1.3

Seminar Report
Oberseminar Informationstechnik

Wireless Test Environment for Evaluating Packet Recovery Algorithms

submitted by

Patrick Ziegler

born on February 5, 1991
in Bad Soden-Salmünster
Imma.-Nr.: 3750096

Dresden University of Technology

Faculty of Electrical and Computer Engineering

Institute of Communication Technology,
Deutsche Telekom Chair of Communication Networks

Supervisors: Prof. Dr.-Ing. Dr. h.c. Frank H. P. Fitzek,
M.Sc. Juan Alberto Cabrera Guerrero

Submitted on March 20, 2017

Contents

1 Introduction

The motivation to this work is to use packet recovery algorithms for recomputing broken packets in order to avoid unnecessary retransmissions and thus increase the throughput of wireless communication systems, for which packet loss due to interference and noise is still a major problem.

Network interfaces in standard configuration automatically drop packets with failed checksum validation and, depending on the chosen protocol, may induce retransmissions without the users knowledge.

It is therefore necessary to deliver broken packets to application layer and deactivate the automatic dropping of packets to be able to validate packet recovery algorithms on a real channel and in a realistic setup.

In chapter 2, we will investigate the options one has to adapt the processing of incoming packets to his needs. Ways of accessing the kernels network protocol stack and basic driver configuration are described and the Berkley Packet Filter is introduced to reduce incoming traffic.

In chapter 3, the techniques described before are incorporated to solve the task of delivering broken packets to application layer. A fully working example written in Python is discussed.

The following explanations all base on the use of sockets as an interface to the kernels network protocol stack. It is assumend that the reader has knowledge about the mentioned protocols. All following explanations refer to the use of wireless networks, although most of it should also be transferable to Ethernet.

How the explanations in chapter 2 are affected when using an encrypted wireless channel is discussed in another seminar report for the module *Problem Based Learning*.

The source code described in the following is appended in digital form. It can be found in the public repository [Zie17]. This account has the login *osbpf@protonmail.com* and the password *python5005*.

2 Options on how to Manipulate Packet Processing

2.1 Accessing Protocols of Lower Layers

Computer networking is usually described by the OSI model[1]. It introduces several layers of abstraction to simplify the implementation of interfaces between arbitrary devices and networks. Figure 2.1 shows the OSI model and some corresponding protocols that are important for IP-based communication as it is used for the internet and most local networks today. The latter are typically either organized as corded ethernets following the standard IEEE 802.3 or as wireless networks (WiFi) following IEEE 802.11.

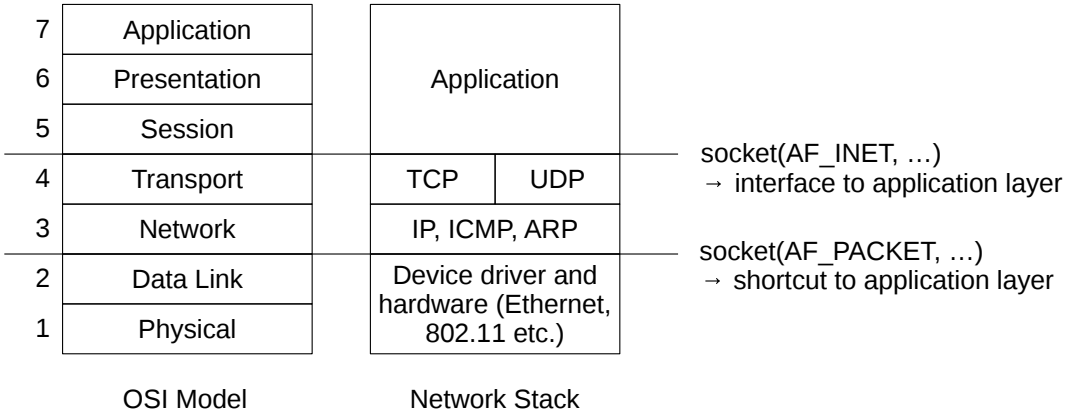

Figure 2.1: Sockets as an interface lower layers in the OSI model

Communication on a layer of abstraction means that one doesn't have to worry about limitations of underlying layers. In example when sending messages through a successfully set up TCP channel (Layer 4), there is no need to consider routing paths (Layer 3) or physical addressing (Layer 2).

With this principle comes that information about the operation of lower level protocols is usually shadowed and only the content that is relevant for the next higher layer is handed over. Packets are divided in header and payload whereby the header contains information that is needed for handling the packet in the particular layer.

Figure 2.2 shows how headers are stripped as packets move up the protocol stack. Only layer 2 incorporates a frame check sequence field (FCS) which is appended at the end of

[1]the *Open Systems Interconnection Model* was published by the International Organization for Standardization (ISO) in 1984 as a contribution to the standardization of computer networking

a frame and holds a checksum of header and payload to allow the detection of errors.

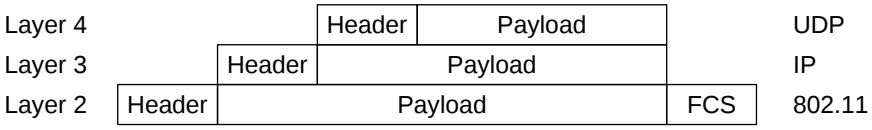

Figure 2.2: Stripping of headers as a packet moves up the protocol stack

End of line for all incoming packets is the application layer. The distinction between application layer and the underlying protocol layers reveals itself in the widely used terms user space and kernel space. The latter is a restricted area provided by the operating system that is not to be penetrated by the user.

As seen in figure 2.1, sockets serve as an entry point to application layer for incoming packets which can also be seen as interfaces between user and kernel space. Such interfaces not only allow to retrieve messages that were captured by the network interface but also to configure how the captured frames are processed on the way up to application layer.

In most programming languages, sockets are implemented as an object providing a standardized set of functions (such as `listen, accept, send` and `recv`) for setting up the socket and using it for sending and receiving messages. Listing 2.1 shows a very basic example of how to set up a socket for UDP communication in Python and use it to send a message to `127.0.0.1:5005` and receive an answer of 1024 bytes afterwards.

```
from socket import socket, AF_INET, SOCK_DGRAM
sock = socket(AF_INET, SOCK_DGRAM)
sock.sendto("Hello World", ("127.0.0.1", 5005))
msg, addr = sock.recvfrom(1024)
```

Listing 2.1: Simple example on how to use sockets in Python

The initialization of a socket (as seen in line 2 in the listing above) takes three numerical arguments of which the last one is usually set to zero or is omitted. The Linux manual tells us that the first argument chooses the *socket family* and the second specifies the *socket type* while the third argument is used to select a *protocol*. Referring to the OSI model shown above, the *socket family* can also be seen as the layer 3 protocol (IP, X.25, etc.) to be used while the *socket type* distinguishes between connection oriented and connectionless communication on transport layer. The *protocol* is usually chosen automatically as the combination of *family* and *type* often determines the protocol to be used. The set of usable protocols strongly depends on the socket family. In example only protocols such as `IPPROTO_*` are available for the family `AF_INET`. In chapter 3, we will use this argument to register the socket interface as a handler for all packets leaving the driver.

Incoming frames are at first processed by the driver, who provides an interrupt routine that delivers frames captured by the network device to the network stack after some processing. If a packet handler for the *protocol* `ETH_P_ALL` was registered, the frame is handed over to this handler immediately. Otherwise, the appropriate network level

protocol handler (such as `ip_rcv` or `arp_rcv` provided by the kernel) is called and the frame takes its course up the protocol stack. Further details on how received frames are processed by the Linux network stack can be found in [Ros09] and [Van16].

Sockets can register themselves as protocol handlers at almost any point in the network stack. Special socket families and types are defined for this purpose. Listing 2.2 demonstrates what configurations can be used to access different stack depths.

```
from socket import socket, AF_INET, SOCK_DGRAM, SOCK_RAW, IPPROTO_UDP,
    htons
ETH_P_ALL = htons(0x0003)                       # def. in if_ether.h
s1 = socket(AF_INET, SOCK_DGRAM)                # just UDP payload
s2 = socket(AF_INET, SOCK_RAW, IPPROTO_UDP)     # IP/UDP headers incl.
s3 = socket(AF_PACKET, SOCK_DGRAM)              # IP/UDP headers incl.
s4 = socket(AF_PACKET, SOCK_RAW, ETH_P_ALL)     # full MAC frames
```

Listing 2.2: Configuring sockets for access of lower layers

Receiving messages with `recvfrom` as in listing 2.1 would only give us the UDP payload when using the socket `s1`. With this, we are operating at application layer and all information of lower layers is therefore stripped. If we would also like to investigate the headers of network and transport layer, we would have to use the socket type `SOCK_RAW` or switch the socket family to `AF_PACKET`.

Raw sockets omit the protocol stack and enable us to work on network layer. Incoming UDP packets therefore still include the IP and UDP headers. Interestingly, the same can be obtained with the combination of packet sockets and UDP communication as used for `s3`. *Packet sockets* go even further and operate on data link layer. Using raw sockets with the packet family enables us to receive full MAC frames[2] including the Ethernet or WiFi header.

The protocol for `s4` was set to `ETH_P_ALL` to obtain packets of all types. Browsing the kernel source code leads to the file `if_ether.h` in which more `ETH_P_*` protocols such as `ETH_P_IP` are defined as constants with a length of two bytes. Network devices typically use big-endian[3] notation which does not necessarily be the case for the local machine. Therefore the function `htons` is used to handle the order of a series of bytes if necessary. Protocol constants for `IPPROTO_*` (as declared in the kernels `in.h`) do not need this treatment as their length is only one byte.

Raw sockets also allow us to manipulate header information when sending packets. For it is possible to break the integrity of protocol headers by this, root privileges are required to run software that is using raw sockets on Linux hosts.

Using simple socket configuration as seen in the listings above, we can now access protocols of lower layers. We come back to sockets for attaching filters to reduce incoming traffic in section 2.3.

---

[2] *Media Access Control (MAC)* implements hardware addressing for data link layer, according frame definitions can be found in figure 2.3

[3] *Big-endian* byte order means the most significant bit (MSB) is the first bit on the left side. This is opposed to *little-endian*, where the MSB is found on the right end of a byte

2.2 Configuring the Network Interface Driver

The network interface driver operates at physical and data link layer. There is no API such as sockets to easily manipulate the drivers behaviour out of userspace. The only option is to use device configuration tools to switch between possible modes of operation. For this work, the most important modes are *managed* and *monitor mode*.

Managed mode is the usual mode of operation that allows to connect to wireless networks that are managed by a central access point. Capturing network traffic in this mode reveals that received frames hold a MAC header as specified for Ethernet although they were received by a wireless network interface and therefore a WiFi frame would be expected. This is due to the drivers behaviour to recompute the MAC header of received WiFi frames to *fake Ethernet headers* missing the preamble, the start frame delimiter (SFD) and the frame check sequence (FCS) and are therefore only consisting of destination and source MAC addresses, protocol type and payload. A comparison of both frame types is shown in figure 2.3. All received frames that are not directed to the network interfaces MAC address or frames that do not pass the Cyclic Redundancy Check (CRC) will be dropped in managed mode.

CRC is an algorithm to condense series of bits into short checksums. Such checksums are appended to outgoing frames as *frame check sequence (FCS)* as shown in figure 2.3. The checksum is again computed in the receiving device and immediately compared to the value found in the FCS field. If these checksums are not equal, the frame must have changed on its way to the receiver. It is therefore considered faulty and will be dropped by the driver when the network interface is in managed mode.

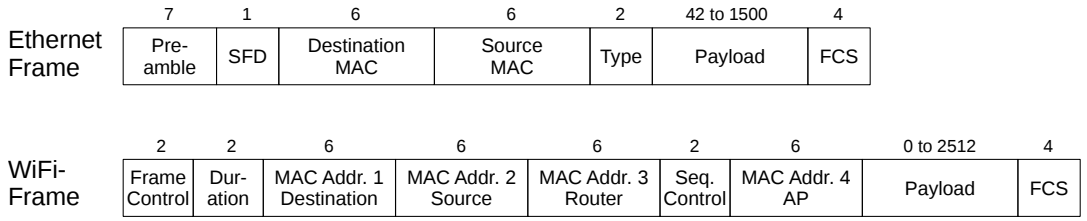

Figure 2.3: Comparison of Ethernet and 802.11 frames

The use case described in chapter 3 makes it necessary to omit the checksum validation as it is desired to deliver broken packets to application layer. This can be achieved by using a network interface in monitor mode.

Monitor mode is fully passive which means that no packets can be send out and therefore no connection to available access points can be established. The advantage of monitor mode in this scenario is that no packets are dropped by the driver even if the checksum validation fails. Also no fake Ethernet but original WiFi frames are delivered to the protocol stack. Even more information is added by the driver as a so called *Radiotap header* is appended in front of the frame. A lot of useful flags such as `bad_fcs`[4] can be

[4] *bad_fcs* shows the outcome of checksum validation even in monitor mode, where no packets will be dropped due to failed CRC

found there.

With Linux kernel 2.6.22, virtual network interfaces (VIF) were introduced. Physical devices and logical interfaces are therefore decoupled which allows to attach several virtual interfaces to one physical device. This makes it possible to attach an interface in monitor mode to a network device that already has an interface in managed mode while the latter doesn't loose connection to its access point. Even though interfaces in monitor mode are fully passive, it is therefore possible to receive packets that are not directed to the local network interface while being connected to a WiFi network. Sadly, it is not possible to use VIF to be connected to more than one WiFi network at the same time in managed mode with only one network device.

```
iw phy0 interface add mon0 type monitor
ip link set mon0 up
```

Listing 2.3: Setting up a network interface in monitor mode

How an interface in monitor mode can be attached to a physical network device is shown in listing 2.3. What physical device to use in the notation `phy*` can be found out with `iw dev` that returns a list of physical devices and its attached interfaces.

The commands `iw` and `ip` are used to configure the drivers behaviour. They are replacing the still widely used commands `iwlist`, `iwconfig` and `ifconig` which are considered deprecated because of incompatibility with IPv6.

2.3 Filtering with Berkley Packet Filter

Using the techniques described above to capture all packets on a specific WiFi channel can lead to a very high amount of incoming traffic. In most cases, only a small set of packets fulfilling some conditions are of interest. To allow every packet to pass the entire protocol stack and use some logic on application layer to drop unwanted packets can lead to high computing time. A way of filtering packets as early as possible is therefore desired to overcome these circumstances.

This is provided by a mechanism called *Berkley Packet Filter*[5] which can be interpreted as a virtual device in the Linux kernel that receives incoming frames from the network device and decides whether to drop a frame or not. The device can be programmed just like a microcontroller with a special assembly language which is described in [SBS].

The BPF is able to load data from the frame and perform operations on it such as masking and comparing the outcome with some constant. Arithmetic operations are also possible.

On application layer, the BPF is represented by its operation code (opcode) that is attached to a socket with the option `SO_ATTACH_FILTER` as can be seen in the last line of listing A.1. We see that filters are set up as further socket configuration.

To write a filter in the BPF assembly language by hand can develop into a very painful

[5] *Berkley* is part of the name as this mechanism was introduced in the Berkley Software Distribution (BSD), a Unix-like operating system developed at the University of California, Berkley

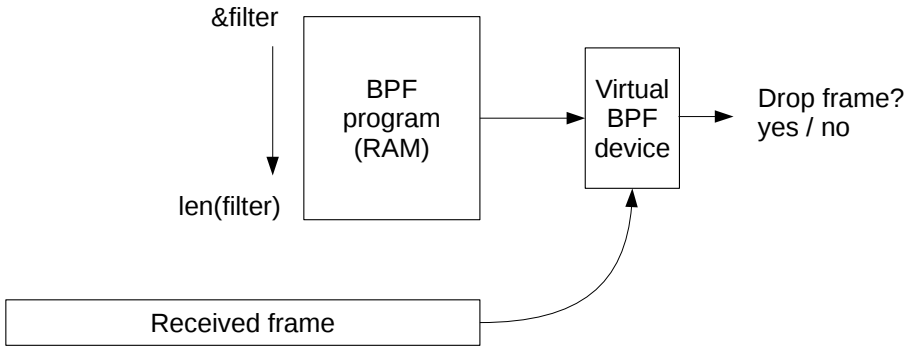

Figure 2.4: Data flow for filtering with BPF

and time-consuming task. Therefore it is possible to let such programs be put together by `tcpdump` with a more readable filter definition as described in [Mie04]. Using the flag `-d`, this tool will print out human readable code in the assembly language and the according opcode as an array in C can be obtained by setting the flag `-dd`. Listing 2.4 shows the output of a very simple filtering example. Only `arp` requests will be let trough. A more sophisticated example is to be found in chapter 3.

```
patrick@thinkpat:~/> sudo tcpdump -d arp
[sudo] Passwort für root:
(000) ldh      [12]
(001) jeq      #0x806           jt 2    jf 3
(002) ret      #262144
(003) ret      #0
```

Listing 2.4: Output of a simple BPF filter put together by `tcpdump`

As already mentioned, listing A.1 shows how sockets can be extended with a low level packet filter. The code was inspired by [Psw15] and can be used as a tool to attach arbitrary filter definitions to sockets. Although written in Python, it can easily be reimplemented in C as the opcode obtained by `tcpdump -dd` is parsed and put in an array of type `struct`, just like it would be done in C. No external libraries are needed to use this function. The reason for implementing the structures `__sock_filter__` and `__sock_fprog__` as seen in A.1 and further encapsulating them into another structure `bpf_prog` is that [SBS] clearly states that when configuring a socket with `SO_ATTACH_FILTER`, a structure containing a pointer to the start of the opcode together with the number of instructions is expected as parameter.

The standard configuration of `tcpdump` assumes Ethernet frames to be processed. If other frame types are expected, this can be specified with the flag `-y` as done in the following example. Frame types depend on the interface to be used, it could therefore be necessary to select the appropriate interface as it is done in the example.

```
tcpdump -dd -i mon0 -y IEEE802_11_RADIO udp and port 5005

{ 0x30, 0, 0, 0x00000003 },
{ 0x64, 0, 0, 0x00000008 },
```

```
{ 0x7, 0, 0, 0x00000000 },
...
```

Listing 2.5: Obtaining BPF opcode for filtering WiFi frames

3 Deployment in the Setting of Package Recovery

3.1 Implementation in Python

To be able to test the different socket configurations described in section 2.1, a pair of Python scripts was written to send and receive packets between two devices.

The script `tx.py` is used as a sender of dummy packets. This function constantly sends messages like *msg37*, *msg38* etc. including an incremented number to easily detect lost packets. The transport layer protocol can be specified with the flag `-m`, where possible values are *TCP* and *UDP*. The delay in seconds and the port number as an integer can be specified in the same way with `-d` and `-p`.

```
tx.py [-h] [-p PORT] [-m MODE] [-d DELAY] ip

positional arguments:
  ip                    Destination IP-Adress

optional arguments:
  -h, --help            show this help message and exit
  -p PORT, --port PORT
  -m MODE, --mode MODE
  -d DELAY, --delay DELAY
```

Accordingly, the script `rx.py` is used as a receiver. Like in tx.py, several modes of operation are supported; *TCP* and *UDP* are two of them. To test the behaviour of raw sockets, the mode *RAW* is defined. Incoming messages are then formatted as byte array to be easily comparable with the output of Wireshark. A socket with the family `AF_PACKET` along with a BPF packet filter is used in the mode *BPF* to implement a basic packet sniffer. Using a device in monitor mode with `-m` *BPF* will fulfil the task of delivering broken packets to application layer.

Packet sockets cannot be bound to an IP address as no protocol handling on network and transport layer is carried out. Therefore the local interface name instead of the destination IP address is expected as parameter. The script will automatically detect the appropriate IP address if necessary. Other arguments are the destination port as an integer and the socket timeout in seconds. The receive buffer size can be specified with `-b RCVBUF` which should not be necessary under usual circumstances.

```
rx.py [-h] [-p PORT] [-b RCVBUF] [-t TIMEOUT] [-m MODE] iface

positional arguments:
  iface                 Network interface
```

```
optional arguments:
  -h, --help            show this help message and exit
  -p PORT, --port PORT
  -b RCVBUF, --buffer RCVBUF
  -t TIMEOUT, --timeout TIMEOUT
  -m MODE, --mode MODE
```

Using `rx.py` for receiving packets with `-m` *RAW* and `-m` *BPF* leads to the output shown below. In both cases, the received frames were sent out with `tx.py` in UDP mode.

```
patrick@shuttle:~/> sudo python rx.py -m raw mon0

Received at 192.168.178.38:
45 00 00 20 5f 83 40 00 40 11 f5 a6 c0 a8 b2 26
c0 a8 b2 2b dd ed 13 8d 00 0c 44 1b 70 6b 74 31

patrick@shuttle:~/> sudo python rx.py -m bpf mon0

Received at mon0 (bad_fcs: False):
00 00 27 00 2b 40 08 a0 20 08 00 00 00 00 00 00
22 28 b0 ee 19 00 00 00 10 00 80 09 80 04 bd 00
00 00 27 04 07 bd 00 88 42 2c 00 64 70 02 b5 47
bb c8 0e 14 49 fc 94 00 24 d7 b3 db d0 d0 76 00
00 71 48 00 20 00 00 00 00 aa aa 03 00 00 00 08
00 45 00 00 20 7a d2 40 00 40 11 da 57 c0 a8 b2
26 c0 a8 b2 2b d7 02 13 8d 00 0c 4b 03 70 6b 74
34 8f 80 cc 4d c1 b9 3b ba 9d a8 1b f3
```

Listing 3.1: Output of `rx.py` in modes *RAW* and *BPF*

We see that the received frame in the second case is very much longer than in the first one. Relating this difference with the interpretation of such byte arrays in Wireshark reveals that in the first case we only see the IP and UDP headers along with the payload as opposed to the second case, where a full WiFi frame including the FCS is printed out. With an interface in managed mode, we would instead see a *fake Ethernet* frame without FCS. This supports the explanations in chapter 2.

3.2 Attempt to Calculate the Error Rate

There are simulation results on how the network throughput would evolve as a function of bit error rate (BER) and transmission speed when incorporating packet recovery algorithms and thus avoiding retransmissions.

These results should be validated by applying the analysis to a realistic setup incorporating the techniques and scripts described above. For this, an access point with configurable transmission speed and power was set up and `tx.py` and `rx.py` were used to send and receive packets.

To test the worst case before carrying out the actual analysis, the first step was to set transmission speed to the highest possible value of 54 Mbit/s and transmission power to 10 mW as the lowest possible value. A notebook was used for receiving the packets

which allowed to increase distance to the access point until all packets would be lost. On the way out of the access points reach, some minor errors such as bit flips were expected to occur.

The mode *BPF* in `rx.py` was extended to print out the `bad_fcs` flag found in the Radiotap header and indeed there were some packets with failed checksum validation detected. But further analysis showed that all packets with the `bad_fcs` flag set were in fact retransmissions coming from the access point. Analysing the network traffic with Wireshark revealed a communication scheme as shown in figure 3.1

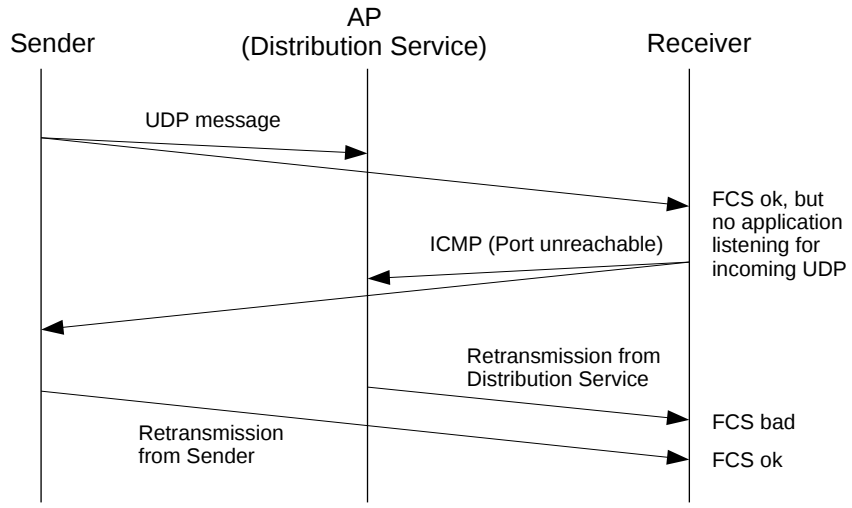

Figure 3.1: Analysis of network traffic with retransmissions

The information found in [Ros09, p. 30] clarified that using a receiver who is not able to handle IP information such as address and port (which is the case for `rx.py` in mode *BPF*) triggers the network driver send out the ICMP message *port unreachable* what leads both sender and access point (who was obviously buffering the network traffic) to retransmit the packet. On retransmission by the access point, the MAC addresses in the header changed, but the FCS value did not what led to the detection of bad checksum in the receiver.

With this insight, a more complex packet filter was formulated to drop all unwanted retransmissions. The command for obtaining the opcode for BPF is shown below.

```
tcpdump -dd -i mon0 -y IEEE802_11_RADIO udp
        and port 5005
        and ether host 01:23:a7:b3:db:f0
        and 'wlan[1] & 8 = 0'           # no retransmissions
        and 'wlan[1] & 3 = 1'           # only STA to DST
```

Listing 3.2: Extended filter definition for sorting out all retransmissions

With this filter attached, no checksum violations were detected any more, even on the way out of the access points reach. Packets were either received without errors or lost completely. Therefore no comparable error rate could be calculated.

4 Conclusion

Even though it was not possible to calculate the error rate as a function of transmission speed, the analysis shown in section 3.2 shows that the task of delivering broken packets to application layer can be solved by incorporating the explanations in chapter 2 as packets were printed out even when the checksum validation failed. Only the reason for bad FCS values were unexpected.

The scripts described in chapter 3 can be used to easily test and verify all possible socket and driver configurations. They also serve as a working example on how to implement these techniques in Python or similar languages. This can be used to get deep insight in the processing of packets in the Linux kernel. Not relying on external libraries such as `libpcap`, which is used for most known packet sniffers, leads to greater flexibility in implementation and further strengthens the understanding of tools as `tcpdump` and Wireshark.

With the *Berkley Packet Filter*, a little known but nevertheless very useful tool was introduced. It can not only serve to reduce unwanted traffic as done in section 3.2, but also be used to implement intrusion detection against known attacking scenarios and much more.

A Appendix

A.1 Berkley Packet Filter in Python

```
import ctypes, ast
from socket import SOL_SOCKET

ETH_P_ALL = 0x0003 # as defined in if_ether.h
SO_ATTACH_FILTER = 26 # as defined in socket.h

class __sock_filter__(ctypes.Structure):
    _fields_ = [
        ('code', ctypes.c_uint16),
        ('jt', ctypes.c_uint8),
        ('jf', ctypes.c_uint8),
        ('k', ctypes.c_uint32),
    ]

class __sock_fprog__(ctypes.Structure):
    _fields_ = [
        ('len', ctypes.c_int),
        ('filter', ctypes.c_void_p),
    ]

def sock_attach_bpf(sock, bpf_opcode):
    """Attach Berkley Packet Filter to socket

    Example:
    > from socket import socket, AF_PACKET, SOCK_RAW, ntohs
    > from lib.bpf import ETH_P_ALL, sock_attach_bpf
    > # tcpdump -dd arp
    > bpf_opcode = '''[
    >     { 0x28, 0, 0, 0x0000000c },
    >     { 0x15, 0, 1, 0x00000806 },
    >     { 0x6, 0, 0, 0x00040000 },
    >     { 0x6, 0, 0, 0x00000000 },
    > ]'''
    > sock = socket(AF_PACKET, SOCK_RAW, ntohs(ETH_P_ALL))
    > sock_attach_bpf(sock, bpf_opcode) % this function

    Implementation as described in:
    [1] https://www.kernel.org/doc/Documentation/networking/filter.txt
    [2] http://pythonsweetness.tumblr.com/post/125005930662/
        fun-with-bpf-or-shutting-down-a-tcp-listening
    """
    bpf_opcode = bpf_opcode.replace("\n","␣")
    bpf_opcode = bpf_opcode.replace("{","[")
    bpf_opcode = bpf_opcode.replace("}","]")
    bpf_opcode = ast.literal_eval(bpf_opcode)

    bpf_instns = (__sock_filter__ * len(bpf_opcode))()
    for i, (code, jt, jf, k) in enumerate(bpf_opcode):
        bpf_instns[i].code = code
        bpf_instns[i].jt = jt
        bpf_instns[i].jf = jf
        bpf_instns[i].k = k
```

```
53
54      bpf_prog = __sock_fprog__()
55      bpf_prog.len = len(bpf_opcode)
56      bpf_prog.filter = ctypes.addressof(bpf_instns)
57
58      sock.setsockopt(SOL_SOCKET, SO_ATTACH_FILTER, buffer(bpf_prog))
```

Listing A.1: `lib/bpf.py`

Bibliography

[Mie04] Miessler, Daniel: *A tcpdump Tutorial and Primer with Examples*. https://danielmiessler.com/study/tcpdump/#gs.y=nZHtY, 2004. – [Online; accessed 03-March-2017]

[Psw15] *Fun with BPF, or, shutting down a TCP listening socket the hard way*. http://pythonsweetness.tumblr.com/post/125005930662/fun-with-bpf-or-shutting-down-a-tcp-listening, 2015. – [Online; accessed 03-March-2017]

[Ros09] Rosen, Rami: *Sockets in the Linux Kernel*. http://haifux.org/hebrew/lectures/217, 2009. – [Online; accessed 07-March-2017]

[SBS] Schulist, Jay ; Borkmann, Daniel ; Starovoitov, Alexei: *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*. https://www.kernel.org/doc/Documentation/networking/filter.txt, . – [Online; accessed 03-March-2017]

[Van16] Vandecappelle, Arnout: *Networking, Kernel Flow*. https://wiki.linuxfoundation.org/networking/kernel_flow, 2016. – [Online; accessed 03-March-2017]

[Zie17] Ziegler, Patrick: *Source Code Repository*. https://bitbucket.org/osbpf/pybpf.git, 2017. – [Online; accessed 20-March-2017]

Declaration of authorship

I hereby certify that this report has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this report. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Dresden, March 20, 2016 Patrick Ziegler