FACULTY OF ENGINEERING AND ARCHITECTURE

Cybersecurity: Breaking IEEE 802.11 Devices at the Physical Layer

Thomas Schuddinck
Student number: 01504679

Supervisors: Prof. dr. ir. Ingrid Moerman, Dr. Xianjun Jiao
Counsellor: eng. Rafael Cavalcanti (Keysight Laboratories)

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2021-2022

GHENT UNIVERSITY

Admission To Use

Preface

Cybersecurity has been a topic I have gained a lot of interest in and which I could, thanks to this thesis, explore some more. While I already gotten some basic introduction to hacking websites and local programs, this thesis gave me a different perspective on "how to hack".

I want to thank Prof. Dr. Ir. Ingrid Moerman for all the help and assistance she provided during the thesis. I also want to give my thanks to Dr. Xianjun Jiao for helping me get a good understanding of the Openwifi platform and being always being available to provide quick support when I got stuck. Finally, I also want to express my gratitude to BSc. Rafael Cavalcanti for always trying to find ways to help me and assisting me on all matters concerning Keysight.

Abstract

This thesis further explores the capabilities of fuzzing the physical layer of the IEEE 802.11 standard (more commonly known as Wi-Fi) by creating a framework for PHY fuzzing through user space. The framework utilizes the Openwifi platform to access and change the PHY properties of injected packets in the driver. In earlier work, PHY fuzzing could be done by generating a new driver file for each possible value for the legacy signal field. However, this thesis provides improvements on that work in several areas.

The greatest improvement that the framework offers, is the injection of the signal field from user space. This method avoids the enormous memory overhead which the previous work struggled with, as well as greatly improving on time usage. At the same time, it creates an opportunity for third party fuzzers to implement PHY fuzzing themselves by following the Radiotap format described in this work. Support for the Greenfield (or High Throughput) signal field was also added, while newer versions might be added in the future when the Openwifi platform implements the more recent IEEE 802.11 standards. Validation on TX/RX paths for both attacker and test device were done after each fuzzing attempt. This was done to administer any crashes on either device.

Another goal of this thesis was the split up between on-air information and the hardware parameters. This was going to be achieved by combining this work with the FPGA changes of a parallel thesis. However, due to time problems, this part could not be finished.

Keywords: IEEE 802.11, Wi-Fi, Fuzzing, PHY, Openwifi

Cybersecurity: Breaking IEEE 802.11 Devices at the Physical Layer

Thomas Schuddinck
University Ghent (UGent)
Ghent, Belgium
Thomas.schuddinck@ugent.be

***Abstract*—Cybersecurity is a topic that frequently appears in the media lately. Unfortunately, it is almost always in concurrence with data breaches and hacktivism. One could argue that in cybersecurity no news is good news. With the arrival of wireless networks and the increasing amount of connected devices, the amount of attack surfaces an attacker can exploit increases as well. The need for proper defence mechanisms is therefore essential for fulfilling the security goals of any network. While there are already some solutions to increase security, they have not been proven resistant to all attacks. One way to close these gaps is by actively testing it, using techniques such as fuzzing.**

This thesis further explores the capabilities of fuzzing the physical layer of the IEEE 802.11 standard [1] (more commonly known as Wi-Fi) by creating a framework for PHY fuzzing through user space. The framework utilizes the Openwifi platform to access and change the PHY properties of injected packets in the driver. In earlier work, PHY fuzzing could be done by generating a new driver file for each possible value for the legacy signal field. However, this thesis provides improvements on that work in several areas.

***Index Terms*—IEEE 802.11, Wi-Fi, Fuzzing, PHY, Openwifi**

I. Introduction

Wireless communication, and the associated protocols such as Wi-Fi, Bluetooth, Cellular, Zigbee... are terms that used to be unprecedented a few decades ago, but are infamous today. With 18 billion Wi-Fi devices worldwide at the start of 2022, and with another 4.4 billion to be shipped by the end of the year [2], the number of Wi-Fi devices is only increasing. Unfortunately, the number of potential devices for cyber attacks is also climbing just as fast.

Therefore, the need for solid security is now more important than ever. The harsh reality concerning security is that a defender has to ensure all possible vulnerabilities in a system are closed, while an attacker may only need one weakness to get in. In the ongoing never-ending battle between attackers and defenders, new protections are introduced, forcing attackers to be more creative in their attacks to circumvent these protections. One route is to explore new attack surfaces that have no or limited protections.

For IEEE 802.11 the MAC layer has been the main target for attackers for years with new attacks, like KRACK and FragAttacks, rising up every day. The reality is that as a standard, IEEE 802.11 has a different implementation for every vendor. While an implementation might be conform to the standard, it doesn't necessarily mean it has all the security measures in place, or that these implementations are correct. In an attempt to discover new vulnerabilities in Wi-Fi networks, a rather unexplored attack surface will be studied and subjected to tests: the physical layer.

Earlier work [3] attempted fuzzing the OFDM PLCP legacy signal field by generating new driver files that overwrote the hard-coded PHY parameters. This came at a high price as the time, and especially memory overhead, was massive. This work explores a more dynamic way of fuzzing the information in the physical layer.

II. Fuzzing

Software testing is definitely a research domain on its own. It is a necessary step when developing a new software project and a lot of different strategies are available to use on a system or software under testing (SUT). Fuzzing is one of these techniques, and a quite popular one. The idea behind fuzzing is that malformed data packets are sent to a specific program with the intent of crashing it, or to make it show unexpected behaviour. That program being used to send these customized packets to a system is called a fuzzer. [4]

In Fuzzing there are, just like with hacking/testing in general, three different environments. The distinction between each environment is based on the knowledge an attacker/tester has about the system. These environments are as follows: white-box, grey-box and black-box fuzzing, where an attacker has a glass-box overview of the system in case of white-box fuzzing and almost no knowledge when performing black-box fuzzing. For this thesis, grey-box fuzzing is used.

III. Test Setup

Three devices were used for the test setup: 1) A Xilinx development board (with SDR board for the RF frontend), which ran the Openwifi image and was used as the attacking device, sending fuzzed packets over air, 2) a Raspberry Pi, which monitored the packets sent over air and was also used as the target device for fuzzing and 3) the control device, being a laptop running Ubuntu 18.04, that was utilized to control the before mentioned devices via a wired connection. This same wired connection was used by the board to perform life-checks on the targeted WNIC on the Raspberry Pi.

IV. Fuzzing Architecture

The fuzzing architecture spanned over multiple layers as well as crossed the boundary between kernel and user space. The idea was that the injection of PHY information should happen from user space, so that the signal field could dynamically be changed at runtime. This also provided an interface for existing fuzzers, enabling them to support PHY fuzzing themselves.

The extraction of the signal field information and setting the hardware parameters was done in the driver file of the Openwifi platform. These values would be used later on by the FPGA for transmitting the fuzzed packet over air (via the SDR) in order to fuzz the target device (Raspberry Pi).

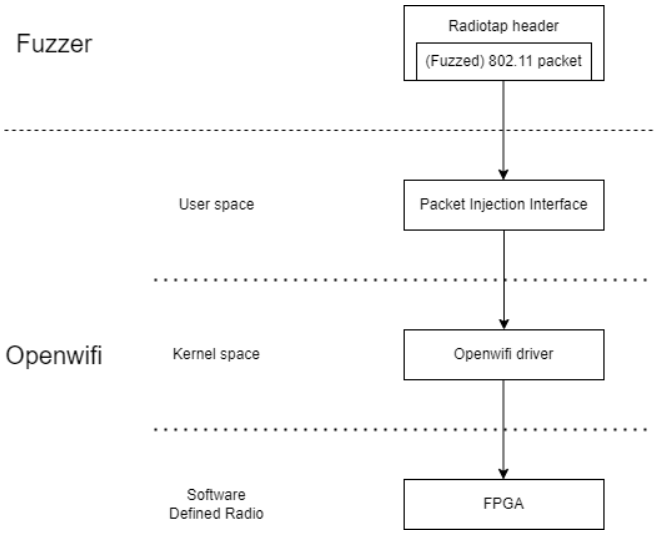

Fig. 1. Fuzzing Architecture

A. Openwifi

The Openwifi project [5] is a SDR based 802.11 implementation for Linux. As of the time of writing this thesis, the project supports 802.11 versions a, g and n, with version ax being under development. The complete open-source project can be found on github [6].

Openwifi consists of three main physical components: the ARM processor, FPGA and RF frontend. The FPGA allows to change the PHY properties of packet before transmission and therefore it is possible to fuzz physical layer. In code, the most import part for the fuzzing is the driver file, where the extraction and configuration of the signal field information would take place.

B. Fuzzer

The fuzzer was built around an already existing packet spammer script (as part of the Openwifi project) which could be called from user space and was able to inject Wi-Fi packets with predefined MAC and PHY parameters. PHY parameters could be set using the Radiotap header. Radiotap [7] is a header format that provides additional information on top of 802.11 frames but is not part of the IEEE 802.11 standard. One of the provided fields was the Time Synchronization Function Timer, or TSFT, which isn't used during packet transmission and could therefore be used to transport the signal field.

The fuzzer allowed for both legacy and Greenfield/HT signal field injection, as well as configuring the increment value after each iteration and multiple other parameters.

C. Linux Wireless Networking Stack

From top to bottom, the Linux wireless networking stack consists of the nl80211, cfg80211 and mac80211 layer. For the signal field (transported via the Radiotap header) to reach the driver, it first had to pass these layers. This is where things become tricky: for the fuzzing to work as desired, the PHY data should not be altered when moving down these layers. In the networking stack, a certain struct is of utmost importance, the sk_buff (or socket buffer). This struct will not only be used by three earlier mentioned layers, but also by the Openwifi driver file, to further process the packets to be transmitted. Since neither the networking stack nor the buffer provides functionality to handle the TSFT field, the contained signal field data appears to be lost.

One way to solve this would be to alter the code in these layers and perhaps the struct itself. This approach is very likely to fail since one simple line of code, may cascade in countless problems down the line. Another solution came from analyzing the buffer, more specifically the headroom. In monitor mode, the signal field information was moved to the headroom,which is considered garbage, but was not wiped. Taking advantage of this discovery, the PHY data could be extracted from the sk_buff headroom in the driver, where it can be used to override the already constructed signal field.

V. Validation

Since manually verifying if both devices are still functioning properly, every time a fuzzed message is being injected and transmitted, is quite time-consuming task, an automatic check was implemented. Depending on the operational mode of the SDR (ad-hoc or monitor), a ping request would be sent from either the board or the control the device, to the targeted NIC. This operation would thereby increase the TX and RX counters for that interface (in case these paths are still up).

Afterwards, the fuzzer will verify all four counters (its own TX and RX, as well as for the target) and compare them to the last measured values for these counters. The result is a four bit word for which every bit represents a single path. In case a counter is the same as its old value, its corresponding bit would be set to 1. Finally, the validation result, as well as the injected values, would be written to a log file.

VI. Conclusion

While fuzzing tests were run on the board, there number was quite limited. Most tests focused on verifying the functionality of this approach, rather than discovering vulnerabilities in the targeted device. From the small number of tests, it could be learned that length field was prone to break the TX path on

the board for both the legacy and Greenfield signal fields. This meant that the board was no longer able to send packets, i.e the board itself was fuzzed. The other fields did not seem to affect the transmission itself, but the reception was impacted by changing the data rates and the tail. Since no extensive fuzzing was done, and no issues were found, the work remains a proof-of-concept for now. This work simply proofs that dynamic PHY fuzzing from user space at runtime is possible.

Overall, this approach definitely has its merits over previous work. The time, and especially memory overhead, is drastically decreased, since no driver files have to be generated and the signal field can be changed at runtime. Using the same Radiotap format as the fuzzer for this work, third party fuzzers might implement PHY fuzzing themselves. Deviating from the suggested format is possible, but the necessary changes to the driver file are necessary, as is checking that the data has not been changed in any of the Linux wireless networking stack layers.

This approach also has its downsides. A future change in the networking stack might cause this approach to work no longer. The injection of the signal field is also only possible in monitor mode. In other modes the Radiotap information is not available in the driver. Finally, the current field (TSFT) is not large enough to support newer modes (VHT and HE), and while this might be fixed by adding more (custom) fields, it must be noted that the size of the headroom must be respected (to avoid kernel panics) and that some information might still be lost. As mentioned before, making changes to the Linux wireless networking stack is highly discouraged and is almost certainly bound to bring along a ton of (unpredictable) issues.

One of the more important subgoals of the thesis was to differentiate between the signal field parameters set in the driver and the on-air signal field information. This will avoid the FPGA from crashing when certain illegal values are used. This might also open the door for more fuzzing opportunities, as the hardware information in the on-air bytes does not match the hardware parameters that were used to send the packet. Unfortunately, this part was completed by the end of this work.

Finally, this thesis also briefly looked into the possibility to combine an existing fuzzer, Greyhound, with this physical layer fuzzing. This was done by utilizing the same injection library that Greyhound uses and appending a Radiotap header, conform to the format that fuzzer in this work uses, and verifying the injected data in the driver. No effort was made to port the Greyhound itself onto the board and to test the combination of the two directly, since this porting was already done in previous work [8] and would therefore be trivial.

REFERENCES

[1] Institute of Electrical and Electronics Engineers. 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 Std 802.11-2020 (Revision of IEEE Std 802.11-2016)*, pages 1–4379, 2021.

[2] Wi-Fi Alliance. Wi-fi alliance® 2022 wi-fi® trends, Jan 2022. Accessed on 14 May 2022.

[3] Steven Heijse and Ingrid promotor (viaf)306106406 Moerman. Ieee 802.11 fuzz-testen van de fysische laag gebruik makend van openwifi, 2021. Accessed on 19 October 2021.

[4] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Chapter one - security testing: A survey. In Atif Memon, editor, *Security Testing: A Survey*, volume 101 of *Advances in Computers*, pages 1–51. Elsevier, 2016. Accessed on 10 October 2021.

[5] Xianjun Jiao, Wei Liu, Michael Mehari, Muhammad Aslam, and Ingrid Moerman. openwifi: a free and open-source ieee802. 11 sdr implementation on soc. In *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pages 1–2. IEEE, 2020. Accessed on 12 March 2022.

[6] Xianjun Jiao, Wei Liu, and Michael Mehari. open-source ieee802.11/wi-fi baseband chip/fpga design, 2019.

[7] David Young. Radiotap, 2009. Accessed on 11 April 2022.

[8] Stef Pletinck and Ingrid promotor (viaf)306106406 Moerman. Greybox wi-fi fuzzing op het openwifi platform, 2021. Accessed on 28 February 2022.

Contents

List of Figures

List of Tables

List of listings

List of Abbreviations

ACK	Acknowledgement
AES	Advanced Encryption Standard
AP	Access Point
API	Application Programming Interface
AS	Authentication Server
BSS	Basic Services Set
BSSID	Basic Service Set Identifiers
CCMP	Counter Mode Cipher Block Chaining Message Authentication Code Protocol
CFP	Contention-Free Period
CIA	Confidentiality, Integrity and Availability
CLI	Command Line Interface
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CTS	Clear to Send
DCF	Distributed Coordination Function
DDoS	Distributed Denial of Service
DIFS	Distributed Inter-Frame Space
DLL	Data Link Layer
DoS	Denial of Service
DSSS	Direct Sequence Spread Spectrum
DUT	Device Under Test
EAP	Extensible Authentication Protocol
EAPOL	EAP over LAN
ESS	Extended Service Set
FC	Frame Control
FCS	Frame Check Sequence
FHSS	Frequency Hopping Spread Spectrum
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface

HCF Hybrid Coordination Function
HE High Efficiency
HT High Throughput

IBSS Independent Basic Service Set
IP Internet Protocol

KRACK Key Reinstallation Attack

LAN Local Area Network
LLC Logical Link Control
LSB Least Significant Bit

MAC Medium Access Control
MIC Message Integrity Code
MIMO Multiple-Input Multiple-Output
MITM Man-in-the-Middle
MPDU MAC Protocol Data Unit
MSB Most Significant Bit
MSDU MAC Service Data Unit

NAV Network Allocation Vector
NIC Network Interface Controller
NIST National Institute of Standards and Technology

OFDM Orthogonal Frequency Division Multiplexing

PBSS Personal Basic Service Set
PCAP Packet Capture
PCF Point Coordination Function
PCP PBSS Control Point
PHY Physical Layer
PLCP Physical Layer Convergence Protocol
PMD Physical Medium Dependent
PNAC Port-Based Network Access Control
PPDU PLCP Protocol Data Unit
PSDU PLCP Service Data Unit
PTK Pairwise Transient Key

QoS	Quality of Service
RADIUS	Remote Authentication Dial In User service
RDP	Remote Desktop Protocol
RF	Radio Frequency
RTS	Request to Send
RX	Receive
SAE	Simultaneous Authentication of Equals
SDR	Software Defined Radio
SIFS	Short Inter-Frame Space
SSH	Secure Shell
SSID	Service Set Identifier
STA	Station
SUT	System Under Test/Software Under Test
TCP	Transport Control Protocol
TKIP	Temporal Key Integrity Protocol
TSFT	Time Synchronization Function Timer
TX	Transmit
VHT	Very High Throughput
VNC	Virtual Network Computing
WDS	Wireless Distribution System
WEP	Wired Equivalent Privacy
WLAN	Wireless Local Area Network
WNIC	Wireless Network Interface Controller
WPA	WiFi Protected Access
WPA2	WiFi Protected Access 2
WPA3	WiFi Protected Access 3

1

Introduction

Wireless communication, and the associated protocols such as Wi-Fi, Bluetooth, Cellular, Zigbee... are terms that used to be unprecedented a few decades ago, but are infamous today. With 18 billion Wi-Fi devices worldwide at the start of 2022, and with another 4.4 billion to be shipped by the end of the year[1], the number of Wi-Fi devices is only increasing. Unfortunately, the number of potential devices for cyber attacks is also climbing just as fast.

There are numerous reasons why cyber criminals would attack your devices: extract and sell data, obstructing the normal operations and demand payment, creating chaos for political reasons, or even just for fun. The price tag for failing to prevent these attacks can be extremely high and everyone is a target, whether you are an individual or company. One of the larger cyber attacks in 2021, the attack on Colonial Pipeline in May, cost the company 4.4 million USD, of which the FBI was able to recover 2.3 million[2]. It is estimated that by 2025 cybercrime will come at a cost of 10.5 trillion USD globally per year[3].

So with all this cybercrime, how can these attacks be prevented? The best solution to protect your devices, data, valuables... would be to bury them in an unbreakable vault at the bottom of the ocean. While this is definitely a go-to option if protection is your primary goal, it renders these items useless. Hackers won't be able to access your device, but neither will you (except if you're in the safe). Increasing ease of use and accessibility almost always goes hand-in-hand with decreasing protection, and this is also visa versa. In other words, in the field of cybersecurity compromises are inevitable, so it is up to the developers of security standards and systems to take in account the ease of use while still providing maximum security.

To negate attacks on Wi-Fi devices and enforce certain layers of protection, protocols such as the different WPA versions have been introduced and these have definitely protected against a lot of attacks from intruders. Unfortunately, it is virtually impossible to write an airtight program that is protected against every possible attack. Attacks such as KRACK[4] and DragonBlood[5] have proven that there are still some ways for attackers to get in, despite the fact that these security protocols have been reviewed by teams of experts. This holes in security don't necessarily have to originate from the definition of the standards, but were potentially introduced in the implementation of them. It is up to the development teams of the vendors of Wi-Fi products to not only implement these protections using the standard as a raw blueprint, but also to ensure that there are no vulnerabilities introduced in software or hardware during development.

One approach for finding and exploiting weaknesses in the IEEE 802.11 standard and the devices supporting it, is by using fuzzing. Fuzzing is a testing technique where packets are crafted and sent to a target, whether it be a different device or

software on the attacker's device. For Wi-Fi, these packets can be crafted more precisely, using knowledge about the standard and thereby avoiding the need for brute forcing all possible inputs. These packets may or may not be valid, but their purpose is the same either way: find and/or exploit a weakness in the target. While manual fuzzing is possible, automated fuzzing is a lot more efficient. Different open source Wi-Fi fuzzers that automate this process are already available (Owfuzz[6], Wifuzzit[7]...).

Not only does the input of a packet impact the targeted device but the current state a device is in is also of great importance. A recent fuzzer called Greyhound [8], uses a state machine to observe forbidden transitions between states by using the standards as a blueprint for valid behaviour. This means that if a change in state does not adhere to the defined set of state transmissions, it is considered an illegal action that may lead to the possible exploits. Greyhound was not only able to verify existing vulnerabilities but also discover new ones.

While these fuzzer definitely have their merits, they only look at one out of the two layers that the IEEE 802.11 standard defines: the MAC layer. The bottom most layer in the standard and the ISO model in general, the physical layer, has only recently been getting some attention[9]. So there is a lot of room left to explore this layer and the vulnerabilities that still lie hidden within it.

Problem Statement

While fuzzing on IEEE 802.11 devices is definitely not new, the fuzzing of the Physical layer is. The Physical layer is usually implemented in hardware only, but with the help of the Openwifi platform and the on-board Field-Programmable Gate Array (FPGA), this can be achieved. In earlier research, fuzzing the legacy signal field of the OFDM PLCP header was possible by generating a new driver file where the hard-coded hardware properties were changed. While this was a working solution, it suffered from a few drawbacks.

This thesis aims to find an alternative approach where the signal field can be fuzzed at runtime, without the need of rewriting and recompiling driver files every time, to reduce overhead in both time and memory. Since the legacy signal field is not the only version, providing support for e.g. Greenfield/HT mode is also addressed.

Since there are already a few IEEE 802.11 fuzzers for the MAC layer, providing an interface for these third-party fuzzers and thereby allowing multilayered fuzzing, might also reveal more (unknown) vulnerabilities. Therefore this approach should take in account kernel and user spac by changing the driver in kernel space, while providing injection in the user space. It is also important that this approach doesn't affect the layers in-between, i.e. the Linux Wireless Networking stack (nl80211, mac80211...), meaning that the injection should not break the standard.

Another thesis, which ran in parallel with this one, would allow to differentiate between the on-air signal field and the one used in the hardware as parameter for transmission of the packets. However, this research was not concluded by the end of this thesis so this functionality has yet to be explored.

Because the Openwifi platform currently only supports IEEE 802.11 versions a, g and n, evidently the injection will only target

these versions of the standard.

The remainder of the thesis is structured as follows: In chapter 2, some background will be given on the IEEE 802.11 standard. Next up will be some more information on the standard, specifically on the security side. In chapter 4, the test technique (fuzzing) will be briefly discussed, as some short introduction on the Openwifi platform and previous work concerning fuzzing on this platform. Chapter 5 will discuss the physical setup, as well as the configuration of the devices involved. Chapter 6 will discuss the fuzzing architecture for this approach, the software that will be used and the steps in the fuzzing script. Chapter 7 will look into the results of the testing and will be followed by a summarizing and a conclusion in chapter 8.

2

IEEE 802.11

In this section the network standard for IEEE 802.11, or Wi-Fi, networks will be discussed. After completing this chapter, the reader should have the necessary understanding of the principles, the architecture and protocols in the 802.11 standard that are relevant for this thesis. For the complete description of the 802.11 specification, the reader can find more information in [10].

2.1 Network Protocol Stack

A computer network is a very complex system and saying it is just a combination of hardware and software components used to send data from one device to another, would be vastly oversimplifying it. To group network protocols and devices, the International Organization for Standardization, or ISO, created the OSI model. This model conceptualises seven layers on top of each other, with each having different functionalities. Network protocols and devices are assigned to a layer, based on the functionality they provide.

The seven layers are shown in figure 2.1, starting from the physical layer on the bottom up to the application layer at the top. For this thesis, only lower two (physical and data link layer) are of interest. Each of these layers are implemented in software, hardware or a mix of both. For the upper layers, from the Application up to the Transport layer, are always implemented in software, while the Physical and Data Link layer are usually implemented in hardware, or more specifically, in the network card. In most cases, the Network layer is a combination of the two. [11]

The OSI model is not the only reference model for network protocols out there: the TCP/IP model is another popular model that groups the upper three layers from the OSI model, the Application,Presentation and Session layers, into one single 'Application' Layer.

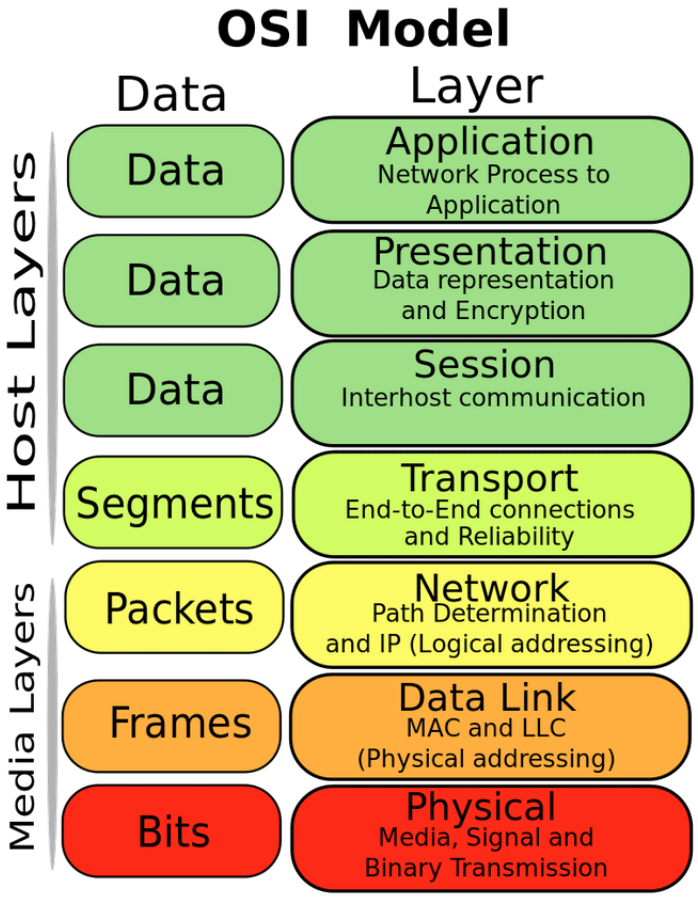

Figure 2.1: OSI model (source: https://www.subpng.com/png-arhoqz)

Standard	Frequency	Bandwidth	Modulation	Alt. Name
802.11	2.4 GHz	20 MHz	DSSS, FHSS	
802.11b	2.4 GHz	20 MHz	DSSS	Wi-Fi 1
802.11a	5 GHz	20 MHz	OFDM	Wi-Fi 2
802.11g	2.4 GHz	20 MHz	DSSS, OFDM	Wi-Fi 3
802.11n	2.4 GHz, 5 GHz	20 MHz, 40 MHz	OFDM	Wi-Fi 4
802.11ac	5 GHz	20 MHz, 40 MHz, 80 MHz, 160 MHz	OFDM	Wi-Fi 5
802.11ax	2.4 GHz, 5 GHz, 6 GHz	20 MHz, 40 MHz, 80 MHz, 160 MHz	OFDM	Wi-Fi 6

Table 2.1: Overview of 802.11 standards

2.2 802.11 Introduction

IEEE 802.11, or more commonly referred to as **Wi-Fi**, is an IEEE standard for wireless networks. It is part of the wider collection, known as the 802 Network standards. This standard works on the two lowest layers of the OSI model: the Data Link layer and Physical layer. The upper layers (Network layer, Transport layer,...) are therefore not involved and have no impact on the Wi-Fi standard. Hence, they will not be further discussed.

Since the first version in 1997, 802.11 underwent a lot of changes and multiple different versions of the standard are available since then. A list with the most popular versions can be found in table 2.1. What is interesting about these different versions, is that they are backwards compatible. A user with a smartphone that is 802.11g compatible should have no problem connecting to an AP with the 802.11ac standard.

2.3 802.11 Architecture

The 802.11 architecture exists of two physical elements: stations and access points. A **station** (STA) is a device that's 802.11 compatible and is almost always an end user device such as a smartphone, tablet, computer,... It's sometimes referred to as a client (device).

An **access point** (AP) provides network access to all connected devices. It functions as a central hub for stations in the vicinity that routes traffic from a source to its designated destination.

The most basic 802.11 architecture consists of one or more stations and an access point. This structure is referred to as a **Basic Service Set** (BSS). It can be identified by stations in the range of the BSS using its **Service Set Identifier** (SSID). This identifier will be send out as a beacon frame by the AP. Note that doesn't have to be unique, it's perfectly possible to have multiple APs with the same SSID. Since To differentiate between APs that share the same SSID, the MAC addresses of the APs are used. These MAC addresses are called **Basic Service Set Identifiers** (BSSID).

An access point has a limited range and can therefore not supply an entire building (effectively). This is where the **Extended Service Set** (ESS) comes in. They are in essence a group of multiple BSSs that are working together to provide a wider coverage. All the access points in this ESS share the same SSID but have a different BSSID.

Finally, there is the **Independent Basic Service Set** (IBSS). In this service set, the stations don't use an access point but communicate directly with each other. It is also commonly referred to as an ad hoc network. This is mostly used for file transfers between user devices without the need of internet access. The range for these IBSSs is also quite limited.

There's also a service set that is very similar to IBSS and the BSS: the **Personal Basic Service Set** (PBSS). For this architecture, there are no APs but a station takes on the role that is similar to that of an AP. This station is referred to as the **PBSS Control Point** (PCP).

2.4 Data Link Layer

The **Data Link layer** (DDL), the second layer in the OSI model,consists of two sublayers: the **Logical Link Control** (LLC) and **Medium Access Control** (MAC) sublayers. While the **Network layer** is responsible for the traffic between a source and destination host (identified by their corresponding IP addresses), the Data Link layer handles the traffic between two intermediate nodes, connected by a communication link or medium. This medium can be wired (UTP cable, fiber optic cable...) or wireless (using radio frequencies or infrared signals). The DLL can perform the following services/tasks:

1. Framing
2. Addressing
3. synchronization
4. Error Control
5. Flow Control
6. Multi-Access

2.5 Logical Link Control

The **Logical Link Control sublayer** (LLC) is the upper sublayer within DLL. Aside from exposing an interface between the MAC layer and the upper layers, it's responsible for performing the **Error** and **Flow Control** task. If for instance, a frame loss is detected, the LLC may resend the lost frames. The data that passes the LLC is referred to as the MAC Service Data Unit or MSDU. These data units contain the IP packets (from the Network layer), that contain information from the upper layers and some additional LLC data.

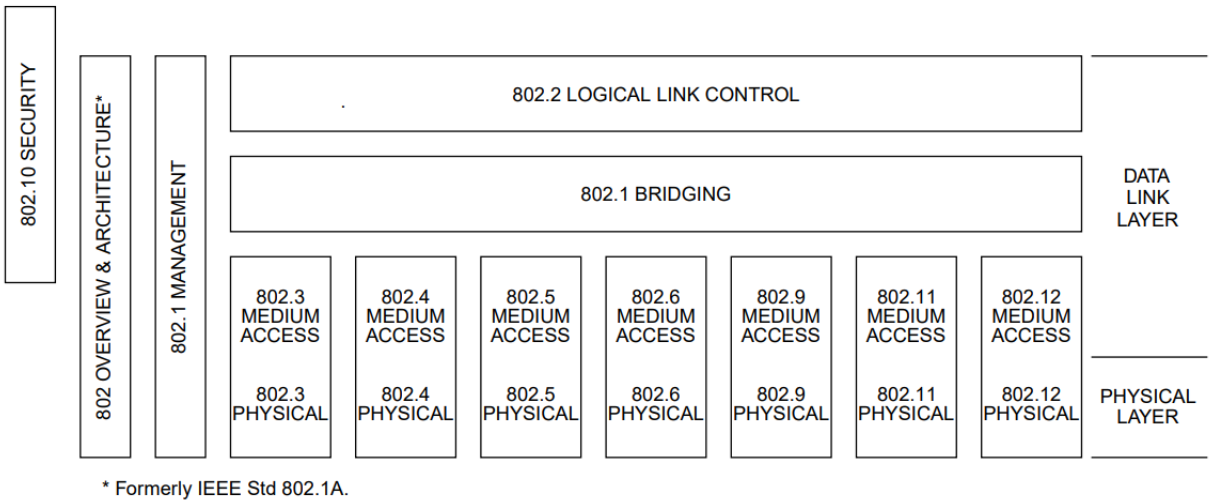

Figure 2.2: 802 family of standards[12]

Figure 2.2 shows an overview of the 802 standards, going from 802.1 to 802.12. A lot of these standards and their corresponding research groups have been disbanded and a lot of new ones have been introduced, as of today. It is clearly shown that the Logical Link Control sublayer is defined in the IEEE 802.2 standard and is therefore not part of the 802.11 standard but used in combination with the LLC standard. As of today, the IEEE 802.2 standard has been adopted into the ISO/IEC 8802-2 standard. Since the upper sublayer of the DLL is not part of the 802.11 standard, it will not be targeted by the fuzzing tests, nor will it be discussed any further in this paper. The reader can read more about the 802.2 LLC in [13].

2.6 MAC 802.11

While not accounting for the LLC sublayer, the 802.11 standard does define it's own version of the **Medium Access Control** sublayer of the DLL. This specific sublayer is referred to as the 802.11 MAC. From now on when the MAC sublayer is mentioned, it refers to the one in the IEEE 802.11 standard, unless specified otherwise.

The MAC sublayer in general fulfils the remaining tasks of the DLL: framing, addressing, synchronization and multi-access. The 802.11 version also provides reliable data delivery and wireless access control protocols. Before going further into the mechanisms the MAC 802.11 provides some problems that wireless networks face will be briefly discussed.

Noise and interference

The first problem that wireless networks face is that the environment where they operate can be highly unpredictable. There can be noise from nearby electronic devices that emit RF radiation, such as microwaves. Walls and furniture can also obstruct the data transmission and drastically limit the operational range.[14]

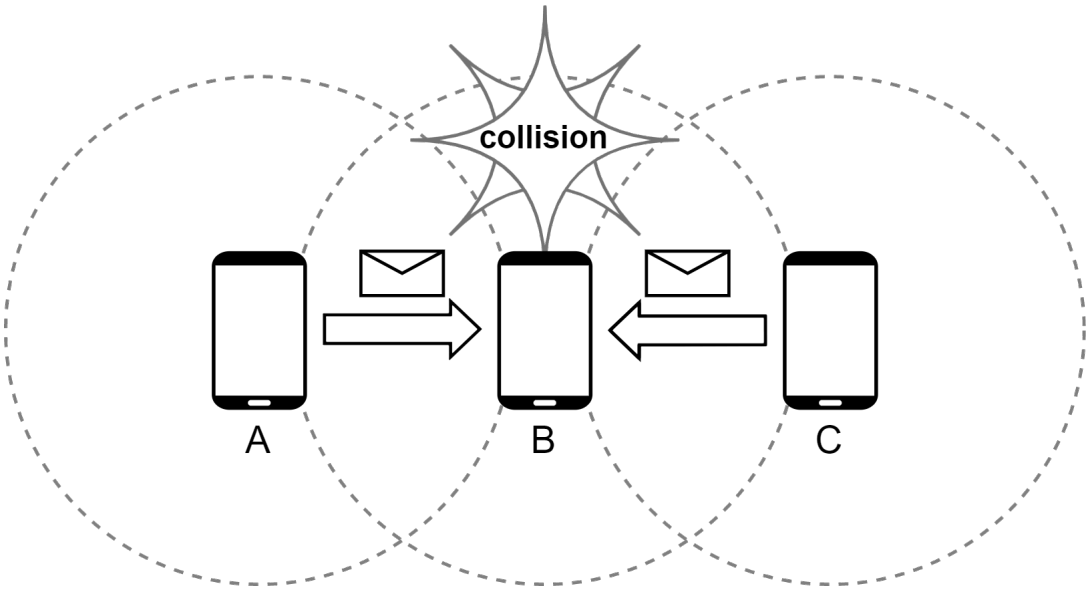

Figure 2.3: Hidden Node Problem (inspired by Fig. 1[15])

Hidden Node Problem

Consider the network in Figure 2.3: two nodes, A and C, in this network try to communicate with the same node B. If node B receives both messages simultaneously, it will be unable to comprehend either message. An analogy to the real world could be screaming your order at the same time as someone else in a Starbucks. Not only will you get dirty looks from the surrounding people, you'll have to repeat the order again (and without others interfering) in order for the personnel to prepare your coffee.

Just like in the real world this could easily be resolved by just verifying that there are no other nodes in the vicinity sending out messages to node B, before sending your own. This is however where the hidden node comes in. Neither A not C is aware of the other and has no way of knowing beforehand that the other is already communicating with B. From node A's perspective, Node C is a *hidden* node (hence the name **hidden node problem**).[14]

Collisions

No matter what type of network, collisions are bound to happen and should be properly handled to unsure the proper delivery of data transmissions. A scheme was implored to tackle this issue: **Carrier Sense Multiple Access** (CSMA).

There are two types for CSMA: one with collision avoidance CSMA/CA and one with collision detection CSMA/CD. The latter is used in Ethernet: rather than avoiding collisions, there's a mechanism to detect collisions. Once a collision is detected a node will stop sending data and will wait for a certain back-off period.

Since wireless transceivers are half-duplex, meaning they don't transmit and receive at the same time, detecting collisions

with hidden nodes (see earlier) are hard. Therefore CSMA/CA is used in MAC 802.11 instead.[14]

2.6.1 Medium Access

In order for a node in a network to send data to others nodes, it first has to get access to the medium. In case of a wireless network, this medium is over air using RF.

There are three modes defined for accessing the medium in 802.11: DCF, PCF and HCF.

DCF

Distributed Coordination Function, or DCF, is the base function which the other two are built upon. In order to prevent collisions every station has to wait a certain period of time between transmissions of frames. DCF enforces the Distributed Inter-Frame Space (DIFS) for every station to wait for a random duration before trying to access the medium. Another option to reduce collisions is the use of CTS/RTS.

PCF

Point Coordination Function (PCF) is an optional extension of DCF in order to provide contention-free. It's a centralized way of granting access to the medium using a point coordinator that informs the network on which node can access the medium at that time. It's rarely used and will not be further discussed.

HDCF

As a way of a compromising between DCF and HCF, a hybrid solution was introduced. **Hybrid Coordination Function** (HCF) provides a prioritized Quality of Service (QoS) by utilizing a QoS coordinator which is similar to the point coordinator in PCF.[16]

CTS/RTS

In order for wireless nodes to further improve the prevention collisions the MAC provides two frames: the **Request to Send** (RTS) and **Clear to Send** (CTS) frames. Figure 2.4 illustrates how these frames are used to prevent collisions.

Consider a node A that wants to send data to node B. Before starting to sending data it first sends a RTS message to indicate its intention of sending data to B. if it's the first node to send this frame, node B will send a CTS message to A and all other nodes in the vicinity to indicate that only A can send data to B. A can now start sending data to B while all others are unable

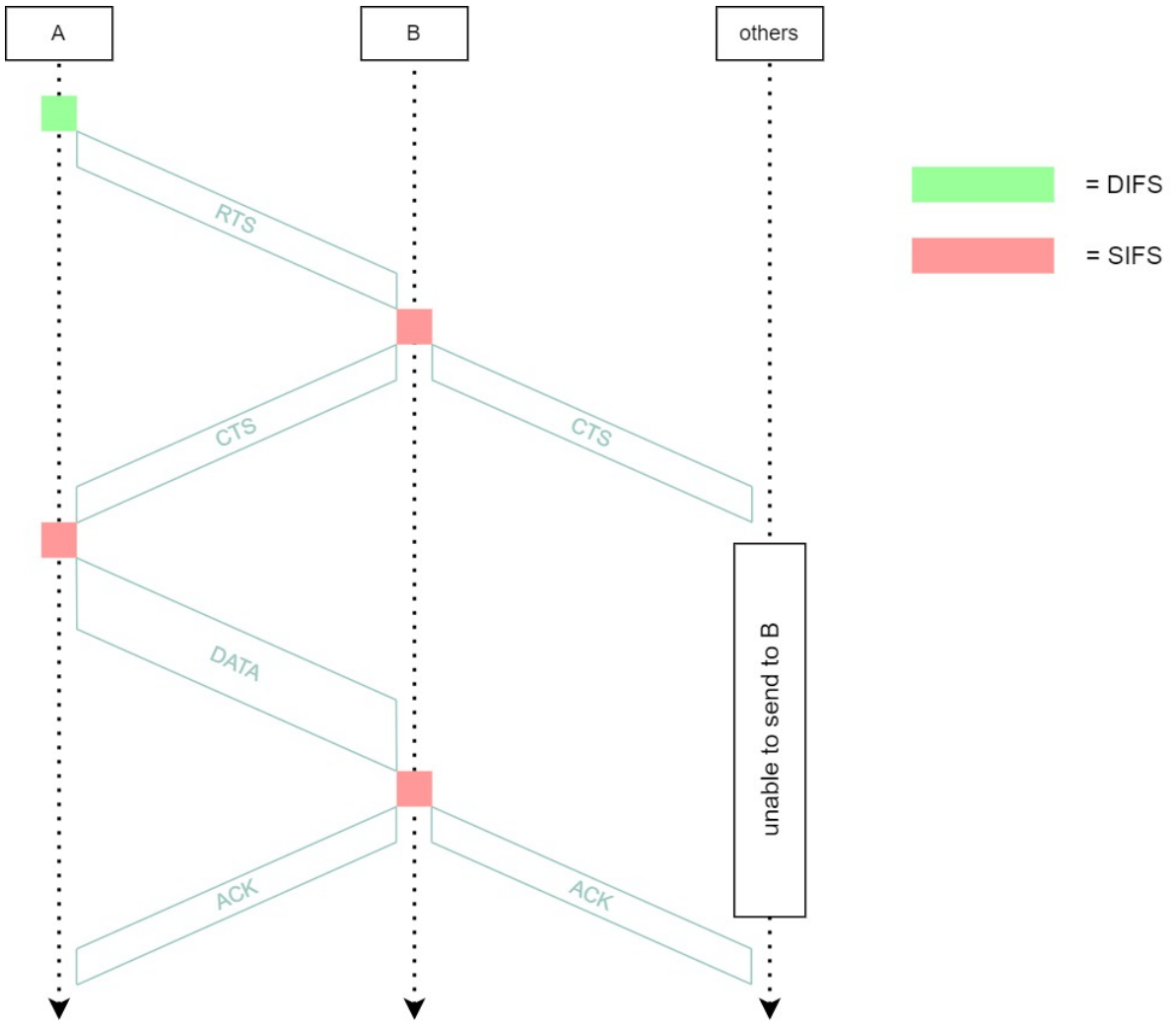

Figure 2.4: Collision prevention in MAC 802.11 (inspired by Afbeelding 7.12[11])

to. When A is done sending data B will broadcast an acknowledgement ACK message to all nodes to indicate a node may request permission to send data by sending a new RTS message (after waiting out their DIFS period).

Note that hidden nodes are no longer a problem since the destination node controls whether the data can be send, not the source.[11]

2.6.2 MAC Frame

The first function of the MAC layer is framing: The data from the LLC, the MSDUs, are passed down to the MAC layer where they are packed into a frame with a MAC header in front and a trailer at the end. The MAC frame has the following structure as shown in Figure 2.5. All fields, except for the frame body and FCS, are part of the frame header.

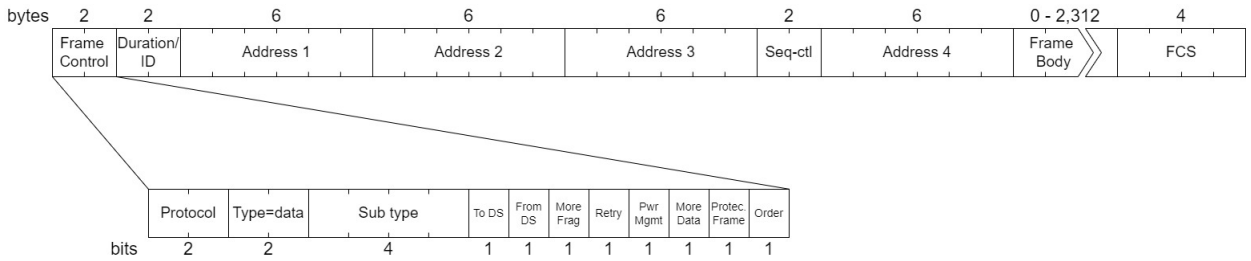

Figure 2.5: The 802.11 frame structure (inspired by Figure 3-10[14])

The first thing that stands out is that there is room for up to four MAC **addresses**. The first and second addresses are used as the destination and source address respectively. To determine to which subnet an AP belongs to, the MAC address of the router interface is saved in address 3. This is used when switching between 802.11 and 802.3 frames. Address 4 is used when bridge mode is utilized.

The **FCS** field is used for error detection of the frame, using the CRC algorithm. On arrival the FCS value is calculated again and when the two values differ, the frame is dropped.

The **Frame Body** field contains all the data that is not part of the header nor FCS (in case of data frames data from the upper layers) and has no fixed size (it does have a maximum size). The **Sequence Control** field is used in both management and data frames for defragmentation and duplicate frame discarding. It can be extended with QoS options. Depending on the type and subtype values, the **Duration/ID** takes on different forms. The value of this field is either the NAV, the Contention-Free Period (CFP), or the Association ID (AID). I.e. it either holds the period of time in which the medium is occupied or the IDs of stations when sending out PS-Poll frames.

Finally there's the **Frame Control** field. This field consists of a lot of subfields of which most of them are used as flags. These may indicate if a frame is retransmitted (Retry flag), if the frame was fragmented (More Fragments flag)... An overview of the Frame Control and its subfields can also be found in Figure 2.5.[14]

Frame Types

There are three types of frames: **Data**, **Control** and **Management** frames. Each of them have multiple subtypes which can be found in Table 2.2.

The first two types, the data and control frames, work close together. The data frames are used to send all the common data while the control frames help facilitate the delivery of these frames. The earlier discussed CTS and RTS are two examples of control frames.

The management frames are part of the services 802.11 provides, i.e. they are used to manage the BSS. The data in the frame body is frequently a combination of fixed fields and variable fields. The former are fields like the reason code and status code field, while the latter carry data such as the SSID, support rates and channels, and the challenge text. The variable are

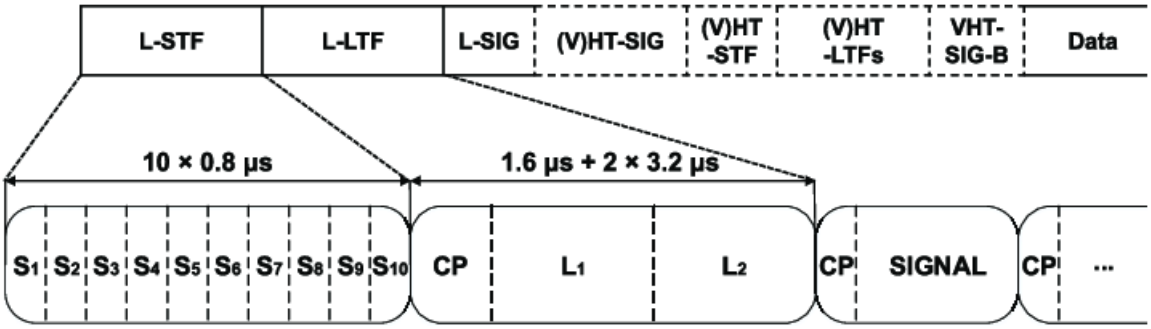

Figure 2.6: Preamble and frame start[17]

also referred to as information elements. Association and Probe requests are example of management frames.[14]

2.7 PHY 802.11

Once the medium is available and a frame is ready to be sent, all left to do is physically send the data over air. How this is achieved is the responsibility of the **Physical Layer** (PHY). Unlike the MAC, 802.11 has multiple PHY standards as seen in Table 2.1. For modulation three technologies are present: **Direct Sequence Spread Spectrum** (DSSS), **Frequency Hopping Spread Spectrum** (FHSS) and **Orthogonal Frequency Division Multiplexing** (OFDM). OFDM is the most popular and is also the one supported by the Openwifi project (see later). From this point the other two will no longer be discussed.

Just like MAC, the PHY consists of two sublayers: the **Physical Layer Convergence Protocol** (PLCP) and the **Physical Medium Dependent** (PMD) sublayer. The purpose of the PLCP sublayer is to encapsulate the data passed down from the MAC sublayer, adding additional data, and finally pass it further down to the PMD sublayer. The PMD in turn will then send the data it received over the air using an antenna.

2.7.1 OFDM PLCP

When the PLCP sublayer receives data from the MAC sublayer in the form of a **MAC Protocol Data Unit** (MPDU) or **PLCP Service Data Unit** (PSDU), it will be encapsulated with a preamble and header before transmission in the PMD. The resulting **PLCP Protocol Data Unit** (PPDU) is shown in Figure 2.7.

2.7.2 PLCP Preamble

In order for the sender and transceiver to synchronise their timers a combination of ten short and two long training sequences are added before the PLCP header. Together with a single guard interval in between, they form the preamble of the PLCP

type name (value)	subtype value	Subtype name
Management frames (00)	0000	Association Request
	0001	Association Response
	0010	Reassociation Request
	0011	Reassociation Response
	0100	Probe Request
	0101	Probe Response
	1000	Beacon
	1001	Announcement Traffic Indication Message
	1010	Disassociation
	1011	Authentication
	1100	Deuthentication
	1101	Action
Control frames (01)	1000	Block Acknowledgement Request
	1001	Block Acknowledgement
	1010	Power Safe-Poll (PS-Poll)
	1011	RTS
	1100	CTS
	1101	Acknowledgement
	1110	Contention-Free-end (CF-End)
	1111	CF-End + CF-ACK
Data frames (10)	0000	Data
	0001	Data + CF-Ack
	0010	Data + CF-Poll
	0011	Data + CF-Ack + CF-Poll
	0100	Null
	0101	CF-Ack
	0110	CF-Poll
	0111	CF-Ack + CF-Poll
	1000	QoS Data
	1001	QoS Data + CF-Ack
	1010	QoS Data + CF-Poll
	1011	QoS Data + CF-Ack + CF-Poll
	1100	QoS Null
	1101	QoS CF-Ack
	1110	QoS CF-Poll
	1111	QoS CF-Ack + CF-Poll

Table 2.2: MAC frame types overview [14]

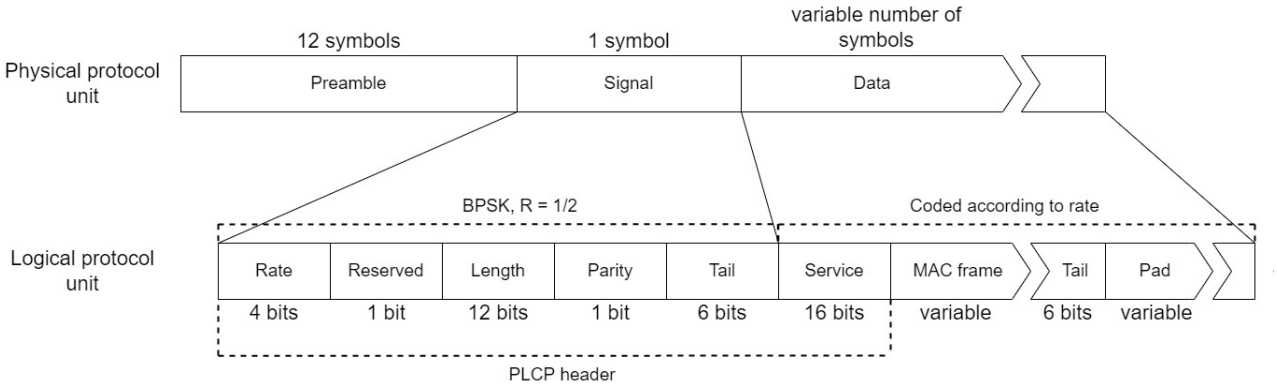

Figure 2.7: OFDM PLCP legacy mode framing format (inspired by Figure 13-14[14])

frame. The short training intervals are used signal lock-on and antenna selection, while the log training intervals are used for channel estimation and fine-tuning timing acquisition.[14, 10]

2.7.3 PLCP Data

The **data** field consists of four parts: the **service** field, **MAC frame**, **tail** and **padding**. After the MAC frame, another **tail** field (consisting of six 0 bits) is inserted with the same purpose as the one in the MAC header: ending the convolutional code. Since OFDM works with fixed-size data blocks, additional padding is added to ensure the data block has the correct length. [14, 10]

2.7.4 PLCP Header: Legacy mode

The PLCP header consists of the signal field and the service field. The legacy signal field, or in other words, the signal field for devices operating in legacy mode (versions < IEEE 802.11n), contains five subfields: the rate, reserved, length, parity and tail fields.

The **rate** field encodes the data rate, in other words the value of this field indicate how the data field in the PPDU should be encoded. To inform the PHY of the size of the MPDU, the number of bytes is saved inside the **length** field. As a protection against possible data corruption a **parity** bit is introduced as part of the signal field, while another bit is **reserved** for future use (and is always set to 0).

The **tail** field consists of six 0 bits and is used to unwind convolutional code. Finally, the **service** field, although part of the PLCP header, is not encoded conform with the other fields of the header but instead encoded just like the MPDU, according to the encoding defined by the rate field. The first seven bits of the service field are used are used to synchronize the descrambler while the remaining bits are reserved for future use.[14, 10]

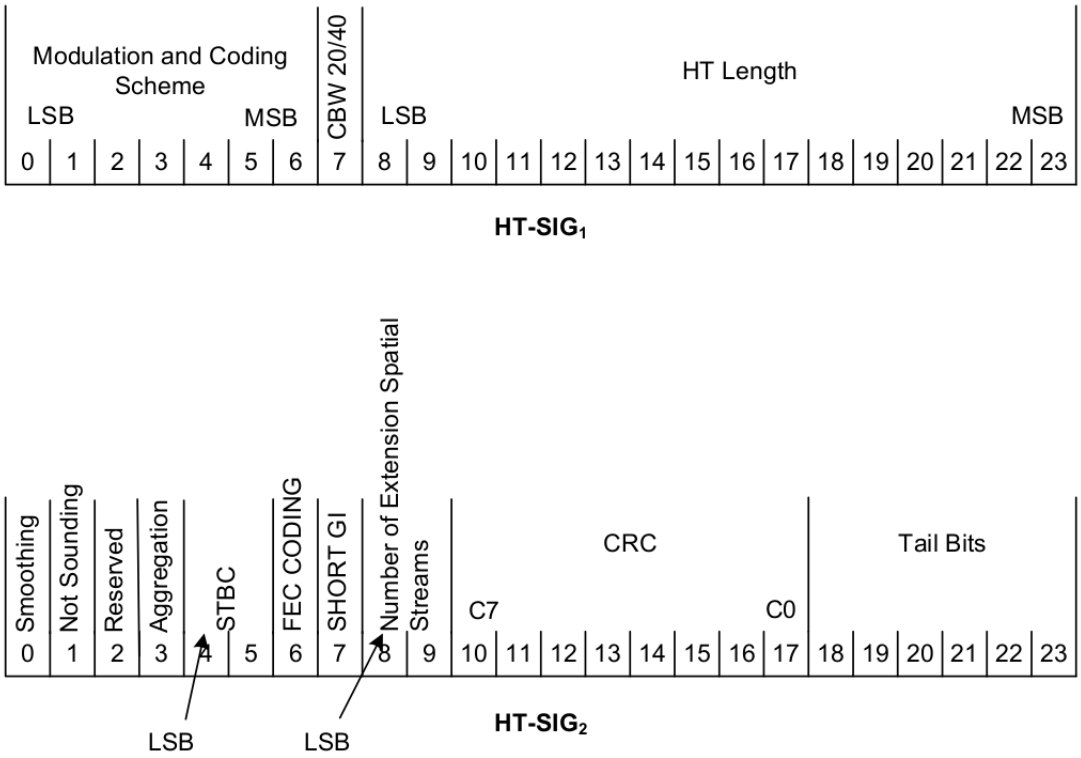

Figure 2.8: OFDM PLCP Greenfield/HT mode framing format [18]

2.7.5 PLCP Header: HT mode

With the arrival of IEEE 802.11n in 2003 and the introduction of MIMO, or Multiple-Input Multiple-Output, devices that supported this standard became capable of something that was called High Throughput. **High Throughput** mode, or also commonly referred to as **Greenfield** mode, allowed for higher bit rates between IEEE 802.11n compliant devices. The PLCP header for Greenfield can be found in Figure 2.8.

The Greenfield signal field is twice as long for the Legacy signal field and also consists of two parts. Some fields have changed, others have been omitted and new ones have been introduced. The length field is a few bits longer, the reserved field is only 1 bit long and and the parity bit has been removed altogether. There are also a lot of new fields, which won't be discussed here. For a more detailed explanation, the reader is referred to the standard[10]. [14]

2.7.6 PLCP Header: Mixed mode

As mentioned before, backwards compatibility was very important for the IEEE 802.11 standard, so that older devices can still interact with more recent ones. Since Greenfield and Legacy mode are not directly compatible, a workaround solution had to be introduced: Mixed mode. Simply put and as the name also suggests, **Mixed** mode combined both the Legacy and HT modes thereby enabling communications between devices supporting different operating modes. The PLCP header for

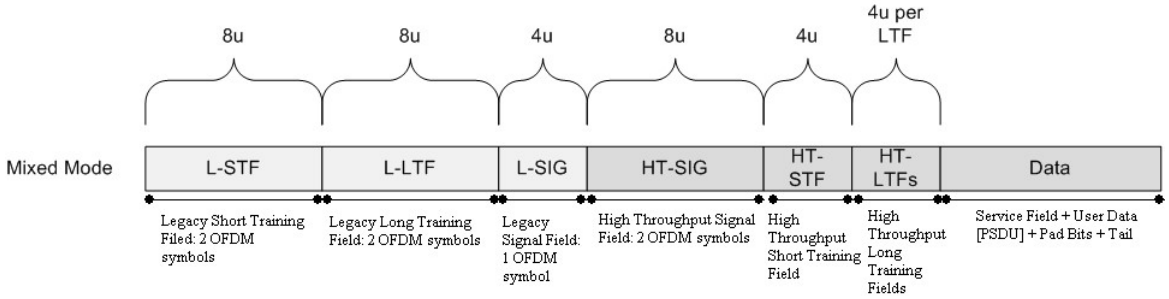

Figure 2.9: OFDM PLCP Mixed mode framing format [19]

Mixed mode can be found in Figure 2.9.

While Mixed mode sounds like a magical solution, it comes at the cost of a significant drop in throughput due to a lot more overhead. It is therefore recommended to use HT mode in most cases and only to switch to Mixed mode when in doubt (about the supported modes on devices in the network). [20]

2.7.7 PLCP Header: VHT and HE mode

With IEEE versions 802.11ac and 802.11ax come the VHT (Very High Throughput) and HE (High Efficiency) modes respectively. Both improve on the throughput of previous versions while still allowing for backwards compatibility. Each version does bring along its own new and improved signal fields, which opens the door for more fuzzing opportunities. Because the Openwifi platform currently does not support either version, these fields were not explored and will not be discussed further.[21, 22]

3

Security and threats

3.1 CIA Triad

The National Institute of Standards and Technology (NIST) defines cybersecurity as follows[23]:

> Prevention of damage to, protection of, and restoration of computers, electronic communications systems, electronic communications services, wire communication, and electronic communication, including information contained therein, to ensure its availability, integrity, authentication, confidentiality, and non-repudiation.

Confidentiality, integrity and availability form what is widely known as the CIA triad, the three fundamental security goals that any system should ensure at any time. Violating one or more of the CIA goals is considered a security breach. Authentication and non-repudiation are two extra goals to improve upon security and are often used in combination with the CIA triad.

Confidentiality

The goal of **confidentiality** is that data is protected from unauthorized parties and that only the people with the correct privileges have access to that data. Cryptography is typically used to ensure confidentiality.[24]

Integrity

Without **integrity**, any malicious party can alter, change or even delete data without the required permissions. The goal of integrity is to guarantee that data was not altered during transit. Hashes are used to detect modifications of the data. Before a message is send, a hash is generated based on the data in the message. If the calculated hash on the receiving end doesn't match the attached hash, the frame is dropped. Note that hashes don't protect against alterations but are used detect them, so that altered data may be ignored/dropped by the receiver. Alterations don't necessarily relate to a cyberattack, it is also

possible that data was changed during transmission, i.e. that accidental bit flips occurred. CRC is used to detect and correct these errors (if possible).[24]

Availability

Availability means that data should be available at all times, or at least when it is expected to (e.g. data might be temporarily unavailable due to server maintenance, but this doesn't violate availability). Denial of Service (DoS) is an attack were the availability goal is thwarted by continuously draining the resources of a target. This can be done by for example flooding the target with network traffic, preventing legitimate traffic from being handled. A Distributed Denial of Service (DDoS) attack, is a variant of the previous attack, were the attacker uses multiple devices to attack a target, compared to a normal DoS attack. This is a lot more difficult for a defender to protect against.[24]

Authentication, Non-repudiation and Access Control

Authentication is a service that verifies an individuals identity before granting access to that individual. **Non-repudiation** on the other hand assures that the sender of a message can't deny sending the message and the receiver can't deny receiving it. In short, it's a proof of delivery. Finally, **access control** or **authorization** grants access to resources based on the identity verified by authentication [23]

3.2 Threats to a LAN

When comparing wireless and wired networks, there's immediately one security weakness that comes to mind that the former has over the latter. With a wireless network, there's no need for a physical connection to communicate with nearby devices. As such, it is possible for someone to eavesdrop on data traffic over air, a weakness inherent to all wireless data transportation. The answer to this problem is strong encryption but this has to be enforced and encryption algorithms might be proven vulnerable later on and be replaced by a safer alternative. [25]

In 2014, Md Waliullah and Diane Gan[26] created a list of available attacks against a wireless LAN. In their article they differentiate between passive and active attacks as well as the security goal they threaten. A summary of their findings can be observed in Table 3.1. This list is nowhere complete and as mentioned by the authors, upper layer attacks were excluded in their work but that doesn't mean a WLAN is not vulnerable to them. For an explanation on these attacks, the reader is referred to their respective work[26].

Security Goal	Type	Attack
Confidentiality	Passive	Traffic Analysis Eavesdropping
	Active	Evil Twin AP
	Active & Passive	Man-In-The-Middle (MITM)
Access Control	Passive	War Driving
	Active	Rogue Access Point MAC Addresses Spoofing Unauthorized Access
Integrity	Active	Session Hijacking Replay Attack 802.11 Frame Injection Attack 802.11 Data /802.11X EAP / 802.11 RADIUS Replay Attack 802.11 Data Deletion Attack
Availability	Active	Denial-of-Service (DoS) Attack Radio frequency (RF) Jamming 802.11 Beacon Flood 802.11 Associate/Authentication Flood 802.11 De-authentication & Disassociation Attack Queensland DoS / Virtual carrier-sense attack Fake SSID Attack EAPOL flood AP theft
Authentication	Active	Dictionary & Brute force Attack Shared Key Guessing PSK Cracking Application Login Theft VPN Login Cracking Domain Login Cracking 802.1X Identity Theft 802.1X LEAP Cracking 802.1X Password

Table 3.1: Overview of attack vectors in wireless networks[26]

3.3 Security in 802.11

When the 802.11 standard was made, there had to be some security solutions for the vulnerabilities that WLANs naturally have. These solution were created as a part of IEEE security protocol standard, the **802.11i standard**. Some of the protocols of the 802.11i standard will be further discussed in this section as will the services they provide and their relevance as of today.

3.3.1 Wired Equivalent Privacy

Before the 802.11i amendment was finished, **Wired Equivalent Privacy**, or WEP, was the first security solution in 802.11. It provided authentication, confidentiality and integrity checking. However, it also contained some serious vulnerabilities like a lack of key management and protection against replay attacks. The keys that are used, for both encryption and authentication, are also small and static. As of today WEP is considered unsafe and should not be used anymore.

3.3.2 WiFi Protected Access

WiFi Protected Access (WPA) served as a temporary solution that addresses the issues that WEP had before being replaced WPA2. Besides authentication, confidentiality and integrity checking, it also includes key management and protection against replay attacks. To handle encryption, WPA utilizes the Temperal Key Access Protocol[27].

Temperal Key Integrity Protocol

TKIP, or **Temperal Key Integrity Protocol** is a protocol of 802.11i that provides data confidentiality and integrity and just like in WEP, the RC4 stream cipher is used. However, for TKIP the initialization vector is twice as long (48 bits) and the keys are not static. TKIP is also able to negate forged messages by using a Message Integrity Code, commonly referred to as *Michael*. While being useful at the time, TKIP is no longer considered secure as demonstrated by the **NOMORE**[28] and **Royal Holloway Attacks**[29].

3.3.3 WiFi Protected Access 2

A year after the release of WPA, the 802.11i group released an improved version named WPA2. WPA2 provides two versions to handle authentication: WPA-Personal and WPA-Enterprise. The main improvements are the following:

- replace TKIP encryption with CCMP
- a longer password is required

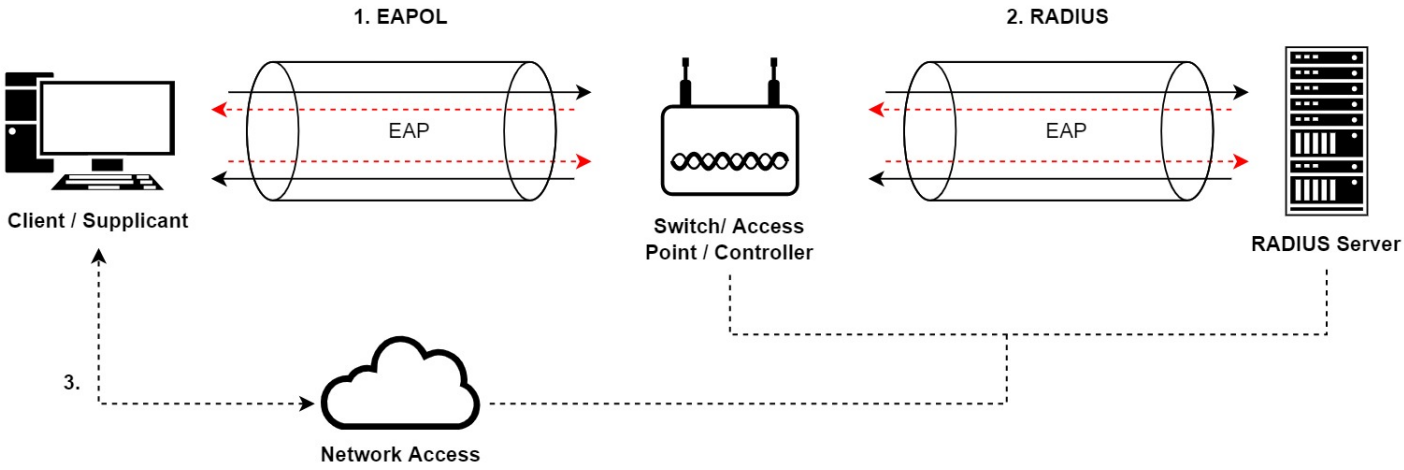

Figure 3.1: IEEE 802.1x components (inspired by [31])

Unfortunately, there are also a few disadvantages that WPA2 compared to WPA. First, it requires extra processing which results in a decrease in the network performance. Another issue is that WPA2 is not compatible with older software[30].

WPA-Personal

In small home networks, authentication is achieved by using a pre-shared key (PSK) mode is utilized. This implies that all users on the same AP, share the same key and is therefore considered less secure than WPA-Enterprise.

WPA-Enterprise

To achieve authentication for enterprise networks, WPA2-Enterprises uses the **802.1X standard**. This is another IEEE 802 standard, that provides **Port-Based Network Access Control** (PNAC) for LANs. This standard makes use of **Extensible Authentication Protocol** (EAP) to handle the authentication requests. The message flow between the STA and AP usually happens with the help of **EAP over LAN**, or EAPOL. The message flow between the AP and Authentication Server (AS) is handled by utilizing the **Remote Authentication Dial In User service** (RADIUS) protocol. Hence these servers are commonly referred to as RADIUS servers. Figure 3.1 shows a WPA-Enterprise setup. [32][31]

CCMP

Counter Mode CBC-MAC Protocol, or **Counter Mode Cipher Block Chaining Message Authentication Code Protocol** in full, is the most common encryption protocol that is used in WPA2. It is based on the Advanced Encryption Standard (AES) cipher and just like TKIP, it provides both **integrity** and **data confidentiality**.

TKIP is still available as a WPA2 encryption protocol for backward compatibility, but as previously mentioned, is now considered obsolete. [33]

3.3.4 WiFi Protected Access 3

WPA3 is the latest generation of the WPA family and is the recommended security protocol for wireless LANs. It improves on its predecessor (WPA2) in multiple ways:

- a longer session key for enterprise networks
- replaces 4-way handshake with SAE

SAE

WPA2 was vulnerable to **key reinstallation attacks** (KRACK), which allowed attackers to reinstall a key that is currently being used. [4] To prevent KRACK attacks WPA3-Personal replaces the 4-way handshake from WPA2-personal with Simultaneous Authentication of Equals (SAE) authentication [34], which is based on the Dragonfly key exchange [35], which ensures forward secrecy. [36]

Although it is an great improvement over WPA2, it is still vulnerable to attacks like the Dragonblood[5] attack in 2019 and the FragAttacks[37] in 2021.

3.4 4-way Handshake

With the arrival of 802.1X came along the 4-way handshake. The goal of this so-called "handshake" was to securely exchange keys between devices so that future traffic between the two could be safely encrypted. Figure 3.2 depicts the different phases before encrypted communication is achieved. The first three phases (discovery, authentication and association) will not be discussed, so the next phase is the 4-way handshake. We'll briefly discuss each step.

In the first step, **M1** the access point will generate a random a send this together with its own MAC address to the station. Such a value is only to be used once and is commonly referred to as a nonce. In the context of cybersecurity, a nonce means a "number-used-once". It is a randomly generated value used to bring entropy to the output of cryptographic algorithms as it should be appended to the clear text before the cryptographic operation. This value will prevent replay attacks and is resistant to message modification. This means that if this value was altered, the handshake will fail and has to be re-initiated afterwards.

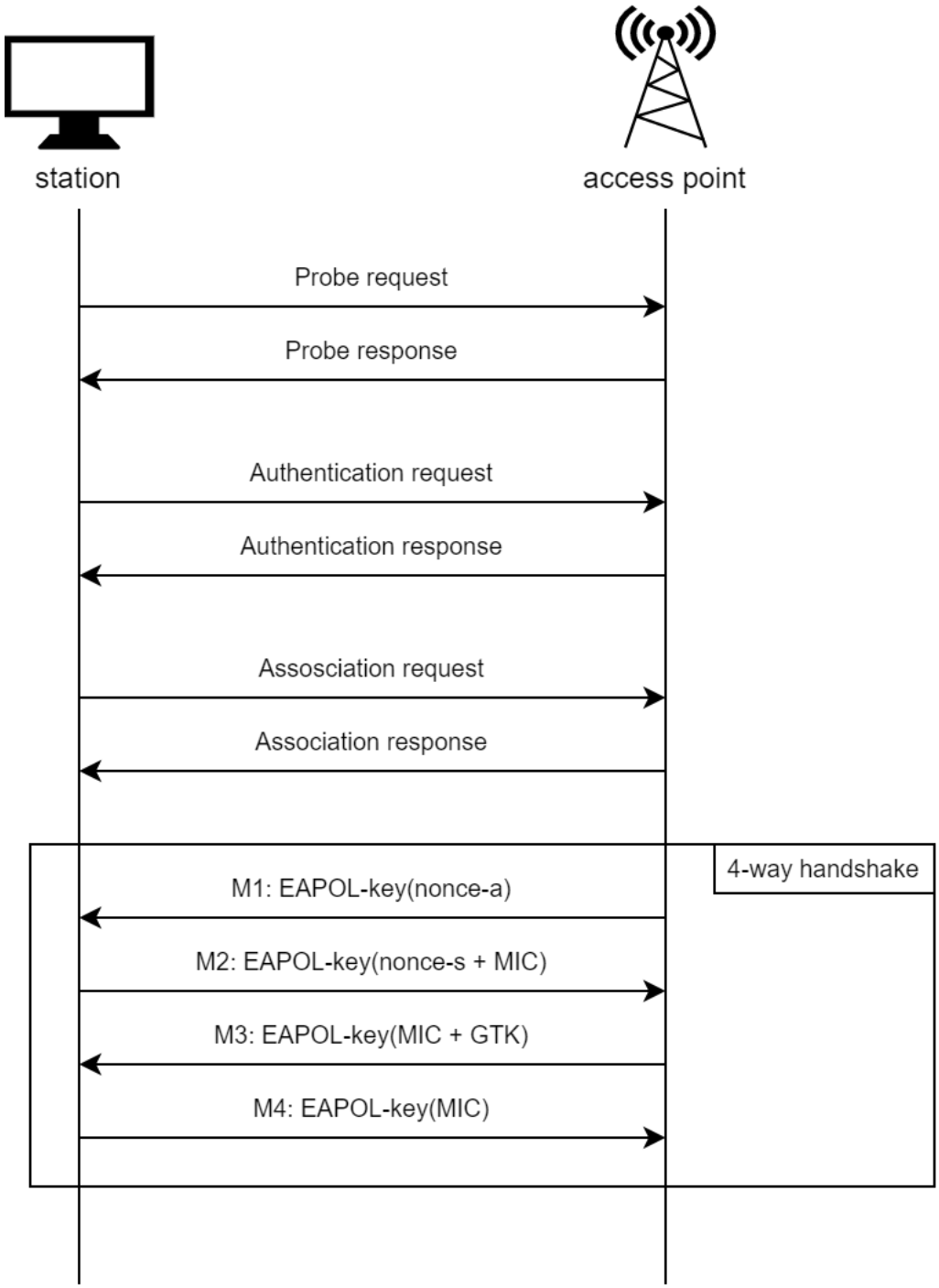

Figure 3.2: IEEE 802.11i Operation Phases (inspired by Figure 7.8 [32]

When the message arrives at the station it will generate its own nonce, allowing to generate the **Pairwise Transient Key** (PTK) using the nonces and the MAC addresses of both. The PTK consists of three parts: the Temporal Key (TK), EAPOL Key Encryption Key (EAPOL-KEK) and the EAPOL Key Confirmation Key (EAPOL-KCK).

Now the station sends it own nonce back to the station (**M2**), while also including the station's MAC address and a **message integrity code** (MIC). At this point, the AP can generate the PTK as well and can verify the MIC using the EAPOL-KCK.

The AP will now send another message to the station, containing the MIC and the Group Transient Key (**M3**). The latter will be used for multicasting between stations. Finally, the station will send back an acknowledgment message **M4** containing the MIC. From this point on, all traffic between the station and access point is encrypted.

4

Fuzzing

4.1 Introduction

Software testing is definitely a research domain on its own. It is a necessary step when developing a new software project and a lot of different strategies are available to use on a system or software under testing (SUT). Fuzzing is one of these techniques and a quite popular one. The idea behind fuzzing is that malformed data packets are sent to a specific program with the intend of crashing it or making it show unexpected behaviour. That program being used to send these customized packets to a system is called a fuzzer. [38]

In Fuzzing there are just like with hacking/testing in general three different environments. The distinction between each environment is based on the knowledge an attacker/tester has about the system. These environments are as follows: white-box, grey-box and black-box fuzzing.

The first environment, the white-box fuzzing is used mostly by the development team of a certain product. For this environment, the attacker has all the knowledge about the software that is being tested. Having an (almost) complete understanding about the system, allows an attacker to send more customized data to the system. If for example a certain format is required and all other formats are dropped at the first check, it is quite useless to test those inputs anyway, unless it is expected that there might be a vulnerability there.

Secondly, black-box fuzzing, is the complete opposite of white-box fuzzing. The system is completely closed off from the attacker, meaning he has no idea what calculation are done on the received information. Therefore a lot more ground has to be covered and the amount of tests required to test all possible combinations can scale up astronomically.

Finally, grey-box fuzzing is somewhere in-between the two previous environments and depending on the knowledge on attacker has it can lean more towards one of the two. With grey-box fuzzing, the attacker has not a complete overview of the software and its inner workings, but rather has certain knowledge about what kind of underlying template a system follows, to make more educated assumptions and better targeted data input.

In case of this thesis, grey-box fuzzing is used. The fuzzer does not take in account what is happening behind the scenes but it knows that it requires WiFi packets. Using the wildly available 802.11 standards, the input can be crafted so that the data

packets fit known Wi-Fi frames and are less likely to be dropped immediately. [39]

Following example can illustrate the difference further: lets take a message consisting of nine bits. In case of a black-box environment, it would take 2^9 fuzzings to test all possible combinations. Now take the grey-box environment, where the attacker knows that the message consists of three fields of three bits each and they are independent of each other. In this situation, there only need to be $2^3 + 2^3 + 2^3$ messages created to test all "valid" combinations. Now for the final situation, you are the developer of the software and you know that the first field can only have the value "001" or "000" and the second field only has four valid values. Now there only need to be $2 + 4 + 2^3$ messages tested. This example clearly shows that the number of tests drastically decreases, the more knowledge the attacker has and therefore the closer the attacker is to a white-box environment.

Testing all possible combinations for an input is very time-consuming but it isn't necessarily a bad technique. Sometimes the software developer doesn't take in account edge cases which are covered when all possibilities are tested. Another case could be that because of backwards compatibility some functionality isn't maintained properly and what used to be spare bits in a previous versions, now are actively used. In case a message is received that utilizes these (previously) spare bits, and the program doesn't account for them, crashes might occur because of an invalid value.

4.2 Openwifi

4.2.1 Introduction

Simply put, a **Software Defined Radio** (SDR) is a radio technology where physical layer functionality is implemented in software rather than completely integrated in the hardware for traditional radios. This allows for more flexibility for developers to tweak system settings and trying out new configurations, and allows for faster updates and upgrades. Updates in software are, compared to hardware updates, faster and generally more cost-efficient. [40]

The **Openwifi**[41] project is a SDR based 802.11 implementation for Linux. As of the time of writing this thesis, the project supports 802.11 versions a, g and n. The complete open-source project can be found on github[42].

Openwifi consists of three main physical components: the ARM processor, FPGA and RF frontend. The FPGA allows to change the PHY properties of packet before transmission and therefore it is possible to fuzz physical layer. In code, the import parts for the fuzzing are the driver file and packet injection interface. The injection interface resides in the user space and will transfer the data concerning the physical layer, to the driver file in kernel space.

Handling the PHY data in the driver and implementing the necessary changes in the FPGA is done by a fellow student, Jasper Devreker. You can find his work here[43]. The code changes he made to the driver that can be used in the future to join his work with this one, can be found on his Github[44].

4.2.2 Previous Work in Openwifi Fuzzing

In 2021, Steven Heijse, a student at the university of Ghent, did research on fuzzing of the physical layer[9]. The final result was a proof of concept on how to fuzz the physical layer. The fuzzing was achieved by generating a driver file for every possible combination of the signal field in the PLCP header. This approach, however produced a lot of overhead, especially in memory. A more dynamic approach will be required to improve on the weaknesses of this design. Another problem is that the fuzzing is limited to what FPGA allows and is capable. If these limitations are being violated, the FPGA itself might crash. Differentiating between the hardware parameters and the on-air information might solve this issue.

Another student, Stef Pletinck, worked on combining the Openwifi platform with an existing Wi-Fi fuzzer so-called Greyhound which was originally developed by Keysight. By the end of his work, he successfully managed to combine the two and was able to fuzz packets and capture them on the targeted device. While his work doesn't directly affect this thesis, there is a short section dedicated to the possibility of combining the Physical layer fuzzing, with other existing (higher layer) fuzzers.

4.3 Existing Fuzzers

Fuzzers come in all shapes and sizes and discussing them all would be a thesis on itself. Fuzzers like OSS-Fuzz [45], Google's open source fuzzing solution for open source software, or BeStorm [46], a closed source fuzzer for both hardware and software, are both interesting fuzzing products, but the former focuses more on fuzzing software rather than protocols, and the latter is closed source and are therefore not very suitable for this topic. In this section, a few fuzzers will be discussed that focus on fuzzing the IEEE 802.11 standard and are accessible for further examination.

4.3.1 Wifuzzit

One of the first results when googling "wifi fuzzer", is Wifuzzit[7]. This open source Wi-Fi fuzzer is available on Github and is able to fuzz both stations and access points. Unfortunately, the main branch hasn't been updated for 9 years, so this solution can be considered deprecated.

4.3.2 Owfuzz

Another option is the Owfuzz fuzzer[6]. This more recent project is build on top of the Openwifi project, which is already more on the right path, since this is the platform were the fuzzing for this thesis will happen. It uses the osdep library (part of aircrack-ng)[47] for frame injection.

4.3.3 Greyhound

Greyhound [8] is a Keysight product which uses a state machine to keep track of the current state to monitor state transfers (as described by the standard) and checks whether the behaviour of the targeted device is irregular or not. The fuzzer makes use of the Scapy library to send packets to kernel space.

4.3.4 Evaluation

While the latter two are valid options, neither does implement PHY layer fuzzing. For this reason a bottom-up approach will be utilized: first a working solution for injecting packets with fuzzed PHY information will be created, before combining existing fuzzers with this solution.

This thesis will start from Steven's idea on how to pass fuzzed signal field information from the driver to the FPGA for transmission and work up to the user space, where these existing fuzzers reside. For the user space injection, this work will be build on top of the injection script that is already implemented in Openwifi.

5

Environment Setup

In this chapter the components of physical setup will be briefly discussed as well as the necessary configurations that were made. The configurations were required to allow each device to communicate with the other ones. In Figure 5.1 the complete hardware setup can be seen, with the IP addresses for each interface and connections between the devices. Open connections (lines with only one end connected to a device) can be used to access the device over air from any device inside the network (if the user of requesting device holds the login credentials for the targeted device) and if port-forwarding is enabled, even from outside the network.

Some software that was used to configure the setup or was required for the setup to function properly will also briefly discussed in the appropriate section.

5.1 Hardware

5.1.1 Openwifi Board

The Xilinx Zync-7000 ZCU102 board[48], with a mounted AD-FMCOMMS2-EBZ board[49] as RF front end, was used to run the Openwifi image. This device functioned as the fuzzer or attacker in the setup. To configure the board, the reader is referred to the Openwifi Github documentation[42]. The board communicated wireless with the test device for fuzzing, and used a wired connection (via the control device) to access the DUT for lifechecks.

The real-world setup can be found in Figure 5.2. Note the smaller board with the antennas on top of the larger one. This smaller board is the before mentioned RF front end that will be used to transmit the fuzzed packets over air to the Raspberry Pi to the right of the board.

Besides the setup instructions in the Github documentation, there were some additional changes that had to be done for the setup to work correctly. To ensure the ping requests and other network traffic used the correct interface, some routing rules needed to be introduced. To ensure correct routes:

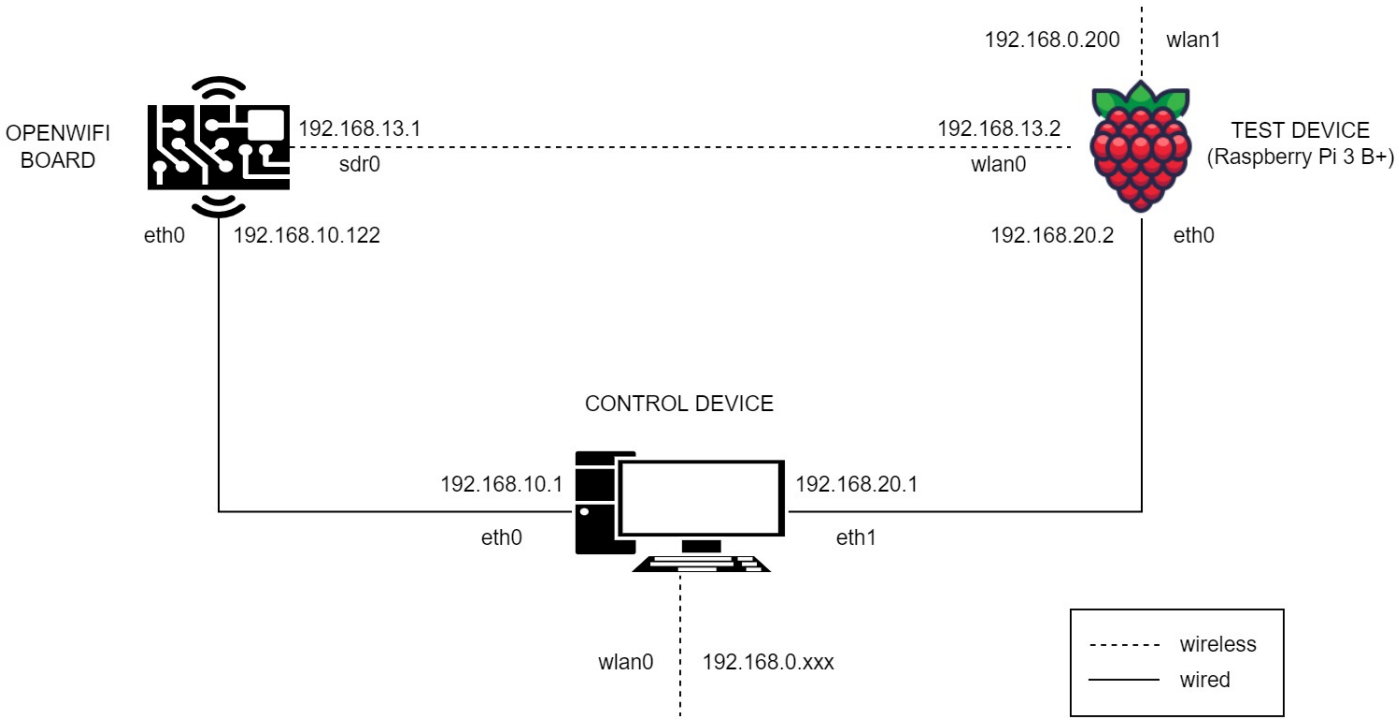

Figure 5.1: Hardware Setup

1. All traffic for subnet 192.168.13.0/24 had to go through **sdr0**
2. All other traffic should go through **eth0**

Note that the first line is irrelevant when the sdr0 interface is set to monitor mode. It was also important to set both antennas (on the SDR) as for apart as possible, i.e. creating a wide angle between the two to reduce interference between the two. Sometimes it was necessary to fiddle around a bit with the antennas when the signal wasn't optimal.

5.1.2 Device Under Test

For the test device, or DUT, a Raspberry Pi 3 B+ (running Raspian OS)[50] was used. Before the device could be properly used, static IP addresses, routes, SSH and VNC (optional) had to be configured. Because the integrated wireless network card did not support monitor mode, an additional eSynic USB adapter was used during development.

There are a few different ways to setup static glsip addresses but for the Raspberry Pi it was done by changing the **/etc/dhcpdcd.conf** file. To add static routes, the file **/lib/dhcpcd/dhcpcd-hooks/40-route** was created. The changes for both files can be found in Appendix B.

However for the **wlan0** interface (the interface that was going to be tested) to work in ad-hoc mode, an additional shell script was written that disables the dhcpcd service and sets the correct settings for the wlan0 interfaces while also adding

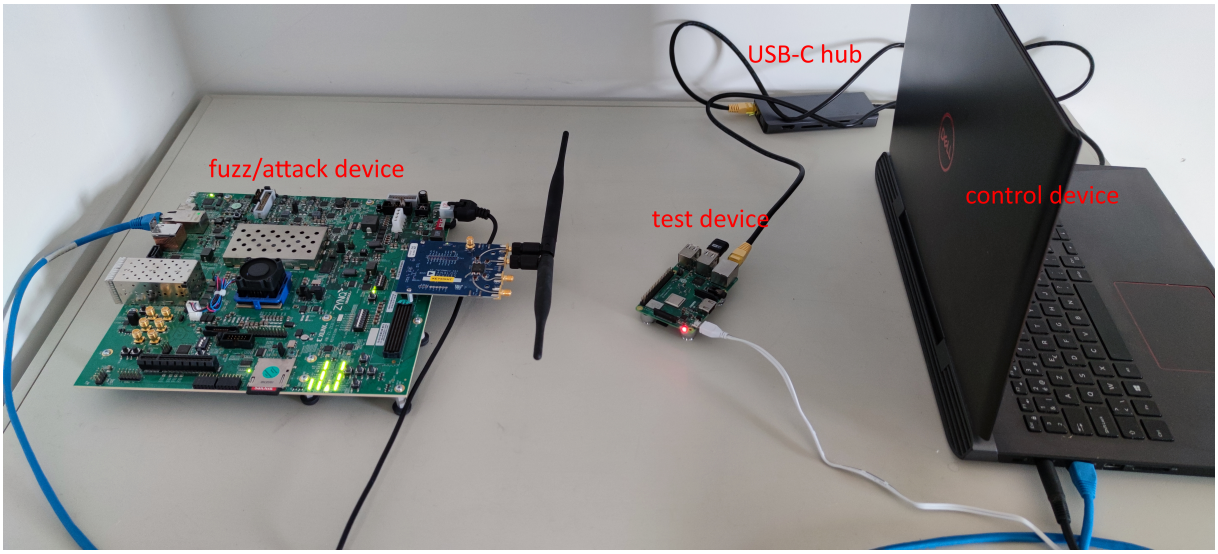

Figure 5.2: Real-World Hardware Setup

the necessary routing rules so that the other two interfaces (eth0 and wlan1) still function properly. This script can also be found in Appendix B.

5.1.3 Control Device

The control device was a personal Dell Inspiron laptop which had originally Windows 10 as operating system but was later reformed to an Ubuntu 18.04 computer. This laptop was used to setup connection to both the board and the DUT. Since the laptop had only one Ethernet port, an USB-c hub (with an additional Ethernet port) was used. This way, all devices were connected over a wired medium and could also contact each other over air.

The wired connection was required to check the operational state of the wireless interfaces of both the board and the test device, since both could fail during testing (as mentioned in the previous chapter). The configuration for both the wired and wireless connections can be found in Appendix C. Besides an extra connection between the board and test device, both links were also used to access both devices. The required network configurations (IP addresses and paths) can be found on Figure 5.1.

5.1.4 Work Device

A Lenovo Legion (Windows 10) laptop was used as the primary working station. While not part of the test setup itself, this device was used the most during the thesis. It could access devices within the test setup using remote desktop and SSH connections (see later).

5.2 Software

5.2.1 Wireshark

In order to capture the injected packets on the SUT, Wireshark will be used. **Wireshark** is considered the de facto standard for packet analysis. It uses PCAP, an API for capturing network traffic, to retrieve packets visible on a selected network interface. There are a few libraries that implement PCAP. For this thesis libpcap[51] was used, which is the PCAP library for Unix systems. On Windows machines, the WinPcap/Npcap[52, 53] libraries are used. While Wireshark is a graphical user interface (GUI) program, there is also a CLI version called **Tshark**.

5.2.2 SSH

Secure Shell or **SSH** allows for secure connections between two devices. This protocol was of vital importance to control devices within the setup. When starting a new SSH connection, the user will usually be prompted to enter a password before being able to access the targeted device. This can be quite cumbersome when starting a new SSH connection inside a script. There are multiple solutions to get around this obstacle.

The first solution would be to utilise **sshpass**. This tool allows a user to pass a password directly (in plain-text), via a file or file descriptor or by using an environment variable, to an ssh command. While the first one is not very secure (password can be retrieved from the code), the other options are a viable alternative.

Another option and the option that was used here is to use SSH keys to login instead. A user can generate a public and private SSH key by invoking the **ssh-keygen** command. Afterwards, the public key can be copied to the target device using the **ssh-copy-id** command. When this succeeds, the current device will no longer be prompted for a password when trying to connect to the target device over SSH.

5.2.3 Remote Desktop

While it isn't required for the setup, the **remote desktop** can be used to facilitate working with the setup devices by providing a GUI instead of the usual CLI (using ssh). During the thesis, both **RDP** and **VNC** were used. The former was used to work remotely on the control device while the latter was used to communicate with the test device. Using the Wireshark GUI instead of the Tshark CLI was a bit more friendly on the eyes.

Another and even more important reason to use remote desktop was that it allowed for working on a bureau in one room while the whole setup (including the test device) could be stationed in another room. This also allowed for working and testing remotely (outside the home network) by introducing Port Forwarding rules.

6

PHY Fuzzing Architecture

6.1 Difficulties PHY Fuzzing

6.1.1 Access Fields for Fuzzing

The first step in any fuzzing process is to create or alter existing packets and ensure the changes persist throughout the whole process (ensuring they're not overwritten later on). However, altering packets might not be so straightforward. Network devices can be split into two groups: hardMAC and softMAC. The difference between the two is that the former implements the MAC layer in hardware while the latter does it in software. Therefore, hardMAC devices makes it near impossible to alter MAC, and definitely PHY, information. With softMAC it is possible to mutate the MAC properties but not necessarily PHY level information. This is where the Openwifi platform comes in handy.

6.1.2 Avoid Breaking Implementation on Fuzzer Device

Not only does the fuzzed information need to be persisted during the whole process, it is also of critical importance that the packets don't crash the transmitting device. If, for example an illegal packet length is registered somewhere in the packets to perhaps exploit a buffer overflow on the receiving device, the tester needs to ensure he won't introduce a buffer overflow on his own device by confusing the networking stack when transmitting the packet. This is another advantage of the Openwifi platform, due to the accessibility of the full transmission stack with inclusion of a FPGA to handle the transmission.

However, this doesn't mean the FPGA knows how to handle it, at least not initially. This is why another student is researching how to change the FPGA, so that hardware parameters and the on-air information can be separated and that it doesn't break when passing illegal values for the on-air signal field.

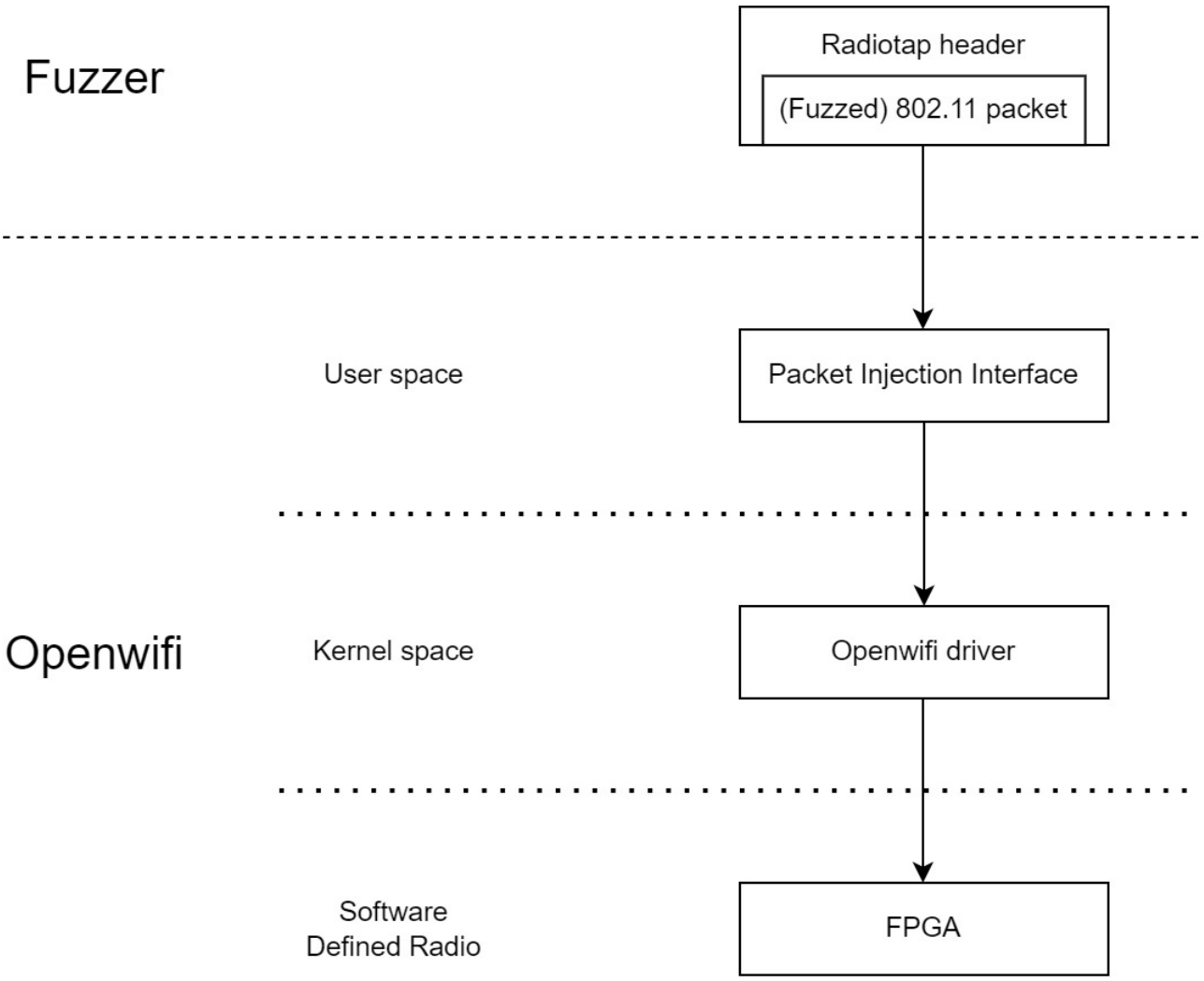

Figure 6.1: PHY Fuzzing Architecture overview

6.2 Software Architecture

In this section, the software architecture for the fuzzing of the physical layer in 802.11 will be discussed, as will the individual components.

6.2.1 mac80211

The **mac80211** subsystem is part of the Linux wireless networking stack and provides an API for drivers to (as the name suggests) implement the MAC layer functionalities for wireless communication. It is used to implement softMAC drivers, meaning that the MAC layer is implemented in code rather than hardware. The Openwifi driver is thus a softMAC driver.

The API will be called from the upper layers when performing the packet injection, which in turn will be handled by the function implementations in the SDR driver.

6.2.2 SDR driver

One of the key components of the Openwifi platform is the **SDR driver** that serves as the glue between upper lying mac80211 subsystem and the underlying FPGA implementation. It provides handles for the functions in the *mac80211_ops* struct. This struct is used to describe the available mac80211 functions and offering developers the freedom of implementing the MAC-level functionalities themselves.

Implementations for the mac80211 API are not the only functions defined within the driver. One of these functions (that is not part of the **mac80211_ops** struct), the **calc_phy_header** function, calculates the PHY header which will be later used to pass data PHY data to the FPGA for further processing. This function is called from the *openwifi_tx* function which implements the *tx* function of the API.

6.3 Packet Injection in Openwifi

In this section, some concepts are discussed that are necessary for the transmission of fuzzed packets.

6.3.1 Radiotap

Radiotap[54] is a header format that provides additional information on top of 802.11 frames but is not part of the IEEE 802.11 standard. Information like the channel, data rate and flags can be attached to the frames. An optional field is the the Time Synchronization Function Timer (TSFT) field. This field consists of 64 bits which means it is large enough to contain all the data for the signal field (as discussed in section 2.7.4). Therefore this field will be used to provide the frames with additional fuzzed PHY data.

In order for fuzzers to utilize the PHY fuzzing in the Openwifi platform, the 802.11 frames will need to be encapsulated within a Radiotap header and the data for the signal field will have to be inserted into the TSFT field.

6.3.2 PCAP

PCAP, or packet capture, is an interface for capturing network packets. There are different implementations: Windows users will utilize the *Npcap* library while macOS and Unix-like systems will utilize the *libcap* library[51]. Since Openwifi is a Raspian computer, it uses the latter. For this thesis the libcap library was utilized to pass on the crafted packets to underlying components (mac80211, driver, FPGA).

6.3.3 Scapy

Scapy[55] is a popular open-source packet manipulation program written in Python. It has a wide range of functionalities (sniffing, replay attacks...) with the most prominent being manipulating packets and decoding packets. It utilizes PCAP for reading and transmitting packets.

6.3.4 Wireless Operation Mode

Wireless interfaces have multiple modes in which they can operate. For mac802.11 the following modes are supported[56]:

- ad-hoc mode
- managed station infrastructure mode
- AP mode
- monitor mode
- WDS mode
- mesh point mode

Currently, Openwifi supports the first four modes.

In order to use packet injection in Openwifi, the device needs to be set to monitor mode. In monitor mode, all packages, within operational range from the device, are captured, whether they were destined for that device or not. Therefore it is commonly used in (ethical) hacking for sniffing on a target network in hopes of finding vulnerabilities and capture data. Monitor mode is used primarily during the development phase of the fuzzing architecture, to verify the scripts.

It is also possible to use packet injection in ad-hoc mode. In ad-hoc mode two devices are connected over a wireless medium without an AP in-between them. This can mode be used for performing MAC layer injections, but was not unusable for injecting at the Physical layer (see later).

6.4 Fuzzing Process

6.4.1 Injecting the Signal Field

As discussed, the libcap library can be used to inject IEEE 802.11 frames in monitor mode. In the Openwifi project (located in the openwifi/user_space/inject_80211 directory) there is already a script for injecting frames, which was used as a basis for

the PHY fuzzing. a static Radiotap and IEEE 802.11 header were already present, and send to the logical mac80211 interface. The 9th up till the 17th (index 8 to 16) byte made up the TSFT field, which we will use to pass the information from the signal field to the driver.

First Improvement: Add Dynamic Injection

Manually changing this field and recompiling is very time-consuming and not very efficient. A first improvement was to add an additional flag which allowed for overwriting the static value in the struct with the provided argument. To add some more flexibility, this argument can be both decimal or hexadecimal. While this is already some improvement, this still only allows for one fuzzing iteration of the signal field, or at least one value for that field (the same value can be sent more than once).

Second Improvement: Add Fuzzing Tools

To account for this weakness, an additional script was made to automate the fuzzing to some extend. At first a bash script was created, which was later on translated to a C script. The script provided the following features:

1. configure fuzzing type
2. configure number of fuzzings/iterations
3. configure number of repetitions
4. configure the jump size
5. configure sleep time
6. configure initial signal field value

There are two options for the type: random and incremental. The first one will randomly generate a value for the signal field in each iteration within the boundaries of the signal field. This last part means that the value cannot be larger than 3 bytes (or 0xFFFFFF), since there's no point in trying to fuzz a value that falls outside the reserved space for the signal field. (e.g. 0x1'FFFFFF' has the same value as 0x56'FFFFFF' for the signal field since only the six least significant hexadecimal values are used).

The incremental fuzzing starts with the initial value of the signal field, default this value is 0x000000 or 0, but a different value can be provided with an argument. After every iteration, the current value will be incremented with the jump size. The default value is 1 but this can also be configured with an argument. In case the new value would go outside the boundaries of the signal field, the loop is halted and the script will exit normally.

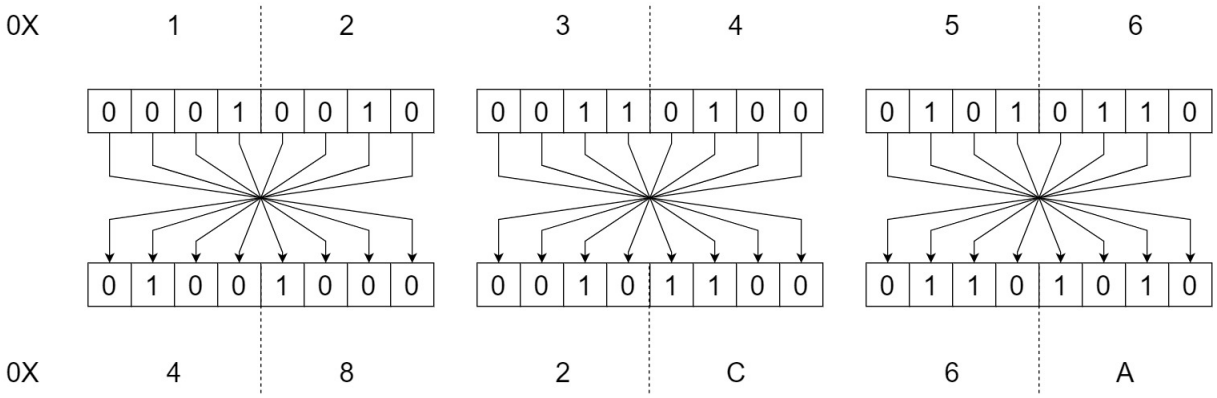

Figure 6.2: Conversion of signal field from MSB to LSB representation

The number of fuzzings or iterations indicates how many times a new value for the signal field will be generated (either random or by increasing the proceeding value with the jump size), while the number of repetitions indicates how many times a single value is injected. E.g. the number of fuzzings is 10 and the number of iterations is 5 which means (at most) 10 unique signal field values will be created and each of them will be injected 5 times resulting in at most 50 signal field injections.

Additionally, the time between each iteration can be configured as well but this feature was not really utilized.

Third Improvement: Add Auxiliary Tools

While the second improvement was a great step in the right direction, it didn't take in account a major part of the transmission of the signal field: when transmitting the signal field, the bytes are in reverse bit order, or *Least Significant Bit (LSB) first* order. In contrast, we are used to read numbers in Most Significant Bit (MSB) order. To illustrate, lets consider the binary representation of the number ten in both MSB and LSB. These are **1010** and **0101** respectively. Note that the byte order remains unchanged: if the MSB representation of the signal field is 0x123456, then the LSB representation is equal to 0x482C6A . Figure 6.2 shows the conversion between the two.

Not only has this an influence on the representation itself, increasing the value of the signal field might not have the desired effect. E.g. the rate field consists of the 4 most significant bits in the MSB representation, while the four most significant bits are occupied by the first three bits of the length field and the reserved field in the LSB notation. If the rate field is 8 (or 1000 in binary) and the first three bits of the length field and the reserved field are zero, then the first byte (in LSB notation) would look as follows: 0x01 (in hexadecimal), 1 (in decimal) and 0000 0001 (in binary). If we would like to increment the rate by one, so that it results in a value of 9 (in decimal), then we need to pay attention to the notation it is in. In MSB notation, this poses no trouble, just increment the rate by 0x10 (increasing by 1 or 0x01 would change third bit of the length field). However, if we would do the same with the LSB notation, this becomes harder: not only is the rate located in the four least significant bits of the first byte, the bit order is reversed. So to increase the rate value by one, we would have to increase the bit representation by 8 (or 1000 in binary).

The strategy is as follows: to keep the fuzzing intuitive, the operation on the signal field are performed on the MSB repre-

sentation, but before injecting the fuzzed value, the signal field is transformed to its LSB form. Auxiliary functions, such as the transformation of bit order, are provided in a different header file.

Fourth Improvement: Joining Scripts

Since the fuzzing and injection happened in two different scripts; the former had to invoke the latter in some way. This could be done by creating child processes and performing a single unique fuzz in each one by calling the injection script with the **execl/execv** function. The creation of child processes is necessary to not interrupt the fuzzing loop since **execl/execv** overwrites its current process. Calling it from inside the loop would cause the loop, and by extension the program, to stop after one iteration. Since the management of child processes can be a hassle and proved to be failing after a certain amount of iterations, the choice was made to simply merge both scripts and extend the injection script with the necessary flags and parameters.

Fifth Improvement: Add support for Partial MAC Header Fuzzing

The first four bytes in the MAC header are used by the Frame Control (FC) and Duration field, each occupying 2 bytes. When using this field for fuzzing, it is important to note that these fields have a lot of influence on how each packets will be interpreted. Changing the **ToDs** and **FromDS** bits, will have a drastic affect on how the**mac** addresses are conceived. E.g. if the ToDS bit is 0 and the FromDS is one, the first MAC address will be considered the BSSID of the AP, while if the former is 1 and the latter is 0, this address will be seen as the Destination Address (DA).

An operational mode was added to facilitate the management of the MAC address by configuring these addresses beforehand and placing them correctly according to the operational mode. However, combining this with the MAC fuzzing should be done with care since these bits can be overwritten, with can lead to unintended results.

Sixth Improvement: Add Lifechecks

It is important to be able to tell when a device is malfunctioning when performing these kind of tests. Since a network driver has two paths, transmission and reception, the decision was made to distinguish between both since one might fail but perhaps the other might fail as well.

To trigger all paths for both the attack and test device a ping request would be ideally. This works for ad-hoc, but unfortunately not for monitor mode (see later). After the ping is send out and returned with either a success or failed response, the path counters should be compared with their previous values.

For the script to work the ping command, as well as the commands for requesting the counters, should be tested and configured in the header file beforehand. This is because the test device might have a different IP address from this setup and the **ifconfig** command might have a different output format as well.

Seventh Improvement: Add Logging

This part is quite straightforward: a logging file is created with a name consisting of the current date and time, as well as randomly generated post-fix. After every iteration, a line is added containing the fuzzed signal field, fuzzed MAC field, result of validation and ping result.

Eight Improvement: Add support for Greenfield/HT

While IEEE 802.11a and IEEE 802.11g use the legacy mode signal field, the latest (supported) version of Openwifi, namely IEEE 802.11n, uses Greenfield/HT signal fields. To support this mode some modifications were done to move away from static field sizes and to look into variable length, depending on the selected mode in the script parameters. Since the Greenfield signal field spans six bytes, that still leaves two unused bytes. These last two will be used to tell the driver which kind of PHY fuzzing it's dealing with.

6.4.2 Locating the Signal Field

One of the most important data structure (if not the most) and by far the most important for this thesis in the Linux wireless networking stack, is the **sk_buff** or **socket buffer**. This structure is quite extensive so we will not discuss it in its whole but rather the necessary features and components that we needed to understand and manipulate the struct in order to access the injected data.

The struct is divided in at least one and at most three regions. At the border of each region, one or more pointers (depending on the number of regions is located. These four pointers are the head, data, tail and end pointer. Initially, when allocating the buffer, the head, data and tail pointer are pointing to the start of the buffer, and the end pointer (unsurprisingly) to the very end. The head and end pointer will stay at their initial location for the entire life cycle of the buffer.

Many functions and fields have been added to assist the developer in minimizing programming mistakes when altering the struct. A few of these functions handle alterations to the pointers. The first interesting function is the **reserve** function. This function moves the data and tail pointer further away from the head pointer by a provided number of bytes. The created space between the head and data pointer is what is revered to as the headroom, this will be useful in a second. Note that the end pointer is left untouched. Next up are the two most important functions: the pull and push instruction. These two manipulate the headroom so that data can be added or removed from the front of the packet (the packet is located between the data and tail pointer). Finally the put and trim are the cousins of the previous two functions but they alter the tail pointer and the tailroom instead.

Besides these functions there are some fields that can help a developer quickly located fields within the buffer. A couple of these fields hold certain length values: the length between the head and data pointer (or in other words, the size of the headroom), the length of the MAC header, the length of the IP header...

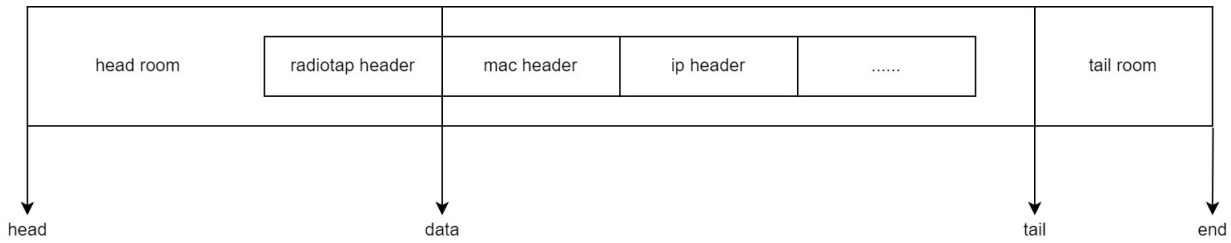

Figure 6.3: Location Radiotap Header in sk_buff Struct

Now for the part that concerns the location of signal field data: when injecting packets in monitor mode, at one point the **ieee80211_monitor_start_xmit** function is called. This function start organizing the socket buffer and the headers so that the Radiotap header is added before the others. Once the headers are properly allocated, the function calls the **ieee80211_parse_tx_radiotap**.

As the name suggests, this function processes all the "interesting" data in the Radiotap header and passed to the **ieee80211_tx_info** struct before removing the Radiotap header by invoking the **skb_pull** function. Unfortunately, the TSFT is not processed and cannot be called upon in the SDR driver later on. This would be an annoying issue to fix. One solution would be adding the TSFT field processing but this has multiple disadvantages. The first one being that the Linux wireless networking stack is used in Openwifi by overwriting some of the operations in the **mac80211_ops** but the mac80211 layer itself remains untouched (at least by the Openwifi project). The second problem would be finding a suitable location to keep the signal field data. If no suitable location can be found, alterations to one or struct would have to be made, resulting in breaking structs defined by the standard.

Fortunately, there is an alternative solution which is a bit more unorthodox since it deviates from the way the socket buffer is supposed to be used. Rather then actually clearing or overwriting the Radiotap header, the pull operation just moves data pointer behind the Radiotap header (and at the start of the MAC header). So the data is still there and could be lost when the **skb_push** function would be called later on. Luckily it doesn't and by the time the socket buffer arrives at the driver function where the signal field is passed to the FPGA the data is still there. Figure 6.3 captures the layout of the **sk_buff** struct when it arrives at the **openwifi_tx** function.

6.4.3 Passing the Signal Field

In the driver file, we will use the **openwifi_tx** function for passing on the signal field data. Since this function uses the **skb_push** function, the signal field had to be extracted as soon as possible.

To access the signal field we first need to know the location of the data pointer. There are multiple ways to do this: one way would be utilizing fields in the socket buffer. We can use the **mac_header** field for this, as we know this header is right behind the Radiotap header. Another way would be subtracting the head pointer from the data pointer. The **skb_headroom** function that is part of the *skbuff.h* header file does just that. We will use the latter to retrieve the number of bytes in the headroom.

Now that we know where the Radiotap header ends we can also find the start by subtracting the size of the Radiotap header from the size of the headroom of the buffer. Since we also know where the TSFT field is located, we can use this offset to directly access the signal field data.

To differentiate between the types of signal fields, or whether signal field injection should even be performed, the last two bytes of the TSFT field will be used. For legacy signal field injection, the last two bytes both need to be equal to **0xAA**, for Greenfield, they should be equal to **0xBB**. All other values indicate that there will be no injection, meaning no PHY fuzzing will occur.

As mentioned earlier, the passing of the signal field data to the FPGA will be done by another student who's writing his thesis in parallel. Therefore how the FPGA handles these bytes will not be discussed here but can be found in his thesis. However, in order for us to cooperate, a certain interface had to be decided so that both our works can interact with each other. For this, the initial length for the PHY header was extended with 8 bytes (originally 16 bytes) in the **sdr.h** file and some code changes were made to the **calc_phy_header** (in the **sdr.c** file) function by the other student.

While it would be possible to change the function signature by adding another parameter, and injecting the bytes at the correct location, the choice was made to leave the function as is and rather inject the bytes after the function call.

6.5 Fuzzing in newer version

During the thesis a newer version of the Openwifi platform was released. While improving the overall platform, it also brought some inconveniences along. The **calc_phy_header** function no longer existed and the variable holding the length of the header was commented out. Since this removed the interface that was in place, the previous approach had to be reassessed. The header and by extension the signal field, were now calculated in the FPGA.

Since the Linux wireless networking stack functions were left untouched, the fuzzing and injections scripts still function. Locating the signal field can also be done in the same way as before. The necessary changes only need to be done in the driver, in the **openwifi_tx** function. To change the rate and length, the following variables had to be changed: **rate_hw_value** and **len_psdu**.

However, even though we can access them, we can't inject incorrect information because the FPGA checks the provided information and drops packets that do not adhere to the prescribed rules. The problems don't stop there: the other three fields (reserved, parity and tail) are all hard-coded in the FPGA. In order to fuzz these they need to be accessible from the driver.

Thus it is no longer possible to fuzz the full signal field, at least not with the current version. In future releases, it should be possible to implement the necessary changes in the FPGA to enable fuzzing for the two supported fields and add access for the missing three fields as well. These changes are outside the scope of this thesis and could be explored in further research.

6.6 API for Third Party Fuzzers

It would be preferable if this PHY fuzzing could be combined with existing MAC layer Wi-Fi fuzzers, such as Greyhound. To make this work a standard format or API should be defined which these fuzzers can implement.

The biggest drawback in the current version of the PHY fuzzing process is the use of a fixed size Radiotap header. The Radiotap standard states that in case the TSFT field, will always start from the 9th byte and end at the 17th byte, at least if the field is present. Unfortunately, we don't know the start of the header, only where it ends. We use the known length of 28 bytes to figure out the location of the signal field data but when we work with a variable length header, this is no longer possible.

However, despite this drawback it is possible to combine existing fuzzers with the current version as long as they follow the format or make changes in the driver to set a new size for the Radiotap header. To illustrate with an existing fuzzer: the Greyhound fuzzer utilizes Scapy to send packets to the driver. This is similar to how libcap injects packets (since Scapy uses pcap injection underneath). In a previous work[57] a student already managed to connect the Greyhound fuzzer to the Openwifi platform to fuzz higher layers packets.

While the previous work was not replicated here (the fuzzer was not directly connected to the board), a small part was reused. The first step was to install Scapy on the board. Afterwards, a test script written in Python by the previous student was used to test if Scapy could be used in the same way as the libcap library (that has been used up till now). The script had to be slightly altered to work in the same way as the custom fuzzer (the one using libcap): the interface name was changed so that it points to the **sdr0** interface (the interface of the SDR/RF-frontend) and the Radiotap header was changed so that it's conform to the format for the PHY layer fuzzing (28 bytes). This code fragment can be found in Appendix E.

6.7 Validating Injection and Transmission of Fuzzed Packets

Wireshark is a very useful tool to verify if, when, and how packets are send and received. Since the lowest layer is handled in hardware, the PHY information is already stripped before it is captured with pcap and displayed in Wireshark. This is another opportunity to use Radiotap, since it will be provided with PHY level data on the receiving side and can therefore provide each captured packet in Wireshark with the corresponding Physical layer information.

An alternative way to verify the fuzzing and reception, is to use kernel level log functions in functions that are know to be called in the transmission process. The reception can be verified by Wireshark in case the receiving device doesn't encounter any issues, but for details on the reception and processing at the PHY layer, kernel level logs should again be used. When testing hardMAC or commercial products in general, this might not be possible, as we have no access to the MAC layer and by extension the PHY information.

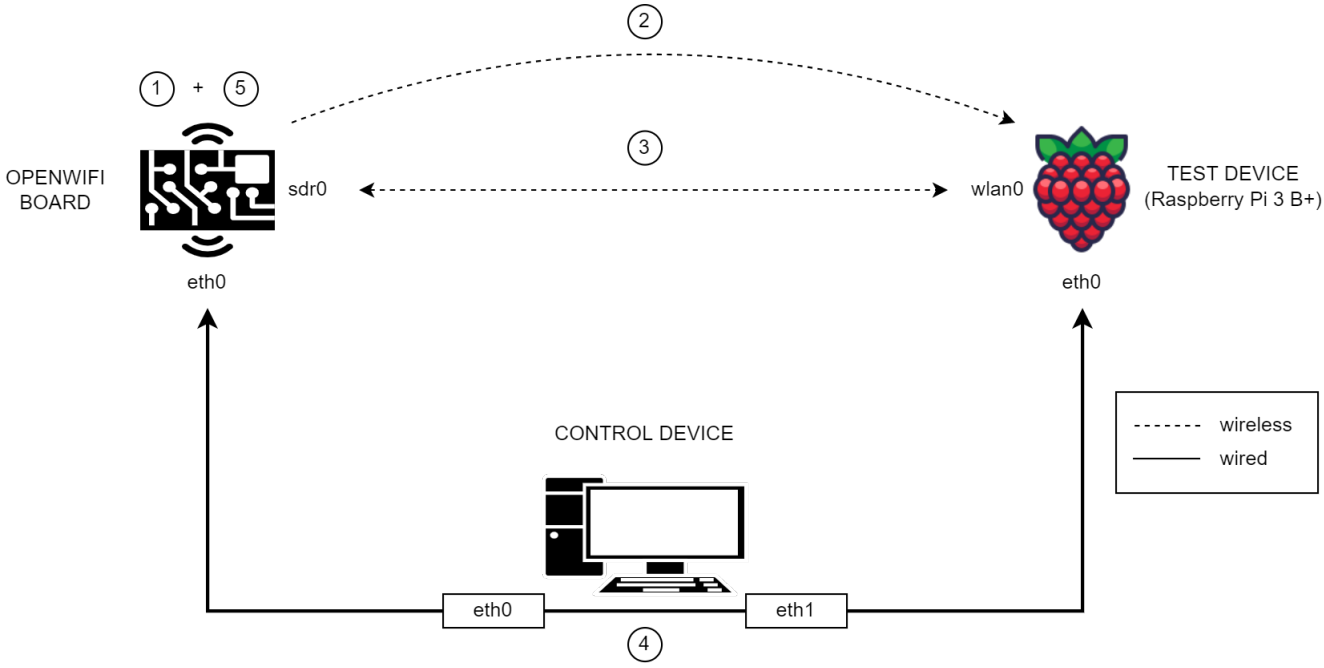

Figure 6.4: Fuzzing Steps in Ad-Hoc mode

6.8 Fuzzing Steps

In this section, a high-level explanation will be given on the functionality and steps in the fuzzing script. C and Bash code will not be discussed here. For the complete code base, the reader is referred to the openwifi fork for this thesis[58].

6.8.1 Ideal Scenario (Ad-Hoc mode)

Before going into this section, it should be stated that this is a preferred scenario, but it wasn't possible to fuzz the physical layer this way. In the next section, the steps that do work for the PHY fuzzing, will be discussed. Fuzzing the higher layers however, is possible this way. Figure 6.4 gives an overview of the steps for ad-hoc mode (the ideal scenario).

While the general idea and the separate steps will be discussed, the code itself will not be delved into but can be found in Appendix D. Before the fuzzing script can be called, the configuration as described in chapter 5 should be verified. This can be done quickly by sending ping requests through both the wired and wireless path.

The first step is to setup the ad-hoc mode for both the board and the test device to have a clean communication without outside communication. This was already touched upon earlier. Sending Ping requests and verifying the configurations of both with the **iwconfig** command should not be forgotten.

Before the script will be called some parameters should be set in the **phy_fuzzer.h** file. If the tested interface has another IP address as the one in Figure 5.1 then this should be changed. Since the output of the **ifconfig** might be different for

the test device, the TX and RX commands should be tested beforehand and changed so that the desired output is achieved. Additionally, the locations were the log file will be stored can be changed here as well. The MAC addresses for BSSID, receiver and sender should be set as well (the last one is optional). The configuration settings for the commands in the header can be found in Appendix F.

The script can be called with a whole lot of parameters and discussing them all individually would be time-consuming and pointless. Instead a table with a brief description of each can be found in Appendix A (Table A.1).

The first part of the script will validate the parameters, create the log file and opening a packet injection handler. After everything is set up correctly the fuzzing loop will start.

This brings us to the first step of the actual fuzzing (step 1 in figure). In this step the fuzzed packet will be composed. It's up to the user to chose to fuzz the signal field, a part of the mac header or both. It is also possible to define by which value the signal field and mac header increase. This can be interesting since for example the tail field in the signal field: if the user doesn't want to try 2^6 combinations every time, this field can be skipped by setting the jump value to 0x40 or 64 (1 000 000 in binary). The script accepts both decimal and hexadecimal values for the jump values of both the signal field and mac header and for the initial value of both as well.

Depending on the frame type (date, management or control) and the passed operation mode (for this test plan, it will be ad-hoc) the predefined MAC addresses will be swapped around.

When the packet is crafted, it will be send out over air to the targeted device (step 2). There it will be handled by the test device which may or may not behave unexpectedly (accept, drop, crash...).

In step 3, to test the operational state of both the board and test device, a ping request is sent the targeted device. This will test both the TX and RX paths on either devices. This is because both paths work separately and while one might have crashed, the other might still be up and running.

After the ping is done (succeeded or failed), the TX and RX counters on both devices will be retrieved and compared to the counters since the last injection (retrieval of the counters is also done before the first injection). If the counters are all higher than the previously acquired ones, all paths are still up and running, if not, a crash might have occurred. However, it is possible that the packet simply was dropped somewhere (the ping reply might not have been noticed by the board).

Finally, in step 5, the result of step 4, as well as the values for the signal field and mac header, will be written to the log file.

The signal field and mac header are now incremented by their respective jump counter before starting over from step 1. If one of the two fields has reached its max value, or the number of iterations is reached, the loop will exit peacefully instead, as will the program.

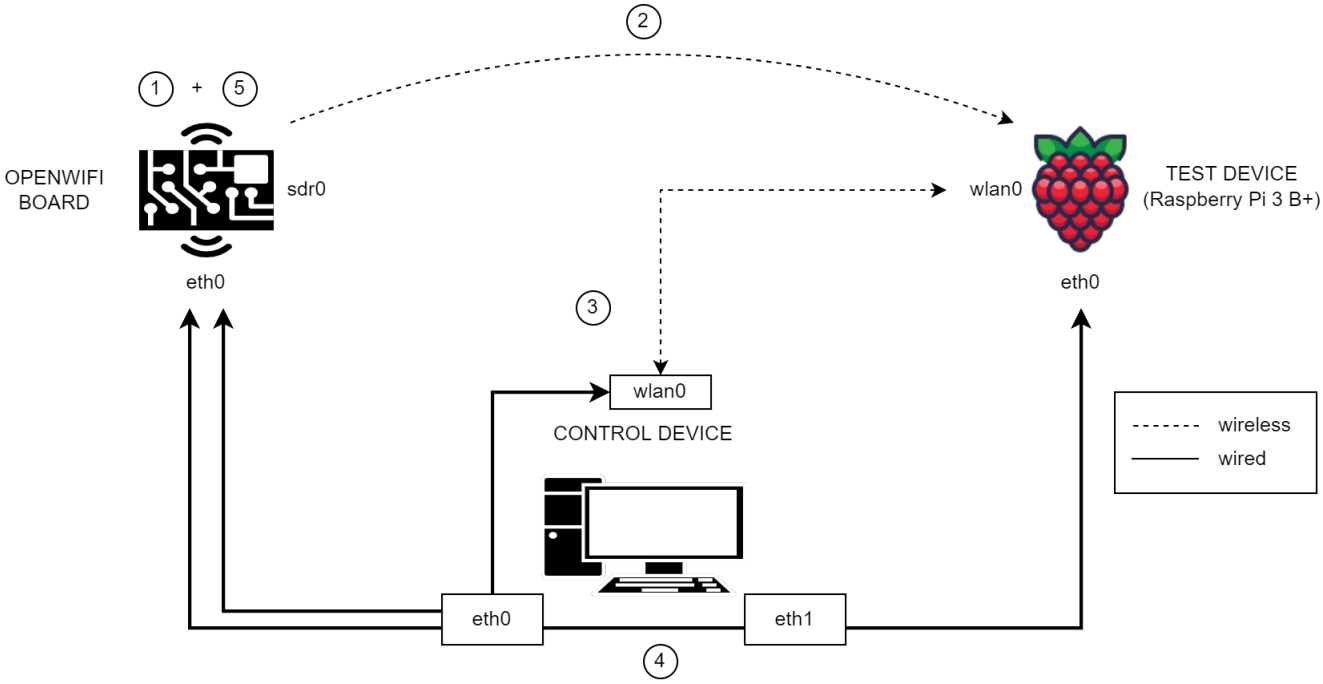

Figure 6.5: Fuzzing Steps in Monitor mode

6.8.2 Scenario for PHY Fuzzing (monitor mode)

When testing out the signal field injection in ad-hoc mode, it became clear something wasn't right. The first assumption was that perhaps the headroom space was different than the one in monitor mode. This resulted in the wrong bytes being injected. A first attempt to solve this was by changing the thus far hard coded offset for the TSFT field, to a relative offset based on the location of data pointer, and by extension the start of the MAC header.

However, this didn't fix the issue. By studying the whole headroom and comparing it to the injected packet it became clear that the method for injecting the signal field via the Radiotap header was no longer valid. The Radiotap header was overwritten somewhere in the Linux Wireless Stack and the data was no longer accessible. The only way was to stick to the monitor mode to fuzz the signal field.

This does change the test steps and configuration to some extend. The integrated WNIC (wlan0) on the test device, will connect to the same AP as the wireless NIC of the control device, while the sdr0 interface will be set to monitor. Since the USB network adapter (on the Raspberry Pi) supports monitor mode, it can used to easily monitor the traffic originating from the board.

Since a NIC in monitor mode no longer has an IP address, sending a ping via the sdr0 interface is no longer an option. So step 3 should either be removed, or another way to trigger the paths should be used. The solution was to use the wireless interface of the control device to ping the test device instead. The ping command will be remotely executed on the control device, using an ssh connection over the wired connection between the board and the control device. To verify if the ping succeeded, the exit code is send back over the same ssh connection to the board. The other steps remain the same.

When working with monitor mode, the TX counter shown by the ifconfig command on the board is no longer used and will always remain on 0. To solve this issue, the approach that was used by Steven[9] will be used here as well. A new command was defined in the header that also accounts for monitor mode by looking at the **/proc/interrupts** directory and gets the counter for the sdr0 there.

This approach does neglect the RX path of the board, so the check for this path is irrelevant for the tests in this mode. This isn't necessarily bad, since this path is the least important path by far as well as the least likely to fail. Since the device also operates in monitor mode, it is also quite likely it will detect packets from nearby devices anyway it doesn't necessarily need a ping reply to increase the RX counter. In fact it is almost guaranteed that one of the packets, which is part of the ping between the control and test device, is captured and thereby increasing the RX counter on the board. The modified plan can be found in Figure 6.5.

7

Results

When everything is set up the fuzzing script can be run with the desired parameters to start testing. Figure 7.1 shows an example where all the possible values for the rate will be tested without changing the other fields. This is achieved by carefully choosing the jump value.

7.1 Legacy Signal Field Results

While there were not a lot of fuzzing tests performed to search for vulnerabilities, there were tests performed to verify their transmission and whether they were visible to the target device by running monitor mode on the Raspberry Pi. In Table 7.1, the findings are summarised. Due to time constraints, the amount of tests was minimal: not all values for each field were tested, and each field was tested separately. An example of captured traffic via Wireshark can be found in Figure 7.2.

For each test, a payload of 64 bytes was used. For a normal operation, the value of the signal field (in LSB first notation) was **0x0b0b00**. This was used as the baseline for the tests.

7.1.1 Data Rates

The data rates could all be transmitted by the board, except for the data rate with value 0. The board would fail to transmit all packets from the moment 0 was used for the data rate field. The Raspberry Pi wasn't able to detect all packets. The packets with rates 6, 9, 12, 18 were always captured, while other rates frequently failed to be detected. Illegal rates were not detected at all.

7.1.2 Reserved Bit

Changing the value of the reserved bit did not seem to have any influence on the transmission or on the reception of the packets.

```
root@analog:~/openwifi# ./inject_80211/phy_fuzzer -m g -n 16 -v 2 -c 0x080b00 -j 0x100000 -o -p -s 64 sdr0
TX number of packets board = 165
RX number of packets board = 1300
TX number of packets target = 161866
RX number of packets target = 106904
Provided signal field in little endian. Reversing...
Following settings will be used:

OPTION                  |       Value
------------------------|-----------------
nr. of injections       |       16
IEEE version            |       802.11g
operation mode          |       AP
rate index              |       0
SHORT GI                |       0
delay                   |       100000 usec
MAC HDR fuzzing         |       no
PHY fuzzing             |       yes
target IP address       |       192.168.0.202
encap. hdr              |       DLT_IEEE802_11_RADIO

Following fuzzing parameters are used:

OPTION                  |       Value
------------------------|-----------------
fuzzing mode            |       incremental
signal field type       |       legacy
init signal field       |       0x10d000
PHY jump value          |       0x100000

Packets have the following (initial) properties:

OPTION                  |       Value
------------------------|-----------------
type                    |       d
subtype                 |       1
length payload          |       68
length ieee hdr         |       24

LOGFILE: /root/logs/20220414-1852-VIXhcvyw.txt


---------------------------------------------

                ITERATION 1 (2 repetition(s))

---------------------------------------------

BUFFER:
0x00 0x00 0x1c 0x00 0x6f 0x08 0x08 0x00
0x08 0x0b 0x00 0xaa 0xaa 0xaa 0xaa 0xaa
0x00 0x0c 0x71 0x09 0xc0 0x00 0xde 0x00
0x01 0x00 0x00 0x0f 0x18 0x02 0x00 0x00
0xb8 0x27 0xeb 0xfe 0xc5 0x39 0x30 0xd3
0x2d 0xae 0x0a 0x3f 0x66 0x55 0x44 0x33
0x22 0x22 0x10 0x86 0xd9 0x12 0x8f 0x3f
0xfd 0xfd 0x06 0x89 0x3f 0xba 0x1d 0xf9
0x41 0x7c 0xe3 0x37 0xcf 0x6d 0x18 0xf5
0xde 0x23 0xbc 0xac 0x0e 0x87 0xb1 0x39
0xb7 0xc1 0xce 0x92 0xd3 0x5e 0xd1 0xd2
0x5c 0xd8 0x5c 0x9c 0x94 0x7a 0x96 0xd5
0xf7 0x7b 0x0e 0xc7 0xe8 0x26 0xbd 0xc7
0x4a 0x7b 0x75 0x58 0x04 0x27 0x92 0xbc
0xe8 0x61 0x4f 0xbd 0xc0 0x22 0x91 0x1e

VALIDATION RESULT = 0x00

VALIDATION RESULT = 0x00


---------------------------------------------

                ITERATION 2 (2 repetition(s))

---------------------------------------------
```

Figure 7.1: Script Usage Example

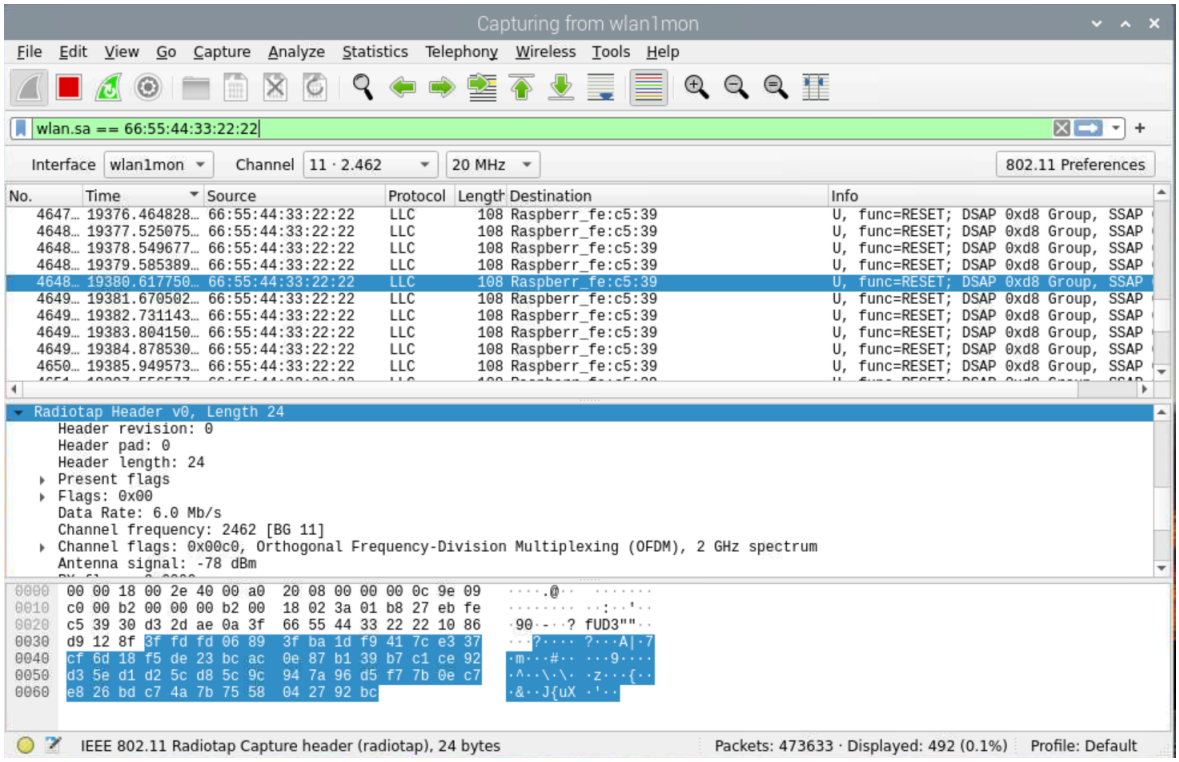

Figure 7.2: Packet Capture Example

7.1.3 Length field

Due to the size of the length field, only a small number of values were tested: very large values, very small values and values close to the valid value. Changing the length field didn't change anything in most tests but in rare cases it could break the TX path preventing all future packets from being sent. The value 0 was one of them, as well as the values 409 (199 in hexadecimal, 110011001 in binary) and 410 (19A in hexadecimal, 110011010 in binary).

7.1.4 Parity Bit

An incorrect parity bit did not influence the transmission of the packets, however the Raspberry Pi would only receive the packets with a valid parity bit set.

7.1.5 Tail Bits

Just like the reserved bits messing with the tail bits did not have any impact on the transmission of the frames. However, a lot of packet loss occurred which made it difficult to test. To ensure the reception each unique packet was sent 10 times and some that still failed were resent afterward.

field	packets are transmitted	packets are received
rate	yes, except for value 0	for rate 6, 9, 12 and 18
reserved	yes	yes
length	most packets	yes
parity	yes	only valid parity bits
tail	yes	yes, but a lot of packet loss

Table 7.1: Legacy Signal Field Summary

7.2 HT Signal Field Results

While some tests were done with the HT signal field, these were just to verify it's functionality, with the focus on transmission, not reception. Changing the MCS, flags, tail and CRC fields did not cause any problems with the transmission of data. However, just like with the legacy signal field, messing with the length field can result in the transmission path to stop working.

Similar to the legacy mode test, a working Greenfield signal field as a baseline for testing each individual field. For the base value, the HT signal field **0x000600203200** (hexadecimal representation) was used. For every test (except for the CRC tests) a valid CRC was generated.

7.3 Issues During Tests

During tests, an issue arose after a while: **sdr,sdr openwifi_tx: WARNING ieee80211_stop_queue prio 0 queue 0 no room flag 0 hw queue len 0000003d request 14 wr 61 rd 0** (some values might vary). This issue resulted in the FPGA halting the packet transmissions of packets. A lot of time went into finding the issue (recreating the Openwifi image from scratch and adding the changes in an iterative fashion) but the cause could not be determined. Rebooting the board temporarily solved the issue. Waiting enough time between injections also seem to work.

The expected cause is that the TX ring buffer can't handle the amount of packets being injected and/or that the same was true for the FPGA DMA (Direct Memory Access) module.

7.4 Signal Field Dissector

When Wireshark analyses packets, it uses predefined structures called dissectors to give meaning to the data. Even when data is encrypted, e.g. when analyzing all traffic in monitor mode it can still retrieve a lot of information through these dissectors.

When analysing logs and studying the injected hexadecimal values (bytes, it can be useful to have some kind of tool to

```
root@analog:~/openwifi# ./inject_80211/signal_field_dissector -f 0x0b0b00 -r
name:           dec     hex             description
-----------------------------------------------------
rate:           13      0x0d            6 Mbps
reserved:       0       0x00            legal
length:         416     0x01a0          packet is 416 bytes long
parity:         0       0x00            legal
tail:           0       0x00            legal

--------------------------------------
The provided signal field is valid
--------------------------------------
root@analog:~/openwifi# ./inject_80211/signal_field_dissector -f 0x123405 -r
name:           dec     hex             description
-----------------------------------------------------
rate:           4       0x04            illegal (least significant bit is always 1)
reserved:       1       0x01            illegal (must be zero bit)
length:         89      0x0059          packet is 89 bytes long
parity:         0       0x00            illegal (first 18 bit contain uneven '1' bits)
tail:           32      0x20            illegal (must be all zero bits)

--------------------------------------
The provided signal field is NOT valid
--------------------------------------
root@analog:~/openwifi#
```

Figure 7.3: Signal Field Dissector Examples

convert these hexadecimal representations to a more comprehensive format, i.e. translate them the values they encode. To help understand the meaning of these bytes a dissector script is written that can decode both the normal and reversed (LSB first) format and can tell which individual fields are valid and which are not. Currently this script can only decode legacy signal fields. The code can be found in Appendix D.

Two examples for the dissector script can be found in Figure 7.3. the first one is the default value for the signal field when the injection script is called without a value for the signal field, and is thus a legal signal field. The second value is intentionally an invalid signal field to show the output for an illegal value. Note the **"-r"** option, this tells the script that the provided signal field is in reverse bit order. The output translates these values to their MSB first representation.

8

Conclusion

8.1 Improvements over Previous Work

As discussed in Section 4.2.2, there were some limitations with fuzzing the signal field by generating driver files for every combination. This approach definitely has its merits over the previous one.

The first significant improvement is to eliminate the need for generating driver files and thus getting rid of the computational and memory overhead. This approach also allows HT signal field injection, via fuzzing, and switching easily between legacy, Greenfield or no PHY fuzzing. By providing user space injection capabilities, this approach also introduces more flexibility by giving third-party fuzzers the liberty to perform PHY fuzzing as well as long as they adhere to the defined format.

This dynamic approach also keeps track of all paths (TX and RX) which might reveal issues that influence one or both paths. If there is more traffic (in case of a non-ad-hoc environment) the validation of all four paths might be more precise. In ad-hoc mode, the traffic is more limited and if for example the ping fails to be send out, all other paths might fail due to no other packets increasing the packet counters.

8.2 Disadvantages of Current Approach

As mentioned earlier, it is not possible to inject PHY information from user space when the NIC operates in ad-hoc mode due to the use of different methods while going down the Wireless Networking stack. Making changes to the standard to fix this problem is highly discouraged due to the extremely high chance of breaking the Networking stack and obstructing the transmission of packets.

While this approach works currently in monitor mode, there is no way of telling how the standard might evolve in the future and due to small changes, this approach might fail later on. In this way, it can't be seen as a durable option.

Another thing to take in account is the addition of newer signal fields (HT, mixed mode, VHT...) which keeps increasing in size. Predefined fields in the Radiotap header might not cut it to store all the necessary data of these newer signal fields. A

solution could be to add custom Radiotap header fields. However, this might simply just cascade the problem to the sk_buff struct. If the headroom is not large enough, it cause a kernel panic. Increasing the total size of the headroom when allocating a new sk_buff struct, or reallocating the headroom when problems occur. This however, might not be so straightforward as it sounds and might again result in making changes to the standard, which again, is highly discouraged.

8.3 Joint work

One of the objectives for this was combining it with another thesis[43] that focused on the FPGA side. The main idea was that the hardware parameters can be left up to the driver, but that the on-air bytes can be changed from user space. This ensures that the FPGA will not fail due to illegal settings, while still allowing to sent incorrect signal fields.

Unfortunately, the FPGA changes were not finished by the end of this work so there is currently no way of proving that it possible to separate the on-air information and the hardware parameters. For now, this part of the work remains a concept that has yet to be proven functional.

8.4 Ethical and social reflection

When doing research, it is import to take in account how this particular research might reflect in the Sustainable Development Goals (or SDGs) as defined by the UN. These goals were put into place in 2015, with hopes to push all members of the UN towards a better world centered on peace and prosperity. The current target date for these goals is by the year 2030.

However, this thesis cannot be linked directly to any of the 17 goals. The closest goal to this work, would be goal 9 (industry, innovation and infrastructure). The main objective for this goal is as follows:

> Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

The keyword connecting this goal to the thesis would be 'resilient'. Detecting and getting rid of vulnerabilities increases the resilience of any IT infrastructure be it for a company or for an individual. This more resilient infrastructure provides better protection attacks and thereby lowering the chances of damages, costs and, in case of critical safety systems such as nuclear plants, more safety and security.

Since we live in this very connected (digital) world, now more than ever, cyber security can be found almost everywhere. So while this thesis cannot be directly connected to the other goals, it has an influence on a multiple goals. One of the objectives of goal 11 (sustainable cities and communities) is to:

> provide access to safe, affordable, accessible and sustainable transport systems for all, improving road safety, notably by expanding public transport, with special attention to the needs of those in vulnerable situations,

women, children, persons with disabilities and older persons

With transport being more computerized, providing the necessary safety is of the utmost importance. E.g. if hackers were able to control self-driving cars the consequences could be catastrophic.

Goal 3, which focuses on good health and well-being, is another goal that is affected by cyber security in some way. Compromising medical equipment definitely impacts this goal in a negative, and as such, securing these devices is a necessity that can't be overlooked.

Another one could be the quality education (goal 4). The COVID-19 pandemic forced schools to find alternative ways to educate children, by organizing online classes. For these classes, platforms such as Teams and Zoom were used. If hackers were to bring down these platforms, the education would be greatly obstructed.

Thus it can be concluded that this work will probably not affect the SDG goals directly, but rather in a few indirect and perhaps unexpected ways.

8.5 Future Work

A first improvement would be the addition of mixed mode. This could achieved by using another Radiotap field that isn't processed in the TX path. While the current version of Openwifi only supports up to IEEE 802.11n (Wi-Fi 4), support for IEEE 802.11ax (Wi-Fi 6) is under development. With this addition, the opportunity to fuzz the HE (High Efficiency) signal fields might present itself.

Currently, the injection via the Radiotap header uses the last two bytes of the TSFT field to indicate which kind of signal field should be used. When the fuzzed PHY information doesn't have to change for a long time, a static approach can be used, similar to Steven's work. However, instead of rewriting the driver file, a few constants could be defined in the header file that depending on the type (last two bytes TSFT), can be called upon from the driver. This way it is also possible to provide support for mixed mode.

Another interesting direction would be to use software simulated devices instead of real ones. This is research that Keysight is currently doing. The idea is that the simulated device should behave in a similar way as the real device. This technique might allow to quickly test a wide range of devices without the need to have them all in the lab.

The way the signal field is injected is somewhat unorthodox and actually violates the way that the sk_buff struct is meant to be used. Finding another (less unorthodox and more elegant) way to dynamically fuzz the signal field is also an interesting topic to delve further into. Linux has a kernel interface called Netlink which allows for communication between kernel and user space. Perhaps this might prove to be a more elegant and durable option.

As mentioned before, Openwifi currently has a newer version than the one used in this thesis. While the new one obstructs the current way of doing the injection, it also opens the door to new and perhaps better ways to handle the changes in the

signal field.

Combining this PHY fuzzing with existing fuzzers (such as Greyhound) was very briefly discussed, but should work if the third-party fuzzer follows the correct Radiotap header format together with Scapy or Libpcap (other injection tools have not been tested yet). Combining this approach with existing fuzzers and observing the behavior and results can also be an interesting topic.

References

[1] Wi-Fi Alliance, "Wi-fi alliance® 2022 wi-fi® trends," Jan 2022, accessed on 14 May 2022. [Online]. Available: https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-2022-wi-fi-trends

[2] S. M. Kerner, "Colonial pipeline hack explained: Everything you need to know," Apr 2022, accessed on 15 May 2022. [Online]. Available: https://www.techtarget.com/whatis/feature/Colonial-Pipeline-hack-explained-Everything-you-need-to-know

[3] S. Morgan, "2022 cybersecurity almanac: 100 facts, figures, predictions and statistics," Jan 2022, accessed on 14 May 2022. [Online]. Available: https://cybersecurityventures.com/cybercrime-damage-costs-10-trillion-by-2025/

[4] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in WPA2," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[5] M. Vanhoef and E. Ronen, "Dragonblood: Analyzing the dragonfly handshake of wpa3 and eap-pwd," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 517–533, accessed on 16 February 2022.

[6] Alipay, "owfuzz," accessed on 10 May 2022. [Online]. Available: https://github.com/alipay/Owfuzz

[7] L. Butti, "wifuzzit," accessed on 10 May 2022. [Online]. Available: https://github.com/0xd012/wifuzzit

[8] M. E. Garbelini, C. Wang, and S. Chattopadhyay, "Greyhound: Directed greybox wi-fi fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 817–834, 2022.

[9] S. Heijse and I. p. v. Moerman, "Ieee 802.11 fuzz-testen van de fysische laag gebruik makend van openwifi," 2021, accessed on 19 October 2021. [Online]. Available: http://lib.ugent.be/catalog/rug01:003014751

[10] Institute of Electrical and Electronics Engineers, "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 Std 802.11-2020 (Revision of IEEE Std 802.11-2016)*, pp. 1–4379, 2021.

[11] J. F. v. Kurose, L. r. Rijk, and K. W. v. Ross, *Computernetwerken : een top-down benadering*, 2nd ed. Amsterdam : Addison-Wesley, 2003. [Online]. Available: http://lib.ugent.be/catalog/rug01:001402622

[12] Institute of Electrical and Electronics Engineers, "Ieee standard for information technology – telecommunications and information exchange between systems – local and metropolitan area networks –specific requirements – part 5: Token ring access method and physical layer specifications – amendment 1: Dedicated token ring operation and fibre optic media," *IEEE Std 802.5r and IEEE 802.5j, 1998 Edition (ISO/IEC 8802-5:1998/Amd.1)*, pp. 1–420, 1998.

[13] ——, "Ieee international standard for information technology – telecommunications and information exchange between systems – local and metropolitan area networks – specific requirements – part 2: Logical link control," *ISO/IEC 8802-2:1998 ANSI/IEEE Std 802.2, 1998 edition (Incorporating ANSI/IEEE Stds 802.2c-1997, 802.2f-1997, and 802.2h-1997)*, pp. 1–256, 1998.

[14] M. Gast, *802.11 Wireless Networks: The Definitive Guide*. O'Reilly Media, Incorporated, 2005. [Online]. Available: https://books.google.be/books?id=n9WFz7CWkhAC

[15] E. Weyulu, M. Hanada, and M. Kim, "Optimizing rts/cts to improve throughput in ad hoc wlans," 09 2017, pp. 885–889.

[16] Michael Fischer. (2000, 12) A hybrid coordination function for qos. Intersil Corporation. Accessed on 25 February 2022. [Online]. Available: https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fwww.ieee802.org%2F11%2FDocuments%2FDocumentArchives%2F2000_docs%2F04528E-Hybrid_CF_for_QoS.ppt&wdOrigin=BROWSELINK

[17] Y. Son, S. Kim, S. Byeon, and S. Choi, "Symbol timing synchronization for uplink multi-user transmission in ieee 802.11ax wlan," *IEEE Access*, vol. PP, pp. 1–1, 11 2018.

[18] OpenOFDM, "Signal and ht-sig," accessed on 7 May 2022. [Online]. Available: https://openofdm.readthedocs.io/en/latest/sig.html

[19] Keysight, "802.11n signal structure," accessed on 29 April 2022. [Online]. Available: https://rfmw.em.keysight.com/wireless/helpfiles/n7617a/mixed_mode1.gif

[20] E. Lopez-Aguilera, E. Garcia-Villegas, and J. Casademont, "Evaluation of ieee 802.11 coexistence in wlan deployments," *Wireless Networks*, vol. 25, no. 1, pp. 87–104, Jan 2019. [Online]. Available: https://doi.org/10.1007/s11276-017-1540-z

[21] CWNP, "Ieee 802.11ac – optional and required features," accessed on 3 May 2022. [Online]. Available: https://www.cwnp.com/cwnp-wifi-blog/80211acoptionalandrequiredfeatures/

[22] Aboul-Magd, Osama , "Ieee 802.11ax - an overview," url: https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fgrouper.ieee.org%2Fgroups%2F802%2F11%2FWorkshops%2F2019-July-Coex%2FCoexistence-Workshop-802-11ax-Overview.pptx&wdOrigin=BROWSELINK, 6 2019.

[23] N. I. of Standards and Technology, "Security and privacy controls for information systems and organizations," U.S. Department of Commerce, Washington, D.C., Tech. Rep. NIST Special Publication 800-53 Revision 5, 2020.

[24] P. L. Wylie and K. Crawley, *The pentester BluePrint*. Nashville, TN: John Wiley & Sons, 1 2021.

[25] M. Chapple, "Wired vs. wireless network security: Best practices," Feb 2020, accessed on 15 February 2022. [Online]. Available: https://www.techtarget.com/searchsecurity/answer/Wireless-vs-wired-security-Wireless-network-security-best-practices

[26] M. Waliullah and D. Gan, "Wireless lan security threats & vulnerabilities," *International Journal of Advanced Computer Science and Applications*, vol. 5, 01 2014.

[27] D. Shinder, "802.11i, wpa, rsn and what it all means to wi-fi security," Jul 2004, accessed on 15 February 2022. [Online]. Available: https://techgenix.com/80211i-WPA-RSN-Wi-Fi-Security/

[28] M. Vanhoef and F. Piessens, "Practical verification of wpa-tkip vulnerabilities," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 427–436. [Online]. Available: https://doi.org/10.1145/2484313.2484368

[29] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt, "On the security of RC4 in TLS," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, 2013, pp. 305–320, accessed on 15 February 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan

[30] B. Mitchell, "Wpa2 vs. wpa: What's the difference for wireless security?" Nov 2021, accessed on 15 February 2022. [Online]. Available: https://www.lifewire.com/wpa2-vs-wpa-for-wireless-security-3971350

[31] "What is 802.1x? how does it work?" Jan 2022, accessed on 16 February 2022. [Online]. Available: https://www.securew2.com/solutions/802-1x

[32] W. Stallings, *Network security essentials: Applications and standards, global edition*. Philadelphia, PA: Pearson Education, 2016.

[33] R. Awati, "What is counter mode with cipher block chaining message authentication code protocol (ccmp)?" Feb 2022, accessed on 16 February 2022. [Online]. Available: https://www.techtarget.com/searchsecurity/definition/CCMP-Counter-Mode-with-Cipher-Block-Chaining-Message-Authentication-Code-Protocol

[34] D. Harkins, "Simultaneous authentication of equals: A secure, password-based key exchange for mesh networks," in *2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008)*, 2008, pp. 839–844.

[35] D. Harkins, "Dragonfly key exchange," RFC 7664, 11 2015, accessed on 16 February 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc7664

[36] S. Cooper, "Wi-fi security enhancements: Part 1 – wpa3-personal (sae)," Oct 2019, accessed on 16 February 2022. [Online]. Available: https://wificoops.com/2019/07/28/wi-fi-security-enhancements-part-1-wpa3-personal-sae/

[37] M. Vanhoef, "Fragment and forge: Breaking Wi-Fi through frame aggregation and fragmentation," in *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, August 2021, accessed on 27 February 2022.

[38] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Chapter one - security testing: A survey," in *Security Testing: A Survey*, ser. Advances in Computers, A. Memon, Ed. Elsevier, 2016, vol. 101, pp. 1–51, accessed on 10 October 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245815000649

[39] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 206–215. [Online]. Available: https://doi.org/10.1145/1375581.1375607

[40] A. C. Tribble, "The software defined radio: Fact and fiction," in *2008 IEEE Radio and Wireless Symposium*, 2008, pp. 5–8, accessed on 12 March 2022.

[41] X. Jiao, W. Liu, M. Mehari, M. Aslam, and I. Moerman, "openwifi: a free and open-source ieee802. 11 sdr implementation on soc," in *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*. IEEE, 2020, pp. 1–2, accessed on 12 March 2022.

[42] X. Jiao, W. Liu, and M. Mehari. (2019) open-source ieee802.11/wi-fi baseband chip/fpga design. [Online]. Available: https://github.com/open-sdr/openwifi

[43] J. Devreker, "Developing ieee 802.11 phy fuzzing capabilities using the open source openwifi project," 2022.

[44] J. Devrecker. (2022) openwifi. [Online]. Available: https://github.com/redfast00/openwifi

[45] Google. (2016) Oss-fuzz: Continuous fuzzing for open source software. Accessed on 10 May 2022. [Online]. Available: https://github.com/google/oss-fuzz

[46] Beyond Security, "Bestorm: Dast with black box fuzzing," accessed on 10 May 2022. [Online]. Available: https://www.beyondsecurity.com/solutions/bestorm-dynamic-application-security-testing.html

[47] aircrack-ng, "Aircrack-ng," accessed on 10 May 2022. [Online]. Available: https://github.com/aircrack-ng/aircrack-ng/tree/master

[48] Xilinx, "Xilinx zynq-7000 soc zc702 evaluation kit," accessed on 1 May 2022. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html

[49] Anolog Devices, "Ad-fmcomms2-ebz," accessed on 1 May 2022. [Online]. Available: https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/eval-ad-fmcomms2.html

[50] Raspberry Pi, "Raspberry pi 3 model b+," accessed on 1 May 2022. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-modelb-p-lus/

[51] The Tcpdump Group. (2019) Tcpdump & libpcap. Accessed on 11 April 2022. [Online]. Available: https://www.tcpdump.org

[52] Riverbed Technology. (2018) Winpcap. Accessed on 30 May 2022. [Online]. Available: https://www.winpcap.org

[53] Nmap Project. (2021, 12) Npcap. Accessed on 30 May 2022. [Online]. Available: https://npcap.com

[54] D. Young. (2009) Radiotap. Accessed on 11 April 2022. [Online]. Available: https://www.radiotap.org/

[55] P. Biondi. (2008) Scapy. Accessed on 11 April 2022. [Online]. Available: https://scapy.net/

[56] J. M. Berg, "Mac80211 overview," Feb 2009, accessed on 12 March 2022. [Online]. Available: https://wireless.wiki.kernel.org/_media/en/developers/documentation/mac80211.pdf

[57] S. Pletinck and I. p. v. Moerman, "Greybox wi-fi fuzzing op het openwifi platform," 2021, accessed on 28 February 2022. [Online]. Available: http://lib.ugent.be/catalog/rug01:003014901

[58] T. Schuddinck. (2022) openwifi (version phy fuzzing). [Online]. Available: https://github.com/Thomas-Schuddinck/openwifi

Appendices

Appendix A: Script Parameters

flag	name	description
c	signal_field	the initial value for the signal field
d	delay	the delay in microseconds between each packet injection
e	sub_type	the packet subtype
f	fuzzing mode	the fuzzing mode: incremental (default) or random
g	mac_fuzz_field	the initial value for the first 4 bytes of the MAC header
h	help	show the help menu
i	sgi_flag	set the Short Guard Interval flag
j	jump_size_phy	the value indicating by how much the signal field should be increased after every iteration
k	jump_size_mac	the value indicating by how much the MAC header field (first 4 bytes) should be increased after every iteration
l	operation_mode	the operation mode of the NIC: a(d-hoc), m(aster or AP) or s(tation)
m	hw_mode	the IEEE 802.11 hardware mode: a, g or n (default)
n	num_packets	the number of (unique) packets will be send, i.e. the number of iterations
o	byte_order_is_reversed	flag to indicate reversed bit order (little endian) in the signal field
p	fix_parity_bit	flag to indicate if parity bit should be fixt after crafting fuzzed signal field
q	signal_field_mode	the mode of the signal field: legacy or Greenfield/HT
r	rate_index	the MCS index for rate
s	payload_size	the size for the payload
t	packet_type	the type of the packet: data (default), management, control or reserved
u	is_mac_header_reversed	flag to indicate reversed bit order (little endian) in the MAC header
v	num_repetitions	the number of times a single fuzzed packet should be sent out

Table A.1: Fuzzing Script Parameters

Appendix B: Raspberry Pi Configuration

Listing B.1: network settings Raspberry Pi(/etc/dhcpcd.conf)

```
interface eth0
static ip_address=192.168.20.2/24
static routers=192.168.20.1

# settings for integrated wireless network card
interface wlan0
static ip_address=192.168.13.2/24
static routers=192.168.13.1
static domain_name_servers=192.168.13.1

# settings for USB wireless network adapter
interface wlan1
static ip_address=192.168.0.201/24
static routers=192.168.0.1
static domain_name_servers=192.168.0.1
```

Listing B.2: route settings Raspberry Pi (/lib/dhcpcd/dhcpcd-hooks/40-route)

```
ip route add 192.168.10.0/24 via 192.168.20.1 dev eth0
```

Appendix C: Control Device Configuration

Listing C.1: network settings Control Device (/etc/netplan/01-network-manager-all.yaml)

```
network:
  ethernets:
    eth0:
      match:
        macaddress: <mac address of interface>
      set-name: eth0
      addresses: [192.168.10.1/24]
      dhcp4: false
      dhcp6: false
    eth1:
      match:
        macaddress: <mac address of interface>
      set-name: eth1
      addresses: [192.168.20.1/24]
      dhcp4: false
      dhcp6: false
  renderer: NetworkManager
  version: 2
  wifis:
    wlan0:
      access-points:
        WiFi-2.4-38A1:
          password: <password of AP>
      addresses: [192.168.0.44/24]
      dhcp4: false
      dhcp6: false
      gateway4: "192.168.0.1"
      match:
        macaddress: <mac address of interface>
      nameservers:
        addresses:
          - "8.8.8.8"
          - "8.8.4.4"
      set-name: wlan0
```

Appendix D: Signal Field Dissector

Listing D.1: dissector header file (dissector.h)

```
// Title:  signal field dissector
// Author:  Thomas Schuddinck
// Year:  2022

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <getopt.h>

typedef unsigned char u8;

static const u8 bits_to_rates[16] = {0, 48, 0, 54, 0, 12, 0, 18, 0, 24, 0, 36,
    0, 6, 0, 9};
```

Listing D.2: signal field dissector (dissector.c)

```
// Title:  signal field dissector
// Author:  Thomas Schuddinck
// Year:  2022

#include "signal_field_dissector.h"
#include "signal_field_utilities.h"



/**
 * @brief Check if the provided legacy signal field is valid
 * @param sf the provided signal field
 * @param is_reverse_bit_order whether the field is in reverse bitorder or not
 * @return whether the signal field is valid or not
 */
bool is_valid_legacy_signal_field(unsigned long int signal_field, bool
    is_reverse_bit_order){

    int length;
        u8 tail, parity, reserved, rate, rate_value;
        bool is_valid = true;
        bool is_parity_correct = check_parity(signal_field, true);
```

```
// reverse the bit order if necessary
if(is_reverse_bit_order){
       signal_field = switch_bit_order(signal_field, true);
}

// extract tail
tail = signal_field & 0x3f;
signal_field = signal_field>>6;

// extract parity
parity = signal_field & 0x01;
signal_field = signal_field>>1;

// extract length
length = signal_field & 0x0fff;
signal_field = signal_field>>12;

// extract reserved
reserved = signal_field & 0x01;
signal_field = signal_field>>1;

// extract rate
rate = signal_field & 0x0f;

// get rate value
rate_value = bits_to_rates[rate];

printf("name:\t\tdec\thex\t\tdescription\n");
printf("-------------------------------------------------\n");

if(rate_value){
   printf("rate:\t\t%d\t0x%02x\t\t%d Mbps\n", rate, rate, rate_value);
}else{
   printf("rate:\t\t%d\t0x%02x\t\t%s\n", rate, rate, "illegal (least
       significant bit is always 1)");
       is_valid = false;
}
```

```
	if(reserved == 0){
		printf("reserved:\t%d\t0x%02x\t\t%s\n", reserved, reserved,
		    "legal");
	} else {
		printf("reserved:\t%d\t0x%02x\t\t%s\n", reserved, reserved,
		    "illegal (must be zero bit)");
		is_valid = false;
	}

	printf("length:\t\t%d\t0x%04x\t\tpacket is %d bytes long\n", length,
	    length, length);

	if(is_parity_correct){
		printf("parity:\t\t%d\t0x%02x\t\t%s\n", parity, parity, "legal");
	} else {
		printf("parity:\t\t%d\t0x%02x\t\t%s\n", parity, parity, "illegal
		    (first 18 bit contain uneven '1' bits)");
		is_valid = false;
	}

	if(tail == 0){
		printf("tail:\t\t%d\t0x%02x\t\t%s\n", tail, tail, "legal");
	} else {
		printf("tail:\t\t%d\t0x%02x\t\t%s\n", tail, tail, "illegal (must
		    be all zero bits)");
		is_valid = false;
	}

	return is_valid;
}



/**
 * @brief Check if the provided greenfield signal field is valid
 * @param sf the provided signal field
 * @param is_reverse_bit_order whether the field is in reverse bitorder or not
 * @return whether the signal field is valid or not
 */
```

```
bool is_valid_greenfield_signal_field(unsigned long int signal_field, bool
    is_reverse_bit_order){

       // NOT SUPPORTED YET

       return false;

}


void usage(void)
{
       printf(
          "(c)2022 Thomas Schuddinck <thomas.schuddinck@gmail.com> \n"
          "Usage: signal_field_dissector [options]\n\nOptions"
             "\n-f/--signal_field <hexadecimal representation of signal field
                for PHY fuzzing> (hex value. example:\n"
             "     0xff2345\n"
             "     WARNING: the signal field is 24 bits, or 3 bytes long, so
                the value can't be longer than that.\n"
             "     if the value contains less than six hexadecimal values,
                they will be supplemented with zeros at the front."
          "\n-r/--the bit order (per byte) is reversed\n"
             "-m/--signal_field_mode <signal field mode>
                (l[egacy],g[reenfield/high throughput],h[ybrid])\n"
             "     [NOTE] hybrid and greenfield mode are not yet supported\n"
          "Example:\n"
          "  signal_field_dissector -f 0x8b0a00 -m l -r\n"
          "\n");
       exit(1);
}


int main(int argc, char *argv[])
{
       unsigned long int signal_field;
       bool is_reverse_bit_order = false, field_parsed = false,
          is_legacy_signal_field = true;
       char signal_field_mode = 'l';
```

```
while (1)
{
        int nOptionIndex;
        static const struct option options[] =
        {
                { "signal_field",    required_argument,  NULL, 'f' },
                { "is_reverse_bit_order",  no_argument,   NULL, 'r' },
                { "signal_field_mode",   required_argument,  NULL, 'm'},
                { 0, 0, 0, 0 }
        };
        int c = getopt_long(argc, argv, "f:m:r", options, &nOptionIndex);

        if (c == -1)
                break;
        switch (c)
        {
                case 'f':
                        signal_field = strtol(optarg, NULL, 0);
                        if(signal_field > 16777215){
                                usage();
                        }
                        field_parsed = true;
                        break;
                case 'r':
                        is_reverse_bit_order = true;
                        break;

                case 'm':
                        signal_field_mode = optarg[0];
                        if (signal_field_mode != 'l' && signal_field_mode
                            != 'g')
                                usage();
                        is_legacy_signal_field = signal_field_mode == 'l';
                        break;

                default:
                        printf("unknown switch %c\n", c);
                        usage();
```

```
                        break;
                }
        }

        if(!field_parsed ){
                usage();
        }

        printf(
                "\n--------------------------------------\n"
                "The provided signal field is %s"
                "\n--------------------------------------\n",
                is_legacy_signal_field
                ? (is_valid_legacy_signal_field(signal_field,
                   is_reverse_bit_order) ? "valid" : "NOT valid")
                : "NOT SUPPORTED YET"
        );
        return (0);
}
```

Appendix E: Modified Scapy Injection

Listing E.1: Modified Scapy injection file (scapy-test.py)

```
from scapy.all import Dot11,Dot11Beacon,Dot11Elt,RadioTap,sendp,hexdump

netSSID = 'scapytest'
iface = 'sdr0'
mac = '66:55:44:33:22:11'

dot11 = Dot11(...)
beacon = Dot11Beacon(...)
essid = Dot11Elt(...)
rsn = Dot11Elt(...)

# the Radiotap as raw bytes
raw_rad = RadioTap('\x00\x00\x1c\x00\x6f\x08\x08\x00\x12\x34\x46\x78\x9a\xbc
\xde\xf0\x00\x6c\x71\x09\xc0\x00\xde\x00\x01\x02\x00\x0f')
frame = raw_rad/dot11/beacon/essid/rsn

frame.show()
print("\nHexdump of frame:")
hexdump(frame)
raw_input("\nPress enter to start\n")

sendp(frame, iface=iface, inter=0.100, loop=1)
```

Appendix F: Fuzzer Header File Configuration

Listing F.1: Interesting lines for PHY fuzzer header (phy_fuzzer.h)

```
...

#define LOG_FILE_NAME_LENGTH 40

const char* OPERATION_MODES[] = {"AD-HOC", "AP", "STATION"};


// ---------------------------------------- START PHY CONFIG
    ----------------------------------------

// DUMMY MACS
//const u8 DESTINATION_MAC[] = {0x66, 0x55, 0x44, 0x33, 0x22, 0x11}; // DUMMY
    Destination address (another STA under the same AP)
const u8 SOURCE_MAC[] = {0x66, 0x55, 0x44, 0x33, 0x22, 0x22}; // DUMMY Source
    address (STA)
//const u8 BSSID_MAC[] = {0x66, 0x55, 0x44, 0x33, 0x22, 0x33}; // DUMMY
    BSSID/MAC of AP

// REAL MACS
const u8 BSSID_MAC[] = {<mac address of BSSID>};
//const u8 SOURCE_MAC[] = {<mac address of source>};
const u8 DESTINATION_MAC[] = {<mac address of destination>};

#define TARGET_IP_ADDRESS "192.168.0.202"
#define LOCAL_PING_CMD "ping 192.168.13.2 -c 1 > /dev/null"
#define REMOTE_PING_CMD "ssh mordred@192.168.10.1 'ping 192.168.0.202 -c 1 >
    /dev/null; echo $?'"
#define LOG_DIR "/root/logs"

#define TX_COUNT_TARGET_CMD "ssh arthur@192.168.20.2 'sudo ifconfig eth0 |
    grep \"TX packets\" | tr -s \" \" \":\"  | cut -d\":\" -f4'"
#define RX_COUNT_TARGET_CMD "ssh arthur@192.168.20.2 'sudo ifconfig eth0 |
    grep \"RX packets\" | tr -s \" \" \":\"  | cut -d\":\" -f4'"

#define TX_COUNT_LOCAL_MONITOR_CMD "cat /proc/interrupts | grep tx_itrpt | awk
```

```
    '{print $2}'"
#define TX_COUNT_LOCAL_AD_HOC_CMD "ifconfig sdr0 | grep \"TX packets\" | sed
    \"s/ errors.*//g\" | sed \"s/[^0-9]*//g\""
#define RX_COUNT_LOCAL_CMD "ifconfig sdr0 | grep \"RX packets\" | sed \"s/
    errors.*//g\" | sed \"s/[^0-9]*//g\""


// ---------------------------------------- END PHY CONFIG
    ----------------------------------------


...
```