网络嗅探器项目报告

515030910023 肖佳伟
515030910195 王林

一、 概述

(一) 运行环境

全部程序使用 python3.6 编写，在 python3.6 环境和依赖库安装齐全的情况下可在 Windows 和 Mac OS 系统中直接从源码运行 main.py。/bin 文件夹中提供了 Windows 和 Mac OS 环境下的可执行文件，无需安装 python3.6 环境和依赖库，可直接运行。

(二) 编译工具

可使用 python3.6 在命令行中运行 /source 文件夹中提供的 setup.py，将源码编译为可执行文件，编译前需要安装全部依赖库（Windows 可以不安装 py2app，Mac OS 可不安装 cx_Freeze），Windows 系统下使用命令行进入/source 目录输入 python3 setup.py build，编译后可执行文件储存于/build 文件夹下。Mac OS 系统下使用命令行进入/source 目录输入 python3 setup.py py2app，编译后可执行文件储存于/dist 文件夹下。

(三) 程序文件列表

```
/bin
|→/bitmaps          png form pictures for toolbar
|→/Sniffer          main code for program
|    |→/protocols                main code for program
|    |    |→/application              application layer protocols
|    |    |    |→/NotImplemented        unfinished protocol classes
|    |    |    |    |→application.py          application layer protocols
|    |    |    |    |→ftp.py                  ftp protocol class
|    |    |    |    |→http.py                 http protocol class
|    |    |    |    |→pop.py                  pop protocol class
|    |    |    |    |→rip.py                  rip protocol class
|    |    |    |    |→smtp.py                 smtp protocol class
|    |    |    |    |→snmp.py                 snmp protocol class
|    |    |    |    |→ssh.py                  ssh protocol class
|    |    |    |    |→telnet.py               telnet protocol class
|    |    |    |→__init__.py            base class for application layer
|    |    |→/internet                 internet layer protocols
|    |    |    |→/NotImplemented        unfinished protocol classes
|    |    |    |    |→esp.py                  esp class
|    |    |    |    |→icmp.py                 icmp protocol class
|    |    |    |    |→icmpv6.py               icmpv6 protocol class
|    |    |    |    |→igmp.py                 igmp protocol class
|    |    |    |→__init__.py            base class for internet layer
|    |    |    |→ah.py                  authentication header class
```

```
|   |   |   |->internet.py              internet layer protocols
|   |   |   |->ip.py                    basic internet protocol class
|   |   |   |->ipsec.py                 internet protocol security class
|   |   |   |->ipv4.py                  internet protocol version 4 class
|   |   |   |->ipv6.py                  internet protocol version 6 class
|   |   |   |->ipx.py                   internet packet exchange class
|   |   |->/link                    link layer protocols
|   |   |   |->__init__.py              base class for link layer
|   |   |   |->arp.py                   address resolution protocol class
|   |   |   |->ethernet.py              ethernet Protocol class
|   |   |   |->l2tp.py                  layer 2 tunneling protocol class
|   |   |   |->link.py                  link layer protocols
|   |   |   |->ospf.py                  open shortest path first header class
|   |   |   |->rarp.py                  reverse address resolution protocol class
|   |   |->/transport               transport layer protocols
|   |   |   |->/NotImplemented          unfinished protocol class
|   |   |   |   |->dccp.py                  dccp class
|   |   |   |   |->rsvp.py                  rsvp class
|   |   |   |   |->stcp.py                  stcp class
|   |   |   |->__init__.py              base class for transport layer
|   |   |   |->tcp.py                   transmission control protocol class
|   |   |   |->transport.py             transport layer protocols
|   |   |   |->udp.py                   user datagram protocol class
|   |   |->__init__.py              import classes
|   |   |->frame.py                 frame header class
|   |   |->header.py                global header class
|   |   |->protocol.py              basic protocol class
|   |   |->utilities.py             utility functions and classes
|   |->/reassembly              main code for program
|   |   |->/deprecated              description files
|   |   |   |->demo.py                  usage description
|   |   |   |->reassembly.txt           pseudo code
|   |   |->__init__.py              import classes
|   |   |->Info.py                  basic class
|   |   |->ip.py                    basic IP class
|   |   |->ipv4.py                  IPv4 reassembly class
|   |   |->ipv6.py                  IPv6 reassembly class
|   |   |->reassembly.py            basic reassembly class
|   |   |->TCP.py                   TCP reassembly class
|   |   |->utilities.py             utility functions and classes
|   |->__init__.py              basic UI class
|   |->FilterWindow.py          filter window class
|   |->PacketGrid.py            grid class to display packet summary
|   |->ReassemblyWindow.py      reassembly window class
```

| | |→SearchWindow.py search window class
|→GetIface.py function for obtaining name of iface
|→main.py starting program file
|→setup.py setup file
|→search_cat.ico icon for Windows system

二、 主要算法

(一) 报文重组

报文重组独立开发库 jspcap 是本项目的核心之一，其中实现了对 PCAP 文件的解析以及各种网络协议的解析，此外其还集成了 TCP 报文重组和 IP 分片重组的算法。其并未使用任何第三方库，也并未参照其他主流解析工具，如 dkpt 和 pyshark 等。

需要指出的是，在本项目中，仅使用到了该库的 jspcap/reassembly/ 部分，其余内容因客观原因并未采用。在 jspcap 库中，各文件主要可视为如下五个部分：

- ✧ 解析算法，即 Extraction —— 综合文件读写与协议解析，协调网络各层级的信息读取等（这一部分在本项目并未使用）
- ✧ 重组算法，即 Reassembly —— 基于 RFC 815 中所描述的算法，实现对 TCP 应用层报文的重组，以及 IP 包的分片重组
- ✧ 根协议，即 Protocol —— 抽象协议，包含协议类应有的常用函数和属性，并指定抽象方法等
- ✧ 协议族 —— 通过根协议派生，根据协议的具体结构实现的具体解析方案等
- ✧ 异常类 —— 异常处理，根据异常情况抛出并显示异常信息等

各部分内容的派生逻辑如下图所示：

1. 解析算法

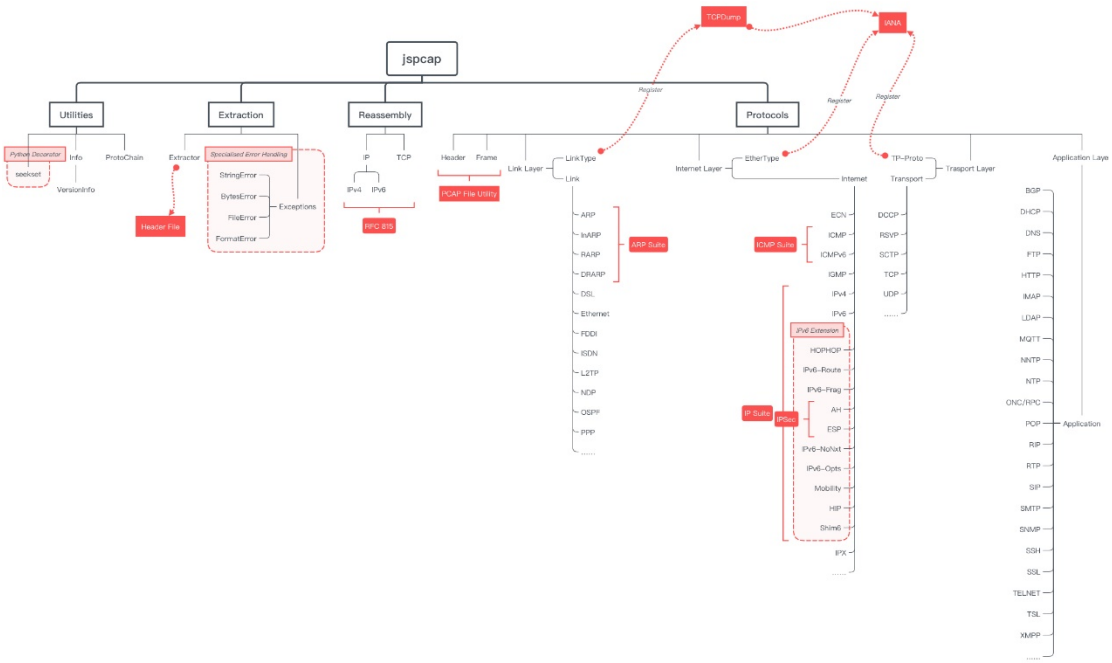


图 2-1 jspcap 架构

本部分采取了流式读取的策略，即逐段读入文件，减少对内存空间的占用，同时从某种程度上提高了解析效率。在实现上，以 PCAP 文件中的 Frame 为单位，通过在各层协议间传递 io.BytesIO 的形式进行流式读取。

头文件，特指在 jspcap/extractor.py 中实现的 Extractor 类。其负责处理文件名称的补全 —— 如输入文件名缺省后缀 .pcap，则需添加；如未指定输出类型，则通过输出文件名指明类型，若否则返回 FormatError 错误提示；如未指定输出文件名称，则默认为 out（见类方法 make_name）—— 以及根据 PCAP 文件的 Global Header 中指定的链路层协议类型，调取对 Frame 的解析操作，并返回解析结果；最终，记录必要信息，便于在如 UI 中交互调用，及后续协议拼接重组操作。

此外，Extractor 类支持自动解析，或迭代解析两种模式。以基于 jspcap 实现的命令行工具 jspcapy 的使用为例， verbose 模式下为迭代解析，从而可获取每一数据帧的信息；而自动模式则为自动解析，直接完成解析过程，并输出文件。

2. 重组算法

本部分参考了 RFC 791 和 RFC 815 中描述的两种算法。前者详细描述了 IP 包分片及重组的算法实现，其中使用到了 RCVBT，即"已接收比特哈希表"来维护接收顺序；而后者则针对上述 RCVBT 进行了优化，提出了一种替代算法。

为了便于算法的实现和使用，此处首先在 jspcap/reassembly/reassembly.py 中声明了一个名为 Reassembly 的抽象基类（Abstract Base Class），其效果等同于根协议。其中指定了一些抽象属性，如 name、count 和 datagram 等；一些抽象函数，如 reassembly 和 submit，分别用于重组过程和重组完成后提取结果；以及一些工具函数。

需要指出的是，同解析算法中的 Extractor 一样，此处也提供两种模式，可通过 run 方法完成多个数据包的重组；或通过直接调用，即 __call__ 方法，逐次输入进行重组。

A. IP 分片重组

由于操作内存占用较小，故 IP 包的分片重组直接采用了 RFC 791 的原始算法。其算法伪代码表示如下：

```
Notation:
    FO  - Fragment Offset
    IHL    - Internet Header Length
    MF  - More Fragments flag
    TTL    - Time To Live
    NFB    - Number of Fragment Blocks
    TL  - Total Length
    TDL    - Total Data Length
    BUFID - Buffer Identifier
    RCVBT - Fragment Received Bit Table
    TLB    - Timer Lower Bound

Procedure:
    DO {
        BUFID <- source|destination|protocol|identification;

        IF (FO = 0 AND MF = 0) {
            IF (buffer with BUFID is allocated) {
                flush all reassembly for this BUFID;
                Submit datagram to next step;
                DONE.
            }
```

```
        }

        IF (no buffer with BUFID is allocated) {
            allocate reassembly resources with BUFID;
            TIMER <- TLB;
            TDL <- 0;
            put data from fragment into data buffer with BUFID
                [from octet FO*8 to octet (TL-(IHL*4))+FO*8];
            set RCVBT bits [from FO to FO+((TL-(IHL*4)+7)/8)];
        }

        IF (MF = 0) {
            TDL <- TL-(IHL*4)+(FO*8)
        }

        IF (FO = 0) {
            put header in header buffer
        }

        IF (TDL # 0 AND RCVBT bits [from 0 to (TDL+7)/8] are set) {
            TL <- TDL+(IHL*4)
            Submit datagram to next step;
            free all reassembly resources for this BUFID;
            DONE.
        }

        TIMER <- MAX(TIMER,TTL);

    } give up until (next fragment or timer expires);

    timer expires: {
        flush all reassembly with this BUFID;
        DONE.
    }
```

其中，特别对算法中使用的报文数据缓冲区 buffer 的数据结构描述如下：

```
(dict) buffer --> memory buffer for reassembly
    |--> (tuple) BUFID : (dict)
    |       |--> ip.src     |
    |       |--> ip.dst     |
    |       |--> ip.id      |
    |       |--> ip.proto   |
    |                       |--> 'TDL' : (int) total data length
    |                       |--> RCVBT : (bytearray)
    |                       |              fragment received bit table
```

```
    |                       |               |--> (bytes)
    |                       |               |     b'\x00’
    |                       |               |       not received
    |                       |               |--> (bytes) b’\x01'
    |                       |               |       received
    |                       |               |--> (bytes) ...
    |                       |--> 'index' : (list) list of
    |                       |               reassembled packets
    |                       |               |--> (int)
    |                       |                       packet range number
    |                       |--> 'header' : (bytearray) header buffer
    |                       |--> 'datagram' : (bytearray) data buffer,
    |                                           holes set to b'\x00'
    |--> (tuple) BUFID ...
```

a. IPv4

针对上述算法，以下将大致解释 IPv4 分片重组的使用方法和符号意义。

```
>>> from reassembly import IPv4_Reassembly
# Initialise instance:
>>> ipv4_reassembly = IPv4_Reassembly()
# Call reassembly:
>>> packet_dict = dict(
...  bufid = tuple(
...      ipv4.src,        # source IP address
...      ipv4.dst,        # destination IP address
...      ipv4.id,          # identification
...      ipv4.proto,        # payload protocol type
...  ),
...  num = frame.number,    # original packet range number
...  fo = ipv4.frag_offset,  # fragment offset
...  ihl = ipv4.hdr_len,    # internet header length
...  mf = ipv4.flags.mf,    # more fragment flag
...  tl = ipv4.len,      # total length, header includes
...  header = ipv4.header,   # raw bytearray type header
...  payload = ipv4.payload, # raw bytearray type payload
... )
>>> ipv4_reassembly(packet_dict)
# Fetch result:
>>> result = ipv4_reassembly.datagram

(tuple) datagram
    |--> (Info) data
    |    |--> 'NotImplemented' : (bool) True --> implemented
    |    |--> 'index' : (tuple) packet numbers
    |    |                |--> (int) original packet range number
```

```
    |    |--> 'packet' : (bytes/None) reassembled IPv4 packet
    |--> (Info) data
    |    |--> 'NotImplemented' : (bool) False --> not implemented
    |    |--> 'index' : (tuple) packet numbers
    |    |                |--> (int) original packet range number
    |    |--> 'header' : (bytes/None) IPv4 header
    |    |--> 'payload' : (tuple/None)
    |                       partially reassembled IPv4 payload
    |                      |--> (bytes/None) IPv4 payload fragment
    |--> (Info) data ...
```

b. IPv6

针对上述算法，以下将大致解释 IPv6 分片重组的使用方法和符号意义。

```
>>> from reassembly import IPv6_Reassembly
# Initialise instance:
>>> ipv6_reassembly = IPv6_Reassembly()
# Call reassembly:
>>> packet_dict = dict(
...  bufid = tuple(
...      ipv6.src,         # source IP address
...      ipv6.dst,         # destination IP address
...      ipv6.label,         # label
...      ipv6_frag.next,     # next header field
...                          #   in IPv6 Fragment Header
...  ),
...  num = frame.number,     # original packet range number
...  fo = ipv6_frag.offset,  # fragment offset
...  ihl = ipv6.hdr_len,     # header length,
...                          #   only headers before IPv6-Frag
...  mf = ipv6_frag.mf,  # more fragment flag
...  tl = ipv6.len,      # total length, header includes
...  header = ipv6.header,   # raw bytearray type header
...                          #   before IPv6-Frag
...  payload = ipv6.payload, # raw bytearray type payload
...                          #   after IPv6-Frag
... )
>>> ipv6_reassembly(packet_dict)
# Fetch result:
>>> result = ipv6_reassembly.datagram

(tuple) datagram
    |--> (Info) data
    |      |--> 'NotImplemented' : (bool) True --> implemented
    |      |--> 'index' : (tuple) packet numbers
    |      |                |--> (int) original packet range number
```

```
    |      |--> 'packet' : (bytes/None) reassembled IPv6 packet
    |--> (Info) data
    |      |--> 'NotImplemented' : (bool) False --> not implemented
    |      |--> 'index' : (tuple) packet numbers
    |      |               |--> (int) original packet range number
    |      |--> 'header' : (bytes/None) IPv6 header
    |      |--> 'payload' : (tuple/None)
    |                      partially reassembled IPv6 payload
    |                      |--> (bytes/None) IPv6 payload fragment
    |--> (Info) data ...
```

B. TCP 报文重组

TCP 报文重组采用了 RFC 815 中描述的对 RCVBT 进行优化后的算法。其重组算法同上述 IP 分片重组，而“已接收比特哈希表”的替代算法过程描述如下：

① 从孔隙表中选取下一空隙。如孔隙表为空，则执行第八步操作。
② 如果片段首部（fragment.first）大于孔隙尾部（hole.last），则执行第一步操作。
③ 如果片段尾部（fragment.last）小于孔隙首部（hole.first），则执行第一步操作。
④ 从孔隙表中删除当前孔隙。
⑤ 如果片段首部（fragment.first）大于孔隙首部（hole.first），则创建新孔隙 new_hole，令其首部（new_hole.first）等于原孔隙首部（hole.first），其尾部等于片段首部（fragment.first）减一。
⑥ 如果片段尾部（fragment.last）大于孔隙尾部（hole.last），且更多片段标签（fragment.more_fragments）为真，则创建新孔隙 new_hole，令其首部（new_hole.first）等于片段尾部（fragment.last）加一，其尾部等于原孔隙尾部（hole.last）。
⑦ 执行第一步操作。
⑧ 如果孔隙表为空，则报文已重组完成。将其传递至高层协议处理机制。否则，返回。

其中，特别对算法中使用的报文数据缓冲区 buffer 的数据结构描述如下：

```
(dict) buffer --> memory buffer for reassembly
    |--> (tuple) BUFID : (dict)
    |    |--> ip.src    |
    |    |--> ip.dst    |
    |    |--> tcp.secport |
    |    |--> tcp.dstport |
    |                   |--> 'hdl' : (list) hole descriptor list
    |                   |             |--> (Info) hole -->
    |                   |                    hole descriptor
    |                   |                   |--> "first" -> (int)
    |                   |                         start of hole
    |                   |                   |--> "last" --> (int)
    |                   |                         stop of hole
    |                   |--> (int) ACK : (dict)
    |                   |             |--> 'ind' : (list) list of
    |                   |             |    reassembled packets
    |                   |             |         |--> (int)
```

```
    |                   |             |    packet range number
    |                   |             |--> 'isn' : (int) ISN of
    |                   |             |    payload buffer
    |                   |             |--> 'len' : (int) length
    |                   |             |    of payload buffer
    |                   |             |--> 'raw' : (bytearray)
    |                   |                    reassembled
    |                   |                    payload, holes set
    |                   |                    to b'\x00'
    |                   |--> (int) ACK ...
    |                   |--> ...
    |--> (tuple) BUFID ...
```

针对上述算法，以下将大致解释 TCP 报文重组的使用方法和符号意义。

```
>>> from reassembly import TCP_Reassembly
# Initialise instance:
>>> tcp_reassembly = TCP_Reassembly()
# Call reassembly:
>>> packet_dict = dict(
...  bufid = tuple(
...      ip.src,                  # source IP address
...      ip.dst,                  # destination IP address
...      tcp.srcport,            # source port
...      tcp.dstport,            # destination port
...  ),
...  num = frame.number,          # original packet range number
...  ack = tcp.ack,                # acknowledgement
...  dsn = tcp.seq,                # data sequence number
...  syn = tcp.flags.syn,        # synchronise flag
...  fin = tcp.flags.fin,        # finish flag
...  len = tcp.raw_len,            # payload length, header
excludes
...  first = tcp.seq,            # this sequence number
...  last = tcp.seq + tcp.raw_len,   # next (wanted) sequence
number
...  payload = tcp.raw,            # raw bytearray type payload
... )
>>> tcp_reassembly(packet_dict)
# Fetch result:
>>> result = tcp_reassembly.datagram

(tuple) datagram
    |--> (Info) data
    |    |--> 'NotImplemented' : (bool) True --> implemented
    |    |--> 'index' : (tuple) packet numbers
```

```
    |    |             |--> (int) original packet range number
    |    |--> 'payload' : (bytes/None)
    |    |               reassembled application layer data
    |--> (Info) data
    |    |--> 'NotImplemented' : (bool) False --> not implemented
    |    |--> 'index' : (tuple) packet numbers
    |    |             |--> (int) original packet range number
    |    |--> 'payload' : (tuple/None) partially reassembled
payload
    |                  |--> (bytes/None) payload fragment
    |--> (Info) data ...
```

3. 根协议

根协议，特指在 jspcap/protocols/protocol.py 中实现的 Protocol 类。其为一抽象基类（Abstract Base Class），定义了在协议族中需要用到一些通用方法，如 unpack、binary 和 read 等。此外，还指定了一些抽象属性，需要在协议族中重载，如 name、info 和 length 等。

需要指出的是，在 jspcap/protocols/utilities.py 文件中，定义了用于在不干预原有文件读取指针情况下，对文件进行操作的 seekset 函数，其通常以装饰器的形式进行使用。同时，定义了 ProtoChain 类。其用于保存当前协议的协议链，使得在协议层的传递过程中，得以清晰和便捷地保留并获取上层及下层协议信息。

此外，上述文件中，还定义了 Info 类，用于将字典参数（dict）转化为对象属性的类，便于在协议层中传递并读取和使用。

```
# 字典对象，及其访问
>>> dict_ = dict(
...  foo = 'foo_arg',
...  bar = 'bar_arg',
...  baz = 'baz_arg',
... )
>>> dict_
{'foo': 'foo_arg', 'bar': 'bar_arg', 'baz': 'baz_arg'}
>>> dict_['foo']
'foo_arg'


# Info 对象，及其访问
>>> info = Info(dict_)
>>> info
Info(foo='foo_arg', bar='bar_arg', baz='baz_arg')
>>> info.foo
'foo_arg'
```

4. 协议族

协议族，指包含 PCAP 文件特有的 Global Header 和 Frame Header 以及计算机网络 TCP/IP 四层架构在内的所有协议，在 jspcap/protocols/ 中实现。但由于能力和时间所限，

目前仅完成了链路层 Ethernet 等，网络层 IPv4 和 IPv6 等，及传输层 TCP 和 UDP 等的解析。

其中，PCAP 文件的 Global Header 在 jspcap/protocols/header.py 中的 Header 类实现，而 Frame Header 则在 jspcap/protocols/frame.py 中的 Frame 类实现。

通过根协议 Protocol，派生得到各层级的副根协议，即

- ✧ 链路层 —— jspcap/protocols/link/link.py 中的 Link 类
- ✧ 网络层 —— jspcap/protocols/internet/internet.py 中的 Internet 类
- ✧ 传输层 —— jspcap/protocols/transport/transport.py 中的 Transport 类
- ✧ 应用层 —— jspcap/protocols/application/application.py 中的 Application 类（暂未实现）

除此之外，上述文件中还定义有在 IANA 注册的协议编号表，即 LINKTYPE、ETHERTYPE 和 TP_PROTO。同时，在这些副根协议中，重载并实现了对下一层协议的导入和解析。

随后，基于各层级的副根协议，派生了各层级特定协议的具体实现，详细内容在此不做赘述。需要指出的是，其中部分暂未实现或完成的协议，均被置于 jspcap/protocols/*/NotImplemented/ 中。

此外，在 TCP 协议头中有 options 区域。程序专门设计了简易明了的数据结构加以分析处理，在代码 transport/tcp.py 中有详细的注释和说明。但由于时间仓促，因此暂未实现对 IPv4 中 options 区域的解析。

由于 jspcap 采取了流式读取的策略，在协议族中数据帧以 io.BytesIO 的形式传递，内存占用极小。但这使其变为 IO 密集型程序，后期或考虑协程（coroutine）进行优化。

5. 异常类

异常类，特指在 jspcap/exceptions.py 中声明的异常。这些异常由 BaseException 派生，是为用户定制异常。笔者曾在 jsntlib 的开发中探讨过如何定制化异常信息，但此处并无此需求，故略去。

(二) 用户界面

用户界面基于 wxpython 库编写，包括基于 wx.Frame 的主界面、FilterWindow、ReassemblyWindow、SearchWindow，以及基于 wx.Grid 的 PacketGrid 和辅助的 GetIface。

1. GetIface

使用 platform.system()获得操作系统，Windows 系统使用 wmi.WMI()获得网卡信息，Mac OS 系统使用 netiface.gateways()获得网卡信息，并返回一个储存网卡名的列表。

2. PacketGrid

基于 wx.Grid 实现 PacketGrid 作为显示数据包 summary 的界面，具体架构如下：

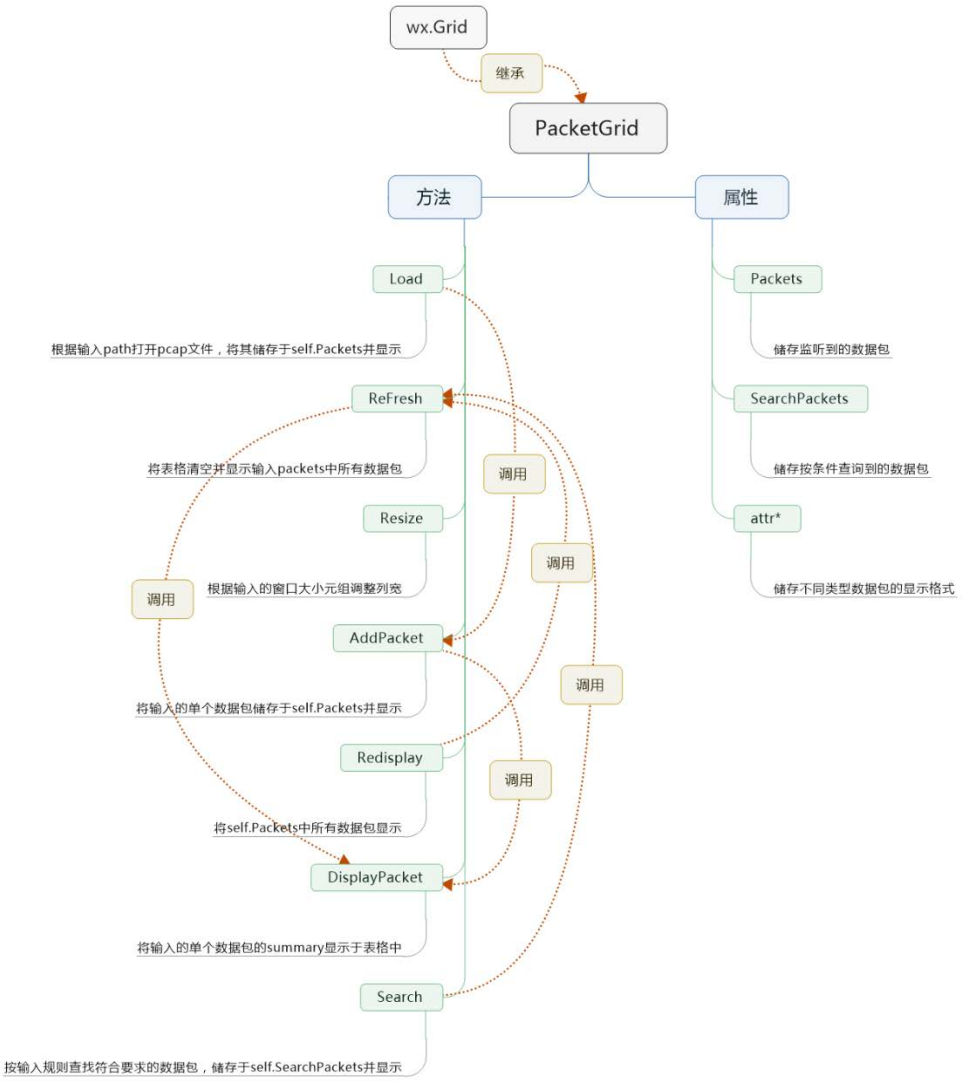


图 2-2 PacketGrid 类架构

3. FilterWindow

FilterWindow 基于 wx.Frame，用于选择过滤策略，确认后将主界面的 filter 属性置为设置值。

4. SearchWindow

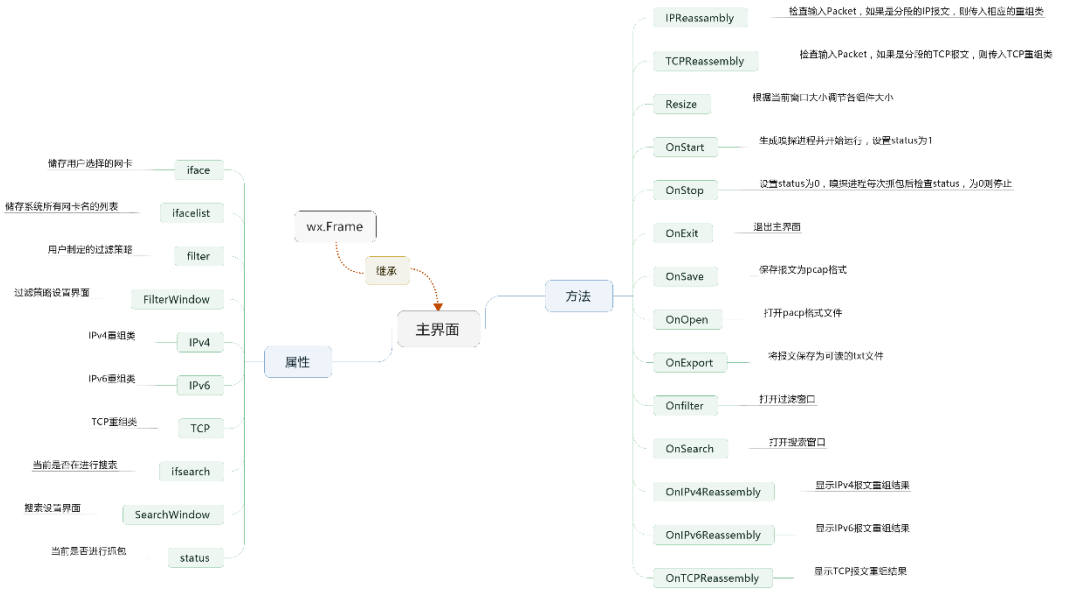
SearchWindow 基于 wx.Frame，用于查找，每次查找调用 PacketGrid 的 Search 方法。

5. ReassemblyWindow

ReassemblyWindow 基于 wx.Frame，用于显示数据包重组结果。

6. 主界面

主界面架构如下：



三、 主要数据结构

多个数据报文以列表的形势储存。

单个数据报文的不同协议数据以链的形式储存，链首为链路层协议，后面依次是更高层协议，协议数据作为链节点的属性。

四、 程序测试截图及说明

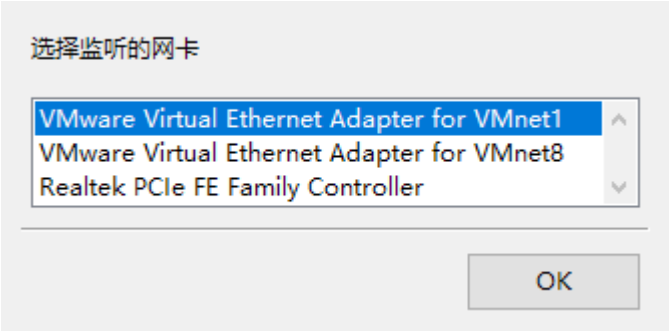


图 4-1 网卡选择界面

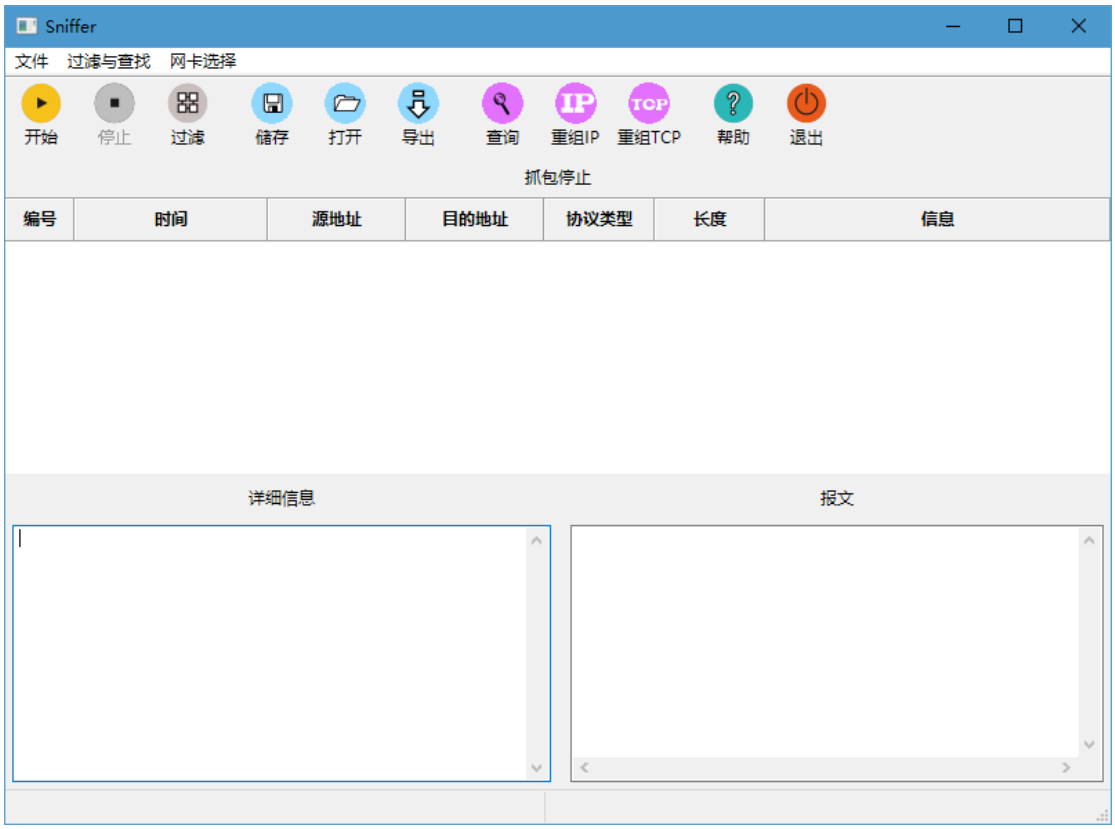


图 4-2 程序主界面

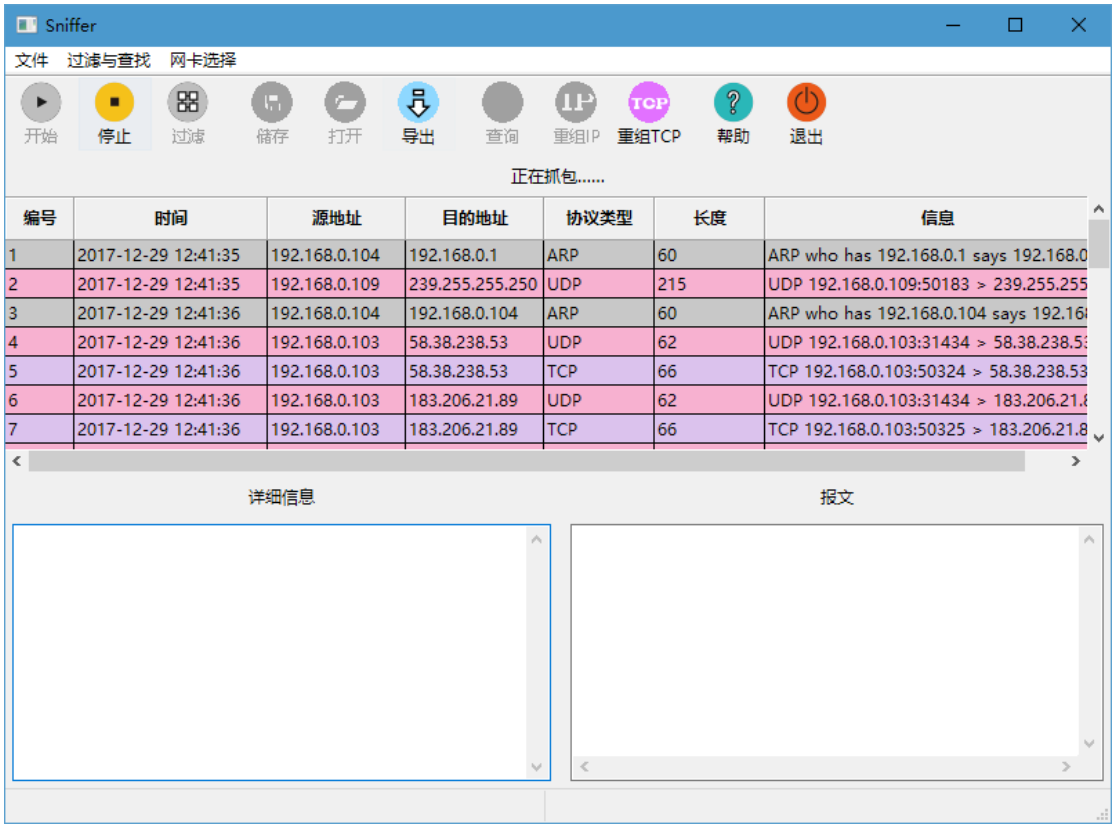


图 4-3 抓包过程

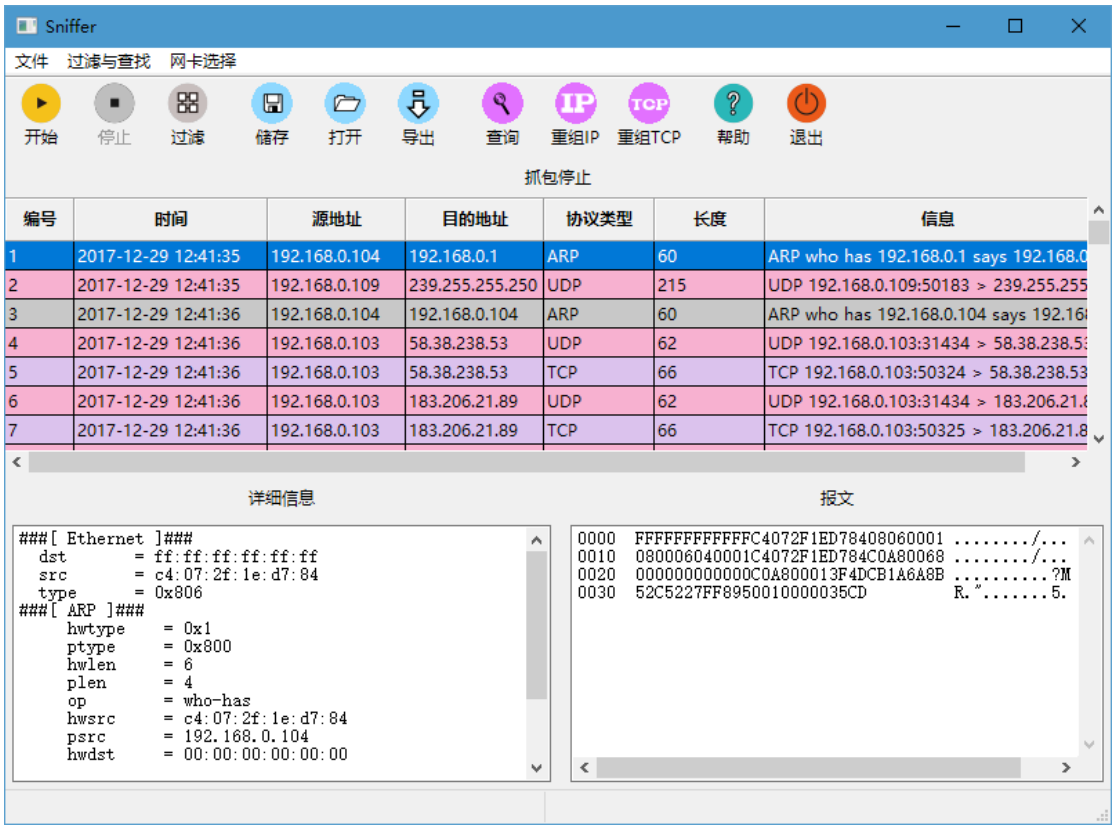


图 4-4 报文详细信息

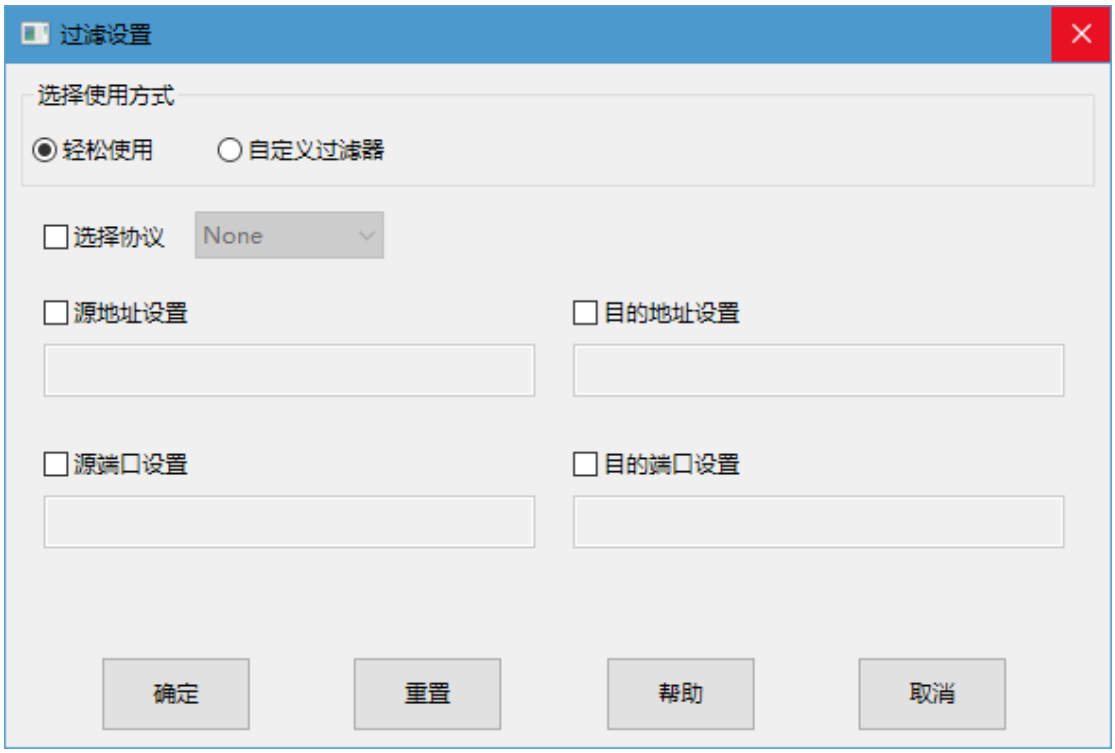


图 4-5 过滤界面

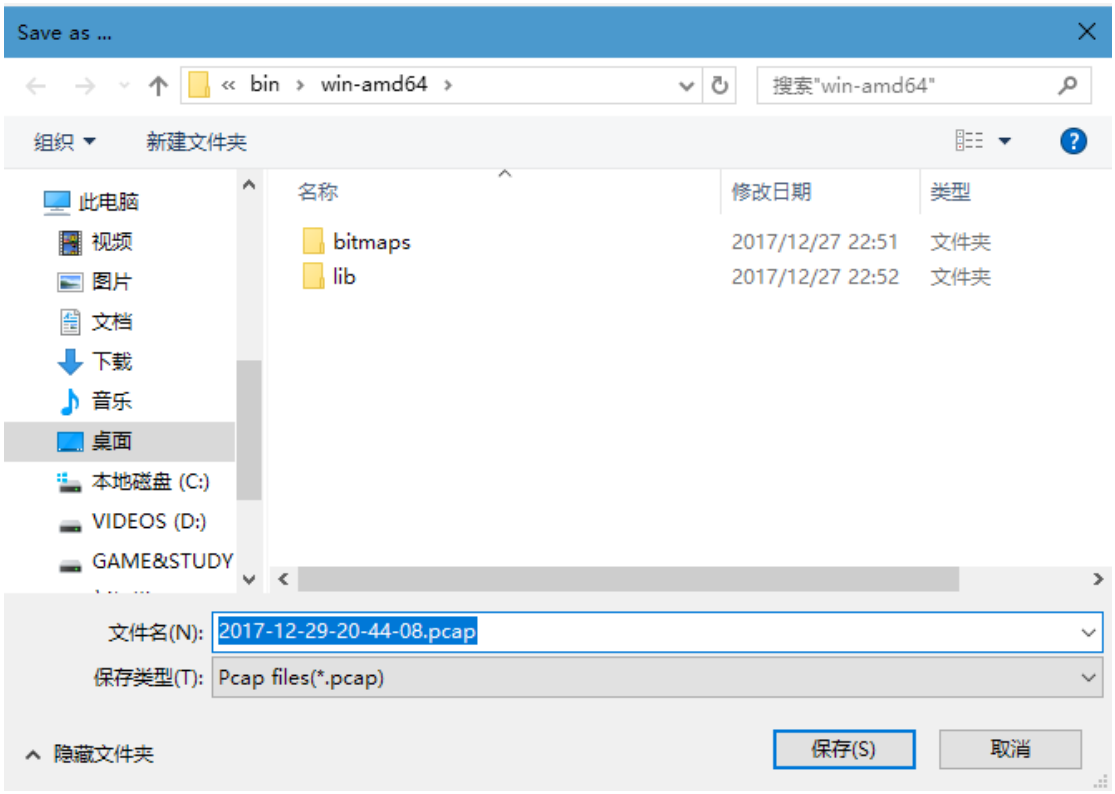


图 4-6 储存界面

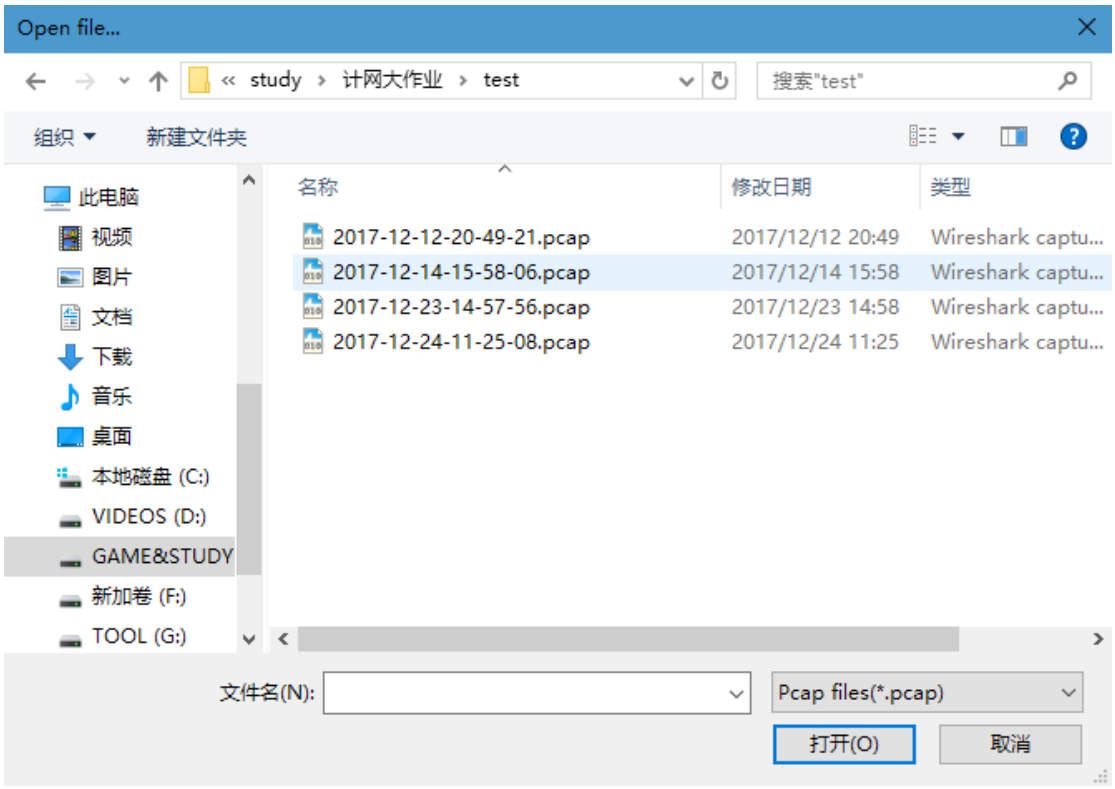


图 4-7 打开文件

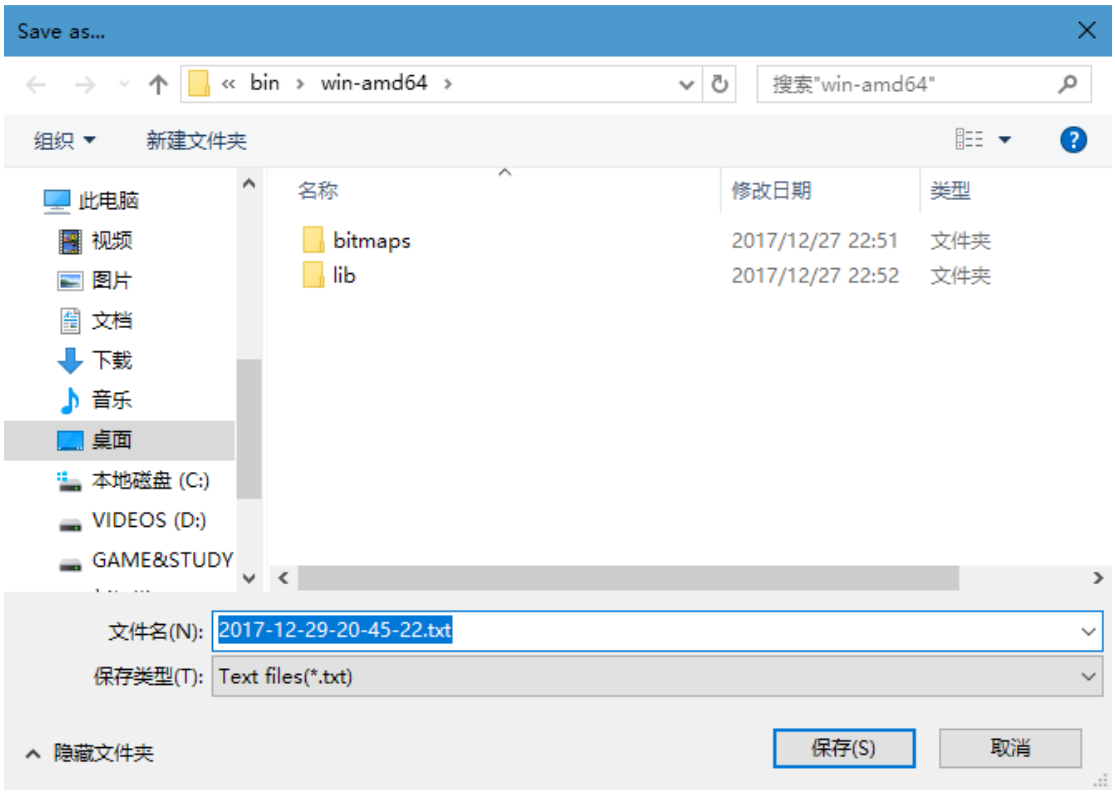


图 4-8 导出可读的报文详细信息

```
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = c4:07:2f:1e:d7:84
  type      = 0x806
###[ ARP ]###
     hwtype    = 0x1
     ptype     = 0x800
     hwlen     = 6
     plen      = 4
     op        = who-has
     hwsrc     = c4:07:2f:1e:d7:84
     psrc      = 192.168.0.104
     hwdst     = 00:00:00:00:00:00
     pdst      = 192.168.0.1
###[ Padding ]###
        load      = b'?M\xcb\x1aj\x8bR\xc5"\x7f\xf8\x95\x00\x10\x00\x005\xcd'
```

图 4-9 可读的报文详细信息示例

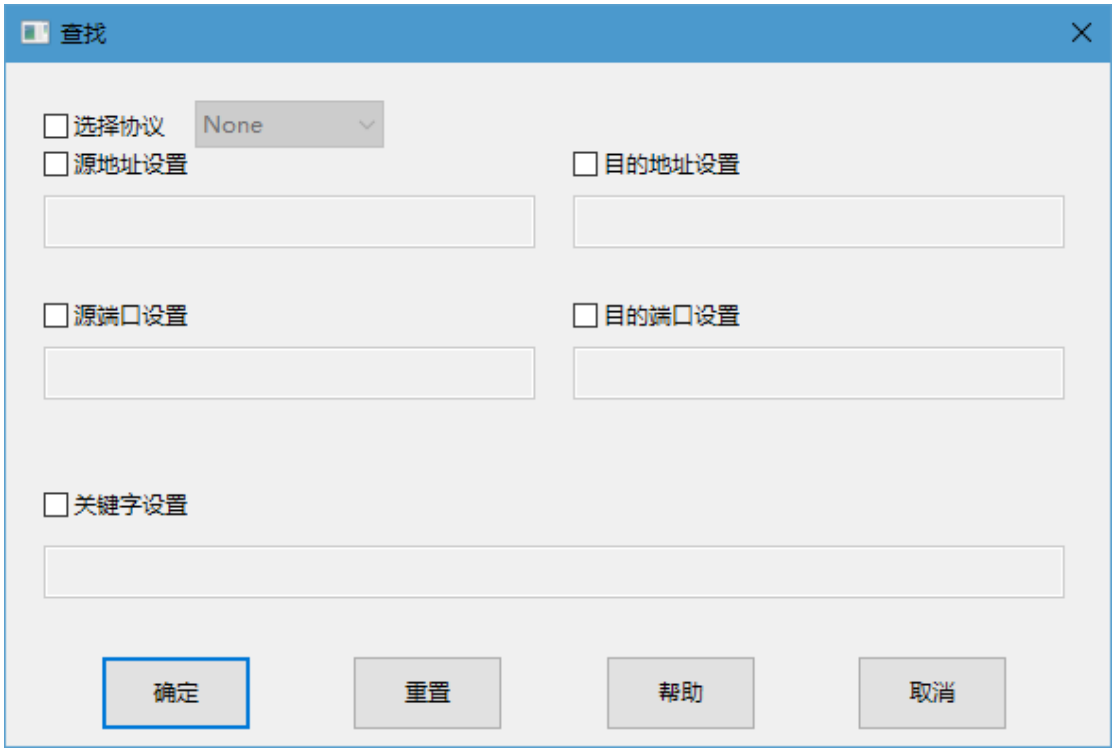


图 4-10 查询界面

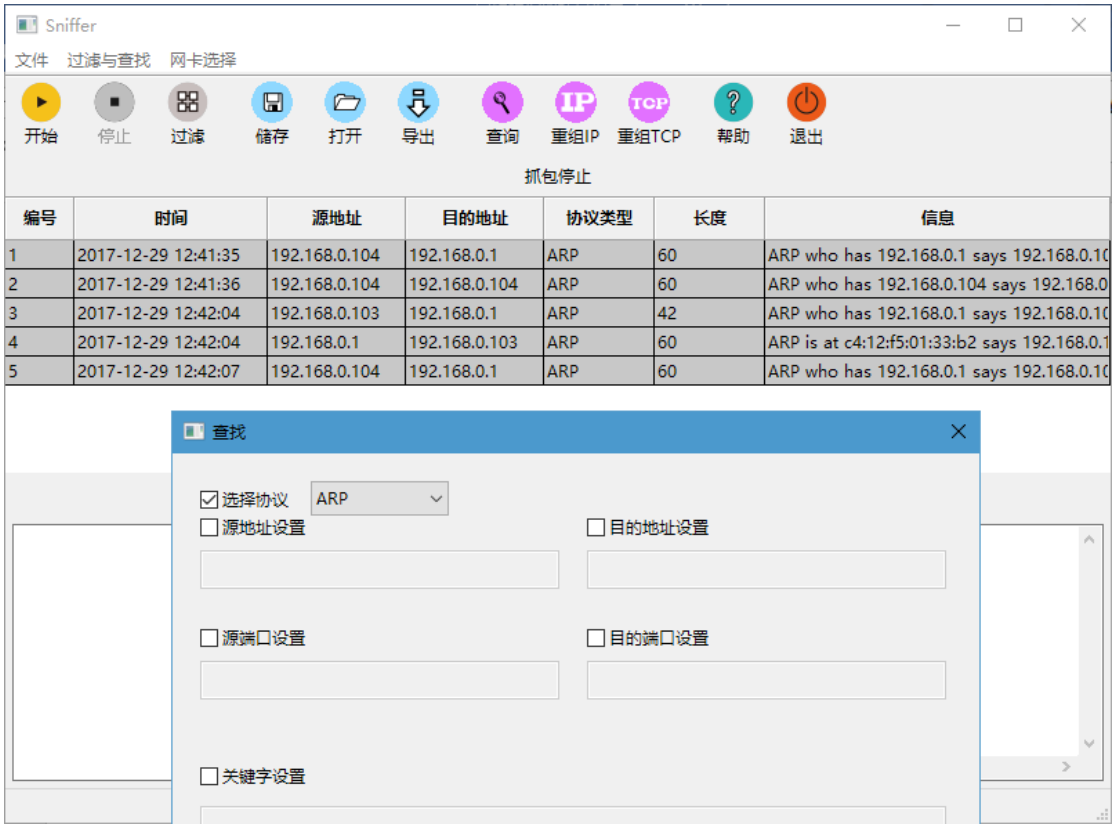


图 4-11 查询结果示例

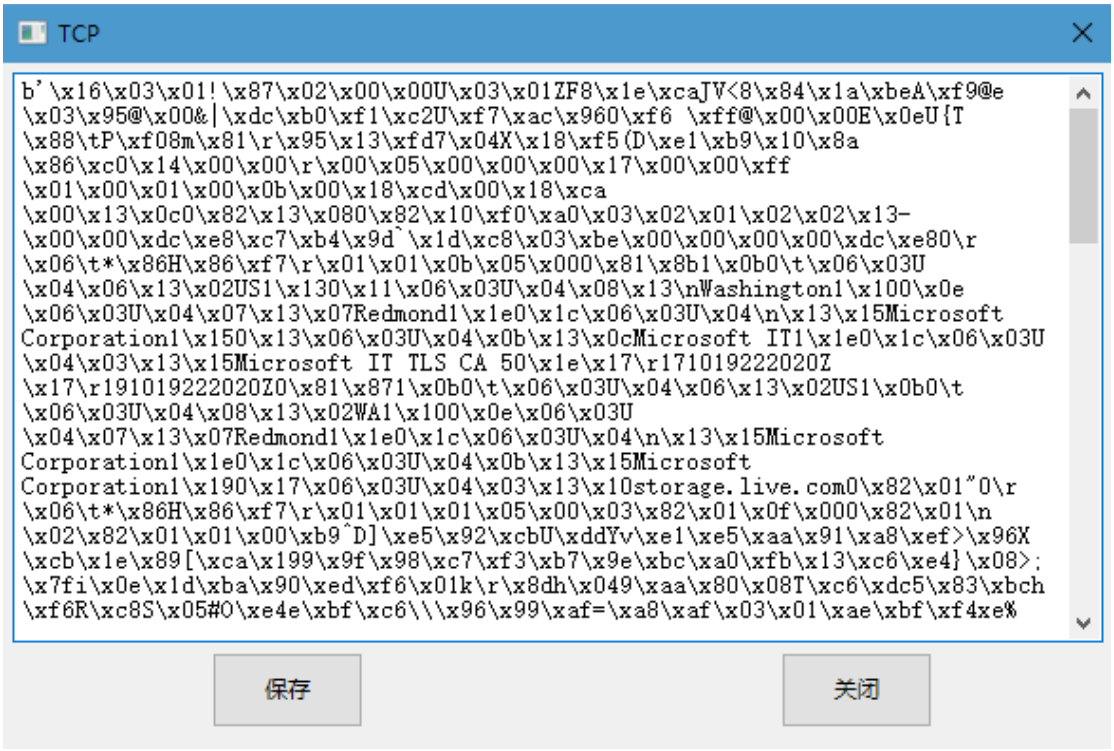


图 4-12 TCP 报文重组示例

五、 遇到的问题及解决方法

1. python 调用 sniff 函数会占用一个进程，并且 sniff 无接受停止信号的功能，只能通过预先设定抓包数量或 stop_filter 停止。

使用 status 属性标识嗅探是否正在运行，_PackageOperat 函数作为 prn 参数对每个包进行处理以实现实时更新，_IfRunning 函数作为 stop_filter 参数在每次抓包后调用，检查外部 status 是否为 0，为 0 则停止抓包并关闭线程。这样，用户开始抓包时只需要建立线程，停止抓包操作只需将 status 置为 0 即可。

2. 报文重组耗时较长，并且需要实时进行。

将重组操作作为类实现，并将相应方法写入_PackageOperat 函数，对每个新的数据报文进行重组操作并及时将重组完成的报文删去。

3. 缺少测试数据，没有抓到过 IP 报文分片的数据包。

六、 体会与建议

本项大作业对协议格式的掌握有很大帮助，原本我们组计划自己编写程序从二进制编码解析协议，但由于协议种类繁杂，部份协议如 IPv6 可选项较多，最终未能完成此项工作，因此退而求其次选择使用 scapy 已有的解析。但对协议结构的理解也有所加深。另外，在测试过程中缺少数据也拖延了完成进度，我们组在校园网环境中未能抓到分段的 IP 报文，建议以后可以提供少许难以获得的测试数据。