A Machine-Learning Packet-Classification Tool for Processing Corrupted Packets on End Hosts

Martin Henze

Diploma Thesis

Student Reports on Communication and Distributed Systems CSR-2011-01

Chair of Communication and Distributed Systems, RWTH Aachen University March 2011

Impressum

Publisher:	Chair of Communication and Distributed Systems (Informatik 4) RWTH Aachen University Ahornstr. 55 52074 Aachen, Germany
Editor:	Prof. Dr.-Ing. Klaus Wehrle
Technical Editor:	Martin Henze

Please use the following BibTeX entry to reference this work:

```
@mastersthesis{CSR-2011-01,
author = {Henze, Martin},
title = {{A Machine-Learning Packet-Classification Tool for Processing Corrupted Packets on End Hosts}},
school = {{RWTH Aachen University, Chair of Communication and Distributed Systems}},
type  = {{Diploma Thesis}},
year  = {2011},
note  = {Student Reports on Communication and Distributed Systems CSR-2011-01}
}
```

Communication & Distributed Systems

RWTH AACHEN UNIVERSITY

A Machine-Learning Packet-Classification Tool for Processing Corrupted Packets on End Hosts

Diploma Thesis

Martin Henze

RWTH Aachen University, Germany

Chair of Communication and Distributed Systems

Advisors:

Dipl.-Inform.	Florian Schmidt
Prof. Dr.-Ing.	Klaus Wehrle
Prof. Dr. rer. nat.	Thomas Seidl

Registration date: September 02, 2010
Submission date: March 02, 2011

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Aachen, den 2. März 2011

(Martin Henze)

Kurzfassung

Mobile Geräte wie zum Beispiel Handys, Netbooks oder Notebooks haben in den vergangenen Jahren enorm an Popularität gewonnen. Einige der Anwendungen, die auf solchen Geräten benutzt werden, benötigen weiche Echtzeitbedingungen. Typische solchen Anwendungen sind zum Beispiel Internettelefonie oder Audio- und Videostreaming. Diese Anwendungen leiden merkbar unter Paketverlusten durch Fehler in der drahtlosen Übertragung.

In dieser Diplomarbeit wird ein Paketklassifikationswerkzeug vorgestellt das auf Konzepten des maschinellen Lernens beruht. Unter Benutzung dieses Werkzeuges ist es möglich, fehlerhafte Netzwerkpakete zu derjenigen Anwendung zuzuordnen, zu der sie mit der höchsten Wahrscheinlichkeit gehören. Somit können defekte Pakete, auch wenn sie Fehler in den Kopfdaten enthalten, durch den Netzwerkstack bearbeitet werden.

Abstract

The popularity of mobile devices such as mobile phones, netbooks, or laptops has enormously increased in the past years. Some applications which are used on these devices require soft real-time requirements. This holds, for example, for Voice-over-IP or audio and video streaming. These applications suffer notably from packet losses due to errors in the wireless communication.

In this thesis a packet-classification tool which is based on the concept of machine learning is presented. Using this tool, it is possible to hand corrupted packets over to the application which they most probably belong to. Thus, corrupted packets can be processed by the network stack even if they contain errors in the header data.

Acknowledgments

While writing this thesis, a number of people supported my in various ways. I would like to thank everyone, who made the writing of this thesis possible:

Prof. Dr.-Ing. Klaus Wehrle, for being my supervisor and providing such a pleasant working atmosphere at the Chair of Communication and Distributed Systems.

Prof. Dr. Thomas Seidl, for kindly agreeing to be the second supervisor for my thesis.

My advisor, Florian Schmidt, for suggesting the interesting topic of this thesis, many fruitful discussions, his useful advices, contributing some code, giving me a lot of freedom, and always having time for my questions.

Ismet Aktas and all the students in the cross-layer team, for many fruitful discussions and always being supportive. Especially, I would like to thank Mario Göttgens, Dominik Dennisen, and Tobias Drüner, whose programs could be used for this thesis.

Everyone at the Chair of Communication and Distributed Systems, for many interesting discussions, contributing to the pleasant working atmosphere, and one or another foosball match.

Justin Zobel, for his very helpful book "Writing for Computer Science" [71].

Sebastian Wüsten and Hendrik vom Lehn, for their thorough help in proof-reading this thesis.

My girlfriend, Lina, for her invaluable support and understanding, especially during the last months of writing this thesis.

Last, but not least, my family, who always supported me during the time of my studies.

Contents

1

Introduction

In the past years, the popularity of applications with soft real-time requirements, for example, Voice over IP (VoIP) or video streams grew massively. However, satisfying soft real-time constraints over unreliable links, for example in wireless networks, is a challenging task. One of the major problems in these environments is packet corruption. Packet corruptions occur when errors to one or more symbols of the transmitted signal are induced. In today's wireless networks these errors occur all the time.

For some applications, it is desirable to rather receive a corrupted packet than having to wait for a retransmission of this packet. Previously, for example, it has been shown that the speech-quality of a VoIP transmission can be improved if erroneous packets are processed to the application instead of being dropped [20]. It was observed that lost packets have a higher impact on the speech-quality than errors in the coded speech data.

In principle, all widely used Internet and Transport Layer protocols will simply drop corrupted packets. This leads to implicit or explicit retransmission of a erroneous packet and thus to delays or even losses of part of the data. However, delays and data losses should be avoided for applications with soft real-time requirements.

The goal of this thesis is to address this issue by exploring a novel approach in processing corrupted packets. Instead of retransmitting corrupted packets, they should be passed to the application. There already exist some approaches which will deliver packets with a corrupted payload to the application [35, 34]. The major drawback of these approaches is their incapability to handle errors in header fields which contain essential information for processing the packet in the network stack.

This thesis introduces a new paradigm in handling corrupted packets. The first bytes of correctly received packets are used to build a classifier. This classifier is then used to predict the destination socket of a corrupted incoming packet. If an application signals that it is capable of handling corrupted packets, this prediction can then be used in the Internet and Transport Layer protocol implementations as

a routing decision rule. One major challenge is to receive a high rate of correct predictions while keeping the number of false predictions to a minimum.

A major challenge is the implementation of the proposed solution in an existing operating system. In this thesis this will be done for the network stack of the Linux [39] operating system. Operating Systems are highly complex systems which are optimized to reach high performance while using the limited resources carefully. Performing changes to such a systems requires a deeper understanding of the relations of the different components which together form the operating system.

The solution proposed by this thesis mainly consists of two components: Firstly, the *packet-classification tool* which is used to build a classifier and to provide predictions for incoming corrupted packets. Secondly, an implementation of a *use case* which makes use of the prediction provided by the *packet-classification tool* in order to process corrupted IPv4 and UDP packets.

To summarize, the work described by this thesis contributes the following:

1. Design of a novel approach in processing corrupted packets in the network stack based on machine learning.
2. Implementation of this novel approach for the Linux kernel's network stack.
3. Evaluation, if the use of machine learning in the network stack is feasible to handle corrupted packets.

In order to design and implement the proposed solution, methods from the areas of communication systems engineering on the one side and data mining and machine learning on the other side had to be brought together. As it is not expected that the reader is proficient in all these areas, their main concepts will be presented before discussing the design of the proposed approach.

The remainder of this thesis is structured as follows: Firstly, Chapter 2 gives background knowledge to fields which are related to this thesis. Chapter 3 describes the conceptual design of the approach proposed by this thesis. In Chapter 4, the implementation of the presented approach is discussed. The evaluation of the implementation is covered in Chapter 5. Work which is related to the approach of this thesis is discussed in Chapter 6. Chapter 7 gives an outlook to future work which could be conducted to the approach presented in this thesis. Finally, Chapter 8 concludes this thesis.

2

Background

The approach proposed by this thesis is based on techniques from different fields of computer science. This mainly involves computer networking, wireless networking, data mining and machine learning. In addition, some knowledge of the implementation of the Linux kernel's network stack is required.

All required background knowledge for the topic of this thesis is presented in this chapter. As this presentation is thought as an abbreviatory introduction to the respective topic, not all details can be covered comprehensively. Thus, for each topic recommended literature for further studying is given.

In addition to the fields presented briefly in this chapter, some knowledge of the C programming language can be helpful to fully understand the details of the implementation. A good introduction to the C programming language is given by Kernighan and Ritchie [31].

2.1 Layered Protocol Stacks and the TCP/IP Model

Tanenbaum [63] gives an intuitive introduction to layered protocol stacks and protocol hierarchies. According to him, most networks consist of a stack of layers where each layer depends on the layer below it. Each layer offers services for the layer above it and thereby hides detailed information on the implementation of these services. This can be compared to the well-known concepts of information hiding and data encapsulation.

An instance of a protocol of a certain layer always communicates with an instance of the same protocol of the same layer on another system. It uses the services of the underlying layers and their protocols to perform the communication. There are clearly defined interfaces at which the communication of a protocol with the upper or lower protocol in the stack takes place.

The number of layers, their names and their responsibilities vary between different types of networks. A network architecture, which is often used in literature and education, is the ISO Open Systems Interconnection (OSI) Reference Model [27]. However, the protocols defined by the ISO OSI Reference Model and the model itself serve mainly as a theoretical model and are rarely used. Thus, in the following the focus will be on the widely used TCP/IP model.

The TCP/IP Model

The TCP/IP model, which is used in today's Internet, was defined by a Request for Comments (RFC) [10]. An RFC is a document published by the Internet Engineering Task Force (IETF). Most protocols in the Internet are standardized as an RFC. Although not all RFCs are standards, most RFCs referenced in this thesis are.

In contrast to the ISO OSI Reference Model, the TCP/IP model consists of only four layers, which will be discussed briefly: Link Layer, Internet Layer, Transport Layer, Application Layer (see Figure 2.1).

The *Link Layer* is the bottommost layer in the TCP/IP Model. It is responsible for providing an interface to a directly connected network, which allows only point-to-point communication. A very important duty of the Link Layer is media access. Thus, it is sometimes also called Media-Access Layer. There are not many details about the Link Layer in the RFC that defines the TCP/IP model. However, media access in today's systems is mainly realized in hardware. Typical Link Layer protocols are Ethernet, the IEEE 802.11 family, Fiber Distributed Data Interface (FDDI), and Point-to-Point Protocol (PPP).

Next is the *Internet Layer*, which is often also named Network Layer. Internet Layer protocols implement an unreliable multi-hop packet oriented transport from one host to another, which allows for end-to-end communication. However, no connection context is established. Unreliable here means that packets might be received broken, in the wrong order, duplicated or might even get lost. As the packets mostly travel over intermediate hosts, Internet Layer protocols have to implement a routing scheme. Some Internet Layer protocols are Internet Protocol version 4 (IPv4), and Internet Protocol version 6 (IPv6).

On top of this resides the *Transport Layer*. Its protocols offer applications a connection from an instance running on one host to an instance running on another host. Thus, they offer multiplexing which is typically based on port numbers. Some Transport Layer protocols also establish a connection context and offer reliability. This might also include flow control to prevent congestion at the receiver's side. The most common Transport Layer protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

The *Application Layer* is located at the very top. It hosts all the protocols used by the various applications. To mention just a few there are: Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), Real-time Transport Protocol (RTP), Domain Name System (DNS), and Dynamic Host Configuration Protocol (DHCP).

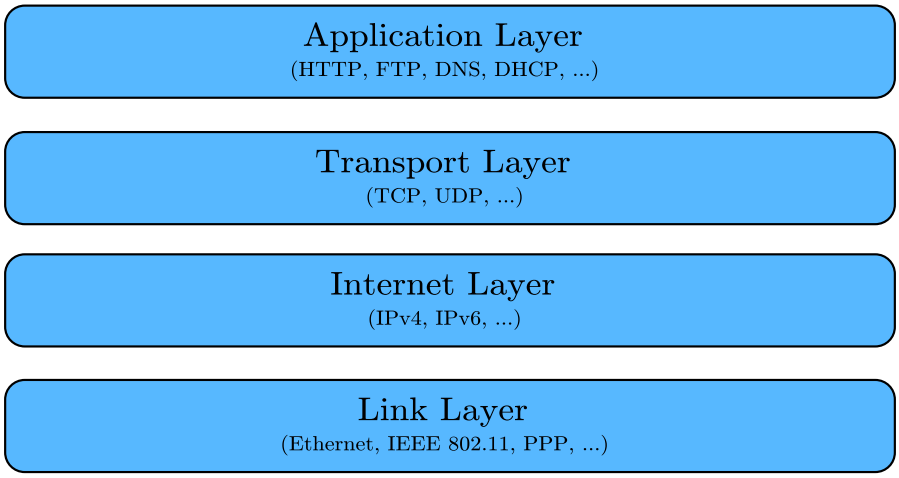


Figure 2.1 The TCP/IP Network Model consists of four layers which are shown in this figure.

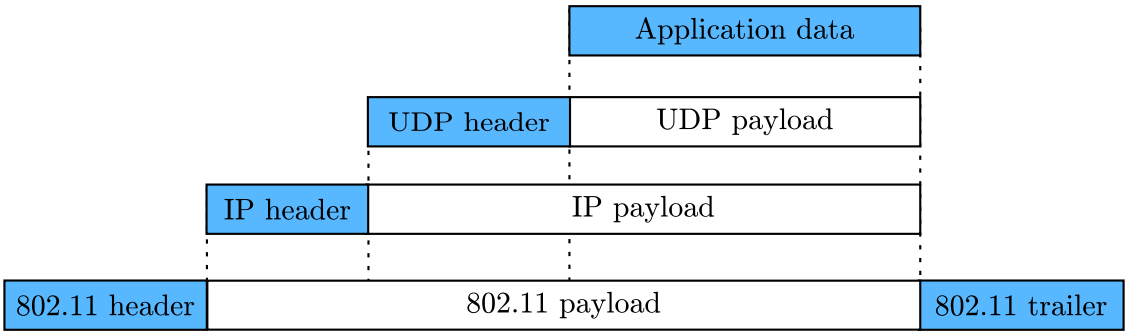


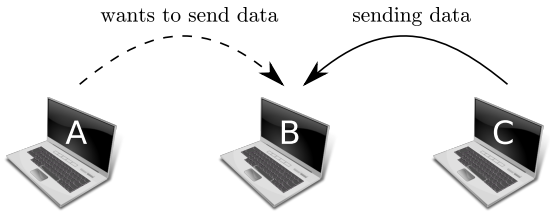
Figure 2.2 Each protocol in the network stack adds its own header (and potentially trailer) to the data.

2.2 Network Protocols and Headers

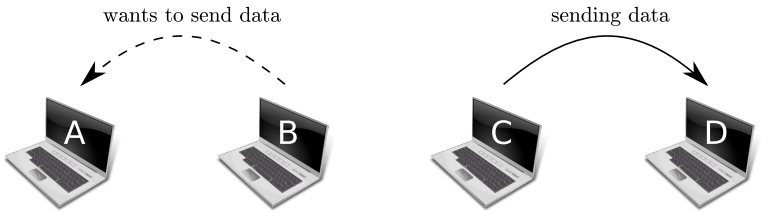
As seen in the previous section, each network layer hosts a number of protocols. Now, those protocols that are especially important for this thesis will be presented in more detail. Beforehand, a short introduction to network headers (and trailers) is given.

Each protocol requires control information, which has to be transmitted along with the so-called payload. Typical control data include addressing information, fragmentation information, length, and checksums. Typically, these control data are put in front of the payload. These data are then called the (protocol) header. Sometimes it is however necessary to put some control data (for example, the Link Layer checksum) behind the payload, which are then called the (protocol) trailer. Figure 2.2 shows how each layer in the network stack adds its own header (and trailer).

The remainder of this section is structured according to the layers in the TCP/IP model. Starting at the Link Layer and then climbing up the network stack, all protocols relevant to this thesis will be presented and discussed.



(a) The hidden station problem: A wants to send to B but cannot hear that C is already sending to B. This leads to a collision at B.



(b) The exposed station problem: B wants to send to A but falsely refrains from sending as it overhears the transmission from C to D.

Figure 2.3 The hidden station problem and the exposed station problem occur in wireless networks.

2.2.1 IEEE 802.11

The IEEE 802.11 standard (current version IEEE 802.11-2007 [26]) defines the implementation of wireless communication and is the Link Layer protocol typically used in wireless networks. It targets the physical medium and Medium Access Control (MAC). Detailed descriptions are given by Hiertz et al. [23] and Tanenbaum [63].

Due to the nature of the wireless channel, none of the existing MAC schemes could be used for IEEE 802.11. Neither the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) scheme of Ethernet (IEEE 802.3) nor the token passing scheme of token bus (IEEE 802.4) was feasible for wireless topologies.

CSMA/CD does not work in wireless settings because collision detection is not possible on a wireless channel. The two main problems with collision detection are often referred to as *hidden station problem* and *exposed station problem* (see Figure 2.3). In the hidden station scenario (Figure 2.3a) station A wants to transmit data to another station B but cannot sense that there is already a transmission from station C to station B. Contrary, in the exposed station scenario (Figure 2.3b) station B wants to send to station A and thus senses the channel. It falsely declares the channel busy, because it overhears a transmission from station C to station D. However, there would not be a collision at station A. In addition to these two problems, there exists a technical issue: today's radios typically cannot send and listen on the same frequency at the same point in time. Thus, it is not possible to sense the wireless channel while sending.

Using the MAC scheme of token bus is not feasible, because passing a token over an unreliable wireless channel with stations appearing and disappearing is very difficult to implement and error-prone.

Facing the characteristics of the wireless channel, a new MAC scheme had to be invented. Actually, two schemes, called Distributed Coordination Function (DCF)

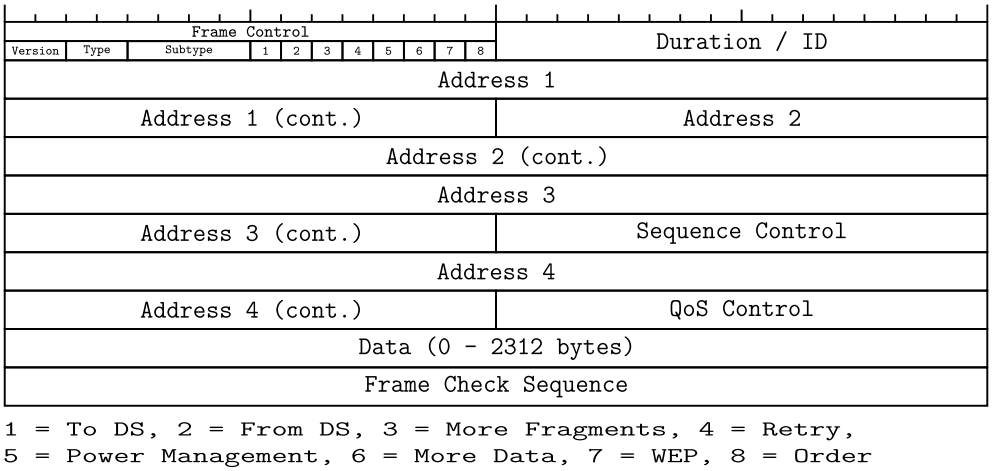


Figure 2.4 The IEEE 802.11 data frame is used to transmit data at the Link Layer.

and Point Coordination Function (PCF), were standardized. While DCF has to be implemented, PCF is declared as optional.

The DCF implements Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) which can use physical as well as virtual channel sensing. Physical channel sensing is realized by listening to the channel before sending for a certain amount of time. If the channel is idle, transmission is allowed. Otherwise, the sender has to wait for a random amount of time, calculated using the truncated Binary Exponential Backoff (BEB) algorithm. However, physical channel sensing does solve neither the hidden station problem nor the exposed station problem. Thus, virtual channel sensing can additionally be used to overcome this issue. A station that wants to start a transmission sends out a Request to Send (RTS) frame to which the receiver responds with a Clear to Send (CTS) frame if the channel is idle. All stations that overhear a RTS, a CTS, or both packets, store the expected allocation of the wireless channel in their virtual Network Allocation Vector (NAV). If its NAV indicates that the channel is busy, a station waits until the channel is indicated as idle before it transmits data. Using these mechanisms, both the hidden station and the exposed station problem can be avoided.

If a central entity is present, PCF can be used. The central entity polls all stations in its wireless range and hence gives them the possibility to request a time slot for transmitting. In wireless networks, the central entity periodically sends out so-called *beacon frames*, which announce its network. When using PCF, these beacon frames are additionally used for clock synchronization and channel management. All stations in the wireless range of the central entity can sign up for the polling service. This way they get a guaranteed share of the wireless channel. Transmission is only allowed during the time slot which was assigned by the central entity. It is also possible to use DCF and PCF in parallel. IEEE 802.11 defines a fine-grained partitioning of the time between two network frames.

Figure 2.4 shows the format of an IEEE 802.11 data frame. Not all fields are used in all types of data frames. This applies to the fields Address 2, Address 3, Sequence Control, Address 4, QoS Control, and Data. Which fields are used implicitly follows from the values of the Type and Subtype fields. Note that the checksum is located at the end of the frame, which is then called a frame trailer.

In traditional IEEE 802.11 wireless networks, data frames have to be acknowledged. This means that the receiver has to send out an acknowledgment frame after successfully receiving an uncorrupted data frame. If the sender does not receive the acknowledgment frame after a certain period of time it considers the frame as corrupted or lost and thus initiates a retransmission.

Quality of Service

In 2005 the IEEE approved the IEEE 802.11e (or IEEE 802.11e-2005) amendment to the IEEE 802.11 standard. It was later included into the IEEE 802.11-2007 version of the standard [26]. IEEE 802.11e introduces Quality of Service (QoS) extensions to the IEEE 802.11 standard. The important changes for the topic of this thesis are the introduction of four access categories and the service class `QoSNoAck`.

Traffic can now be distinguished between the four access categories voice, video, best effort, and background on a per-stream basis [44]. The different access categories are treated differently when contending for medium access. Especially the implementation of the backoff algorithm differs for the four access categories.

If a frame belongs to the `QoSNoAck` service class, no acknowledgment frame has to be sent when receiving this frame. This way it is possible to disable retransmission of corrupted frames at the Link Layer.

2.2.2 Internet Protocol version 4

The Internet Layer protocol which will be used in this thesis is IPv4. It was standardized in 1981 through an RFC [55]. It mainly acts as a routing protocol which enables communication with systems outside the own physical network. It offers several features such as QoS and fragmentation. The header is protected by a checksum. Figure 2.5 shows the structure of the IPv4 header.

The *Version* field stores the version number of the used IP version. For IPv4 this value is for obvious reasons statically 4. Next is the *IP Header Length (IHL)* field, which indicates the length of the header in 32-bit words. The minimum value for this field is 5. QoS is implemented using the *Type Of Service (TOS)* field, which stores information for different QoS implementations. Using the *Total Length* field, the length of the whole IP packet (header and payload) is stored in bytes. Considering the size of the field, IP packets have a maximal length of 65,535 bytes. According to the RFC, each host has to be able to handle packets with size up to 576 bytes. Each datagram has an unique identifier, which is stored in the *Identification* field. This field is used, together with the two flags *Don't Fragment (DF)* and *More Fragments (MF)* as well as the *Fragment Offset* field, to realize datagram fragmentation. However, IP fragmentation is rarely used in today's systems and thus not covered in more detail here. The *Time To Live (TTL)* field is used as hop count. Each system decreases this value by one and drops the packet if the value reaches 0. Thus, infinite travel of a packet (for example in loops) is prevented. IP packets encapsulate transport layer packets. The *Protocol* field stores to which Transport Layer protocol the packet should be passed. Integrity of the IP header is guaranteed by the *Header Checksum*, which is implemented using one's complement.

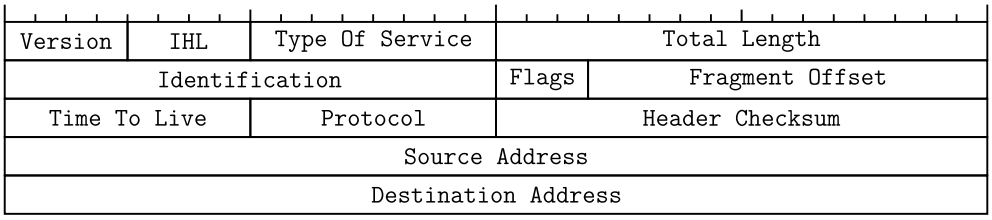


Figure 2.5 The IPv4 protocol header is used to process IPv4 packets.

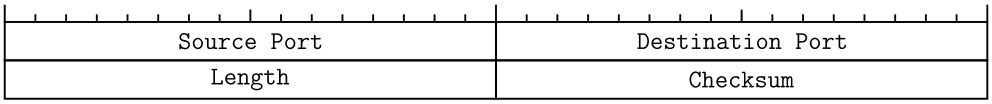


Figure 2.6 The UDP protocol header is used to process UDP packets.

A more detailed description of the checksum algorithm can be found in the RFC [55]. The *Source Address* and *Destination Address* fields store the IPv4 addresses of the sender respectively the receiver. Additionally, the IPv4 header can have up to ten 32-bit words of header options which succeed the destination address entry. As these options are rarely used, a detailed description is omitted.

2.2.3 User Datagram Protocol

UDP is a fairly simple transport layer protocol, which was standardized by an RFC as well [54]. Its main functionality is application multiplexing. Therefore, it uses *port numbers* to distinguish between different applications. UDP also offers integrity as it protects header and payload with a checksum. Communication over UDP is not reliable (packets might get lost, reordered or duplicated) unless reliability has been implemented in the application. The structure of the UDP header is shown in Figure 2.6.

Most important are the *Source Port* and *Destination Port* fields. They offer the possibility to distinguish between different communication endpoints on a single system. Port numbers range from 0 to 65,535, and were divided into three categories by the Internet Assigned Numbers Authority (IANA). The first range from 0 to 1023 is used for so-called well-known services. Additionally, ports from 1024 to 49,151 are used for services which are registered with the IANA. Finally, ports 49,152 to 65,535 can be used freely. They are often referred to as ephemeral ports. The *Length* field stores the total length of the UDP packet (header and payload) in bytes. The minimal value is 8 bytes, which is exactly the size of the header, and the maximal size is 65,535 bytes. However, a UDP packet with the maximal size would be fragmented by the underlying IPv4 protocol, because together with the IPv4 header information the maximal size of an IPv4 packet will then be exceeded. As last element, the UDP header contains a checksum. It covers the UDP header and payload as well as the so-called pseudo header, which contains selected fields of the IP header. Thus, the conceptual separation between the two layers and their protocols is broken up.

Lightweight User Datagram Protocol

The Lightweight User Datagram Protocol (UDP-Lite) is a modification of the original UDP standard. It was proposed by an RFC [35] and allows partial checksums. Thus, it becomes possible to receive partially corrupted packets. The *Length* field from UDP is replaced by the *Checksum Coverage* field. This field indicates the number of bytes which should be covered by the checksum. The header always has to be covered by the checksum, but it is possible that the payload is not protected by a checksum at all. This is especially interesting for audio or video streaming applications which can cope with partially corrupted data.

2.2.4 Real-Time Transport Protocol

RTP is a packet-oriented Application Layer protocol which is used to ease end-to-end transportation of real-time critical data such as audio or video streams. Its current version, two, is standardized by an RFC [58]. Although RTP is standardized to work with a wide range of Transport Layer protocols, it typically utilizes UDP. This is because UDP is suited better for real-time requirements than TCP. Other Transport Layer protocols than UDP and TCP are rarely seen on the Internet.

As RTP is not a primary target of this thesis and is mainly treated as Application Layer data, a detailed description of its header fields is left out. The RTP header allows for an in-order, timed playback of the received data. There is no checksum to protect header or data as this is the task of the Transport Layer protocol.

RTP is used in conjunction with other Application Layers protocols. Together with RTP, the Real-time Transport Control Protocol (RTCP) is standardized. It synchronizes the RTP stream and collects statistics on lost packets, packet jitter, and round-trip time. This information can be used by the application to ensure QoS, for example by decreasing the used bandwidth. In addition, a signaling protocol for initiating and closing a connection can be used. Typical examples for signaling protocols are H.323 and Session Initiation Protocol (SIP).

As RTP is designed for streaming audio and video data, it offers support for a wide range of different audio and video codecs. Some example codecs are Vorbis, Theora, MPEG-4, MP3, and various GSM codecs [25].

2.3 Errors in Wireless Transmission

In contrast to wired transmission, which is nowadays essentially error-free, there are several reasons for errors in wireless transmission. Schiller names several reasons for the occurrence of errors in the wireless signal [57]. Basically, wireless signal propagation follows a straight line, the so-called Line-of-Sight (LOS). However, wireless communication nearly always takes place outside the LOS. The wireless signal is reflected by obstacles or refracted when passing through objects. Although especially reflection is crucial for transferring the wireless signal, the signal gets weaker with every step of reflection. There are also effects which are even worse. The most intuitive one is called shadowing or blocking. Objects, for example steel

beams, are standing in the LOS and thus absorbing the signal. Additionally, the wireless signal might be scattered or diffracted when hitting objects. This means that the signal is spread up into several signals at (slightly) different frequencies. These signals will reach the receiver at different points of time and thus interfere with each other.

A lot of effort is put into different modulation schemes which try to minimize the effects of the above-mentioned phenomena. However, there will still be errors in the network packets which are processed by the end hosts. When considering typical coding schemes, this means that one or more bits of the packet are flipped, that is, they have the inverse value.

The interesting question is, how these bit errors are distributed within a network packet. When ignoring physics, it could be expected that bit errors are uniformly distributed over a network packet. Willig et al. conducted measurements and state that bit errors are not equally distributed over a network packet [69]. They showed that the error distributions for the two modulation schemes Quadrature Phase-Shift Keying (QPSK) and Binary Phase-Shift Keying (BPSK) observe a common pattern. Additionally they state that the other modulation schemes each follow a typical pattern as well. For QPSK and BPSK a peak is observable approximately within the first 80 to 250 bits, depending on whether scrambling (changing bit positions in order to better cope with the characteristics of the wireless channel) is used during coding or not. From this point onwards, smaller peaks occur with a periodicity of 128 bits for QPSK respectively 64 bits for BPSK. There is no sound explanation for this surprising phenomenon, but it is thought that this is due to artifacts of the bit synchronization algorithm of the wireless receiver.

Thus, in the setting considered in this thesis, it can be assumed that the large peak occurs within the IEEE 802.11 header. This header is not covered by the methods presented later on. For the sake of simplicity, the smaller bursts in the Link Layer payload are not considered. Simulations will only be performed to evaluate parameter settings. As choosing the right parameter values should only negligibly influenced by the distribution of bit errors within one packet, an uniform distribution of bit errors in the Link Layer payload will be assumed. In order to gain more certainty, real measurements in a wireless network will be performed. These measurements will unavoidable observe the real bit error distribution.

2.4 Linux Kernel Networking

The Linux kernel is an open source operating system kernel which ships with an implementation of the TCP/IP model. Because of the availability of the source code and the increasing deployment of Linux as an operating system, using its kernel for network related research is a straightforward choice. Additionally, a large number of documentation and literature on Linux kernel programming and especially on Linux kernel networking is available.

Throughout this thesis, version 2.6.32.15 [39] of the Linux kernel will be used. This version was chosen for two reasons. Firstly, the 2.6.32 release is maintained as a long-term release. This guarantees security updates for two to three years for the kernel

itself. Secondly, it is shipped with the long-term support version 10.04 of the widely used Ubuntu distribution [66], which offers security updates for the surrounding software for the period of three years.

In the remainder of this section, the most important structures for handling incoming packets in the Linux kernel will be discussed. Additionally the path of an incoming packet through the network stack will be sketched. Finally, the Linux kernel implementation of linked lists as an essential data structure will be presented.

2.4.1 The `sock` Structure

A socket is the endpoint of a bidirectional data flow in a network. For communication purposes, it is the interface between user space and kernel space. This means that it also acts as interface between Transport Layer and Application Layer. It offers the application a standardized way to access the communication services offered by the network stack.

The Linux kernel utilizes the Berkeley Software Distribution (BSD) socket interface. The `sock` structure (see section A.4.2) stores the used protocols of a socket and their states. Therefore, it can be extended by other structures, for example the `inet_sock` structure, to refine the stored information. Extending of structures in C is done by putting the parent structure as first element into the child structure. Doing so, the structure can be cast to both types and accessed in two ways. This is a very simple way to use object-oriented like inheritance for data structures in C.

Basically, the `sock` structure is the representation of a socket. It extends the `sock_common` structure, which holds the bare minimum needed for a representation of a socket.

For this thesis, the most important information stored in the `sock` structure are the `sk_family` and `sk_protocol` fields. The `sk_family` field stores the type of the protocol family used with this socket. Typical values are `PF_INET` (IPv4) and `PF_INET6` (IPv6). Similarly, the `sk_protocol` field holds the type of the transport layer protocol. Thus, the interesting values for this thesis are `IPPROTO_UDP` and `IPPROTO_TCP`.

Later in this thesis, the `sock` structure will be utilized to enhance the socket interface with a possibility to signal corrupted packets.

2.4.2 The `sk_buff` Structure

The `sk_buff` structure (see section A.4.1) is the structure in the Linux network stack which allows to access the data of a network packet. Benvenuti [7] gives an extensive overview about the details of this structure. As the major interest of this thesis lies in using `sk_buff` structures and not in managing them, only the usage will be discussed in the following. Thereby the focus will be put on using existing `sk_buff`s, as for incoming packets they are created by the network device respectively its driver.

Each network packet has a designated packet type, which is stored in the `pkt_type` field. Interesting values for incoming packets are `PACKET_HOST`, `PACKET_MULTICAST`,

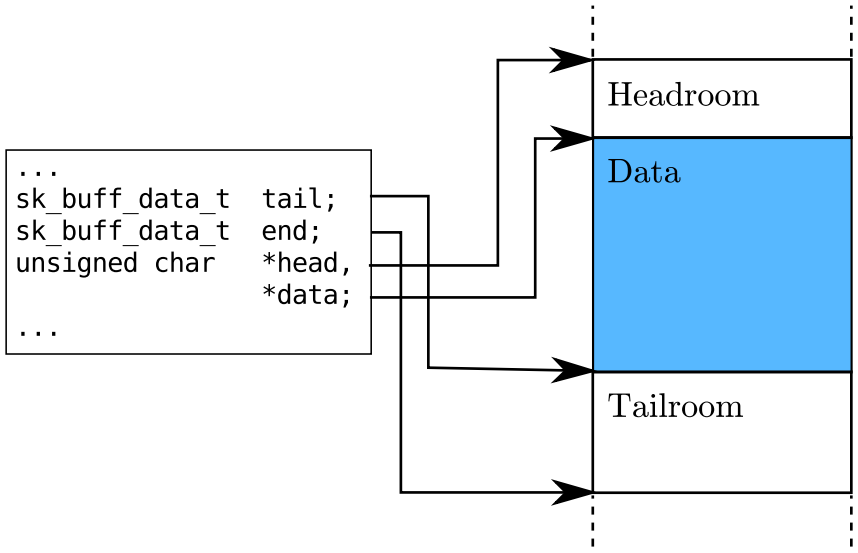


Figure 2.7 Pointers are used to access the `sk_buff` data buffer.

and `PACKET_BROADCAST`. Received unicast packets which are designated for this host are marked with `PACKET_HOST`. The other two types intuitively signal broadcast and multicast packets to be handled by this host.

The device driver uses the `protocol` field to signal the Internet Layer protocol to whose handler the received packet should be passed. Most common values for this field are `ETH_P_IP` (IPv4), `ETH_P_IPV6` (IPv6), and `ETH_P_ARP` (Address Resolution Protocol (ARP)).

At the Transport Layer, the socket to which the packet should be delivered is determined and stored in the `sk` field, which points to a `sock` structure.

Accessing Packet Data

The actual packet data is stored in a buffer which is accessed by the four pointers `head`, `data`, `tail`, and `end` (see Figure 2.7). Thereby `head` points to the start of the buffer space and `end` to the end of the buffer space. Similarly `data` points to the beginning of the actual data and `tail` to the end of the actual data. As a packet travels up the stack, `data` is set to the begin of each layer's header and thus moves on forward in the packet data. In addition, the pointers `mac_header`, `network_header`, and `transport_header` offer access to the header of the protocols for the indicated layer. They are set as soon as the corresponding layer is reached. The length of the data stored in the buffer is stored in the `len` field. Its value decreases while the packet travels up the stack.

2.4.3 Receiving a Network Packet

Understanding how received network packets travel within the network stack is essential for understanding the approach proposed by this thesis. Thus this section presents a detailed walkthrough for a received IPv4/UDP packet.

Sevy gives a very detailed documentation on how network packets travel within the Linux network stack [59]. Although his walkthrough is based on the old 2.4.20

version of the Linux kernel, most of it is still applicable today. Thus, the following explanation is based on his information and the experience made while implementing the code for this thesis. It is also stated at which layer in the TCP/IP Model each step is located. Of course not all details can be covered, but everything which is necessary to understand how network packets are handled is mentioned.

At the Link Layer, the network card driver calls the `netif_rx()` function which is responsible for enqueueing the passed `sk_buff` for later processing and then initiating the receive process. After enqueueing has finished, `netif_rx_schedule()` is called to signal the `NET_RX_SOFTIRQ` Software Interrupt Request (softirq). Thus, the signal handler for this softirq, `net_rx_action()`, is called. Using helper functions to access the right queue, the signal handler finally calls `netif_receive_skb()` which passes along the `sk_buff`. This is the main receive function at the Link Layer. It reads the `protocol` field from the passed `sk_buff` and calls the handler routine which was registered for this protocol. As IPv4 is considered here, the registered handler is `ip_rcv()` and by calling this function the `sk_buff` is passed to the Internet Layer.

The main IPv4 receive routine `ip_rcv()` and its helper function `ip_rcv_finish()` perform sanity checks on the IPv4 header and the passed `sk_buff`. If the packet is not dropped due to these checks, the route for the packet is calculated. In the context of this thesis the packet has to be delivered locally which results in the call of `ip_local_deliver()`. With the help of `ip_local_deliver_finish()`, the packet is defragmented if necessary and then the Transport Layer protocol is read from the IPv4 header. Then the corresponding protocol handler for the specified protocol is called. In this example this protocol is UDP and the protocol handler `udp_rcv()`. By passing the `sk_buff` to this function, the packet has reached the Transport Layer.

As `udp_rcv()` is a wrapper for `__udp4_lib_rcv()`, the later function actually is the main receive function for UDP packets. After performing sanity checks, the socket to which this packet has to be delivered has to be discovered. Therefore `__udp4_lib_lookup_skb()` is called which performs, using a few helper functions, the task of looking up the right socket by comparing source IPv4 address, destination IPv4 address, source port number, and destination port number with stored values. As the right `sock` structure the packet belongs to, is now known, the `sk_buff` can be enqueued on this `sock`'s receive queue using the function `udp_queue_rcv_skb()`. The packet has now reached the Application Layer.

The handling of a network packet in the Application Layer is now described from the point of view of the application. If the application calls the `recv()`, `recvfrom()`, or `recvmsg()` function from the socket interface, the handler function `udp_recvmsg()` is called for UDP sockets. If there is a packet in the `sock`'s receive queue, this packet is passed to the application, otherwise the function blocks. This is the point where the packet is finally passed to the Application Layer and its journey through the network stack ends.

2.4.4 Linked Lists

There are only few data structures available in the Linux kernel. Most commonly used are linked lists, which are implemented as doubly circular linked lists. Linked

```
struct list {
    void        *item;
    struct list *next;
    struct list *prev;
};
```

(a) Classical

```
struct item {
    struct list_head list;
    int    data1;
    // more data ...
};
```

(b) Linux kernel

Listing 2.1 Two different approaches to the implementation of linked lists can be distinguished.

```
#define list_entry(ptr, type, member) \
((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

Listing 2.2. The (simplified version) of the `list_entry` macro is used to access a list entry.

lists are often used to implement queues and stacks, but also serve as foundation for the implementation of more abstract data structures, for example hash tables. Corbet, Rubini and Kroah-Hartman give a brief introduction into the Linux kernel's linked list implementation [11] and Shanmugasundaram offers a deeper understanding of this implementation [61].

As the Linux kernel's version of linked lists will be used later on in this thesis to store collected data, its functionality is described in the remainder of this section.

Linked lists are typically implemented such that the items that should be linked are included within a data structure that implements the list [12, 33]. Contrary, the linked list implementation in the Linux kernel follows a different approach. The data structure that implements the list is included within the items that should be linked. See Listing 2.1 for a comparison of the two different approaches.

Using the Linux kernel's implementation, only one structure is necessary for putting the same item into a fixed number of different lists.

Each list is initialized by initializing a sentinel list head structure using the `INIT_LIST_HEAD()` macro. Using a sentinel list head is a well-known technique for implementing doubly circular linked lists [12].

There is a `list_add()` and a `list_del()` function to add and delete list items. There also exists the `list_for_each_entry()` macro to iterate over a list and the `list_for_each_entry_safe()` macro to iterate over a list where elements might get removed while iterating. Locking is not implemented for the kernel's linked lists. Thus, locking has to be implemented by the programmer if concurrency cannot be excluded.

The implementation of the several list functions in the Linux kernel is mainly straightforward. However, it is interesting to observe how the ambient structure can be accessed if only the address of one of its elements is known. This is basically implemented by a macro that performs pointer arithmetic (see Listing 2.2 for a simplified version of this macro). The idea is to take the address of the member and subtract the offset of the member within the structure. To calculate the offset, the memory address 0 is casted to the type of the structure and then the address of the

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};
```

Listing 2.3. The `work_struct` structure stores information on queued work.

member of interest is read. Thus, the offset of the member is known and can now be subtracted from the address of the member of an instance of the structure.

2.4.5 Work Queues

Processing of network packets mostly takes place in interrupt context. Thus, not all operations are possible. For example, as tasks running in interrupt context are not schedulable, they cannot sleep. However, sleeping is required for numerous tasks such as allocating large amounts of memory or blocking I/O. One way to overcome these limitations is to defer work to a process which is not running in interrupt context. The Linux kernel offers a convenient concept for this purpose which is called *work queue*. An introduction to work queues is given by Love [41]. The piece of work which requires sleeping is scheduled to a queue. A kernel thread (the so-called *worker thread*) running in process context periodically polls the queue and performs the scheduled task.

Using Linux kernel's work queue is straightforward. An instance of the `work_struct` (see Listing 2.3) has to be created. The `data` field stores information which should be passed to the handler function `func`. With the `entry` field, the item is put into the actual work queue, which is implemented as a linked list (see Section 2.4.4). This shows that linked lists are commonly used in the kernel where they serve as an efficient data structure. When the worker thread wakes up, it looks at its queue. If the queue is empty, it goes back to sleep. Otherwise, it removes the first entry from the queue and calls the handler function `func`, passing a pointer to the `work_struct`.

2.5 Cross-Layering

Srivastava and Motani motivate the usage of cross-layering and give a brief definition of different cross-layer design approaches [62, references therein]. As seen in section 2.1, protocols are only allowed to communicate with protocols of the immediately upper respectively lower layer and specified interfaces have to be used for this communication. Especially in wireless networks, it can be argued that breaking up these strict separations can lead to a performance increase. One example for the poor performance of a strictly layered architecture is TCP over a lossy wireless link [60]. The sender will interpret lost packets as congestion and thus reduce the sending rate. This might lead to an enormous performance fall-off.

According to Srivastava and Motani [62], breaking up the strict separation of layers can in principle be done in four ways:

- *Creation of new interfaces:* New interfaces between not necessarily adjacent layers are introduced. Upwards interfaces go from lower layers to upper layers, downwards interfaces from upper layers to lower layers. Back and forth interfaces implement an iterative communication between two or more layers.
- *Merging of adjacent layers:* Two or more layers are grouped to one super layer which communicates to the adjacent layers using the existing interfaces.
- *Design coupling without new interfaces:* Two layers are designed such that changes to one layer necessarily imply changes to another layer. Thus, it is not possible to exchange one layer without touching the other ones.
- *Vertical calibration across layers:* One or more parameters are adjusted at different layers within the network stack.

As the layout of the network stack should only be modified slightly in this thesis, there is no plan to implement a complete cross-layer design. Thus, only the implementation of cross-layer communication will be deepened now.

Cross-Layer Communication

Cross-layer communication can be realized in different ways. Again, the classification by Srivastava and Motani who classify the existing cross-layer communication approaches into three classes [62] is used:

- *Direct communication between layers:* The layers communicate using shared variables. These variables can be passed along with the packets or using internal packets.
- *A shared database across the layers:* Communication takes place over a shared database. Each layer can read from and write to the database.
- *Completely new abstractions:* The layered protocol hierarchy is completely abolished and a new abstraction created.

The approach presented in this thesis makes use of the first and the second class described above. As cross-layering is a rapidly developing field of research, these classifications might evolve over time. For this thesis it is important to have a motivation for breaking up the strict layer boundaries and knowledge about different approaches in implementing cross-layering within an existing network stack.

2.6 Data Mining and Machine Learning

Data mining and machine learning are two closely related fields in the area of extracting patterns from large amounts of data. An introduction to data mining is given by Han and Kamber [22], background information on machine learning is given by Mitchell [48].

Attributes				Label
Outlook	Temperature	Humidity	Wind	Play tennis?
overcast	medium	normal	weak	yes
sunny	hot	high	weak	no
rain	cool	high	strong	no
sunny	hot	normal	weak	yes

Table 2.1 A training set is used to build a classifier. Each row contains one tuple of attributes.

Data mining and the whole Knowledge Discovery in Databases (KDD) cycle, of which data mining is an essential part, are the use of methods to extract patterns or interesting information from very large amounts of data. Often methods from the field of stochastics and mathematics in general are used. Data mining is widely used in commercial scenarios where large amounts of data need to be analyzed. Some typical scenarios are market basket analysis, credit card fraud detection, customer relationship management, and quality control. Typically, data mining is applied to relational databases, transactional databases, or data warehouses. But there are also more advanced fields of application, like the World Wide Web or data streams.

Machine learning, on the other hand, comes from the area of artificial intelligence. A machine learning system learns from examples and is then able to use this experience to make generalized assumptions. Again, the fundamental methods are often taken from mathematics in general and stochastics in particular. Typical fields of application include automatic speech recognition, optical character recognition, object recognition, and autonomous systems. In contrast to data mining, machine learning is not necessarily applied to databases or similar storage concepts. Often there are only the training data which were collected only for the purpose of training the machine learning system.

In both fields, data mining and machine learning, there is the differentiation between classification (or supervised learning) on the one hand and clustering (or unsupervised learning) on the other hand. Due to the relevance for this thesis, only the former will be discussed in more detail now.

2.6.1 Classification / Supervised Learning

Classification or supervised learning is an area that is located in the intersection of data mining and machine learning. The approach this thesis proposes is heavily based on the concept of supervised learning, hence the concept will be described in detail and important definitions will be introduced.

Using supervised learning is a typical approach which is used to build a *classifier* in data classification. Typically, the classifier is constructed once and then applied to the data that should be classified. As shown later on, this approach is not feasible for the setting proposed by this thesis. To ease the understanding of the underlying concepts, the general case will be introduced nevertheless.

The classifier is trained using a *training set*. The training set consists of labeled tuples of attributes. Thereby the *label* denotes the class the tuple of attributes

belongs to. Table 2.1 shows an example of a training set with the two classes "yes" and "no". While processing the data from the training set, the *classification algorithm* builds up the data structures needed for building the decision rules. After the training phase the classifier can be used for classification. It receives a tuple of attributes – for example, ("sunny", "cool", "medium", "weak") – as input and outputs the predicted label ("yes" or "no") for this entry. If the quality of the classifier should be tested, a *test set* can be used. The labels for the entries in the test set are known but are not shown to the classifier. Thus, the output of the classifier can be compared with the correct label. This is often used to guarantee a certain quality of the classifier. The classifier is trained using the training set until the results of classifying the elements from the test set exceed a certain threshold.

2.6.2 Classification of Data Streams

Typical approaches to supervised learning require multiple or recurring scans over the whole data set. When considering data streams, this is not feasible any more. A *data stream* is a possibly infinite flow of data that arrive with varying pace. Typical fields in which data streams occur are sensor networks, stock-market transactions, power grids, measurements of experiments, and network traffic. Typically it is not possible to store all items of a data stream. Thus, the data stream can be scanned at maximum once while the data already flows by. This is then called *on-line processing* of data streams. Sometimes the data arrive so fast that it is not even feasible to look at all data once.

Several approaches exist that address the problems which are induced by classifying data streams. However, classification of data streams is a trade-off between storage, processing time and classification accuracy. A typical approach is the use of *sliding windows*. Only a fixed number of data elements is used for computation. Whenever a new data element arrives, it evicts the oldest element in the sliding window. This approach especially addresses the issue that in some of the scenarios the data might change over time. Using sliding windows eliminates the influence of very old data on the classification. Another widely used approach is the utilization of *histograms* to approximate an underlying probability distribution. The value range is partitioned into buckets with possibly varying width. Each data element is put into the bucket its value falls into by simply incrementing the count of this bucket. This method is only able to approximate the probability function for selected values and offers no possibility to examine correlations between different values of the same data element. As an addition to nearly all other approaches *random sampling* can be used. This is especially a good idea if data arrives too fast to process all of it. The idea is to take random samples from the data stream and only operate on them. The challenge is the statistical representative selection of elements from the data stream.

As stated earlier, classification of data streams requires aggregation and/or ignoring of data. The approach which is chosen for a certain scenario always has to be tailored to the constraints of that specific scenario.

3

Conceptual Design

This thesis introduces a new paradigm in handling corrupted packets. Correctly received packets are used to build a classifier which is then used to predict the destination socket of corrupted incoming packets. If an application signals that it is capable of handling corrupted packets, this prediction can be used in Internet and Transport Layer protocol implementations as packet handling decision rule.

At first, objectives, challenges, and constraints will be presented. Then the design principle of the packet-classification tool will be introduced. At the end of this chapter a design of an use case, which serves as an example how the packet-classification tool can be utilized, is given.

3.1 Objectives

This thesis aims at evolving a new scheme of packet handling. Similar to the approach taken by UDP-Lite [35], it is desirable to have the possibility to receive packets with corrupted payload. Furthermore, even packets with errors in the Internet or Transport Layer header should be processed.

The main objective of this thesis is the design and implementation of a packet-classification tool which is based on supervised learning within an existing network stack and thus within an existing operating system.

There are mainly two requirements, which have to be met. Firstly, as many corrupted packets as possible should be passed to the correct application. Secondly, as few packets as possible should be delivered to a wrong application. These two requirements are (at least partly) contrary to each other. Thus, balancing both requirements is a major task within this thesis.

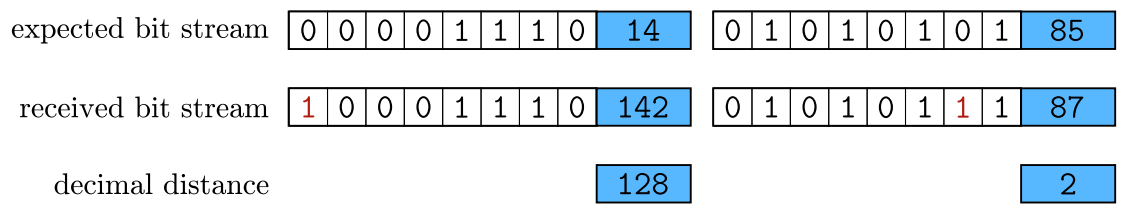


Figure 3.1 Bit errors have a variable impact if binary data is interpreted as decimal numbers.

3.2 Challenges and Constraints

Especially when adapting a complex system such as the Linux kernel, careful considerations have to made. This section presents the major challenges and constraints which had to be considered while both designing and implementing the approach addressed by this thesis.

Error-prone Data

The algorithm, which had to be invented, learns from error-free data, but has to classify error-prone data. This is a major difference to typical data mining or machine learning approaches, which are presented in Chapter 6.

As discussed in Section 2.3, in this thesis it is assumed that errors occur everywhere in the payload of a Link Layer packet with roughly the same probability. Thus, it is not feasible to interpret the bit string of a network packet as a string of characters. A character is created by evenly dividing the bit stream into groups of bits and assigning a representation to each of these groups. Errors in the most significant bit of one of the groups would completely change the meaning of the received character. This problem is illustrated by Figure 3.1, where each byte is interpreted as a decimal number. Although there is exactly one error in both bytes, the decimal distances differ substantially.

Thus the proposed solution has to cope with error-prone data, no matter at which positions errors in the bit stream occur. For this purpose, a novel way of learning bit-oriented data has to be considered.

Avoiding False Assignments

The approach proposed by this thesis is based on statistics. Thus, errors in the assignment cannot be fully excluded. However, assigning a network packet to the wrong application directly raises two problems. The major problem is that an application receives a packet which is not designated for it. This can have several implications. In the best case, the application is able to recognize this fact and simply ignores the packet. In the worst case, the application will crash because it does not expect malformed data. The minor problem is that the application for which the packet was originally designated does not receive the packet. This is a problem which can be solved by the used Transport Layer protocol (for example, by using sequence numbers) or has to be handled by the application itself. Thus, an implicit

or explicit retransmission of the packet is provoked. If such a mechanism is used, the application will only suffer from the delay which is introduced by the retransmission.

As false assignments are a serious problem, the design of a packet classification tool has to avoid false assignments to the most possible extent. Additionally, methods to signal potentially malformed data to the application should be considered.

Reducing Overhead

Each extension of a network stack comes with the risk of introducing computational overhead to a time critical environment. Processing network packets partly takes place in interrupt context and thus should execute as fast as possible. Even if outside interrupt context, processing times should be kept short as an increase in processing time decreases throughput.

Even today, memory consumption is still an issue. The amount of required space of Random-Access Memory (RAM) should be limited to the bare minimum.

All in all, special considerations to reduce the introduced overhead – in both time and space – have to be taken.

Minimizing Network Stack Modifications

The proposed approach shall extend the functionality of the Linux network stack. However, actual modifications to the code of the network stack implementation should be minimized for two reasons. Firstly, it has to be ensured that the processing of uncorrupted packets does not change. This is especially important when later on comparing measurement results with unmodified implementations of the network stack. Secondly, few, carefully performed modifications maximize upstream compatibility with new versions of the Linux kernel.

No Use of Floating-Point Numbers

Not only challenges but also constraints induced by the used environment have to be considered. As Love points out [41], floating-point numbers should not be used within the Linux kernel. The Floating-Point Unit (FPU) or its emulation is reserved for user space applications. Although using the FPU in kernel space is theoretically possible, it requires an enormous overhead. The floating-point registers have to be stored and restored manually. Additionally, the code must neither sleep nor block while using the FPU. Otherwise, the registers which are in fact used by an user space application might contain wrong values when control is handed over to the application again.

Thus, calculation is limited to fixed-point arithmetic. Risks, namely overflows and precision loss, have to be considered.

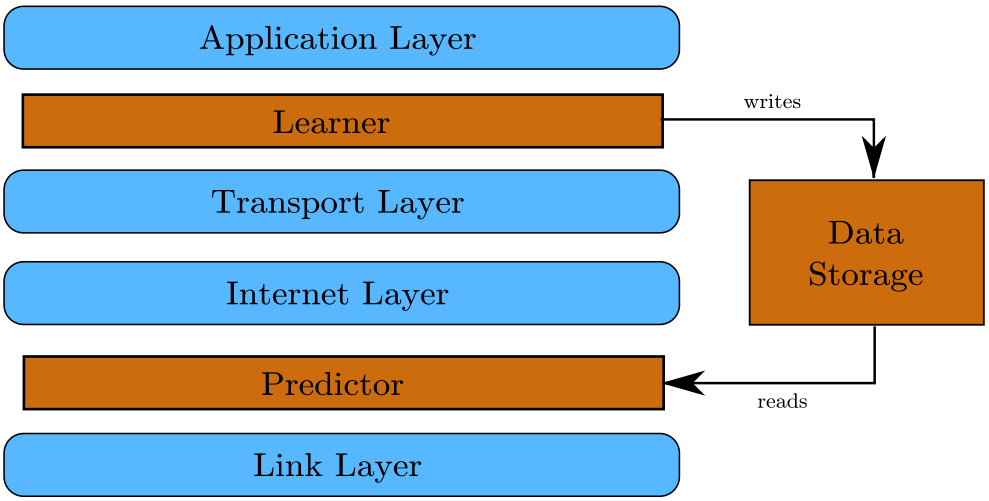


Figure 3.2 The learner and the predictor are integrated into the network stack and communicate over a common data storage.

3.3 Packet-Classification Tool

The packet-classification tool is located within the network stack. Its duty is to predict destination sockets of corrupted incoming packets. This information is then stored along with the packet data. Using this information, Internet and Transport Layer protocols can handle corrupted packets, which would be dropped otherwise.

Conceptually, the approach consists of two main components, the *learner* and the *predictor*. The learner operates only on uncorrupted packets, while the predictor is only invoked for corrupted packets. They communicate in a cross-layer fashion using a shared data storage and information which is passed alongside the received packets. The learner writes data to the storage while the predictor reads data from the storage (see Figure 3.2).

The following conventions are used during the description of the conceptual design of the packet-classification tool: Variables are set in *italic* and parameters of the packet-classification tool are denoted by the `typewriter` font. Bits are transformed to bipolar notation, which eases the later on described calculations. A bit $x \in \{0, 1\}$ is mapped to its bipolar representation $f(x) \in \{-1, 1\}$ as follows:

$$f(x) := \begin{cases} -1 & , x = 0 \\ 1 & , x = 1 \end{cases}$$

3.3.1 The Learner

The learner inspects all uncorrupted, incoming packets right before they are passed to the application. Thus, it has to be located at the interface between Transport and Application Layer (see section 2.1). Its job is to extract information from the packets which later on will be the basis for the predictor's decision. Typical end hosts observe an enormous amount of incoming (and outgoing) packets. Additionally, the predictor only has a very limited amount of time to make its decision. Thus, it is not

feasible to simply store the data of each incoming packet. The packet data needs to be aggregated somehow. In the following will be shown how aggregation of packet data is done in the packet-classification tool.

Aggregation of Packet Data

The aggregation of data for incoming packets is done on a per network socket basis. For each network socket that received at least one packet, there exists exactly one representative that summarizes the received data.

First of all, the beginning[1] of each incoming packet is cut into `NUMBER_OF_CHUNKS` chunks of size `SIZE_OF_CHUNKS` bytes. It is advisable to choose these parameters such that the Internet and the Transport layer protocol header are covered. For each socket and each chunk, two items are stored:

1. A vector $V \in [-1, 1]^{\texttt{SIZE\_OF\_CHUNKS} \cdot 8}$ which holds a mean value for each bit of this chunk.
2. A weight factor $w \in [0, 100]$ which illustrates the inter-packet variance of this chunk.

Additionally, for each socket, the number of received packets n is stored. This number is incremented after all aggregation steps for that packet have been processed.

With X_i, $i \in \{1, \ldots, \texttt{SIZE\_OF\_CHUNKS} \cdot 8\}$, the i-th bit of the chunk X is referenced. Similarly, V_i denotes the i-th entry of the vector V. On every reception of a chunk X, the vector V for that chunk is updated as follows:

$$V_i := \frac{n \cdot V_i^{old} + f(X_i)}{n+1}, \forall i$$

In order to limit the influence of rather old values, an aging parameter `MAX_PACKETS` which acts as an upper bound for n is introduced. This way the influence of old data decreases with every new packet once the number of packets exceeds `MAX_PACKETS`.

The weight factor w for a vector V is updated after every recomputation of the vector and is defined as

$$w := \left(\sum_i |V_i| \right) \cdot \frac{100}{\texttt{SIZE\_OF\_CHUNKS} \cdot 8}$$

Thus, a weight factor w of 100 denotes that the values of all bits in the chunk are the same in each packet of the stream. Contrary, a weight factor w of 0 shows that the values of the bits in the chunk are changing frequently as new packets arrive.

The whole process of aggregation is illustrated in Figure 3.3. Seven incoming packets are shown which all belong to the same socket. Each row represents one packet. To

[1] Which, in this case, is at the beginning of the Link Layer payload respectively the Internet Layer header.

Chunks X	Count n	Vectors V	Weights w
1 0 1 0 1 0 1 0	0	1.0000 -1.0000 1.0000 -1.0000 1.0000 -1.0000 1.0000 -1.0000	100.00
1 1 0 0 1 1 0 0	1	1.0000 0.0000 0.0000 -1.0000 1.0000 0.0000 0.0000 -1.0000	50.00
1 1 1 0 1 1 1 0	2	1.0000 0.3333 0.3333 -1.0000 1.0000 0.3333 0.3333 -1.0000	66.67
1 1 0 1 1 0 1 1	3	1.0000 0.5000 0.0000 -0.5000 1.0000 0.0000 0.5000 -0.5000	50.00
1 0 1 0 1 0 1 0	4	1.0000 0.2000 0.2000 -0.6000 1.0000 -0.2000 0.6000 -0.6000	55.00
1 0 0 1 0 0 1 0	5	1.0000 0.0000 0.0000 -0.3333 0.6667 -0.3333 0.6667 -0.6667	45.83
1 1 1 1 0 0 0 0	5	1.0000 0.1667 0.1667 -0.1111 0.3889 -0.4444 0.3889 -0.7222	42.36

Figure 3.3 Incoming packet data is aggregated. `SIZE_OF_CHUNKS` = 1 byte, `MAX_PACKETS` = 5.

simplify matters only the first chunk of each packet is considered. For each chunk, the first step is the update of the vector V using the above-defined formula. This step can by done separately for each bit. After that, the weight w for this vector can be computed. After processing of a packet is done, the counter n is incremented.

Considering the last version of V, the first entry shows that the first bit of all packets that belong to the socket is always 1. Similarly, the last entry indicates that the eighth bit of these packets is nearly always 0. Contrary, the second, third and fourth entry express that the corresponding bits in the packet fluctuate between 0 and 1. The weight factor of the last vector illustrates that this chunk contains a certain degree of unsteadiness.

3.3.2 The Predictor

The predictor is invoked for all corrupted incoming packets right before they are passed to the Internet Layer protocol handler. Thus, it is located at the interface between Link Layer and Internet Layer. The decision if a packet is corrupted depends on the Link Layer protocol's Frame Check Sequence (FCS).

For each packet it has to classify, the predictor compares its data with the stored aggregated data of all open network sockets. Again, the incoming packet is sliced in chunks. Then the comparison is done by comparing chunk by chunk and applying the weight for each chunk.

Let $V^{j,k}$ be the vector of the k-th chunk of the j-th socket and X the received chunk. Then the chunk score $s^{j,k}$, which denotes the conformity of the received chunk and the aggregated chunk data, is defined as:

$$s^{j,k} := \left(\sum_i V_i^{j,k} \cdot f(X_i) \right) \cdot \frac{100}{\texttt{SIZE\_OF\_CHUNKS} \cdot 8}$$

A positive chunk score $s^{j,k}$ denotes a positive correlation between the two chunks, while a negative chunk score denotes a negative correlation between the two chunks.

When the chunk score is 100, the chunks match exactly. The contrary holds for the chunk score of -100 which indicates that the two chunks are completely different. If the chunk score is close to 0, the both chunks are not correlated at all.

Using the weights $w^{j,k}$ it is then possible to compute the packet score S^j, which is a measure of the similarity between the received packet and the aggregated data for the network socket j:

$$S^j := \frac{\sum_k \left(w^{j,k} \cdot s^{j,k}\right)}{\sum_k w^{j,k}}$$

A positive packet score again indicates a positive correlation between the packet and the socket while a negative packet score indicates a negative correlation between the packet and the socket. Note that the packet score is implicitly normalized by the normalization of the chunk scores.

After the packet scores for all network sockets have been computed, it is now possible to select the network socket *pred* the packet most probable belongs to:

$$pred := \arg\max_j(S^j)$$

Thus, the packet is predicted to belong to the socket with the highest packet score.

In order to make it less likely that a packet gets assigned to the wrong network socket, two additional protections can be used (where `null` indicates that no prediction is possible):

1. If the number n of uncorrupted packets that the predicted socket already received is smaller than the parameter `SKIP_FIRST_PACKETS`, the prediction is changed to `null`.
2. Compare *pred* with $pred' := \arg\max_{j \neq pred}(S^j)$, the second best match. If $S^{pred} - S^{pred'} <$ `MIN_DELTA` then the match is too close to the second best match and the prediction is changed to `null`. Again, `MIN_DELTA` is a parameter.

A label for the predicted network socket can now be attached to the packet data. This information is thus provided to the Internet and Transport Layer protocols. How this information can be used to process corrupted packets in the network stack will be shown in an use case.

3.4 Use Case

To make use of the information provided by the above described Packet-Classification Tool, the network stack of an operating system has to be modified. In this thesis, this is done for the IPv4 and UDP implementation of the Linux kernel. The combination of IP and UDP is widely used for multimedia applications. Additionally, both are

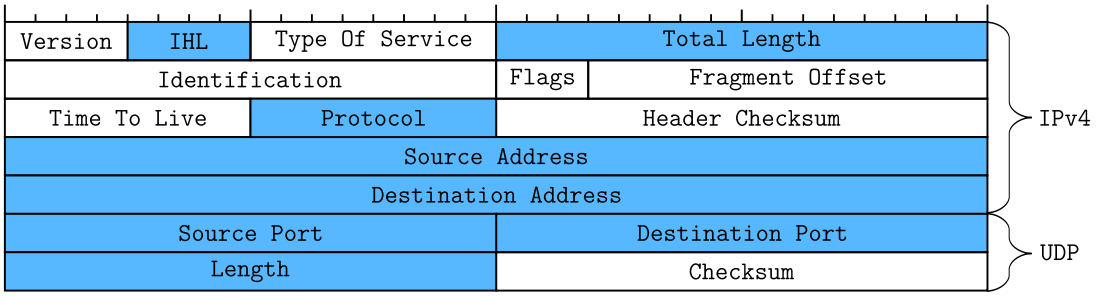


Figure 3.4 IPv4 and UDP header fields which are especially important for processing a packet to upper layers are highlighted.

stateless protocols. Thus, corrupted control data of one packet have (nearly) no impact on the handling of other packets of the same stream.

From the design point of view, the modifications to the IP and UDP protocol implementations are rather simple. Because only incoming packets are handled, only the input routines have to be considered.

As stated earlier, passing corrupted packets to an application which is not able to handle them is dangerous. This leads to the need of an interface, which allows to communicate packet errors to the application. The application has to be able to state that it is willing to receive corrupted packets. Whenever a packet is delivered to the application, it is informed whether the packet is erroneous or not.

The locations at which packets are dropped due to a failed checksum have to be identified. If the application signaled that it is capable of handling corrupted packets, the packets are not dropped and continue traveling through the network stack. Thus, packet dropping has to be disabled for this type of packets.

Additionally, packet processing heavily depends on the header information. Especially important is information on higher-level protocols, header and packet length, as well as address information (see Figure 3.4). As this approach aims at using as little knowledge about the involved protocols as possible, these fields should not be recomputed or replaced with previously stored data unless it is unavoidable. Thus, the challenge for the implementation of this use case is to use only little stored or recomputed data, but still processing as many corrupted packets as possible.

4

Implementation

This chapter describes how the system presented in the previous chapter has been implemented in the Linux kernel. Following the challenges and constraints mentioned there, modifications to the network stack should be minimized. This leads to the idea of implementing the system as a Linux Kernel Module (LKM), which can be loaded to and unloaded from the Linux kernel during system runtime. An introduction to the implementation of LKMs is, for instance, given by Love [41]. However, not all changes can be realized using a LKM. Components which allow the LKM to interact with the remaining parts of the kernel as well as the use case have to be implemented by modifying the existing kernel code base.

Thus, this chapter is structured as follows. Firstly, the modifications which have to be made to the kernel in order to implement the remainder of the system as a LKM are described. This includes a way to plug the module into the network stack, changes to data structures, and the socket Application Programming Interface (API). Secondly, the actual implementation of the system as a LKM is presented. Lastly, the changes to the network stack which are required in order to utilize the implemented system in a use case are depicted.

4.1 Modifications to the Kernel

Several modifications to the Linux kernel are necessary in order to provide a foundation upon which the actual system can operate. This chapter describes which modifications are necessary and sketches how they were realized.

4.1.1 Hooks

Both the learner and the predictor need access to specific data at certain points in the network stack. In order to minimize changes to the kernel, the packet-classification

tool is implemented as a kernel module. In order to connect the module to the network stack, it is necessary to call functions of the module from the network stack. A common approach to this issue is called *hooking*. Hooking is generally implemented as follows: A so-called *hook* is added to the point at which other applications should be able to be called. These applications can then register to the hook by supplying a callback function. Whenever the execution of the program reaches the hook, all registered callback functions for this hook are called.

A famous example for hooking in the Linux kernel is `netfilter` [49], a set of hooks in the Internet Layer. These hooks are used by various applications, for example, by the `iptables` kernel firewall.

This thesis utilizes a very simple hooking scheme[1], which allows the registration of only one callback function per hook. For each hook, a pointer to the callback function (`null` if none) is stored. In addition, a spinlock is used to synchronize read and write access to these pointers. Whenever a hook is called, it calls the registered callback function if it does not equal `null`. Simple functions for registering and unregistering callback functions exist.

In order to offer the learner and the predictor access to the data in the network stack, two hooks have been implemented. The challenging part was not the actual implementation of the hooks but finding the right positions in the network stack. These positions will be presented in the following.

Learner Hook

As shown in Figure 3.2 (on page 24), the learner sits between the Transport Layer and the Application Layer. Thus, a hook at the interface between these two layers is required in order to allow the learner to inspect all packets. In contrast to the theoretical model (see Section 2.1), such a clear interface does not exist in the Linux kernel. This is due to the different queueing models used by, for example, UDP and TCP. UDP simply adds each received packet to the end of the receive queue. TCP however has to ensure in-order delivery. Thus the queueing model of TCP is much more complicated, which leads to a different implementation. This leads to the idea of implementing multiple hooks, one for each Transport Layer protocol.

Fortunately, a concept called *Linux Socket Filter* exists. This is an implementation of the *Berkeley Packet Filter* [45], a packet filtering method implemented in kernel space. An application can specify patterns and all packets that match these patterns are dropped before they are passed to the application. As the Berkeley Packet Filter runs completely in kernel space, it saves the overhead which is introduced by copying each packet to the user space and back, which is done by other packet filtering approaches. The Linux Socket Filter function `sk_filter()` is called by both UDP and TCP right before the packet is added to the receive queue. Instead of actually using the provided filtering function, the `sk_filter()` function is used as a hooking point. Thus, although no clear interface to the Application Layer exists by the design of the Linux network stack, the Linux Socket Filter helps to overcome this issue by

[1] The hooking scheme is based on work done by Tobias Drüner, Chair of Communication and Distributed Systems, RWTH Aachen University, but has been extended for more convenient usage.

```
struct repair_flags{
 int l2_check_failed;
 int l3_check_failed;
 int l4_check_failed;
};
```

(a) Indication of failed checksums

```
struct pacl_flags {
 struct sock *pred_sk;
 int         diff;
 __be16      pred_protocol:16;
};
```

(b) Prediction data

Listing 4.1 Two structures are added to the `sk_buff` structure.

providing a function which is intended to be called by all Transport Layer protocols after processing of a packet has finished.

The learner hook was added to the end of the `sk_filter()` function. In order to only handle packets which are not dropped due to a filter, the hook is only called for packets which are not to be dropped. Using this hook, references to both the `sk_buff` structure of the packet and the `sock` structure of the socket the packet belongs to, can be passed to the learner.

Predictor Hook

The predictor is located between the Link Layer and the Internet Layer (see Figure 3.2 on page 24). As it operates on the Internet Layer and Transport Layer data, the earliest point at which it can be invoked is after the pointer to the Internet Layer header has been set. The latest point at which it can be called is immediately before the receive handler of the Internet Layer protocol is invoked. Both tasks are performed by the `__netif_receive_skb()` function.

It was decided to invoke the predictor as early as possible. This allows to use the information provided by the predictor in the remainder of the function. Thus, this information could be used, for example, to choose the right Internet Layer receive handler for corrupted packets. Considering this decision, the predictor hook was added to the `__netif_receive_skb()` function immediately after the instruction at which the pointer to the beginning of the Internet Layer header has been set. This way the `sk_buff` structure of the network packet can be passed to the predictor at the right point in time.

4.1.2 Extending the `sk_buff` Structure

As described in Section 2.4.2, the `sk_buff` structure allows access to the data of a network packet. Thus, additional meta data of the packet should be stored there. Extending the `sk_buff` structure can be done as follows: Firstly, a new structure which will hold the additional data is defined. Secondly, this structure (and not a reference to it) is added as an element to the `sk_buff` structure. Thus, no additional changes to functions which create, copy, or destroy `sk_buff`s are necessary.

The approach of this thesis requires two different sets of data to be stored. The first set signals failed lower layer checksums to the upper layers and the second set is used to communicate the prediction result to the upper layers.

Indication of Failed Checksums

Checksums can be used at the Link Layer, the Internet Layer, and the Transport Layer. Thus, the failure of up to three checksums has to be notified to upper layers. Listing 4.1a shows the layout of the `repair_flags` structure. The numbering corresponds to the numbering used in the ISO OSI reference model. Thus, layer 2 references the Link Layer, layer 3 the Internet Layer, and layer 4 the Transport Layer. The instance of the `repair_flags` structure in a `sk_buff` is referenced by `sk_buff.rep`.

Prediction Data

Listing 4.1b shows the `pacl_flags` structure, which holds the prediction data provided to the upper layers. The `pred_sk` field holds a reference to the predicted socket of the packet. Furthermore, the identifier of the Internet Layer protocol is stored in the `pred_protocol` field, as this value, in contrast to the Transport Layer protocol identifier, cannot be retrieved from the `sock` structure referenced by `pred_sk`. It has the same type as the corresponding field in the `sk_buff` structure. In order to signal the quality of prediction, the `diff` field provides the difference of the best and the second best score. The `pacl_flags` structure is embedded into the `sk_buff` using the `sk_buff.pacl` field.

4.1.3 Link Layer Modifications

In order to receive corrupted packets, few modifications to the Link Layer have to be made[2]. This thesis uses wireless network drivers based on the widely used `mac80211` framework [42]. This combination allows to pass corrupted packets to the network stack. Thus, both the driver and `mac80211` have to be instructed to forward frames with a failed FCS to the upper layers. Then the flag in the `sk_buff` structure which indicates a failure of the Link Layer checksum has to be set if the FCS failed. Depending on the wireless network driver used, it might be necessary to check the FCS again, while other drivers set a flag which indicates whether the FCS failed or not.

4.1.4 Socket Interface Modifications

As mentioned earlier, applications have to signal that they are willing to receive corrupted packets. In addition to that, the application should be notified whether a passed network packet was corrupted or not. Thus, a modified socket interface for the Linux kernel that supports both requirements was used[3]. The implementation of both modifications will be explained in the following.

[2]The Link Layer modifications were mainly implemented by Mario Göttgens, Chair of Communication and Distributed Systems, RWTH Aachen University.

[3]The socket interface modifications were implemented by Florian Schmidt, Chair of Communication and Distributed Systems, RWTH Aachen University.

```
struct pacl_aggr_data {
   struct sock         *sk;
   unsigned int        count;
   paclfp              buf[PACL_NUMBER_OF_CHUNKS][PACL_SIZE_OF_CHUNKS
                                                  * PACL_BLOCK_SIZE];
   unsigned int        weight[PACL_NUMBER_OF_CHUNKS];

   __be16              protocol:16;
   struct dst_entry    *skb_dst_entry;

   struct list_head    aggr_data_list;
   short               delete;
};
```

Listing 4.2. The structure `pacl_aggr_data` stores aggregated packet data.

Allowing Passing of Corrupted Packets

A common method for applications using the Linux socket API for signaling options to the kernel space is known as *socket options*. By issuing a call on `setsockopt()` the application can change the value of a socket option. A new binary socket option named `SO_BROKENOK` has been introduced. By setting this option to 1, the application can signal that it is willing to receive corrupted packets.

Signaling a Corrupted Packet

If a corrupted packet is forwarded to an application, the application should be notified that the packet is corrupted. This information can then be passed, for example, to a multimedia codec, which handles corrupted packets differently. The mechanism the Linux kernel socket API uses to signal information alongside a packet from the kernel space to the user space is called *message flags*. An additional message flag named `MSG_HASERRORS` was defined. Setting this flag has to be done by the Transport Layer protocols using the approach proposed by this thesis and thus is described in Section 4.3.

4.2 Kernel Module

As motivated earlier, the main part of the approach suggested by this thesis was implemented as a LKM. This allows for simple turning on and off the new approach of this thesis by simply loading respectively unloading the LKM. The LKM consists of four components: data storage, learner, predictor, and a debugging interface. These components will be presented in the following.

4.2.1 Data Storage

As described in Section 3.3.1, aggregated packet data has to be stored for each socket. This section describes how the data storage is implemented in the LKM.

The core of the storage is the `pacl_aggr_data` structure which holds the aggregated data for one socket. This structure is depicted in Listing 4.2. A reference to the `sock` structure for this socket is stored in the `sk` field. The `count` field is used to count the number of correctly received packets for this socket. A two-dimensional array of fixed-point numbers (see below) named `buf` is used to store a vector V for each chunk (compare Section 3.3.1). Note that `PACL_BLOCK_SIZE` is eight (bits) in typical systems. Furthermore, `weight` is a one-dimensional array that stores the weight factor w for each chunk. The identifier of the Internet Layer protocol is stored in the `protocol` field. Why the `skb_dst_entry` field is needed, will be explained later on, when the implementation of the use case is discussed.

All aggregated data is collected in a linked list (see Section 2.4.4) which is accessed using the `aggr_data_list` item of the storage structure. If the aggregated data for a socket is requested, the list is traversed to linearly search for the storage structure for this socket. If there is no storage structure for the requested socket in the linked list, no correct packet for this socket has been processed before. Thus, an empty storage structure is created and added to the beginning of the linked list. Adding to the beginning instead of the end of the list was chosen because it is assumed that more packets for a newly created socket will arrive in the future than for a socket which was created in the past. Looking up the right storage structure for a given socket could be improved by adding a reference to this structure in the `sock` structure. This, however, would require more changes to the Linux kernel. If a socket is closed, the stored data for this socket cannot be used for prediction anymore and thus has to be deleted. Whenever the predictor accesses the stored data for reading, it is checked if the referenced socket of the stored data is still open. If this is not the case, the storage structure is marked as obsolete by setting the `delete` flag and the actual deletion is scheduled by queueing a job to a work queue (see Section 2.4.5). When the LKM is unloaded, all stored data is deleted immediately.

Fixed-point Numbers

As shown in Section 3.2, floating-point numbers should not be used in the Linux kernel. Thus, the `buf` array which stores rational numbers has to use fixed-point numbers. In contrast to the often used floating-point numbers, fixed-point numbers consist of a fixed amount of digits before and after the decimal point. In this thesis, fixed-point numbers utilize the signed integer implementation of the Linux kernel, which are scaled by a factor of 100 000. Thus, the fixed-point numbers used in this thesis have five digits after the decimal point, with 21 474.83647 being the largest possible number and −21 474.83648 being the smallest possible number. In order to avoid integer overflows, these boundaries had to be kept in mind when implementing the learner and the predictor.

4.2.2 The Learner

The objective of the learner has been described earlier (see Section 3.3.1). It operates on uncorrupted packets before they are passed to the application. Thus, it connects to the learner hook, which has been described earlier.

At the beginning, it was assumed that the learner should operate on all packets that successfully passed the network stack. During implementation, it soon became clear that this is not a good idea. It turned out that packets are duplicated if a packet capturing tool, for example `Wireshark`, [70] is running. In addition, broadcast or multicast packets sent out from a computer are routed back to itself over the loopback device. Thus, packets are not processed by the learner if the socket family is `PF_PACKET`, if the socket type is `SOCK_PACKET` (both used when capturing packets), or if the packet type is `PACKET_LOOPBACK` (used for packets received on the loopback device).

If a packet should be processed by the learner, at first the corresponding storage structure is retrieved (see above). Then the stored aggregated data is updated as described in Section 3.3.1. Hence, the learner iterates bitwise over the part of the packet data, which is used for classification, and updates the corresponding entry in the `buf` array. The number of the chunk and the bit position within the chunk are calculated from the overall bit position. If the packet counter for the socket exceeds the upper bound `MAX_PACKETS`, the value of this bound is used instead of the packet counter for calculating. In the same way, the entry in the `weight` array is updated for every chunk. Lastly, the packet counter is incremented and the identifier of the Internet Layer Protocol copied from the `sk_buff`.

4.2.3 The Predictor

Contrary to the learner, the predictor operates on corrupted packets only. These packets are identified by the Link Layer protocol's FCS (`rep.l2_check_failed`). A description of the conceptual design of the predictor can be found in Section 3.3.2. In order to make its prediction, the predictor has to compare the incoming packet with the stored aggregated data for all currently open sockets. Hence, it iterates over the list of the stored aggregated data. For each storage structure, it iterates over the part of the packet data which is used for classification and thus incrementally calculates firstly the chunk scores and finally the packet score. Iterating over the packet and calculating the number of the chunk and the bit position within the chunk are done exactly as in the learner.

The predictor stores the packet scores of the best and the second best matches seen so far as well as a reference to the storage structure of the best match seen so far. Whenever a better packet score is observed, these values are updated accordingly. Finally, these values can be used to check if the difference of the scores of the best match and the second best match exceeds the threshold `MIN_DELTA`. In this case, the packet is predicted to belong to the socket with the best match. Thus, the values of the `pacl` structure inside the `sk_buff` structure are set accordingly. If a prediction is not possible because the difference of the scores of the two best matches does not exceed the threshold, the predicted socket is set to `NULL`.

4.2.4 Debugging Interface

In order to ease debugging of the LKM it was desirable to have the possibility to have read access to the storage structure. For this purpose, the Linux kernel offers

```
henze@virtualbox:~$ cat /proc/pacl/f412e600
sk:         f412e600
sk->proto:  TCP (6)
sk->sk_family: 2
Count:      622
Buf00 (100): -100000  100000 -100000 -100000 -100000  100000 -100000  100000 -100000 -100000 -100000 ...
Buf01 ( 80): -100000 -100000 -100000 -100000 -100000   52901  -92127   49047   55472  -65764   74122 ...
Buf02 ( 33): -100000 -100000 -100000 -100000  -31424  -50673   25266  -12091    9523   -1019    2420 ...
Buf03 (100): -100000 -100000 -100000 -100000 -100000 -100000 -100000 -100000 -100000 -100000 -100000 ...
Buf04 (100): -100000  100000 -100000 -100000 -100000 -100000 -100000 -100000 -100000 -100000 -100000 ...
Buf05 ( 27):  100000 -100000  100000  -23486   27339  -25317  -14683   -3789    8529   19141   -5161 ...
Buf06 (100): -100000  100000 -100000  100000  100000 -100000  100000  100000  100000 -100000  100000 ...
...
```

Listing 4.3. A procfs file can be used for debugging.

the Proc File System (procfs) [9]. The procfs is a file system which allows user space applications to read and write data exported from processes running in kernel space. This is a convenient way to access data from kernel memory. In typical system setups the procfs is mounted as `/proc` in the root file system.

The implemented LKM creates a new directory in the procfs. For each open socket a file is created in this directory. Whenever the file is read, the content of the storage structure is output in human readable form. Listing 4.3 shows an excerpt from the output of one of the files in the procfs. It firstly lists some information on the socket, then outputs the packet counter, and finally outputs for each chunk the weight (in braces) and the representation of each bit position.

Due to performance and privacy issues, the debugging interface should only be activated when necessary for debugging.

4.3 Use Case

Earlier it has been motivated that a use case should be implemented in order to make actual use of the approach proposed by this thesis (see Section 3.4). As stated, this will be done for the combination of IPv4 and UDP. Furthermore, a small change to the interface between the Link Layer and the Internet Layer is made. This section describes the modifications which were done to the network stack in order to implement the use case[4]. Additionally, all changed code fragments with modifications being highlighted can be found in Appendix A.2 and will be referenced while describing them.

The main constraint when implementing the use case was to introduce as few changes to the network stack as possible. In fact, no changes to the actual packet data should be made. The remainder of this section is structured as follows: The modifications to the network stack are described in chronological order. That is, the description follows the path of incoming network packets (compare Section 2.4.3).

4.3.1 Modifications outside IPv4 and UDP

Although this approach aims at the Internet Layer and Transport Layer, a small modification to the Link Layer is done as well. The Internet Layer protocol handler

[4]Part of the modifications are based on work done by Mario Göttgens and Florian Schmidt, Chair of Communication and Distributed Systems, RWTH Aachen University.

which is called by the Link Layer depends on a field in the Link Layer header. If this field is corrupted, the packet cannot be processed. Thus, the protocol identifier, which was stored in the `sk_buff` structure alongside the prediction data, is used for choosing the Internet Layer protocol if the Link Layer checksum failed (Listing A.1). This step is performed directly after the predictor has finished its work.

The look-up of the right socket which is performed by UDP depends on the IPv4 destination and source address as well as the UDP destination and source port number. If these fields are corrupted, the look-up is likely to fail. Fortunately, the Linux kernel uses protocol independent destination caches. If a destination cache entry is found in the `sk_buff` of a packet, no look-up is performed and the value from the cache is used instead. After a look-up, a destination cache entry is added to the `sk_buff` of the packet. The packet-classification tool makes use of this behavior. Whenever the learner inspects a packet, it stores a reference to the packet's destination cache in the storage structure. After the predictor matched a corrupted packet to a socket, it takes the reference to the destination cache from the storage structure and attaches it to the `sk_buff` structure of the packet. This does not only lead to a higher number of processed corrupted packets but also saves processing time as no socket look-up for corrupted packets has to be performed. In fact, no changes to the network stack are needed to accomplish the described method. All changes can be performed inside the LKM.

4.3.2 Modifications to IPv4

The main IPv4 receive function `ip_rcv()` validates the header checksum, checks if the header length field is at least five (minimum size of an IPv4 header), and if the IP version is indeed four. In the unmodified version of IPv4 a packet that fails one of these tests is immediately dropped. Instead of dropping the packet, the routine is modified such that the flag for indicating a failure of the Internet Layer checksum is set. If the IPv4 header is corrupted, the length field in the header might contain a wrong value. As this field is used to access data in the memory, it is crucial that it does not contain a wrong value. Otherwise, accesses to unallocated memory could occur. To prevent this, for corrupted packets the length is recalculated by subtracting the address of the beginning of the IPv4 header from the address of the end of the packet data (Listing A.2).

As described earlier, the IPv4 header can contain additional fields, so called header options. This is indicated by a header length greater than five. If this is the case the processing of header options is initiated by `ip_rcv_finish()`. Wrongly calling the header options handling function can lead to severe errors. Thus, and because header options are rarely used, corrupted packets are assumed to not contain header options (Listing A.3).

IPv4 supports packet fragmentation and thus has to reassemble packet fragments if fragmentation is indicated by the corresponding header fields. This is done by `ip_local_deliver()`. As fragmentation of IPv4 packets is rarely used today, corrupted packets are assumed to not be fragmented. Thus, fragmentation and reassembly is disabled for corrupted packets (Listing A.4).

The choice of the Transport Layer protocol to which an IPv4 is passed depends on a header field and is taken by `ip_local_deliver_finish()`. For corrupted packets,

instead of the header field the Transport Layer protocol field from the predicted socket is used (Listing A.5).

4.3.3 Modifications to UDP

The main UDP receive function (`__udp4_lib_rcv`) performs some sanity checks on the UDP packet, looks up the socket the packet should be delivered to, and then queues the packet for this socket. Similarly as with IPv4, the length of a corrupted packet is recomputed. Additionally, validating the packet length and trimming packet data is disabled for corrupted packets. If the socket look-up did not succeed, the socket is set to the predicted socket. Beforehand, it is validated that the predicted socket has not been closed in the meantime. In the case that no prediction was possible, a corrupted packet is dropped for safety reasons. Finally, if the application to which the socket belongs did not allow the processing of corrupted packets, the packet is dropped. As UDP and UDP-Lite share the same code, distinctions between the two protocols have to be taken (Listing A.6).

If an application uses one of the socket API's functions to retrieve a UDP packet, the kernel function `udp_recvmsg()` is called which is responsible for checking the UDP checksum and copying the packet to user space. If the requested packet was corrupted, the `MSG_HASERRORS` message flag is set. Furthermore, checksum validation is disabled for this packet (Listing A.7).

5

Evaluation

In the preceding chapters, design and implementation of a novel scheme in handling corrupted packets has been presented. Now the outcome of the proposed approach has to be evaluated.

In order to do so, the right choice for setting the parameters has to be made. This will be discussed in the first part of this chapter. When the right parameter setting has been chosen, the advantages and disadvantages when using the packet-classification tool have to be quantified. Thus, the middle part of this chapter deals with measurements performed in a real environment. Finally, the costs in terms of memory and CPU usage incurred by using the proposed approach are calculated.

5.1 Parameter Settings

As seen in Chapter 3, the packet-classification tool proposed by this thesis has five parameters, which influence the classification results. Table 5.1 gives a brief recall of the parameters.

When evaluating the implementation, the first step thus has to be the identification of the right parameter settings. The first part of this section describes how the evaluation of different parameter settings was performed. In the second part, the data used for evaluation is described. Finally, in the third part the results of the evaluation are discussed and good parameter settings suggested.

`NUMBER_OF_CHUNKS`	Number of chunks the beginning of a packet is cut into
`SIZE_OF_CHUNKS`	Size (in byte) of chunks each packet is cut into
`MAX_PACKETS`	Upper bound for number of packets (aging factor)
`SKIP_FIRST_PACKETS`	Skip prediction for first number packets of each socket
`MIN_DELTA`	Minimal difference of scores between top two sockets

Table 5.1 A summary of the parameters of the packet-classification tool.

5.1.1 Evaluation Setup

The comparison of different combinations of parameter settings requires the possibility to perform simulations that run again and again under the exact same conditions. Thus, real-world experiments are not feasible for this task. This means that at least parts of the different aspects in a wireless communication setting have to be simulated in software. In this thesis, the following way is pursued: An approximately 5.5 minutes long recording of different UDP streams is taken and stored in a trace file. Then a tool is used to introduce bit errors into this trace file. This modified trace file is then injected into the network stack by using a software-simulated wireless device. Thus, it is possible to evaluate different parameter combinations under the exact same conditions.

5.1.1.1 Corrupted Packets Evaluation Environment

The Corrupted Packets Evaluation Environment (CPEE) is an environment which was primarily invented for testing the influence of corrupted packets on the network stack. It was mainly developed by Dennisen [15] and is based on a modified version of `mac80211`'s `hwsim` tool. Basically, `hwsim` simulates IEEE 802.11 devices in software. Thus, it operates at the Link Layer and as such very similar to a `TAP` device [67], which can be used to simulate an Ethernet device in software.

CPEE's main functions are packet capturing, feeding of packets into the network stack and writing response packets into a file. The interesting part for the work of this thesis is the ability to feed packets into the network stack. CPEE is used to conduct repeatable measurements on real wireless network data. Thus, only the functions for feeding packets into the network stack will be covered in the following.

A file which contains captured network packets is read by an user space application which passes the packet data to a kernel space buffer using a character device. As soon as the user space application sends out a certain signal, the packets are delivered to the network stack using the modified version of `hwsim`. This allows to replay a captured network trace arbitrarily often under the exact same conditions.

5.1.1.2 Tracking Network Packets

One of the measures to be examined is the misattribution rate. Thus, the application needs to know, if a packet it received was originally designated for it. This thesis introduces a method for tracking packets which travel through the network stack.

The `sk_buff` structure is extended to hold an additional flag. This flag is then passed to the application in the same way as the `MSG_HASERRORS` flag described in Section 4.1.4. The interesting question is how the flag can be set for evaluation purposes. CPEE requires packets to have a radiotap header [56] in place. In the radiotap header there is a two-byte field holding `RX flags` which has space left for further extension.

A C program was written which can be used to set a flag in the high-order byte of the `RX flags` field. The program parses a network trace in C array byte-oriented

	App. 1	App. 2	App. 3	App. 4
Radiotap flag	0x8000	0x4000	0x2000	0x1000
Message flag	0x80000000	0x40000000	0x20000000	0x10000000
	App. 5	App. 6	App. 7	App. 8
Radiotap flag	0x0800	0x0400	0x0200	0x0100
Message flag	0x08000000	0x04000000	0x02000000	0x01000000

Table 5.2 The newly introduced radiotap flags are mapped to message flags.

description. It searches for UDP packets inside IPv4 packets. According to the UDP destination port, a flag is set in the high-order byte of the `RX flags` field. The output is written to a file with the same format as the input file.

If a flag is present in the high-order byte of the `RX flags` field, it has to be copied to the corresponding `sk_buff` structure for this socket. This structure is created by CPEE's modified version of `hwsim`. Thus, `hwsim` has again been modified. It reads the flag from the `RX flags` field, transforms it to another representation and stores it in the newly introduced `msg_flags` field in the `sk_buff` structure. The transformation is necessary, because all lower bits in the `msg_flags` field, which will be passed to the application later on, are already in use or reserved for usage. Table 5.2 shows how the flags and their mapping are defined.

The UDP implementation of the Linux kernel has been modified to copy the flags from the `sk_buff`'s `msg_flags` field to the `msg_flags` field which is passed to the application. This way the application knows for which application the packet originally was intended. It is then able to calculate the misattribution rate.

5.1.1.3 Bit Error Generation

In order to select the right values for the parameters, different parameter settings have to be tested under various conditions. One of these conditions is the BER. This thesis utilizes the simple Internet Layer bit error generation model from Hammer et al. [20]. They state that their model is sufficient to simulate an additive white Gaussian noise channel.

Let N be the size of the Internet Layer packet in bits and p the Bit Error Rate (BER). The number of bit errors E for a certain packet is then calculated using a binomial distribution:

$$E \sim B(N, p)$$

The actual locations of the E bit errors in the packet are assumed to be distributed uniformly. However, there can only be one bit error at each location. Thus the set of the locations of bit errors $L \subseteq \{0, \ldots, N-1\}$ in the packet can be computed as follows:

$$L = \dot{\bigcup_{i=1,\ldots,E}} \{L_i \sim \lfloor N \cdot U[0,1) \rfloor\}$$

If the positions of the bit errors in a network packet are known, inserting the errors can be done be simply flipping the designated bits X_i using the XOR operator:

$$X_i := X_i^{old} \oplus 1$$

For the purpose of this thesis, a C program was written that inserts bit errors according to the above described model into a supplied trace file. It expects four parameters:

- The *BER* that should be used to generate the errors.
- The *skip* value that indicates how many packets at the beginning of the trace should be kept intact, which is necessary, because the learner only operates on uncorrupted packets. This corresponds to the behavior of real systems, in which a connection is typically established under good channel conditions.
- The input file with a network trace in C array byte-oriented description.
- The output file to which the modified trace should be written. It has the same format as the input file.

Basically, the program iterates over all packets and searches for IPv4 packets which contain a UDP packet. It then uses the beginning and end of the IPv4 packet to calculate the length of the packet. Now the above described error model is applied to this part of the packet. For generating random numbers according to the presented model, the GNU Scientific Library (GSL) [18] is used.

5.1.2 Evaluation Data

As mentioned earlier, an approximately 5.5 minutes long trace of wireless communication has been captured. The network was setup as follows: A normal personal computer with an off-the-shelf wireless card[1] was connected to an unencrypted wireless IEEE 802.11 network offered by an access point. The access point served as a router and allowed access to the Internet using Network Address Translation (NAT).

The trace recorded for evaluating parameter settings consists of four flows from four applications. Not all applications ran the whole time, some started late and other finished early. This models the behavior of a typical system. Characteristics of the different flows will be discussed briefly. The first application flow is a MPEG-1 encoded audio and video stream carrying a television broadcast packed in UDP, which ran over the whole capture time. This is the largest flow (in terms of number of packets) within the trace, outperforming the other flows by a factor of three to ten. The second application flow is an approximately one minute long VoIP call to an echo server. VoIP packets are typically encapsulated in RTP packets which again are embedded into UDP packets. This flow is especially challenging because the first packets contain the SIP handshake, which is used to establish the connection.

[1] The wireless card has to support monitor mode in order to capture packets including the `radiotap` header. In this thesis, wireless cards with the Atheros AR2413 chipset were used.

	App. 1	App. 2	App. 3	App. 4
Type	audio/video (MPEG-1)	SIP/RTP/VoIP	RTP/audio (MP3)	audio (MP3)
No. of packets	25,736	2,154	4,260	7,261
IPv4 source address	137.226.33.60	mainly 217.10.77.41	137.226.59.96	137.226.59.96
IPv4 dest. address	192.168.1.100	192.168.1.100	192.168.1.100	192.168.1.100
UDP source port	42107	mainly 45186	40793	54595
UDP dest. port	1234	7078	5004	4444

Table 5.3 An overview of the applications used in the trace for evaluating parameter selection.

These packets are quite different to those carrying the actual data. As the duration of the call is rather short, this flow is the smallest flow (with respect to the number of packets) in the trace. The next application flow again utilizes RTP. This time an audio stream is encapsulated in it. The last application flow is again an audio stream, this time sent raw over UDP. Table 5.3 gives a summary of the application data in the trace used for evaluating parameter settings.

In order to record the trace, a monitor interface on the wireless card of the computer was created using the `iw` tool, which is typically shipped with Linux distributions:

```
# iw phy phy0 interface add mon0 type monitor
```

Using the `Wireshark` [70] packet capture and analyzation tool to capture packets on this monitor device, it is possible to create a trace file, which includes `radiotap` headers. The right format for using this trace file with the tools mentioned earlier can be generated by using the "C Arrays (packet bytes)" export filter of `Wireshark`.

5.1.3 Performing the Evaluation

The evaluation of parameter settings was performed using the above described tools and trace file. Different combinations of the parameters from Table 5.1 were tested for varying BERs.

For this purpose, a `bash` script was written that iterates over all combinations and performs the commands needed for inducing the trace into the network stack and collecting measurement data. The different steps that have to be performed are described in the appendix (Section A.3).

5.1.4 Evaluation Results

The parameter values and bit error rates for which the evaluations were performed are shown in Table 5.4. These values were chosen because preliminary experiments led to the expectation that these parameter values might be well-suited. The number of combinations that could be evaluated had to be kept very small, because each evaluation run took roughly as long as the real-time of the captured trace.

In order to minimize the effect of statistical outliers each combination of parameter values was run five times using different seeds for the random number generator provided by the GSL, which turned out to be sufficient in terms of standard deviation.

(NUMBER_OF_CHUNKS, SIZE_OF_CHUNKS)	(10, 4), (20, 2), (40, 1)
MAX_PACKETS	1000, 2000
MIN_DELTA	3, 5
SKIP_FIRST_PACKETS	0
Bit Error Rates	0.0001, 0.0005, 0.001, 0.005

Table 5.4 Parameter values and bit error rates used for evaluating parameter selection.

The results presented in the following always show the mean of these five runs. Due to clarity, standard deviations are not shown in the figures. However, they have been considered when interpreting the results.

Two measures were used to interpret the results: the total number of correctly received packets and the misattribution rate. These measures and the results with respect to them will be presented in the following.

Correctly Received Packets

The number of correctly received packets is defined as the number of packets an application received and which where actually designated for it. Thus, this number contains both corrupted and uncorrupted packets but no packets which were falsely assigned to this application. As this number depends on the number of packets which were sent to an application (compare Table 5.3), these results should always be considered with having the number of sent packets in mind.

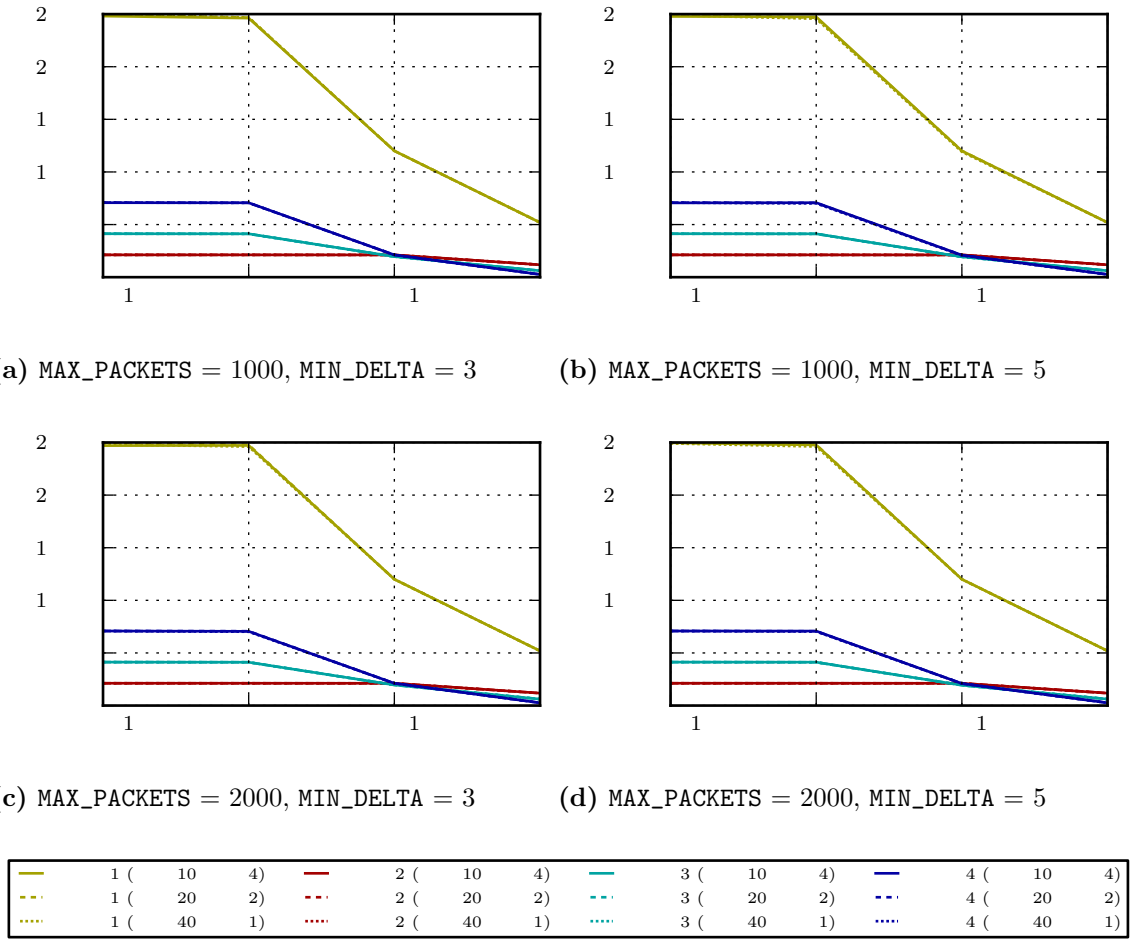
Figure 5.1 shows the results with respect to the number of correctly received packets. Each subfigure shows a combination of MAX_PACKETS and MIN_DELTA. Within a subfigure, the different applications are denoted by different colors, while the combinations of NUMBER_OF_CHUNKS and SIZE_OF_CHUNKS are indicated by different line styles.

The first observation is that the results show the expected behavior. With an increasing BER, the number of correctly received packets decreases. Besides that, the results for one application are (except minimal divergence) the same for all parameter value combinations under investigation. This is a rather surprising outcome, which will be discussed later on. However, the results with respect to the number of correctly received packets do not help to choose the right parameter settings.

Misattribution Rate

The misattribution rate is defined as the number of incorrectly received packets divided by the number of total received packets. Incorrectly received packets are packets which where not originally designated for the application which received them. Thus, the misattribution rate indicates how many percent of the totally received packets where not designated for the application. This means that the predictor made a wrong prediction.

Figure 5.2 shows the results with respect to the misattribution rate. Again, each subfigure shows a combination of MAX_PACKETS and MIN_DELTA. Within a subfigure, the different applications are denoted by different colors, while the combinations of



(a) `MAX_PACKETS` = 1000, `MIN_DELTA` = 3 **(b)** `MAX_PACKETS` = 1000, `MIN_DELTA` = 5

(c) `MAX_PACKETS` = 2000, `MIN_DELTA` = 3 **(d)** `MAX_PACKETS` = 2000, `MIN_DELTA` = 5

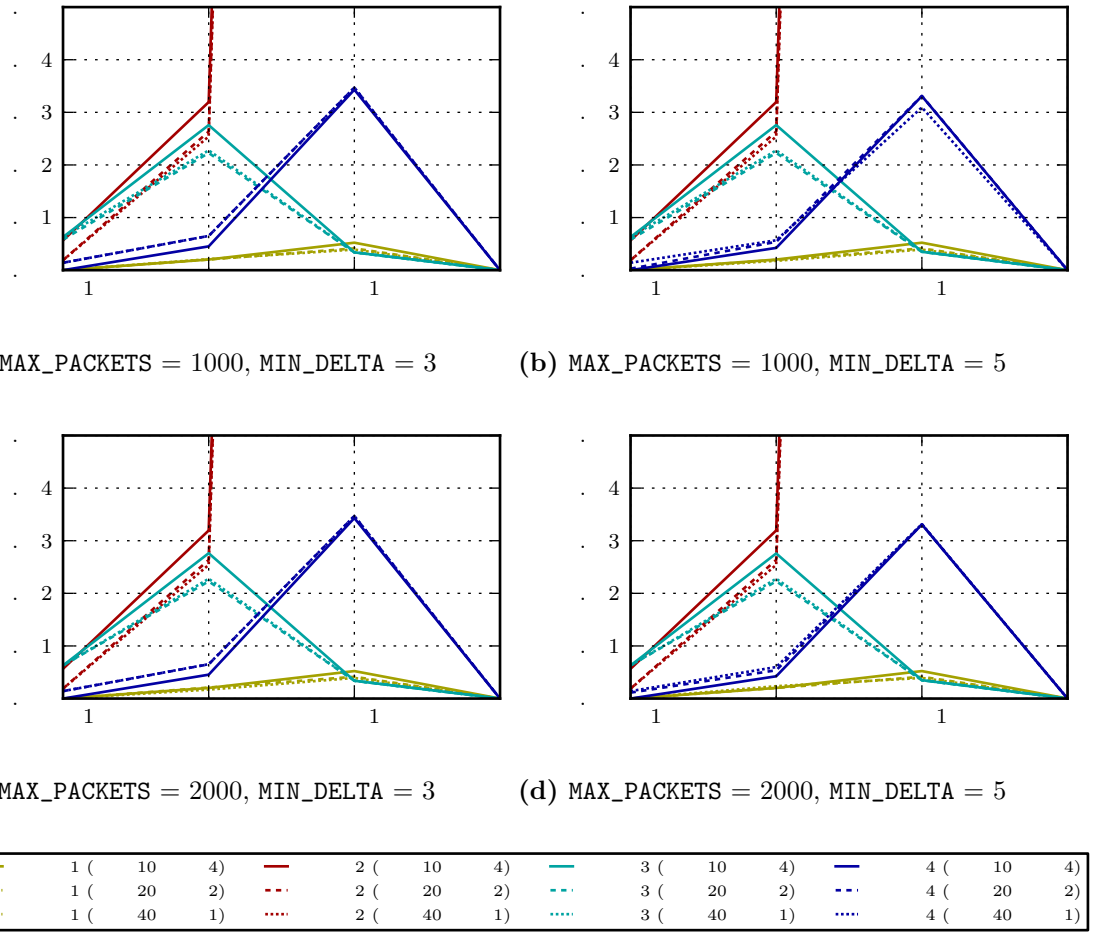
Figure 5.1 Results of measurements of correctly received packets for parameter selection. `NC` indicates `NUMBER_OF_CHUNKS` and `SC` indicates `SIZE_OF_CHUNKS`.

`NUMBER_OF_CHUNKS` and `SIZE_OF_CHUNKS` are indicated by different line styles. Note that parts of the results for the second (red) application lie outside the figure. The misattribution rates for a BER of 0.001 are roughly 0.09 while those for a BER of 0.005 lie around 0.37. These are rather high misattributions rates. Possible reasons for this will be discussed later on.

As these results are presented with a very fine granularity, slight differences between different combinations of parameter settings can be observed. However, the differences are not clear enough to serve as a respectable basis for decision-making. One slight tendency that can be observed is that the results for a `SIZE_OF_CHUNKS` of 1 byte are often outperformed by the results of larger values for `SIZE_OF_CHUNKS`.

5.1.5 Conclusion

The results of the measurements for parameter selection are only of slight help in actually choosing the best parameter settings. It is assumed that this is due to overfitting both the test data and the parameter settings under test. Furthermore, no background traffic was present. This could also be an explanation for the unexpected high misattributions rates.



(a) `MAX_PACKETS` = 1000, `MIN_DELTA` = 3

(b) `MAX_PACKETS` = 1000, `MIN_DELTA` = 5

(c) `MAX_PACKETS` = 2000, `MIN_DELTA` = 3

(d) `MAX_PACKETS` = 2000, `MIN_DELTA` = 5

Figure 5.2 Results of measurements of misattribution rate for parameter selection. `NC` indicates `NUMBER_OF_CHUNKS` and `SC` indicates `SIZE_OF_CHUNKS`.

In order to have a reliable basis for choosing the right parameter settings, the measurements would need to be repeated with more realistic test data, which also include background traffic. Additionally, a larger range of possible parameter settings would need to be investigated. As each test run takes a bit longer than the length of the time frame in which the data was captured and increasing the number of parameter settings soon leads to an explosion of possible parameter combinations, a refined repetition of the measurements was not possible for this thesis.

Thus, the parameter settings for the following experiments had to be chosen heuristically. For choosing suitable values for `NUMBER_OF_CHUNKS` and `SIZE_OF_CHUNKS`, common header combinations were considered. By setting `NUMBER_OF_CHUNKS` to 20 and `SIZE_OF_CHUNKS` to two, the headers of the combination of IPv4 and TCP or IPv4, UDP, and RTP are covered. Choosing a chunk size of two bytes reflects the fact that header fields which are interesting respectively uninteresting for the packet-classification tool can roughly be grouped into two byte-words. Quite early, it turned out that `SKIP_FIRST_PACKETS` is best set to zero. This is due to the fact that it otherwise takes too long to receive a sufficient amount of correct packets under bad channel conditions. As results of preliminary tests suggested, `MAX_PACKETS` was set to 1000 and `MIN_DELTA` to 5. These parameter settings will be used in the following.

5.2 Real Measurements

In order to be able to give resilient results on the performance of the implementation presented earlier in this thesis, measurements in a real environment were conducted. Firstly, the setup of the evaluation will be described. Then the results of the measurements will be presented and discussed. Finally, conclusions from the observed results will be drawn.

5.2.1 Evaluation Setup

All measurements were performed with the same setup over a period of five days. The setup consists of two desktop computers which are located roughly 20 meters from each other in a campus building. The first computer (station A) is used as an access point and to send out data, while the second computer (station B) acts as the receiver which runs the packet-classification tool.

Station A is equipped with an Intel Core2 Duo E6550 CPU which runs at a clock speed of 2.33 GHz with 6 GB of RAM and an Atheros AR2413 based wireless card. In contrast, station B runs with an Intel Pentium 4 CPU running at a clock speed of 3.0 GHz with 1 GB of RAM and an Atheros AR2413 based wireless card.

The access point running on station A was created using `hostapd` [24], which allows to create a wireless access point on a Linux system. It provides an IEEE 802.11g network using channel 5 (2.432 GHz). This channel was chosen, because it partly overlaps with the campus' wireless network and thus increases the chance of interference to occur. This is desirable, because interference leads to bit errors. No encryption was used in the wireless network. For IPv4 addresses the private class A address range 10.0.0.0/8 was used. This allows the use of a wide variety of IPv4 addresses without the hassle of using diverse public addresses.

In order to profit from the benefits of the packet-classification tool, retransmissions of corrupted packets have to be disabled. The decision whether a retransmission is necessary or not has to be made in a very short time frame. Thus, acknowledgment handling is nowadays done in hardware before a packet is passed to the network stack. Waiting for the result of the predictor before acknowledging a packet is hence not feasible. To disable retransmissions of corrupted packets the `QoSNoAck` service class and the voice access category of IEEE 802.11e (see Section 2.2.1) are used. The `mac80211` subsystem on station A has been modified such that all packets belonging to the voice access category have the `QoSNoAck` flag set. Classifying flows as belonging to the voice access category is used for data flows under test. Thus, the receiver does not have to send out acknowledgments for the received packets. However, if the predictor fails to assign a packet to the right socket, the packet is ultimately lost and will not be retransmitted.

In order to measure the performance of the packet-classification tool, a pair of measurement programs[2] was used. One component runs on station A and periodically sends out data while the other component running on station B receives the data and

[2]The programs used for the measurements were mainly developed by Florian Schmidt, Chair of Communication and Distributed Systems, RWTH Aachen University.

	App. 1	App. 2	App. 3	App. 4	UDP-Lite
Packet size	1,000	200	300	800	1,000
Application header	random	RTP	RTP	RTP	random
IPv4 source address	10.0.0.1	10.253.100.196	10.42.24.33	10.93.76.30	10.111.222.33
IPv4 dest. address	10.0.0.100	10.0.0.100	10.0.0.100	10.0.0.100	10.0.0.100
UDP source port	1234	5004	7078	55000	8888
UDP dest. port	1234	5004	31875	45215	8888

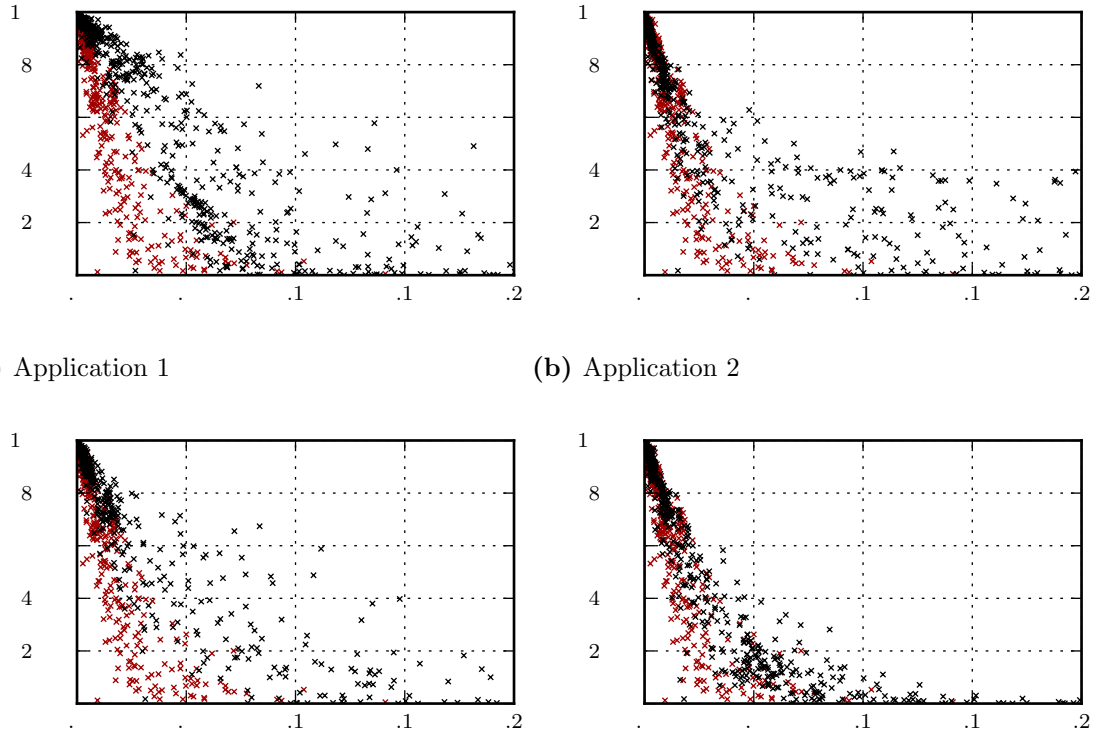
Table 5.5 Overview of the applications used in real measurements.

collects measurement information. In order to figure out if a packet was delivered to the right application, the payload of each packet consists of a byte pattern which is unique for each application. In addition, the patterns for the different applications where chosen such that they have a high hamming distance to each other. If the most common pattern in a packet is not the expected one, the packet is called *suspicious*. Suspicious packets are a sign of misattribution. These patterns will also be used later on to estimate the BER of a packet (see below). In order to prevent influence on the prediction, these patterns must not be visible to the packet-classification tool. This leads to the need of adapting the measurement programs. The area of the payload which is inspected by the packet-classification tool is filled with bits which do not correspond to the pattern. Two options for filling this area were used. The first one forges a RTP header while the second one simply inserts random bits. Of course, the filled in bits are skipped when matching for the unique byte pattern of an application.

Four different combinations of packet size, IPv4 source address, and UDP ports were used in the measurements. In addition, different methods to fill the application header with bits were used. As a reference, one UDP-Lite (see Section 2.2.3) stream was used which did not use the prediction of the packet-classification tool. Table 5.5 gives an overview of the different application settings used in the measurements. In order to create background traffic, a TCP connection was created using the `netcat` [17] tool. Thus, at least six sockets were open during the measurements. All measurements were performed by sending 10 000 packets for each application and a fixed channel bandwidth of 12, 18, 24, 36, 48 and 54 Mbit/s. Thus, different channel performances could be observed. As the measurements were run at day and night, different influences on the channel quality were also covered. For example, the channel quality is much higher during nights or lunchtime when there are less humans and electronic devices around.

Estimating Bit Error Rates

As it is not possible to set a wireless channel in a real environment to a certain BER, the BER has to be estimated from the received packets. For this purpose, the measurement program running on station B utilizes the patterns in the payload of each packet. The number of bits in the payload that match the pattern are counted. Together with the known number of total bits in the payload, it is then possible to calculate the payload bit error rate. If assuming a roughly uniform distribution of bit errors in a packet, the payload bit error rate and be used to estimate the packet bit error rate. The BERs in the following are always calculated over the received packets of a stream of 10 000 sent packets.



(a) Application 1 **(b)** Application 2

(c) Application 3 **(d)** Application 4

Figure 5.3 Number of correctly received packets in real measurements. The results of the UDP-Lite stream are given in red as reference.

5.2.2 Results

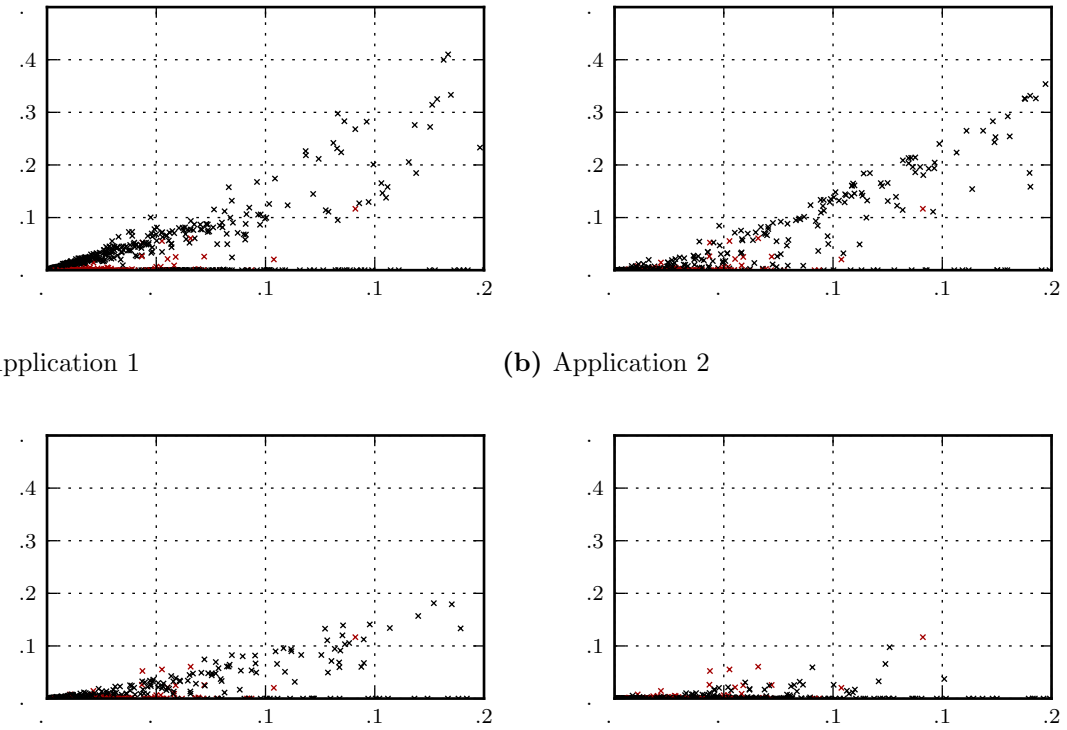
For each channel bandwidth roughly 250 measurements could be performed. This leads to a total number of approximately 1,500 measurements. However, especially for high channel bandwidth a number of measurements exists where not a single packet could be received due to very bad channel characteristics.

This sections presents the results of the performed measurements with respect to two measures: The number of correctly received packets and the rate of suspicious packets. These results are compared to the results of the UDP-Lite stream, which covers only the UDP and the IPv4 header by a checksum. UDP-Lite was chosen because it is a error-tolerant Transport Layer protocol which is already available in the Linux kernel.

In the following, results are always compared regarding the estimated BER. Thus, the results for all channel bandwidths are examined together and only implicitly distinguished by the BER. Results are only shown for BERs up to 20 percent for clarity reasons. In addition, packets which a BER significantly higher than 20 percent are expected to be useless to the application nevertheless. For some of the applications, packets could be received for estimated BERs of up to 50 percent.

Correctly Received Packets

The results of the real measurements with respect to the number of correctly received packets are shown in Figure 5.3. Each subfigure shows the results of one of



(a) Application 1 **(b)** Application 2

(c) Application 3 **(d)** Application 4

Figure 5.4 Rates of suspicious packets in real measurements. The results of the UDP-Lite stream are given in red as reference.

the four applications with black markers. The results of the reference UDP-Lite measurements are displayed in each subfigure with red markers. The comparison of the results of the first application with the reference measurements is especially interesting as both streams share the same packet size.

It can be observed that the results of all four application measurements outperform the results of the UDP-Lite reference measurements. Especially for BERs of five percent and higher the approach proposed by this thesis performs significantly better than UDP-Lite. When comparing the results of the first application with the reference UDP-Lite measure (both share the same packet size) it can be seen that the packet-classification tool outperforms UDP-Lite in nearly all cases. For a BER of five percent the amount of correctly received packets can on average roughly be doubled when using UDP with the packet-classification tool instead of using UDP-Lite.

The results of the stream for the fourth application are worse than the results of the streams for the other three applications. Probably the aggregated data for this application is close to the aggregated data of another open socket. Thus, less packets are forwarded to the application in order to avoid misattributions.

Suspicious Packets Rate

To recall, suspicious packets are packets which are assumed to have been processed to the wrong application. The suspicious packets rate then denotes the amount of suspicious packets with respect to the total number of received packets. For

example, a suspicious packets rate of one percent denotes that one percent of the received packets were considered suspicious. Thus, the suspicious packet rate can be used to measure misattribution.

Figure 5.4 shows the results of the measurements with respect to the suspicious packets rate. Each subfigure shows the results of one of the four applications denoted with black markers. The results of the reference UDP-Lite measurements are displayed in each subfigure with red markers. Unexpectedly, misattributions for UDP-Lite are observed. It is assumed that packets are falsely considered as suspicious due to bit errors in the payload. Again, the comparison of the results of the first application with the reference measurements is especially interesting as both streams share the same packet size.

As expected, an increasing BER leads to a higher suspicious packets rate. It can be observed that this relation behaves roughly linearly. For all four applications a BER of five percents leads to a suspicious packets rate which is around five percent. When considering even lower BERs the suspicious packets rate is very close to zero.

5.2.3 Conclusion

The real measurements showed that a significantly increase in the number of correctly received packets is possible when using the approach presented in this thesis. However, this introduces the risk of misattributions which becomes worse with increasing BERs.

In typical systems the channel bandwidth is adjusted to the channel conditions. Thus, when the channel conditions are degrading the channel bandwidth is decreased in order to cope with the changing channel conditions. For the short term in which high BERs may occur, the packet-classification tool can still provide an improved packet throughput. This of course rises the risk of misattributions. When the channel bandwidth has been adapted, the BERs are lower again. The packet-classification tool can still process more packets than an unmodified network stack. In a setting with an adapted bandwidth, the risk of misattributions is much lower.

5.3 Consumption of Resources

As the usage of the packet-classification tool leads to a computational overhead, the impact on the limited resources of a computer has to be considered. This holds especially for mobile devices. In this section, consequences of the packet-classification tool on the consumption of resources are discussed.

5.3.1 Memory Usage

The approach suggested by this thesis has to store some additional data. Thus, it might be questioned if significantly more memory is required when using this approach. In order to answer this question, an analysis was done.

Additional memory is used at two points. Firstly, the `sk_buff` structure, which allows access to the information related with a network packet has been extended by 22 bytes, which increases the size of the `sk_buff` structure by roughly ten percent. Secondly, aggregated data for each open network socket has to be stored. The size consumed by this data depends on the values of the parameters and will be analyzed later on.

The Linux kernel uses a very efficient way to handle `sk_buff`s. The memory used by a `sk_buff` is freed as soon as a network packet has been passed to the application. Instead of actually freeing the memory, the kernel reuses `sk_buff`s which saves the overhead of freeing an old `sk_buff` and creating a new `sk_buff`. Due to this mechanism, only a few tens of `sk_buff`s exist in the Linux kernel at a certain point of time. Thus, the impact of the small increase in size of the `sk_buff` structure can be neglected.

As stated earlier, the size of the storage structure for the aggregated data (see Listing 4.2) depends on the settings of the parameters. For typical systems with a block size of one byte, the size (in bytes) can be calculated as follows:

$$20 + \texttt{NUMBER\_OF\_CHUNKS} \cdot (\texttt{SIZE\_OF\_CHUNKS} + 1) \cdot 4$$

Thus, the overall used memory (in bytes) if n network sockets are currently open can be computed in the following manner:

$$n \cdot (20 + \texttt{NUMBER\_OF\_CHUNKS} \cdot (\texttt{SIZE\_OF\_CHUNKS} + 1) \cdot 4)$$

In a system with $\texttt{NUMBER\_OF\_CHUNKS} = 20$ and $\texttt{SIZE\_OF\_CHUNKS} = 2$ for each open network socket 260 bytes have to be stored. In order to estimate the number of open network sockets in a typical system, this number has been measured periodically on the computer used for writing this thesis. Over a period of five hours, the number of currently open network sockets was read using the `netstat` tool [50]. As this computer uses quite a lot of network services such as a network file system, a constant base load is present. The result of the measurement is shown in Figure 5.5. As the results show, the number of sockets does not exceed 65 in this setting. Thus, a maximum of 16.5 kilobytes is required for storing the aggregated data in this case. If using one additional megabyte, the aggregated data of 62 sockets can be stored. This should be tolerable for today's systems where RAM is available in gigabytes.

5.3.2 Throughput and CPU Usage

Throughput is a measure for the amount of successfully transferred data over time. It is usually measured in bits per second. The throughput which a system can achieve depends on several factors: the transmission medium used, the coding used at the Physical Layer, buffer sizes, and the time needed for processing a packet in the network stack. This time depends, among other influences, on the implementation of the network stack and the clock speed of the Central Processing Unit (CPU).

As the approach which is suggested by this thesis introduces computational overhead, it has to be evaluated whether this has an impact on the performance of the communication over a wireless link.

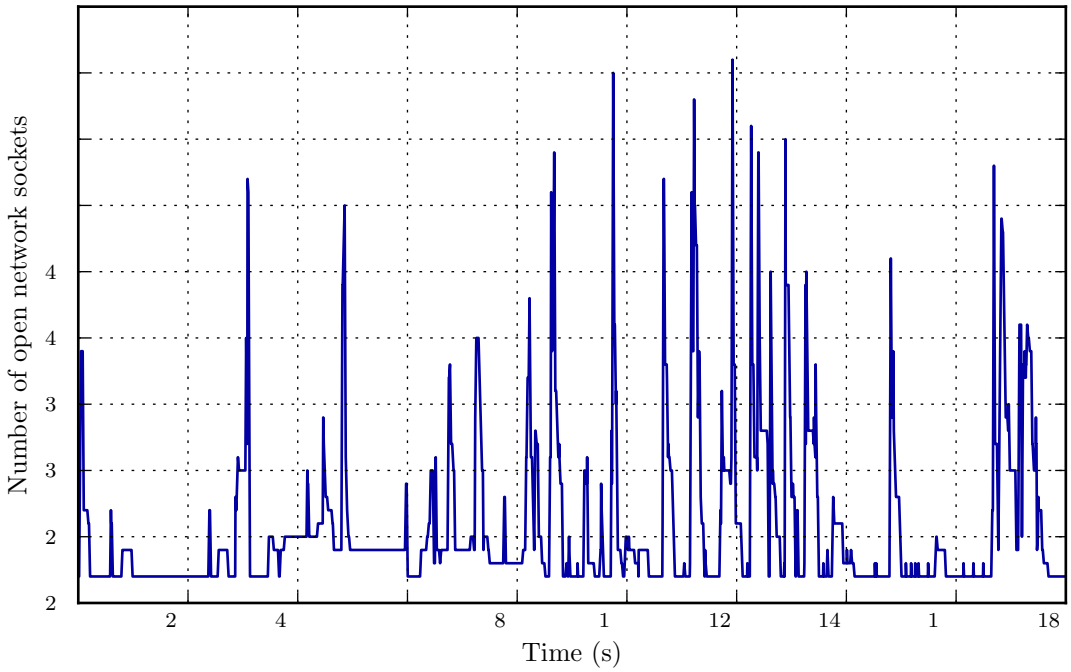


Figure 5.5 Number of open network sockets on a workstation collected over a period of five hours.

Evaluation Setup

In order to measure the bandwidth of a system using UDP, a tool named `Iperf` [28] can be utilized. It consists of two components, a server and a client. The client sends out UDP packets at a fixed bandwidth (1 Gbit/s in this setting) over a certain period of time (here: 10 seconds). The server receives the packets and outputs several statistics, of which one is throughput.

With today's wireless network cards and computers, it is not possible to fully utilize a system's CPU. Thus, measuring throughput of a wireless link does not make much sense. In order to evaluate the impact on the system throughput nonetheless, measurements are performed on the loopback device. The loopback device is a pseudo-device which maps all input to itself. Thus, no restrictions induced by the underlying physical medium have to be considered.

In this setting, no corrupted packets occur. Thus, the learner will inspect every packet. In order to examine the impact of different error rates on the throughput, a very simple Packet Error Rate (PER) model is used. For a given PER, (uncorrupted) packets are selectively passed to the predictor.

As both the learner and the predictor have to iterate over a list of (possibly) all storage structures, it can be assumed that the performance depends on the amount of stored data. Thus, different numbers of data streams were opened during evaluation runs in order to address this question.

Each combination of PER and number of streams was run 50 times on a machine with an Intel Pentium 4 CPU running at a clock speed of 3.0 GHz and 1 GB of RAM. The parameters of the classification were set as follows: `NUMBER_OF_CHUNKS` =

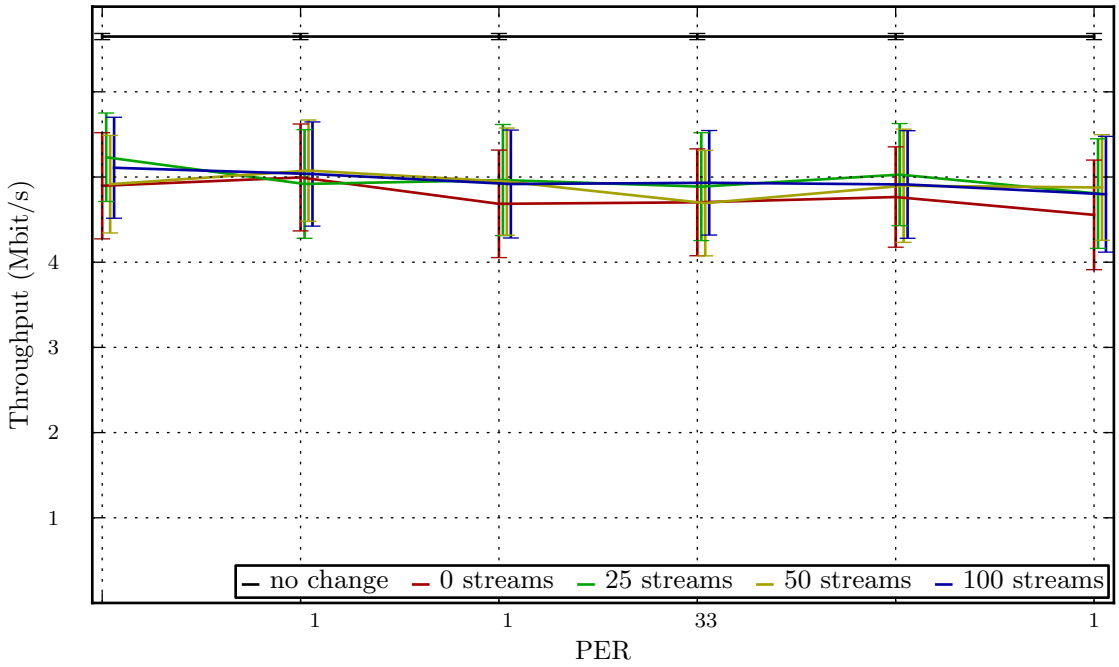


Figure 5.6 Throughput over the loopback device for a varying number of open streams.

20, `SIZE_OF_CHUNKS` = 2, `SKIP_FIRST_PACKETS` = 0, `MAX_PACKETS` = 1000, and `MIN_DELTA` = 5.

Evaluation Results

The results of the throughput evaluation are shown in Figure 5.6. The black line shows a reference measure on the same system with a completely unmodified network stack and without the packet-classification tool being loaded. Four conclusions can be drawn from these results:

1. The throughput seems to decrease by roughly 25 % when the packet-classification tool is used in comparison to the throughput of a system with an unmodified network stack.

2. The throughput does not seem to depend on the number of currently open network streams.

3. The deviation of the throughput is significantly higher when the packet-classification approach is used.

4. Higher PERs and thus more invocations of the predictor have only a very slight impact on the throughput.

While the first result was at least by trend expected, the other three are rather surprising. It is believed that the reason for these other three results is the same. This phenomenon might be explainable by memory caching. As seen above, the approach

suggested by this thesis requires some additional storage space. Because memory access is quite expensive (in terms of time), small differences in the computation time might not make a big difference to the achievable throughput. This would explain why the throughput seems to not depend on the number of currently open network streams and why the performance only decreases very slightly for higher error rates. The high deviation rates could then be explained by the occurrence of cache misses. If the accessed storage data is still cached, accessing the data is much faster as if it has to be fetched from main memory first. Further analysis of this phenomenon has to be performed as future work.

In conclusion, a decrease of throughput has been observed when fully utilizing the CPU. As bandwidths in today's wireless networks are far away from leading to a full utilization of the CPU, this decrease does not impact the throughput which can be achieved in real systems. However, it can be assumed that using the approach presented by this thesis leads to higher CPU utilization. This might be an issue for mobile systems with only a limited energy supply. These might however profit by a reduced amount of retransmissions. Investigating this issue is an interesting field for further studies.

5.4 Conclusion

In this chapter, different approaches to the evaluation of the approach presented in this thesis have been discussed. At first, it has been described how measurements for selecting optimal values for the parameters of the packet-classification tool can be performed. Then, measurements in a real wireless network have been performed. This allowed for comparing the performance of the packet-classification tool with the performance of UDP-Lite. It turned out, that using the packet-classification tool instead of UDP-Lite can significantly increase the number of received packets. As a trade-off, misattributions become an issue when using the approach proposed in this thesis. Finally, the additional system resources required by the packet-classification have been analyzed. Only a moderate amount of additional RAM is required, but the throughput in a system with unlimited bandwidth decreases by 25 %.

6

Related Work

In this chapter, work which is related to the topic of this thesis will be presented. As the approach taken by this thesis as well as the whole field of processing corrupted packets are developing fields of research, related work is presented in a broader form. Thus, especially work which gives a broader overview over certain topics and compares several approaches will be considered.

Firstly, work in the area of coping with erroneous network packets will be presented. This covers the area of processing corrupted packets, which is very close to the topic of this thesis, as well as approaches which aim at recovering or preventing errors. Secondly, contributions to the field of mining data streams, with a focus on the classification of data streams, will be discussed. Lastly, the cognate area of traffic classification will be enlightened. A special aspect will be machine learning approaches to traffic classification.

6.1 Erroneous Network Packets

As motivated earlier in this thesis, erroneous network packets occur very frequently, for example, in wireless settings. Reducing the number of unusable network packets can lead to a better overall performance of wireless communication. In this section, approaches which address this issue from different points of view will be presented. However, there are no strict borders, which allow a clear distinction between the presented fields of research. Some approaches combine different aspects in order to gain even more performance.

6.1.1 Processing Corrupted Packets

Forwarding corrupted packets to the application is not a new idea. The earlier on presented UDP-Lite (see Section 2.2.3) follows this goal. Packets with errors in

the Transport Layer payload can be passed up to the application by setting the checksum coverage accordingly. But UDP-Lite has one major drawback: it is not fully backward compatible with UDP. If UDP-Lite would replace UDP, the sender had to know in advance if the receiver is capable of processing UDP-Lite packets. This could be accomplished by out-of-band signaling but is not very handy. Instead of replacing UDP, UDP-Lite can be used side-by-side with UDP. Actually, this is how it is implemented right now in the Linux kernel. The major downside of this approach is that UDP-Lite has its own protocol identifier (136) and thus packets sent from legacy UDP hosts will still be forwarded to UDP. In order to overcome this issue, Lam and Liew propose UDP-Liter [34]. UDP-Liter gives applications on an end host the possibility to turn off UDP checksum validation for receiving a packet. The socket API is modified such that the application receives a signal if the UDP checksum was correct or not. One big disadvantage of UDP-Liter is that checksum validation can only be disabled for (pseudo-)header and payload together. Thus, there is a certain risk of processing a packet to the wrong application. This problem increases if the Internet Layer protocol does not protect its header and thus none of the UDP pseudo-header fields is protected by a checksum. One example of such an Internet Layer protocol is the evolving IPv6. From the application's point of view, UDP-Liter behaves very similarly to the solution proposed by this thesis as the application can request to receive corrupted packets. However, the underlying technique is completely different. The approach presented in this thesis is superior to UDP-Liter as it is also able to process packets with errors in the header fields.

Another option in handling corrupted packets is to repair corrupted packets. Usually detailed knowledge of the protocol is required which leads to solutions that are extremely customized to the specific protocol.

Göttgens [19] is currently working on heuristic IPv4 and UDP repair techniques for processing corrupted packets on end hosts. The application can signal its capability to handle corrupted packets to the network stack by setting a socket option. A message flag then informs if a bit error occurred in the passed packet. The fields of the IPv4 and UDP header are grouped into two classes: *don't-care* and *vital*. The header fields of the don't-care class are stated to not be important for processing packets on end hosts. Nonetheless *static* and *variable* don't-care fields are distinguished. While variable fields such as the time to live field cannot be repaired, the repair of static fields is attempted. The version field, for example, is always repaired to 4. The repair of static don't-care fields is done in the hope of being able to completely repair the header and thus potentially end up with an error-free packet. Vital fields in contrast have to be repaired in order to correctly process the packet. Thus, known destination and source addresses as well as ports are stored in lists. This stored data is compared to the corresponding header fields of the received corrupted packets. A header field is then repaired to the match with the closest hamming distance. In order to prevent misattributions, an upper bound for the hamming distance used for matching is introduced. One major drawback of this repair approach is that it requires the protocol field in the IPv4 header and in the MAC header to be intact, as different protocol identifiers might differ by only one bit. Thus, these fields are highly prone to bit errors. In contrast, the approach proposed by this thesis does not rely on the protocol fields and can process a packet with corrupted protocol fields if a sufficient amount of other header fields is intact.

Alfredsson and Brunstrom propose TCP-L, a bit error tolerant modification of TCP [2, 3]. TCP especially suffers from errors in packets because it cannot tell apart if a packet got lost because of transmission errors or congestion. Thus, the congestion algorithm will decrease the sending rate and thus throughput performance drops. They show that a bit error tolerant version of TCP could perform far better in erroneous environments. It is proposed to use an approach similar to those above. A study showed that there are header fields which are important for processing a packet and some which are not crucial for processing a packet. Thus, only the former have to be repaired. All but one header field which have to be repaired in a flow stay static over time and can thus be more or less easily repaired using techniques similar to those in the UDP case. The big challenge is the handling of the sequence header field which increases over time and is crucial for in-order delivery of packets and sending out acknowledgment packets. If and how this field can be repaired is a challenging open question for research. The work contributed by the authors shows that approaches like the one proposed by this thesis that try to process corrupted packets can lead to increased performance when applied to TCP.

Another approach that utilizes processing corrupted packets is TCP HACK [6]. However, corrupted packets are not passed to the application but used in TCP to distinguish between packet corruption and link congestion. In principle, two new TCP header options are introduced. The first header option holds an additional checksum. This checksum covers only the TCP header and IP pseudo-header. If a corrupted packet is received by TCP HACK it calculates the header checksum of this packet and compares it to the one stored in the header option. Should the header be corrupted, the packet is discarded. Otherwise, the error has to be located in the payload. Thus, an acknowledgment packet containing the second new header option is sent back to the sender. The header option contains the sequence number of the corrupted packet. Thus, the sender knows that the packet got lost due to packet corruption and will resend the packet without triggering the congestion control algorithm. In contrast to the concept of this thesis, TCP HACK requires changes at both the receiver and the sender.

6.1.2 Partial Packet Recovery

In today's wireless networks, even a single bit error leads to the retransmission of the complete network packet. If only the corrupted parts of the packet would be retransmitted, performance gains could be earned. The major challenge is the question how to figure out the bit error positions in the packet. In this section, two different approaches to partial packet recovery and bit error localization are discussed.

The first approach is simply called Partial Packet Recovery (PPR) [30]. It uses a different Physical Layer implementation than IEEE 802.11 or 802.15.4. Instead of transforming the coded wireless signal into a binary stream, it offers the upper layers information on how confident it is about the correctness of each bit. In addition, a postamble is added to the transmitted signal which allows delayed synchronization in case of an erroneous preamble. Both methods lead to an increase of receivable packets. The upper layers are now able to figure out in which parts of a packet errors might have occurred. Thus, an Automatic Repeat reQuest (ARQ) protocol is

suggested which allows to request a retransmission of only those parts of a packet that are likely to be corrupted. The major drawback is the usage of additional information on the physical layer. Today's wireless cards do not offer this possibility and it is questionable if such hardware will be widely available in the near future.

In order to make use of the promising idea of partial packet recovery in today's systems, Maranello was developed [21]. It is mainly implemented in an open source firmware [52] for Broadcom wireless cards. The underlying idea in Maranello is to calculate block checksums for corrupted packets and sent those back to the receiver. If the receiver receives such a packet, it computes the block checksum of the sent packet (which is kept in a buffer), compares it to the ones received, and only retransmits the blocks whose block checksum failed. This implies a major challenge, as checksum calculation is very costly on embedded devices like a wireless card and the existing CRC chip of the wireless card could not be utilized. Thus, an easier to compute checksum and pre-calculation of checksums is used. The big advantages of Maranello are that it is fully compatible with IEEE 802.11 networks and that it does not require the transfer of additional bits if a packet is received correctly.

Both approaches presented in this section require changes to the receiver and the sender. Instead of passing corrupted packets to the application as it is proposed by this thesis, retransmissions of only the corrupted parts of a packet are scheduled. Nevertheless, partial packet recovery schemes could be used in combination with the approach proposed by this thesis, for example, by only requesting a partial retransmission if no prediction is possible.

6.1.3 Header Compression

The field of header compression contains approaches which aim at reducing packet size by compressing header fields. The main reason for inventing header compression was a better utilization of the bandwidth offered by low speed links (telephone lines, first wireless networks). It turned out that, due to the reduced packet size, header compression can be used to reduce the packet loss rate, for example, in wireless environments. As the packet size decreases, the risk of bit errors within a certain packet decreases as well. In the remainder of this section, three approaches to header compression will be presented. All three were published as proposed standard RFCs.

The first solution was proposed by Jacobson [29]. It targets the compression of IPv4 and TCP headers in order to achieve higher throughput over low speed serial links. The basic idea is based upon the fact that half of the bits in the joined IPv4 and TCP header do not change during a TCP connection. Thus, the concept of *compressed* and *uncompressed* packets is introduced. Uncompressed packets are unmodified packets which are sent only when synchronization is necessary, which is at the begin of a flow or after errors occurred. Compressed packets only carry as header fields a change mask, the connection number which identifies the TCP flow, the TCP checksum, and changes in header fields since the last packet (identified by the change mask). The compressed packet is reassembled at the receiver's side by looking up previously stored header field values using the connection number. Error detection is done by a Cyclic Redundancy Check (CRC) at the Link Layer and the TCP checksum, which is sent with every packet. Using Jacobson's header

compression scheme the size of compressed IPv4 and TCP header can be reduced to ten percent (from 40 byte to 4 byte).

Based on Jacobson's work, the IP Header Compression approach has been suggested [14, 13]. The contribution basically generalizes Jacobson's idea. A new concept, *subheaders*, is introduced. Subheaders are IPv6 base headers, IPv6 extension headers, IPv4 headers, UDP headers, and TCP headers. A subheader is compressed using a similar scheme to the one seen above. Only changes between different packets of a flow are transmitted. In addition, uncompressed packets are sent out periodically to increase the quality of context information. This however decreases the efficiency of the proposed approach. Subheaders can be compressed independently of other subheaders. Thus, an Internet Layer protocol header could be compressed without considering the Transport Layer protocol, and vice versa. However, for some combinations of headers, for example IPv4 and TCP, a combined compression can be used in order to gain a stronger compression. This approach yields a reduction to ten percent as well. An IPv4 header can be reduced to two byte (from 20 byte) and the headers of the combination of IPv4 and TCP to 4 byte (from 40 byte). Due to the periodically sent out uncompressed packets, this approach is not as efficient as Jacobson's for the combination of IPv4 and UDP. Yet it offers much more flexibility.

RObust Header Compression (ROHC) [8] follows a different and more complex way of compression. Different compressing states and operation modes are used in order to improve the compressing factor over time. Each flow starts in the initialization and refresh (IR) state, where uncompressed headers are sent. As soon as both sides of the connection have stored the static header fields, the so-called first-order (FO) state is reached. Now differences of non-static header fields are sent. In the final second-order (SO) state, even non-static header fields are not sent any more. Only a sequence number and a small checksum are sent. Thus, the receiver has to predict the expected header and can only then verify its prediction using the received packet. ROHC can be operated in three different modes. In the unidirectional mode (U-Mode) packets are sent from the compressor to the decompressor only. As no feedback is possible, a periodical refresh is performed by returning to the IR state after a certain amount of time. The bidirectional optimistic mode (O-Mode) introduces a feedback channel from the decompressor to the compressor. This allows for requesting error recovery and optional acknowledgment packets to be sent. Bidirectional reliable mode (R-Mode) utilizes an intensive use of a feedback channel and stricter logics to prevent loss of context and thus performance decrease. With ROHC it is possible to reduce an IPv4, UDP, and RTP header to 2.5 percent of its original size (from 40 byte to 1 byte).

Tye and Fairhurst compared the savings which can be achieved by using the above described approaches [65]. They compared the possible decrease in header length of the different approaches for different combinations of IPv4, IPv6, TCP, UDP, and RTP. ROHC outperforms the other two approaches in all settings, being always able to compress the header to less than ten bytes. A big disadvantage of all three presented approaches is that a modified network stack has to be deployed at all nodes on the path a compressed packet shall travel along. Another disadvantage is error propagation. If the information stored at the receiver contains errors, these errors will be present in all uncompressed headers until the next synchronization.

Using header compression for dealing with erroneous channel conditions follows a completely different approach to the one this thesis proposes. Header compression is used to reduce the size of the header and thus the possibility that errors occur in the header. The approach of this thesis however aims at processing packets although they are erroneous. As header compression significantly reduces the size of the header and reduces similarities of the headers of different packets of the same network stream, it is not suited to be used together with the approach presented in this thesis.

6.1.4 Forward Error Correction

Another option for handling erroneous network packets is to accept that errors will occur and thus make preparations for correcting those errors. The area of Forward Error Correction (FEC) consists of approaches which use coding schemes to correct bit errors. In the following, three different approaches to forward error correction will be presented.

TCP with adaptive forward error correction (TCP-AFEC) [40] dynamically selects the FEC scheme that fits best based on wireless channel characteristics. TCP is especially prone to bit errors, because it will assume that packet loss occurred due to congestion and thus reduces the utilized bandwidth. Minimizing packet loss due to bit errors using FEC can lead to an increased throughput in error-prone wireless environments. However, FEC requires adding additional bits to each packet in order to create redundancy. This of course reduces the throughput of the system. TCP-AFEC senses the wireless channel and predicts based on this observations the quality of the wireless channel. Then a FEC coding with adjustable levels of redundancy is used. If the channel quality is good, a low level of redundancy is used, and vice versa. In order to implement this approach, modification to the Link Layer at both the wireless base station and the end host have to be made. However, no modifications to IPv4 or TCP are necessary. TCP-AFEC could benefit from the additional usage of the approach presented in this thesis. If the packet-classification tool would be applied to TCP, it could be used as a fallback in the case of a rapid aggravation of channel conditions. During the time which TCP-AFEC needs for adjusting to the new channel conditions, corrupted packets could still be processed.

Another approach to FEC is called Multi-protocol Header Protection (MPHP) [5]. MPHP aims at improving multimedia streams which use UDP-Lite by especially protecting the headers of the involved protocols. It is implemented at the Link Layer and utilizes a coding with a very strong code rate, for example 1/3 (which means that two code bits are inserted for each bit that should be protected). Different profiles can be defined which specify how many bits should be protected and which code rate should be used. The Link Layer protocol header is always protected when using MPHP. If the Application Layer protocol uses headers with fixed size, these headers can be protected as well. Thus, it is possible to protect an RTP header or even a MPEG-4 header inside an RTP packet. MPHP can be used to further improve header compression schemes (see above).

The last approach to be presented is not a pure FEC approach as it combines FEC and packet recovery schemes. ZipTx [38] takes measures to estimate the BER

and sends out parity bits only if the receiver did not acknowledge a packet. The estimation of the BER is done by adding so-called pilot bits to the network packet. If a corrupted packet has an estimated BER over a certain threshold, it is assumed that error correction for this packet is computational too expensive or even impossible. The packet is thus dropped and the sender informed. Otherwise, the receiver informs the sender that it received an erroneous packet and thus requests a first set of parity bits. If these parity bits suffice to repair all bit errors, the packet can be processed in the network stack. Elsewise, a second set of parity bits can be requested. Should the union of both parity sets still not suffice for error correction, the packet ultimately has to be dropped and retransmitted. One major disadvantage of the proposed solution is that additional packets from both the sender to the receiver and vice versa have to be sent. To overcome this issue, ZipTx utilizes the fact that IPv4 packets typically have a packet size of not more than 1,500 bytes while IEEE 802.11 frames can have a length of up to 4,095 bytes. This leaves enough space to piggyback the parity bits onto packets that are sent out anyway. The receiver feedback on the other hand is aggregated to batches and thus sent out asynchronously. ZipTx could also be combined with the approach proposed by this thesis. Parity bits would only be requested if the packet-classification tool would fail to predict the destination socket of a corrupted packet.

6.2 Mining Data Streams

The concept of data streams has been introduced earlier in this thesis (see Section 2.6.2). Extracting knowledge from data streams is called *mining data streams* and is a broad area of research. An extensive introduction into this area is given by Gaber, Zaslavsky, and Krishnaswamy [16]. They distinguish between four mining techniques, which are related to data mining techniques in general. *Clustering* and *classification* are of interest in mining data streams, too. In addition, the techniques *frequency counting* and *time series analysis* are used in mining data streams. Frequency counting is used to count frequent item sets while time series analysis deals with the analysis of (typically equidistant) timed data. As the classification domain is closest to the work in this thesis, selected approaches to classification of data streams will be presented in the following.

Classification of Data Streams

Classification of data streams has been described in Section 2.6.2. There, several issues which arise while classifying data streams were revealed. In this section, approaches which address these issues are illustrated.

Last [36] proposes the *On-Line Information Network (OLIN)* approach which operates on non-stationary data streams. Non-stationary in this context means that concept drifts occur. Thus, an algorithm has to adapt to the changed data. The concept is based on a so-called info-fuzzy network. This classification concept looks similar to neural networks. However, classifying an item works similar to decision trees by testing one attribute at each layer of the network. OLIN uses a sliding window over the data stream for the training set which is used to build the info-fuzzy

network. The algorithm constantly compares the error rate of the training set and a validation set. If the error rates significantly differ, the presence of a concept drift is assumed. Thus, the size of the sliding window and the frequency of network creation are adapted, which allows the classifier to readjust faster to the new conditions. If the error rates stop differing, the classifier is assumed to have successfully adapted to the new conditions. Thus, the size of the sliding window and the frequency of network creation can be set back to more conservative settings.

Another approach, the *On Demand Stream Classification Process*, is suggested by Aggarwal, Han, Wang, and Yu [1]. Their training model is designed to adapt quickly to changes of the data stream. This model is based on so-called supervised micro-clusters which are constructed from the training data. Each micro-cluster stores a set of data points from one class. The micro-clusters are constructed such that updating them is computationally easy. As the training data is taken from a sliding window, the adaptability to concept drifts strongly depends on the size of the sliding window. In order to be able to use the sliding window size which allows the highest classification accuracy, the states of the micro-clusters are stored at different points in time. In order to store an amount of micro-clusters which is enough to approximate all possible sliding window sizes and still does not waste resources, a model is suggested which stores micro-clusters logarithmically over time. A small portion (for example 1 percent) of the training data is not used for constructing micro-clusters but for continuously estimating the best sliding window size. Thus, the On Demand Stream Classification Process can readapt very quickly to concept drifts.

In contrast to the presented data stream classification approaches, the packet-classification tool must not be sensitive to (a high number of) errors in the data which has to be classified. Additionally, while learning could be deferred in order to perform more complex operations, classification has to be performed while processing the incoming packet. Thus, high computation times would lead to a delay of packet delivery.

6.3 Traffic Classification

Traffic classification is used to classify a certain traffic flow to an application or application class. There is a broad range of possible appliances that benefit from traffic classification. To name only a few, there are traffic shaping, QoS, intrusion detection, and enforcement of network policies.

There are several different approaches to traffic classification. Kim et al. [32] give an overview over the different types of approaches and compare them. The first approach is the *port-based* approach. Transport Layer protocols typically use the concept of ports in order to multiplex between different applications running on the same host. On the Internet, applications typically use well-known ports for communication. Thus it is possible to look into each packet, read the port numbers, and thus classify the packet to an application class. But in recent years applications try to hide from such approaches by using random port numbers or abusing other applications' well-known ports. The second approach is called *payload-based* approach. Instead of looking at the packet headers, the payload of the network packets is examined. Payload signatures describing typical patterns in the payload have

to be created for every application that shall be classified. The *host-behavior-based* approach can even work on encrypted payload data. For each host communication patterns such as destination addresses and ports it communicates with are compared to typical host-behavior signatures, which allows for application classification. The last approach is named *flow features-based* approach. This approach is closest to the topic of the thesis. Machine learning and data mining techniques are used on flow features. Flow features are a wide range of observations, for example addresses, ports, size of packets, start and end of a flow, number of packets, and throughput. Selected approaches to traffic classification will be presented in the following section.

Traffic Classification Using Machine Learning

There exists a wide variety of machine learning approaches which can be used for classifying traffic [68, 32, references therein]. The most common ones will be presented briefly in this section. The *Naïve Bayes* approach uses Bayes' theorem. It classifies each object to the class it most probably belongs to. For this purpose it is assumed that each class is described by a Gaussian distribution, whose parameters are estimated from the training data. An extension to the Naïve Bayes approach is known as *Naïve Bayes Kernel (Density) Estimation*, where multiple Gaussian distributions are used. The *Bayesian Network* approach is based on a directed acyclic graph. Features and classes are represented by nodes in this graph. Labeled edges between nodes represent the probability of a relation between them. Another method is the *k-Nearest Neighbor* approach. In order to use this approach, a distance metric between data points has to exist. Each data point is labeled with the class which occurs most often within its k nearest neighbors. A refinement of the k-Nearest Neighbor approach weighs the count of class labels by the distance to the data point which should be classified. The *C4.5 Decision Tree* approach consists of a tree where processing starts at the root. Each node represents a test on one attribute of the data set. A data set is labeled with the class with which the reached leaf of the tree is labeled. *(Artificial) Neural Networks* are inspired by neurons, for example, of the human brain. A Neural Network is the weighted interconnection of many neurons. The output of a neuron is determined by the weighted input from other neurons. A Neural Network learns, for example, by creating or deleting neurons and connections, or changing weights. The last approach is called *Support Vector Machines*. Figuratively this approach calculates maximal separating hyperplanes between data points of different classes in the multidimensional feature space.

Comparisons [68, 32] of the different approaches show that each approach has advantages and drawbacks. Nevertheless the Support Vector Machines approach significantly outperforms all other approaches in terms of accuracy. Additionally it has been observed that the number of features and thus the size of the feature space can be considerably reduced which leads to decreased space and computation time requirements.

When referring to traffic classification often the classification of application classes on machines observing traffic as person-in-the-middle is meant. This thesis in contrast follows a different approach. The classification takes place on end hosts and distinguishes even between different communication instances of the same application, that is, two video streams run using the same application belong to two

different classes. This is possible, because on an end host much more information on a network packet is available.

7

Future Work

This chapter presents possible future work on the approach presented in this thesis. Additional work on the presented ideas is considered worthwhile, but could not be covered by this thesis. These ideas were partly considered when initially designing the approach presented in this thesis, but some arose up later on while implementing and evaluating the packet-classification tool.

At first further methods to evaluate the performance of the system will be proposed. Then possible improvements to the actual packet-classification tool will be discussed. After that, the potentials of porting the system to other platforms will be presented. This chapter closes with ideas on advanced use cases which utilize the packet-classification tool.

7.1 Further Evaluation

As wireless channels suffer from a wide range of varying conditions, not all possible settings could be covered in this thesis. Especially settings with large numbers of wireless stations have been left out. Additionally, the impact of the improvement which can be achieved by using the solution presented in this thesis on the actual performance for applications such as audio and video streaming has not been considered. Thus, it seems that further evaluations on the behavior of the solution proposed by this thesis are necessary in order to further support the results and maybe fine-tune the parameters which are used. This section presents ideas on how these further tests could be performed.

Mobility

Until now, no measurements considered mobility. This means that both the sender and receiver did not change their position. Especially for mobile devices such as smartphones, netbooks, and laptops, mobility in wireless networks is an important

issue. These devices do not only change their position within a wireless network, but also roam between different networks. Research on the behavior of the packet-classification tool in mobile settings could show if it is able to rapidly adapt to completely different conditions.

Wireless Ad-hoc Networks

Until now, only tests in managed wireless networks have been performed. This means that all stations have to connect to an access point, which, for example, is responsible for routing. Contrary, in wireless ad-hoc network, no such central entity exists. In order to send a packet to a node which is not directly reachable, packets have to be relayed by intermediate nodes. Due to the nature of ad-hoc networks, the topology and thus the routing behavior can change frequently. Investigating the performance of the packet-classification tool in such environments could lead to interesting results.

Network Emulation

When developing software for wireless networks testing is a difficult issue. There exists a large gap between network simulation on the one side and real experiments on the other side. Network simulations are easy to set up and offer repeatability, but abstract too much from real systems. Contrary, real experiments are hard to set up and do not offer repeatability, but offer very realistic results. Thus, it would be desirable to bridge this gap by bringing both methods together. This is done by a concept called *network emulation*. Briefly said, network emulation allows using real systems within a simulated environment. Vom Lehn [37] developed a virtual wireless device for the Linux kernel which can be used to connect a system to a simulation run by the open-source `ns-3` network simulator [64]. Using `ns-3`, it is possible to simulate an IEEE 802.11 wireless network. As `ns-3` is shipped with error models for wireless channels, it is possible to simulate erroneous environments. With the virtual wireless device developed by vom Lehn it is possible to couple one or more real systems with the packet-classification tool running into a simulated wireless network with a possibly large number of other participants. Thus, the performance of the packet-classification tool could be tested in such an environment under controllable conditions.

Quality of Audio, Speech, and Video Data

Until now, packet loss has been used as an application independent metric for the quality of a connection. When considering audio or video streams, where a codec is used to code the data, the quality of the connection can also be measured by the quality of the received data.

In order to measure differences in the quality of audio, speech, and video streams, three methods which were standardized by the International Telecommunication Union (ITU). Namely, these are Perceptual Evaluation of Audio Quality (PEAQ) [47], Perceptual Evaluation of Speech Quality (PESQ) [53], and Perceptual Evaluation of Video Quality (PEVQ) [51]. All three approaches have in common that

they can be used to automatically measure the quality of the respective data using the same quality measure as humans do. Thus, the enhancement of quality which can be gained by using the packet-classification tool together with an error-tolerant codec could be measured.

7.2 Further Improvements

Although the packet-classification tool already yields a performance increase in the real-time sensitive data transport over lossy links, it is assumed that further improvements are feasible. In this section, possible improvements to the packet-classification tool will be presented.

Variable Parameters

It turned out that the right setting of the parameters highly depends on the protocols being used. This holds especially true for the size of the area which is covered by the packet-classification tool. Especially when used with different protocols whose header sizes are quite distant, it might be beneficial to set the parameters on a per socket basis. Thus, for example, a socket using IPv4 and UDP would set the coverage to 28 bytes while a socket utilizing IPv6 and TCP might choose coverage of 60 bytes. Furthermore, each application could set the `MIN_DELTA` according to its own trade-off between processing rate of corrupted packets and misattribution rate.

Using Soft Information

Jamieson and Balakrishnan [30] suggest *SoftPHY*, an enhancement to the Physical Layer interface. Instead of simply passing the received bits to the upper layers, SoftPHY also offers information on the confidence in each bit value. If this information would be available to the predictor of the packet-classification tool the prediction quality would possibly increase. The predictor compares the bit values of a corrupted packet with stored aggregated data (see Section 3.3.2). These stored data could also be interpreted as confidence in each bit value and thus could be compared to the ones offered by SoftPHY.

Using the confidence information provided by SoftPHY or similar approaches might reduce the impact of bit errors on the prediction quality. It is assumed that erroneous bits have a lower confidence than correct bits and thus contribute less to the packet score which is calculated by the predictor.

7.3 Porting to other Platforms

As the approach proposed by this thesis aims at wireless devices and can cope with changing wireless channel conditions, it is especially well-suited for mobile devices such as notebooks, netbooks, or mobile phones. Thus, porting the packet-classification tool to these platforms would enable them to benefit from the improvements, for example, for audio and video streaming.

Nowadays a wide range of open-source platforms for mobile devices exists. There are, for example, Android (smartphones, netbooks, tablets) [4], Maemo (smartphones, tablets) [43], and MeeGo (netbooks, tablets) [46]. All three are based on the Linux kernel. Thus, porting the packet-classification tool is expected to not be too complicated. However, the reduced resources which are available at devices like smartphones have to be kept in mind. Further optimizations regarding the required processing time and storage space might be necessary.

7.4 Advanced Use Cases

In order to fully profit from the benefits of the proposed packet-classification tool, more use cases have to be implemented. This section gives an overview of promising ideas for further use cases.

More Protocols

Although the combination of IPv4 and UDP today is a well-used protocol stack for transmitting time-sensitive audio and video data, other protocols are used for this task as well or will be in the future. Thus, implementing the utilization of the packet-classification tool for these protocols would enable them to profit from the benefits as well.

Even though TCP was not designed to transport time-sensitive media data, it is nowadays widely used for this purpose as well. Some well-known examples are video streaming (for example YouTube), or internet radio. As mentioned earlier (see Section 6.1.1), TCP especially suffers from packet loss because lost packets will falsely be interpreted as congestion. Previous work has shown both the benefits and challenges of accepting erroneous TCP packets [2, 3]. The major challenge indeed is the correct handling of the sequence number. It is used for in-order delivery of the packets and to detect lost or duplicated packets.

The second protocol which could benefit from utilizing the packet-classification tool is IPv6. As the successor to IPv4, it is likely to become the most used Internet Layer protocol in the next ten years. Its header has a size of 40 bytes of which more than 80 percent are static in a given flow which offers a great data pool for the approach conducted by this thesis. In contrast to IPv4, the header fields of IPv6 are not protected by a checksum. The correctness of the header fields can thus only be checked by the Link Layer and Transport Layer checksums. Hence, with typically used checksums it is no possible to tell if a corrupted packet contains errors in the IPv6 header or not. Thus, utilizing the packet-classification tool could be valuable when processing corrupted IPv6 packets, for example, as a second decision rule if the look-up of the upper layer protocol failed due to bit errors.

Combination with Repair Techniques

As stated earlier, repair techniques, for example the one proposed by Göttgens [19], fail at repairing broken protocol identifier fields. Thus, packets with errors in a

protocol identifier field are either delivered to the wrong upper layer protocol or dropped. A combination of the packet-classification tool with these repair techniques could solve this problem. The prediction result could be used as a hint to which upper layer protocol the packet should be handed. How this is done has been shown in the use case which was implemented (Section 4.3).

But not only a packet repair technique could benefit from using the packet-classification tool. The same holds the other way round. As the repair techniques heuristically repair header fields, it would be possible to refine the prediction by performing a second prediction run on the (partly) repaired header. For the example of IPv4 and UDP the predictor would be run again firstly after the IPv4 header was (partly) repaired and secondly after the UDP header was (partly) repaired. Thus, the prediction result could be verified or even corrected.

8

Conclusion

This thesis has presented a novel approach in handling corrupted packets. Methods from the area of machine learning and data mining were used in order to determine the socket to which a corrupted packet most probably belongs to. This prediction was then used by the network stack to route corrupted packets within the network stack.

At first, the conceptual design of this novel approach has been described. The learner operated on the first part of an uncorrupted packet and transformed these data to an aggregated form. These aggregated data were then stored on a per socket basis. Whenever a corrupted packet arrived, the predictor was invoked. It predicted the destination socket by comparing the data of the incoming packet to the stored aggregated data. This prediction could then be used to process the corrupted packet in the network stack.

By implementing the approach for the network stack of the Linux kernel, it has been shown that the packet-classification tool is usable in practice. This was further supported by measurements performed in a real wireless setting. These measurements showed that the packet-classification performed significantly better than UDP-Lite (which can cope with errors in the payload only) in terms of number of received packets. However, using a probabilistic approach always brings the risk of misattributions.

In order to use the packet-classification tool, some changes to the Linux kernel are necessary. The module which contains the actual packet-classification can however be loaded and unloaded while the operating system is running. One major advantage of the solution proposed by this thesis is that software changes are only necessary on the end host. Thus, the packet-classification tool can be used in every existing wireless network. To fully benefit from the improvements, it is required that the packets are sent with the `QoSNoAck` flag set. As this flag is part of the standardized IEEE 802.11e, support for this should be included in all newer devices.

Measurements in a real wireless network have been performed in order to compare the performance of the packet-classification tool with the performance of UDP-Lite.

Using the packet-classification tool instead of UDP-Lite can significantly increase the number of received packets. The disadvantage of the approach proposed in this thesis is the occurrence of misattributions.

The novel approach which has been introduced by this thesis is based on methods of different fields of research. An overview of these different fields has been given. Whenever appropriate, the approach of this thesis has been compared to other approaches from these fields of research.

During the course of this thesis, some additional code has been developed which might be useful outside the scope of this thesis as well. Firstly, CPEE has been extended to allow the tracking of injected network flows in the network stack. This extension could also be used for all tasks which require the analysis of the behavior of network flows within the Linux kernel. Secondly, a small tool has been developed which allows to induce bit errors to the packets of a network trace in the Link Layer payload only.

To conclude, this thesis has shown that using methods from the machine learning and data mining domain in order to process corrupted packets in the network stack is not only feasible, but also yields very promising results. Improving the design of the system and combining it with other approaches, for example, packet repair techniques, is an exciting opportunity for further research.

Bibliography

[1] Aggarwal, C. C., Han, J., Wang, J., and Yu, P. S. On Demand Classification of Data Streams. In *Proc. ACM KDD Conf.* (2004), pp. 503–508.

[2] Alfredsson, S., and Brunstrom, A. TCP-L: Allowing Bit Errors in Wireless TCP. In *Proc. IST Mobile and Wireless Communications Summit* (June 2003).

[3] Alfredsson, S., and Brunstrom, A. Bit Error Tolerant Multimedia Transport. In *Perspectives on Multimedia*, R. Burnett, A. Brunstrom, and A. G. Nilsson, Eds. John Wiley & Sons, 2004, ch. 10, pp. 175–191.

[4] Android. Online Resource. `http://www.android.com/`.

[5] Arnal, F., Dairaine, L., Lacan, J., and Maral, G. Multi-protocol Header Protection (MPHP), a Way to Support Error-Resilient Multimedia Coding in Wireless Networks. In *High Speed Networks and Multimedia Communications*, Z. Mammeri and P. Lorenz, Eds., vol. 3079 of *Lecture Notes in Computer Science*. Springer, 2004, pp. 740–749.

[6] Balan, R., Lee, B., Kumar, K., Jacob, L., Seah, W., and Ananda, A. TCP HACK: TCP Header Checksum Option to Improve Performance over Lossy Links. In *Proc. IEEE INFOCOM Conf.* (2001), vol. 1, pp. 309–318.

[7] Benvenuti, C. *Understanding Linux Network Internals*, 1st ed. O'Reilly Media, Dec. 2005.

[8] Bormann, C., Burmeister, C., Degermark, M., Fukushima, H., Hannu, H., Jonsson, L.-E., Hakenberg, R., Koren, T., Le, K., Liu, Z., Martensson, A., Miyazaki, A., Svanbro, K., Wiebke, T., Yoshimura, T., and Zheng, H. RObust Header Compression (ROHC): Framework and Four Profiles: RTP, UDP, ESP, and Uncompressed. RFC 3095 (Proposed Standard), July 2001. Updated by RFCs 3759, 4815. `http://www.ietf.org/rfc/rfc3095.txt`.

[9] Bowden, T., Bauer, B., Nerin, J., Feng, S., and Seibold, S. Linux Kernel Documentation: The /proc Filesystem. Online Resource. `http://kernel.org/doc/Documentation/filesystems/proc.txt`, accessed February 22, 2011.

[10] Braden, R. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), Oct. 1989. Updated by RFCs 1349, 4379, 5884. `http://www.ietf.org/rfc/rfc1122.txt`.

[11] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers*, 3rd ed. O'Reilly Media, Feb. 2005.

[12] Cormen, H. T., et al. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[13] Degermark, M., Nordgren, B., and Pink, S. IP Header Compression. RFC 2507 (Proposed Standard), Feb. 1999. `http://www.ietf.org/rfc/rfc2507.txt`.

[14] Degermark, M., Engan, M., Nordgren, B., and Pink, S. Low-loss TCP/IP Header Compression for Wireless Networks. In *Proc. ACM MobiCom Conf.* (1996), pp. 1–14.

[15] Dennisen, D. Integration of a WiFi Tweaking Component into the X-Layer Architecture. Bachelor's thesis, RWTH Aachen University, Germany, Oct. 2009.

[16] Gaber, M. M., Zaslavsky, A., and Krishnaswamy, S. Mining Data Streams: A Review. *ACM SIGMOD Record 34* (June 2005), 18–26.

[17] GNU Netcat. Online Resource. `http://netcat.sourceforge.net/`.

[18] GNU Scientific Library. Online Resource. `http://www.gnu.org/software/gsl/`.

[19] Göttgens, M. Heuristic Packet Repair for UDP/IP in the Linux Network Stack. Bachelor's thesis, RWTH Aachen University, Germany, 2011. to appear.

[20] Hammer, F., Reichl, P., Nordström, T., and Kubin, G. Corrupted Speech Data Considered Useful. In *Proc. 1st ISCA Tutorial and Research Workshop on Auditory Quality of Systems* (Apr. 2003).

[21] Han, B., Schulman, A., Gringoli, F., Spring, N., Bhattacharjee, B., Nava, L., Ji, L., Lee, S., and Miller, R. Maranello: Practical Partial Packet Recovery for 802.11. In *Proc. NSDI Conf.* (2010).

[22] Han, J., and Kamber, M. *Data Mining: Concepts and Techniques*, 2nd ed. Morgan Kaufmann, Mar. 2006.

[23] Hiertz, G., Denteneer, D., Stibor, L., Zang, Y., Costa, X. P., and Walke, B. The IEEE 802.11 Universe. *Communications Magazine, IEEE 48*, 1 (Jan. 2010), 62 –70.

[24] hostapd. Online Resource. `http://hostap.epitest.fi/hostapd/`.

[25] IANA Assignments of Real-time Transport Protocol (RTP) Parameters. Online Resource. `http://www.iana.org/assignments/rtp-parameters`, accessed February 28, 2011.

[26] IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. IEEE 802.11-2007 (Standard), June 2007.

[27] International Standard – Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. ISO/IEC 7498-1 (Standard), Nov. 1994.

[28] Iperf. Online Resource. `http://iperf.sf.net/`.

[29] Jacobson, V. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144 (Proposed Standard), Feb. 1990. `http://www.ietf.org/rfc/rfc1144.txt`.

[30] Jamieson, K., and Balakrishnan, H. PPR: Partial Packet Recovery for Wireless Networks. In *Proc. ACM SIGCOMM Conf.* (2007), pp. 409–420.

[31] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language*, 2nd ed. Prentice Hall, 1988.

[32] Kim, H., Claffy, K., Fomenkov, M., Barman, D., Faloutsos, M., and Lee, K. Internet Traffic Classification Demystified: Myths, Caveats, and the Best Practices. In *Proc. ACM CoNEXT Conf.* (2008), pp. 11:1–11:12.

[33] Knuth, D. E. *Fundamental Algorithms*, 3rd ed., vol. 1 of *The Art of Computer Programming*. Addison-Wesley, 1997.

[34] Lam, P. P.-K., and Liew, S. UDP-Liter: An Improved UDP Protocol for Real-Time Multimedia Applications over Wireless Links. In *Proc. 1st International Symposium on Wireless Communication Systems* (Sept. 2004), pp. 314–318.

[35] Larzon, L.-A., Degermark, M., Pink, S., Jonsson, L.-E., and Fairhurst, G. The Lightweight User Datagram Protocol (UDP-Lite). RFC 3828 (Proposed Standard), July 2004. `http://www.ietf.org/rfc/rfc3828.txt`.

[36] Last, M. Online Classification of Nonstationary Data Streams. *Intelligent Data Analysis 6*, 2 (2002), 129–147.

[37] vom Lehn, H. A Hybrid Emulation Environment for Wireless Networks. Diploma thesis, RWTH Aachen University, Germany, 2010.

[38] Lin, K. C.-J., Kushman, N., and Katabi, D. ZipTx: Harnessing Partial Packets in 802.11 Networks. In *Proc. ACM MobiCom Conf.* (2008), pp. 351–362.

[39] Linux Kernel 2.6.32.15. `http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.32.15.tar.bz2`.

[40] Liu, B., Goeckel, D. L., and Towsley, D. TCP-Cognizant Adaptive Forward Error Correction in Wireless Networks. In *Proc. IEEE GLOBECOM* (Nov. 2002), vol. 3, pp. 2128–2132.

[41] Love, R. *Linux Kernel Development*, 3rd ed. Addison-Wesley, June 2010.

[42] mac80211. Online Resource. `http://wireless.kernel.org/en/developers/Documentation/mac80211`, accessed February 23, 2011.

[43] Maemo. Online Resource. http://maemo.org/.

[44] Mangold, S., Choi, S., Hiertz, G., Klein, O., and Walke, B. Analysis of IEEE 802.11e for QoS Support in Wireless LANs. *IEEE Wireless Communications 10*, 6 (Dec. 2003), 40 – 50.

[45] McCanne, S., and Jacobson, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. USENIX Winter Conf.* (1993).

[46] MeeGo. Online Resource. http://meego.com/.

[47] Method for Objective Measurements of Perceived Audio Quality. ITU-R Recommendation BS.1387-1 (Standard), Nov. 2001.

[48] Mitchell, T. M. *Machine Learning*, 1st ed. McGraw-Hill, 1997.

[49] Netfilter. Online Resource. http://netfilter.org/.

[50] Netstat. Online Resource. http://developer.berlios.de/projects/net-tools/.

[51] Objective Perceptual Multimedia Video Quality Measurement in the Presence of a Full Reference. ITU-T Recommendation J.247 (Standard), Aug. 2008.

[52] OpenFWWF. Online Resource. http://www.ing.unibs.it/~openfwwf/.

[53] Perceptual Evaluation of Speech Quality (PESQ): An Objective Method for End-to-end Speech Quality Assessment of Narrow-band Telephone Networks and Speech Codecs. ITU-T Recommendation P.862 (Standard), Feb. 2001.

[54] Postel, J. User Datagram Protocol. RFC 768 (Standard), Aug. 1980. http://www.ietf.org/rfc/rfc768.txt.

[55] Postel, J. Internet Protocol. RFC 791 (Standard), Sept. 1981. Updated by RFC 1349. http://www.ietf.org/rfc/rfc791.txt.

[56] Radiotap. Online Resource. http://www.radiotap.org/.

[57] Schiller, J. *Mobile Communications*, 2nd ed. Addison-Wesley, 2003.

[58] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003. Updated by RFCs 5506, 5761, 6051. http://www.ietf.org/rfc/rfc3550.txt.

[59] Sevy, J. Linux Network Stack Walkthrough (2.4.20). Online Resource. http://gicl.cs.drexel.edu/people/sevy/network/Linux_network_stack_walkthrough.html, accessed January 18, 2011.

[60] Shakkottai, S., Rappaport, T. S., and Karlsson, P. C. Cross-Layer Design for Wireless Networks. *IEEE Communications Magazine 41*, 10 (Oct. 2003), 74 – 80.

[61] Shanmugasundaram, K. Linux Kernel Linked List Explained. Online Resource. `http://isis.poly.edu/kulesh/stuff/src/klist/`, accessed January 20, 2011.

[62] Srivastava, V., and Motani, M. Cross-Layer Design: A Survey and the Road Ahead. *IEEE Communications Magazine 43*, 12 (Dec. 2005), 112 – 119.

[63] Tanenbaum, A. S. *Computer Networks*, 4th ed. Pearson Education, 2003.

[64] The ns-3 network simulator. Online Resource. `http://www.nsnam.org/`.

[65] Tye, C. S., and Fairhurst, G. A Review of IP Packet Compression Techniques. In *Proc. 4th Annual Postgraduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting* (2003).

[66] Ubuntu Linux Distribution. Online Resource. `http://www.ubuntu.com/`.

[67] Universal TUN/TAP Driver. Online Resource. `http://vtun.sourceforge.net/tun/`.

[68] Williams, N., Zander, S., and Armitage, G. A Preliminary Performance Comparison of Five Machine Learning Algorithms for Practical IP Traffic Flow Classification. *ACM SIGCOMM Comput. Commun. Rev. 36* (Oct. 2006), 5–16.

[69] Willig, A., Kubisch, M., Hoene, C., and Wolisz, A. Measurements of a Wireless Link in an Industrial Environment Using an IEEE 802.11-Compliant Physical Layer. *IEEE Transactions on Industrial Electronics 49*, 6 (Dec. 2002), 1265 – 1282.

[70] Wireshark. Online Resource. `http://www.wireshark.org/`.

[71] Zobel, J. *Writing for Computer Science*, 2nd ed. Springer, 2004.

A

Appendix

A.1 List of Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
ARQ	Automatic Repeat reQuest
BEB	Binary Exponential Backoff
BER	Bit Error Rate
BPSK	Binary Phase-Shift Keying
BSD	Berkeley Software Distribution
BSSID	Basic Service Set Identifier
CRC	Cyclic Redundancy Check
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CPEE	Corrupted Packets Evaluation Environment
CPU	Central Processing Unit
CTS	Clear to Send
DCF	Distributed Coordination Function
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
FCS	Frame Check Sequence
FDDI	Fiber Distributed Data Interface
FEC	Forward Error Correction
FTP	File Transfer Protocol
FPU	Floating-Point Unit
GSL	GNU Scientific Library
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
IHL	IP Header Length
IPv4	Internet Protocol version 4

IPv6	Internet Protocol version 6
ITU	International Telecommunication Union
KDD	Knowledge Discovery in Databases
LKM	Linux Kernel Module
LOS	Line-of-Sight
MAC	Medium Access Control
NAT	Network Address Translation
NAV	Network Allocation Vector
PCF	Point Coordination Function
PEAQ	Perceptual Evaluation of Audio Quality
PER	Packet Error Rate
PESQ	Perceptual Evaluation of Speech Quality
PEVQ	Perceptual Evaluation of Video Quality
PPP	Point-to-Point Protocol
procfs	Proc File System
QoS	Quality of Service
QPSK	Quadrature Phase-Shift Keying
RAM	Random-Access Memory
RFC	Request for Comments
ROHC	RObust Header Compression
RTCP	Real-time Transport Control Protocol
RTP	Real-time Transport Protocol
RTS	Request to Send
SIP	Session Initiation Protocol
softirq	Software Interrupt Request
TCP	Transmission Control Protocol
TOS	Type Of Service
TTL	Time To Live
UDP	User Datagram Protocol
UDP-Lite	Lightweight User Datagram Protocol
VoIP	Voice over IP

A.2 Modifications for the Use Case

In the following, changes to the network stack of the Linux kernel are denoted. The line number of the beginning of a code fragment corresponds to the line number of the unchanged 2.6.32.15 version of the Linux kernel. Changes and additions are highlighted.

A.2.1 Changes to `net/core/dev.c`

```
2350 skb = handle_macvlan(skb, &pt_prev, &ret, orig_dev);
2351 if (!skb)
2352     goto out;
2353
2354 type = skb->protocol;
2355
```

```
2356 if (skb->rep.l2_check_failed) {
2357     type = skb->pacl.pred_protocol;
2358 }
2359
2360 list_for_each_entry_rcu(ptype,
2361         &ptype_base[ntohs(type) & PTYPE_HASH_MASK], list) {
```

Listing A.1. Changes to `netif_receive_skb()` in `net/core/dev.c`.

A.2.2 Changes to `net/ipv4/ip_input.c`

```
408      *  4.  Doesn't have a bogus length
409      */
410
411     if (iph->ihl < 5 || iph->version != 4) {
412         skb->rep.l3_check_failed = 1;
413     }
414
415     if (!pskb_may_pull(skb, iph->ihl*4)) {
416         skb->rep.l3_check_failed = 1;
417     }
418
419     iph = ip_hdr(skb);
420
421     if (unlikely(ip_fast_csum((u8 *)iph, iph->ihl))) {
422         skb->rep.l3_check_failed = 1;
423     }
424
425     len = ntohs(iph->tot_len);
426     if (skb->rep.l3_check_failed) {
427         len = skb_tail_pointer(skb) - (unsigned char*)iph;
428     }
429     if (skb->len < len) {
430         IP_INC_STATS_BH(dev_net(dev),
431                         IPSTATS_MIB_INTRUNCATEDPKTS);
432         goto drop;
```

Listing A.2. Changes to `ip_rcv()` in `net/ipv4/ip_input.c`.

```
353         st[(idx>>16)&0xFF].i_bytes += skb->len;
354     }
355 #endif
356
357     if (!(skb->rep.l3_check_failed)) {
358         if (iph->ihl > 5 && ip_rcv_options(skb))
359             goto drop;
360     }
361
362     rt = skb_rtable(skb);
363     if (rt->rt_type == RTN_MULTICAST) {
```

Listing A.3. Changes to `ip_rcv_finish()` in `net/ipv4/ip_input.c`.

```
259     /*
260      *  Reassemble IP fragments.
261      */
262
263     if (!(skb->rep.l2_check_failed || skb->rep.l3_check_failed)) {
264         if (ip_hdr(skb)->frag_off & htons(IP_MF | IP_OFFSET)) {
265             if (ip_defrag(skb, IP_DEFRAG_LOCAL_DELIVER))
266                 return 0;
267         }
268     }
269
270     return NF_HOOK(PF_INET, NF_INET_LOCAL_IN, skb, skb->dev,
271                    NULL, ip_local_deliver_finish);
272 }
```

Listing A.4. Changes to `ip_local_deliver()` in `net/ipv4/ip_input.c`.

```
207     resubmit:
208         if ((skb->rep.l2_check_failed || skb->rep.l3_check_failed)
209             && skb->pacl.pred_sk != NULL) {
210             protocol = skb->pacl.pred_sk->sk_protocol;
211         }
212
213         raw = raw_local_deliver(skb, protocol);
```

Listing A.5. Changes to `ip_local_deliver_finish()` in `net/ipv4/ip_input.c`.

A.2.3 Changes to `net/ipv4/udp.c`

```
1294    ulen = ntohs(uh->len);
1295    saddr = ip_hdr(skb)->saddr;
1296    daddr = ip_hdr(skb)->daddr;
1297
1298    if (skb->rep.l2_check_failed || skb->rep.l3_check_failed
1299        || skb->rep.l4_check_failed) {
1300        ulen = skb_tail_pointer(skb) - (unsigned char*)uh;
1301    }
1302
1303    if (ulen > skb->len)
1304        goto short_packet;
1305
1306    if (proto == IPPROTO_UDP && !(skb->rep.l3_check_failed) ) {
1307        /* UDP validates ulen. */
1308        if (ulen < sizeof(*uh) || pskb_trim_rcsum(skb, ulen))
1309            goto short_packet;
1310        uh = udp_hdr(skb);
1311    }
1312
1313    if (udp4_csum_init(skb, uh, proto))
1314        goto csum_error;
1315
1316    if (rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
```

```
1317            return __udp4_lib_mcast_deliver(net, skb, uh,
1318                    saddr, daddr, udptable);
1319
1320        sk = __udp4_lib_lookup_skb(skb, uh->source, uh->dest,
1321                                   udptable);
1322
1323        if (sk == NULL && (skb->rep.l2_check_failed || skb->rep.l3_check_failed
1324            || skb->rep.l4_check_failed)) {
1325            if (skb->pacl.pred_sk != NULL)
1326                if (unlikely(!atomic_inc_not_zero(
1327                                &skb->pacl.pred_sk->sk_refcnt)))
1328                    goto drop;
1329            sk = skb->pacl.pred_sk;
1330        }
1331
1332        if ((skb->rep.l2_check_failed || skb->rep.l3_check_failed
1333            || skb->rep.l4_check_failed) && skb->pacl.pred_sk == NULL) {
1334            goto drop;
1335        }
1336
1337        if (sk != NULL) {
1338            int ret;
1339            if (!sk->sk_brokenok) {
1340                if (proto != IPPROTO_UDPLITE) {
1341                    if (skb->rep.l2_check_failed)
1342                        goto csum_error;
1343                }
1344                else {
1345                    if (skb->rep.l3_check_failed)
1346                        goto csum_error;
1347                }
1348            }
1349
1350            ret = udp_queue_rcv_skb(sk, skb);
1351
1352            sock_put(sk);
```

Listing A.6. Changes to `__udp4_lib_rcv()` in `net/ipv4/udp.c`.

```
951             copied = ulen;
952         else if (copied < ulen)
953             msg->msg_flags |= MSG_TRUNC;
954
955         if (skb->rep.l2_check_failed || skb->rep.l3_check_failed
956             || skb->rep.l4_check_failed) {
957             msg->msg_flags |= MSG_HASERRORS;
958             skb->ip_summed = CHECKSUM_UNNECESSARY;
959         }
```

Listing A.7. Changes to `udp_recvmsg()` in `net/ipv4/udp.c`.

A.3 Setup of the Parameter Evaluation

At the beginning, the modules for using the CPEE tool have to be loaded and configured once. In addition, the character device for the communication of the user and kernel space part of CPEE has to be created.

```
# modprobe mac80211
# insmod path_to_cpee/hwsim.ko bssid="00:22:6b:95:20:50"
# insmod path_to_cpee/d2p.ko
# mknod /dev/d2p c 24 0
```

The Basic Service Set Identifier (BSSID) passed to the modified version of hwsim has to be the identifier of the access point used when capturing the trace which should be induced into the network stack. An additional step for preparing the evaluation runs is the invocation of the tool from Section 5.1.1.2 which flags the packets in the trace file according to their UDP destination. As the parameter settings have to be tested for different BER values, the bit error generation program described in Section 5.1.1.3 is called in order to introduce bit errors into the trace file.

For each combination of parameter settings that has to be evaluated, a corresponding version of the packet classification module is built and the module loaded. Now the wireless device simulated by CPEE has to be configured and put up. Furthermore, a routing table entry has to be added.

```
# ifconfig wlan0 hw ether d8:5d:4c:ba:df:90
# ifconfig wlan0 up
# ifconfig wlan0 192.168.1.100 netmask 255.255.255.0
# route add default gw 192.168.1.1
```

The MAC address, IPv4 address, and IPv4 network mask have to be set to the corresponding values of the destination computer used for capturing the trace. The IPv4 gateway address has to be set to that of the gateway used during capturing. For each application flow, one application is started that binds to the corresponding UDP destination port. It counts the total number of received packets, the total number of received uncorrupted packets, the number of received uncorrupted packets designated for this application, the total number of received corrupted packets, and the number of received corrupted packets designated for this application.

The system is now configured to accept injected packets. Thus, the CPEE user space program can now be called to induce the modified trace file into the network stack. When this program terminates, all packets have been processed by the network stack.

Finally, the setup has to be reset in order to create a clean setup for the next test run. Thus, the wireless device is put down, the receiving applications are terminated, and the packet classification module is unloaded.

A.4 Linux Kernel Structures

A.4.1 The sk_buff Structure

```
struct sk_buff {
    /* These two members must be first. */
    struct sk_buff      *next;
    struct sk_buff      *prev;

    struct sock     *sk;
    ktime_t         tstamp;
    struct net_device   *dev;

    unsigned long       _skb_dst;
#ifdef CONFIG_XFRM
    struct  sec_path    *sp;
#endif
    /*
     * This is the control buffer. It is free to use for every
     * layer. Please put your private variables there. If you
     * want to keep them across layers you have to do a skb_clone()
     * first. This is owned by whoever has the skb queued ATM.
     */
    char            cb[48];

    unsigned int        len,
                data_len;
    __u16           mac_len,
                hdr_len;
    union {
        __wsum      csum;
        struct {
            __u16   csum_start;
            __u16   csum_offset;
        };
    };
    __u32           priority;
    kmemcheck_bitfield_begin(flags1);
    __u8            local_df:1,
                cloned:1,
                ip_summed:2,
                nohdr:1,
                nfctinfo:3;
    __u8            pkt_type:3,
                fclone:2,
                ipvs_property:1,
                peeked:1,
                nf_trace:1;
    __be16          protocol:16;
    kmemcheck_bitfield_end(flags1);

    void            (*destructor)(struct sk_buff *skb);
#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
    struct nf_conntrack *nfct;
    struct sk_buff      *nfct_reasm;
#endif
#ifdef CONFIG_BRIDGE_NETFILTER
    struct nf_bridge_info   *nf_bridge;
#endif

    int         iif;
#ifdef CONFIG_NET_SCHED
    __u16           tc_index;   /* traffic control index */
#ifdef CONFIG_NET_CLS_ACT
    __u16           tc_verd;    /* traffic control verdict */
#endif
#endif

    kmemcheck_bitfield_begin(flags2);
    __u16           queue_mapping:16;
#ifdef CONFIG_IPV6_NDISC_NODETYPE
    __u8            ndisc_nodetype:2;
#endif
    kmemcheck_bitfield_end(flags2);

    /* 0/14 bit hole */

#ifdef CONFIG_NET_DMA
    dma_cookie_t        dma_cookie;
#endif
#ifdef CONFIG_NETWORK_SECMARK
    __u32           secmark;
#endif

    __u32           mark;

    __u16           vlan_tci;

    sk_buff_data_t      transport_header;
    sk_buff_data_t      network_header;
    sk_buff_data_t      mac_header;
    /* These elements must be at the end, see alloc_skb() for details.  */
    sk_buff_data_t      tail;
    sk_buff_data_t      end;
    unsigned char       *head,
```

```
            *data;
    unsigned int        truesize;
    atomic_t        users;
};
```

Listing A.8. `include/linux/skbuff.h` (2.6.32.15)

A.4.2 The `sock` Structure

```
struct sock_common {
    /*
     * first fields are not copied in sock_copy()
     */
    union {
        struct hlist_node   skc_node;
        struct hlist_nulls_node skc_nulls_node;
    };
    atomic_t        skc_refcnt;

    unsigned int        skc_hash;
    unsigned short      skc_family;
    volatile unsigned char  skc_state;
    unsigned char       skc_reuse;
    int         skc_bound_dev_if;
    struct hlist_node   skc_bind_node;
    struct proto        *skc_prot;
#ifdef CONFIG_NET_NS
    struct net      *skc_net;
#endif
};

struct sock {
    /*
     * Now struct inet_timewait_sock also uses sock_common, so please just
     * don't add nothing before this first member (__sk_common) --acme
     */
    struct sock_common  __sk_common;
#define sk_node         __sk_common.skc_node
#define sk_nulls_node       __sk_common.skc_nulls_node
#define sk_refcnt       __sk_common.skc_refcnt

#define sk_copy_start       __sk_common.skc_hash
#define sk_hash         __sk_common.skc_hash
#define sk_family       __sk_common.skc_family
#define sk_state        __sk_common.skc_state
#define sk_reuse        __sk_common.skc_reuse
#define sk_bound_dev_if     __sk_common.skc_bound_dev_if
#define sk_bind_node        __sk_common.skc_bind_node
#define sk_prot         __sk_common.skc_prot
#define sk_net          __sk_common.skc_net
    kmemcheck_bitfield_begin(flags);
    unsigned int        sk_shutdown  : 2,
                sk_no_check  : 2,
                sk_userlocks : 4,
                sk_protocol  : 8,
                sk_type      : 16;
    kmemcheck_bitfield_end(flags);
    int         sk_rcvbuf;
    socket_lock_t       sk_lock;
    /*
     * The backlog queue is special, it is always used with
     * the per-socket spinlock held and requires low latency
     * access. Therefore we special case it's implementation.
     */
    struct {
        struct sk_buff *head;
        struct sk_buff *tail;
    } sk_backlog;
    wait_queue_head_t   *sk_sleep;
    struct dst_entry    *sk_dst_cache;
#ifdef CONFIG_XFRM
    struct xfrm_policy  *sk_policy[2];
#endif
    rwlock_t        sk_dst_lock;
    atomic_t        sk_rmem_alloc;
    atomic_t        sk_wmem_alloc;
    atomic_t        sk_omem_alloc;
    int         sk_sndbuf;
    struct sk_buff_head sk_receive_queue;
    struct sk_buff_head sk_write_queue;
#ifdef CONFIG_NET_DMA
    struct sk_buff_head sk_async_wait_queue;
#endif
    int         sk_wmem_queued;
    int         sk_forward_alloc;
    gfp_t           sk_allocation;
    int         sk_route_caps;
    int         sk_gso_type;
    unsigned int        sk_gso_max_size;
    int         sk_rcvlowat;
    unsigned long       sk_flags;
    unsigned long           sk_lingertime;
    struct sk_buff_head sk_error_queue;
```

```
    struct proto        *sk_prot_creator;
    rwlock_t        sk_callback_lock;
    int         sk_err,
                sk_err_soft;
    atomic_t        sk_drops;
    unsigned short      sk_ack_backlog;
    unsigned short      sk_max_ack_backlog;
    __u32           sk_priority;
    struct ucred        sk_peercred;
    long            sk_rcvtimeo;
    long            sk_sndtimeo;
    struct sk_filter        *sk_filter;
    void            *sk_protinfo;
    struct timer_list   sk_timer;
    ktime_t         sk_stamp;
    struct socket       *sk_socket;
    void            *sk_user_data;
    struct page     *sk_sndmsg_page;
    struct sk_buff      *sk_send_head;
    __u32           sk_sndmsg_off;
    int         sk_write_pending;
#ifdef CONFIG_SECURITY
    void            *sk_security;
#endif
    __u32           sk_mark;
    /* XXX 4 bytes hole on 64 bit */
    void            (*sk_state_change)(struct sock *sk);
    void            (*sk_data_ready)(struct sock *sk, int bytes);
    void            (*sk_write_space)(struct sock *sk);
    void            (*sk_error_report)(struct sock *sk);
    int         (*sk_backlog_rcv)(struct sock *sk,
                          struct sk_buff *skb);
    void                    (*sk_destruct)(struct sock *sk);
};
```

Listing A.9. `include/net/sock.h` (2.6.32.15)