IMPLEMENTATION OF A WIRELESS NETWORK DETECTOR FOR IEEE 802.11 NETWORKS

THESIS

Presented to the Graduate Council of

Southwest Texas State University

in Partial Fulfillment of

the Requirements

For the Degree

Master of Science

By

Alexander P. Medvedev, B.S.

San Marcos, Texas

December 2002

ACKNOWLEDGEMENTS

I would like to thank Dr. Thomas McCabe for his supervision, ideas, and supplying me with the right hardware during this research. I also wish to thank Dr. Carol Hazlewood and Dr. Wuxu Peng for their participation.

October 15, 2002

Table of Contents

ABSTRACT vii
CHAPTER I INTRODUCTION TO THE STUDY 1
The Problem 1
Advantages of This System 2
The System's Possible Uses 3
CHAPTER II SYSTEM OVERVIEW 5
General 5
Packet Capture 5
Packet Processing 6
Result Display 7
User Interface 7
CHAPTER III WIRELESS PACKET SNIFFING 10
Packet sniffing overview 10
Conditions for Wireless Packet Sniffing 12
Components of a Packet Sniffer 13
CHAPTER IV 802.11B PROTOCOL SPECIFICS 14
General Description of the Architecture 15
802.11b General Frame Format 18
Important Frames for Sniffing 21
CHAPTER V OVERVIEW OF IEEE 802.11 SECURITY 22
Known Security Vulnerabilities in 802.11 22
Feasibility of Securing 802.11b Networks 24
CHAPTER VI DISCUSSION OF ACTIVE SNIFFER DETECTION TECHNIQUES 25
Which Sniffer is Active and Which Passive? 25
Detecting "Netstumblers" 26
CHAPTER VII IMPLEMENTATION DETAILS 28
Project History 28
General 29
Packet Capture 30
User Interface 31
Network Internal Nodes Discovery 32
Detecting Wireless Clients 32
Internet Address Range Detection 33
CHAPTER VIII CONCLUSIONS 34
Future Research 34
APPENDIX A 36
Prior Research and Systems Built 36
Advantages/Disadvantages of the Predecessors 37
APPENDIX B 39
REFERENCES 82
VITA 84

ABSTRACT

IMPLEMENTATION OF A WIRELESS NETWORK DETECTOR

FOR IEEE 802.11 NETWORKS

by

Alex Medvedev, B S.

Southwest Texas State University

August 2002

SUPERVISING PROFESSOR: Tom McCabe

This work describes the design and implementation of an 802.11b wireless network detector. The system puts the wireless network card into the RF monitor mode, examines traffic, extracts and processes important network identification information. The system is primarily designed to run on handheld devices similar to the Ipaq Pocket PC by Compaq and Zaurus by Sharp running the Linux operating system. The main objectives for this system were to be able to detect wireless networks, extract most useful information about the networks, and to be user-friendly.

CHAPTER I

INTRODUCTION TO THE STUDY

The Problem

The goal of this project was to study the IEEE 802.11 protocol and build a passive 802.11b network detector for a handheld computer. The features of the detector include detection and extraction of maximum information about wireless networks and displaying of the acquired data on the PDA's screen. In particular, the detector, named Discoverer, would extract the following parameters:

- SSID (Service Set Identifier);
- Channel number;
- WEP (Wired Equivalent Privacy) flag;
- Type of the network (Infrastructure or Ad-Hoc);
- MAC address and manufacturer's name of the AP (Access Point).

Additional information includes a timestamp when the network was discovered and last seen, the number of packets received, clients' MAC addresses, and the IP address range of the discovered network. A secondary goal was to determine if it was possible to reliably detect an active wireless network scanner (Netstumbler) in range.

There are several research projects that are studying detectors in wireless networks. For a short discussion, see Appendix A.

Advantages of This System

This system was designed to run on a handheld computer from the ground up. With minor changes it will run on an x86 machine as well. In fact it was developed on an x86 Linux laptop and then cross-compiled for the ARM architecture. With minimal changes it should be portable to any UNIX system that has wireless card drivers that support monitor mode and has wireless tools installed.

The system's operation is undetectable by other detectors or wireless stations because it puts the wireless network card into RF monitor mode and the card does not emit any radio signals. It is not possible to detect the presence of this device by listening to transmissions. This project's software is also capable of detecting non-beaconing access points (AP) through re-association requests. Data frames are also examined for fuller IP network information, such as IP address range and MAC addresses. The IP range is presented in the form of lowest and highest IP addresses transmitting on the network; in addition, if a DHCP reply is intercepted, network mask, gateway's IP address, and the DNS name are extracted. This form gives a better picture of the internal network structure and address range of a surrounding network.

Unlike other systems, such as Kismet (see Appendix A), there is no manual configuration file editing involved and the interface was specifically designed to be PDA user-friendly.

The system can discover cloaked network's SSIDs by monitoring re-association requests. A "cloaked" network is a wireless network that does not broadcast its SSID in management frames. This is usually done for tighter security. The clients should already know the SSID of the network they are trying to associate with. Thus the re-association

request, issued by the client, does contain the SSID, which is read and the network information is updated.

The System's Possible Uses

This system's primary purpose is a quick wireless IEEE 802.11b network audit. It can readily display all information that a network gives out to everyone with the right equipment. It was not, however, intended to be a full-blown network sniffer.

The system can be used to help choose an unused channel at a particular location to minimize cross talk and other interference.

It is also possible to use it to test whether the wireless network is "visible" at a certain location. This can be achieved by monitoring the packet count parameter.

In a secure installation an Access Point should not be a clear bridge without any filters enabled. Thus another use of this package would be to identify the location of misconfigured or "rogue" Access Points by observing MAC addresses on the segment reported by the software. If a MAC address of a particular network switch showed up, the AP must be attached to the same segment. This is particularly useful on complex multi switch networks.

This study confirmed that IEEE 802.11 Standard compliant devices give out a lot of network information that should be kept private. This is true even when 802.11 networks are used with Wired Equivalent Privacy (WEP) to encrypt the data traffic. The study also proved that anyone with the right equipment and motivation can learn a lot about one's wireless IEEE 802.11b network and possibly misuse the data.

The project and its code have been released to the public and placed on the Internet. Most support requests from the users are about wireless card driver installation and configuration. Features frequently requested include color support for active networks as well as signal/noise information display to assess the signal strength and the distance to the Access Point.

CHAPTER II

SYSTEM OVERVIEW

General

The system consists of two main modules: a capture module and a user interface module. The two run as distinct processes and communicate with the help of IPC message queues. The capture module collects the packets, analyzes them, and sends the results to the user interface module that displays them and updates contents when it is sent an update.

Packet Capture

Packet capture is performed using the libpcap library [11] (http://www.tcpdump.org). The network interface must be in promiscuous mode when capturing packets. The promiscuous mode is not enough for the wireless cards though. It will only detect Ethernet frames that fly by. In order to see all 802.11b headers, including the ones not destined for this machine, the capture interface also needs to be in the Radio Frequency (RF) monitor mode. In this mode the wireless network card can read raw frames. It simply works as a radio receiver and accepts all frames transmitted on the frequency. Raw frames are passed from the driver to the user programs that can examine and extract the necessary data. The system relies on the network card driver to calculate checksums and assumes that all packets passed to it are valid packets.

Packet Processing

After a packet was received from the capture engine, it is analyzed and classified by the frame analyzer. There are four main types of frames that contain information useful for network detection: management beacon frames, management reassociation request frames, management probe request frames, and data frames. Beacon frames contain most of the protocol information of interest including SSID, BSSID, channel, and encryption flag. It is possible, however, to "cloak" the network – configure an access point not to broadcast its SSID. This creates beacon frames with null SSID field and makes information extraction more difficult. It is still possible to extract SSID from the reassociation requests. The reassociation requests are issued by the clients whenever they get deassociated for any reason, for example, because of loss of signal due to distance or other conditions. These reassociation requests contain SSID, which they send to the AP while reassociating. These frames are analyzed by Discoverer, SSID extracted, and the entry for the network updated. Finally, the data frames contain the data packets, which usually are encapsulated Ethernet frames. The network type (Infrastructure or Peer-To-Peer) is determined from the capability field of the management frames and classified into AP or Ad-Hoc type accordingly. From the unencrypted Ethernet frames it is possible to learn IP parameters of the network, such as IP and MAC addresses of the clients from regular data frames and ARP requests/replies; default gateway's addresses, DNS server IP address, domain name, and network mask from DHCP replies. The information about discovered networks is stored in a list. IP address detection techniques cannot be applied to wireless networks protected by WEP.

Result Display

Runtime results are presented in a window in a list form. Only most important network identification data, such as SSID, channel, WEP, and manufacturer are displayed initially An entry may be expanded for more detailed information, which includes MAC address of the AP, number of packets received, timestamp when the AP was first seen, IP address range (if detected). This is done by pressing a plus sign next to the network's SSID. A total number of networks discovered is also displayed in the caption of the window.

User Interface

The QPE user interface was chosen, as it is one of the most popular windowing environments in the Linux handheld world. The user interface consists of a main window with 4 tabs: Main, Config, Log, and About. The Main tab contains the list view of discovered networks. The Config tab contains settable configuration information such as wireless network card type (Cisco, Orinoco, or Prism2), the device name (eth0, eth1, wlan0, wlan1, wifi0, wifi1), which may differ from card to card, and hop interval with default value at 300 milliseconds per channel. When the Save button is pressed the current values are stored in /etc/discoverer.conf file. Pressing the Default button resets the values to default values. The Log tab contains a scroll window of diagnostic, information, and error messages that outputs information, such as network names or MAC addresses discovered, and may indicate possible problems with setup. The About tab has the version information and the maintainer's e-mail address.

Figure 1, 2, and 3 represent screenshots of the Discoverer version 0.04.

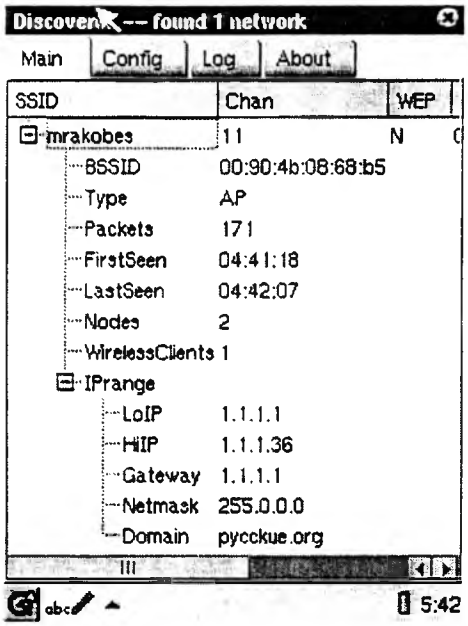

Figure 1 Main Screen. On an unprotected by WEP network all parameters for joining the network can be discovered.

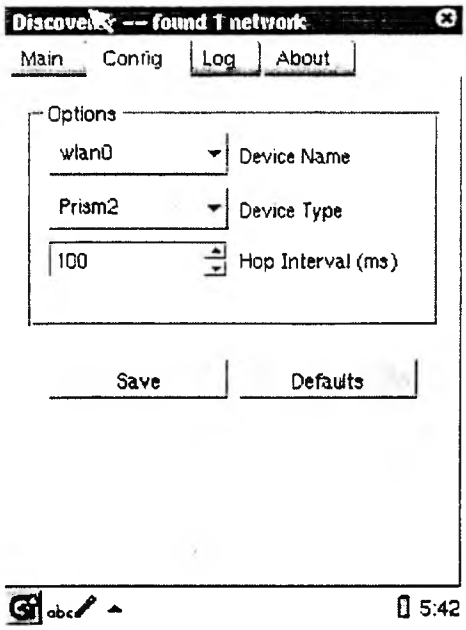

Figure 2 Configuration Screen. The user can select the interface name, the card type, and the channel hopping interval in milliseconds here.

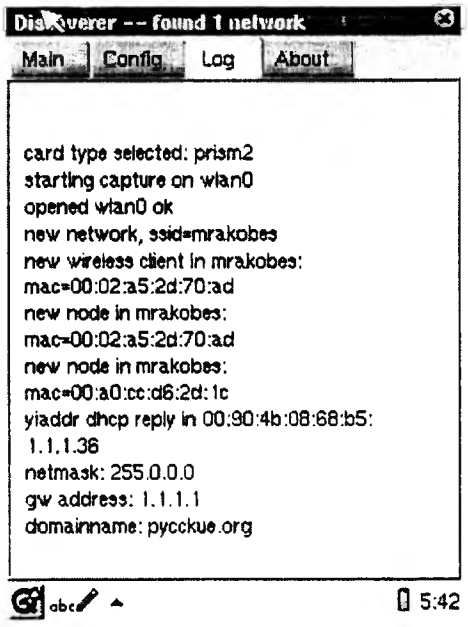

Figure 3 Logging Screen. The Log screen displays information about the current state of the active modules plus recent events and errors, if any.

CHAPTER III

WIRELESS PACKET SNIFFING

Packet sniffing overview

A network sniffer is a device capable of eavesdropping on the network traffic. Sniffers are mainly used on Ethernet networks, exploiting the fact that all communications among machines occur on the same physical medium. In normal communications the Ethernet cards only listen/respond to packets addressed directly for the card or to broadcast packets addressed to all nodes. When, however, the network card is put into promiscuous mode, the card stops filtering packets destined elsewhere and accepts all traffic.

An analogous situation exists on 802.11b wireless networks. In normal operation a wireless network card only accepts broadcasts and packets directly addressed to it (Fig. 4).

```
[root@laptop root]# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:07:0E:B3:5C:4A
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:192 (192.0 b)
          Interrupt:7 Base address:0x100
```

Figure 4. Wireless Network Card in Normal Operation

In the "normal" mode the link encapsulation is "Ethernet", which means that the Ethernet headers will be interpreted by the device driver and the higher levels will not see them. The interface flags, according to /usr/include/net/if.h, are UP, meaning that the interface is active; BROADCAST, meaning that the broadcast address is valid; RUNNING, meaning that resources are allocated; and MULTICAST, meaning that multicast is supported.

To capture all traffic on a wireless network, to which the card is associated, the wireless interface needs to be in promiscuous mode too. It is important to note that the card still needs to be associated with the access point or an ad-hoc network in order to see any traffic even in promiscuous mode (Fig. 5).

```
# ifconfig eth0 promisc
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:07:0E:B3:5C:4A
          inet addr:1.1.1.6  Bcast:1.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:8 dropped:0 overruns:0 carrier:8
      collisions:0 txqueuelen:100
      RX bytes:0 (0.0 b)  TX bytes:384 (384.0 b)
      Interrupt:7 Base address:0x100
```

Figure 5. Wireless Network Card in Promiscuous Mode

In promiscuous mode the link encapsulation is still Ethernet. A new flag appears in the interface flags line: PROMISC, indicating that the interface will not filter out packets destined to other nodes.

Conditions for Wireless Packet Sniffing

Promiscuous mode alone, however, is not enough for a wireless network card to be able to see all packets that "traverse the air" For wireless packet sniffing to work the network card must also be in the RF monitor mode (Fig. 6). This allows the card to capture all frames on a channel without being associated with a network (infrastructure or ad-hoc).

```
# echo "Mode: r" > /proc/driver/aironet/eth0/Config
# echo "Mode: y" > /proc/driver/aironet/eth0/Config
# ifconfig eth0
eth0      Link encap:UNSPEC  HWaddr 00-07-0E-B3-5C-4A-00-00-00-00-00-
00-00-00-00-00
          inet addr:1.1.1.6  Bcast:1.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:8 dropped:0 overruns:0 carrier:8
          collisions:0 txqueuelen:100
     RX bytes:0 (0.0 b)  TX bytes:384 (384.0 b)
     Interrupt:7 Base address:0x100
```

Figure 6. Wireless Network Card in RF Monitor Mode (Cisco card is used as example)

The link status changes to UNSPEC, which means that the raw (unmodified) packets will be passed to the higher level application by the device driver. This is very important precondition for an IEEE 802.11 network detector or a sniffer because all the 802.11b headers and all Ethernet headers are still attached to the packets. All that is left to do is to parse these headers and extract and classify all important network information. The MAC address of the card also changes appearance by acquiring 10 pairs of zeros and using dashes in place of colons. Putting a wireless card into monitor mode varies

significantly from card to card and from OS to OS. A good indication that the wireless network card is indeed in the promiscuous mode is the UNSPEC link type.

To sniff on all frequencies a software known as channel hopper is used. Channel hoppers change receiver's frequency cyclically while reading frames from the interface. This allows coverage of all channels. As the frequencies are being constantly changed not all traffic that present in the air is captured. To capture all traffic a separate receiver, i.e. separate card, for each channel is required. There exist 14 channels that 802.11b operates on. The bands are different for Europe, US, and Japan. In the US channels from 1 to 11 are used. Channel hopping is also card type and OS dependent.

Components of a Packet Sniffer

A typical packet sniffer consists of several components: capture engine, packet decoder, and a user interface. It also needs a mechanism for passing data between these modules. For wireless sniffing one more component is necessary: the channel hopper. It is sometimes possible to combine certain components in a single module. For example, in this project the channel hopper is combined with the user interface and the capture engine with the packet decoder. Thus this project only has 2 modules, but still it has all 4 distinctive parts. This was done for convenience. Also it avoids generation of too many separate processes, which limits interprocess communications.

CHAPTER IV

802 11b PROTOCOL SPECIFICS

This chapter gives a basic technical overview of the 802 11b protocol. It is based on ANSI/IEEE Std 802.11, 1999 Edition [9]. The purpose of this standard is to provide a medium access control (MAC) and physical layer (PHY) specifications for interconnecting wireless devices within a local area. The devices may be portable, handheld, or fixed within the local area.

The standard defines a single MAC that interacts with the following three physical layers (see Fig. 7): FH (Frequency Hopping Spread Spectrum) and DS (Direct Sequence Spread Spectrum) in 2.4 GHz band and IR (Infrared) in infrared band [Ref Breezecom].

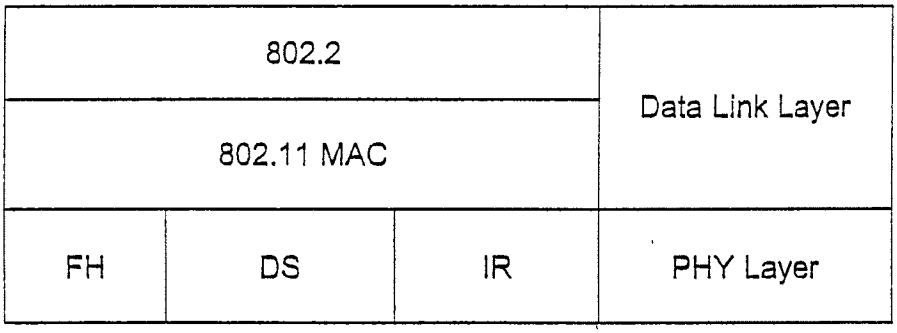

Figure 7 IEEE 802.11 Layers Description [Ref Breezecom]

Besides the standard functions, IEEE 802.11 MAC performs tasks typical of upper level protocols, such as fragmentation, retransmissions, and acknowledgements.

Only protocol aspects relevant to the study were considered. The discussion focuses on the MAC layer, such as frame formats, and omits the physical layer, such as media access methods, fragmentation and reassembly, etc.

General Description of the Architecture

Wireless networks significantly differ from wired LANs [9]. For example, a wireless station does not have a fixed location and therefore the message destination is a station, not a location. The physical layers of IEEE 802.11 are different from wired network physical layers. For instance, the wireless physical layer has no absolute boundaries, has dynamic topologies, is unprotected from outside signals, and lacks full connectivity (one cannot assume that every node sees every other node). It is also less reliable than its wired equivalent and has time-varying and asymmetric propagation properties [9].

Moreover, since most mobile stations are battery powered, it cannot be assumed that a particular node always stays powered on. IEEE 802.11 should appear to the LLC (Logical Link Control) layer as an IEEE 802 LAN. Therefore the 802.11 network needs to handle station mobility within the MAC layer.

An IEEE 802.11 network is based on a cellular technology and therefore subdivided into cells [2]. A cell is called a Basic Service Set (BSS) and controlled by a Base Station called Access Point (AP). Multiple access points may be connected by a backbone (Fig. 8) forming an Extended Service Set (ESS). All data from non-802.11 networks enter the 802.11 architecture via a portal [9], which bridges the two different

architectures. Usually a single device combines functions of both an AP and a portal In other words an AP serves as a bridge between segments of different architectures.

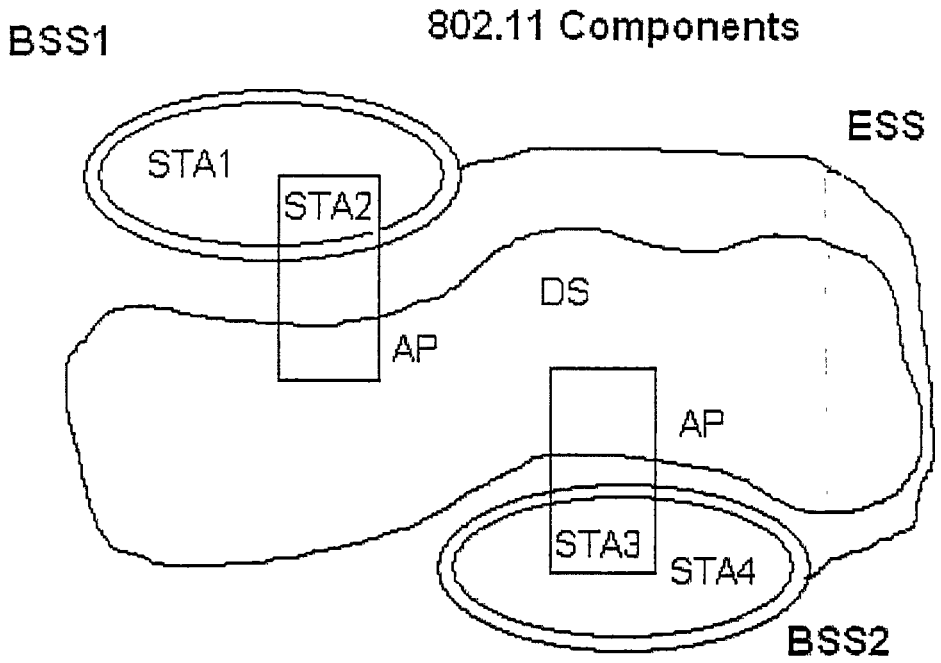

Figure 8 Extended Service Set (ESS)

A minimum IEEE 802.11 LAN can consist of just two stations without an AP. This can be a temporary network between two laptop computers where each station's signals are detectable by the others and thus communication can take place directly. This type of network is usually formed without pre-planning for as long as the LAN is needed and referred to as an Ad-Hoc network. The stations assume some responsibilities of an AP in this case, for example, beacon frame generation. In contrast, the networks where the AP is present are called Infrastructure Networks.

Before a station can participate on the network it first must become associated (or synchronized) with the corresponding AP. A station discovers an AP and associates with it by invoking the association service. The station can obtain the synchronization information in two ways: by actively sending probe requests and waiting for a probe response from the AP or by passively waiting to receive a beacon frame. Once an AP is found the station goes through the authentication process and at this time the station proves the knowledge of a password to the AP. When authenticated, the station starts the association process, which involves exchange of information about the station and BSS capabilities. At any given moment a station can be associated with only one AP, while an AP can be associated with many stations at the same time. To support BSS mobility, for example to move between APs, the reassociation service is used. It simply "moves" current association from one AP to another. Reassociation is always initiated by the station.

An AP may disassociate a station by sending a disassociation request. After disassociation all attempts to send messages to the disassociated station will be unsuccessful. Similarly, a station will attempt to disassociate when leaving a network.

Access and confidentiality control services in 802.11 networks are provided by two services: authentication and privacy [9]. According to the IEEE 802.11 Standard, authentication serves a purpose analogous to the wired media physical connection and privacy provides the confidential aspects of closed wired media. The most commonly used authentication mechanism in IEEE 802.11 networks is Shared Key authentication. It requires the use of the wired equivalent privacy (WEP) with a shared secret (WEP encryption key). WEP is also an optional privacy algorithm. Although it is not designed

as the ultimate security its goal is to provide as much security as a wire [9]. When used, all data frames are encrypted with 40, 64, or 128-bit encryption.

802.11b General Frame Format

Each frame in IEEE 802.11 consists of a MAC header, a frame body, and a frame check sequence (FCS) in a fixed order (Ref Standard) and is depicted in Figure 9. The MAC header comprises frame control, duration, addresses, and sequence control information. The frame body is specific to a frame type and varies in length. FCS is a 32-bit cyclic redundancy check (CRC).

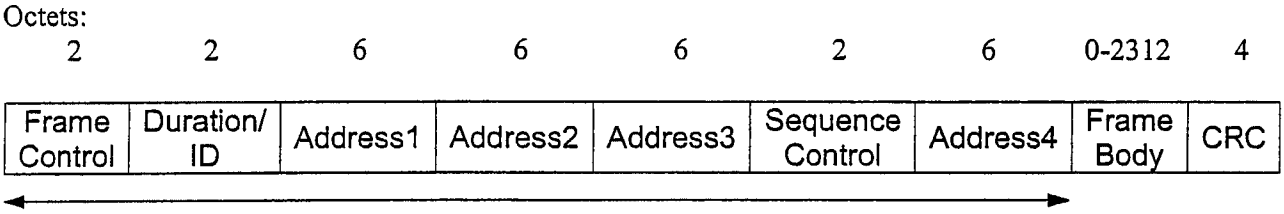

Figure 9 General MAC Header

The frame control field in the MAC header is important in identifying which network frames belong to and has the following subfields (also shown in Figure 10):

1. *Protocol version* field is 2 bit in length and is always 0 in this version of standard.
2. *Type* field is 2 bit in length and identifies the frame type: management, control, or data.
3. *Subtype* field is 4 bit in length and in combination with the field type identifies the frame function. For example, management frame association request or control frame acknowledgement.

4 *To DS* field is 1 bit in length and is set to 1 in data frames destined for the Distribution System (DS). For example, all data frames sent by a station associated with an AP. The DS field is set to 0 otherwise.

5. *From DS* field is 1 bit in length and is 1 in all data frames leaving the DS and is set to 0 in all other frames.

6. *More Fragments* field is 1 bit in length and is set to 1 in all management or data frames, which have another fragment of the current MAC service data unit (MSDU) or current MAC management protocol data unit (MMPDU) to follow. This field is set to 0 in all other frames.

7. *Retry* field is 1 bit in length and is set to 1 in all retransmitted data or management frames. It is 0 in all other frames.

8. *Power Management* field is 1 bit in length and indicates the power management mode of a station in which the station will be after a successful completion of the frame exchange sequence. A value of 1 means power-save mode, 0 – active mode.

9. *More Data* field is 1 bit in length and, when set, indicates to a station in power-save mode that it has data waiting for it at the AP.

10. *WEP* (Wired Equivalent Privacy) field is 1 bit in length and is set to 1 in data frames or management authentication frames if the frame body was processed by the WEP algorithm.

11. *Order* field is 1 bit in length and is set to 1 in data frames containing MSDU (or MSDU fragment) that are transferred using the StrictlyOrdered service class.

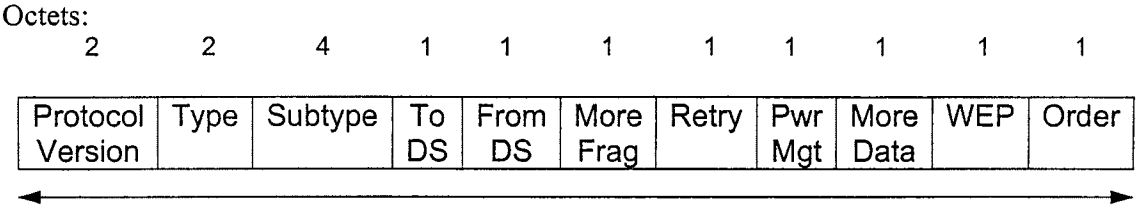

Figure 10 Frame Control Field

The Duration/ID field of the MAC header is 16-bit in length and, depending on the frame type, carries either the association identity (AID) or duration value as defined for each frame. The four address fields are 48 bit in length in MAC address format. An address field can indicate BSSID (basic service set identifier), SA (source address), DA (destination address), RA (receiver address), and TA (transmitter address). Some frames do not have certain address fields. The sequence control filed is 16 bit in length and indicates the sequence number of MSDU or MMPDU. The frame body field is variable length (0-2312 bytes) and is specific to each frame type. The FCS field contains a 32-bit CRC.

As mentioned above there are 3 types of frames: management, control, and data. Management frames are used to exchange management information, but are not forwarded to upper layers [2]. These include beacons, association and reassociation requests, and others. Control frames are used to control access to the medium, for example, when a station wishes to transmit it issues a RTS (Request To Send) control frame. The AP then replies with a CTS (Clear To Send) control frame. Finally, the data frames are used for data transmission and normally encapsulate Ethernet traffic.

Important Frames for Sniffing

Not every frame carries complete network identifying information. To uniquely identify an 802.11b network one needs the following: BSSID, network name (SSID), channel, and presence of WEP. For the purpose of this study a network's "unique identity" is simply the configuration information needed in order to participate on the network. It is also useful to know how many packets a network has sent since it was discovered, the IP address range of the network, and the number of wireless/wired clients participating in it. The frames that carry the most 802.11b network parameters are management frames: beacons, probe requests, and reassociation requests. Data frames contain MAC addresses of the transmitting/receiving clients and, if WEP is not configured, the IP address information, which could be extracted and examined. All other frames carry minimal or, without thorough tracking, even ambiguous information for the purpose of this study and therefore are not considered.

CHAPTER V

OVERVIEW OF IEEE 802.11 SECURITY

The wireless networks also differ from their wired equivalents in several security aspects. For example, on a wired network authentication is implicit in the fact that a client is physically connected to the network [5]. Moreover, altering traffic, as it passes through the wire, is not trivial. The physical topology is quite different on the wireless networks, however. It is much easier to hijack sessions, filter traffic, or inject messages in the authentication sequence in the absence of strong mutual authentication.
These weaknesses in the IEEE 802.11 design come from the fact that IEEE 802.11 is based on IEEE 802.1x group of standards that assumed a network with a fixed physical location.

Known Security Vulnerabilities in 802.11

There are three major security weaknesses in the IEEE 802.11 protocol [14].

The first weakness comes from the use of unencrypted 802.11 networks. Such sessions are vulnerable to snooping and hijacking regardless of authentication methods. Wireless 802.11 networks are easy to discover with the right equipment, such as Netstumber, Kismet, or the software developed during this research. If the communications are unencrypted and no access control is used it is simple to configure an attack machine to join the victim AP and participate on the network. All communications can be observed in clear text with software tools that are normally used on an Ethernet network, e.g. tcpdump or ethereal.

The second problem is weakness in the WEP encryption scheme [Rice university paper]. It is well known that WEP offers only weak encryption. Also WEP lacks support for per-packet integrity protection, thus enabling a wide variety of attacks, such as insertion of packets into the data stream [14]. It is worth noting that only early WEP implementations were vulnerable to key cracking by tools such as Airsnort [6]. The latest firmware releases from most vendors prevent or eliminate all known attacks. For example, avoiding cryptographically weak packet generation makes WEP key retrieval nearly impossible.

Finally, the lack of authentication for 802.11 management messages is the third and the biggest of the known security vulnerabilities of 802.11. These include beacon, probe request/response, association request/response, reassociation request/response, disassociation, and deauthentication. Generally speaking, all these management messages must be authenticated; otherwise DOS (denial of service) attacks are possible. It is possible, for example, to deny service to a 802.11b client by creating a fake deauthentication request that appears to the AP as if it was sent from the victim machine. This is achieved by using the victim's MAC address as the source address in the deauthentication frame. When the AP receives such request, it happily deauthenticates the client, forcing the victim to reestablish the connection. There exist tools that exploit this and more, for instance, Airjack [12] (http://802.11ninja.net). In addition Airjack can take over a connection at layers 1 (Physical: radio channel manipilation) and 2 (Datalink: MAC and LLC). It inserts the attacking machine between the victim and the access point, making the unsuspecting victim associate with the attacking machine instead of the AP on a different channel. The attacking machine in its turn associates with the AP as the

client on the channel advertised by the AP. Thus all communications between the client and the AP are going passing through the attack machine. It was demonstrated to successfully implement a man-in-the-middle (MITM) attack even on encrypted 802.11b networks during the Black Hat Briefings conference in Las Vegas in August 2002. The MITM attack on encrypted networks was possible because many security solutions are implemented with an assumption of secure levels 1 and 2. It is believed that Airjack was also used as a DOS tool at Defcon conference that took place just days after the Black Hat Briefings.

Feasibility of Securing 802.11b Networks

According to the current media coverage it seems that IEEE 802.11 has multiple gaping holes and, therefore, insecure. All of these insecurities, however, can be eliminated by taking reasonable security measures [6]. For example, use of wireless sniffers and scanners can help locate unauthorized access points in the enterprise, adoption of wireless intrusion detection systems (IDS) and monitoring tools can inform about attempts of unauthorized access to the network, implementation of virtual private networks (VPN) with strong authentication will prevent eavesdropping [12], and, finally, radio signal containment and prevention signal leaks by using directional antennae would also aid eavesdropping avoidance.

CHAPTER VI

DISCUSSION OF ACTIVE SNIFFER DETECTION TECHNIQUES

Which Sniffer is Active and Which Passive?

Wireless network sniffers can be classified in two groups: active and passive. In short: the active send out packets or signals and the passive do not.

Active sniffers attempt to participate on the network and can only eavesdrop on traffic belonging to the network they are associated with. Its presence can obviously be noticed by viewing the list of associated clients. Moreover, before an active sniffer can participate on the network it has to associate with it. And before it can associate it must find a suitable network. This search is usually done by rapid switching channels and sending probe requests on each frequency. This is done in hope of receiving a probe response that contains the network information for participating on the network. It is not hard to detect these probe requests. The problem is that these probe requests are not easily fingerprinted, because they look exactly as probe requests from legitimate clients. Therefore, basing netstumbler detection on just receiving probe requests on different channels would generate too many false alarms.

The passive sniffers on the contrary do not have to send out any packets. They simply switch channels and listen on each frequency for a short while in hope of receiving any 802.11 traffic. When a frame that is not classified as noise is received it is parsed and network information extracted. Data frames that usually carry TCP/IP packets are examined and dumped to a file for future analysis. If the goal is just the wireless

network detection, only limited parameters are extracted or the data frames are completely ignored.

It is theoretically possible to detect such sniffers, even passive, due to presence of antennae on the wireless network card [4] but the technique is quite involved and is not very accurate.

Detecting "Netstumblers"

There exist many programs that are considered "netstumblers", which can be classified as active wireless network detectors. Their sole goal is to detect a wireless network by issuing probe requests and reading probe responses. Most of them are written for Linux and BSD-based systems. The original Netstumbler was written for MS Windows and is a closed source project.

As mentioned above the probe requests that all of these netstumblers send are indistinguishable from legitimate clients, and the MAC addresses on the wireless cards may be altered, so that they are hard to detect. It was noted, however, that the MS Windows original Netstumbler emits an LLC data packet containing a certain text string (Fig. 11). The string is different for different versions of the Netstumbler.

This LLC packet is the basis for Netstumbler detection in the latest version of Kismet. It is obvious, however, that this technique is not general and all other non-windows "netstumblers" go unnoticed. This makes this technique unsuitable, for instance, for an Intrusion Detection System (IDS).

```
IEEE 802.11
    Type/Subtype: Data (32)
    Frame Control: 0x0008
        Version: 0
        Type: Data frame (2)
        Subtype: 0
        Flags: 0x0
            DS status: Not leaving DS or network is operating in AD-HOC
mode (To DS: 0  From DS: 0) (0x00)
            .... .0.. = More Fragments: This is the last fragment
            .... 0... = Retry: Frame is not being retransmitted
            ...0 .... = PWR MGT: STA will stay up
            ..0. .... = More Data: No data buffered
            .0.. .... = WEP flag: WEP is disabled
            0... .... = Order flag. Not strictly ordered
    Duration: 0
    Destination address: 01:60:1d:00:01:00 (01:60:1d:00:01:00)
    Source address: de:ad:be:ef:fe:ed (de:ad:be:ef:fe:ed)
    BSS Id: de:23:be:53:00:01 (de:23:be:53:00:01)
    Fragment number: 0
    Sequence number: 1474
Logical-Link Control
    DSAP: SNAP (0xaa)
    IG Bit: Individual
    SSAP: SNAP (0xaa)
    CR Bit: Command
    Control field: U, func = UI (0x03)
        000. 00.. = Unnumbered Information
        .... ..11 = Unnumbered frame
    Organization Code: Unknown (0x00601d)
    Protocol ID: 0x0001
Data (58 bytes)

0000 00 00 00 00 20 20 20 20 20 20 20 20 20 20 69 6e ....          in
0010 74 65 6e 74 69 6f 6e 61 6c 6c 79 20 62 6c 61 6e tentionally blan
0020 6b 20 20 20 20 20 20 20 20 20 20 20 fe ca ba ab k           ....
0030 ad de 0f d0 00 00 00 00 00 00                    ..........
```

Figure 11. LLC packet generated by a Netstumbler version 0.3.30 (captured by Mike Craik <bovine@btinternet.com>). Attributes characteristic of a Netstumbler LLC packet are in bold.

CHAPTER VII

IMPLEMENTATION DETAILS

Project History

The idea for this project belongs to Dr. McCabe. After interest was expressed and the specifications for the software discussed, two Ipaq handheld computers with Orinoco Gold wireless cards were purchased by the Computer Science department. Linux was chosen as the development platform because of the free access to the source code of the operating system and the device drivers. Presently, Linux installation on a handheld computer is still not trivial and risky in terms of damaging the equipment [7]. It involved installation of a Linux bootloader, followed by the kernel image with minimal system, and then the packages required to run useful programs. These tasks were performed using a Linux PC through the serial port. Since specific drivers were needed to run wireless cards in the monitor mode, the Linux kernel and the drivers needed to be recompiled for the Ipaq, (ARM architecture). This involved setting up an ARM cross-compiler and libraries on an Intel x86 machine running Linux. Development libraries, such as libpcap, were also cross-compiled. Orinoco wireless card drivers were patched to include the monitor mode support. Because the Orinoco drivers for Linux are not very stable, some informational, debugging, and version checking code was modified or excluded from the drivers.

After everything needed was cross-compiled and installed on the Ipaq, a command line version of the packet capture engine was written and tested on x86 and ARM Linux. At this point several differences in how compilers generate code were

observed For example, `sizeof(struct)` turned out to calculate sizes of the same structure differently on different platforms. These differences had to be accounted for and incorporated in the code. Similar situations existed with the QT interface: the code that worked on x86 produced segmentation violations on ARM or simply the graphics looked differently. Currently though, identical code compiles and runs on both architectures equally well.

General

The software developed for this project consists of two main modules: capture and user interface. When started, the user interface reads the configuration file, applies current settings, and forks the capture module. Depending on the card type, an appropriate channel hopper is chosen and invoked. The capture module creates a message queue where it puts newly found networks (message type 1) and pushes the updates of the network list (message type 2). The user interface monitors the queue and updates the display when new information arrives. Message queues were chosen as a method of inter-process communication because of its simplicity, ability to distinguish between message types (an update message from a new network or an informational message), and minimal synchronization between the two processes. The sending process puts messages in the queue and does not block. Also, the message queue does not have to be open from the "other end" as in FIFO pipes for example. The receiving process uses a non-blocking receive to check the queue for messages. If the queue is empty it simply continues execution without blocking.

Packet Capture

The packet capture is performed using the libpcap library from the tcpdump.org project [11] (http://www.tcpdump.org). The packet capture process is forked from the user interface module. The network card device name is passed to the capture module as a parameter from the user interface module. The capture module calls the `pcap_open_live()` function, which opens the specified device for promiscuous packet capture. If the device open call succeeds then a `pcap_loop()` function starts an infinite capture of packets from the interface. Every time a packet is acquired, a callback function called `my_callback()` is executed. This function makes copies of received data and its header for further handling and calls the `p_data_analyzer()` function that examines the captured frame and classifies it as a management (beacon, re-association request), control, data, or other frame. Based on this classification, a different processing code is called for each type of frame.

Each of the processing elements extracts all potentially useful information and, if the network is new, adds it to the list of networks, and then puts it in the message queue for the user interface to collect. If no new information was discovered, the frame count is incremented and the frame content is ignored. The capture module also sets a timer that goes off every second and pushes the current network list with all the up-to-date parameters to the user interface. This forces the network list update on the display which includes time the network was last seen, number of packets received, and any new information that may have been discovered or changed.

User Interface

The user interface was written using Qtopia from Trolltech that can be found at http://www.trolltech.com [15]. This is a QT development kit that could be used for cross-platform development. A designer program was used to quickly build the appearance of the interface. This also created a base class called `Discovererbase`. Then a derived class called `Discoverer` was created and virtual functions of the base class were re-implemented when needed.

The user interface performs multiple tasks besides displaying the list of networks and updating display. This includes channel hopping for Orinoco and Prism2-based wireless network cards, enabling monitor mode on Cisco-based cards, reading configuration files and saving changes, and displaying a log of recently performed actions as a debugging aid.

When a new network appears in the message queue it is simply added to the list of networks after the last discovered network (if any). If an update message arrives, the network list tree is traversed for a BSSID match, then the packet count, timestamp, and in certain cases SSID are updated with new data.

The interface was written almost entirely in the C++ programming language.

Portability issues may arise when trying to adapt the system for a non-UNIX environment, because of calls to the `system` function. This function is used mainly to perform network interface controlling actions, which are OS and card driver specific, for example, enabling interface, changing channels, and setting monitor mode.

Network Internal Nodes Discovery

The network internal node discovery is based on the fact that most access points are also portals (or bridges). They usually bridge the Ethernet segment with the Wireless segment. Therefore broadcasts made by the wired nodes are also "heard" on the wireless segment. By examining data frames the MAC addresses of the wired nodes can also be obtained. The wired node discovery is possible even with WEP enabled. When given enough time it is possible to discover all nodes on the subnet.

It is also possible to determine what BSSID a particular MAC address belongs to because a typical data frame contains the following fields: destination MAC, source MAC, and BSSID. All three are examined for new MAC addresses that are collected, classified, and appended to a linked list with the appropriate BSSID. Broadcast and multicast addresses are filtered out. The `Nodes` counter is incremented.

Detecting Wireless Clients

The MAC addresses collected from the data frames include wired and wireless nodes on the network. The wireless client discovery is based on the fact that wired clients do not emit 802.11b specific frames, such as management or control frames.

In order to separate the wireless nodes, the discoverer watches the management frames and if a new MAC address is found in a management frame with the same BSSID a new wireless client is added to the list of wireless clients and `WirelessClients` count is incremented.

Internet Address Range Detection

The IP range detection relies on unencrypted data frames. Thus this feature is not available on networks protected with WEP. The IP addresses are extracted from the Ethernet headers of ARP packets and DHCP replies encapsulated into the body of 802.11b frames. The DHCP replies are part of the Dynamic Host Configuration Protocol and contain all necessary information to participate on the IP network. This includes the client's IP address and may also contain network mask, default gateway, and domain name. The DHCP replies are quite rare and are usually transmitted when a client boots or its DHCP lease expires. Therefore the ARP replies containing IP addresses of local nodes are also of value. The data structure holding network information only keeps track of the highest and the lowest IP numbers detected on the network. This was done to eliminate network mask guessing, which may be quite inaccurate on subnetted networks. Obviously, the best case scenario is to intercept a DHCP reply.

The `get_iprange()` function is called every time a data frame is received. It parses the Ethernet header looking for ARP and IP packets. When an ARP packet is received `process_arp()` is called, an IP address extracted, and, if it is outside discovered range, the network data structure is updated with the new IP address. The `process_ip()` function is called in search of new IP addresses and DHCP replies. When a DHCP reply is found it is parsed and the client's IP address assigned by the DHCP server is extracted; furthermore, the DHCP options field is parsed in search for netmask, default gateway, and domain name. All other DHCP options are ignored and skipped.

CHAPTER VIII

CONCLUSIONS

This study achieved its goal of learning details of the IEEE 802.11 Standard and building a wireless network detector for handheld and mobile computers. The software created surpasses its predecessors and similar concurrent projects in a variety of ways. These include, for example, more precise IP range detection and enumeration of wireless and even wired nodes. This allows easier location of misconfigured or "rogue" Access Points and indicates other wireless network insecurities. It has become a rather useful auditing tool featured on websites such as http://www.freshmeat.net and http://www.zauruszone.com with many users.

This study confirmed that IEEE 802.11 Standard compliant devices give out a lot of network information that should be kept private. This is true even when 802.11 networks are used with Wired Equivalent Privacy (WEP) to encrypt the data traffic. The study also proved that anyone with the right equipment and motivation can learn a lot about one's wireless IEEE 802.11b network and possibly misuse the data.

Future Research

The software produced during this research only works with IEEE 802.11b compliant networks simply because only 802.11b open source drivers were available at the time of writing. There exist several variations of 802.11 (e.g. 802.11a and 802.11g). The software can easily be adapted to detect these wireless networks as well, with very little or no modifications. This will be a simple task because the software relies on IEEE

802.11 frame parsing The frame format (Data Link Layer of OSI) between these variations stays the same while type of modulation and carrier frequency (Physical Layer of OSI) change. The device drivers for the other IEEE 802.11 variations should also support the monitor mode.

Recommended improvements to the software include making display of network list more readable, adding card specific statistics and parameters extraction, returning the card into normal mode from RFMON after sniffing, and ability to save the results of auditing. But these are basically cosmetic changes and improvements.

APPENDIX A

Prior Research and Systems Built

There exist several projects that deal with detection and sniffing wireless networks. The most widely known in the wireless community are Netstumbler, Airsnort, and Kismet.

Netstumbler [13] (http://www netstumbler.com) is an active network detector written for MS Windows. It actively tries to join any wireless network in the range and, if succeeds, extracts and displays the network information: MAC, SSID, first-seen and last-seen timestamps, network type, WEP flag, signal quality, noise, SNR (signal-to-noise ration), and, if GPS (global positioning system) is available, the longitude/latitude. There is a less functional PDA version of it for MS Windows CE 3.0 called Ministumbler. Netstumblers only work with Orinoco chipset based cards because they use firmware features of these cards to send out probe requests and read probe responses from the AP. Netstumbler is a closed-source project.

Airsnort [1] (http://airsnort.shmoo.com) was designed primarily for WEP cracking and passively collecting cryptographically weak packets for future cracking. It, however, displays some basic wireless network information such as SSID, BSSID, channel number, and number of packets received. After it collects enough "weak" packets it computes and displays the shared key of the network. Airsnort is a passive sniffer and works with most cards. It runs on Linux/BSD platforms.

Kismet [10] (http://www.kismetwireless.net) is a passive wireless network monitoring tool. It is rich in features and can be used as a network detector or for dumping all network traffic to a file for later analysis. It has all features of Netstumbler

plus its advanced features include network IP range detection and saving files in ethereal compatible format for later viewing or information extraction. Kismet is a passive sniffer and works with the majority of wireless network cards. Kismet runs on Linux/BSD platforms.

Commercial tools include Network Associates' Wireless Sniffer. There is a PDA Windows version of it but, unfortunately, it could not be tested due to cost involved.

IBM Corp. also built a research prototype of an 802.11 wireless LAN security auditor called Wireless Security Auditor (WSA) [8] described at their web site http://www.research.ibm.com/gsal/wsa/. The prototype information is limited and it is a closed source project. It is not known whether it is active or passive from the description provided on the IBM's website.

The main reason that most systems, including the one described here, are built for Linux and BSD operating systems is mainly because there are no available MS Windows drivers that can do RF monitor mode.

Advantages/Disadvantages of the Predecessors

The software packages described above are all excellent tools. Many features that initially were to appear in this package were not available in the other software packages. For example, IP range detection, MAC addresses of wired nodes detection, cloaked network's SSID detection were features, which surfaced at approximately the same time. The tools described have their advantages as well as disadvantages.

For example, Netstumbler only runs on MS Windows and only supports Orinoco based PCMCIA cards. It does not support other types of cards. It can only do active

network detection, which involves generating traffic that can potentially be detected. The network should be reachable by the Netstumbler's radio to generate a response, thus the range of Netstumblers is limited to its radio power. This is not the case with passive network detectors, however, and the major limiting factor for them is the wireless card radio sensitivity.

The goal of the Airsnort project is encryption key retrieval and involves collection of up to 5-10 million packets, which may not be suitable for a handheld device due to storage constraints. It is also linked with GTK+ library, which means that this library needs to be installed on the handheld taking up space.

Kismet is a console-based system primarily designed to run on Linux laptops. In the author's experience, it is extremely difficult to run Kismet on a handheld. It, however, has a kismet_qt front-end (http://www.livejournal.com/users/mspin) that appeared recently and may be used to run kismet on a handheld device with Linux and QPE installed.

APPENDIX B

```
/*
 * discoverer/packet.h
 *
 * Copyright (C) 2002 Alex Medvedev <alexm@pycckue.org>
 *
 * header file for the capture.c and discoverer interface
 *
 */

#ifndef ALEXM_PACKET_H
#define ALEXM_PACKET_H

#include <ctype.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdint.h>
#include <inttypes.h>
#include <string.h>
#include <netinet/ip.h>

#define SSID_SIZE 32
#define MAC_ADDR_LEN 6
#define MAX_PACKET_LEN 8192
#define WLAN_DEVNAMELEN_MAX 6
#define MANUFILE "/etc/manufacturers.dat"
#define NETWORKCOUNT 100
#define MANUFAC_LEN 64

#define CAPABILITY_OFFSET 34
#define SSID_OFFSET 36

/* 802.11 packet frame header */
typedef struct {
        unsigned short version : 2;
        unsigned short type : 2;
        unsigned short subtype : 4;
        unsigned short to_ds : 1;
        unsigned short from_ds : 1;
        unsigned short more_fragments : 1;
        unsigned short retry : 1;
        unsigned short power_management : 1;
        unsigned short more_data : 1;
```

```
        unsigned short wep : 1;
        unsigned short order : 1;
} frame_control_t;

/* 802.11b general mac header */
typedef struct {
        //frame_control_t fc;                   // 2
      // for some reason sizeof() miscalculates the size of this struct
      // have to use offset 2 in the process_beacon()
        unsigned short duration_id;             // 2
        uint8_t address1[MAC_ADDR_LEN];         // 6
        uint8_t address2[MAC_ADDR_LEN];         // 6
        uint8_t address3[MAC_ADDR_LEN];         // 6
        unsigned short seq_control;             // 2
        uint8_t address4[MAC_ADDR_LEN];         // 6
} general_mac_hdr_t;

typedef struct {
        uint8_t id;
      uint8_t len;
        uint8_t channel;
} channel_t;

typedef struct {
        uint8_t id;
        uint8_t len;
        char str[32];
} ssid_t;

typedef struct {
        unsigned short ess : 1;
        unsigned short ibss : 1;
        unsigned short cf_pollable : 1;
        unsigned short cf_poll_req : 1;
        unsigned short privacy : 1;
        uint8_t reserved[11];
} capability_t;

/* to count the number of clients */
struct client_list {
        uint8_t bssid[6];
        struct client_list *next;
};
typedef struct client_list client_list_t;

typedef struct {
```

```
        struct in_addr lo_ip; // lowest ip found
        struct in_addr hi_ip; // highest ip found
        struct in_addr gw; //gateway
        struct in_addr netmask; // netmask
        char domainname[32]; // limit it to 32 char
} ip_info_t;

/* final structure */
struct net_info_t {
        char ssid[SSID_SIZE];
        uint8_t wep;
        uint8_t bssid[6];
        uint8_t ap;
        uint8_t channel;
        char manufacturer[MANUFAC_LEN];
        int packet_count;
        int n_wclients; // wireless clients
        int n_lclients; // all nodes in the network
        client_list_t *clientw;
        client_list_t *clientl;
        ip_info_t iprange;
        struct net_info_t *next;
};
typedef struct net_info_t network_info_t;

typedef struct {
        uint8_t id;
      uint8_t len;
        uint8_t supp_rates[8];
} supported_rates_t;

struct my_arphdr {
        unsigned short int ar_hrd;        /* Format of hardware address.  */
        unsigned short int ar_pro;        /* Format of protocol address.  */
        unsigned char ar_hln;             /* Length of hardware address.  */
        unsigned char ar_pln;             /* Length of protocol address.  */
        unsigned short int ar_op;         /* ARP opcode (command).  */
        unsigned char ar_sha[6];   /* Sender hardware address.  */
        unsigned char ar_sip[4];        /* Sender IP address.  */
        unsigned char ar_tha[6];   /* Target hardware address.  */
        unsigned char ar_tip[4];        /* Target IP address.  */
};

typedef struct {
        u_int8_t op;
        u_int8_t htype;
```

```
        u_int8_t hlen;
        u_int8_t hops;
        u_int32_t xid;
        u_int16_t secs;
        u_int16_t flags;
        struct in_addr ciaddr;
        struct in_addr yiaddr;
        struct in_addr siaddr;
        struct in_addr giaddr;
        unsigned char chaddr[16];
        char sname[64];
        char file[128];
        //unsigned char options[1222]; // max = 1500-(236+14+20+8)
} dhcp_msg;

#endif
```

```
/*
 * discoverer/main.c
 *
 * Copyright (C) 2002 Alex Medvedev <alexm@pycckue.org>
 *
 * main function for discoverer
 *
 */

#include "discoverer.h"
#include <qpe/qpeapplication.h>


int main( int argc, char ** argv )
{
   QPEApplication app( argc, argv );

   Discoverer d;
   app.showMainWidget( &d );

   return app.exec();
}
```

```
/*
 * discoverer/capture.c
 *
 * Copyright (C) 2002 Alex Medvedev <alexm@pycckue.org>
 * http://www.cs.swt.edu/~am60347
 *
 * libpcap capture of raw frame, their analysis, and sending results to
 * the user interface process
 *
 */

#include "packet.h"

#include <signal.h>
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>
#include <stdarg.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
#include <netinet/in.h>
#include <net/ethernet.h>
#include <netinet/ether.h>

typedef struct pcap_pkthdr pcap_pkthdr;
/* globals for pcap capture */
pcap_pkthdr callback_header;
u_char callback_data[MAX_PACKET_LEN];
network_info_t *net = NULL; /* head pointer for list of networks */
int n_discovered = 0; /* number of networks discovered so far */
int g_packet_count = 0; /* count all packets */

/* struct to pass to IPC */
typedef struct {
        long int message_type;
        network_info_t data;
} ui_message;
```

```
/* inserts new network in the head of the list */
void head_insert(network_info_t *n)
{
        /* act as a constructor too */
        /* these are default values */
        n->iprange.lo_ip.s_addr = 0x00000000;
        n->iprange.hi_ip.s_addr = 0x00000000;
        n->iprange.gw.s_addr = 0x00000000;
        n->iprange.netmask.s_addr = 0xffffffff;

        /* the main function is to insert the new network in the list */
        n->next = net;
        net = n;
}

/*
 * formatted messaging to the UI module
 */
void log ( const char *fmt, ...) /* print into a message queue "2002" */
{
        char str[64];
        va_list argp;
        int qid;
        ui_message msg;

        va_start( argp, fmt);
        vsnprintf( str, 64, fmt, argp);
        va_end( argp);

        memcpy( &(msg.data), str, sizeof(network_info_t));
        qid = msgget( (key_t)2002, 0600 | IPC_CREAT);
        if ( qid == -1) {
                fprintf( stderr, "msgget failed, errno = %d\n", errno);
                return;
        }
        msg.message_type = 3; /* 3 means informational message */
        if ( msgsnd( qid, (void *)&msg, sizeof(msg), 0) == -1) {
                fprintf( stderr, "msgsnd failed\n");
                return;
        }
}

/*
 * searches for a BSSID in the list of already found networks.
 * returns 0 if bssid not found in the list, 1 otherwise
```

```
 */
int bssid_found( network_info_t *n)
{
        network_info_t *temp = net;
        int found = 0;

        if ( temp == NULL )
                return 0;
        while ( temp != NULL) { /* compare from the end -- faster */
                /* if ( temp->bssid[5] == n->bssid[5] &&
                    temp->bssid[4] == n->bssid[4] &&
                    temp->bssid[3] == n->bssid[3] &&
                    temp->bssid[2] == n->bssid[2] &&
                    temp->bssid[1] == n->bssid[1] &&
                    temp->bssid[0] == n->bssid[0] ) {
                */
                /* this if replaces the one above (readability)*/
                if ( memcmp(temp->bssid, n->bssid, 6) == 0) {
                        found = 1;
                        /* just in case if this is a reassoc req */
                        if ( temp->ssid == NULL && n->ssid != NULL) {
                                strncpy( temp->ssid, n->ssid, 32);
                                log( "found ssid=%s via re-assoc request",
                                        temp->ssid);
                        }
                        temp->packet_count++;
                        break; // exit from while()
                }
                temp = temp->next;
        }
        if (found)
                return 1;
        return 0;
}
/*
 * pretty print function
 * constructs and sends messages of type new
 */
void pprint (network_info_t *n) /* print into a message queue "2002" */
{
        int qid;
        ui_message msg;

        memcpy(&(msg.data), n, sizeof(network_info_t));
        qid = msgget( (key_t)2002, 0600 | IPC_CREAT);
        if ( qid == -1) {
```

```
                fprintf( stderr, "msgget failed, errno = %d\n", errno);
                return;
        }
        msg.message_type = 1;
        if ( msgsnd( qid, (void *)&msg, sizeof(msg), 0) == -1) {
                fprintf( stderr, "msgsnd failed\n");
                return;
        }
        //fprintf( stderr, "sent about %s\n", n->ssid);
}

/*
 * updates user interface with ever changing network information
 * like packet count, new ip addresses, ...
 */
void ui_update()
{
        int qid;
      network_info_t *temp = net;
        ui_message msg;

        alarm(1);
        //fprintf(stderr, "alarm\n");
        while( temp != NULL) {
                memcpy(&(msg.data), temp, sizeof(network_info_t));
                qid = msgget( (key_t)2002, 0600 | IPC_CREAT);
                if ( qid == -1) {
                        fprintf( stderr, "msgget failed, errno = %d\n", errno);
                        return;
                }
                msg.message_type = 2;
                if ( msgsnd( qid, (void *)&msg, sizeof(msg), 0) == -1) {
                        fprintf( stderr, "msgsnd failed\n");
                        return;
                }
                //fprintf( stderr, "sent about %s\n", temp->ssid);
                temp = temp->next;
        }
}

/*
 * checks the manufacturer datafile for a match
 */
void lookup_manufac (general_mac_hdr_t *mac, char *name)
{
        FILE *fd;
```

```
        char mac_str[9];
        char str[80];
        int i;
        int j=0;
        int found = 0;

        strcpy(name, "unknown"); /* init name[] */
        snprintf(mac_str, 9, "%02x:%02x:%02x", mac->address3[0], mac-
>address3[1], mac->address3[2]);
        for (i=0;i<8;i++)
                mac_str[i] = toupper(mac_str[i]);

        if ( (fd = fopen(MANUFILE, "r")) == 0) {
                perror("manufacturers file: ");
        } else {
                while ( fgets(str, 80, fd)) {
                        j++;
                        for(i=0;i<8;i++)
                                str[i] = toupper(str[i]);
                        if ( !strncmp(str, mac_str, 8)) { /* if first 8 chars match */
                                strncpy(name, &str[9], 80);
                                name[strlen(name)-1] = '\0'; /* overwrite \n with NULL
*/
                                found = 1;
                                break;
                        }
                }
        }
        if ( fd) fclose(fd);
}

/*
 * disects a beacon frame
 */
void process_beacon (const u_char *data)
{
        general_mac_hdr_t *machdr;
        capability_t *cap;
        char str[33];
        ssid_t *ssid;
        channel_t *chan;
        int channel_offset;
        supported_rates_t *supp_rates;
        int supp_rates_offset;
        char manuf[MANUFAC_LEN];
        network_info_t *cur_net;
```

```
if ( !(cur_net = (network_info_t *) malloc(sizeof(network_info_t))) ) {
        log( "malloc() failed\ncapture module terminated");
        exit (1);
}

cur_net->packet_count = 1; /* like in the constructor */
/* 1 is because we got THIS packet */
//printf("%s\n", "beacon");
ssid = (ssid_t *)(&data[SSID_OFFSET]);
strncpy(str, ssid->str, ssid->len);
str[ssid->len] = '\0';
//printf("ssid->str = %s\n", str);
strncpy(cur_net->ssid, str, SSID_SIZE);

machdr = (general_mac_hdr_t *) &data[2];
memset(manuf, 0, sizeof(manuf));
//memcpy(cur_net->bssid, machdr->address3, sizeof(machdr->address3));
memcpy(cur_net->bssid, &data[16], sizeof(machdr->address3));
cap = (capability_t *) (&data[CAPABILITY_OFFSET]);
if ( cap->privacy == 1 ) {
        cur_net->wep = 1;
        //puts("WEP is ON");
} else {
        cur_net->wep = 0;
        //puts("WEP is OFF");
}
if ( cap->ess && !cap->ibss) {
        cur_net->ap = 1;
        //puts("AP");
}
else if ( !cap->ess && cap->ibss) {
        cur_net->ap = 0;
        //puts("AdHoc");
}
else {
        cur_net->ap = 2;
        //puts("unknown network type");
}

supp_rates_offset = SSID_OFFSET + 2*sizeof(uint8_t) + ssid->len;
supp_rates = (supported_rates_t *) (&data[supp_rates_offset]);

channel_offset = supp_rates_offset + 2*sizeof(uint8_t) +
                supp_rates->len;
chan = (channel_t *) (&data[channel_offset]);
```

```
        cur_net->channel = chan->channel;
        //printf("channel = %d\n", chan->channel);
        if ( ! bssid_found(cur_net)) {
                /* call lookup_manuf() here, to minimize fopen() calls */
                lookup_manufac(machdr, manuf); /* does not work */
                strncpy(cur_net->manufacturer, manuf, MANUFAC_LEN);
                n_discovered++;
                head_insert(cur_net);
                pprint(cur_net);
        } else {
                free(cur_net);
        }
}

/*
 * disects a reassociation request
 */
void process_reassoc_req( const u_char *data)
{
/* capability[16]
 * listen_interval[16]
 * current_AP_addr[48]
 * ssid[16-272]
 * supp_rates[32-80]
 */
        general_mac_hdr_t *machdr;
        capability_t *cap;
        char str[33];
        ssid_t *ssid;
        char manuf[MANUFAC_LEN];
        network_info_t *cur_net;
        int ssid_offset = 34;

        if ( !(cur_net = (network_info_t *) malloc(sizeof(network_info_t))) ) {
                log( "malloc() failed. capture exited\n");
                exit (2);
        }


        cur_net->packet_count = 1; /* just in case if it is a new net */

        cap = (capability_t *) (data);
        if ( cap->privacy == 1 ) {
                cur_net->wep = 1;
        } else {
                cur_net->wep = 0;
```

```
        }
        if ( cap->ess && !cap->ibss) {
                cur_net->ap = 1;
        }
        else if ( !cap->ess && cap->ibss) {
                cur_net->ap = 0;
        }
        else {
                cur_net->ap = 2;
        }

        ssid = (ssid_t *)(&data[ssid_offset]);
        strncpy(str, ssid->str, ssid->len);
      str[ssid->len] = '\0';
      strncpy(cur_net->ssid, str, SSID_SIZE);
        log( "got ssid %s from re-assoc req", ssid);

        /* get current AP address 6 bytes */
        memcpy(cur_net->bssid, &data[28], sizeof(machdr->address3));

        /* can't get channel info so set it to 15 (does not exist) */
        cur_net->channel = 15;

        if ( ! bssid_found(cur_net)) {
            /* call lookup_manuf() here, to minimize fopen() calls */
            lookup_manufac(machdr, manuf);
            strncpy(cur_net->manufacturer, manuf, MANUFAC_LEN);
            n_discovered++;
            head_insert(cur_net);
            pprint(cur_net);
      } else {
            free(cur_net);
      }

}

/*
 * traverses wireless clients list
 * if not found adds as new
 */
int check_lclient( uint8_t *cur_bssid, uint8_t *mac_to_check)
{
        network_info_t *temp = net;
        client_list_t *cl = NULL;
        client_list_t *cl_temp;
      int found = 0;
```

```
        //log("in check_client()");
      if ( temp == NULL )
            return 0;
        cl_temp = temp->clientl;
      while ( temp != NULL) {
              if ( memcmp(temp->bssid, cur_bssid, 6) == 0) {
                    //log("found bssid");
                temp->packet_count++;
                    if ( temp->n_lclients == 0) {
                          //log("empty list");
                          temp->n_lclients++;
                          if ( !(cl = (client_list_t *)malloc(\
                                sizeof(client_list_t)))) {
                                log ("out of memory");
                                exit(3);
                          }
                          cl->next = temp->clientl;
                          temp->clientl = cl;
                          memcpy(cl->bssid, mac_to_check, 6);
                          log( "new node in %s:
mac=%02x:%02x:%02x:%02x:%02x:%02x\n",
                                temp->ssid,
                                mac_to_check[0], mac_to_check[1],
                                mac_to_check[2], mac_to_check[3],
                                mac_to_check[4], mac_to_check[5]);
                          return 2; // means that inserted a new one
                    } else {
                          cl_temp = temp->clientl;

                          while ( cl_temp != NULL) {
                                if ( memcmp(cl_temp->bssid,
                                      mac_to_check, 6) == 0) {
                                      found = 1;
                                }
                                cl_temp = cl_temp->next;
                          }
                          if ( found == 0) {
                                cl_temp = temp->clientl;
                                if ( !(cl = (client_list_t *)\
                                malloc(sizeof(client_list_t)))) {
                                      log ("out of memory");
                                      exit(3);
                                }
                                log( "new node in %s:
mac=%02x:%02x:%02x:%02x:%02x:%02x\n",
```

```
                                        temp->ssid,
                                        mac_to_check[0], mac_to_check[1],
                                        mac_to_check[2], mac_to_check[3],
                                        mac_to_check[4], mac_to_check[5]);
                                        cl->next = temp->clientl;
                                        temp->clientl = cl;
                                        memcpy(cl->bssid, mac_to_check, 6);
                                        temp->n_lclients++;
                                        return 2;
                                }
                        }
                found = 0;
                }
            temp = temp->next;
        }
        return 0; // have not found anything
}

/*
 * same as above but for wired clients
 */
int check_wclient( uint8_t *cur_bssid, uint8_t *mac_to_check)
{
        network_info_t *temp = net;
        client_list_t *cl = NULL;
        client_list_t *cl_temp;
    int found = 0;

        //log("in check_client()");
    if ( temp == NULL )
        return 0;
        cl_temp = temp->clientw;
    while ( temp != NULL) {
            if ( memcmp(temp->bssid, cur_bssid, 6) == 0) {
                    //log("found bssid");
                temp->packet_count++;
                    if ( temp->n_wclients == 0) {
                            //log("empty list");
                            temp->n_wclients++;
                            if ( !(cl = (client_list_t *)malloc(\
                                    sizeof(client_list_t)))) {
                                    log ("out of memory");
                                    exit(3);
                            }
                            cl->next = temp->clientw;
                            temp->clientw = cl;
```

```
                        memcpy(cl->bssid, mac_to_check, 6);
                        log( "new wireless client in %s:
mac=%02x:%02x:%02x:%02x:%02x:%02x\n",
                                temp->ssid,
                                mac_to_check[0], mac_to_check[1],
                                mac_to_check[2], mac_to_check[3],
                                mac_to_check[4], mac_to_check[5]);
                        return 2; // means that inserted a new one
                } else {
                        cl_temp = temp->clientw;

                        while ( cl_temp != NULL) {
                                if ( memcmp(cl_temp->bssid,
                                        mac_to_check, 6) == 0) {
                                        found = 1;
                                }
                                cl_temp = cl_temp->next;
                        }
                        if ( found == 0) {
                                cl_temp = temp->clientw;
                                if ( !(cl = (client_list_t *)\
                                malloc(sizeof(client_list_t)))) {
                                        log ("out of memory");
                                        exit(3);
                                }
                                log( "new wireless client in %s:
mac=%02x:%02x:%02x:%02x:%02x:%02x\n",
                                temp->ssid,
                                mac_to_check[0], mac_to_check[1],
                                mac_to_check[2], mac_to_check[3],
                                mac_to_check[4], mac_to_check[5]);
                                cl->next = temp->clientw;
                                temp->clientw = cl;
                                memcpy(cl->bssid, mac_to_check, 6);
                                temp->n_wclients++;
                                return 2;
                        }
                }
                found = 0;
                }
                temp = temp->next;
        }
        return 0; // have not found anything
}

/*
```

```
 * returns true if mac is 0xffffffff
 * makes code less messy
 */
int check_for_broadcast( uint8_t *mac_to_check)
{
        uint8_t bcast[6];

        memset(bcast, 0xff, 6);
        if ( memcmp(mac_to_check, bcast, 6) == 0)
                return 1;
        else
                return 0;
}

/*
 * inserts a new ip address in the network struct
 */
void insert_new_ip( const u_int8_t bssid[6], const u_int32_t ip,
                        const u_int32_t gw, const u_int32_t netmask,
                        const char domainname[])
{
        network_info_t *temp;
        u_int32_t addr = 0;

        //puts("in insert_new_ip()");

        if ( !(temp = (network_info_t *) malloc(sizeof(network_info_t))) ) {
                log( "malloc() failed. capture exited\n");
                exit (2);
        }
        memset(temp, 0, sizeof(network_info_t));
        memcpy(temp->bssid, bssid, 6);

        if ( ! bssid_found(temp)) {
                log("unknown data frame");
                return; // we can't insert it because this bssid does not exist
        }

        temp = net;
        while ( temp != NULL) {
                if ( memcmp(temp->bssid, bssid, 6) == 0) {
                        //puts("found bssid match");
                        addr = temp->iprange.lo_ip.s_addr;
                        if ( addr == 0 && ip != 0xffffffff) {
                                temp->iprange.lo_ip.s_addr = ip;
                                temp->iprange.hi_ip.s_addr = ip;
```

```
                        }
                        if ( ip != 0 && ip != 0xffffffff && ip < addr) {
                                // we got a new low
                                temp->iprange.lo_ip.s_addr = ip;
                        }
                        addr = temp->iprange.hi_ip.s_addr;
                        if ( ip != 0 && ip != 0xffffffff && ip > addr) {
                                // we got a new high
                                temp->iprange.hi_ip.s_addr = ip;
                                //puts("new hi");
                        }
                        if ( gw != 0) {
                                temp->iprange.gw.s_addr = gw;
                        }
                        if ( netmask != 0) {
                                temp->iprange.netmask.s_addr = netmask;
                        }
                        if ( domainname != NULL) {
                                strncpy(temp->iprange.domainname, domainname,
32);
                        }
                        break; // we are done, exit while loop
                }
                temp = temp->next;
        }

}

/*
 * processes arp packets
 * extracts ip and mac addresses
 */
void process_arp( const u_int8_t bssid[6], const u_char *data, int body_offset)
{
        struct my_arphdr *arpheader;
        u_int32_t s, d;
        char s_addr_str[16];
        char d_addr_str[16];

        arpheader = (struct my_arphdr *) &data[body_offset+14];

        snprintf(s_addr_str, 16, "%d.%d.%d.%d",
                arpheader->ar_sip[0], arpheader->ar_sip[1],
                arpheader->ar_sip[2], arpheader->ar_sip[3]);
        snprintf(d_addr_str, 16, "%d.%d.%d.%d",
                arpheader->ar_tip[0], arpheader->ar_tip[1],
```

```
                arpheader->ar_tip[2], arpheader->ar_tip[3]);

        //printf("s_addr = %s, d_addr = %s\n", s_addr_str, d_addr_str);
        inet_pton(AF_INET, s_addr_str, &s);
        inet_pton(AF_INET, d_addr_str, &d);
        //printf("s = %d d = %d\n", s, d);
        if ( s>>24 != 0x00 && s>>24 != 0xff)
                insert_new_ip( bssid, s, 0, 0, 0);
        if ( d>>24 != 0x00 && d>>24 != 0xff)
                insert_new_ip( bssid, d, 0, 0, 0);

        /*
        log("ip addresses via arp: src=%d.%d.%d.%d, dst=%d.%d.%d.%d\n",
                        arpheader->ar_sip[0], arpheader->ar_sip[1],
                        arpheader->ar_sip[2], arpheader->ar_sip[3],
                        arpheader->ar_tip[0], arpheader->ar_tip[1],
                        arpheader->ar_tip[2], arpheader->ar_tip[3]);
        */
}

/*
 * processes ip traffic, mainly looking for DHCP replies
 * once a DHCP reply is found -- parse options field, extract all good info
 */
void process_ip( const u_int8_t bssid[6], const u_char *data, int body_offset)
{
        struct iphdr *ipheader;
        struct tcphdr *tcpheader; // can be tcp or udp we do not care here
        int ip_data_offset;
        int tcp_data_offset;
        struct in_addr s, d;
        char domainname[255];
        //network_info_t *n;
        dhcp_msg *dhcp;
        int option, offset, i;

        ip_data_offset = body_offset + sizeof(struct ether_header);
        tcp_data_offset = ip_data_offset + sizeof(struct iphdr);

        ipheader = (struct iphdr *) &data[ip_data_offset];
        tcpheader = (struct tcphdr *) &data[tcp_data_offset];

        s.s_addr = ipheader->saddr;
        d.s_addr = ipheader->daddr;

        if ( ipheader->saddr>>24 != 0x00 && ipheader->saddr>>24 != 0xff) {
```

```
                insert_new_ip(bssid, s.s_addr, 0, 0, 0);
        }
        if ( ipheader->daddr>>24 != 0x00 && ipheader->daddr>>24 != 0xff) {
                insert_new_ip(bssid, d.s_addr, 0, 0, 0);
        }
        if ( ntohs(tcpheader->dest) == 68) {
                /* process dhpcd response */
                dhcp = (dhcp_msg *) &data[ip_data_offset+sizeof(struct iphdr)+8];
                memcpy(&s, &dhcp->yiaddr, sizeof(struct in_addr));
                log("yiaddr dhcp reply in %02x:%02x:%02x:%02x:%02x:%02x: %s",
                        bssid[0], bssid[1], bssid[2], bssid[3],
                        bssid[4], bssid[5], inet_ntoa(s));

                /* parse dhcp options */
                option = -1;   // dhpcd option
                offset = ip_data_offset+sizeof(struct iphdr)+8+sizeof(dhcp_msg)+4;
                printf("offset = %d\n", offset);
                i = 3;          // elements to collect netmask, gateway, domain
                memset(domainname, 0, 255);
                while( i != 0 && option != 0) {
                        option = (u_int8_t) data[offset];
                        printf("option=%d\n", option);
                        if( option == 1) { // netmask
                                memcpy(&s.s_addr, &data[offset+2],
                                        (u_int8_t)data[offset+1]);
                                offset += (u_int8_t)data[offset+1]+2;
                                i--;
                                log("netmask: %s", inet_ntoa(s));
                        }
                        else if( option == 3) { // default gateway
                                memcpy(&d.s_addr, &data[offset+2],
                                        (u_int8_t)data[offset+1]);
                                offset += (u_int8_t)data[offset+1]+2;
                                i--;
                                log("gw address: %s", inet_ntoa(d));
                        }
                        else if( option == 15) { // domainname
                                memcpy(domainname, &data[offset+2],
                                        (u_int8_t)data[offset+1]);
                                offset += (u_int8_t)data[offset+1]+2;
                                i--;
                                log("domainname: %s", domainname);
                        }
                        else {
                                offset += (u_int8_t)data[offset+1]+2;
                                // printf("new offset = %d\n", offset);
```

```
                        }
                }
                insert_new_ip(bssid, 0, d.s_addr, s.s_addr, domainname);
        }
}

/*
 * frontend to process_arp() and process_ip()
 */
void get_iprange( const u_int8_t bssid[6], const u_char *data)
{
        int body_offset = 0;
        frame_control_t *fc;
        general_mac_hdr_t *machdr;
        struct ether_header *ethhead;

        fc = (frame_control_t *) data;
        machdr = (general_mac_hdr_t *) &data[2];

        if ( fc->to_ds == 1 && fc->from_ds == 1)
                body_offset = 24;
        else
                body_offset = 18;

        ethhead = (struct ether_header *) &data[body_offset];

        if ( ntohs(ethhead->ether_type) == ETHERTYPE_ARP) {
                process_arp(bssid, data, body_offset);
        }
        else if( ntohs(ethhead->ether_type) == ETHERTYPE_IP)
                process_ip(bssid, data, body_offset);
        else
                fprintf(stderr, "type 0x%04x\n", ntohs(ethhead->ether_type));


}

/*
 * extracts macs from a data frame
 * puts them into the network struct
 * calls get_iprange()
 */
void process_data_frame( const u_char *data)
{
        frame_control_t *fc;
        general_mac_hdr_t *machdr;
```

```
network_info_t *cur_net;

if ( !(cur_net = (network_info_t *) malloc(sizeof(network_info_t))) ) {
        log( "malloc() failed\ncapture module terminated");
        exit (4);
}

fc = (frame_control_t *) data;

machdr = (general_mac_hdr_t *) &data[2];

if ( fc->to_ds == 0 && fc->from_ds == 0) {
        //log("0,0");
        memcpy(cur_net->bssid, machdr->address3, 6);
        if ( ! check_for_broadcast( machdr->address1))
                (void) check_lclient(cur_net->bssid, machdr->address1);
        if ( ! check_for_broadcast( machdr->address2))
                (void) check_lclient(cur_net->bssid, machdr->address2);
}
else if ( fc->to_ds == 0 && fc->from_ds == 1) {
        //log("0,1");
        memcpy(cur_net->bssid, machdr->address2, 6);
        if ( ! check_for_broadcast( machdr->address1))
                (void) check_lclient(cur_net->bssid, machdr->address1);
        if ( ! check_for_broadcast( machdr->address3))
                (void) check_lclient(cur_net->bssid, machdr->address3);
}
else if ( fc->to_ds == 1 && fc->from_ds == 0) {
        //log("1,0");
        memcpy(cur_net->bssid, machdr->address1, 6);
        if ( ! check_for_broadcast( machdr->address2))
                (void) check_lclient(cur_net->bssid, machdr->address2);
        if ( ! check_for_broadcast( machdr->address3))
                (void) check_lclient(cur_net->bssid, machdr->address3);
}
else if ( fc->to_ds == 1 && fc->from_ds == 1) {
        //log("1,1");
        if ( ! check_for_broadcast( machdr->address1))
                (void) check_lclient(cur_net->bssid, machdr->address1);
        if ( ! check_for_broadcast( machdr->address2))
                (void) check_lclient(cur_net->bssid, machdr->address2);
}

if ( fc->wep == 0) {/* if wep enabled -- nothing to see here */
        /* try getting IP range */
        get_iprange(cur_net->bssid, data);
```

```
        }
    free(cur_net);

}

/*
 * extracts wireless clients mac addresses
 */
void get_wireless_clients( const u_char *data)
{
        general_mac_hdr_t *machdr;

        machdr = (general_mac_hdr_t *) &data[2];

        if ( memcmp(machdr->address1, machdr->address3, 6) != 0) {
                if ( ! check_for_broadcast( machdr->address1)) {
                        /* calling both since wireless client is a node */
                        (void)check_wclient(machdr->address3, machdr->address1);
                        (void)check_lclient(machdr->address3, machdr->address1);
                }
        }
        if ( memcmp(machdr->address2, machdr->address3, 6) != 0) {
                if ( ! check_for_broadcast( machdr->address2)) {
                        /* calling both since wireless client is a node */
                        (void)check_wclient(machdr->address3, machdr->address2);
                        (void)check_lclient(machdr->address3, machdr->address2);
                }
        }
}

/*
 * pcap_pkthdr only contains 3 elements
 * simply print them here
 * helper function
 */
void p_header_analyzer(pcap_pkthdr *header)
{
        printf("timestamp = %ld, caplen = %d, len = %d\n",
                header->ts.tv_sec,
                header->caplen,
                header->len);
}

/*
 * looks in the frame control field of each frame and
 * according to the packet type calls appropriate
```

```
 * processing code
 */
void p_data_analyzer(const u_char* data)
{
        u_char buf[MAX_PACKET_LEN];
        frame_control_t *fc = (frame_control_t *) data;

        g_packet_count++;

        memcpy(buf, data, callback_header.len);

        switch (fc->type) {
                case 0:         //printf("%s: ", "got a management frame");
                        get_wireless_clients(buf);
                        switch (fc->subtype) {
                                case 2: process_reassoc_req(buf);
                                        break;
                                case 5: process_beacon(buf); //probe request
                                        // it looks like a beacon to some point
                                        //log( "probe request");
                                        break;
                                case 8: process_beacon(buf);
                                        break;
                                default: //log("unknown frame subtype. microwave
oven open (?)");
                                         break;
                        }
                        break;
                case 1:         printf("%s\n", "got a control frame");
                        break;
                case 2:         process_data_frame(buf);
                        break;
                default: printf("fc->type was %u\n", fc->type);
                         break;
        }
}

/*
 * pcap callback function
 * copies pkthdr to callback_header -- timestamp, caplen, len;
 * copies packet to callback_data -- the actual packet (data)
 * calls p_header_analyzer() to print out the header
 */
void my_callback(u_char *args, const struct pcap_pkthdr* pkthdr,
                const u_char* packet)
{
```

```
        memset(&callback_header, 0, sizeof(pcap_pkthdr));
        memset(callback_data, 0, MAX_PACKET_LEN);
        memcpy(&callback_header, pkthdr, sizeof(pcap_pkthdr));
        memcpy(callback_data, packet, pkthdr->len);
        p_data_analyzer(callback_data);
}

/*
 * simply catch ctrl-c and act accordingly
 */
void sig_int(int signo)
{
      network_info_t *temp = net;

        puts("caught a SIGINT. terminating...");
        printf("discovered %d networks\n", n_discovered);
        printf("total packets: %d\n", g_packet_count);

      while ( temp != NULL) { /* compare from the end -- faster */
            printf("%s: %d\n", temp->ssid, temp->packet_count);
            temp = temp->next;
      }
        exit(15);
}

/*
 * useful diagnostics when you do not have a terminal
 */
void sig_segv( int signo)
{
        log("capture segfaulted :(");
        exit(11);
}

/*
 * useful diagnostic for monitor mode detection
 * needs to be UNSPEC, while usually it is Ethernet
 */
void check_link_type(pcap_t *descr)
{
        if ( pcap_datalink(descr) == DLT_EN10MB) {
                log("wireless card is not in monitor mode");
                log("does your driver support it? if you use cisco use");
                log("kernel drivers; if you use orinoco, use");
                log("patched pcmcia-cs drivers");
                exit(-1);
```

```
        } else {
                log("link type is not Ethernet. good.");
        }
}

int main(int argc,char **argv)
{
        char dev[6] = "eth0";
        char errbuf[PCAP_ERRBUF_SIZE];
        pcap_t* descr;
        int count = 0;
        struct sigaction sa_new;

        if ( argc > 1)
                strncpy( dev, argv[1], 6);
        //log( "capture: dev = %s\n", dev);

        signal(SIGINT, sig_int);
        //signal(SIGSEGV, sig_segv);

        sa_new.sa_handler = ui_update;
        sigemptyset(&sa_new.sa_mask);
        sa_new.sa_flags = 0;

        sigaction( SIGALRM, &sa_new, 0);

        alarm(2); // call once to init timeout
        descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf);
        if ( descr == NULL ) {
                log( "pcap_open_live(): %s\n", errbuf);
                exit(EXIT_FAILURE);
        } else {
                log( "opened %s ok\n", dev);
        }

        check_link_type(descr);

        count = pcap_loop(descr,-1,my_callback,NULL);
        if ( count < 0 ) {
                log( "pcap_loop(): %s", errbuf);
                log( "need to run \"ifconfig %s up\"\?", dev);
                log( "capture exited gracefully");
                log( "must restart discoverer manually");
                exit(-1);
        }
```

```
        return EXIT_SUCCESS;
}
```

```
/*
 * discoverer/discoverer.h
 *
 * Copyright (C) 2002 Alex Medvedev <alexm@pycckue.org>
 *
 * derived class from discovererbase.h
 *
 */

#ifndef DISCOVERER_H
#define DISCOVERER_H

#include "discovererbase.h"
#include "packet.h"

#include <fcntl.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib h>
#include <errno.h>
#include <signal.h>

#define CONFIGFILE "/etc/discoverer.conf"

struct n_info_t {
        char ssid[32];
        uint8_t wep;
        uint8_t bssid[6];
        uint8_t ap;
        uint8_t channel;
        char manufacturer[64];
        int packet_count;
        int n_wclients;
        int n_lclients;
        client_list_t *clientw;
        client_list_t *clientl;
        ip_info_t iprange;
        struct n_info_t *next;
};

typedef struct {
        long int message_type;
        n_info_t data;
} ui_message;
```

```
class final_list_t {
public:
        QListViewItem *entry;
        final_list_t *next;
};

class Discoverer : public DiscovererBase
{
        Q_OBJECT
public:
        Discoverer( QWidget *parent = 0, const char *name = 0 );
        ~Discoverer();

protected:
        void load_config();
        void save_config();
        bool save_captured();
        void cisco_monitor();
        void orinoco_hopper();
        void hopper();
        int fork_capture();
        void alles_kaput();

        char devname[6];
        pid_t capture_pid;
        pid_t hopper_pid;
        final_list_t *L;
        QListViewItem* add(n_info_t n);
        void getnetinfo(n_info_t *n);
        void log( const char *fmt, ...);
        void dsp_net_count( const int i);
        void update(n_info_t *n);
        void update_ip_range(QListViewItem *item, n_info_t *n);
        int amiroot();
        void add_mac( QListViewItem *item, client_list_t *list);

public slots:
        virtual void timerEvent( QTimerEvent *event);
        virtual void save_clicked();
        virtual void defaults_clicked();
};

#endif
```

```
/*
 * discoverer/discoverer.cpp
 *
 * Copyright (C) 2002 Alex Medvedev <alexm@pycckue.org>
 *
 * implementation of discoverer.h and main user interface code
 *
 */

#include "discoverer.h"
#include <stdarg.h>
#include <qmessagebox.h>
#include <qdatetime.h>
#include <qlistview.h>
#include <qtextview.h>
#include <qtabwidget.h>
#include <qcombobox.h>
#include <qspinbox.h>
#include <qheader.h>
#include <qstring.h>
#include <string>
#include <vector>
#include <map>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define UNDEF 0
#define CISCO 1
#define ORINOCO 2
#define PRISM2 3
int hopper_req = 0;

/*
 * Constructor, also decides what card is selected and whether hopper
 * is needed.
 */
Discoverer::Discoverer( QWidget *parent = 0, const char *name = 0 ) :
DiscovererBase( parent, name)
{
        if ( amiroot()) {
                L = NULL; // network linked list empty
                ListView->setSorting(-1, TRUE); // do not sort list
                load_config();

                if ( DevtypeComboBox->currentItem() == CISCO ) {
```

```
                log( "card type selected: cisco");
                hopper_req = 0; // for clarity
                cisco_monitor();
        }
        else if ( DevtypeComboBox->currentItem() == ORINOCO) {
                log( "card type selected: orinoco");
                hopper_req = 1;
                orinoco_hopper();
        }
        else if ( DevtypeComboBox->currentItem() == PRISM2) {
                log( "card type selected: prism2");
                hopper_req = 1;
                orinoco_hopper();
        }
        else if ( DevtypeComboBox->currentItem() == UNDEF) {
                ;
        }
        if ( (fork_capture()) != -1)
                log("starting capture on %s",
                        (DevnameComboBox->currentText()).latin1());
    } else {
        qWarning("you are not root");
    }
}

/*
 * destructor, destroyes the message queue, and kills all procs
 */
Discoverer::~Discoverer()
{
    int qid;

    qid = msgget( (key_t)2002, 0600 | IPC_CREAT);
    if ( msgctl( qid, IPC_RMID, 0) == -1) {
        fprintf( stderr, "couldn't destroy msg, run ipcrm manually");
    }
    alles_kaput();
}

/*
 * loads values from the configuration file
 */
void Discoverer::load_config()
{
    FILE *f;
    int name, type, interval;
```

```
        char c;

        if( !(f = fopen(CONFIGFILE, "r"))) {
                defaults_clicked();
                return;
        }
        fscanf(f, "%d%c%d%c%d", &name, &c,
                                &type, &c,
                                &interval);
        DevnameComboBox->setCurrentItem(name);
        DevtypeComboBox->setCurrentItem(type);
        HopintervalSpinBox->setValue(interval);
}

/*
 * channel hopper, starts the timer using value from the drop down menu
 */
void Discoverer::orinoco_hopper()
{
        char cmd[40];

        snprintf(cmd, 40, "%s %s %s", "/sbin/ifconfig",
                (DevnameComboBox->currentText()).latin1(), "promisc up");
        system(cmd);

        /* starts timer for timerEvent() */
        startTimer( HopintervalSpinBox->value());
}

/*
 * called when the timer goes off (the value in the drop down menu)
 */
void Discoverer::timerEvent( QTimerEvent *event)
{
        n_info_t *n;
        // calls hopper() every time timer hits
        // log("timerEvent was called");
        n = new n_info_t;
        if ( hopper_req == 1)
                hopper();
        getnetinfo(n);
        while ( n->channel > 0 && n->channel < 15) {
                log("new network, ssid=%s", n->ssid);
                add(*n);
                //puts("in while");
                getnetinfo(n);
```

```
        }
}

/*
 * saves the current configuration to the config file
 */
void Discoverer::save_clicked()
{
        FILE *f;
        f = fopen(CONFIGFILE, "w");
        fprintf(f, "%d,%d,%d", DevnameComboBox->currentItem(),
                                DevtypeComboBox->currentItem(),
                                HopintervalSpinBox->value()
                                );
        fclose(f);
}

/*
 * resets all drop down menus to default values
 */
void Discoverer::defaults_clicked()
{
        HopintervalSpinBox->setValue(30);
        DevnameComboBox->setCurrentItem(0);
        DevtypeComboBox->setCurrentItem(0);
}

/*
 * enables cisco monitor mode
 */
void Discoverer::cisco_monitor()
{
        FILE *f;
        char path[100];
        char cmd[32];

        snprintf(path, 100, "%s%s%s", "/proc/driver/aironet/",
                DevnameComboBox->currentText().latin1(), "/Config");
        if ( !(f = fopen(path, "w")) ) {
                log( "Can't open Cisco Config file %s", path);
                return;
        }
        fprintf(f, "%s", "Mode: r");
        fprintf(f, "%s", "Mode: y");
        fprintf(f, "%s", "XmitPower: 1");
        fclose(f);
```

```
        snprintf(cmd, 32, "%s %s %s", "/sbin/ifconfig",
                (DevnameComboBox->currentText()).latin1(), "up");
        if ( (system(cmd)) != -1)
                startTimer( HopintervalSpinBox->value());
        else
                log("cisco_monitor(): system() failed");
}

/*
 * the main hopper code, physically changes channels
 */
void Discoverer::hopper()
{
      static int i = 0;
      char str[69];
      int n;
      int channel[] = { 1,6,11,2,7,3,8,4,12,9,5,10};
      //int channel[] = { 1,7,13,2,8,3,14,9,4,10,5,11,6,12};

        if (i > 11) i = 0;
        if ( DevtypeComboBox->currentItem() == ORINOCO) {
                snprintf(str, 29, "iwpriv %s monitor %d %d",
            (DevnameComboBox->currentText()).latin1(), 2, channel[i]);
        }
        else if ( DevtypeComboBox->currentItem() == PRISM2) {
                snprintf(str, 69,
                "wlanctl-ng %s lnxreq_wlansniff channel=%d enable=true
>/dev/null",
                (DevnameComboBox->currentText()).latin1(), channel[i]);
        }

      if ( (n=system(str)) == -1) {
            log("hopper(): system() failed");
                return;
      }
        i++;
}

/*
 * adds the passed network to the list
 */
QListViewItem* Discoverer::add(n_info_t n)
{
        QListViewItem *entry0;
        QListViewItem *entry1;
        QListViewItem *entry2;
```

```
QListViewItem *entry3;
QListViewItem *entry4;
QListViewItem *entry5;
QListViewItem *entry6;
QListViewItem *entry7;
QListViewItem *entry8;
QListViewItem *entry9;
QListViewItem *entry10;
QListViewItem *entry11;
QListViewItem *entry12;
QListViewItem *entry13;
final_list_t *element;
QString ssid(n.ssid);
QString channel;
QString wep;
QString manuf(n.manufacturer);
QString ap;
QString bssid;
QString packet_count;
QString active_clients;
QTime ts; // timestamp for printing
QString timestamp;
static int count = 0;

dsp_net_count(++count); // increment network count

//log( "packet_count = %d", n.packet_count);

if (n.wep == 1)
        wep = "Y";
else
        wep = "N";
if (n.ap == 1)
        ap = "AP";
else
        ap = "Ad-Hoc";
bssid = bssid.sprintf("%02x%c%02x%c%02x%c%02x%c%02x%c%02x",
                n.bssid[0], ':', n.bssid[1], ':',  n.bssid[2], ':',
                n.bssid[3], ':', n.bssid[4], ':', n.bssid[5]);

ts = QTime::currentTime();
timestamp.sprintf("%02d:%02d:%02d",
                        ts.hour(), ts.minute(), ts.second());

if ( L != NULL)
        entry0 = new QListViewItem( ListView, L->entry, ssid,
```

```
                        QString(channel).setNum(n.channel),
                        wep,
                        manuf);
        else
                entry0 = new QListViewItem( ListView, ssid,
                        QString(channel).setNum(n.channel),
                        wep,
                        manuf);
        entry1 = new QListViewItem(entry0, entry0, "BSSID", bssid);
        entry2 = new QListViewItem(entry0, entry1, "Type", ap);
        entry3 = new QListViewItem(entry0, entry2, "Packets",
                        QString(packet_count).setNum(n.packet_count));
        entry4 = new QListViewItem(entry0, entry3, "FirstSeen", timestamp);
        entry5 = new QListViewItem(entry0, entry4, "LastSeen", timestamp);
        entry6 = new QListViewItem(entry0, entry5, "Nodes",
                        QString(active_clients).setNum(n.n_lclients));
        entry7 = new QListViewItem(entry0, entry6, "WirelessClients",
                        QString(active_clients).setNum(n.n_wclients));
        if (wep == "N") { // IP range only possible with WEP off
                entry8 = new QListViewItem(entry0, entry7, "IPrange");
                entry9 = new QListViewItem(entry8, entry8, "LoIP", "0.0.0.0");
                entry10 = new QListViewItem(entry8, entry9, "HiIP", "0.0.0.0");
                entry11 = new QListViewItem(entry8, entry10,
                                        "Gateway", "0.0.0.0");
                entry12 = new QListViewItem(entry8, entry11,
                                        "Netmask", "255.255.255.255");
                entry13 = new QListViewItem(entry8, entry12,
                                        "Domain", "unknown");
        }

        // do a head insert in the final_list
        element = new final_list_t;
        element->entry = entry0;
        element->next = L;
        L = element;

        return entry0;
}

/*
 * starts the capture module, passes the capture module the interface name
 */
int Discoverer::fork_capture()
{
        // run hopper once to invoke RFMON before capture is forked
        // if( hopper_required) hopper();
```

```
        strcpy(devname, (DevnameComboBox->currentText()).latin1());
        capture_pid = fork();
        if ( capture_pid == 0) {
                fprintf( stderr, "capture: sending interface name %s\n", devname);
                execlp( "capture", "capture", devname, 0);
                log("fork_capture(): fork() failed\n");
                return -1;
        }
        return 0;
}

/*
 * gets a message from the queue and depending on its type performs actions
 */
void Discoverer::getnetinfo( n_info_t* network)
{
        int qid;
        long int msg_to_receive = 0;
        ui_message msg;

        memset(network, 0, sizeof(n_info_t));
        qid = msgget( (key_t)2002, 0600 | IPC_CREAT);
        if ( qid == -1) {
            log( "getnetinfo(): msgget failed");
    }
        if ( msgrcv( qid, (void *)&msg, sizeof(msg),msg_to_receive,
IPC_NOWAIT) == -1) {
                network->channel = 0;
        } else {
                memcpy( network, &(msg.data), sizeof(network_info_t));
        }
        if ( msg.message_type == 2) { /* got an update */
                update( network);
                network->channel = 0;
        }
        else if ( msg.message_type == 3) { /* got info to log */
                log((char *)(&msg.data));
                network->channel = 0;
        }
}

/*
 * kills the capture process
 */
void Discoverer::alles_kaput()
{
```

```
        if ( (kill( capture_pid, 15)) < 0) {
                log("alles_kaput: capture does not die!");
                perror("alles_kaput: ");
                log("alles_kaput: trying kill -9");
                kill( capture_pid, 9); // do not care what happens here
        }
}

/*
 * logs events, accepts formatted input, similar to printf
 */
void Discoverer::log( const char *fmt, ...)
{
        char str[64];
        va_list argp;

        va_start(argp, fmt);
        vsnprintf( str, 64, fmt, argp);
        va_end(argp);

        TextView1->append(str);
}

/*
 * network count display in the title bar
 */
void Discoverer::dsp_net_count( const int i)
{
        QString str1 = "Discoverer -- found ";
        QString str2 = "0";
        QString str3_si = " network";
        QString str3_pl = " networks";
        QString cap;
        if ( i == 1)
                cap = str1 + str2.setNum(i) + str3_si;
        else
                cap = str1 + str2.setNum(i) + str3_pl;

        QWidget::setCaption( tr(cap));
}

/*
 * updates the user interface with new information
 */
void Discoverer::update( n_info_t *n)
{
```

```
final_list_t *temp_L = L; // final list entry
QListViewItem *temp_item = NULL; // items in the entry
QString str, bssid;
int flag = 0; // flag if update needed
QTime ts; // timestamp for printing
QString timestamp;

bssid.sprintf( "%02x%c%02x%c%02x%c%02x%c%02x%c%02x",
                n->bssid[0], ':', n->bssid[1], ':',  n->bssid[2], ':',
                n->bssid[3], ':', n->bssid[4], ':', n->bssid[5]);
ts = QTime::currentTime();
timestamp.sprintf("%02d:%02d:%02d",
                        ts.hour(), ts.minute(), ts.second());

while ( temp_L != NULL) {
        temp_item = temp_L->entry->firstChild();
        /* if this was a re-assoc request (chan=15), get SSID */
        if ( temp_L->entry->text(0) == "" && n->channel == 15) {
                //qWarning("channel 15");
                temp_L->entry->setText(0, n->ssid);
        }
        while ( temp_item != NULL) {
                //qWarning(temp_item->text(0));
                //qWarning( temp_L->entry->text(0));
                if ( temp_item->text(1) == bssid)
                        flag = 1; // needs an update
                else if ( temp_item->text(0) == "Packets" && flag == 1) {
                        if ( temp_item->text(1) != str.setNum(n-
>packet_count)) {
                                temp_item->setText(1, str.setNum(n-
>packet_count));
                                flag = 2;
                        }
                }
                else if( temp_item->text(0) == "LastSeen" && flag == 2) {
                        temp_item->setText(1, timestamp);
                }
                else if( temp_item->text(0) == "Nodes" && flag == 2){
                        temp_item->setText(1, str.setNum(n->n_lclients));
                        //if ( n->n_lclients != 0)
                        //      add_mac(temp_item, n->clientl);
                }
                else if( temp_item->text(0) == "WirelessClients" && flag ==
2){
                        temp_item->setText(1, str.setNum(n->n_wclients));
                        //if ( n->n_wclients != 0)
```

```
                        //      add_mac(temp_item, n->clientw);
                    }
                    else if( temp_item->text(0) == "IPrange" && flag == 2) {
                        update_ip_range( temp_item, n);
                    }
                    temp_item = temp_item->nextSibling();
                }
                flag = 0;
                temp_L = temp_L->next;
        }
}

/*
 * updates IP range for non-WEPped networks
 */
void Discoverer::update_ip_range(QListViewItem *item, n_info_t *n)
{
        QString str;
        struct in_addr a;
        char s[32];

        item = item->firstChild();
        while( item != NULL) {
                if (item->text(0) == "LoIP") {
                        a.s_addr = n->iprange.lo_ip.s_addr;
                        strncpy(s, inet_ntoa(a), 16);
                        item->setText(1, str.sprintf("%s", s));
                }
                else if (item->text(0) == "HiIP") {
                        a.s_addr = n->iprange.hi_ip.s_addr;
                        strncpy(s, inet_ntoa(a), 16);
                        item->setText(1, str.sprintf("%s", s));
                }
                else if (item->text(0) == "Gateway") {
                        a.s_addr = n->iprange.gw.s_addr;
                        strncpy(s, inet_ntoa(a), 16);
                        item->setText(1, str.sprintf("%s", s));
                }
                else if (item->text(0) == "Netmask") {
                        a.s_addr = n->iprange.netmask.s_addr;
                        strncpy(s, inet_ntoa(a), 16);
                        item->setText(1, str.sprintf("%s", s));
                }
                else if (item->text(0) == "Domain") {
                        item->setText(1, str.sprintf("%s", n->iprange.domainname));
                }
```

```
                item = item->nextSibling();
        }
}

/*
 * checks for the UID and EUID, must be 0 to continue
 */
int Discoverer::amiroot()
{
        if ( getuid() || geteuid() ) {
                log( "Please, run discoverer as root");
                return 0;
        } else {
                return 1;
        }
}

/*
 * adds MAC addresses to the list
 * not used due to impractically long lists in complex networks
 */
void Discoverer::add_mac( QListViewItem *item, client_list_t *list)
{
/* for this to work need to get rid of pointers in the network_info_t */
        QListViewItem *qtemp;
        client_list_t *temp;

        QString macstr;

        qtemp = item->firstChild();

        if ( qtemp == NULL) {
                temp = list;
                while ( temp != NULL) {
                        macstr.sprintf("%02x:%02x:%02x:%02x:%02x:%02x",
                  temp->bssid[0], temp->bssid[1],
                        temp->bssid[2], temp->bssid[3],
                        temp->bssid[4], temp->bssid[5]);
                        //(void) new QListViewItem(item, macstr);
                        qWarning ("new one");
                        temp = temp->next;
                }
        } else {

        while ( qtemp != NULL) {
                temp = list;
```

```
            while ( temp != NULL) {

        macstr.sprintf("%02x%c%02x%c%02x%c%02x%c%02x%c%02x",
                  temp->bssid[0], ':', temp->bssid[1], ':',
                        temp->bssid[2], ':', temp->bssid[3], ':',
                        temp->bssid[4], ':', temp->bssid[5]);
                        if ( macstr != qtemp->text(0))
                              (void) new QListViewItem(item, macstr);
                        temp = temp->next;
                  }
                  qtemp = qtemp->nextSibling();
        }
        }
}
```

```
# discoverer QPE project file
# mainly used to generate Makefile
TEMPLATE  = app
CONFIG          = qt warn_on debug
HEADERS         += discoverer h
SOURCES         += main.cpp discoverer.cpp
INCLUDEPATH     += $(QPEDIR)/include
DEPENDPATH     += $(QPEDIR)/include
LIBS          += -lqpe
INTERFACES      = discovererbase.ui
TARGET          = discoverer
```

REFERENCES

[1] *Airsnort's website*, http //airsnort shmoo com

[2] Brenner P., *A Technical Tutorial on the IEEE 802.11 Protocol*, BreezeCOM Wireless Communications, 1997

[3] Fluhrer S., Mantin I., Shamir A., *Weaknesses in the Key Scheduling Algorithm of RC4*, 2001

[4] Foust, R., *Identifying and Tracking Unauthorized 802.11 Cards and Access Points*, ;login: the Magazine of USENIX and SAGE, August 2002 volume 27, number 4

[5] Gast M., *Wireless LAN Security: A Short History*, The O'Reilly Network, 2002, http://www.oreillynet.com/pub/a/wireless/2002/05/24/wlan.html

[6] Gast M., *Seven Security Problems of 802.11 Wireless*, The O'Reilly Network, 2002, http://www.oreillynet.com/pub/a/wireless/2002/04/19/security.html

[7] *Handhelds.org website*, http://www.handhelds.org

[8] *IBM's Wireless Security Auditor (WSA)*, http://www.research.ibm.com/gsal/wsa/

[9] IEEE Computer Society, *ANSII/IEEE 802.11 Std, 1999 Edition*, http://www.ieee.org

[10] *Kismet's website*, http://www.kismetwireless.net

[11] *Libpcap website*, http://www.tcpdump.org

[12] Lynn M., Baird R., *Advanced 802.11 Attack*, Black Hat 2002 Presentation, 2002

[13] *Netstumbler's website*, http://www.netstumbler.com

[14] *The Unofficial 802.11 Security Webpage*, http://www.drizzle.com/~aboba/IEEE/

[15] *Trolltech Inc. website*, http://www.trolltech.com

[16] Stevens, W., *UNIX Network Programming*, Engelwood Cliffs, NJ: Prentice Hall, 1990

VITA

Alex Medvedev was born in Volgograd, Russia, on July 28, 1971. After completing his work at Volgograd high school #92, Volgograd, Russia, in 1988, he entered Gubkin Oil and Gas Academy in Moscow, Russia, in 1989. He received a diploma in Petroleum Engineering in June 1994. After that he worked as an English-Russian interpreter for various petroleum companies in Russia. In January 2001 he entered the Graduate College of Southwest Texas State University, San Marcos, Texas.

This thesis was typed by Alex Medvedev.