

计算机通信网络大作业

——网络嗅探器 (Sniffer)

张晴钊 杨正宇

一、概述

1.1 报告概要

报告主要分为四个部分。

第一个概要部分主要阐述运行环境，编译工具，程序文件列表等。

第二个 GUI 部分主要说明了 GUI 的实现算法，以及程序运行的截图。

第三个具体功能实现部分阐述了各功能实现的主要算法，程序测试截图及说明。

第四个部分是遇到的问题及解决方法。

第五个部分是体会与建议。

第六个部分主要的数据结构（由于篇幅过长，我把它放在了最后）。

1.2 运行环境

我们此次 sniffer 嗅探器是在 linux 操作系统下利用 C++ 在 qt 上编写,其中主要利用了 libpcap 库中的相关函数和数据结构。

此处简述抓包的实现，在 GUI 界面中开一个线程，利用 libpcap 中的函数进行抓包，参照各种格式的报文结构进行比特流分析，选出我们要需要的包的种类，将其信息存储在 vector 中，并将基本信息发送给 GUI 进行显示。

具体实现的过程在下文中会详细说明，若要深入理解，可以参看我们的源码。

关于具体分工以及使用流程也可以参看 readme 文件。

1.3 目录结构

mainwindow.h/cpp/ui: 主窗口文件，包含大部分功能的调用。

networkchoice.h/cpp/ui: 网络选择对话框，在软件开启时打开，或者在主窗口调用。

filedialog.h/cpp/ui: 文件提取对话框，在主窗口调用。

sniffer.h/cpp: 抓包器的基础类，包装基本的 libpcap 库函数。

csniffer.h/cpp: 继承 sniffer，生成方便调用的借口。包含获取网络，监听网络，抓取包，保存文件等操作。被 networkchoice, mainwindow, capturethread 调用。

capturethread.h/cpp: 抓包线程类。由 mainwindow 调用，完成抓包动作，调用后续处理模块 multiview, filter 等。

listview.h/cpp: 实现网络包摘要信息列表的生成，预处理、格式化包信息，为后续处理做准备。

multiview.h/cpp: 继承 listview，实现树形详细信息表以及十六进制字节流的显示。完成对包的逐字节解析。

filter.h/cpp: 包过滤功能模块，被 capturethread 和 mainwindow 调用。

slideinfo.h/cpp: IP 分片重组功能模块。被 capturethread 调用。

type.h: 定义各种报文头部格式、标识宏定义以及全局结构体格式。

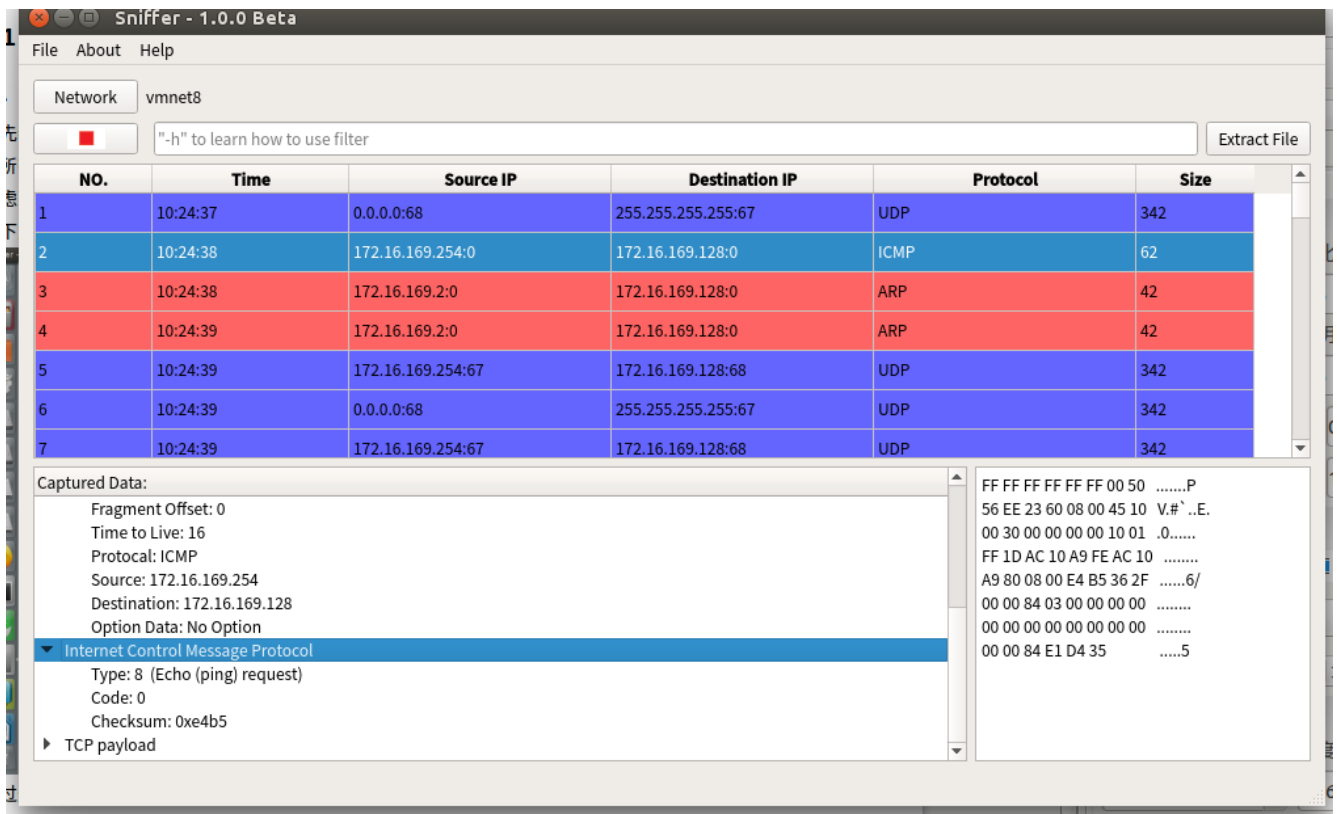
log.h: 调试辅助模块。

二、GUI

2.1 主界面

我们 GUI 的设计理念是追求页面的简化。摒弃繁杂的按钮、菜单和工具栏，而使用多功能的输入栏、动态的按钮、右键菜单、额外对话框使得整体页面简洁优雅。最终的主页面以 3 个展示栏目为主，有效的展示空间占页面总面积的 80% 以上。

首先参考助教在 ftp 中给出的案例以及 wireshark 的界面，大致了解了界面的主要架构，分为一个列表展示所有抓到的包，一个树型界面对抓到的包进行解析，还有一个区域存储该包的十六进制字节流。同时考虑到接下来功能的实现，还包括过滤器的输入栏，暂停开始按钮，上方的和该应用相关的菜单栏，参照下方的截图会比较直观：

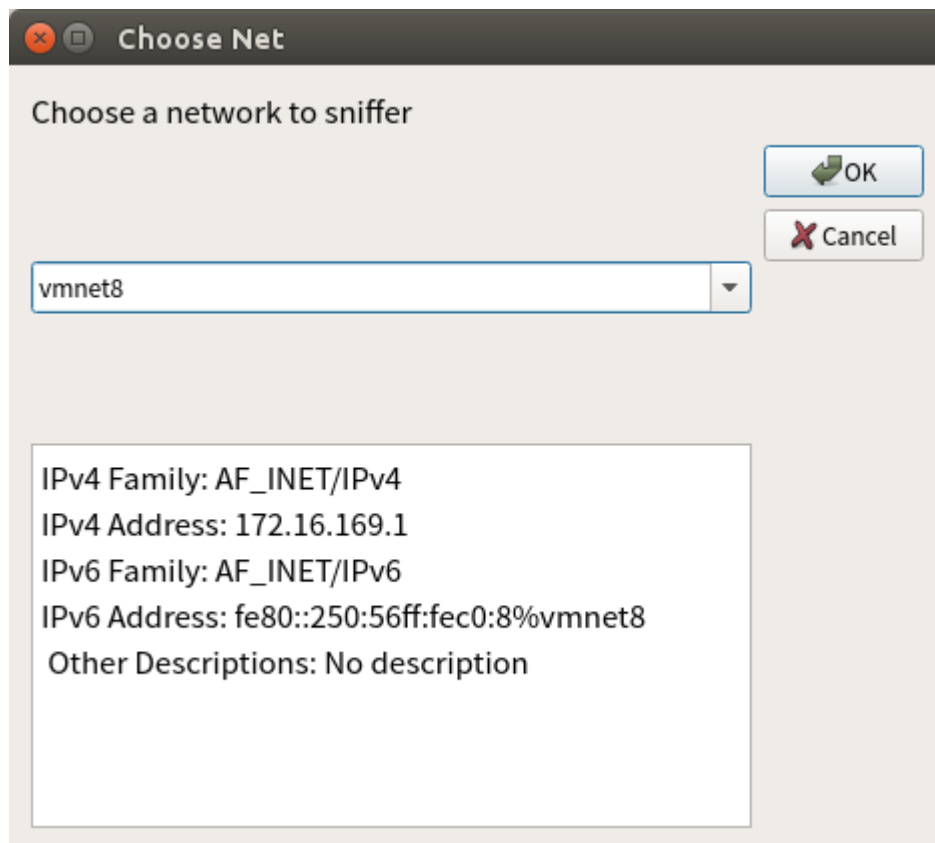


通过这个截图，大致能够明白界面各个部分的作用。其中 network 右方是显示当前网卡 (Vmnet8), 下方的红色正方形是暂停按钮，暂停按钮按下后会变为开始按钮。该按钮右方是过滤器输入器，输入器右方是文件提取，对应第七项功能，在之后细说。按钮下方是抓到的包列表，最下排左边是树型分析结果，右边是字节流。

主界面三个表的实现细节主要在 listview.h,listview.cpp,multiview.h,multiview.cpp 中。

2.2 网卡选择界面

这个界面比较简单，只要利用 libpcap 中的函数分析出所有网卡，然后展现在列表中并供用户进行选择即可，看截图便一目了然：

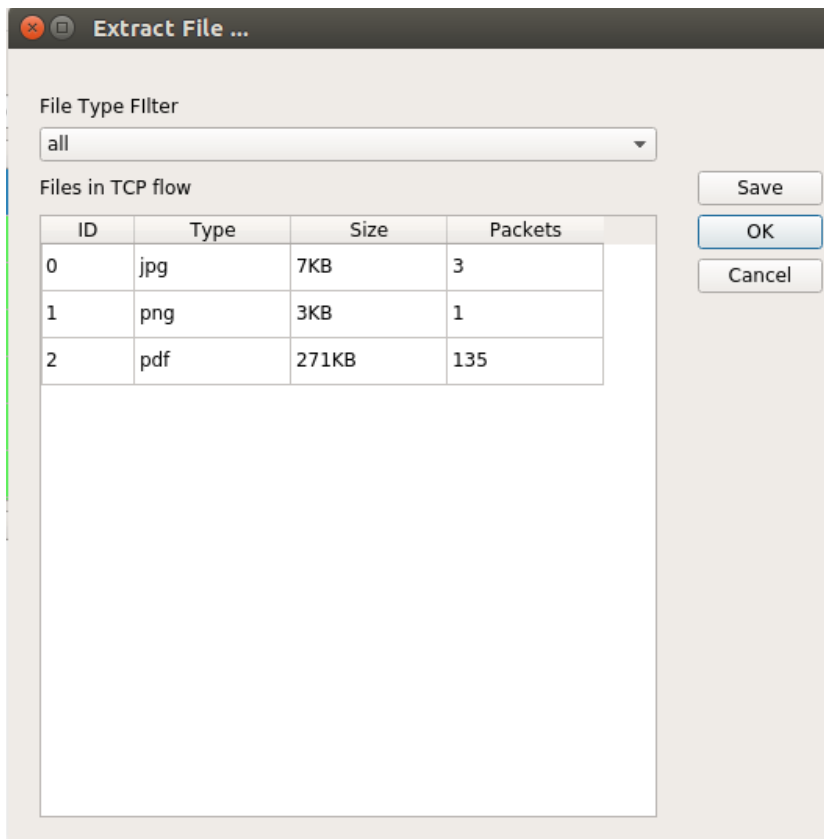


界面下部显示了该网卡的具体信息，点击 OK，或者 Cancel 便可转入抓包主界面。

其具体实现代码在 `networkchoice.h` 以及 `networkchoice.cpp` 中。该对话框在提供网络选择的同时，一目了然地展示对应的网络信息，使用标准确认对话框格式。

2.3 文件提取对话框

展示提取到的文件基本信息：文件编号、文件类型、文件大小以及分组的包数。复选框提供文件类型筛选功能，Save 按钮提供文件保存功能，OK 按钮讲返回主页面并筛选出传输选中文件的包。



三、具体功能实现

3.1 功能一的实现

3.1.1 ARP 包

arp 报文结构：

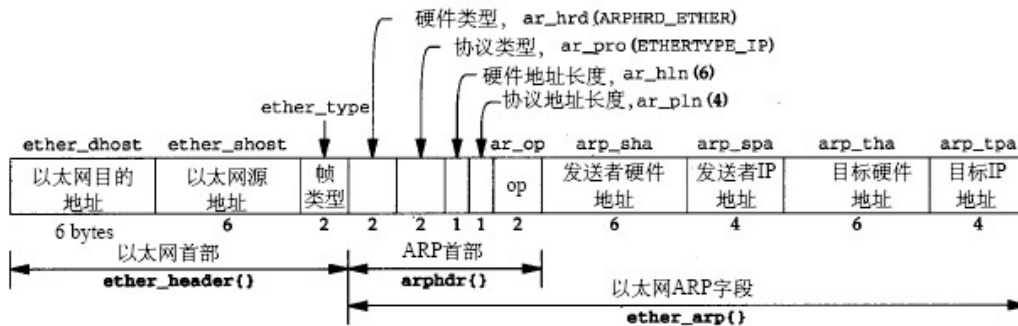


图21-7 在以太网上使用时ARP请求或回答的格式

sniffer 抓包显示:

Captured Data.

- Ethernet II
 - Destination: FF-FF-FF-FF-FF-FF
 - Source: 00-50-56-FE-D1-52
 - Ethernet Type: Address Resolution Protocol (0x0806)
- Address Resolution Protocol
 - Hardware Type: Ethernet (1)
 - Protocol Type: IPV4 (0x0800)
 - Hardware Size: 6
 - Protocol Size: 4
 - Opcode: Request (1)
 - Sender MAC Address: 00-50-56-FE-D1-52
 - Sender IP Address: 172.16.169.2
 - Target MAC Address: 00-00-00-00-00-00
 - Target IP Address: 172.16.169.128

```
FF FF FF FF FF FF 00 50 .....P
56 FE D1 52 08 06 00 01 V..R....
08 00 06 04 00 01 00 50 .....P
56 FE D1 52 AC 10 A9 02 V..R....
00 00 00 00 00 00 AC 10 .....
A9 80 ..
```

字节流

树型分析结果

将我们抓包分析的结果与 arp 报文结构对比, 发现解析正确。

下面给出以太帧和 arp 报文解析代码, 由于代码相对比较冗长, 先给出一个全局性的介绍。由 libpcap 进行抓包, 将抓到的数据存储在 tmpsnifferdata.strdata 中, 通过调用 tmpsnifferdata.strdata.data() 便可以得到原始的比特流数据, 由于以太帧是报文最开始部分, 所以以太帧报头指针定位在 tmpsnifferdata.strdata.data() 首部即可, 然后在以太帧报头中可以分析出上一层协议是 IP 协议或者是 arp 协议, 并且以太帧报头长度固定为 14 字节, 故而将 arp 报头指针以及 IP 报头指针定位在 tmpsnifferdata.strdata.data()+14 即可 (指针定位的代码加了下划线)。只要定位出了报头所在位置, 按照报文格式便可以解析出所有的数据。在解析过程中比较重要的是要注意数端的转换。接下来所有的分析都是基于这一个理念。相关代码全部在 multiview.cpp, multiview.h, capturethread.cpp, capturethread.h 中, 可以参看源代码具体理解。

下面给出我们以太帧和 arp 报头定位代码:

```
tmpSnifferData.strData=rawByteData;
eth = ( eth_header*) tmpSnifferData.strData.data();
```

```
tmpSnifferData.protoInfo.peth = (void*) eth;
arph = (_arp_header*) (tmpSnifferData.strData.data()+14);
```

以上便是 arp 报文解析的定位过程，在分析接下来的几个协议时，也只展示相关抓包结果以及指针定位代码对解析细节不展示。

3.1.2 IP 报文解析



IP 报文结构：

sniffer 抓包分析结果：

Captured Data:

- ▼ Ethernet II
 - Destination: 01-00-5E-00-00-FB
 - Source: 00-0C-29-54-A1-61
 - Ethernet Type: IPV4 (0x0800)
- ▼ Internet Protocol
 - Version: 4
 - Header Length: 20
 - Total Length: 73
 - Identification: 0x4a96 19094
 - Flags
 - Fragment Offset: 0
 - Time to Live: 255
 - Protocol: UDP
 - Source: 172.16.169.128
 - Destination: 224.0.0.251
 - Option Data: No Option
 - User Datagram Protocol
 - TCP payload

```
01 00 5E 00 00 FB 00 0C  ..^.....
29 54 A1 61 08 00 45 00  )T.a..E.
00 49 4A 96 40 00 FF 11  .IJ.@...
FA 80 AC 10 A9 80 E0 00  .....
00 FB 14 E9 14 E9 00 35  .....5
9D AA 00 00 00 00 00 02  .....
00 00 00 00 00 00 05 5F  ....._
69 70 70 73 04 5F 74 63  ipps._tc
70 05 6C 6F 63 61 6C 00  p.local.
00 0C 00 01 04 5F 69 70  ...._ip
70 C0 12 00 0C 00 01    p.....
```

下面是 ip 指针定位代码：

```
iph = (_ip_header*) (tmpSnifferData.strData.data()+14);
```

3.1.3 ICMP 报文

icmp 报文结构：



sniffer 抓包分析：

Captured Data:

- ▶ Ethernet II
- ▼ Internet Protocol
 - Version: 4
 - Header Length: 20
 - Total Length: 48
 - Identification: 0xb53c 46396
 - ▶ Flags
 - Fragment Offset: 0
 - Time to Live: 64
 - Protocol: ICMP
 - Source: 172.16.169.128
 - Destination: 172.16.169.254
 - Option Data: No Option
- ▼ Internet Control Message Protocol
 - Type: 0 (Echo (ping) reply)
 - Code: 0
 - Checksum: 0xecb5
- ▶ TCP payload

```

00 50 56 EE 23 60 00 0C  .PV.#`..
29 54 A1 61 08 00 45 00  )T.a..E.
00 30 B5 3C 00 00 40 01  .0.<..@.
19 F1 AC 10 A9 80 AC 10  ....
A9 FE 00 00 EC B5 36 2F  ....6/
00 00 84 03 00 00 00 00  ....
00 00 00 00 00 00 00 00  ....
00 00 84 E1 D4 35      ....5

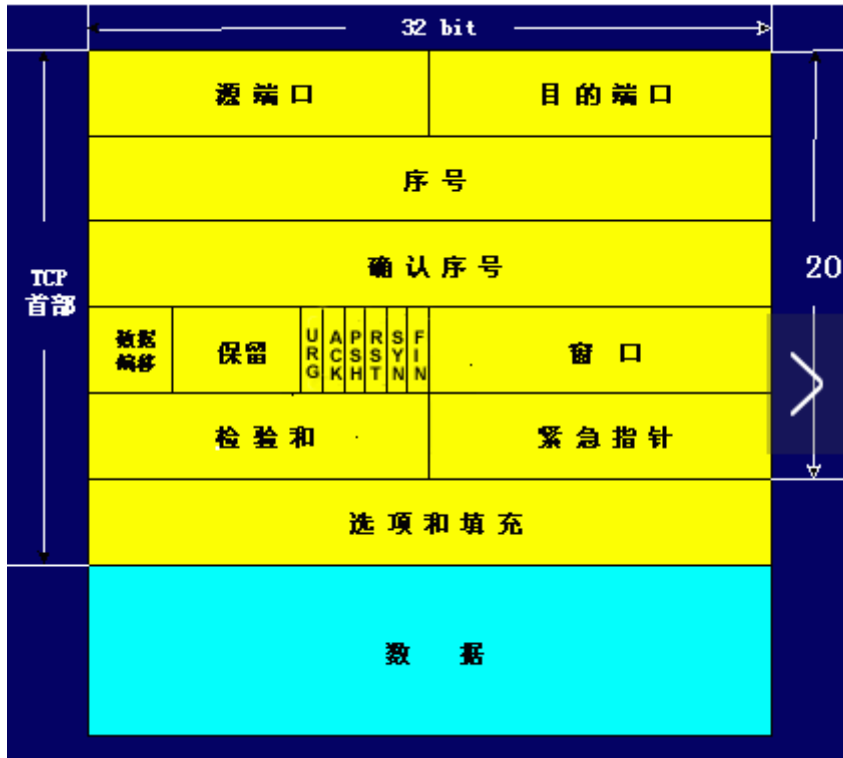
```

icmp 报头定位代码：

```
icmp_h = (_icmp_header*) (tmpSnifferData.strData.data()+14+ip_lenth);
```


3.1.4 TCP 报文

tcp 报文结构:



sniffer 抓包分析:

Transmission Control Protocol

Source Port: 52618
Destination Port: 80
Sequence Number: 333939616
Acknowledgment Number: 0
Header Length: 40

Flags

Reserved: 0
Nonce: 0
CWR: 0
ECN-Echo: 0
Urgent: 0
ACK: 0
Push: 0
Reset: 0
Syn: 1
Fin: 0
Win Size: 29200
Checksum: 0x4c56

Urgent Pointer: 0
Option: Maximum Segment Size

左边及以上为 tcp 树型解析结果

```

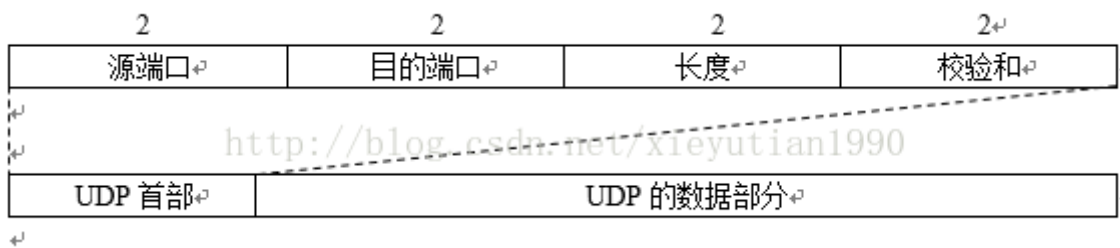
00 50 56 FE D1 52 00 0C .PV..R..
29 54 A1 61 08 00 45 00 )Ta..E.
00 3C B6 6E 40 00 40 06 <.n@.@.
95 E4 AC 10 A9 80 DE C0 .....
BA 17 CD 8A 00 50 13 E7 .....P..
83 A0 00 00 00 00 A0 02 .....
72 10 4C 56 00 00 02 04 r.LV...
05 B4 04 02 08 0A FF FF .....
35 CE 00 00 00 00 01 03 5.....
03 07 ..
                    
```

tcp 报头定位代码:

```
tcp_h = (_tcp_header*) (tmpSnifferData.strData.data()+14+ip_lenth);
```

3.1.5 UDP 报文

UDP 报文结构：



sniffer 抓包分析：

Captured Data:

- ▶ Ethernet II
- ▼ Internet Protocol
 - Version: 4
 - Header Length: 20
 - Total Length: 73
 - Identification: 0x54bf 21695
 - ▶ Flags
 - Fragment Offset: 0
 - Time to Live: 255
 - Protocol: UDP
 - Source: 172.16.169.128
 - Destination: 224.0.0.251
 - Option Data: No Option
- ▼ User Datagram Protocol
 - Source Port: 5353
 - Destination Port: 5353
 - Length: 53
 - Checksum: 0x9daa

```

01 00 5E 00 00 FB 00 0C  ..^.....
29 54 A1 61 08 00 45 00  )T.a..E.
00 49 54 BF 40 00 FF 11  .IT.@...
F0 57 AC 10 A9 80 E0 00  .W.....
00 FB 14 E9 14 E9 00 35  .....5
9D AA 00 00 00 00 00 02  .....
00 00 00 00 00 00 05 5F  ....._
69 70 70 73 04 5F 74 63  ipps._tc
70 05 6C 6F 63 61 6C 00  p.local.
00 0C 00 01 04 5F 69 70  ...._ip
70 C0 12 00 0C 00 01    p.....
          
```

udp 报头定位代码：

```
udph = (_udp_header*) (tmpSnifferData.strData.data()+14+ip_lenth);
```

3.1.6 IGMP 报文

IGMP 报文格式：

IGMP报文类型Type (8bit)		最大响应时间 Max_Resp_Code(8bit)		校验和Checksum(16bit)	
位组地址Group Address(32bit)					
Resv(4bit)	S(1bit)	QRV(3bit)	查询间隔时间QQIC(8bit)	源个数Number_of_Source(n,16bit)	
源地址Source address[i] (i取值从1到n)					

IGMPv3查询报文格式

sniffer 抓包解析：

Captured Data:

- ▶ Ethernet II
- ▼ Internet Protocol
 - Version: 4
 - Header Length: 24
 - Total Length: 40
 - Identification: 0x0 0
 - ▶ Flags
 - Fragment Offset: 0
 - Time to Live: 1
 - Protocol: IGMP
 - Source: 172.16.169.128
 - Destination: 224.0.0.22
 - Option Data: Router Alert
- ▼ Internet Group Management Protocol
 - Type: Membership Report (0x22)
- ▶ TCP payload

```

01 00 5E 00 00 16 00 0C  ..^.....
29 54 A1 61 08 00 46 C0  )T.a..F.
00 28 00 00 40 00 01 02  .{..@...
AE 68 AC 10 A9 80 E0 00  .h.....
00 16 94 04 00 00 22 00  .....".
F9 02 00 00 00 01 04 00  .....
00 00 E0 00 00 FB       .....

```

由于上面的报文格式给出的是查询报文的格式，且作业要求中的显示字段对应的是查询报文，类型代码为 0x11，而抓到的包为 0x22，格式不同，故而未进行分析。

IGMP 报头定位代码：

```
igmp_h = (_igmp_header*) (tmpSnifferData.strData.data()+14+ip_lenth);
```

3.2 IP 分片重组

互联网协议使网络互相通信。设计要迎合不同物理性质的网络; 它是独立于链路层使用的基础传输技术。具有不同硬件的网络通常会发生变化，不仅在传输速度，而且在最大传输单元 (MTU)。倘若一个 IP 包的长度大于 MTU，那么在允许分片的情况下，该包会被分为几个 IP 包来进行传输。考虑部分 IP 报头结构：

固定	标识	标志	片偏移
----	----	----	-----

上面这三个部分就是 IP 报头中所有和 IP 重组有关的部分了。标志位的第三位标志这他之后还有没有分片，记为 MF 位，片偏移标记这它在原来要发送的数据中处于什么位置，对一个 IP 报文拆分成的所有分片，具有相同的标识位。

先来展示我们的 sniffer 实现的 IP 重组效果，由于 TCP 和 UDP 的 IP 分片重组相对难以测试，所以我选择了 icmp 报文进行测试，由于 icmp 报文和 tcp 和 udp 报文一样，都是基于 IP 协议之上的同级协议，所以用 icmp 进行测试和用 tcp 进行测试是等价的。

这儿设置的分片重组的包会在协议字段会出现 (rebuild) 标识，如下。

利用 ping 指令中的 -s 指令设定 icmp 报文的长度，我在虚拟机上对宿主机发送 ping xx.x.x.x -s 4000 进行测试，下面是测试结果：

172.16.169.128:0	10.162.217.171:0	(Rebuild)ICMP	4042
10.162.217.171:0	172.16.169.128:0	(Rebuild)ICMP	4042

▶ Ethernet II
▼ Internet Protocol
Version: 4
Header Length: 20
Total Length: 4028
Identification: 0xf59e 62878
▶ Flags
Fragment Offset: 0
Time to Live: 64
Protocol: ICMP
Source: 172.16.169.128
Destination: 10.162.217.171
Option Data: No Option
▼ Internet Control Message Protocol
Type: 8 (Echo (ping) request)
Code: 0
Checksum: 0x1a38
▶ TCP payload

```

00 50 56 FE D1 52 00 0C .PV..R..
29 54 A1 61 08 00 45 00 )T.a..E.
0F BC F5 9E 00 00 40 01 .....@.
45 E2 AC 10 A9 80 0A A2 E.....
D9 AB 08 00 1A 38 A9 C9 ....8..
00 01 23 82 2F 5A 00 00 ..#./Z..
00 00 06 8C 06 00 00 00 .....
00 00 10 11 12 13 14 15 .....
16 17 18 19 1A 1B 1C 1D .....
1E 1F 20 21 22 23 24 25 ...!"#$%
26 27 28 29 2A 2B 2C 2D &'()*+,-
2E 2F 30 31 32 33 34 35 ./012345
36 37 38 39 3A 3B 3C 3D 6789;<=
3E 3F 40 41 42 43 44 45 >?@ABCDE
46 47 48 49 4A 4B 4C 4D FGHIJKLM
4E 4F 50 51 52 53 54 55 NOPQRSTU
56 57 58 59 5A 5B 5C 5D VWXYZ[\]
5E 5F 60 61 62 63 64 65 ^_`abcde
66 67 68 69 6A 6B 6C 6D fghijklm
6E 6F 70 71 72 73 74 75 nopqrstu

```

部分字节

接下来详述操作过程。相关代码主要在 silidinfo.h 以及 slideingo.cpp 中，若想详细研究可以参考源代码。

下面给出的代码并是精简过的源码（一些语句改用自然语言描述），旨在方便理解。

3.2.1 判断是否为分片

对于一个 ip 分片而言，它的 offset 和 MF 不可能全为 0。

```
if(!(tmpSlidePacketInfo.fragmentFlag==1||tmpSlidePacketInfo.fragmentOffset!=0))
{
    return false;//不是一个分片的包
} else {
    insertPacket(tmpSlidePacketInfo);
    rebuildInfo();
    return true;
}
```

3.2.2 存入 vector

假设现在已经得到一个 IP 分片，在将它存储进 vector 之前，我们要判断它是否重复（利用 vector 的自带方法）：

```
std::vector<SlidePacketInfo>::const_iterator its;
its=find(packetInfoForRebuild.begin(),packetInfoForRebuild.end(),tmpslide);
if(its==packetInfoForRebuild.end()) {
    //it is a new packet, push it into vector
}
```

将包塞入缓冲区之后，考虑这个包的 identification 是否曾出现，若不曾出现，则将其存入另一个保存 identification 的 vector：

```
std::vector<int>::iterator it;
it=find(allIpId.begin(),allIpId.end(),tmpslide.fragmentIdentification);
if(it==allIpId.end()) {
    //the identification never appear before, push into vector
}
```

3.2.3 重组

每次抓到一个新的 IP 分片之后，测试是否能组成一个完整的 IP 报文。由于下一个分片的 offset 等于前一个分片的 offset+数据长度/8，所以可以先对同一个 identification 的所有分片按照 offset 进行排序，并判断是否前者的 offset+数据长度/8=后者的 offset。同时设置两个 flag 位，分别标记是否捕获到（MF==0，Offset!=0），（MF==1，offset==0）的分片，若三个条件全部满足，则确认可以组成一个完整的报文，并将这些包在缓冲区删除，并将这个包的 identification 在 identification 的 vector 中删除。代码如下：

```
std::sort(sequence.begin(),sequence.end(),LessSort);
std::vector<slideSort>::iterator it2=sequence.begin()
for(it2;(it2+1)!=sequence.end())&&it2!=sequence.end();++it2) {
    if(tailFlag&&headFlag) {
        if(当前包的 nextoffset==下一个包的 offset) {
            complete=true;
        } else {
```

```

        complete=false;
        break;
    }
} else{
    complete=false;
    break;
}
}

```

已经判断出可以重组，接下来将各个分片的数据拼接起来即可：

```

for(std::vector<slideSort>::iterator it2=(sequence.begin());it2!
=(sequence.end());++it2) {

    rebuildByteData.append(packetInfoForRebuild.at(it2->index).fragmentByteData)

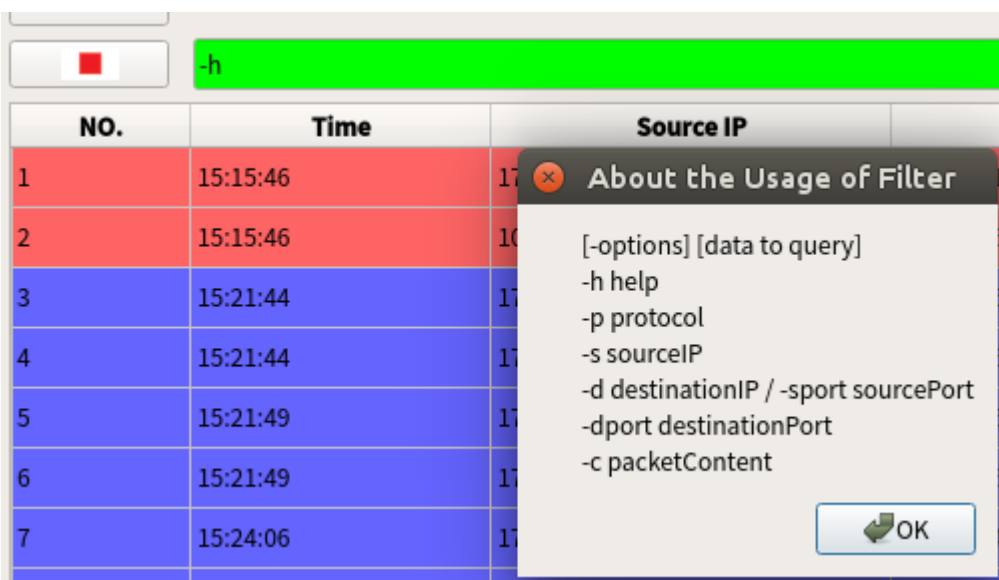
}

```

如此，便完成了 IP 重组。

3.3 包过滤

使用命令行格式的用户输入。设置了-h 选项进入帮助对话框，提示使用规则：



为了最大化筛选的灵活性，最终的版本没有使用 libpcap 自带的包过滤功能，而是使用自定义的索引进行包的过滤和筛选。支持的筛选索引有源地址、源端口、目的地址、目的端口、协议类型以及数据部分关键字（有关功能 3.4）。

用户也可以输入多个筛选条件，筛选时以逻辑“与”关系连接。进一步地，可以借助 tcpdump 的包过滤命令编译器实现更复杂的逻辑筛选。目前的功能足以完成比作业要求更复杂的筛选逻辑。

筛选的条件可以任意时间点输入。用户可以再启动抓包（点击开始按钮）前、抓包过程中以及抓包完成（点击停止按钮）后执行筛选，软件都可以返回正确的结果，并可以回复到完整的包列表。

-p UDP

NO.	Time	Source IP	Destination IP	Protocol
3	15:21:44	172.16.169.128:68	172.16.169.254:67	UDP
4	15:21:44	172.16.169.254:67	172.16.169.128:68	UDP
7	15:24:06	172.16.169.128:52855	91.189.94.4:123	UDP
10	15:24:06	91.189.94.4:123	172.16.169.128:52855	UDP
13	15:24:38	172.16.169.128:59570	91.189.94.4:123	UDP
14	15:24:39	91.189.94.4:123	172.16.169.128:59570	UDP
17	15:25:11	172.16.169.128:33907	91.189.94.4:123	UDP
18	15:25:11	91.189.94.4:123	172.16.169.128:33907	UDP
21	15:25:43	172.16.169.128:32860	91.189.94.4:123	UDP
22	15:25:44	91.189.94.4:123	172.16.169.128:32860	UDP
25	15:26:16	172.16.169.128:39464	91.189.94.4:123	UDP
26	15:26:16	91.189.94.4:123	172.16.169.128:39464	UDP

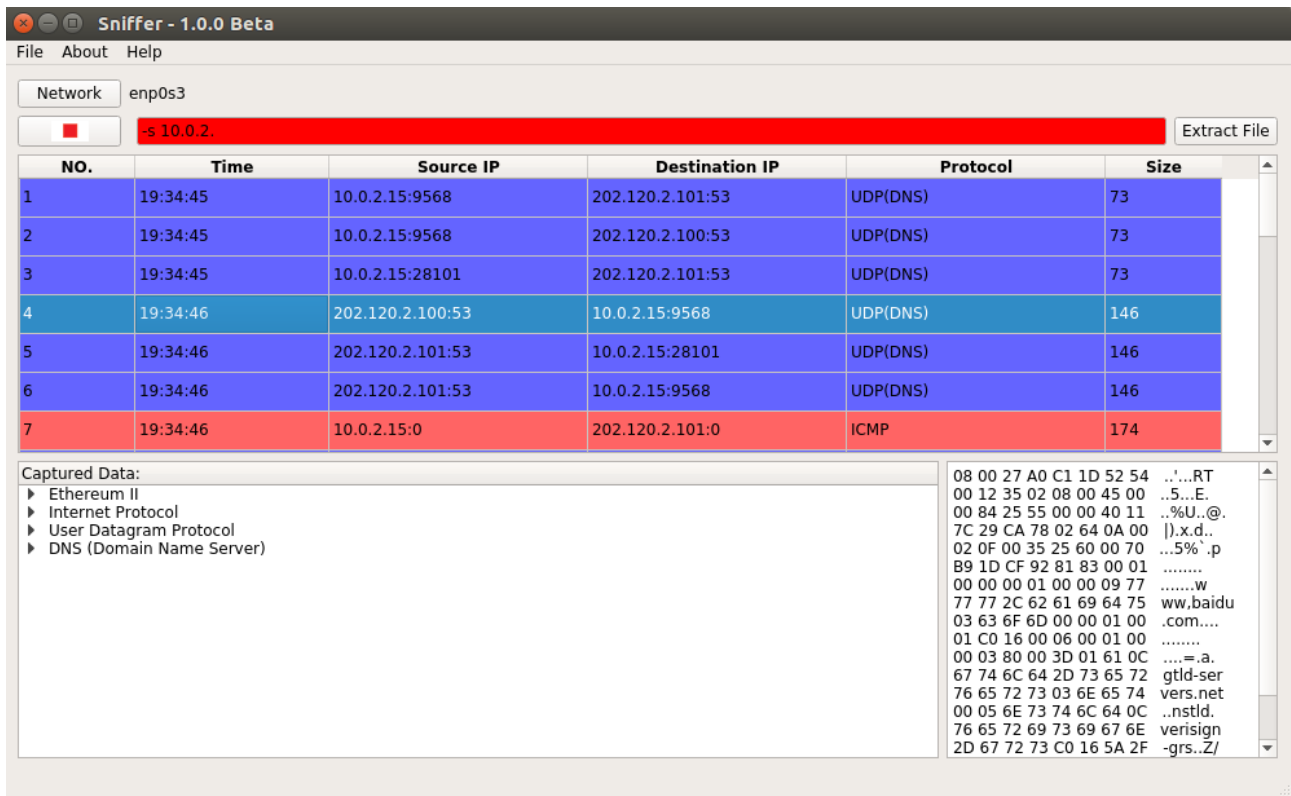
-s 172.16.169.128

NO.	Time	Source IP	Destination IP	Protocol
1	15:15:46	172.16.169.128:0	10.162.217.171:0	(Rebuild)ICMP
3	15:21:44	172.16.169.128:68	172.16.169.254:67	UDP
5	15:21:49	172.16.169.128:0	172.16.169.254:0	ARP
7	15:24:06	172.16.169.128:52855	91.189.94.4:123	UDP
9	15:24:06	172.16.169.128:0	172.16.169.2:0	ARP
11	15:24:11	172.16.169.128:0	172.16.169.2:0	ARP
13	15:24:38	172.16.169.128:59570	91.189.94.4:123	UDP
15	15:24:43	172.16.169.128:0	172.16.169.2:0	ARP
17	15:25:11	172.16.169.128:33907	91.189.94.4:123	UDP
19	15:25:16	172.16.169.128:0	172.16.169.2:0	ARP
21	15:25:43	172.16.169.128:32860	91.189.94.4:123	UDP
22	15:25:44	91.189.94.4:123	172.16.169.128:32860	UDP

-dport 67 -p UDP

NO.	Time	Source IP	Destination IP	Protocol
3	15:21:44	172.16.169.128:68	172.16.169.254:67	UDP
52	15:33:15	172.16.169.128:68	172.16.169.254:67	UDP
71	15:46:57	172.16.169.128:68	172.16.169.254:67	UDP
87	16:00:37	172.16.169.128:68	172.16.169.254:67	UDP

除此之外，模仿 wireshark 的过滤器表现形式，用户输入不符合格式时，输入框背景以红色作为提示；输入正确可以执行时以绿色背景提示。



Sniffer - 1.0.0 Beta

File About Help

Network: enp0s3

☐ -s 10.0.2. Extract File

NO.	Time	Source IP	Destination IP	Protocol	Size
1	19:34:45	10.0.2.15:9568	202.120.2.101:53	UDP(DNS)	73
2	19:34:45	10.0.2.15:9568	202.120.2.100:53	UDP(DNS)	73
3	19:34:45	10.0.2.15:28101	202.120.2.101:53	UDP(DNS)	73
4	19:34:46	202.120.2.100:53	10.0.2.15:9568	UDP(DNS)	146
5	19:34:46	202.120.2.101:53	10.0.2.15:28101	UDP(DNS)	146
6	19:34:46	202.120.2.101:53	10.0.2.15:9568	UDP(DNS)	146
7	19:34:46	10.0.2.15:0	202.120.2.101:0	ICMP	174

Captured Data:

- Ethernet II
- Internet Protocol
- User Datagram Protocol
- DNS (Domain Name Server)

08 00 27 A0 C1 1D 52 54 ...RT
 00 12 35 02 08 00 45 00 ...S...E.
 00 84 25 55 00 00 40 11 ...%U...@.
 7C 29 CA 78 02 64 0A 00 ...).x.d...
 02 0F 00 35 25 60 00 70 ...5%` .p
 B9 1D CF 92 81 83 00 01
 00 00 00 01 00 00 09 77w
 77 77 2C 62 61 69 64 75 ww.baidu
 03 63 6F 6D 00 00 01 00 .com....
 01 C0 16 00 06 00 01 00
 00 03 80 00 3D 01 61 0C=.a.
 67 74 6C 64 2D 73 65 72 gtld-ser
 76 65 72 73 03 6E 65 74 vers.net
 00 05 6E 73 74 6C 64 0C ..nstld.
 76 65 72 69 73 69 67 6E verisign
 2D 67 72 73 C0 16 5A 2F -grs..Z/

3.3.1 识别命令

利用正则表达式识别命令，具体代码如下：

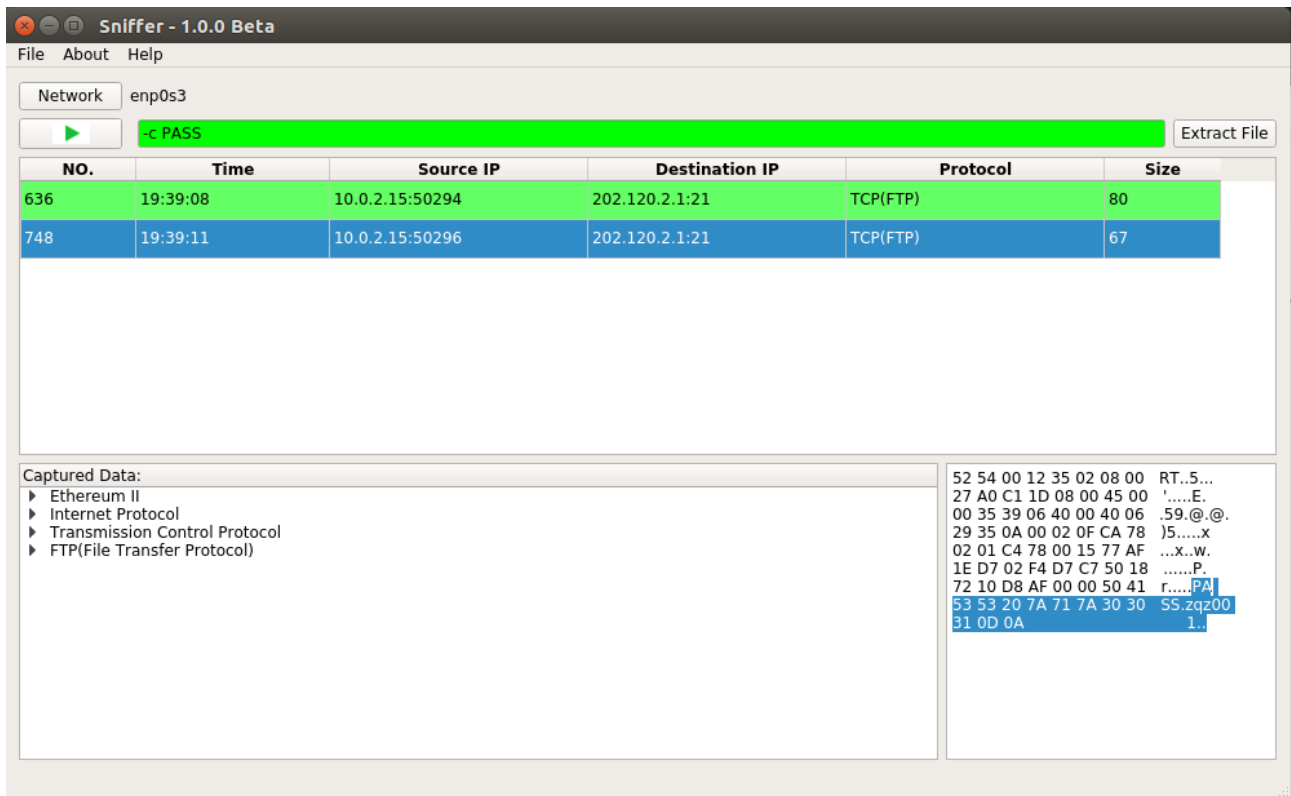
```
std::string pattern{ "(-h)|([ ]*((-p[ ]+[a-zA-Z]+)|((-s|-d)[ ]+\\d{1,3}.\\d{1,3}.\\d{1,3}.\\d{1,3}))|((-sport|-dport)[ ]+\\d+)|(-c[ ]\\S+))([ ]+)*((-p[ ]+[a-zA-Z]+)|((-s|-d)[ ]+\\d{1,3}.\\d{1,3}.\\d{1,3}.\\d{1,3}))|((-sport|-dport)[ ]+\\d+)|(-c[ ]\\S+))?" };

std::regex re(pattern);
return std::regex_match(command.toStdString(), re);
```

3.4 数据包查询

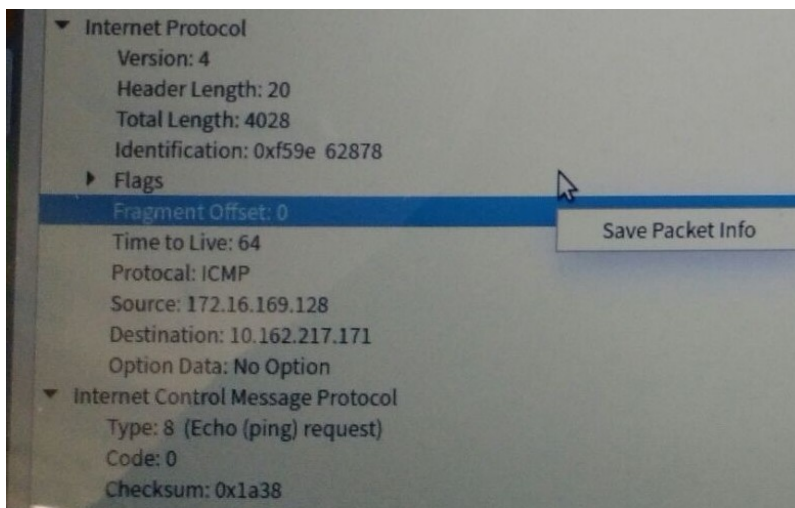
为简洁起见，这一功能被合并到包过滤功能中。在输入框内输入 -c keyword 即可查找数据部分含有 keyword 的数据包。

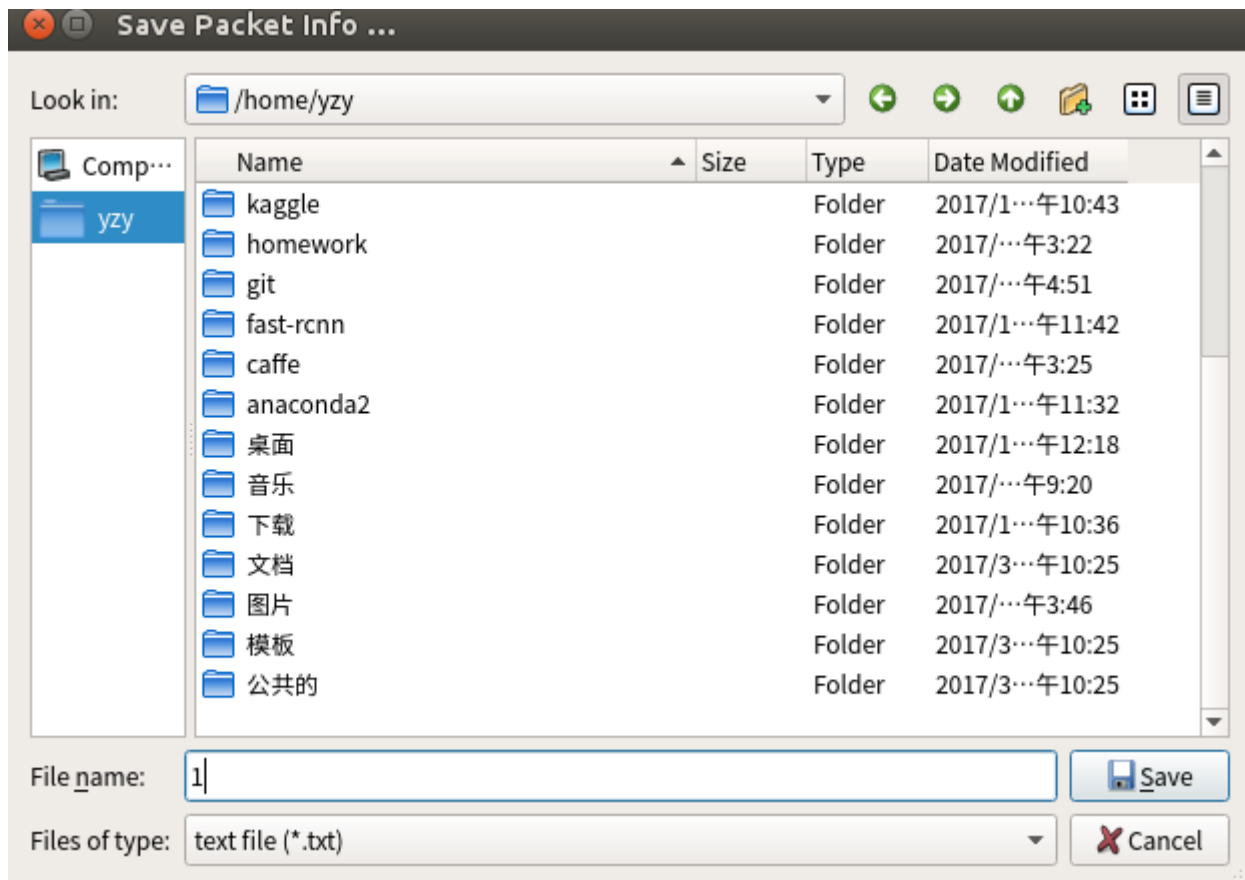
试验：使用密码登录 ftp://portal.sjtu.edu.cn 并抓包，输入 -c PASS 直接可以找到 FTP 请求密码以及传输密码的数据包。



3.5 数据包保存

数据包保存的话只需要在树型结构中右击，并 save packetinfo 即可。





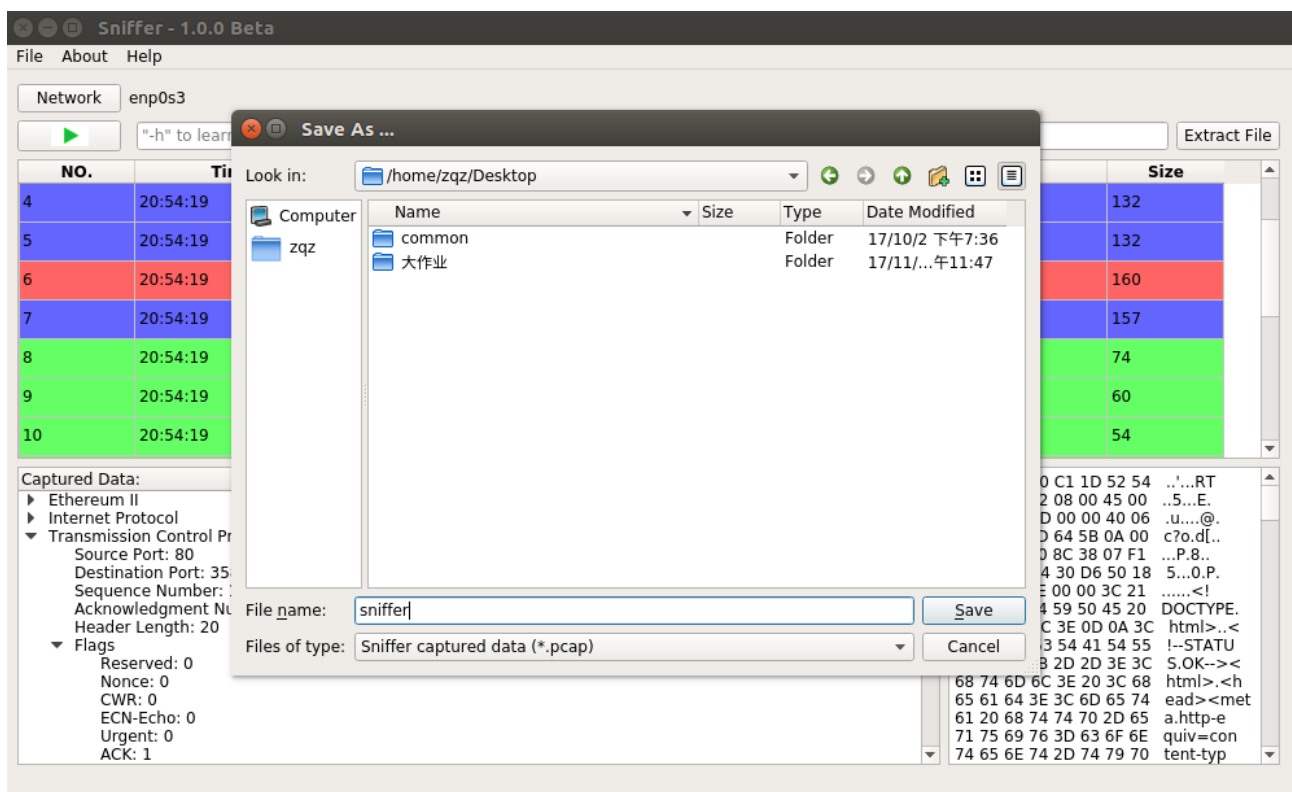
此为保存文件显示的结果。

```
1
Ethereum II
Destination: 31-00-00-00-D7-7F
Source: 00-00-29-54-A1-61
Ethernet Type: IPV4 (0x0800)
Internet Protocol
Version: 4
Header Length: 20
Total Length: 4028
Identification: 0xf59e 62878
Flags
Reserved Bit: 0
Don't Fragment: 0
More Fragment: 0
Fragment Offset: 0
Time to Live: 64
Protocal: ICMP
Source: 172.16.169.128
Destination: 10.162.217.171
Option Data: No Option
Internet Control Message Protocol
Type: 8 (Echo (ping) request)
Code: 0
Checksum: 0x1a38
TCP payload
```

具体实现是利用 Qfile 的函数将树型解析结果存入文件中，主要代码如下：

```
if (currentFile.isEmpty()) {  
    return false;  
}  
if (!QFile::copy(currentFile, saveFileName)) {  
    QMessageBox::warning(this, tr("Open As ..."), tr("<h3>ERROR Opening  
File</h3><p>A error occurs when opening the file.</p>"), QMessageBox::Ok);  
    return false;  
}  
return false;
```

另外也可以打开或保存 tcpdump 标准格式的 dumptfile。通过顶部菜单栏 File->open 和 File->Save 调用。



3.6 文件重组

3.6.1 概述

文件重组功能主要分为文件流监听、文件识别以及文件提取拼装三部分。这一部分功能集中在 FileDialog 中完成，抓包后在主页点击 Extract file 按钮进入。

3.6.2 文件识别提取

首先建立索引结构：

```
map<identifier, map<seq, packetID>> > fileFlow
```

说明：identifier = 源地址 | 源端口 | 目的地址 | 目的端口... (| 表示字符串拼接)用来唯一标示一个连接。seq 序列号，是实现文件恢复的重要参数，作为 key；packetID 是该包在包列表中的序号。

```
map<filename, vector<packetID>> files
```

说明：以文件名（文件标识）为 key，传输该文件的所有包序号为 value

提取文件伪代码如下：

```
//监听文件流
```

```
while (packet) {
```

```
    identifier = packet.源地址 | 源端口 | 目的地址 | 目的端口;
```

```
    //如果文件流已存在，添加一个包；反之添加文件流
```

```
    if ( fileFlow.find(identifier) ) fileFlow[identifier] .insert(packet.seq, packet.ID);
```

```
    else {
```

```
        fileFlow.insert(identifier, null);
```

```
        fileFlow[identifier] .insert(packet.seq, packet.ID);
```

```
    }
```

```
}
```

```
//过滤文件流
```

```
for each file in fileFlow {
```

```
    start_seq = min(file.seq); //以该文件流中最小序列号为起始序列号
```

```
    if (file[start_seq].data 不包含文件头部) continue; //留下很可能是文件的数据流
```

```
    files[filename].push_back(file[start_seq]);
```

```
    while True {
```

```
        next_seq = start_seq + 数据段长度 = start_seq + ipheader.total_length - ipheader.length -  
tcpheader.length    //关键规则：下一序列号为当前序列号加上数据段长度。数据段长度由以上运算得到  
较为准确。
```

```
        If (next_seq not in file) break; //找不到下一序列号，退出
```

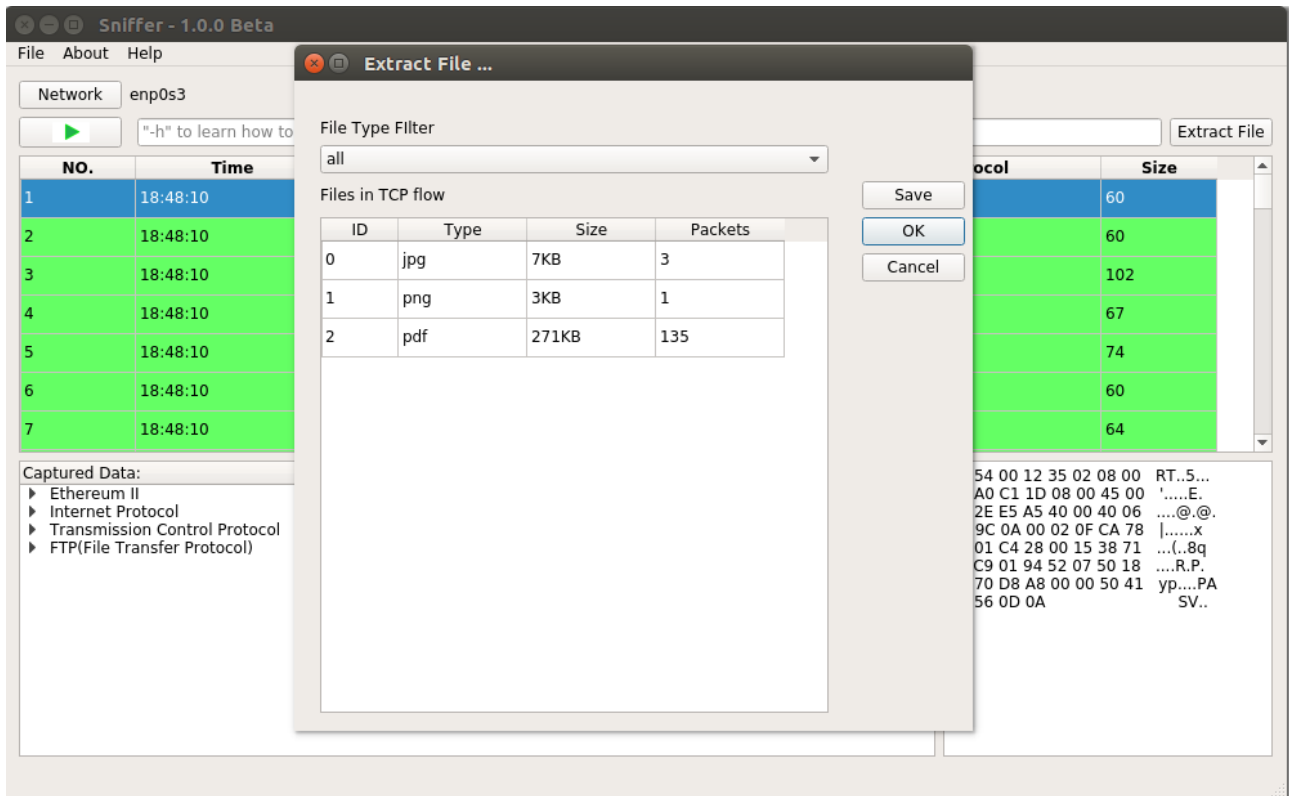
```
        files[filename].push_back(file[next_seq]);
```

```
        start_seq = next_seq
```

}

}

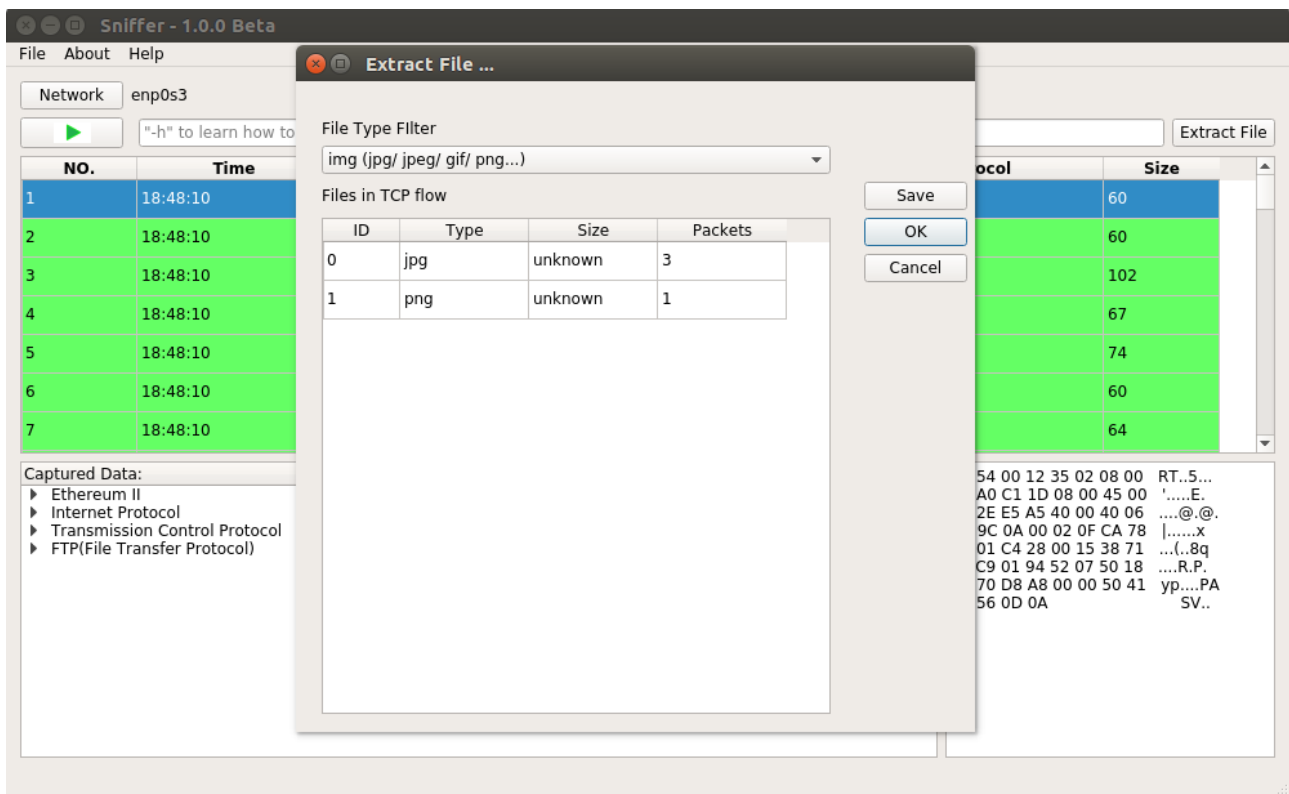
试验：使用 ftp/http 传输两张图片，一个 pdf 文档并抓包；点击 Extract filename



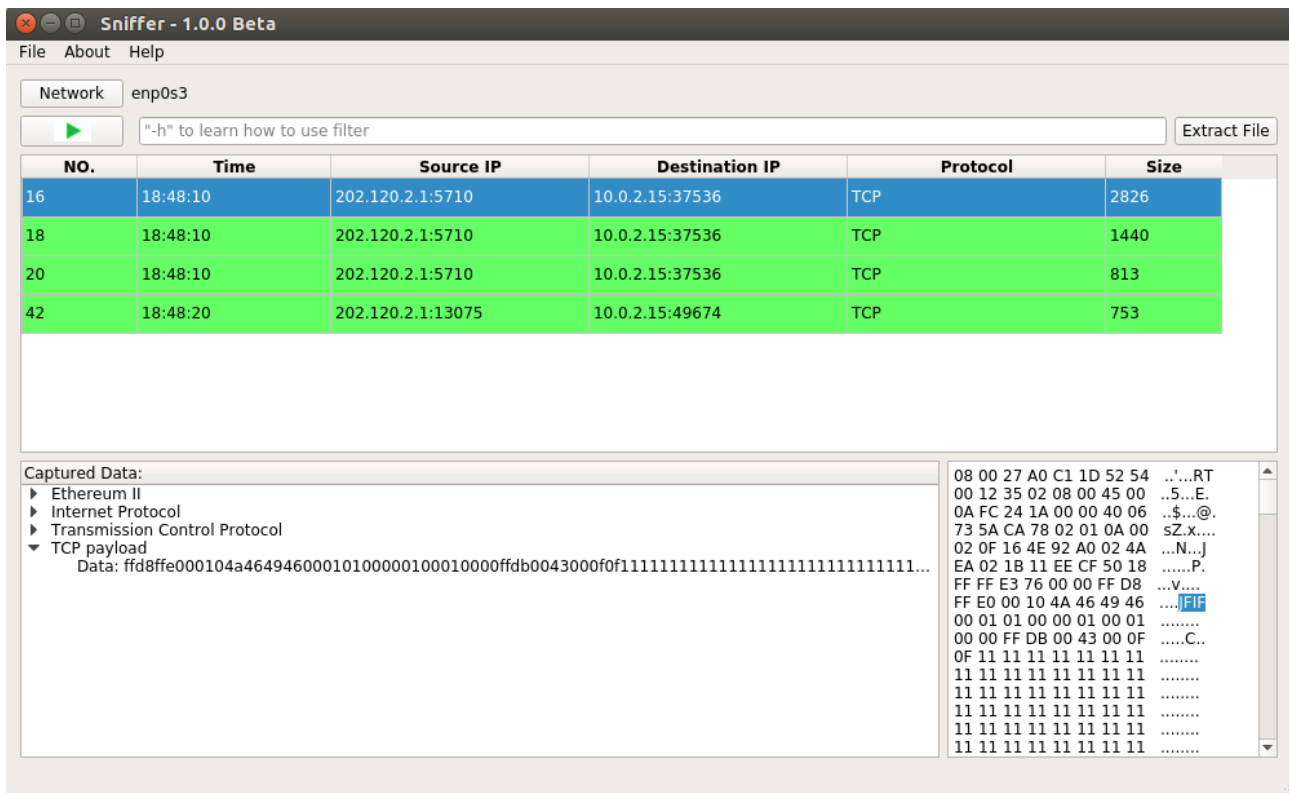
文件信息显示正常，其中最大的 pdf 文件由 135 个包组成。

3.6.3 找到传输该文件的包

在提取文件时已经维护了传输该文件所有包序号的列表，通过查找包列表，很容易显示传输该文件的所有包。



经过类型过滤（功能 3.7），选择了 jpg 和 png 图片文件，点击 OK

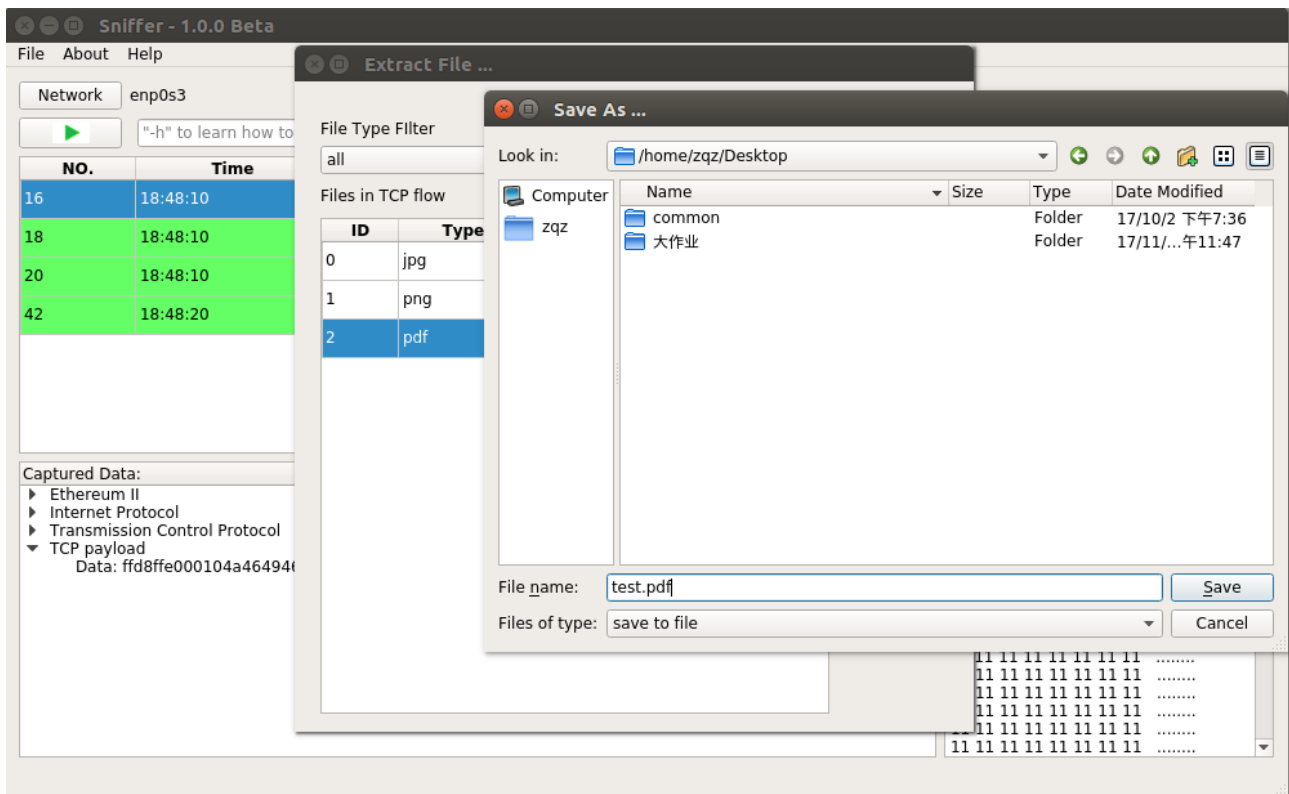


软件返回主页面并显示了这 4 个包。其中第 16 号包的数据部分有 jpeg 文件头标识 JFIF。也可以使用鼠标点选指定的一个文件，点击 save，等到传输该文件的数据包列表。

3.6.4 拼接文件

同样的，既然找到了传输该文件的所有包，那么按次序遍历这些包的数据段并写入同一文件，就可以恢复出原文件。

如图，回到之前的 Extract file 对话框，点选 pdf 文件，点击 save。

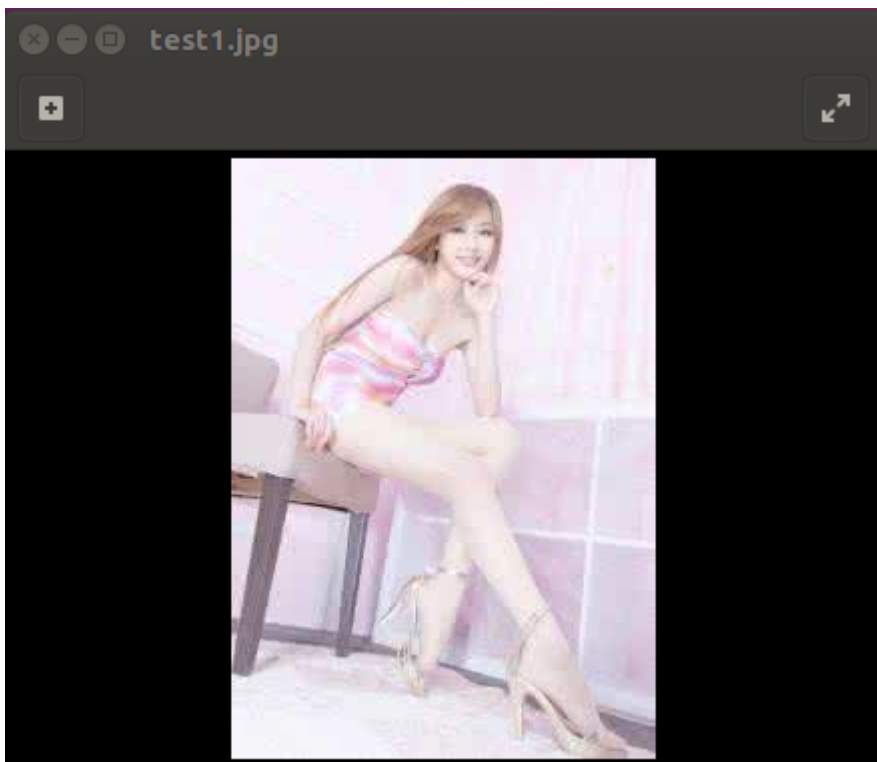
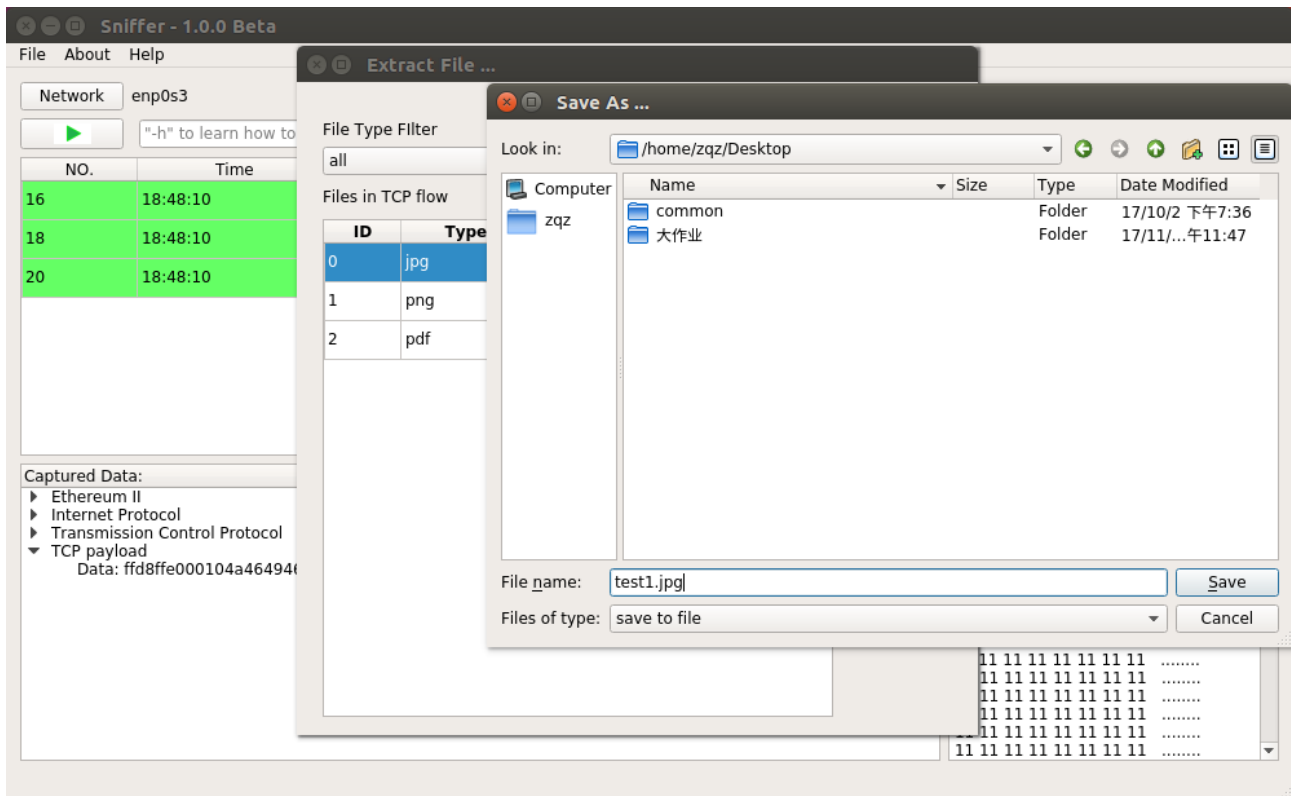


在弹出的文件保存对话框中，选择文件路径及文件名保存文件。查看该文件。



该文件是张晴钊的 pdf 个人简历。文件保存完整无错误。

再用同样的方法保存图片文件



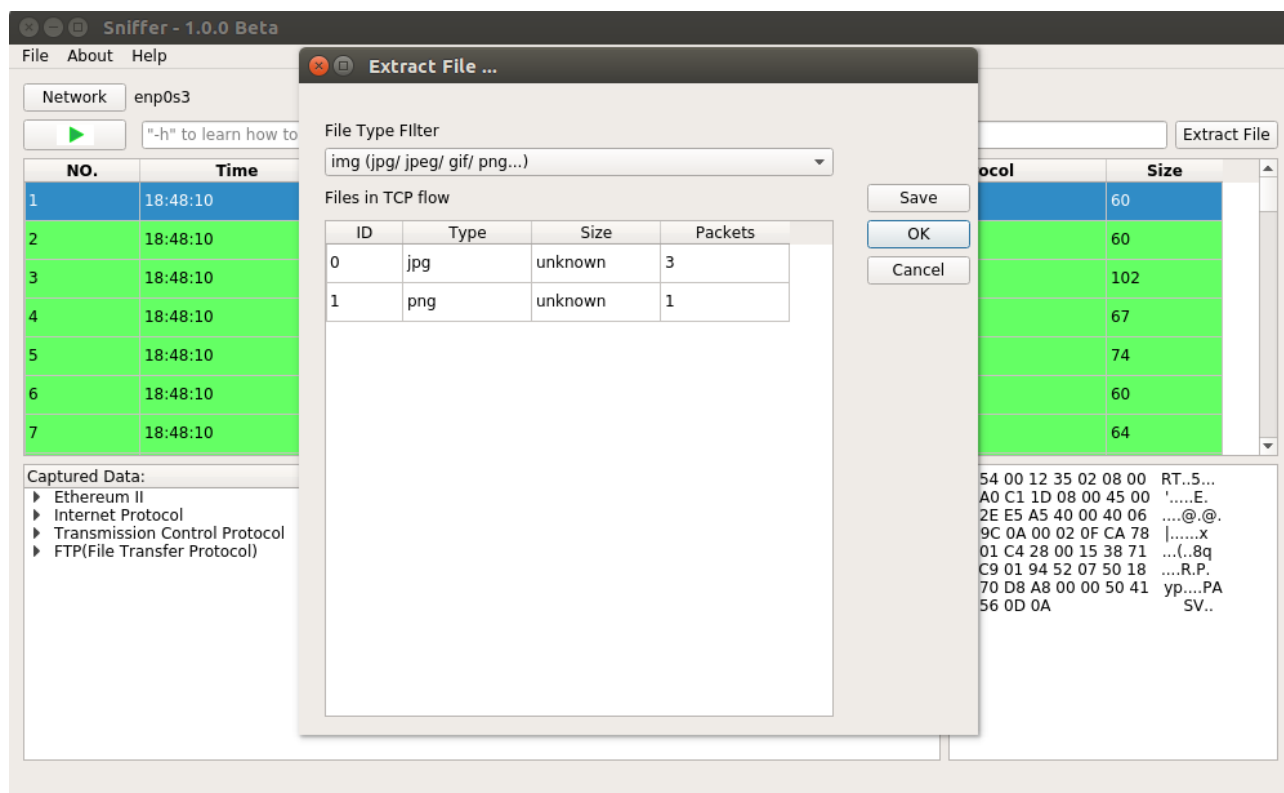
这张美女图片与原图片完全相同。不必在意图片的内容。

3.7 文件类型过滤

这一部分功能集成在功能 3.6 中。处理的策略是使用已知文件的文件头标识匹配文件流。

比如：jpg 文件头部 ffd8ff；avi 文件头部 41564920 等等，目前软件能够识别包含图片、视频、文本在内的十余种常见文件格式。包含其他文件格式或文件标识可以直接在代码中补充，扩展容易。

在之前抓到的三个文件列表中，选择复选框中的 img 选项，变过滤掉了 pdf 文件，留下了图片文件。



四、遇到的问题及解决方法

在 Sniffer 设计过程中，我们是利用了 qt 进行编程，刚开始时对 qt 中的各种数据结构都十分不熟悉。

前期不明白 qt 中各种 view 的区别，但起码网络上有一些代码可以参考，虽说网络上的代码比较杂乱，但是结合 qt 官方文档，还是比较顺利地完成了 GUI 界面的设计。

其后，开始对数据进行解析时，发现解析出来的数据和比特流并不相符。我们先分析在数据解析是主要用到的一些数据结构，即 qstring 和 qbytearray，这两种数据结构直接转换比较多，也比较多，我们并未能完全掌握。所以我们全部使用 qbytearray 一种数据结构进行操作，到最后才将 qstring 结合进来展示。然后我们同时利用 wireshark 和我们自己编写的 sniffer 进行抓包比对，刚解决完字节流问

题,发现虽然字节流正确,但分析出来的一些数据不准确。然后又参看了 github 上的一些代码,发现了 ntohs 以及 htons 等函数,才想起数端的问题,在综合考虑数据的位数和数端后,顺利分析出正确的结果。

五、体会与建议

这算是大学以来做过的工作量比较大的大作业了,刚开始准备写的时候觉得会比较困难,当然在写的时候也觉得比较困难。在项目进行的过程中,遇到了不少困难,在经历队友讨论,网络搜索,参看其他代码的过程后,最终也算是顺利解决了这些困难,在解决困难之后,在看着 sniffer 一步步得完善起来,心中还是充满了满足感与成就感。同时也在队友的交流与合作之中,学习了队友身上很多关于编程的优秀品质和习惯,自己也获得了很大的提升。

这次大作业,除却个人在代码方面的进步,也加深了对于这些数据协议的理解,更促进了在计算机网络这门课的学习。

六、主要数据结构

在此处,程序自带的数据结构便不再赘述,主要介绍自定义的一些数据结构。

6.1 GUI 界面的数据结构

6.1.1 MainWindow

MainWindow 类主要定义了主界面的相关操作。

下面给出 MainWindow 类定义:

```
namespace Ui {
    class MainWindow;
}
class Filter;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    //void on_pushButton_clicked();

    void on_start_clicked();
```

```
void on_stop_clicked();

void on_chooseNetButton_clicked();

void quit();

void save();

void open();

void about();

void saveTree();

void on_filter_textChanged(const QString &arg1);

void on_filter_returnPressed();
void showInfoInListView();

void on_tableView_clicked(const QModelIndex &index);

void on_treeView_customContextMenuRequested(const QPoint &pos);

void on_fileButton_clicked();

private:
    Ui::MainWindow *ui;
    Sniffer *sniffer;
    QString currentFile;
    NetworkChoice *netDevDialog;
    FileDialog *fileDialog;
    CaptureThread *captureThread;
    Filter *filter;
    bool snifferStatus; //true for running; false for stopped;

    bool saveFile(QString saveFileName);
    bool openFile(QString openFileName);
    bool changeFile(QString newFileName);
    CaptureThread *pCaptureThread;
    MultiView *view;
};
```

6.1.2 NetworkChoice

NetworkChoice 类定义和网卡选择界面相关的操作
下面给出 Networkchoice 类定义：

```
class NetworkChoice : public QDialog
{
    Q_OBJECT
```

```
public:
    explicit NetworkChoice(Sniffer *snifferObj, QWidget *parent = 0);
    ~NetworkChoice();

private slots:

    void on_choiceBox_activated(const QString &arg1);

    //void on_buttonBox_accepted();

    void on_buttonBox_clicked();

private:
    Ui::NetworkChoice *ui;
    void addNetDevs();
    void showCurrentNetInfo(const QString &netName);
    Sniffer *sniffer;
    std::vector<NetDevInfo>::iterator netIndex;
};
```

6.2 解析数据流使用的一些数据结构

该部分的代码都在 type.h 中

6.2.1 报文头部结构

这儿主要给出各种报文的头部的数据结构，和 2.1 中的报文结构想类似。

主要代码如下：

```
// Mac 头部 (14 字节)
typedef struct _eth_header
{
    unsigned char dstmac[6]; // 目标 mac 地址
    unsigned char srcmac[6]; // 来源 mac 地址
    unsigned short eth_type; // 以太网类型
}eth_header;

// ARP 头部 (28 字节)
typedef struct _arp_header
{
    unsigned short arp_hrd; // 硬件类型
```

```
unsigned short arp_pro;      // 协议类型
unsigned char arp_hln;      // 硬件地址长度
unsigned char arp_pln;      // 协议地址长度
unsigned short arp_op;      // ARP 操作类型
unsigned char arp_sha[6];   // 发送者的硬件地址
unsigned char arp_spa[4];    // 发送者的协议地址
unsigned char arp_tha[6];   // 目标的硬件地址
unsigned char arp_tpa[4];    // 目标的协议地址
}arp_header;

// IP 头部
typedef struct _ip_header
{
    unsigned char    ver_ihl;    // 版本 (4 bits) + 首部长度 (4 bits)
    unsigned char    tos;        // 服务类型(Type of service)
    unsigned short    tlen;       // 总长(Total length)
    unsigned short    identification; // 标识(Identification)
    unsigned short    flags_fo;   // 标志位(Flags) (3 bits) + 段偏移量(Fragment offset) (13 bits)
    unsigned char    ttl;        // 存活时间(Time to live)
    unsigned char    proto;       // 协议(Protocol)
    unsigned short    crc;        // 首部校验和(Header checksum)
    unsigned char    saddr[4];    // 源地址(Source address)
    unsigned char    daddr[4];    // 目标地址(Destination address)
    unsigned char    optionData;
}ip_header;

// TCP 头部
typedef struct _tcp_header
{
    unsigned short sport;          // 源端口号
    unsigned short dport;          // 目的端口号
    unsigned int  seq_no;          // 序列号
```

```
    unsigned int  ack_no;           // 确认号
    unsigned char thl;              // tcp 头部长度
    unsigned char flag;             // 12 位标志
    unsigned short wnd_size;        // 16 位窗口大小
    unsigned short chk_sum;         // 16 位 TCP 检验和
    unsigned short urgt_p;          // 16 为紧急指针
    unsigned char tcpOptionData;

}tcp_header;

// UDP 头部 (8 字节)
typedef struct _udp_header
{
    unsigned short sport;          // 源端口(Source port)
    unsigned short dport;          // 目的端口(Destination port)
    unsigned short len;            // UDP 数据包长度(Datagram length)
    unsigned short crc;            // 校验和(Checksum)
}udp_header;

//icmp 头部
typedef struct _icmp_header
{
    unsigned char  type;
    unsigned char  code;
    unsigned short  crc;

}icmp_header;

//igmp 头部
typedef struct _igmp_header
{
    unsigned char type;
    unsigned char maxRespCode;
    unsigned short crc;
```

```
    unsigned char  groupAddress[4]; //ip  
}igmp_header;
```

6.2.2 存储网络结构及包信息的数据结构

该部分主要给网卡信息展示，数据包保存，以及 IP 分片重组相关的几种数据结构。

代码如下：

```
// 网络设备信息结构
```

```
struct NetDevInfo
```

```
{  
    std::string strNetDevname;  
    std::string strNetDevDescribe;  
    std::string strIPv4FamilyName;  
    std::string strIPv4Addr;  
    std::string strIPv6FamilyName;  
    std::string strIPv6Addr;  
};
```

```
#include <QString>
```

```
// 树形显示结果的数据结构
```

```
struct AnalyseProtoType
```

```
{  
    uint ipFlag;  
    uint tcpFlag;  
    uint appFlag;  
    void* peth;  
    void* pip;  
    void* ptcp;  
    QString ipProto;  
    QByteArray strSendInfo;  
};
```



```
// 捕获的数据结构
struct SnifferData
{
    QString          strNum;          // 序号
    QString          strTime;        // 时间
    QString          strSIP;          // 来源 IP 地址, 格式 IP:port
    QString          strDIP;          // 目标 IP 地址, 格式 IP:port
    QString          strProto;        // 使用的协议
    QString          strProtoForShow;
    QString          strLength;       // 数据长度
    QByteArray       strData;         // 原始数据
    AnalyseProtoType protoInfo;       // 树形显示结果的数据结构
};

//存储分片信息
struct SlidePacketInfo
{
    int      fragmentIdentification;
    int      nextFragmentOffset;
    bool     fragmentFlag;
    unsigned short fragmentOffset;
    QByteArray fragmentByteData;
    QByteArray fragmentheader;
    void * header;
};

//存储每一个分片的相关信息, 用于排序
struct slideSort
{
    int index;
    int sortOffset;
};
```

6.3 实现后续功能时相关数据结构

6.3.1 CaptureThread

CaptureThread 继承自 QThread，主要负责在 GUI 进程中开启抓包线程。

具体定义如下：

```
class CaptureThread : public QThread
{
    Q_OBJECT
public:
    CaptureThread();
    CaptureThread(Sniffer *psniffer, QString filename, MultiView *view, Filter
*filter);
    void stop();
    void run();
    void setCondition();
    bool getCondition();
private:
    volatile bool bstop;
    Sniffer *sniffer;
    QString filename;
    MultiView *view;
    Filter *filter;
    //SlideInfo *pslideInfo;
    SlideInfo *pslideInfo;
};
```

6.3.2 Csniffer 与 Sniffer

Csniffer 类和 Sniffer 类是嗅探器类，Sniffer 继承自 Csniffer，主要负责抓包相关事项。

下面给出这 Sniffer 和 Csniffer 的定义：

```
class Csniffer {
protected:
    //friend class CaptureThread;
    struct pcap_pkthdr *header;
    const u_char *pktData;
    pcap_if_t *pNetDevs;
    pcap_t *pHandle;
    pcap_dumper_t *pDumpFile; //the file to save packet

    bool findAllNetDevs();
    int capture(); //change from bool to int

public:
```

```
Csniffer();
~Csniffer();
bool openDumpFile(const char* fileName);
bool saveCaptureData();
bool closeDumpFile();
bool freeNetDevs();
bool closeNetDevs();
bool openNetDev(char *devName, int flag=PCAP_OPENFLAG_PROMISCUOUS, int
lengthLimit = 65536);
bool setCaptureConfig(const char* config);
char err[PCAP_BUF_SIZE];

};

class Sniffer: public Csniffer {
public:
    friend class CaptureThread;
    QString currentNetName;
    Sniffer():Csniffer(){}
    ~Sniffer(){}

    bool getNetDevInfo();
    void testPrint();
    int captureOnce(); //change from bool to int
    bool openNetDevInSniffer();
    std::vector<NetDevInfo> netDevInfo; //provide available network interface
info
    std::vector<SnifferData> snifferData; //provide detail info of packets
    char* ip2s(struct sockaddr *sockaddr, int addlen, bool ipv6flag=true, char*
stripv6=NULL);
};
```

6.3.3 ListView 与 MultiView

ListView, MultiView 主要负责主窗口中三大块区域数据的传输与设置。

下面给出 ListView 与 MultiView 的定义：

```
class ListView
{
public:
    QTableView *view;
    ListView(QTableView *view);
    ~ListView();

    void rebuildInfo();
    bool isChanged();
    int getPacketsNum();
    void addPacketItem(SnifferData &data, bool fnew=true, bool display=true);
```

```
void loadByIndex(std::vector<int> &indexs);
void getOrderNumber(QModelIndex &index, QString &strNumber);

void clearData();
void addFilePacket(QString id, unsigned int seq, int index);
protected:
    friend class Filter;
    friend class FileDialog;
    QStandardItemModel *mainModel;
    int index;
    std::vector<SnifferData> packets;
    std::vector< QString > status;
    std::vector< std::map<unsigned int, int> > fileFlow;
};

class MultiView : public ListView
{
private:
    QStandardItemModel *treeModel;
    QTreeView *treeView;
    QTextBrowser *textBrowser;

    void reload();
    void setTreeViewByIndex(SnifferData SnifferData);
    void setHexViewByIndex(SnifferData SnifferData);
public:
    MultiView(QTreeView *tree, QTextBrowser* hex, QTableView
*list):ListView(list),treeView(tree),textBrowser(hex){
        reload();
    }
    ~MultiView();
    void packetInfoByIndex(QModelIndex index);    //call to update treeView $
hexView; responding signal list_item_clicked
    QList<QStandardItem*> returnTreeItems();
};
```

6.3.4 SlideInfo

SlideInfo 类主要负责 ip 分片重组。

下面给出 SlideInfo 类定义：

```
class SlideInfo
{
public:
    SlideInfo(int a=100);
    bool checkWhetherSlide(_ip_header*, SnifferData &, QByteArray &);
    bool complete;           // is get all packets to rebuild info
    QByteArray rebuildByteData;
    QByteArray rebuildheader;
    int rebuildTotalLength;
    void* preheader;
```

```
private:
    bool insertPacket(SlidePacketInfo & tmpslide);
    std::vector<SlidePacketInfo> packetInfoForRebuild;
    int defaultWindowSize;
    std::vector<int> allIpId; //save all packets' identifications which need to
be rebuilt
    std::vector<slideSort> sequence; //save the sequence of the same id
    bool headFlag;
    bool tailFlag;           //flags of the first fragment and the last
fragment

    bool isFull();
    bool rebuildInfo();
};
```

6.3.5 Filter

Filter 类负责过滤。

下面给出 Filter 类定义：

```
class Filter
{
public:
    Filter();
    ~Filter();
    bool checkCommand(QString command);
    bool loadCommand(QString command);
    void launchFilter(MultiView* view);
    void printQuery(); //test
    bool launchOneFilter(SnifferData &snifferData);
private:
    std::map<int, std::string> query;
    std::string findWord(std::string com, size_t pos);
};
```

6.3.6 FileDialog

FileDialog 类负责文件重组。

下面给出 FileDialog 类定义：

```
class FileDialog: public QDialog
{
    Q_OBJECT

public:
    explicit FileDialog(MultiView *v, QWidget *parent = 0);
    ~FileDialog();
```

```
void filtrateFile();
void prepare();
void displayFile();
std::vector<QString> choice;
QModelIndex targetFileIndex;

private slots:

    void on_buttonBox_accepted();

    void on_fileView_clicked(const QModelIndex &index);

    void on_fileButton_clicked();

    //void on_fileTypeBox_activated(int index);

    void on_fileTypeBox_activated(const QString &arg1);

private:
    Ui::FileDialog *ui;
    MultiView *view;
    QStandardItemModel *model;
    std::map< QString, std::vector<int> > files;
    void rebuild();
    void rebuildByType();
};
```
