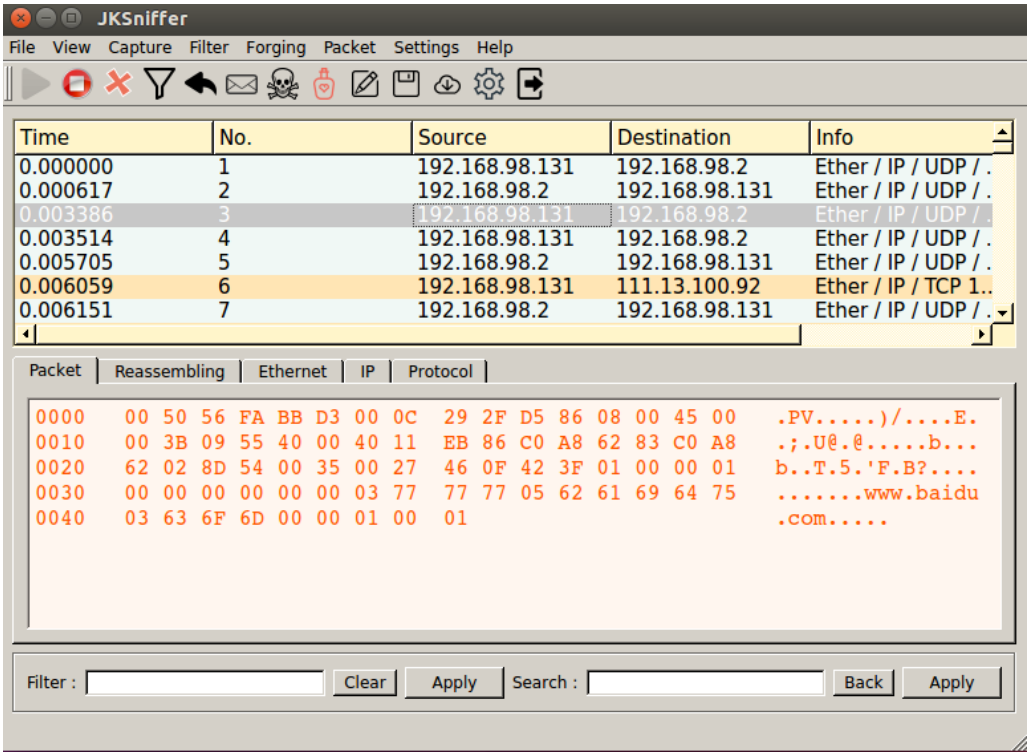


图 4-3（b） 选定数据包显示数据包具体信息

4.2.3 IP 分片重组：

A.运行程序，开始侦听，嗅探到数据包；

B.找到被拆分的数据包（flags=1），选择，如图 4-4（a）（b）；

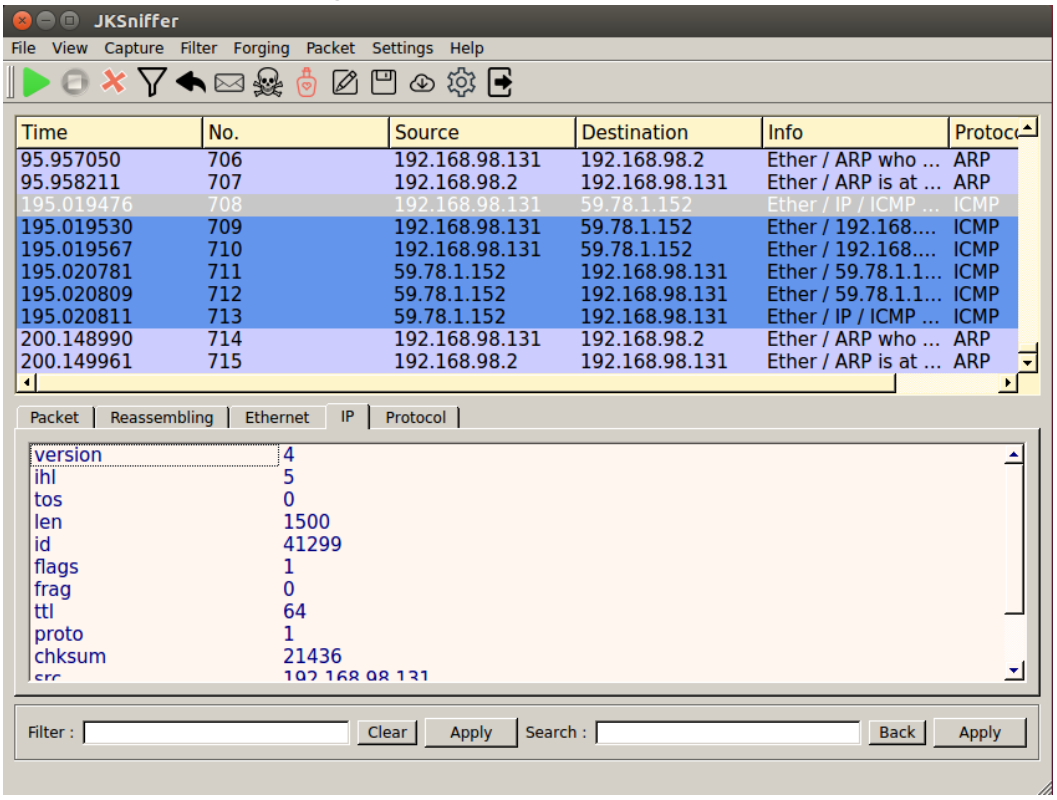


图 4-4（a） 找到分片后的数据包

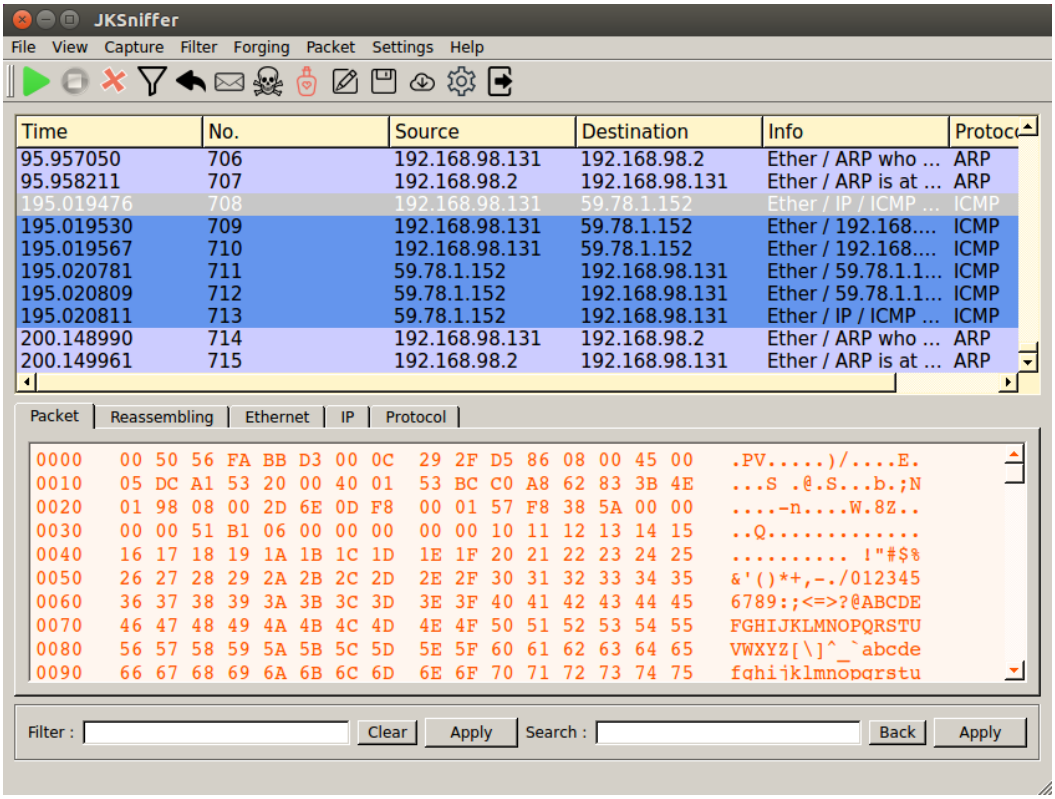


图 4-4（b） 选择该数据包

C.在下面 Reassembling 标签中查看重组之后的数据包数据，如图 4-5；

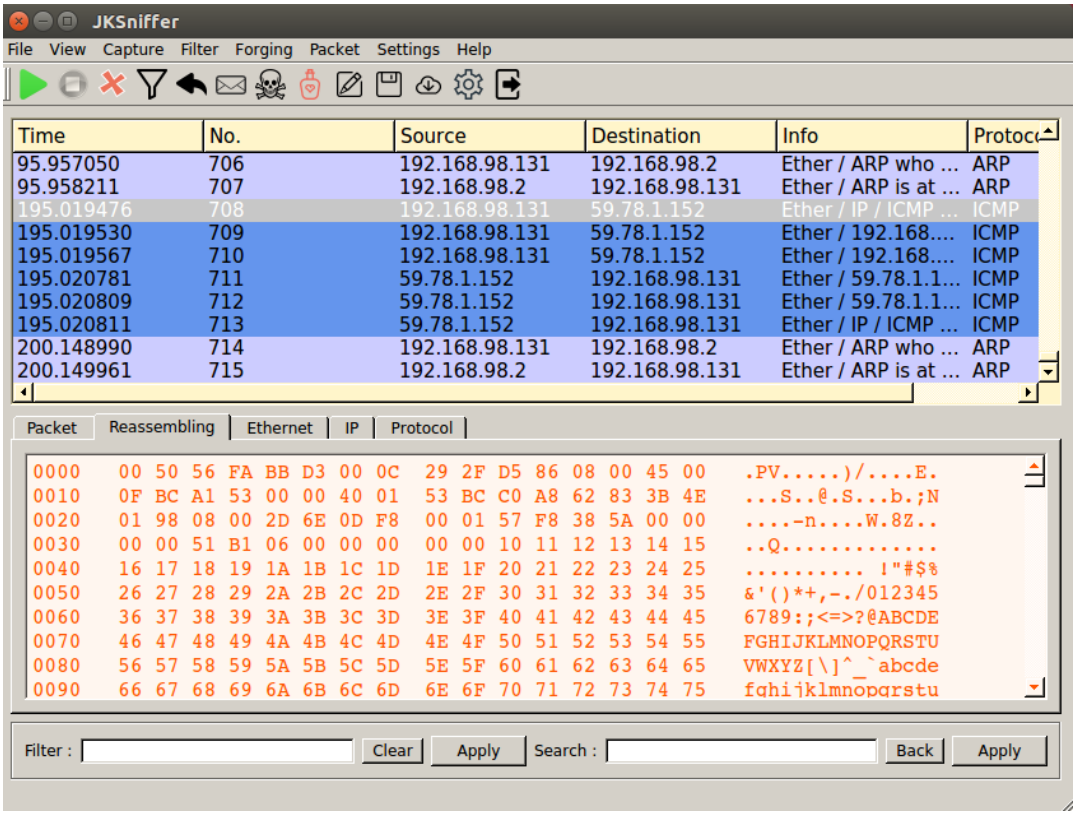


图 4-5　查看重组之后数据包数据

说明：

A.测试方式为，ping 局域网内主机一次，发送大小为 4000 字节的　数据包，加上 ICMP 报头 8 字节与 IP 报头 20 字节，共 4028 字节，如图 4-6。

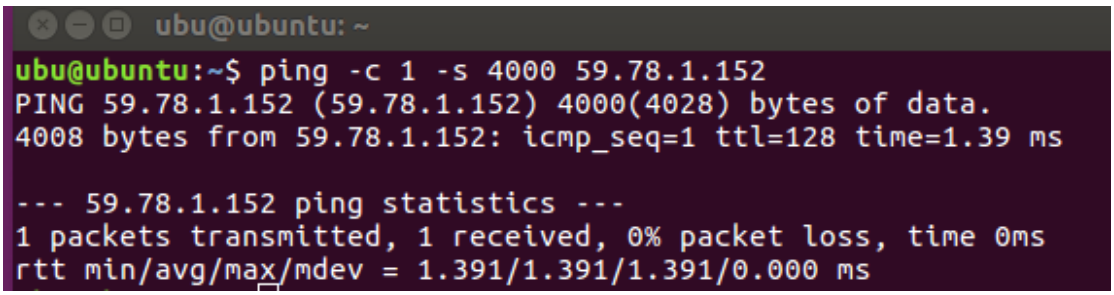
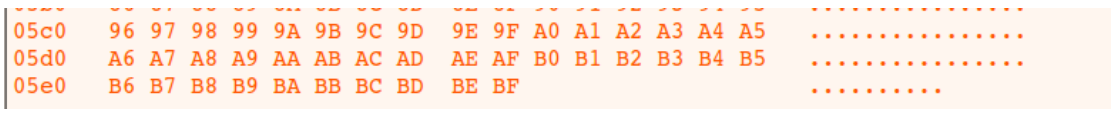


图 4-6 发送 4000 字节数据包

该数据包由于长度大于 MTU（1500 字节），被分片成三个数据包，分别为 1514,1514,1082 字节，共 4110 字节，如图 4-7（a）（b）（c）。其中每个数据包都包含了了 14 字节的 Ethernet 头部、20 字节的 IP 头部。（4110=4028+20+20+14*3）

分为三个数据包，多了两个 Ethernet 头部（14 字节）、IP 头部（20 字节），所以重组后的数据包长度应为 4042（4110-20-20-14-14）字节。



```
05c0   5E 5F 60 61 62 63 64 65  66 67 68 69 6A 6B 6C 6D   ^_`abcdefghijklm
05d0   6E 6F 70 71 72 73 74 75  76 77 78 79 7A 7B 7C 7D   nopqrstuvwxyz{|}
05e0   7E 7F 80 81 82 83 84 85  86 87                     ~.........
```

```
0400   66 67 68 69 6A 6B 6C 6D  6E 6F 70 71 72 73 74 75   fghijklmnopqrstu
0410   76 77 78 79 7A 7B 7C 7D  7E 7F 80 81 82 83 84 85   vwxyz{|}~.......
0420   86 87 88 89 8A 8B 8C 8D  8E 8F 90 91 92 93 94 95   ................
0430   96 97 98 99 9A 9B 9C 9D  9E 9F                     ..........
```

图 4-7（a）（b）（c） 分片后产生的三个数据包末尾部分

查看重组后数据包信息，数据包长度为 4042 字节，如图 4-8，

```
0fa0   76 77 78 79 7A 7B 7C 7D  7E 7F 80 81 82 83 84 85   vwxyz{|}~.......
0fb0   86 87 88 89 8A 8B 8C 8D  8E 8F 90 91 92 93 94 95   ................
0fc0   96 97 98 99 9A 9B 9C 9D  9E 9F                     ..........
```

图 4-8 重组后数据包末尾部分

重组后数据包 flags、frag 应为 0x00，总长度为 4028，如图 4-9，

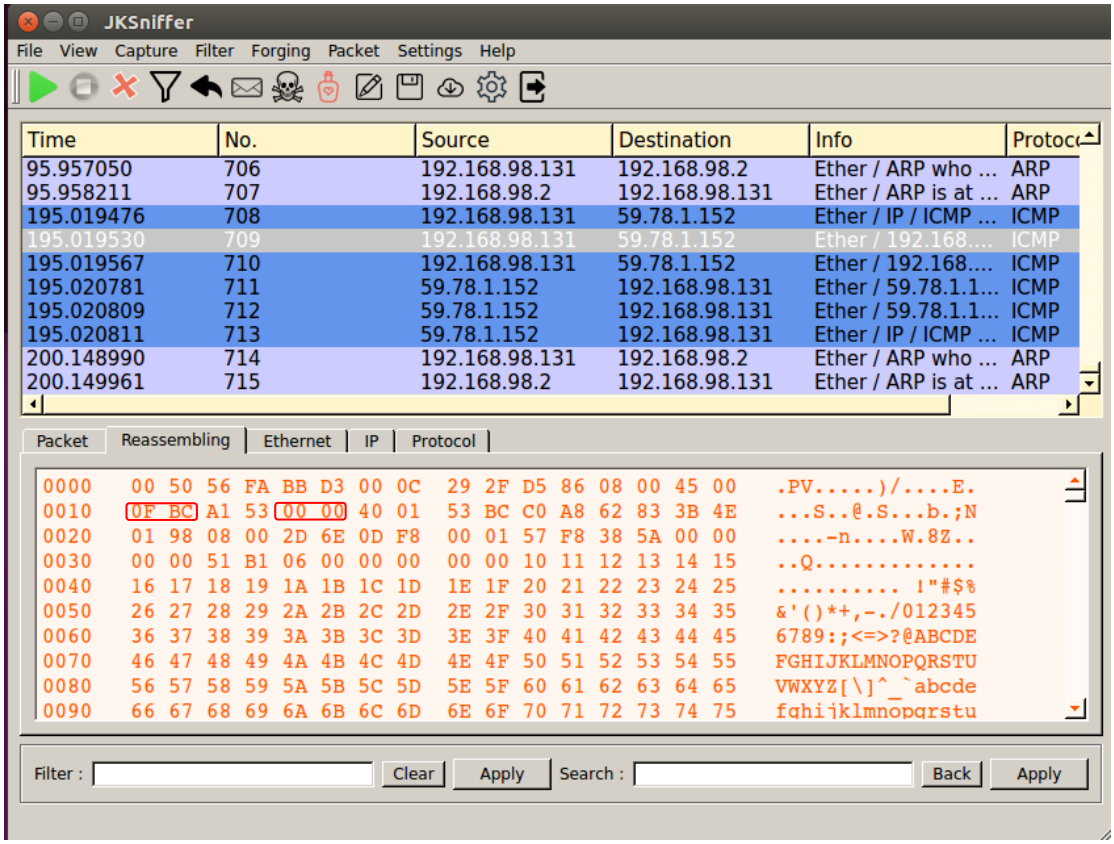


图 4-9 重组数据包头部做相应改变

B.重组数据包 IP 报头的 tos、flags、frag 均会根据实际情况进行调整，各个的数据部分（除 IP 报头）按序组合构成新的重组数据包的数据部分；

C.不需要进行 IP 重组的数据包显示包自身的数据；

4.2.4 包过滤：

A.侦听暂停状态下，打开包过滤页面输入相应的想要过滤的规则参数（或在主页面下方手动输入包过滤全部规则），如图 4-10；

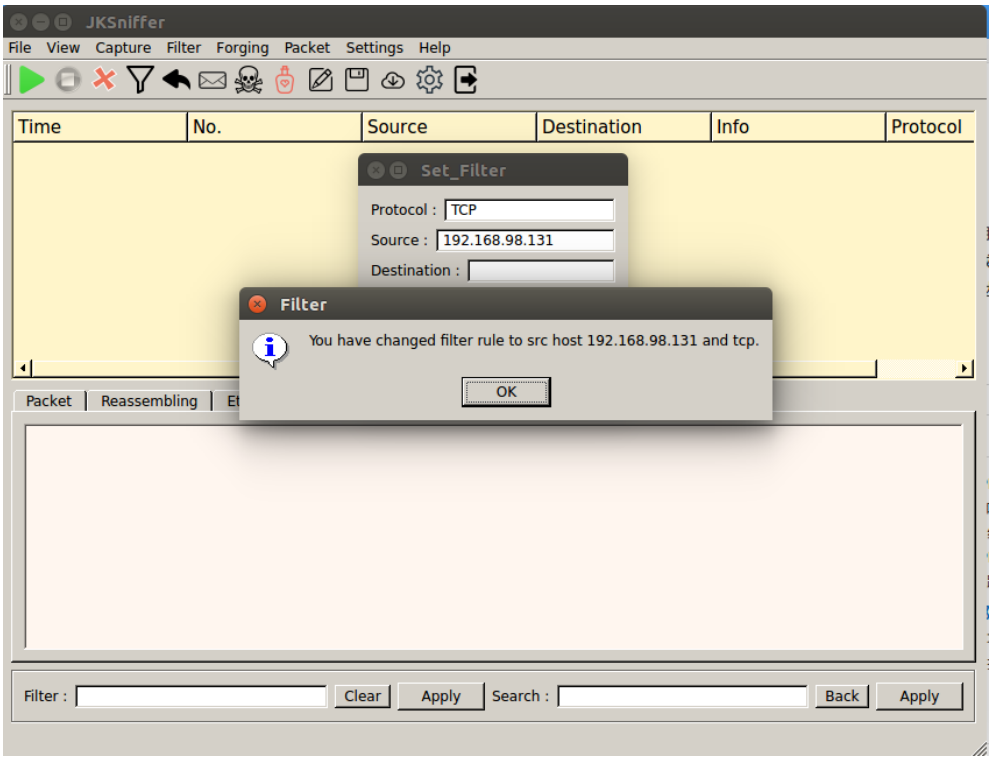


图 4-10 包过滤规则设置

B.规则确定后，开始侦听，如图 4-11；

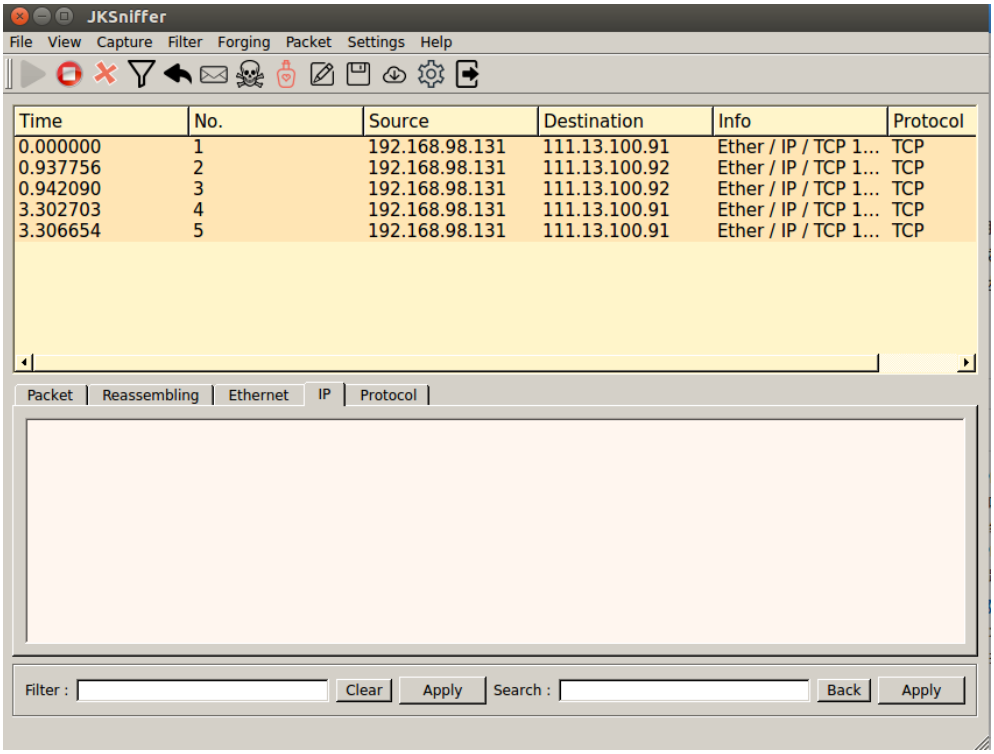


图 4-11 按照包过滤规则进行抓包

说明：

A.包过滤功能实现有两种方式，包过滤页面与主页面下方输入包过滤全部规则；

B.第一种方式输入规则参数，只有输入的参数是正确的才能更改当前包过滤规则（如IP 地址格式需正确、协议名称有效等），若部分正确部分错误，只对正确的部分进行更改，如图 4-12；

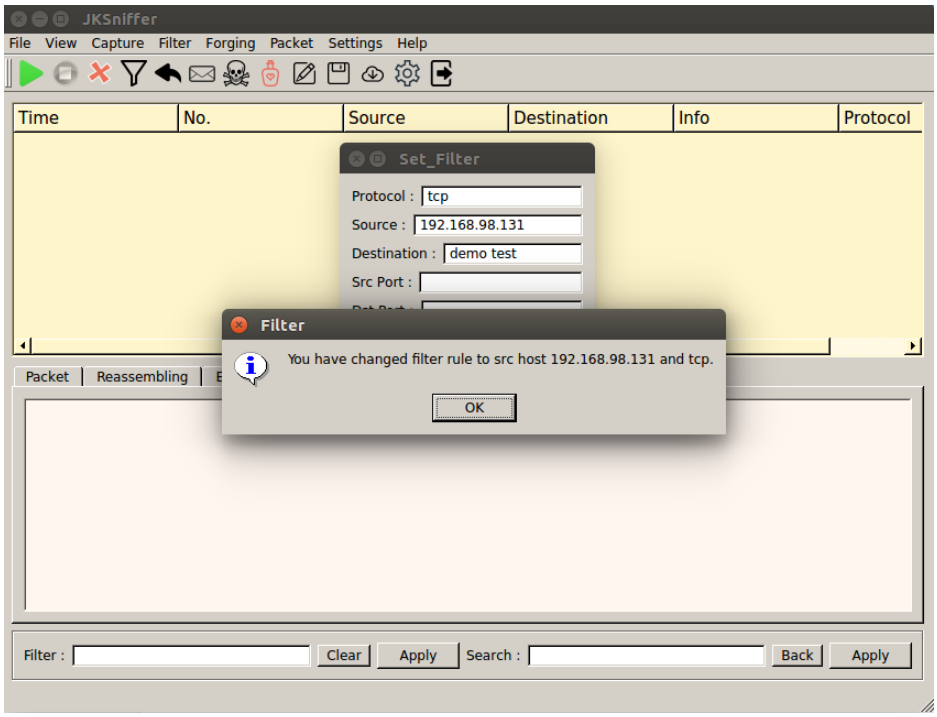


图 4-12 只有正确的包过滤规则才会被设置

C.第二种方式输入包过滤规则，只有规则被正确读取才能更改当前包过滤规则，如图 4-13；

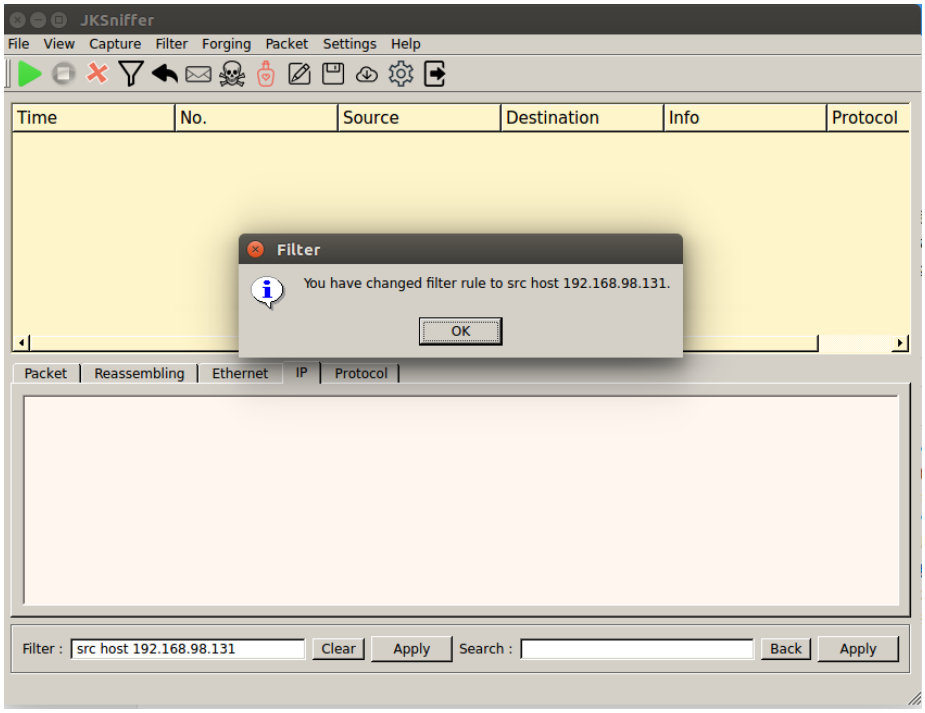


图 4-13 只有正确的包过滤规则才会被设置

D.正确更改规则后成功更改提示；

E.默认不进行过滤，Restore 可以清空过滤规则，如图 4-14；

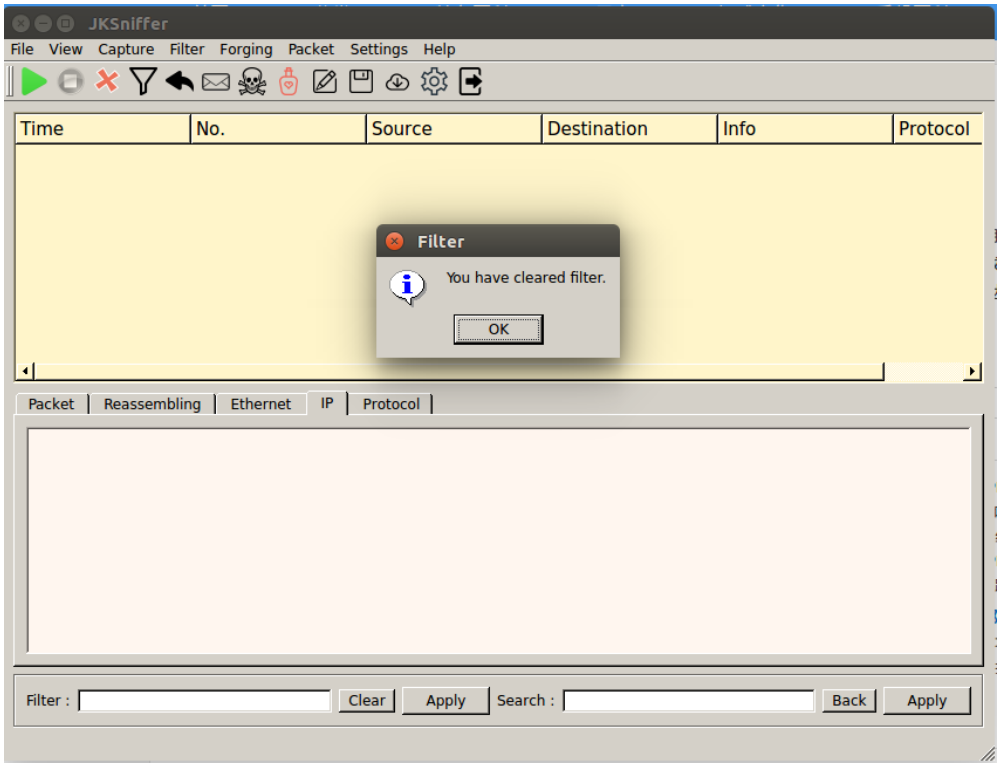


图 4-14 Restore 操作后，包过滤规则被清空

4.2.5 数据包查询：

A.运行程序，侦听到多个数据包，如图 4-15；

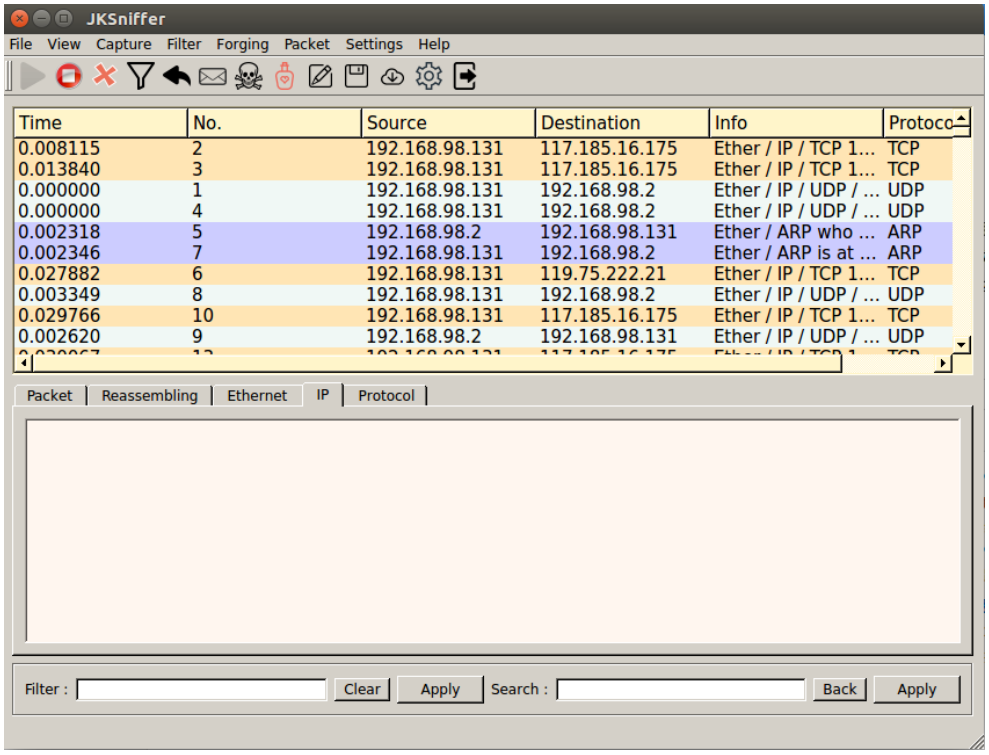


图 4-15 抓取到不同的数据包

B.在主页面下方输入数据包查询规则；

C.确认后显示规则规定的数据包，同时侦听暂停，如图 4-16；

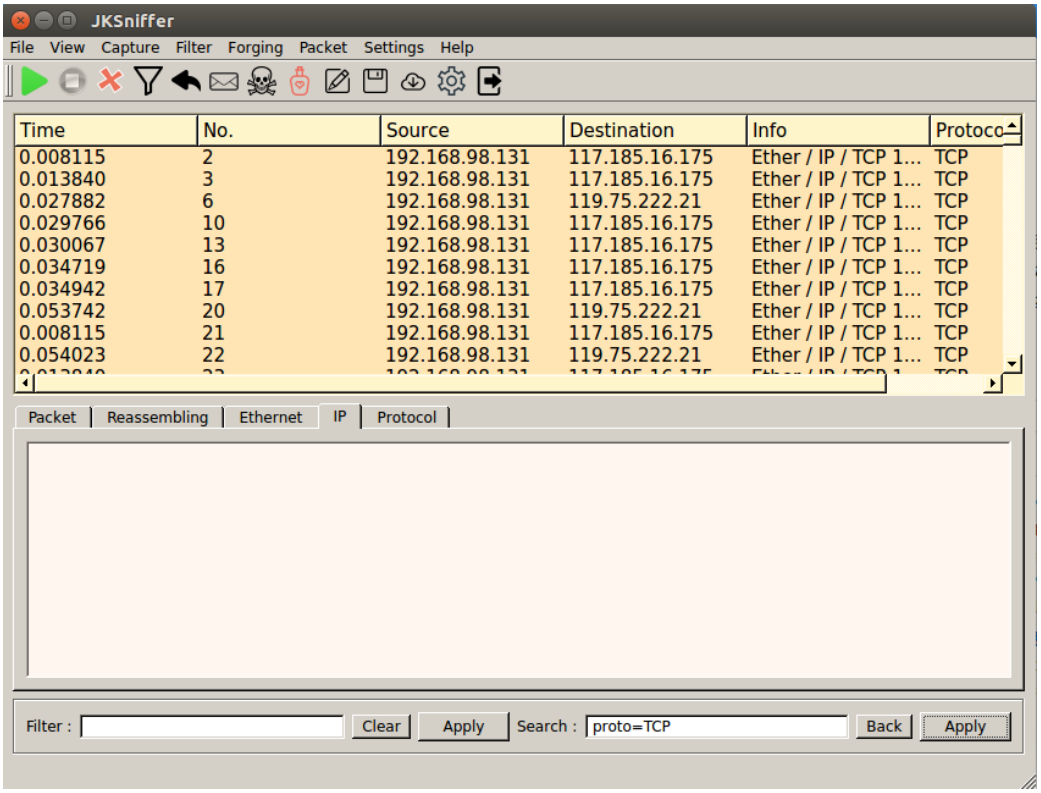


图 4-16 显示已经嗅探到的符合搜索规则的数据包

D.查看完毕，点击 Back 按钮返回，侦听继续，如图 4-17；

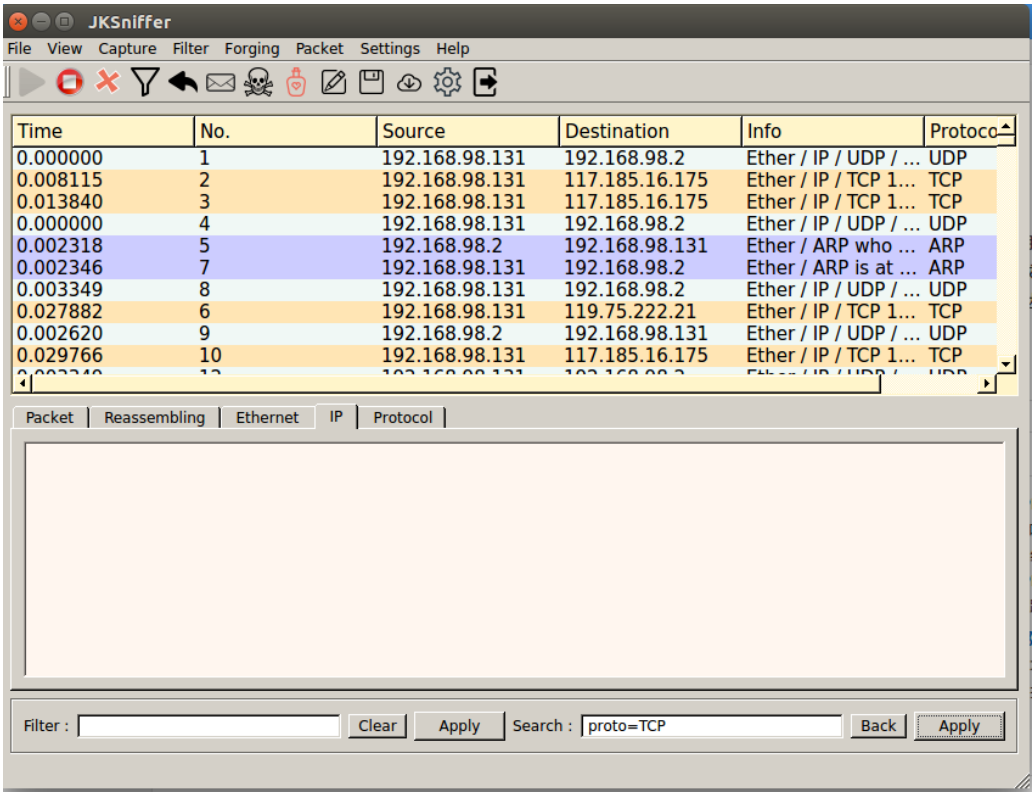


图 4-17 数据包搜索结束，返回

说明：

A.查询规则形式：***=***；

4.2.6 数据包保存：

A.选择数据包，保存，如图 4-18（a）（b）；

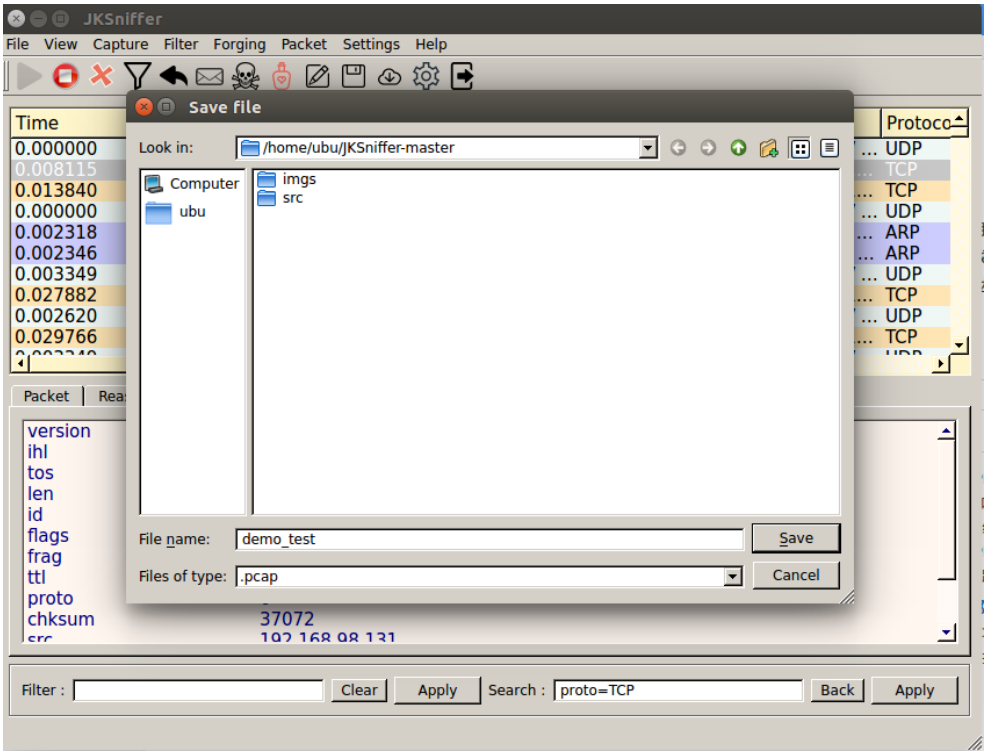


图 4-18（a） 选择数据包，保存为.pcap 文件

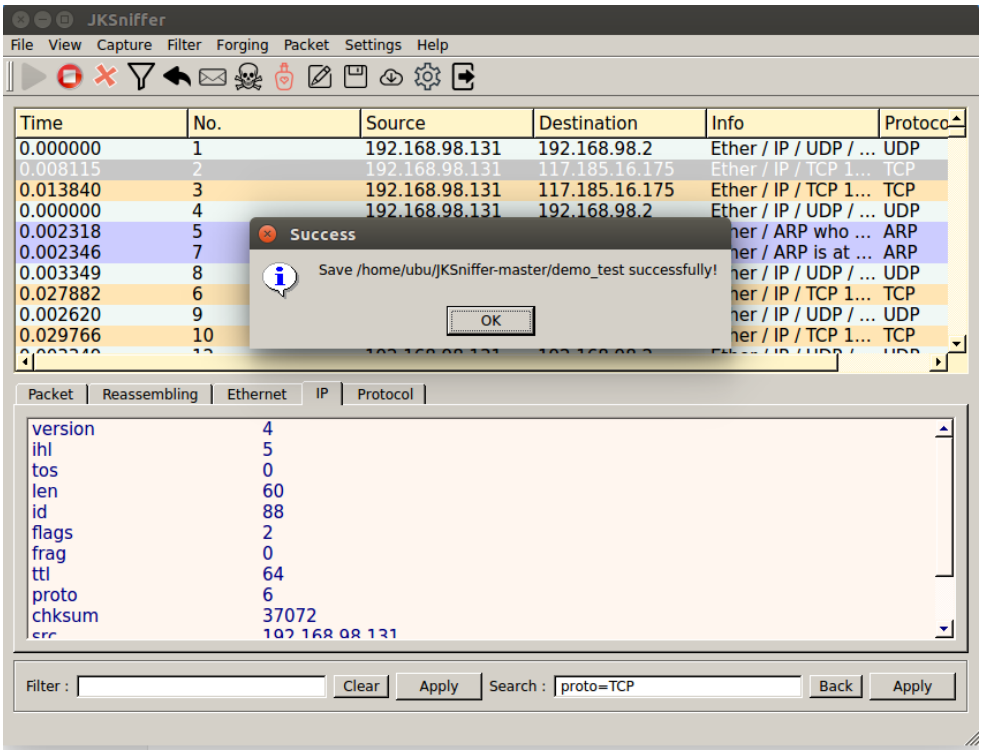


图 4-18（b）保存成功

4.2.7 简单数据包发送：

A.运行程序，打开发包页面；

B.输入数据包对应规则，如图 4-19；

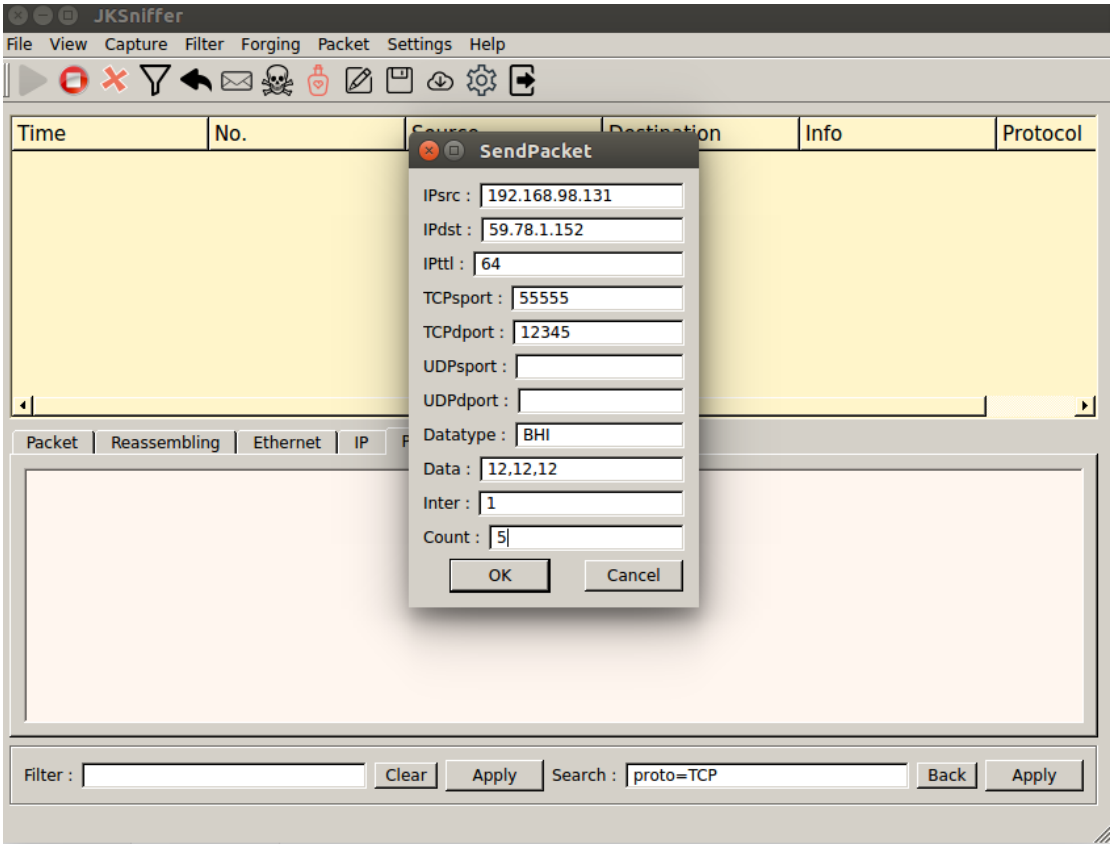


图 4-19 简单发包输入相应协议内容

C.确定后，数据包发送，可以被嗅探到，如图 4-20（a）（b）；

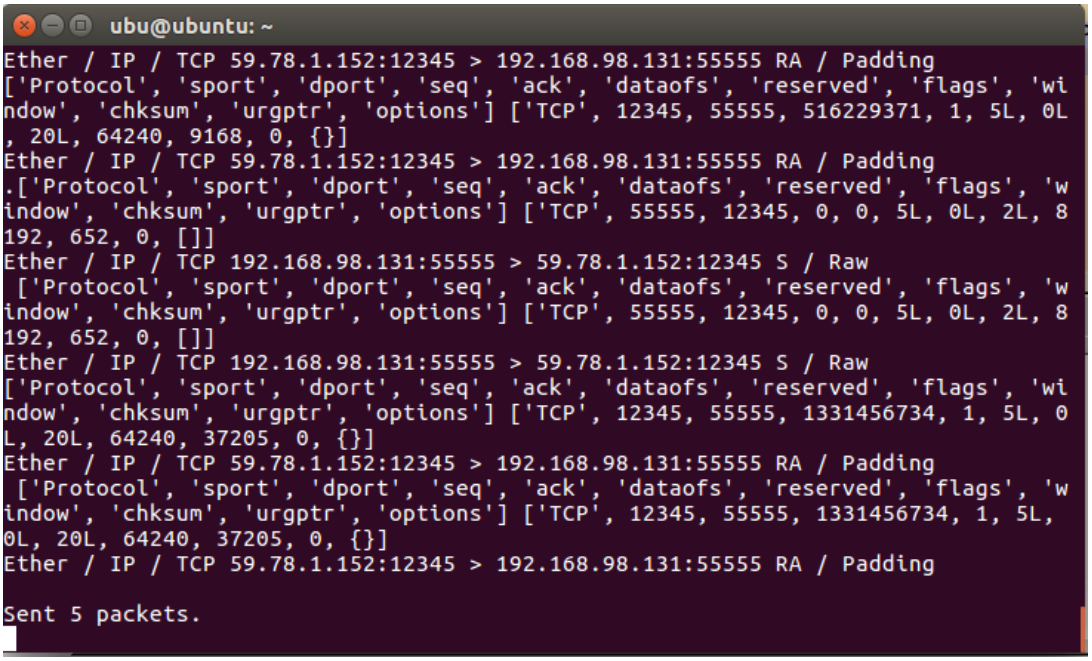


图 4-20（a） 成功发包

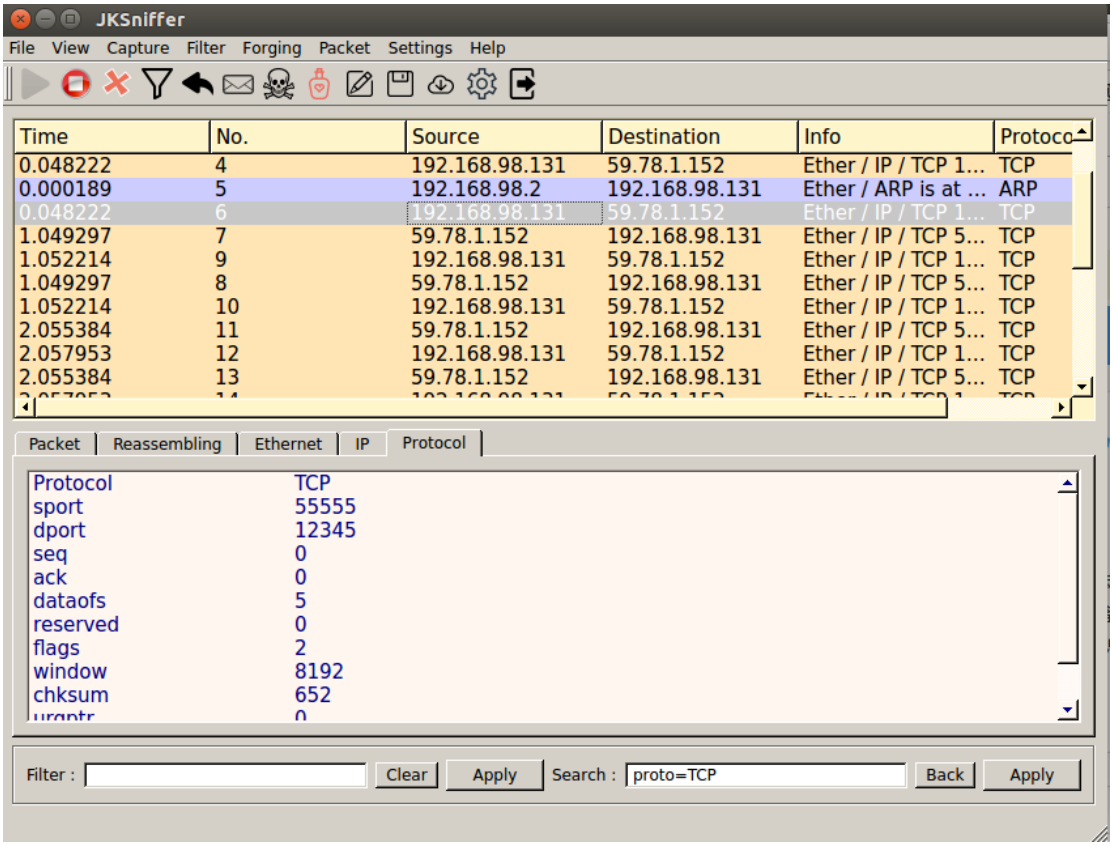


图 4-20（b）抓取到刚刚发送的数据包

4.2.8 ARP 欺骗：

A. 运行程序，打开 ARP Spoofing 页面，输入目标主机 IP 地址与网关 IP 地址，确认；

B. ARP 欺骗成功；

C. 点击 ARP Storing， 解除 ARP 欺骗；

说明：

A. 具体测试：

本机 IP 地址为 192.168.98.131，MAC 地址为 00:0C:29:2F:D5:86；

目标主机 IP 地址为 192.168.98.129，MAC 地址为 00:0C:29:FA:A8:51；

路由器（网关）端口地址为 192.168.98.1，MAC 地址为，00:05:56:C0:00:08 如图 4-21；

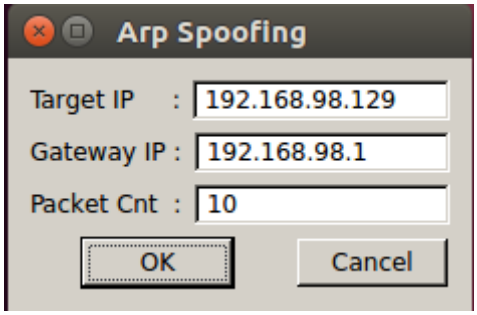


图 4-21 ARP 欺骗设置

ARP 欺骗攻击成功，如图 4-22，

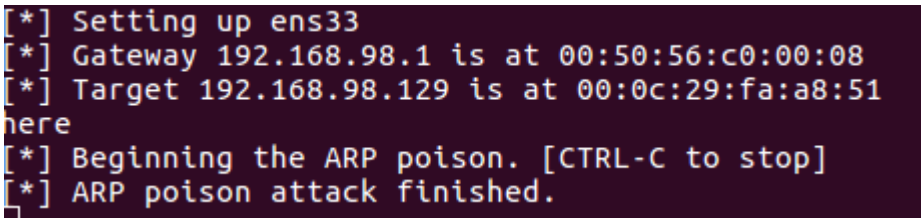
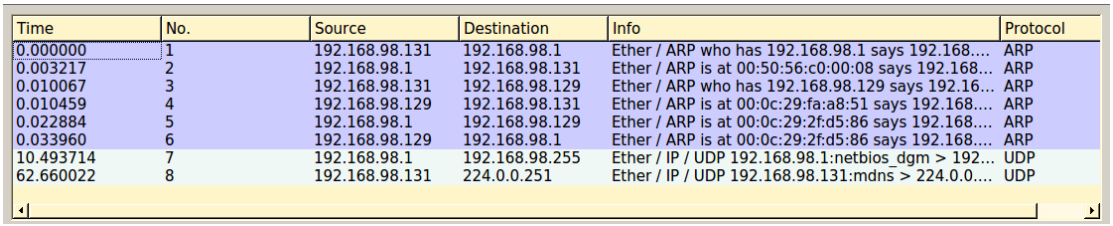


图 4-22 ARP 欺骗成功

抓取发送的 ARP 数据包，如图 4-23，



Time	No.	Source	Destination	Info	Protocol
0.000000	1	192.168.98.131	192.168.98.1	Ether / ARP who has 192.168.98.1 says 192.168....	ARP
0.003217	2	192.168.98.1	192.168.98.131	Ether / ARP is at 00:50:56:c0:00:08 says 192.168...	ARP
0.010067	3	192.168.98.131	192.168.98.129	Ether / ARP who has 192.168.98.129 says 192.16...	ARP
0.010459	4	192.168.98.129	192.168.98.131	Ether / ARP is at 00:0c:29:fa:a8:51 says 192.168....	ARP
0.022884	5	192.168.98.1	192.168.98.129	Ether / ARP is at 00:0c:29:2f:d5:86 says 192.168....	ARP
0.033960	6	192.168.98.129	192.168.98.1	Ether / ARP is at 00:0c:29:2f:d5:86 says 192.168....	ARP
10.493714	7	192.168.98.1	192.168.98.255	Ether / IP / UDP 192.168.98.1:netbios_dgm > 192...	UDP
62.660022	8	192.168.98.131	224.0.0.251	Ether / IP / UDP 192.168.98.131:mdns > 224.0.0....	UDP

图 4-23 抓取到 ARP 欺骗实施过程发送的数据包

分析如下，

No.1 数据包：本机（192.168.98.131）询问路由器（192.168.98.1）MAC 地址；

No.2 数据包：路由器（192.168.98.1）应答本机（192.168.98.131），发出自己的 MAC 地址；

No.3 数据包：本机（192.168.98.131）询问目标主机（192.168.98.129）MAC 地址；

No.4 数据包：目标主机（192.168.98.129）应答本机（192.168.98.131），发出自己的 MAC 地址；

No.5 数据包：ARP 欺骗包，本机（192.168.98.131）冒充路由器（192.168.98.1）向目标主机（192.168.98.129）发包，声明路由器（192.168.98.1）MAC 地址为 00:0C:29:2F:D5:86（实际为本机（192.168.98.131）MAC 地址）；

No.6 数据包：ARP 欺骗包，本机（192.168.98.131）冒充目标主机（192.168.98.129）向路由器(192.168.98.131)发包，声明目标主机(192.168.98.129)MAC 地址为 00:0C:29:2F:D5:86（实际为本机（192.168.98.131）MAC 地址）；

此时 ARP 欺骗完成，目标主机（192.168.98.129）认为本机（192.168.98.131）为路由器（192.168.98.131），而路由器（192.168.98.131），认为本机（192.168.98.131）为目标主机（192.168.98.129）。目标主机（192.168.98.129）与路由器（192.168.98.131）之间通信的数据包都将经过本机（192.168.98.131）。

5. 遇到的问题和解决方法

5.1 数据包变量无法直接赋值与网络数据类型‘\x00’处理

大多数情况下，网络嗅探器均为读取嗅探到的数据包的数据，但是当想实现对数据包内容进行提取并更改又不改变原数据包内容时，类似 a=packet 或 a=packet.getlayer(Ether)无法达到预想中的效果。原因是这种赋值在 python2.7 中类似于加上一个标签，而不是复制。解决办法是将数据包内容强制类型转换为字符串后再进行赋值。字符串中的字符为网络数据形式'\x00'，不能直接进行 int 类型转换、加减运算等操作。转化为整数类型可用 struct.unpack('B','\x00'),运算后整数类型转化为网络数据形式可用 bytearray（）与强制类型转换。

但是这种办法效率较低，后面有机会可以继续探究更有效的解决办法。

5.2 IP 分片重组测试困难

课堂上老师详细讲述了 IP 分片重组的原理及方法，在了解后，实现 IP 分片重组功能不是十分困难，但在实际测试过程中会发现几乎找不到数据包是分片的，也就无法对这一功能进行测试。所以我们采用 ping 给内网主机一个大于 MTU 的数据包，并进行分析。实际测试时，注意 ubuntu 终端中的 ping 命令与 windows cmd 中的用法不同。若想向 192.168.1.1 发送一个大小为 4000bytes 的包，

```
Ubuntu:ping -c -1 -s 4000 192.168.1.1
Windows:ping -n 1 -l 4000 192.168.1.1
```

5.3 丢包问题

在使用 socket 和 scapy 初始抓包成功并解析后，我们和 wireshark 软件抓包的情况进行了对比，发现在访问视频流等数据时，会造成大量包缺失。经过相关资料查询并向老师咨询后，发现这是由于处理的速度跟不上抓包的速度（由于将处理和抓包在一个进程内完成，即对每个包抓取后直接进行处理）。因此，在查阅了相关资料的情况下，我们将抓包和处理包分成两个进程来处理，从而解决了这个问题。在视频流、下载大文件等需要大量包传输的情况下，我们的程序能够保证和 wireshark 抓包结果基本相同（不支持的协议除外）。

5.4 程序并发要求（线程/进程）

在 JKSniffer 的实现过程中，十分关键的一点就是多线程（进程）的运用。由于抓包、解析包、GUI、发包等需要同时进行，即不能产生丢包或让用户感觉到明显的延迟，进程或线程的运用是非常关键的一点。在经过广泛的查阅资料后，我们发现，强行对进程 kill 是非常不好的做法，也是不推荐的做法，而且这样容易产生平台依赖等不好的结果。最后，我们采取了 Scapy 自带的 filter 选择停止函数、threading 库以及 PyQt4 提供的 Qthread 完成了程序并发执行的需求，也避免了丢包、卡顿等现象的发生。

6.体会与建议

本次大作业我们在一开始使用 python 语言进行 socket 编程，实现抓取数据包、解析数据包、IP 分片重组功能。在这之后，实现例如文件存储（含 pdf 格式）、发送数据包、ARP 欺骗等拓展功能时，我们感觉到 socket 编程的可拓展性不强，继续下去难度较大且可拓展性不强（代码量较多）。由于我们已经熟悉包解析的过程，所以我们最终决定迁移到 scapy，

利用其中比较成熟的函数继续完善本次网络嗅探器的开发工作，实现更多高级功能。

从前期利用 socket 编程进行数据包的抓取、解析等，到后面利用 PyQt4 库与 scapy 库进行网络嗅探器的设计与实现，使得我们更深入、具体地了解了课堂上学习到的关于计算机通信网络知识在实际中的实现应用。我们希望借本次大作业的机会，对计算机通信网络中嗅探、简单发包、ARP 欺骗等内容做一些尝试和了解，而不仅仅是为了完成作业任务。

在实际进行过程中，由于是两人合作，所以在一个人基于另外一个人的工作基础上再开展前，我们发现往往需要一次较为深入的探讨。这样的讨论，一方面是让彼此互相了解对方使用的一些函数与变量的调用关系，使得能够避免在进一步开发过程中走弯路，让开发过程更迅速、高效；另一方面，讨论的内容不仅局限于代码相关，更要交换对于功能实现方式、任务优先级、用户界面等看似简单却十分重要的方面的看法。从开始的跌跌撞撞到后期项目进展突飞猛进，这次实践机会让我们第一次深切地体会到一个团队如果能够配合默契、交流深入，是能够做到“1+1>2”的。这也是对于我们这些在校学生今后在一个团队中能更好发挥作用的一些启示。

在界面化设计部分，我们时刻谨记从用户者的角度想问题，能够让整个项目更加成熟和友好。同时，我们为增强程序的通用性和鲁棒性查阅了很多异常情况控制、UI 设计的相关书籍和资料，使我们收获颇多。

通过这次两人配合的大作业实践，我们也更加熟练地学会如何快速学习和应用一个陌生的库，规范地统一代码风格等。总的来说，这次大作业使得我们在许多方面都成长很多。关于建议，我们希望在布置大作业的时候，能附加一些简单的实现思路。比如我们这次大作业的 IP 分片重组功能，当时实现的时候课堂上还未讲到这部分的内容，所以从查找资料，到清晰思路和尝试实现，颇费了一番周折，实际上这一部分功能并不复杂。前面的相关知识经过老师讲解后，后面的实践能够对它们再一次加深理解，也提高项目的进行速度；而还没讲到的部分，希望能有一些提示和介绍，使得整个过程的效率得到提高。

最后十分感谢老师和助教在整个开发过程中给予诸多耐心的解答与帮助，让我们能够克服许多困难，也少走很多弯路。