

IEEE 802.11 Physical Layer Fuzzing Using OpenWifi

Steven Heijse

Student number: 01610472

Supervisor: Prof. dr. ir. Ingrid Moerman

Counsellors: Dr. Xianjun Jiao, Dr. German Madueno (Keysight)

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de industriële wetenschappen: elektronica-ICT

Academic year 2020-2021

IEEE 802.11 Physical Layer Fuzzing Using OpenWifi

Steven Heijse

Student number: 01610472

Supervisor: Prof. dr. ir. Ingrid Moerman

Counsellors: Dr. Xianjun Jiao, Dr. German Madueno (Keysight)

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de industriële wetenschappen: elektronica-ICT

Academic year 2020-2021

Admission to use

“The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.”

“De auteur(s) geeft (geven) de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik. Elk ander gebruik valt onder de bepalingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.”

August 15, 2021

Preface

I chose this topic to get a better insight into what “hacking” is and to better understand what it actually means in the context regarding Wi-Fi. I learned that networking in general is very complex and a lot can go wrong as well as what fuzzing means, how it works and how it can be used to attack a protocol. More practical knowledge I acquired is how to use Linux and the Linux shell as well as how to write an executable bash script.

My goal was to generate an automated fuzzer to test the physical implementation made by Openwifi. This goal was almost reached but not quite. Full automation was not possible because some steps had to be done on one computer while other test needed to be done on another and without the possibility of a direct connection files had to be transferred manually. Besides this there was a problem late into the automation process, that could not be fixed in time as well as analysis that still needs to be done manually.

A special thanks for helping me goes to Xianjun Jiao for providing insight into the Openwifi implementation and answering the many questions I posed. I also want to thank German Madueno, Michael Dieudonne and Maxime Remy from Keysight for providing insight into the principle of intellectual property as well as aiding with the setup of the OpenTAP program. Lastly I want to thank Kobe Simoens for proofreading the book on a short notice, which was very helpful in correcting small mistakes as well as clearing up vague sentences.

Abstract

This thesis attempted to make a fuzzer that can automatically change the physical layer of the Wi-Fi access point implemented by Openwifi, hence the name and it has as sole author Steven Heijse. The fuzzer consists of multiple parts executed on 2 different computers. The first computer used a Linux operating system and generates and compiles multiple drivers for the development board that is used as an access point. These drivers are a modified version of the original one made by Openwifi. The modification was made to the part of the driver that is responsible for making the Physical Layer Convergence Protocol (**PLCP**) Protocol Data Unit (**PPDU**) header. There are four possible fields in this header that can be changed: 'rate', 'reserved', 'length' and 'tail'. By modifying the driver in this way, all packets sent will be changed. This means that packets that get discarded early will not have the chance to be processed by a higher layer, which means only surface level bugs can be found. Only one of the mentioned four fields gets changed in most tests because those will have the highest chance to be decoded and used by the device that is tested. To differentiate the generated drivers, which needed to have a specific name, they were placed into directories that were named after the fields and values that changed and stayed the same. These directories were then manually transferred to a different computer running on Windows. The reason for this change is that the program used (OpenTAP) to test sequences and log the outputs of commands was much more user friendly in Windows. After tests were done, the logs had to be analysed manually because there was no sufficient time to make a robust analysis script from scratch.

Keywords: Fuzzing, PHY, 802.11, Wi-Fi, OpenTAP

Extended abstract

Introduction

Current state-of-the-art, Wi-Fi only fuzzers only look at the datalink layer of the Open Systems Interconnection (OSI) model while the 802.11 standard defines both the datalink and the physical layers. The reason for this is that these fuzzers use commercial Wi-Fi chips. These chips have the physical implementation of the 802.11 standard baked into the hardware. This approach has the consequence that it is not possible to change or otherwise interfere with the physical layer as a user.

The open-source implementation of the 802.11 standard that Openwifi provides changes this and allows full access to not only the physical layer but also the complete implementation of the Wi-Fi chip itself. This includes both the driver and the Field-programmable gate array (FPGA) design. An approach like this allows for new avenues of research regarding the physical layer, which were not possible before, as well as potentially new ways to attack wireless devices. The latter could potentially affect the security of users and corporations and the reputation of its manufacturers.

If changing or misusing the physical layer actually has an effect on the tested implementation or not needs to be thoroughly researched to raise awareness to potential new wireless attacks that might become possible and to create countermeasures ahead of time.

This thesis attempted to explore one of these avenues and tested the implementation of the physical layer of a commercial Wi-Fi chip. More specifically this means sending a signal with malformed information in the PPDU header to a test device and seeing if it has an effect and logging the results. This was done by manually testing certain cases and by implementing a fuzzer to automatically test and

log the results of specific modifications.

Implementation

The Approach takes to make the fuzzer has 2 parts that are executed on different computers. The computer that does the first part uses Linux as an operating system, which the same as in the Openwifi documentation on their Github page. The computer uses a bash script to go loop over values for the 'rate', 'reserved', 'length' and 'tail' fields used in the header of the physical layer otherwise known as the PPDU. For each value a driver is generated using a Python script and compiled. Since these driver files require an exact name, a way is needed to differentiate them. This differentiation is accomplished but naming the directories in which they are stored after the values modified in the driver. After all the wanted drivers are generated they are manually transferred to the second computer which uses windows.

By using this approach all packet that are sent from the Access Point (AP) will be modified according to these values. Since individual packets can't be modified, it is impossible to guaranty that they will propagate further into the protocol and only surface level fuzzing is possible. To increase the chance that modified packets interact with the phone, only one of the four field gets modified in most tests.

The second computer is necessary because OpenTAP is a lot more user friendly on Windows. OpenTAP is used to run a sequence of test steps that in general start the AP, logs internal parameters of the Device Under Test (DUT) and then terminates the access point for every driver tested. The DUT, in this case an Android phone, is set to automatically connect to only the Openwifi AP.

Results

The results for the driver using the even rate values, which have no meaning and are not defined by the standard, implies that the safety built into the **FPGA** implementation either does not work correctly or only modifies the rate at which the packets are sent and not the rate value in the header that determines the encoding of the following data.

The drivers where an odd value for the rate was used (besides 13) and the **AP** was started using 'fosdem.sh', resulted in malformed Dynamic Host Configuration Protocol (**DHCP**) and Bootstrap Protocol (**BOOTP**) packets being sent to the phone, which was not able to acquire an Internet Protocol (**IP**) address and thus no usable connection was established. When the same drivers were used in combination with 'fosdem-11ag.sh' there was no problem connecting to the phone and ping replies were successfully received.

As for the changes in the length value, the only cases where a connection was established occurred when the length was only slightly larger than the original when 'fosdem.sh' was used. When 'fosdem-11ag.sh' was used some increases allowed for a connection while others did not and would require more testing to make a more definite judgement. There was no connection possible when decreases in length were used for both modes. These results could be also be attributed to the degree of successful decoding that was possible of the data that came after the header.

For the drivers using modified tail values there was no difference between the mode used ('fosdem.sh' or 'fosdem-11ag.sh'). The driver could be separated in 3 groups. One where

there was no problem connecting, one where the phone was able to see the **AP** was active but was not able to connect to it and one where neither the computer running Wireshark nor the phone was able sense the packets. These results can presumably be contributed to the degree of decoding was correct.

Lastly, the tests done using a changed reserved field did not differ from regular operation.

Conclusion

This thesis shows that it is possible to fuzz the physical layer of the IEEE 802.11 standard by modifying the implementation made by Open-wifi. Automation was shown to be possible using OpenTAP and a combination a different scripts although there is still a problem that needs to be solved which most likely a result of the spaces in the directory names used. The results from the manual tests as verification show that physical layer did have an influence.

Instead of a fully useful fuzzing implementation the can be considered more of a proof of concept that modifying the physical layer is possible. The reason for saying this is that the tests did not have any visible effect on the functionality of the phone that could be attributed to the changed driver with absolute certainty. In conclusion, there is still uncertainty about the impact the changed drivers actually had on the phone. This could be partly solved by using being able to fully understand the contents of the 'dumpsys wifi' and 'dumpsys connectivity' commands as well as doing automatic analysis on logs where these commands and the others mentioned in this thesis are called a lot more frequently instead of just once or twice.

Uitgebreide samenvatting

Introductie

De huidige state-of-the-art fuzzers die zich richten op Wi-Fi, kijken alleen naar de datalink-laag van het OSI-model. Terwijl de 802.11-standaard zowel de datalink als de fysieke lagen definieert. De reden hiervoor is dat deze fuzzers commerciële Wi-Fi chips gebruiken. De fysieke implementatie van de 802.11-standaard is ingebakken in de hardware van deze chips. Dit betekent dus dat het niet mogelijk is om de fysieke laag als gebruiker te wijzigen.

De open-source implementatie van de 802.11-standaard die Openwifi voorziet, verandert dit en geeft volledige toegang tot niet alleen de fysieke laag, de volledige implementatie van de Wi-Fi chip zelf. Dit omvat zowel het stuurprogramma als het FPGA-ontwerp. Dit maakt nieuwe onderzoeksrichtingen mogelijk die voorheen niet mogelijk waren, maar heeft ook het gevolg dat er mogelijks nieuwe manieren zullen worden ontwikkeld om draadloze apparaten aan te vallen. Dit laatste kan mogelijk gevolgen hebben voor de veiligheid van gebruikers en bedrijven en voor de reputatie van de fabrikanten.

Als het wijzigen of misbruiken van de fysieke laag daadwerkelijk een effect heeft op de geteste implementatie of niet, moet grondig worden onderzocht om het bewustzijn te vergroten voor mogelijke nieuwe draadloze aanvallen die mogelijk worden en om van tevoren tegenmaatregelen te nemen.

Deze Thesis probeerde één van deze wegen te verkennen en de implementatie van de fysieke laag van een commerciële Wi-Fi chip te testen. Meer specifiek betekent dit het sturen van een signaal met foutieve informatie in de PPDU-header naar een testapparaat, het kijken of het effect heeft en het loggen van de resultaten. De testen zelf werden gedaan door

in bepaalde gevallen handmatig te testen of door een fuzzer te implementeren om de resultaten van specifieke wijzigingen automatisch te testen en te loggen.

Implementatie

De implementatie die nodig is om de fuzzer te maken bestaat uit 2 delen die op verschillende computers worden uitgevoerd. De computer die het eerste deel doet, gebruikt Linux als besturingssysteem, wat hetzelfde is als in de Openwifi-documentatie op hun Github-pagina. De computer gebruikt een bash-script om de waarden te doorlopen voor de velden 'rate', 'reserved', 'length' en 'tail' die worden gebruikt in de header van de fysieke laag, ook wel bekend als de PPDU. Voor elke waarde wordt met behulp van een Python-script een driver gegenereerd en gecompileerd. Aangezien deze driver bestanden een exacte naam vereisen, is er een manier nodig om ze te onderscheiden. Deze onderscheiding wordt bereikt, door de mappen waarin ze zijn opgeslagen, te noemen naar de waarden die in de driver zijn gewijzigd. Nadat alle gewenste drivers zijn gegenereerd, worden ze handmatig overgebracht naar de tweede computer die Windows gebruikt.

Door deze benadering te gebruiken, zullen alle pakketten die vanaf de AP worden verzonden, worden aangepast volgens deze waarden. Aangezien individuele pakketten niet kunnen worden gewijzigd, is het onmogelijk om te garanderen dat ze zich verder in het protocol zullen verspreiden en is alleen fuzzing op oppervlakteniveau mogelijk. Om de kans te vergroten dat gewijzigde pakketten met de telefoon communiceren, wordt in de meeste tests slechts één van de vier velden gewijzigd.

De tweede computer is nodig omdat OpenTAP op Windows een stuk gebruiksvriendelijker is. OpenTAP wordt gebruikt om een reeks

teststappen uit te voeren die in het algemeen de **AP** starten, interne parameters van de **DUT** loggen en vervolgens het toegangspunt voor elk getest stuurprogramma beëindigen. De **DUT**, in dit geval een Android-telefoon, is ingesteld om automatisch verbinding te maken met alleen de Openwifi **AP**.

Resultaten

De resultaten voor de drivers die de even ratewaarden gebruiken, deze hebben geen betekenis en niet worden gedefinieerd door de standaard, impliceren dat de veiligheid ingebouwd in de **FPGA**-implementatie ofwel niet correct werkt of alleen de snelheid wijzigt waarmee de pakketten worden verzonden en niet de snelheidswaarde in de header die de codering van de volgende gegevens bepaalt.

De stuurprogramma's waarbij een oneven waarde voor de snelheid werd gebruikt (behalve 13) en de **AP** werd gestart met 'fosdem.sh', resulteerden in het verzenden van misvormde **DHCP** en **BOOTP** pakketten naar de telefoon, die geen **IP**-adres kon verkrijgen en dus geen bruikbare verbinding tot stand kon brengen. Wanneer dezelfde stuurprogramma's werden gebruikt in combinatie met 'fosdem-11ag.sh' was er geen probleem om verbinding te maken met de telefoon en werden pingantwoorden met succes ontvangen.

Wat betreft de veranderingen in de 'length'-waarden, de enige gevallen waarin een verbinding tot stand werd gebracht, deden zich voor wanneer de lengte slechts iets groter was dan het origineel toen 'fosdem.sh' werd gebruikt. Wanneer 'fosdem-11ag.sh' werd gebruikt, lieten sommige verhogingen een verbinding toe, terwijl andere dat niet deden en meer testen zouden vergen om een definitiever oordeel te vellen. Er was geen verbinding mogelijk wanneer voor beide modi lengteverminderingen werden gebruikt. Deze resultaten kunnen ook worden toegeschreven aan de mate van succesvolle decodering die mogelijk was van de gegevens die na de kop kwamen.

Voor de drivers die aangepaste 'tail'-waarden gebruikten, was er geen verschil tussen

de gebruikte modus ('fosdem.sh' of 'fosdem-11ag.sh'). De drivers konden in 3 groepen worden verdeeld. Eén waarbij er geen probleem was om verbinding te maken, één waarbij de telefoon kon zien dat de **AP** actief was maar er geen verbinding mee kon maken en één waarbij noch de computer waarop Wireshark draaide, noch de telefoon de pakketten kon detecteren. Deze resultaten kunnen vermoedelijk worden bijgedragen aan de mate waarin de decodering gelukt is.

Ten slotte verschilden de testen die werden gedaan met een gewijzigd 'reserved' veld niet van de normale werking.

Conclusie

Deze Thesis laat zien dat het mogelijk is om de fysieke laag van de IEEE 802.11-standaard te fuzzen door de implementatie van Openwifi aan te passen. Automatisering bleek mogelijk te zijn met behulp van OpenTAP en een combinatie van verschillende scripts, hoewel er nog steeds een probleem moet worden opgelost dat hoogstwaarschijnlijk het gevolg is van de spaties in de gebruikte mapnaam. De resultaten van de handmatige testen als verificatie laten zien dat fysieke laag wel degelijk invloed heeft gehad.

In plaats van een volledig bruikbare fuzzing-implementatie kan deze thesis meer worden beschouwd als een proof of concept dat het wijzigen van de fysieke laag mogelijk is. De reden om dit te zeggen is dat de testen geen zichtbaar effect hadden op de functionaliteit van de telefoon die met absolute zekerheid kon worden toegeschreven aan de gewijzigde driver. De conclusie is dus dat er nog onduidelijkheid over is de impact die de gewijzigde drivers daadwerkelijk hebben gehad op de telefoon. Dit kan gedeeltelijk worden opgelost door gebruik te maken van het volledig begrip van de inhoud van de 'dumpsys wifi' en 'dumpsys connectiviteit' commando's en door automatische analyse uit te voeren op logs waar deze commando's en de andere die in deze thesis worden genoemd veel vaker worden aangeroepen in plaats van slechts één of twee keer.

Table of contents

List of Figures

List of Tables

List of Code Listings

1	Introduction	1
2	IEEE 802.11	4
2.1	Communication Flow Description	4
2.2	OFDM PHY layer: PLCP header	6
3	Fuzzing	9
3.1	What is fuzzing?	9
3.1.1	Types of fuzzers	9
3.2	Why use fuzzing?	10
3.3	Potential problems	10
4	Components	11
4.1	Hardware	11
4.1.1	Personal computers & DUT	11
4.1.2	Development board	11
4.1.3	Hardware setup and connections	12
4.2	Software	14
4.2.1	OpenTAP	14
4.2.2	Wireshark	16
4.2.3	Opensource fuzzers	18
5	Testing	22

5.1	Precautions	22
5.2	Setup	22
5.2.1	Changes to the driver	23
5.2.2	Generating drivers	25
5.2.3	OpenTAP	28
5.2.4	Remarks	31
6	Results	33
6.1	Rate values	33
6.2	Length values	36
6.3	Tail values	38
6.4	Possible future work	42
6.4.1	Possible optimizations	42
6.4.2	Future research	43
	Appendix A How to install the adb-tool	44
	Appendix B modified 'calc_phy_header()'	45
	Appendix C change_sdr.py	48
	Appendix D generate_test_drivers.sh	50
	Appendix E Scp_Driver_To_Board.cs	54
	Appendix F Summary of information from an example log	57
	Bibliography	69

List of Figures

1.1	Logos of related institutions	1
1.2	Estimated number of Wireless Local Area Network (WLAN) connected devices worldwide from 2016 to 2021 [1]	2
2.1	OSI-model [2]	4
2.2	Association process [3]	5
2.3	Orthogonal Frequency Division Multiplexing (OFDM) PLCP framing format [4]	6
2.4	Preamble and frame start [4]	7
2.5	Signal field of OFDM PLCP frame [4]	7
4.1	Currently supported boards for Openwifi with the board highlighted in yellow, being the setup used in this thesis [5]	12
4.2	Reminder about the potential dangers that enabling the developer options can have and to what access is given	14
4.3	USB-debugging options that were enabled throughout the thesis	15
4.4	Result of the 'adb devices' command on a Linux computer where the Rivest–Shamir–Adleman (RSA)-key was not accepted	15
4.5	Result of the 'adb devices' command on Windows computer where RSA-key was accepted	16
4.6	Pop-up screen asking if the user to allow the USB-debugging	16
4.7	Part of a Wireshark output only showing the traffic between the board and the phone by using a display filter	17
4.8	Section of the 'iw list' output that shows the supported interface modes for this device	17
4.9	Section of the 'iw list' output that shows the supported frequencies for the second band for this device	18
4.10	The given shell inputs and received outputs when enabling monitor mode using the bashscript provided by Openwifi	19
4.11	The given shell inputs and received outputs when enabling monitor mode	20
4.12	The given shell inputs and received outputs when going back to the 'managed' interface mode	21

5.1	Interrupt 53 and 54 of the output from the command 'cat /proc/interrupts' . . .	22
5.2	Ethernet IP settings used when connecting to the board on the Windows computer	23
5.3	Settings for the Secure Shell (SSH) instrument in OpenTAP	28
5.4	General TapPlan used to run tests	29
5.5	Test sequence using the changed driver	30
5.6	Settings of the custom test step that copies the driver to the board	30
5.7	TapPlan reference used to log the output from the adb commands	31
6.1	802.11 Radio information from a broadcast packet sent using 'fosdem.sh' and rate set to nine	35
6.2	Information reported by Wireshark for a Boot Reply packet sent using 'fosdem.sh' and a rate value set to nine	37
6.3	Expanded analysis from Wireshark on a broadcast packet sent with a length value that was increased with one	38
6.4	Expanded analysis from Wireshark on a broadcast packet sent with a length value that was decreased with one	38

List of Tables

2.1	Possible rate subfield values and corresponding data rate and modulation[6]	7
6.1	Results first automated test regarding even rate values part 1	34
6.2	Results first automated test regarding even rate values part 2	35
6.3	Results of manual tests that change the rate value using fosdem.sh	36
6.4	Results of manual tests that change the rate value using fosdem-11ag.sh	36
6.5	Results of manual tests that change only the tail value	39

List of Code Listings

5.1	Field values are modified by the user or the program, with the value range in comments	24
5.2	Lines of code responsible for mirroring the tail and rate bytes	24
5.3	Difference between the original bytes and the modified bytes	25
5.4	Command to prepare the kernel	26
5.5	Loop generating and compiling the driver file 'sdr.c'	27

List of abbreviations

adb	Android Debug Bridge
AP	Access Point
ADC	Analog-to-digital converter
BOOTP	Bootstrap Protocol
CSIS	Center for Strategic & International Studies
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DUT	Device Under Test
FPGA	Field-programmable gate array
HT	High Throughput
ICMP	Internet Control Message Protocol
IDLab	Internet Technology and Data science Lab
IEEE	Institute of Electrical and Electronics Engineers
IMEC	Interuniversity Microelectronics Centre
IP	Internet Protocol
KRACK	Key Reinstallation Attack
LSB	Least Significant Bit
MAC	Media Access Control
MCS	Modulation Coding Scheme
MSB	Most Significant Bit
OFDM	Orthogonal Frequency Division Multiplexing
OS	Operating System
OSI	Open Systems Interconnection
PCB	Printed Circuit Board
PHY	Physical layer
PLCP	Physical Layer Convergence Protocol
PMD	Physical Medium Dependent
PPDU	PLCP Protocol Data Unit

RSA	Rivest–Shamir–Adleman
RSSI	Received Signal Strength Indicator
SDK	Software Development Kit
SDR	Software-defined radio
SIFS	Short Interframe Space
SOC	System-on-a-Chip
SOM	System-on-Module
SSH	Secure Shell
SSID	Service Set Identifier
WEP	Wired Equivalent Privacy
WLAN	Wireless Local Area Network
WNIC	Wireless Network Interface Controller
WPA2	Wi-Fi Protected Access II

Chapter 1: Introduction



Figure 1.1: Logos of related institutions

In the first quarter of 2021 alone there have already been multiple significant cyber incidents and since the world is becoming more and more digital, it is safe to assume that these kinds of incidents will only become more prevalent. These cyber attacks can have a huge influence on the lives of millions of people. These incidents are also not limited to stealing money or denying services but can physically harm people or ruin their career by acting in other people's name on social media or revealing private information.

An example of one of these incidents is the breach of the water systems in Florida. [7] The cyber criminal tried to poison the water supply by increasing the amount of sodium hydroxide to a dangerous level. If the intervention had not been not on time this incident could have lead to permanent damage or death of a large number of people. A second example is the recent attack on the Microsoft exchange server, during which four vulnerabilities where exploited. The attack affected millions of Microsoft clients. Nine government agencies and 60.000 private companies were affected in the United States alone. A timeline of all the recorded cyber incidents since 2006 can be found in [8] by the Center for Strategic & International Studies (CSIS).

New security protocols need to be developed or old ones revised. The raw computing power that becomes available is always increasing, which allows hackers to brute force their way in without even relying on human error or carelessness. Quantum computing will also have a huge impact in the coming decade. Once Quantum computers become more powerful and more widely available, they will make the most powerful encryption methods that are available today obsolete.[9] If the advancements are combined with the flaws that exists in protocols, it will become relatively easy to access or change data that is considered secure.

Nothing is flawless. This includes applications, operating systems and wireless protocols. All of these can be attacked in different ways and for different reasons. Examples are hacking an iphone to get access to root privileges, hacking and distributing games for free, denying access to servers of a certain company to make them lose money, or getting access to sensitive information for blackmail.

Of the multiple wireless protocols, such as Bluetooth, LoRa, cellular. the most widespread is Wi-Fi. According to statista (1.2) the number of WLAN connected devices in 2020 was 18.21 billion. [1]

Wi-Fi itself has its fair share of trouble with security, as the original security protocol Wired Equivalent Privacy (WEP) had multiple design flaws. [10] Currently, The most widely used security protocol for Wi-Fi is Wi-Fi Protected Access II (WPA2) for which a weakness was found a few years ago, in 2017, which allowed for a new sort of attack called a Key Reinstallation Attack (KRACK). A KRACK allows attackers to read information that was believed to be safely encrypted. Depending on the network configuration it is also possible to inject or manipulate data. Since the weakness is part of the Wi-Fi standard itself, it does not require human error

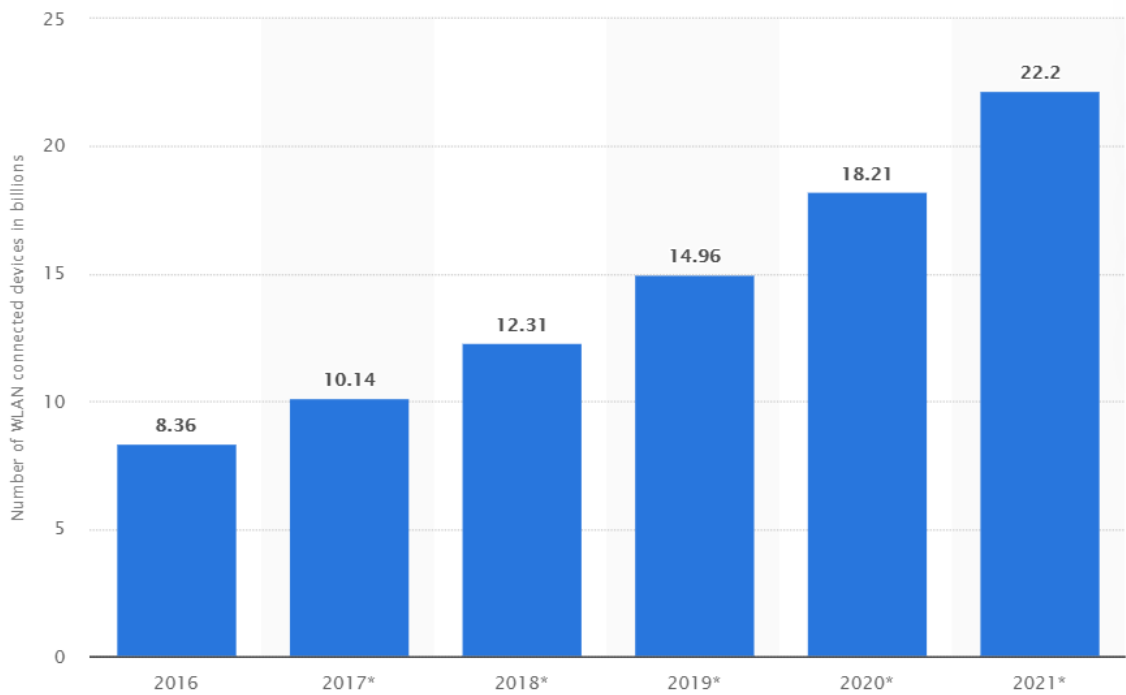


Figure 1.2: Estimated number of **WLAN** connected devices worldwide from 2016 to 2021 [1]

and any correct implementation using **WPA2** will be affected by it.[11] [12] This has led to the development of WPA3.

Searching for vulnerabilities is a time-consuming process because of all the possibilities that need to be tested and verified. This process could greatly benefit from automation and a good method automation method is called fuzzing or fuzz testing.

The current state of the art fuzzer implementation published in 2020 regarding Wi-Fi is called greyhound. It uses a state machine to simulate the Wi-Fi protocols and allows for more in-depth testing. It can explore deep states of the Wi-Fi protocol and find more complex non-compliant behaviours of which older Wi-Fi fuzzing tools were incapable.[13]

Even so, all of the current Wi-Fi fuzzers only take into account the data link layer of the **OSI** model, the Institute of Electrical and Electronics Engineers (**IEEE**) standard defines both the data link layer and the physical layer. The reason for this is that Wi-Fi chips used to test these protocols have a fixed implementation that cannot be changed by the user. This is obviously not necessary for commercial appliances, but what if this implementation contains faults outside the scope of normal operation. An extreme example of this would be the implementations crashes and resets when a certain input or sequence of inputs is received.

Problem statement

The aim of this thesis is to investigate the possibility that future wireless attacks could include a changed physical layer by testing it using fuzzing. This thesis work has been supported by the from Internet Technology and Data science Lab (**IDLab**) UGhent, Interuniversity Microelectronics Centre (**IMEC**) and Keysight Technologies.

The scope of this study is limited to the **OFDM** implementation of the physical layer because this is what is implemented in the Openwifi project. It will only look at the possibilities that changing the driver provides. This thesis will not examine or change the **FPGA** implementation. The Openwifi implementation is capable of using two standards IEEE802.11g and IEEE802.11a (also known as WIFI2). During testing IEEE802.11g proved to be unreliable at times so only IEEE802.11a will be used.

The purpose is to examine whether only changing the physical layer has any influence on the behavior of the connected devices furthermore estimate the risk of new wireless hacks using the physical layer. Since the topic of fuzzing was unknown ground prior to starting this thesis, it also provided a learning opportunity for familiarization with fuzz testing and the impact it has on software development and security testing.

The outline of this thesis will be as follows. In chapter 2 the value of fuzzing and its basic principles will be explained. Chapter 3 will provide the necessary information about the **IEEE** protocol to understand the implementation that will be discussed in chapter 4. This chapter also contains information about possible alternatives that were considered but ultimately not used. Chapter 5 will explain how the tests were actually conducted followed by chapter 6 that will present the results.

Chapter 2: IEEE 802.11

The purpose of this chapter is to provide an overview of the IEEE 802.11 standard, widely known as Wi-Fi. Only the relevant parts of the standard will be covered here, since its size is vast. More information on the standard itself can be found in [6].

The IEEE 802.11 standard specifies the Media Access Control (**MAC**) (**OSI** Data link layer in Figure 2.1) and Physical layer (**PHY**) (**OSI** physical layer in Figure 2.1) protocols for implementing **WLAN** computer communication. The IEEE standard provides a basis for wireless products, using the Wi-Fi brand. It is mostly used in house-and office networks to allow devices to communicate wirelessly with other devices or to connect to the internet.

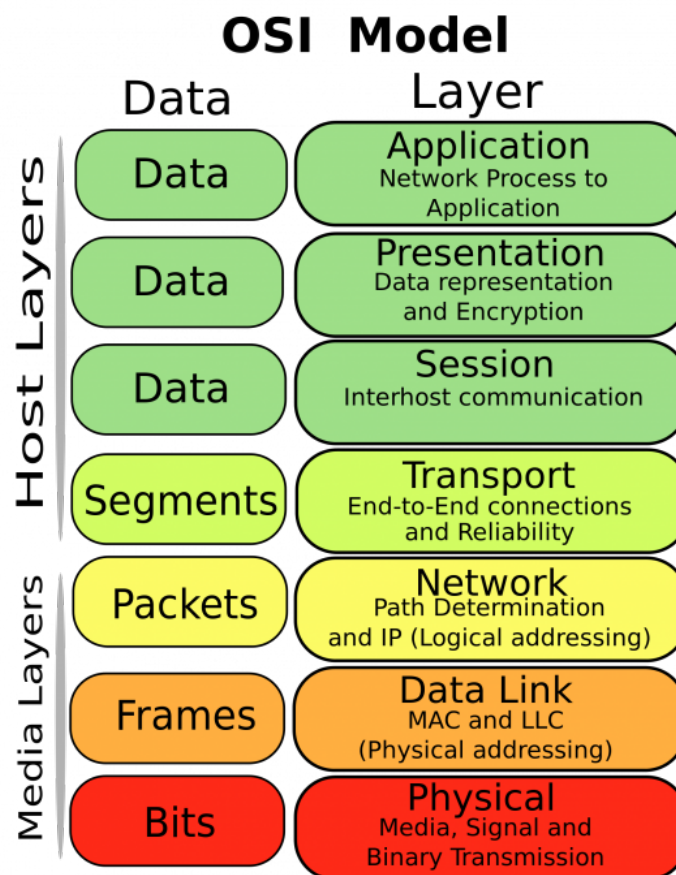


Figure 2.1: **OSI**-model [2]

2.1 Communication Flow Description

Before information can be exchanged between two wireless devices, a connection needs to be established. The process behind this consists of multiple steps shown in Figure 2.2, and they are associated with three possible connection states:

1. Not authenticated nor associated
2. Authenticated but not yet associated
3. Authenticated and associated

State 1 represents a device and a computer that are not connected. Authentication would then be similar to plugging the device into the computer. The process is then in state 2. After association, state 3 is achieved and data can be exchanged between the computer and the device. The third state represents an established connection, which allows data to be transferred. The process necessary to get there is shown in Figure 2.2.

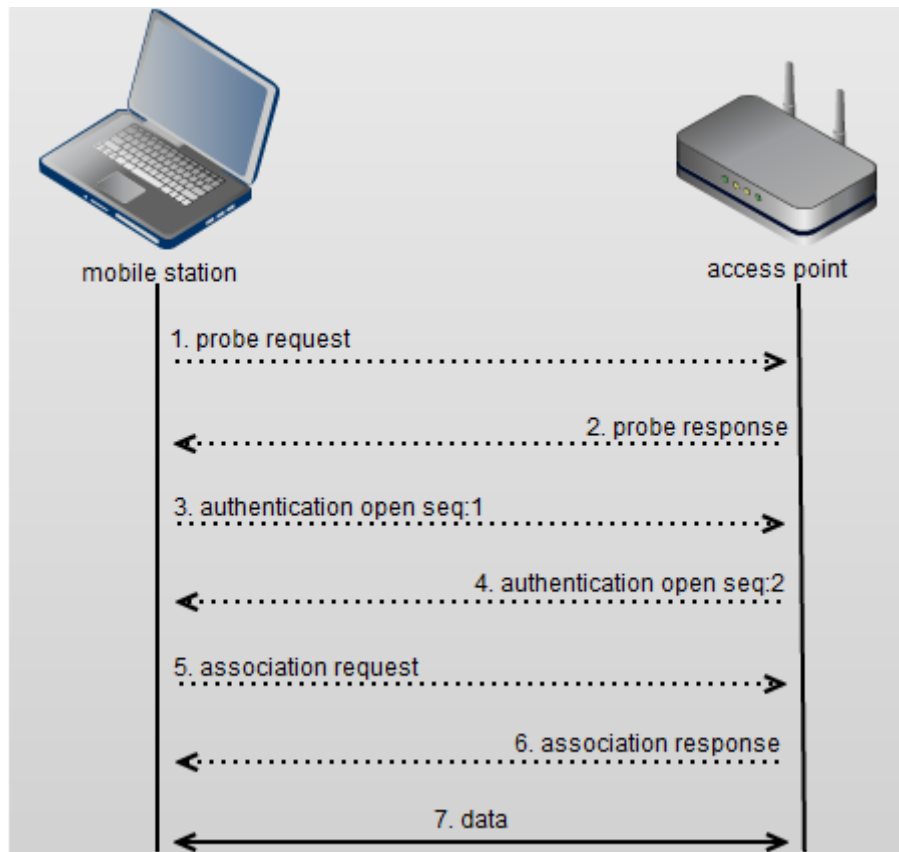


Figure 2.2: Association process [3]

A device that wants to connect to an **AP** is called a client device. It scans the medium for broadcasting networks and selects a Service Set Identifier (**SSID**) (name of the AP) to connect to. The client then sends a *probe request* to that **SSID** and its corresponding **AP**. The *probe request* contains information about the supported data rates and 802.11 capabilities (for example 802.11g is supported).

The **APs** that have a common supported data rate will then respond with a *probe response* which will contain information about the **AP** and its capabilities. This information includes but is not limited to **SSID**, supported data rates and encryption if required.

The client will check the probe's responses it received for compatible encryptions and chooses one of the available networks. The client station then sends an *authentication request*, a low-level 802.11 authentication frame, to the **AP**.

The **AP** receives the authentication frame and responds to the mobile station with an authentication frame , also known as *authentication response*.

If the **AP** receives any frame other than an authentication- or probe request frame from a mobile station that is not authenticated it will respond with a deauthentication frame, placing the mobile station into an unauthenticated and unassociated state. The station will have to begin the association process from the low level authentication step. At this point the mobile station is authenticated but not yet associated.

Once the client station determines which access point it would like to associate to, it will send an *association request* to that access point. The association request contains chosen encryption types if required and other compatible 802.11 capabilities.

If the elements of an association request match the capabilities of the access point, the **AP** will create an Association ID for the mobile station and respond with an *association response* with a success message, granting network access to the mobile station. After successful association, data transfer can occur. [14] [15]

2.2 OFDM PHY layer: PLCP header

The physical layer that is defined by the standard consists of two sublayers: the **PLCP** sublayer, which takes the frame from the **MAC** layer and adds a header; and the Physical Medium Dependent (**PMD**) sublayer, which is responsible for modulating and transmitting the frame. The header added by the **PLCP** sublayer is called the **PPDU** and will be practically changed in this thesis by setting the fields from the header to fixed values.

According to the standard there are multiple modulation techniques that can be used for the physical layer. Openwifi implemented **OFDM** modulation, thus only the **PPDU** structure for **OFDM** will be outlined here.

The **PLCP** protocol adds two fields to the frame received from the **MAC** layer: a preamble and a signal field (Figure 2.3). The preamble consists of twelve **OFDM** symbols that, used to synchronise various timers between transmitter and receiver.

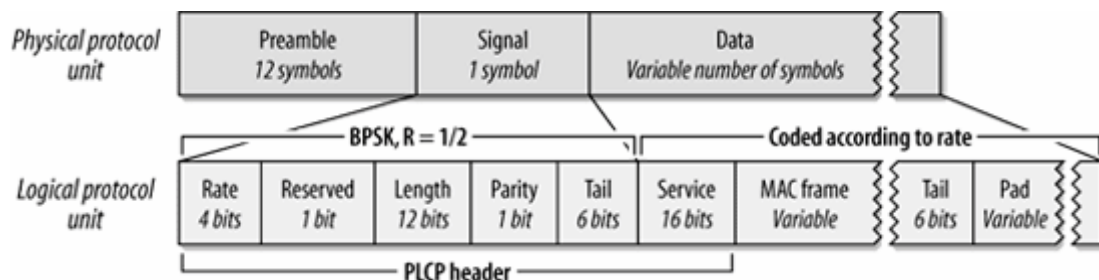


Figure 2.3: **OFDM PLCP** framing format [4]

The first ten symbols are a short training sequences and are represented by t1 through t10 in Figure 2.4. These sequences, are used by the receiver to lock on to the signal, select an appropriate antenna if the receiver is using multiple antennas, and synchronize the large-scale timing relationships required to begin decoding the subsequent symbols. These short training sequences are followed by a guard period, which is followed by the remaining two symbols. These remaining two symbols are long training sequences and are used to fine-tune the timing acquisition, and are represented by LT1 and LT2 in Figure 2.4.

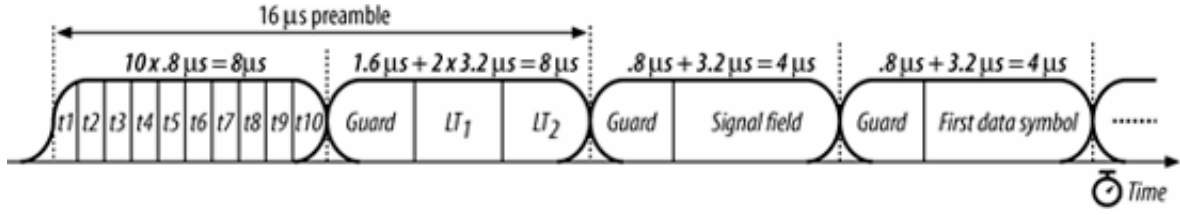


Figure 2.4: Preamble and frame start [4]

The preamble is configured in the **FPGA** and consequently cannot be changed from the driver file. The signal field is made up of five subfields, which add up to a total of 24 bits or 3 bytes as can be seen in Figure 2.5.

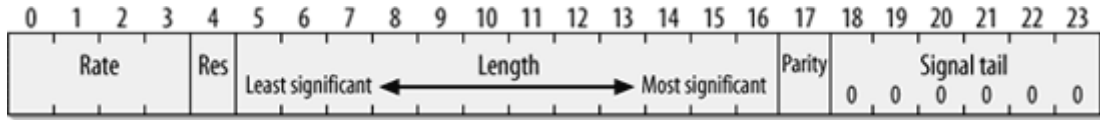


Figure 2.5: Signal field of **OFDM PLCP** frame [4]

The rate field consists of 4 bits and determines the data rate and modulation scheme used for the rest of the **PPDU**, starting immediately after the Signal field. All possible data rates and corresponding modulation schemes are represented in Table 2.1.

Bits 1-4	Data Rate	Modulation
1101	6 Mbps	BPSK
1111	9 Mbps	BPSK
0101	12 Mbps	QPSK
0111	18 Mbps	QPSK
1001	24 Mbps	16QAM
1011	36 Mbps	16QAM
0001	48 Mbps	64QAM
0011	54 Mbps	64QAM

Table 2.1: Possible rate subfield values and corresponding data rate and modulation[6]

The length field consists of 12 bits with the first being the Least Significant Bit (**LSB**) and the last being the Most Significant Bit (**MSB**). This field encodes the number of bytes in the embedded **MAC** frame and both the rate and length fields are required for decoding the DATA part of the packet. [6].

Of the two fields that only consist of 1 bit, the reserved bit is currently reserved for future use. Reserved for future use means that the reserved bit is not used in the current standard but it is already provide as a possibility for a future version. It should always be set to 0. The parity bit is used to make the parity of the first 16 bits even to protect against data corruption.

The last subfield in the signal field is the signal tail, which consists of 6 bits, all of which must be 0. The tail bits in the signal symbol (visible in Figure 2.3) enable decoding of the rate and length fields immediately after the reception of the tail bits. [6]

The service field (16 bits) is not part of the five previously mentioned fields because it is part of the data field. This means that it is sent at the transmission rate of the embedded **MAC** frame. Of these 16 service bits the first 6 are used to initialize a scrambler and the remaining bits are reserved for future use.

The **PLCP** protocol also adds a trailer and padding at the end of the frame. This trailer has 6 tail bits that have to be 0, to end the convolution of the **MAC** frame. The padding is variable because it is used to ensure a fixed block size. This block size depends on the modulation and encoding used.^{[4][6]}

Chapter 3: Fuzzing

The tests that are used to determine whether piece of code or software contains bugs is generally considered either static or dynamic. In a static test, a number of software engineers come together and speculate where potential errors can occur, or where the program might fail. They do this without actually executing the program itself. This approach to testing is prone to human errors and is, therefore, used in combination with dynamic tests. These dynamic tests do execute the code with certain inputs to determine whether the program behaves as expected.

3.1 What is fuzzing?

Fuzzing is a dynamic testing method that generally uses random inputs to test the behavior of a system and log the results. In these logs the used input is recorded as well as the unexpected response from the **DUT**. These tests can be done much faster than static tests because they can be done automatically.

3.1.1 Types of fuzzers

Fuzzers can be classified in multiple ways. A fuzzer can be classified on the knowledge used when generating inputs but they can also be classified based on the way the inputs are generated.

Black-box fuzzers are fuzzers that have no knowledge about the input they are fuzzing. They just create entirely random inputs, which are then fed to the **DUT**. The advantages of black-box fuzzers is that they are easy to program and the inputs can be generated very fast. This comes with the disadvantage that the fuzzing will be superficial. It will be very unlikely that the generated input will propagate through the system and expose bugs hidden deeper within the said system. For example, if the purpose is fuzzing the implementation of a protocol, then to progress further through the protocol, the inputs will have to conform to that protocol which is very unlikely when it is randomly generated.

Next there are white-box fuzzers, which have a complete understanding of the implementation. These have the advantage of generating inputs based on the implementation, which enormously increases the odds to get a reaction out of the system compared to random inputs. A downside of white box fuzzers is the dependence on the knowledge of the inner workings of the protocol/device/program that is targeted by the fuzzer. This knowledge is most likely not easily obtainable for a third party.

Another potential downside is that the time it takes to generate an input. This time will increase because more processing is required to take into account the relevant information when an input is formed.[16]

Greybox fuzzers are inbetween black- and white-box fuzzers. They use established rules of for example a standard to generate the input, which is based on knowledge about the **DUT** while not having a complete understanding of said **DUT**. A great example of this are commercial Wi-Fi chips. The actual hardware implementation is only known by the manufacturer, but it

has to oblige to the **IEEE** 802.11 standard. Therefore by using information about the standard, the fuzzers are able to generate semi-random inputs that are more likely to cause problems.

There are also multiple ways to generate the inputs of a fuzzer. In general, there are mutation- and generation-based fuzzers. Mutation based fuzzers use a prerecorded valid interaction with the device and mutate it to generate new inputs. Generation-based fuzzers ,in contrast, use knowledge, about a protocol for example, to generate new inputs. [17]

3.2 Why use fuzzing?

Besides fuzzing programs, it is also possible to fuzz protocol implementations. The **IEEE** 802.11 standard defines the requirements for the implementation, so that devices using the standard function correctly and so they can interact with devices using the same standard but a different implementation. This is important because the standard does not define what to do with incorrect inputs, which is left up to the designers and manufacturers. Implementations can become vulnerable if they do not take these incorrect or unexpected inputs into account. These inputs might cause a crash or cause the device to provide information which it otherwise would and should not provide. Hackers can and will take advantage of these flaws if and when they are found. This is where fuzzing comes in. Fuzzing tests whether the made implementation is robust and secure enough against hacking during the development cycle.

3.3 Potential problems

Although fuzzing is a good way to test in this case a protocol, It comes with certain problems or uncertainties:

- Did the device receive the sent inputs?
- Did the device receive the input as intended?
Is the sequence that is sent the same as the one that is received by the **DUT** or did it change along the way?
- If the input was received correctly by the **DUT**, was it actually used or was it just discarded?
- Does the device function correctly after receiving the inputs or not?
Verifying this as an external observer is nearly impossible if there is no way to access the internal structure of the device that is being tested. For example, An internal value changed because of the current input and this change is not visible from the outside. If this change has an impact on any of future tests or causes them to crash then it will be nearly impossible to replicate the result from test that failed.

These situations can be hard or impossible to distinguish. Consequently it is important to have an adequate number of tests and a log to see what was send and what the result was. This makes it easier to diagnose potential problem and verify the achieved results. [18]

Chapter 4: Components

4.1 Hardware

This chapter elaborates on the used hardware. This includes the two laptops that were used as well as the development board. Since the hardware specifics are generally out of the scope of this thesis, they will not be discussed in much detail.

4.1.1 Personal computers & DUT

Two laptops were used for this thesis. To interact with the Linux OS on the board an HP laptop (HP Elitebook 840) was provided by Keysight with Ubuntu 20.10 as the Operating System. It was used to run preliminary tests via the shell and to generate and compile the driver files which will be used in testing (see Chapter 5).

The second computer is a personal laptop running on Windows, which was originally not supposed to be used. Since a commitment was made to OpenTAP (see section 4.2), which is more user friendly on Windows, and to avoid excessive communication between multiple devices, the Windows laptop was used to run the fuzzing tests.

It is possible (and depending on the use case preferred) to run both Linux and Windows on the same computer. This was not preferred in this case because of the low processing power and storage space of the Windows laptop.

The device that was used as the client and fuzzed in this thesis is the *Xiaomi Poco X3 nfc* and is therefore called the **DUT**. The reason for choosing this device is that it is an Android device that was recently purchased. The android Android Debug Bridge (**adb**) tool which is further explained in section 4.1.3 makes it relatively easy to get information about the state of the device.

4.1.2 Development board

Openwifi supports multiple boards and configurations as mentioned on their Github page[5] and for each of them is an Operating System (**OS**) image provided. This image contains a Linux **OS** (Linaro), the necessary driver and software as well as the Openwifi drivers and software. This OS image can also be made using a prebuilt image from the Analog Devices Wiki as explained on the Openwifi readme in the *Build Openwifi Linux img from scratch* section.

For this thesis the ADRV9361-Z7035 board is used from the list 4.1 available on the Openwifi Github. It consists of the *ADRV9361-Z7035* System-on-Module (**SOM**), which is an Software-defined radio (**SDR**) 2x2 System-On-Module; and the *ADRV1CRR-BOB*, which is a prototyping platform that provides easy access to user I/O, Ethernet, USB, JTAG, and serial connections. [19] [20]

The ADRV9361-Z7035 **SOM** itself is made up of the *Analog Devices AD9361-BBCZ Inte-*

board_name	board combination	status	SD card img	Vivado license
zc706_fmcs2	Xilinx ZC706 dev board + FMCOMMS2/3/4	Done	32bit img	Need
zed_fmcs2	Xilinx zed board + FMCOMMS2/3/4	Done	32bit img	NO need
adrv9364z7020	ADRV9364-Z7020 + ADRV1CRR-BOB	Done	32bit img	NO need
adrv9361z7035	ADRV9361-Z7035 + ADRV1CRR-BOB/FMC	Done	32bit img	Need
zc702_fmcs2	Xilinx ZC702 dev board + FMCOMMS2/3/4	Done	32bit img	NO need
zcu102_fmcs2	Xilinx ZCU102 dev board + FMCOMMS2/3/4	Done	64bit img	Need
zcu102_9371	Xilinx ZCU102 dev board + ADRV9371	Future	Future	Need

Figure 4.1: Currently supported boards for Openwifi with the board highlighted in yellow, being the setup used in this thesis [5]

grated *RF Agile Transceiver™* and the *Xilinx Zynq XC7Z035-2L FBG676I AP* SoC which uses an ARM processor (Dual ARM® Cortex™-A9 MPCore™).

An **SDR** uses software for the modulation and demodulation of radio signals, which would traditionally be implemented in hardware. The advantage is that it is possible to transmit and receive a new form of radio protocol by running new software. [21][22]

The 2x2 **SOM** signifies the number of transmit and receive paths the transceiver has available. In this case the integrated *AD9361 RF Agile Transceiver* provides 2x2 wideband transmit- and receive paths from 70 MHz to 6 GHz. This makes the device ideal for a broad range of fixed- and mobile **SDR** applications.[23]

An **SOM** provides the core components of an embedded processing system on a single production-ready Printed Circuit Board (**PCB**). This includes the processing cores, the communication interfaces and the memory blocks. SOMs include only the necessary digital and analog components in a package that is made to be as small as possible, while retaining the flexibility to accommodate a wide array of applications. An **SOM** is similar to an System-on-a-Chip (**SOC**) in the aspect that they are designed with a special function in mind, contrary to a single-board computer such as a Raspberry Pi, for example. [24]

More in-depth information about the specifics of the board and its components can be found in the ADRV9361-Z7035 User Guide available on the Analog Devices Wiki. [25]

4.1.3 Hardware setup and connections

This section explains the setup of the hardware and how the devices are connected.

The computer (running on either Windows or Linux) is connected to the **DUT** (Xiaomi POCO X3 NFC running Android 11) via a USB-A to USB-C cable. Commands can be given to the Android device via adb commands, for example turning Wi-Fi on and off. The computer is also connected to the board via an Ethernet cable and the board can be operated by sending commands using an SSH connection from the computer. After the SSH connection has been established, the board can be directed using Linux shell commands since it is using an Linux **OS** itself.

SSH

SSH, also known as Secure Shell or Secure Socket Shell, is a network protocol that gives users a secure way to access a computer over an unsecured network. In addition to providing secure network services, SSH refers to the suite of utilities that implement the **SSH** protocol. Secure Shell provides strong password authentication and public key authentication. SSH also ensures encrypted data communication between two computers that connect over an open network, such as the internet.[26] Since SSH has been already used in the Openwifi documentation to connect to the board, it is used here as well.

adb

adb or Android Debug Bridge is a command line tool that allows interaction with an Android device. These commands can be used to install and debug apps or to monitor the device. [27]. A simple guide to install the adb tool on Windows and Linux is available in Appendix A.

To be able to use adb commands, it is necessary to enable USB debugging on the device. USB debugging can be found under developer options. These options are hidden by default and cannot be found under the regular settings. Generally, they can be made visible by tapping the build number (under *settings* → *about device*) seven times. A screen will pop-up as a reminder that personal information can be leaked this way (see Figure 4.2).

After ten seconds *ok* can be pressed and the developer settings can be found under *additional settings*. Once that is done, it's possible to enable *USB-debugging* by scrolling down to the debugging section. If it's not necessary to continuously use the debugging functionality, it's advised to enable the time-out option (see bottom of Figure 4.3). Doing this will ensure that the device will disable these options, in case it gets stolen or to have peace of mind in general.

By running the command 'adb devices', the daemon (a background process on the device that executes the given commands) automatically starts and scans for devices. If only one device is present, all commands will be sent to that device. If it is identified as *unauthorized* (see Figure 4.4) then it is still necessary to manually accept the **RSA**-key (see Figure 4.6), after which it will be shown as *device* instead of as *unauthorized* (see Figure 4.5). **RSA** is a widely used system for secure data transmission using public-key cryptography.

Since the device must be unlocked to acknowledge the **RSA**-key, it should ensure that only users of the device can enable USB-debugging. The setup to enable USB-debugging that is explained here is general and could differ slightly depending on the device.

In this case, **adb** commands will be used to monitor the Wi-Fi status of the device as well as the IP address assigned by the access point. An **adb** command will also be used to ping the IP of the access point to check if data communication between the **DUT** and the **AP** is possible.

Further explanation of the used commands can be found in section 5.2.3.

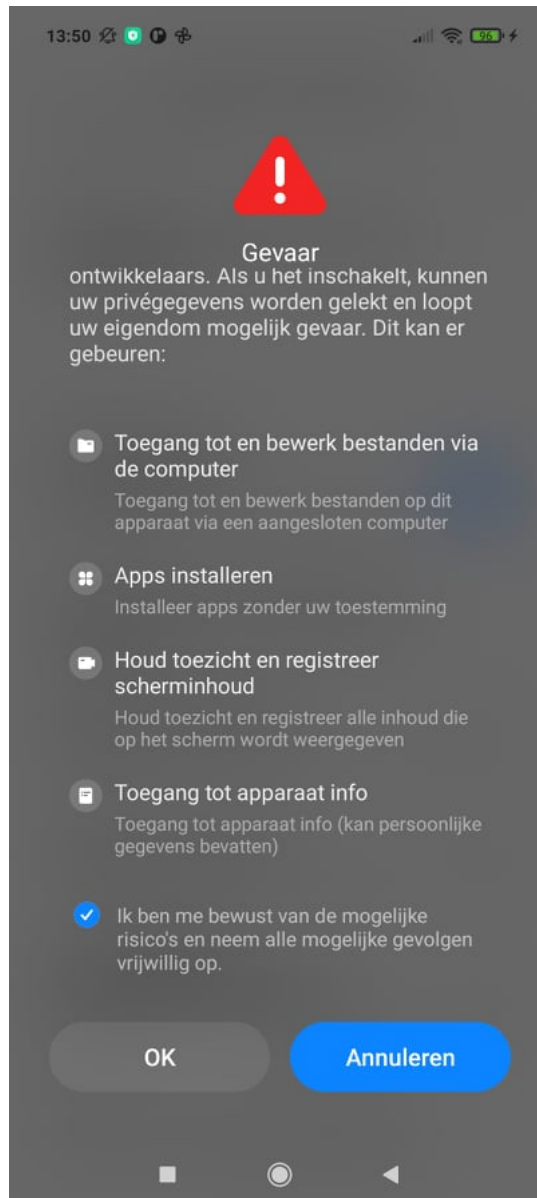


Figure 4.2: Reminder about the potential dangers that enabling the developer options can have and to what access is given

4.2 Software

4.2.1 OpenTAP

OpenTAP is software that was made in collaboration with established breweries to create their own recipes and was later developed as a product by Keysight Technologies to create automated solutions.

“OpenTAP is an Open Source project that allows for fast and easy development and execution of automated tests. It is built with simplicity, scalability and speed in mind, and is based on an extendable architecture that leverages .NET Core.” [28]

It is an program that is easy to understand and use (on Windows) to generate a testplan

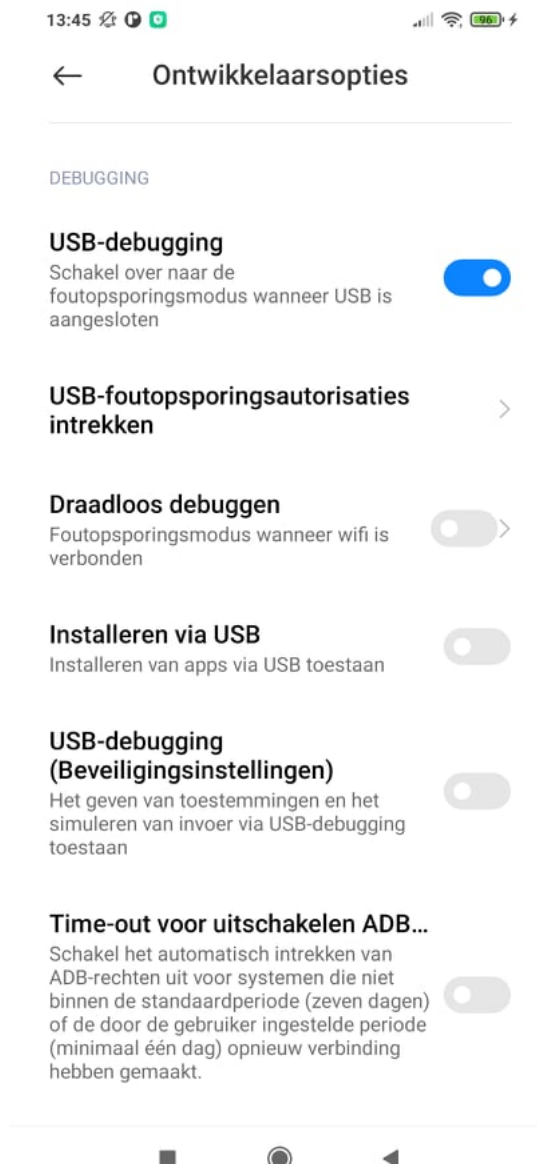


Figure 4.3: USB-debugging options that were enabled throughout the thesis

```
openwifi@openwifi-hp:~$ adb devices
List of devices attached
* daemon not running; starting now at tcp:5037
* daemon started successfully
e75ee026          unauthorized
```

Figure 4.4: Result of the 'adb devices' command on a Linux computer where the **RSA**-key was not accepted

that executes a sequence of steps. Originally the plan was to use OpenTAP on the Linux machine but problems with installation and the lack of a proper user interface. These factors combined with a testplan and test steps that would be made on a Windows machine anyway because of that lack of usability resulted in a change from a Linux computer to a Windows one.

OpenTAP offers a range of sequencing functionality for example delays, parameters sweeps,

```
c:\platform-tools>adb devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
e75ee026      device
```

Figure 4.5: Result of the 'adb devices' command on Windows computer where **RSA**-key was accepted



Figure 4.6: Pop-up screen asking if the user to allow the USB-debugging

the ability to run steps in parallel. Additional packages ,like the one needed for **SSH** [29], can be downloaded using the package manager. The package that allows the use of **adb** commands needed to be build manually using visual studio but might be available on the package manager by the time you are reading this. [29] There is also infrastructure in place that makes it possible to quickly develop plugin in case something specific is needed that in not provided. There is a developer guide available here [30] that explains how to install the Software Development Kit (**SDK**) for developers as well as information about how to make a custom plugin. For this thesis a custom plugin was made to make a teststep that copies a certain pregenerated driver from a directory to the board. The C# file for the custom plugin that was made can be found in Appendix **E**.

4.2.2 Wireshark

Wireshark is the world's foremost and widely-used network protocol analyzer. It lets the user see what's happening on his/her network at a detailed level and is in practice often the used standard across many commercial and non-profit enterprises, government agencies, and educational institutions. [31]

To monitor all traffic on a certain channel instead of just traffic between the device and the access points it is connected to, the Wireless Network Interface Controller (**WNIC**) of the computer will be set to monitor mode. Once monitor mode is enabled, it becomes possible to use Wireshark to see all the packets that are sent over the used channel (see Figure 4.7 for an example) which is in this case channel 44 or 5.22 GHz (using 802.11a). Since this is the frequency used by the Openwifi AP. As mentioned Openwifi also supports channel 11 using 802.11g but this was less stable because the connection got cut of sometimes and the AP would no longer show up in the list of available access points.

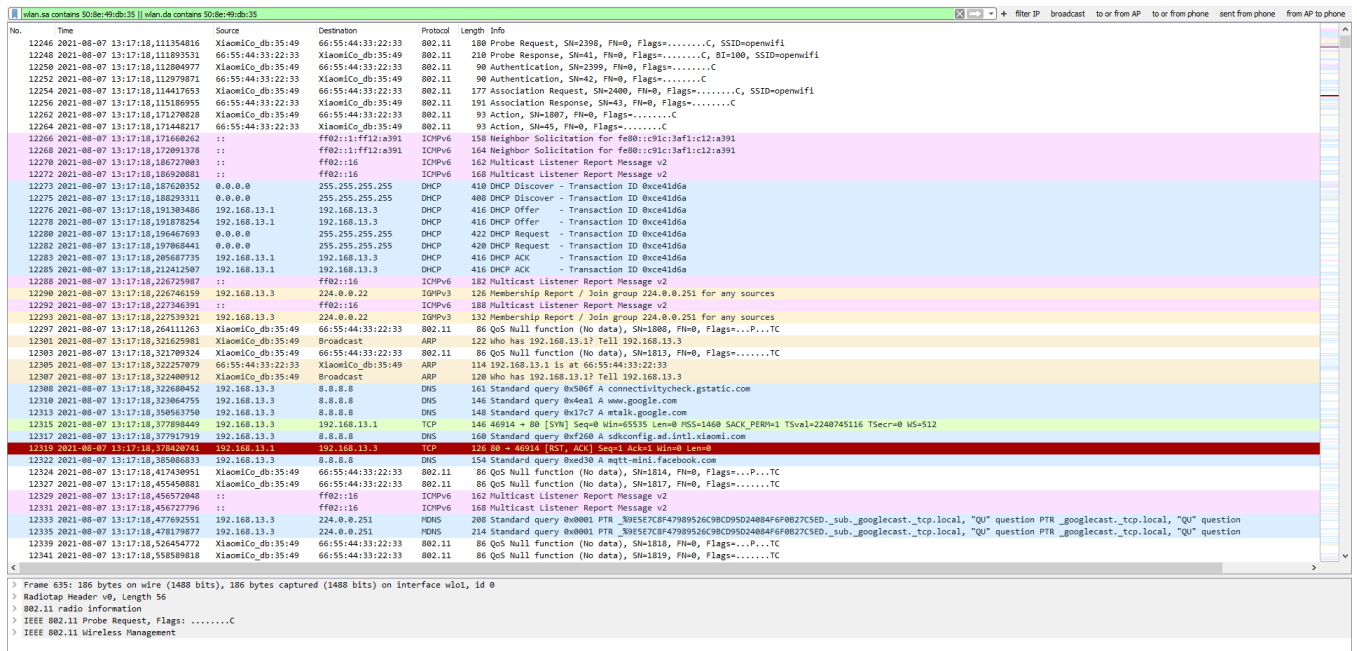


Figure 4.7: Part of a Wireshark output only showing the traffic between the board and the phone by using a display filter

monitor mode

What is monitor mode? It is one of the modes in which the **WNIC** of a computer can be set (if available). It allows the computer to monitor all traffic on a certain wireless channel without having to associate or connect to any access point. Only the Linux computer is run in monitor mode to asses and capture the traffic over the air, but the board has the ability to be run in monitor mode as well.

To asses if a Linux computer can run in monitor mode run the command 'iw list'. Among other information it will show the available modes and frequencies that can be use and set. The following images (Figure 4.8 and Figure 4.9) show the supported interface modes and frequencies from the 'iw list' output. After the frequency in MHz the channel number is displayed in square brackets.

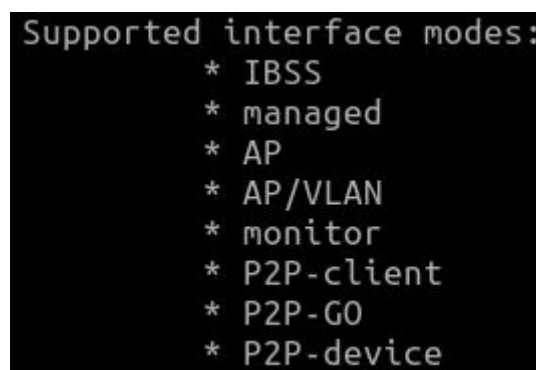


Figure 4.8: Section of the 'iw list' output that shows the supported interface modes for this device

Now to actually enable monitor mode the sequence of shell commands that are shown in Figure 4.11 can be used. 'sudo ifconfig wlan0 down' shuts down the wireless interface which in this

```

Frequencies:
* 5180 MHz [36] (22.0 dBm) (no IR)
* 5200 MHz [40] (22.0 dBm) (no IR)
* 5220 MHz [44] (22.0 dBm) (no IR)
* 5240 MHz [48] (22.0 dBm) (no IR)
* 5260 MHz [52] (22.0 dBm) (no IR, radar detection)
* 5280 MHz [56] (22.0 dBm) (no IR, radar detection)
* 5300 MHz [60] (22.0 dBm) (no IR, radar detection)
* 5320 MHz [64] (22.0 dBm) (no IR, radar detection)
* 5500 MHz [100] (22.0 dBm) (no IR, radar detection)
* 5520 MHz [104] (22.0 dBm) (no IR, radar detection)
* 5540 MHz [108] (22.0 dBm) (no IR, radar detection)
* 5560 MHz [112] (22.0 dBm) (no IR, radar detection)
* 5580 MHz [116] (22.0 dBm) (no IR, radar detection)
* 5600 MHz [120] (22.0 dBm) (no IR, radar detection)
* 5620 MHz [124] (22.0 dBm) (no IR, radar detection)
* 5640 MHz [128] (22.0 dBm) (no IR, radar detection)
* 5660 MHz [132] (22.0 dBm) (no IR, radar detection)
* 5680 MHz [136] (22.0 dBm) (no IR, radar detection)
* 5700 MHz [140] (22.0 dBm) (no IR, radar detection)
* 5720 MHz [144] (22.0 dBm) (no IR, radar detection)
* 5745 MHz [149] (22.0 dBm) (no IR)
* 5765 MHz [153] (22.0 dBm) (no IR)
* 5785 MHz [157] (22.0 dBm) (no IR)
* 5805 MHz [161] (22.0 dBm) (no IR)
* 5825 MHz [165] (22.0 dBm) (no IR)

```

Figure 4.9: Section of the 'iw list' output that shows the supported frequencies for the second band for this device

case is wlo1. If this command is not run before 'sudo iwconfig wlo1 mode monitor' than the output will most likely say that the interface is busy, which means that it is not currently possible to switch to monitor mode. It is possible to re-enable the interface by running 'sudo ifconfig wlo1 up'. After this the 'ifconfig' and 'iwconfig' commands are run to verify if the interface is set up properly.

The commands 'systemctl stop NetworkManager' and 'systemctl restart NetworkManager.service' will stop and restart the actual service responsible for managing the network. In my case the changes to the mode only got applied by shutting down and restarting the this NetworkManager service contrary to the information that was available. [32]

On the Openwifi Github there is a bashscript available, with the interface and the channel as variables, that is functionally identical to the previously mentioned commands as is shown in Figure 4.11.

Now that monitor mode is enabled, it is possible to see all packets send in the set frequency band but it is no longer possible to connect to internet via the same interface.

To go back to the default managed mode, it is possible to use the same lines of code by using 'managed' instead of 'monitor', as can be seen in Figure 4.12.

4.2.3 Opensource fuzzers

Another possible way to approach the fuzzing is by using an existing fuzzer or by making one using an open source fuzzing framework.

There are older opensource fuzzers available online like wifuzz [33] and wifuzzit [34] but they are not actively supported anymore. So building on top of that is not the best idea considering future development as well possible unovercomable roadblocks that might present themselves because of the lack of information.

```

openwifi@openwifi-hp:~$ systemctl stop NetworkManager
openwifi@openwifi-hp:~$ ./openwifi/user_space/monitor_ch.sh wlo1 44
wlo1
44
enp0s25: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.10.1 netmask 255.255.255.0 broadcast 0.0.0.0
    inet6 fe80::1f6e:9cf:d877:40ba prefixlen 64 scopeid 0x20<link>
    ether dc:4a:3e:03:61:cd txqueuelen 1000 (Ethernet)
    RX packets 410 bytes 68297 (68.2 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 476 bytes 56834 (56.8 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 20 memory 0xc1300000-c1320000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 13435 bytes 934743 (934.7 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 13435 bytes 934743 (934.7 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlo1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    unspec 4C-34-88-F9-8D-CD-30-3A-00-00-00-00-00-00-00-00 txqueuelen 1000 (UNSPEC)
    RX packets 47710 bytes 29655382 (29.6 MB)
    RX errors 0 dropped 19595 overruns 0 frame 0
    TX packets 7077 bytes 1372570 (1.3 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlo1 IEEE 802.11 Mode:Monitor Frequency:5.22 GHz Tx-Power=-2147483648 dBm
    Retry short limit:7 RTS thr:off Fragment thr:off
    Power Management:on

openwifi@openwifi-hp:~$ sudo systemctl restart NetworkManager.service

```

Figure 4.10: The given shell inputs and received outputs when enabling monitor mode using the bashscript provided by Openwifi

A general open source fuzzing framework that uses python and is still supported is Boofuzz[35]. This seemed promising in the beginning but it requires the entire protocol to be build using the frameworks building blocks for which nor the required amount of time nor knowledge was available.

This lead to the decision of modifying the working driver that the Openwifi board is using. In this way adjustments to the generation of the physical layer can be made without having to implement it completely. This comes with the drawback that all the sent packets will be adjusted but as will be disc used in chapter 6.4 this could be overcome by looking at what type of packet it is.

```

openwifi@openwifi-hp:~$ systemctl stop NetworkManager
openwifi@openwifi-hp:~$ sudo ifconfig wlo1 down
openwifi@openwifi-hp:~$ sudo iwconfig wlo1 mode monitor
openwifi@openwifi-hp:~$ sudo ifconfig wlo1 up
openwifi@openwifi-hp:~$ sudo iwconfig wlo1 channel 44
openwifi@openwifi-hp:~$ ifconfig
enp0s25: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.10.1 netmask 255.255.255.0 broadcast 0.0.0.0
    inet6 fe80::1f6e:9cf:d877:40ba prefixlen 64 scopeid 0x20<link>
    ether dc:4a:3e:03:61:cd txqueuelen 1000 (Ethernet)
    RX packets 412 bytes 68826 (68.8 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 501 bytes 62321 (62.3 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 20 memory 0xc1300000-c1320000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 15638 bytes 1079065 (1.0 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 15638 bytes 1079065 (1.0 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlo1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    unspec 4C-34-88-F9-8D-CD-30-3A-00-00-00-00-00-00-00-00 txqueuelen 1000 (UNSPEC)
    RX packets 63859 bytes 32445767 (32.4 MB)
    RX errors 0 dropped 35390 overruns 0 frame 0
    TX packets 7234 bytes 1406569 (1.4 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

openwifi@openwifi-hp:~$ iwconfig wlo1
wlo1 IEEE 802.11 Mode:Monitor Frequency:5.22 GHz Tx-Power=-2147483648 dBm
    Retry short limit:7 RTS thr:off Fragment thr:off
    Power Management:on

```

Figure 4.11: The given shell inputs and received outputs when enabling monitor mode


```

openwifi@openwifi-hp:~$ systemctl stop NetworkManager
openwifi@openwifi-hp:~$ sudo ifconfig wlo1 down
openwifi@openwifi-hp:~$ sudo iwconfig wlo1 mode managed
openwifi@openwifi-hp:~$ sudo ifconfig wlo1 up
openwifi@openwifi-hp:~$ sudo systemctl restart NetworkManager.service
openwifi@openwifi-hp:~$ ifconfig
enp0s25: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.10.1  netmask 255.255.255.0  broadcast 0.0.0.0
        inet6 fe80::1f6e:9cf:d877:40ba  prefixlen 64  scopeid 0x20<link>
        ether dc:4a:3e:03:61:cd  txqueuelen 1000  (Ethernet)
        RX packets 445  bytes 72135 (72.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 532  bytes 68696 (68.6 KB)
        TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0
        device interrupt 20  memory 0xc1300000-c1320000

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 19070  bytes 1333218 (1.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 19070  bytes 1333218 (1.3 MB)
        TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

wlo1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.0.187  netmask 255.255.255.0  broadcast 192.168.0.255
        inet6 2a02:1811:9c0b:d400:f24b:7efa:bd5e:bc41  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::fc0d:986:b7d2:52eb  prefixlen 64  scopeid 0x20<link>
        inet6 2a02:1811:9c0b:d400:6c47:e05:212e:372d  prefixlen 64  scopeid 0x0<global>
        ether 4c:34:88:f9:8d:cd  txqueuelen 1000  (Ethernet)
        RX packets 89734  bytes 54884986 (54.8 MB)
        RX errors 0  dropped 38257  overruns 0  frame 0
        TX packets 12723  bytes 2709391 (2.7 MB)
        TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0

openwifi@openwifi-hp:~$ iwconfig wlo1
wlo1      IEEE 802.11  ESSID:"telenet-E723C"
        Mode:Managed  Frequency:5.22 GHz  Access Point: 34:2C:C4:A7:FA:3F
        Bit Rate=300 Mb/s   Tx-Power=22 dBm
        Retry short limit:7   RTS thr:off   Fragment thr:off
        Power Management:on
        Link Quality=70/70  Signal level=-30 dBm
        Rx invalid nwid:0  Rx invalid crypt:0  Rx invalid frag:0
        Tx excessive retries:0  Invalid misc:41  Missed beacon:0

```

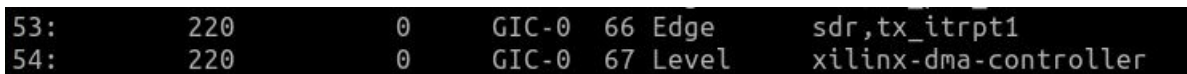
Figure 4.12: The given shell inputs and received outputs when going back to the 'managed' interface mode

Chapter 5: Testing

5.1 Precautions

There are certain uncertainties when testing wireless communications as mentioned in 3.3.

In this case Wireshark is used to make sure the AP was sending packets as well as the command 'cat /proc/interrupts' on the board. On a Linux machine, the file /proc/interrupts contains information about the interrupts in use and how many times the processor has been interrupted. [36]. If the integer indicated by the lines 53 and 54 (see Figure 5.1) keeps increasing between consecutive calls, this indicates that the board is transmitting packets.



53:	220	0	GIC-0	66 Edge	sdr,tx_itrpt1
54:	220	0	GIC-0	67 Level	xilinx-dma-controller

Figure 5.1: Interrupt 53 and 54 of the output from the command 'cat /proc/interrupts'

5.2 Setup

There are some things that need to be done before the testing can take place. The first is setting up the mentioned SSH connection. To be able to connect to the board via an SSH connection, it will be necessary to modify the Ethernet IP address of the computer used to 192.168.10.1/24.

On a Winux machine a temporary IP address can be set with the command 'sudo ip addr add 192.168.10.1/24 dev enp025' with 'enp025' is the interface name. This will have to be redone every time the computer is restarted but it is not a permanent change.

On a Windows 10 machine this can be done by going to Start ↵ Settings ↵ Network and Internet and click settings under Ethernet (cable must be connected). Under IP settings, by setting it to manual, it becomes possible to change it. Thus to enable the SSH connection change *IP address* to 192.168.10.1, *subnet prefix length* to 24 and *gateway* to 192.168.0.0 like shown in Figure 5.2. Contrary to the Linux approach these are permanent in the sense that it will have to set back to *Automatic (DHCP)* instead of manual to allow access to internet again through the same port.

There are undoubtedly other ways to accomplish the same on both Linux and Windows machines but these are what were used in this experiment.

To use the adb tool, the RSA-key needs to be accepted. This can be done before the test by running 'adb devices' or just after the test started because 'adb devices' is also part of the verification sequence.

Besides the accepting the RSA-key it is still necessary to disable automatic connection for all the visible APs. Automatic connection has to be enabled for the Openwifi AP by starting it up manually or by doing enabling it during the verification sequence. If Wi-Fi is enabled when the test starts, it is still necessary to manually disconnect from the current network as well keep

IP-instellingen bewerken

Handmatig

IPv4

☒ Aan

IP-adres

192.168.10.1

Lengte van voorvoegsel van subnet

24

Gateway

198.168.0.0

Figure 5.2: Ethernet IP settings used when connecting to the board on the Windows computer

the phone active, meaning not in a sleep state. So it will be necessary to disable the feature that automatically locks the phone.

To have additional verification of the packets sent, Wireshark can be used with monitor mode enabled (see 4.2.2). Display filters can be used to make it clearer if there is additional traffic going on. Be aware that these filters just limit what is shown on screen and not what is captured. Wireshark will have to be started manually before starting the test in OpenTAP.

The last thing to keep in mind is that the device used (in this case the phone) can't be too close to the antennas on the board. The reason for this is that automatic gain control used is not optimised and can cause the Analog-to-digital converter (ADC) to saturate and clip the signal. These clipped packets can still be detected but won't allow for a connection.[37]

5.2.1 Changes to the driver

As mentioned previously the easiest way to change the PPDU header of the physical layer is by changing the driver. The function that is responsible for generating the header bytes is called 'calc_phy_header()' and it can be found in the 'sdr.c' driver file. The modified version can be found in Appendix B.

The first modification of the 'calc_phy_header()' function was adding print statements after the generation of the bytes. These statements will print the byte values from the variables 'b0', 'b1' and 'b2' as well as the used rate and length. The rate is represented by the variable 'SIG_RATE' and the length by 'l.len'. The reserved and tail values printed will always be zero in regular operation and are not included in the bit operations used to generate the header bytes.

Now come the actual functional changes made to the driver. These lines are highlighted by setting them between the comment lines '//KEYWORD' and '//END_KEYWORD', so they can be found easily.

```

1 rate = SIG_RATE;      //0x00 to 0x0f or 0 to 15
2 reserved = 0x00;      //0x10 or 0x00
3 length = l_len;       //0 to 0xFF
4 tail = 0x00;          //0 to 63 or 0x0 to 0x3f
5 rate_changed = 0;

```

Code Listing 5.1: Field values are modified by the user or the program, with the value range in comments

```

1 tail = (tail & 0xF0) >> 4 | (tail & 0x0F) << 4;
2 tail = (tail & 0xCC) >> 2 | (tail & 0x33) << 2;
3 tail = (tail & 0xAA) >> 1 | (tail & 0x55) << 1;
4
5 if(rate != SIG_RATE){
6     rate = (rate & 0xF0) >> 4 | (rate & 0x0F) << 4;
7     rate = (rate & 0xCC) >> 2 | (rate & 0x33) << 2;
8     rate = (rate & 0xAA) >> 1 | (rate & 0x55) << 1;
9     rate = rate >> 4;
10 }

```

Code Listing 5.2: Lines of code responsible for mirroring the tail and rate bytes

The first four lines shown in Listing 5.1 (line 45 to 48 in Appendix B) represent variables that hold the userdefined value for the fields of the header. With 'rate' and 'length' being equal to the aforementioned 'SIG_RATE' and 'l_len' under normal operation. The 'reserved'- and 'tail' variables are set to zero as defined by the standard. These values will be changed by the userdefined value by a Python script. The Python script will be explained in section 5.2.2. The code itself is available in Appendix C. If the rate is changed then 'rate_changed' also has to be changed to not mirror the rate if the value is accidentally the same as the one stored in 'SIG_RATE', which is already mirrored.

The lines of code shown in Listing 5.2 consist of bit operations and have as result that the bits of the original byte are mirrored around the center efficiently. This means that in the resulting byte, the most significant bit is swapped with the least significant bit in the original byte. In other words this means that the first bit gets switched with the eighth, the second with the seventh and so on. The reason why this is necessary is that the board uses an ARM processor with little-endian memory where the least significant bit of a byte is the first bit and not the last, like in a big-endian architecture. This is implemented in the driver, so that the user does not have to take into account since most people will be used to the big-endian way of thinking, which similar to decimal numbers. This change was not necessary for the length because this had already been taken into account in the original calculation. The mirroring is also not executed for the reserved value for the simple reason that changing the original '0x08' to '0x10' (hexadecimal notation) as input is not really that different, as well as listing the possible value in a comment behind the variable.

The only changes made to the calculation of the three header bytes is the swapping of the 'rate' and 'length' variables with the original 'SIG_RATE' and 'l_len' as well as adding the 'reserved' and 'tail' variables. A comparison is shown in Listing 5.3. The generation of the separate length variables used ('len_2to0', 'len_10to3', 'len_msb') is not impacted in any way since these are generated based on the 'length' variable. These three variables represent the bits

```

1 // ORIGINAL
2 b0=SIG_RATE | (len_2to0 << 5) ;
3 b1 = len_10to3 ;
4 header_parity = gen_parity((len_msb << 16) | (b1<<8) | b0) ;
5 b2 = ( len_msb | (header_parity << 1) ) ;
6
7 // CHANGED
8 b0 = rate | reserved |(len_2to0 << 5);
9 b1 = len_10to3;
10 header_parity = gen_parity((len_msb << 16) | (b1 << 8) | b0);
11 b2 = (len_msb | (tail) | (header_parity << 1));

```

Code Listing 5.3: Difference between the original bytes and the modified bytes

from the 12 bitword that are present in each of the header bytes, shown in Figure 2.5.

The last modification is another set of 'printk' statements that allow the user to verify what was originally set and how it was changed compared to the first set of 'print' statements. The 'printk' statements will be visible in the output of the 'fosdem.sh' script. They can also be called by using 'sudo dmesg'. This command shows the messages stored in the kernel ring buffer which includes the added 'printk' statements. Since both the individual field values are shown as well as the generated bytes the print statements allow for manual verification of the changes to these four fields in the header as well as the bit operations separately. [38]. Be aware that the printed values are in little endian notation, meaning that if a one gets printed for the rate then the original value was 8.

5.2.2 Generating drivers

Automation was highly desired because manually adjusting the value(s), compiling the file and then copying it to the board for every test was a tedious task. The solution for generating multiple different drivers automatically was the combination of a Linux bash script and a Python script. Both scripts are available in Appendix C and D.

Bash script

The bash script is responsible for iterating the values that are used in the 'rate', 'length', 'reserved' and 'tail' variables. It is also responsible for compiling the driver and copying it to an directory with a specific name depending on values used to generate it.

Since the compilation instructions were only defined on Linux, they were executed on the laptop using Linux and manually transferred to the Windows laptop that is running the actual test. The transfer was usually done using a USB flash drive because that was the easiest way. Automating the file transfer could be achieved through direct Ethernet connection, which would still require manual interference switching the Ethernet cable to the board. A different solution would require sending the files over the internet using a router, which would make things unnecessarily complicated for this thesis.

The reason why the compiling is not done on the board itself is that there is not enough

```
1 $OPENWIFI_DIR/user_space/prepare_kernel.sh $OPENWIFI_DIR $XILINX_DIR ARCH_BIT
```

Code Listing 5.4: Command to prepare the kernel

storage to contain the necessary files: a copy of the Github and the Vivado Xilinx 2018.3 installation. A clone of the Github repository is also necessary and can be generated using the following command: 'git clone https://github.com/open-sdr/openwifi.git'.

Both are necessary steps as well as changing export commands for the install directories in the beginning of the bash script. The last step that is necessary is preparing the kernel by using the command from the subsequent Listing 5.4. [5]

To actually generate the files, multiple loops are used to get the right combination of fields and values. Multiple loops were used to generate the drivers. In these drivers only one of the four fields is set to a fixed value. One additional loop was used to generate drivers that have both the tail and length value changed. Not every possible driver was generated. This is expanded upon in section 5.2.4.

For relative changes to the length value a different variable and name was used because of the interaction with the Python script. Another custom step was created to be able to copy from these directories.

Figure 5.5 shows the process of generating the drivers, which used the even rate values. Default values must be reset between every loop to keep the other fields unchanged. The first step that is executed inside the loop, after the increase of the counter, is the generation of the 'sdr.c' file in the default directory. The counter is used to check the number of driver generated once the script is finished. The next step is compiling the driver and creating a new directory to contain it. This is necessary because the name of the driver must be 'sdr.ko', so they are differentiated by the name of the directory they are stored in. The last step copies both the original 'sdr.c' file as well as the compiled 'sdr.ko' file to the new directory. The 'sdr.c' file is not strictly necessary but provides a way to verify the field values.

Python script

The Python script, available in Appendix C, reads the content of the file 'sdr-original.c' into memory and changes the lines of the driver that contains the initialization of the field variables 'rate', 'reserved', 'length' and 'tail'. The script does this by reading the entire content from a template driver file called 'sdr-original.c' into memory and replacing the initialization strings with new initialisation strings that contain the userdefined input if there was one. The script then opens the actual 'sdr.c' file and overwrites the file with the content it has in memory. The script can have multiple arguments, two of which define the path names to the 'sdr-original.c' and 'sdr.c' files. The rest define one of the four header fields. Nothing is done if the arguments are the same as the default value and the argument parser will raise an error if the argument is outside of the defined choices. These are based on the possible values the bit word can contain. Additionally an argument was added to be able to generate length values that are x bigger or smaller than the default instead of using a fixed value. An exception will be raised if both length arguments are used at the same time.

```

1 # back to default values for next loop
2 export rate=-1
3 export reserved=-1
4 export length=-1
5 export tail=-1
6
7 #invalid rates from lowest to highest
8 for rate in 0 2 4 6 8 10 12 14
9 do
10     let "counter += 1"
11     #change sdr.c
12     python3 /home/openwifi/Desktop/MP/change_sdr.py --rate $rate
13
14     # compile driver using openwifi functionalities (see git)
15     $OPENWIFI_DIR/driver/make_all.sh $OPENWIFI_DIR $XILINX_DIR 32
16
17     #make new dir to contain sdr.ko
18     mkdir -p $DIR/"rate$rate reserved$reserved length$length tail$tail"
19
20     #copy sdr.ko to a directory:
21     cp $OPENWIFI_DIR/driver/sdr.ko $DIR/"rate$rate reserved$reserved
22     ↪ length$length tail$tail"
23     cp $OPENWIFI_DIR/driver/sdr.c $DIR/"rate$rate reserved$reserved
24     ↪ length$length tail$tail"
25
26 done

```

Code Listing 5.5: Loop generating and compiling the driver file 'sdr.c'

5.2.3 OpenTAP

Testplans made in OpenTAP have the '.TapPlan' extension and are essentially a sequence of steps. Before explaining the TapPlan that was used to automate the fuzzing process, instruments have to be added to be able to use the **SSH**- and **adb** plugins mentioned in 4.2.1 efficiently.

Two instruments were added. The first one is an **adb** instrument that contains the file path to the 'adb.exe' executable. The second instrument contains the IP address and password and is used to specify the **SSH** connection as an object instead of manually entering the same information for each **SSH** teststep. The settings for the **SSH** instrument are visible in Figure 5.3.

The screenshot shows the 'Bench Settings' dialog box with the 'Instruments' tab selected. On the left, under 'Dev board', the 'ADB' instrument is listed. The main area is titled 'SSH Instrument' and contains the following fields and options:

- Connection** (expanded):
 - Host Name: 192.168.10.122
 - Host Port: 22
 - User Name: root
 - Authentication Method: Password (dropdown menu)
 - Password: [masked with dots]
- Advanced** (expanded):
 - Lazy SSH connection: ☐
 - Lazy SCP connection: ☒

At the bottom left, there are '+' and '-' buttons for adding or removing instruments. At the bottom right, there are 'OK' and 'Cancel' buttons.

Figure 5.3: Settings for the **SSH** instrument in OpenTAP

The general TapPlan, shown in Figure 5.4, contains in essence only the 'Sweep Loop (Range)' teststeps. The first teststep 'killall hostapd' is used to stop the 'hostapd' service, which keeps the **AP** operating. This is to disconnect the **DUT** from the Openwifi **AP** if applicable. The second statement starts the 'adb daemon' if it had not been active already. The third teststep uses the 'adb shell svc wifi enable' command to enable the Wi-Fi if it had not been enabled already.

	Name	Verdict	Sweep \ Parameters	Duration	Flow	Type
○	<input checked="" type="checkbox"/> SSH Command killall hostapd					SSH \ SSH Command
○	<input checked="" type="checkbox"/> adb devices					UMA \ Android \ adb Command
○	<input checked="" type="checkbox"/> start with wifi enabled					UMA \ Android \ adb Command
⊕	<input type="checkbox"/> verification of correct operation					Flow Control \ Test Plan Reference
○	<input type="checkbox"/> Sweep Loop (Range) -> valid rate values					Flow Control \ Legacy \ Sweep Loop (Range)
⊕	<input checked="" type="checkbox"/> changed driver sequence					Flow Control \ Test Plan Reference
⊕	<input checked="" type="checkbox"/> original driver sequence					Flow Control \ Test Plan Reference
⊕	<input type="checkbox"/> Sweep Loop (Range) -> invalid rate values					Flow Control \ Legacy \ Sweep Loop (Range)
⊕	<input type="checkbox"/> Sweep Loop (Range) -> tail values					Flow Control \ Legacy \ Sweep Loop (Range)
⊕	<input type="checkbox"/> Sweep Loop (Range) -> length values (general)					Flow Control \ Legacy \ Sweep Loop (Range)
⊕	<input checked="" type="checkbox"/> Sequence relative L_len					Flow Control \ Sequence

Figure 5.4: General TapPlan used to run tests

The next sequence is a reference to a TapPlan that executes a test sequence that runs the **AP** using the original unmodified driver and that logs the results during the connection. The function of this step is to catch problems early on. An example of such a problem is the **adb** commands not working at all. In this case, testing can be aborted manually and the logs will show something went wrong from the beginning.

The following four teststeps are 'Sweep Loop (Range)' steps. They will change a user defined variable every iteration and rerun the child steps. The range between brackets in 'Sweep Loop (Range)' indicates that this loop will go from a defined start value to a defined stop value in a defined amount of steps. Examples are the even rate values, for which the loop will start at zero, stop at 14 and have a stepsize of two.

The child steps are the same for every 'Sweep Loop (Range)' step. Both of the child steps are references to different TapPlans that change the driver, run the **AP** and log the results. The only difference between them is that 'changed driver sequence' uses the values from the 'Sweep Loop' step to determine which directory contains the driver that must be copied. The default values are -1, which mean that the original value generated by the Openwifi implementation remain unchanged.

The sequence using the teststep to change the driver to a custom one is shown in Figure 5.5. First, 'adb devices' is run to verify the **adb** connection. The driver gets copied from a filename defined by the mentioned 'Sweep Loop' tests. The settings of this teststep are shown in Figure 5.6. The path contains the path name to the directory that contains the directories of the generated drivers. The other fields can be modified manually or iterated over by using the 'Sweep Loop (Range)' step.

The one-second delay and the sync command are in place to ensure that the copy instruction is finished. The next teststep called *parallel* executes all it child steps at the same time. This is necessary because once the teststep starts, the command './openwifi/fosdem.sh' is executed and the service it calls will continue, even if aborted in OpenTAP, until it is killed. This combined with the fact that commands cannot be run in the background means that the 'killall hostapd' must be executed in parallel to stop it, but before the 'hostapd' process is stopped, connection needs to be established and logged. At first a delay of 15 seconds is used to allow the **AP** to start broadcasting. Afterwards another delay of 25 seconds is used to allow the **DUT**, in this case the phone, ample time to establish a connection. At this point, assuming the connection is established, the outputs of multiple **adb** commands are logged. Before the 'webfsd' and 'hostapd' processes are stopped, the four calls '\cat\proc\interrupts' are used to log whether the board is actually transmitting since the 'tx_interrupt' will keep increasing.

Both 'fosdem.sh' and 'fosdem-11ag' can be used to set the board in a different mode, which uses a different subcategory of the 802.11 standard. 'fosdem.sh' uses 802.11n variant while 'fosdem-11ag.sh' can use both 802.11a and 802.11g versions. 11n Variant can use both the 2.4 Ghz and 5 Ghz bands and can theoretically achieve a higher bandwidth than the 11a and 11g versions but that would require the use of multiple antennas which is not the case. The 11a and 11g version can only transmit using the 5 Ghz and 2.4 Ghz frequencies respectively. [39]

	Name	Verdict	Sweep \ Parameters	Duration	Flow	Type
<input checked="" type="checkbox"/>	test sequence using changed driver					Flow Control \ Sequence
<input checked="" type="checkbox"/>	adb devices					UMA \ Android \ adb Command
<input checked="" type="checkbox"/>	copy changed driver to board					MyAwesomePlugin \ copy sdr.ko to board
<input checked="" type="checkbox"/>	Delay 1s					Basic Steps \ Delay
<input checked="" type="checkbox"/>	SSH Command sync					SSH \ SSH Command
<input checked="" type="checkbox"/>	parallel					Flow Control \ Parallel
<input checked="" type="checkbox"/>	start AP (fosdem.sh)					SSH \ SSH Command
<input checked="" type="checkbox"/>	test sequence					Flow Control \ Sequence
<input checked="" type="checkbox"/>	SSH Command cat /proc/interrupts (1)					SSH \ SSH Command
<input checked="" type="checkbox"/>	Delay to make sure AP is broadcasting					Basic Steps \ Delay
<input type="checkbox"/>	connect to AP using app					UMA \ Android \ adb Command
<input checked="" type="checkbox"/>	SSH Command cat /proc/interrupts (2)					SSH \ SSH Command
<input checked="" type="checkbox"/>	Delay to give device time to connect to AP					Basic Steps \ Delay
<input checked="" type="checkbox"/>	SSH Command cat /proc/interrupts (3)					SSH \ SSH Command
<input checked="" type="checkbox"/>	log adb parameters					Flow Control \ Test Plan Reference
<input checked="" type="checkbox"/>	SSH Command cat /proc/interrupts (4)					SSH \ SSH Command
<input checked="" type="checkbox"/>	SSH Command killall webfsd					SSH \ SSH Command
<input checked="" type="checkbox"/>	SSH Command killall hostapd					SSH \ SSH Command
<input checked="" type="checkbox"/>	adb shell dumpsys wifi					UMA \ Android \ adb Command
<input checked="" type="checkbox"/>	ping to verify disconnect					UMA \ Android \ adb Command

Figure 5.5: Test sequence using the changed driver

Test Step Settings	
length	-1
path	D:/test_drivers
rate	-1
reserved	-1
tail	-1

Figure 5.6: Settings of the custom test step that copies the driver to the board

The reference TapPlan responsible for logging the parameters is visible in Figure 5.7. The first command 'adb shell dumpsys connectivity' is mostly used to check the current state of the connection, but this command also holds information about the Domain Name System (DNS) address as well as the IP addresses of both the AP and the phone. The commands 'adb shell ip addr wlan0' and 'adb shell ifconfig wlan0' were used to double check the IP address assigned to the DUT. 'adb shell ping -c10 192.168.13.1' is used to check the actual connection by sending ten ping requests to the AP. 'adb shell ping -c10 192.168.13.1' could be used to ping a website to verify the connection, but this only works if the computer that is connected to the board via Ethernet is able to re-route the packets to another working connection. This feature is only defined for Linux in the Openwifi Github. Lastly, 'adb shell dumpsys wifi' is called. The command dumps a huge amount of unparsed logs about different internal services to the output. This command is the most useful for in-depth analysis of the internal workings but is sadly not well documented and hard to navigate. A summary of the parts of the output that seemed useful is available in Appendix F but due to the limited space it became very hard to read.

	Name	Verdict	Sweep \ Parameters	Duration	Flow	Type
<input checked="" type="checkbox"/>	log adb parameters					Flow Control \ Sequence
<input checked="" type="checkbox"/>	adb shell dumpsys connectivity					UMA \ Android \ adb Command
<input checked="" type="checkbox"/>	adb shell ip addr show wlan0					UMA \ Android \ adb Command
<input checked="" type="checkbox"/>	adb shell ifconfig wlan0					UMA \ Android \ adb Command
<input type="checkbox"/>	adb shell ping -c10 www.google.com					UMA \ Android \ adb Command
<input checked="" type="checkbox"/>	adb shell ping -c10 192.168.13.1					UMA \ Android \ adb Command
<input checked="" type="checkbox"/>	adb shell dumpsys wifi					UMA \ Android \ adb Command

Figure 5.7: TapPlan reference used to log the output from the adb commands

After the 'hostapd' process is killed, the 'adb shell dumpsys wifi' command is still run to have complete state information. The other **adb** commands do not provide useful information before the **AP** is started or after it is shut down. The last ping command is primarily used as a way to prompt OpenTAP to log the information of the 'dumpsys wifi' command and finish the execution of the TapPlan. For some reason the TapPlan timer continues running when this is the last command, even after everything is finished and logged.

At the bottom of the TapPlan shown in Figure 5.4 is sequence that was used to test length changes relative to the original length. Another custom teststep was made that is practically the as the others, with only the directory name changing. The OpenTAP step 'Sweep Loop' was not able to recognise the variables from the teststep and far therefor not possible to automate. 'Sweep Loop' would be more beneficial in this case if it is required to test length changes with various different stepsizes.

5.2.4 Remarks

Not every possible driver is made and compiled. There are multiple reasons of which the first one is that generating all these possible drivers obviously takes some time. However this time pales in comparison to the time it would take to test all of them with the automated setup that was made and used. On average the test duration is between 60 and 90 seconds per driver, so testing the changed driver and verifying the correct operation of the **DUT** would take between two and three minutes. The variation of 30 seconds is caused be the delay between failed ping requests, which is 15 seconds, combined with 2 retries equals 30 seconds.

To test all different drivers with variations of the the four fields (rate, reserved, length and tail) would take 31.92 years, assuming a test length of 60 seconds. 3 Bytes or 24 bit minus the parity bit, which would not be random, gives 2^{23} test drivers times 2 because of the verification after every test equals 2^{24} tests.

The second reason is the storage space required to save both the drivers and the logs. With the compiled driver file having a size of 458 kB and the the log size per driver being approximately 7482 kB would result in a required storage of at least 133.3 TB. This is without taking into account the storage space needed for the Wireshark log, which depends on the traffic on the network and the time spent collecting packets. This could be solved using cloud storage or a data center but it's not feasible to store this amount of data on a hard-drive. Another downside to large amount of data is that it is impossible to analyse it manually in a reasonable amount of time as a single person.

The third reason for not generating every driver is that the packets, generated using most of these driver, are unlikely to be recognised by the **DUT** if most of the fields are changed. The

packet is too different from the expected one.

Because of these 3 reasons, the drivers were only changed in 1 of the 4 fields for most tests. the value of the reserved value made no difference in the manual preliminary testing and was therefore excluded from the automatic tests to save time running and analysing.

Chapter 6: Results

The results from both the manual and automated tests were analysed manually. Manual analysis was necessary because there was too little time to make a robust program to take over this step. The manual analysis consisted of reading through the logged data and entering the relevant information into an excel file.

Changing the reserved value to 1, resulted in a driver operating in exactly the same way as the original driver. It does not seem to have any influence and ,consequently, no tests were automated using a reserved value of 1 to preserve time.

Additional manual tests were necessary because it was unclear if the results were correct. After it became clear that the results were indeed different, there no longer was time to fix the error, rerun the tests and analyse the results. The choice was made to continue using manual tests without logging the output of the 'adb commands'. In most cases only information from Wireshark, the output of 'fostdem.sh' and the result of the ping were analysed if connected.

The problem is presumably in the 'pscp' command used in the custom teststep (line 74 in Appendix D). 'pscp' is a tool that emulates the 'scp' tool available in Linux, which stands for secure copy and allows secure transferring of files between a local host and a remote host or between two remote hosts. [40]

There are embedded string characters (") used in the directory name. They are necessary to be able to use directory names with spaces. The used of these characters is most likely not entirely correct. The reason for the directory names with spaces originates from the bash script, which was only able to generate those directories if the delimiter was a space character. Otherwise the defined variables would not be used correctly.

The result of this is that the driver was not updated, which gives the same result for every test. The teststep used to work if an underscore was used instead of a space but was changed to further the automation aspect. The reason it was spotted to late is that there is error generated if the path is wrong or does not exist, then this makes it hard to spot if the visible outputs seem to align with the expectation.

6.1 Rate values

The automated tests that were done that used even four bit words to replace the rate value had results that did not differ significantly from the verification sequence. Most of the Internet Control Message Protocol (ICMP) (ping request and reply) packets were sent at speeds of 65 Mb/s or 72 Mb/s (reported by Wireshark) with a short guard interval. Less than 10% were with a long guard interval at a speed of 58.5 Mb/s.

For the test using the odd rate values, there was no significant difference. Most ICMP packets were sent with a rate of 65 Mb/s with a short guard interval and some packets were sent at a rate of 57.8 Mb/s with a short guard interval (reported by Wireshark). Again less than 10% of all the packets were sent with a long guard interval at a rate of 52 Mb/s.

The packets sent with the long guard intervals were only sent as the first ping reply but not always. This is true for both drivers modified with the odd and even rate values.

A momentary shutdown of functionality of the Wi-Fi on the phone was recorded during testing of the even rate values. Where the first automated test showed that the internal state was set to a `FRAMEWORK_DISCONNECT` for the rate value 4 (0100) with the listed reason being a `DISCONNECT_GENERIC`. This framework disconnect happened after an `ASSOCIATION_REJECTION_EVENT` on the verification step of the previously rate value (0010). The result is shown in Table 6.1 and 6.2, which are two parts of the same table split up into two separate tables. Certain columns were left out to preserve space since they added no further information.

The headers of the columns represent information extracted from different parts of the log. The *authenticated*, *associated* and *last state (fosdem)* come from the output of the 'fosdem.sh' command after the **AP** has been enabled.

The *curState during*, *StaEventList (latest)* and *BSSID* are from the output of the 'dumpsys wifi' command during a presumed connection to the **AP**. *state (dumpsys connectivity)* hold state information, which means in this case connected or not connected. the same is true for the *curState during* obtained from the 'dumpsys wifi' command.

The 'StaEventList' holds more detailed state information, of which only the latest state is shown, and the *BSSID* holds the **MAC** address of the **AP**. As can be seen in the table, the last byte of the **MAC** address is random every time 'fosdem.sh' is used to restart the **AP**. This conforms to the Openwifi implementation. The last column hold the result of the ping command and represents if there was a useful connection or not.

Driver	authenticated	associated	last state (fosdem)	state (dumpsys connectivity)
verification	yes	yes	accounting session	CONNECTED/CONNECTED
rate = 0	yes	yes	accounting session	CONNECTED/CONNECTED
verification	yes	yes	accounting session	CONNECTED/CONNECTED
rate = 2	yes	yes	accounting session	CONNECTED/CONNECTED
verification	no	no	none	none
rate = 4	no	no	none	none
verification	no	no	none	none
rate = 6	no	no	none	none
verification	no	no	none	none
rate = 8	no	no	none	none
verification	no	no	none	none
rate = 10	yes	yes	accounting session	CONNECTED/CONNECTED
verification	no	no	none	none
rate = 12	yes	yes	accounting session	CONNECTED/CONNECTED
verification	yes	yes	accounting session	CONNECTED/CONNECTED
rate = 14	yes	yes	authenticated	CONNECTED/CONNECTED
verification	yes	yes	authenticated	none

Table 6.1: Results first automated test regarding even rate values part 1

It was unclear what caused the disconnect and rerunning the tests for all these even rates values gave a different result that indicated that all of them worked fine, with no visible differences between the changed driver and the unmodified original.

The results of the manual tests, that were done after are shown in Table 6.3 for the ones using 'fosdem.sh' and in Table 6.4 for the ones using 'fosdem-1lag'.

curState during	StaEventList (latest)	BSSID	ping result (-c10)
ConnectedState	CMD_IP_CONFIGURATION_SUCCESSFUL	66:55:44:33:22:01	0% packet loss
ConnectedState	CMD_IP_CONFIGURATION_SUCCESSFUL	66:55:44:33:22:62	0% packet loss
ConnectedState	CMD_IP_CONFIGURATION_SUCCESSFUL	66:55:44:33:22:a7	0% packet loss
ConnectedState	CMD_IP_CONFIGURATION_SUCCESSFUL	66:55:44:33:22:e5	0% packet loss
DisconnectedState	ASSOCIATION_REJECTION_EVENT	any	Network is unreachable
DisconnectedState	FRAMEWORK_DISCONNECT	any	Network is unreachable
DisconnectedState	no update	no new entry	Network is unreachable
DisconnectedState	no update	no new entry	Network is unreachable
DisconnectedState	no update	no new entry	Network is unreachable
DisconnectedState	no update	no new entry	Network is unreachable
DisconnectedState	no update	no new entry	Network is unreachable
ConnectedState	CMD_IP_CONFIGURATION_SUCCESSFUL	66:55:44:33:22:54	0% packet loss
DisconnectedState	ASSOCIATION_REJECTION_EVENT	any	Network is unreachable
ConnectedState	CMD_IP_CONFIGURATION_SUCCESSFUL	66:55:44:33:22:8d	0% packet loss
ConnectedState	CMD_IP_CONFIGURATION_SUCCESSFUL	66:55:44:33:22:9d	0% packet loss
ConnectedState	CMD_IP_CONFIGURATION_SUCCESSFUL	66:55:44:33:22:3d	0% packet loss
ConnectedState	CMD_IP_CONFIGURATION_SUCCESSFUL	66:55:44:33:22:e3	20% packet loss (2e try)

Table 6.2: Results first automated test regarding even rate values part 2

The first three columns display the value the rate field was changed to in both decimal and binary representation as wells as the matching speeds defined in the standard. The next two columns display if the packets were picket up by Wireshark and which rate value was displayed in *802.11 radio information* section for the broadcast packets. An example taken from test where rate was set nine and 'fosdem.sh' was used is visible in Figure 6.1 where the *Data rate* is equal to 24.0 Mb/s.

```

> Frame 1: 216 bytes on wire (1728 bits), 216 bytes captured (1728 bits) on interface wlo1, id 0
> Radiotap Header v0, Length 56
▼ 802.11 radio information
  PHY type: 802.11a (OFDM) (5)
  Turbo type: Non-turbo (0)
  Data rate: 24,0 Mb/s
  Channel: 44
  Frequency: 5220MHz
  Signal strength (dBm): -75 dBm
  TSF timestamp: 12019105283
  > [Duration: 76µs]
> IEEE 802.11 Beacon frame, Flags: .....C
> IEEE 802.11 Wireless Management

```

Figure 6.1: 802.11 Radio information from a broadcast packet sent using 'fosdem.sh' and rate set to nine

The sixth and seventh column contain if the **AP** was visible to the phone and if it was able to connect to it. The last two columns display information about the ping replies received if a connection was established.

The drivers where the rate was set to an even value, which are not defined in the standard, are neither picked up by the phone nor by the computer using monitor mode and Wireshark. This implies that the data that comes after the signal symbol (see Figure 2.3) cannot be decode. The creators of Openwifi mentioned that there is a safety build into the **FPGA** implementation that changes the rate that does not conform to the standard to the lowest possible value that does, being 6 Mb/s (legacy mode) or 6.5 Mb/s (High Throughput (**HT**) or Greenfield mode) [41]. It was not mentioned if the header got changed or only the rate at which the packet is physically sent. It is safe to assume that the header was not changed because that would make

the packet the same as regular packets sent, but they were not picked up and displayed on the PC running Wireshark and the regular packets were.

If an odd value was used for the rate, then the phone was able to see the **AP** for both 'fosdem.sh' and 'fosdem-1lag.sh' but it was only able to connect when the latter was used. In this case the ping replies reported by Wireshark all had correct rates.

When 'fosdem.sh' was used the phone was not able to connect. When the output of the command 'dumpsys wifi' was inspected, the last state recorded in the 'StaEventsList' before disconnecting was 'CMD_IP.CONFIGURATION.LOST' with the listed reason being 'DEAUTH.LEAVING'. The phone was also stuck at configuring the **IP** for a few seconds before disconnecting. This implies that packets responsible for assigning the **IP** address did not get decoded correctly. The conclusion regarding the **IP** address is further reinforced by the malformed **DHCP** offers and reported in Wireshark. The shown error is visible in Figure 6.2 reports that the total length exceeds the packet length. Both the **DHCP** and the **BOOTP** are responsible for assigning the **IP** address. [42]

Regarding the rate value 13 that was able to connect to the phone when 'fosdem.sh' was used had some irregularities as well. When the board was pinged, the result would usually be 0% packet loss but most of the time the packets were not by Wireshark. The ping replies that did occasionally come up had a reported rate that varied from 6.5 Mb/s to 72.2 Mb/s with no consistency whatsoever. All of them seemed to be sent at different rates.

rate	bits	speed (Mb/s)	Wireshark	speed (broadcast) (Mb/s)	visible on phone	connect to phone	ping	speed (ping reply in Mb/s)
0	0000	none	no	none	no	no	no	none
1	0001	48	yes	48	yes	no ip	no	none
2	0010	none	no	none	no	no	no	none
3	0011	54	yes	54	yes	no ip	no	none
4	0100	none	no	none	no	no	no	none
5	0101	12	yes	12	yes	no ip	no	none
6	0110	none	no	none	no	no	no	none
7	0111	18	yes	18	yes	no ip	no	none
8	1000	none	no	none	no	no	no	none
9	1001	24	yes	24	yes	no ip	no	none
10	1010	none	no	none	no	no	no	none
11	1011	36	yes	36	yes	no ip	no	none
12	1100	none	no	none	no	no	no	none
13	1101	6	yes	6	yes	yes	0%	varying
14	1110	none	no	none	no	no	no	none
15	1111	9	yes	9	yes	no ip	no	none

Table 6.3: Results of manual tests that change the rate value using fosdem.sh

rate	bits	speed (Mb/s)	Wireshark	speed (broadcast) (Mb/s)	visible on phone	connect to phone	ping	speed (ping reply in Mb/s)
0	0000	none	no	none	no	no	no	none
1	0001	48	yes	48	yes	yes	0%	48
2	0010	none	no	none	no	no	no	none
3	0011	54	yes	54	yes	yes	10%	54
4	0100	none	no	none	no	no	no	none
5	0101	12	yes	12	yes	yes	20%	12
6	0110	none	no	none	no	no	no	none
7	0111	18	yes	18	yes	yes	0%	18
8	1000	none	no	none	no	no	no	none
9	1001	24	yes	24	yes	yes	10%	24
10	1010	none	no	none	no	no	no	none
11	1011	36	yes	36	yes	yes	0%	36
12	1100	none	no	none	no	no	no	none
13	1101	6	yes	6	yes	yes	10%	6
14	1110	none	no	none	no	no	no	none
15	1111	9	yes	9	yes	yes	0%	9

Table 6.4: Results of manual tests that change the rate value using fosdem-1lag.sh

6.2 Length values

The automated test that went over a set of fixed lengths was unusable. For unknown reasons the seemed to be aborted at different stages. After this happened an attempt was made to

dhcp						
No.	Time	Source	Destination	Protocol	Length	Info
8414	2021-08-14 15:37:18,857880338	0.0.0.0	255.255.255.255	DHCP	410	DHCP Discover - Transaction ID 0x5230771e
8416	2021-08-14 15:37:18,858070882	0.0.0.0	255.255.255.255	DHCP	408	DHCP Discover - Transaction ID 0x5230771e
8417	2021-08-14 15:37:18,858704971	192.168.13.1	192.168.13.3	DHCP	395	DHCP Offer - Transaction ID 0x5230771e[Malformed Packet]
8419	2021-08-14 15:37:18,859133184	192.168.13.1	192.168.13.3	DHCP	395	DHCP Offer - Transaction ID 0x5230771e[Malformed Packet]
9215	2021-08-14 15:37:26,909041570	0.0.0.0	255.255.255.255	DHCP	410	DHCP Discover - Transaction ID 0x5230771e
9217	2021-08-14 15:37:26,909847376	0.0.0.0	255.255.255.255	DHCP	408	DHCP Discover - Transaction ID 0x5230771e
9218	2021-08-14 15:37:26,909863291	192.168.13.1	192.168.13.3	BOOTP	225	Boot Reply[Malformed Packet]
9220	2021-08-14 15:37:26,910037604	192.168.13.1	192.168.13.3	BOOTP	140	Boot Reply[Malformed Packet]
11529	2021-08-14 15:37:53,404808461	0.0.0.0	255.255.255.255	DHCP	410	DHCP Discover - Transaction ID 0xee00a1db
11531	2021-08-14 15:37:53,405667681	0.0.0.0	255.255.255.255	DHCP	408	DHCP Discover - Transaction ID 0xee00a1db
> 802.11 radio information > IEEE 802.11 QoS Data, Flags:F.C > Logical-Link Control > Internet Protocol Version 4, Src: 192.168.13.1, Dst: 192.168.13.3 0100 = Version: 4 0101 = Header Length: 20 bytes (5) > Differentiated Services Field: 0x10 (DSCP: Unknown, ECN: Not-ECT) Total Length: 330 [Expert Info (Error/Protocol): IPv4 total length exceeds packet length (131 bytes)] [IPv4 total length exceeds packet length (131 bytes)] [Severity level: Error] [Group: Protocol] Identification: 0x0000 (0) > Flags: 0x00 Fragment Offset: 0 Time to Live: 128 Protocol: UDP (17) Header Checksum: 0x9e3e [validation disabled] [Header checksum status: Unverified] Source Address: 192.168.13.1 Destination Address: 192.168.13.3 > User Datagram Protocol, Src Port: 67, Dst Port: 68 Source Port: 67 Destination Port: 68 Length: 310 (bogus, payload length 111) [Expert Info (Error/Malformed): Bad length value 310 > IP payload length] [Bad length value 310 > IP payload length] [Severity level: Error] [Group: Malformed] Checksum: 0x480f [unverified] [Checksum status: Unverified] [Stream index: 1] > [Timestamps] UDP payload (103 bytes) > Dynamic Host Configuration Protocol [Malformed Packet: DHCP/BOOTP] [Expert Info (Error/Malformed): Malformed Packet (Exception occurred)] [Malformed Packet (Exception occurred)] [Severity level: Error] [Group: Malformed]						

Figure 6.2: Information reported by Wireshark for a Boot Reply packet sent using 'fosdem.sh' and a rate value set to nine

change this test so that length values used would be modified by increasing or decreasing it by a certain value instead of changing it to a set value. The goal of this approach is decreasing the sample size by testing a range of value around the original instead of going through all the 4096 possible value or a subset of them. This approach also makes the achieved results clearer by relating the results to the change made instead of the length itself, which could be different for each packet. This automatic test was not possible because the teststep 'Sweep Loop' (not 'Sweep Loop (Range)') was not able to recognise the variables of the custom teststep used.

The following was noticed using manual testing using the approach using relative length changes. Barely any of the tested packets with a faulty length value could be decoded by the phone. Decoding was only possible for the packets with a length value that that was increased slightly. If the length value went beyond an increase of five, the phone was no longer able to decode them if 'fosdem.sh' was used to run the AP. The results were less clear when the length was increased and 'fosdem-11ag.sh' was used. (Decreases gave the same results as 'fosdem.sh'). Increases up to eleven were tested and the phone was not able to connect to AP using the driver which increased the length with 9 and 10. The other values were not possible.

The packets were sent with a length increase of up to five were still reported as malformed by Wireshark but it was not able to identify the exact fault as can be seen in Figure 6.3.

```
> Frame 2: 217 bytes on wire (1736 bits), 217 bytes captured (1736 bits) on interface wlo1, id 0
> Radiotap Header v0, Length 56
> 802.11 radio information
> IEEE 802.11 Beacon frame, Flags: .....C
> IEEE 802.11 Wireless Management
▼ [Malformed Packet: IEEE 802.11]
  ▼ [Expert Info (Error/Malformed): Malformed Packet (Exception occurred)]
    [Malformed Packet (Exception occurred)]
    [Severity level: Error]
    [Group: Malformed]
```

Figure 6.3: Expanded analysis from Wireshark on a broadcast packet sent with a length value that was increased with one

For a small decrease, like minus one, Wireshark was able to pinpoint that the length was smaller even for smallest decrease possible, which is one, as can be seen in Figure 6.4. Wireshark rightly points out that the specified length is smaller than the actual length of the packet. But weirdly enough the same error occurs for decent (10, 20) to high (50, 100) increases and decreases in length.

```
> Frame 3: 215 bytes on wire (1720 bits), 215 bytes captured (1720 bits) on interface wlo1, id 0
> Radiotap Header v0, Length 56
> 802.11 radio information
> IEEE 802.11 Beacon frame, Flags: .....C
> IEEE 802.11 Wireless Management
▼ [Malformed Packet: IEEE 802.11: length of contained item exceeds length of containing item]
  ▼ [Expert Info (Error/Malformed): Malformed Packet (Exception occurred)]
    [Malformed Packet (Exception occurred)]
    [Severity level: Error]
    [Group: Malformed]
```

Figure 6.4: Expanded analysis from Wireshark on a broadcast packet sent with a length value that was decreased with one

Wireshark reports an additional tag error for both increases and decrease higher than 10. The tag on which the error occur varies but the error is always the same, “Tag Length is longer than remaining payload”.

6.3 Tail values

At first only the multiples of eight were used as tail values in the automated setup. The observed result was that a connection from the board with the DUT could be established with no report of a malformation in Wireshark. In hindsight this was not an optimal decision because multiples of eight have a difference of at most two bits with the original value, which is all zero’s. These values do not represent the full range of possible tail values enough.

This test was also verified afterwards because of the suspicion that there was something wrong. Manual tests that changed only the tail value gave the results shown in table 6.5. These results were obtained by trying every tail value and looking if the packets were broadcasted by using Wireshark, if the ‘openwifi’ SSID appeared on the phone and if that was the case was the phone able to connect to the AP.

tail	Wireshark	visible on phone	connect to phone	tail	Wireshark	visible on phone	connect to phone
0	yes	yes	yes	32	yes	no	no
1	yes	yes	yes	33	yes	no	no
2	yes	yes	yes	34	no	no	no
3	yes	yes	yes	35	no	no	no
4	yes	yes	yes	36	yes	no	no
5	yes	yes	yes	37	yes	no	no
6	yes	yes	yes	38	yes	yes	no auth.
7	yes	yes	yes	39	no	no	no
8	yes	no	no	40	no	no	no
9	yes	yes	yes	41	no	no	no
10	yes	yes	yes	42	yes	sometimes	no auth.
11	yes	yes	yes	43	yes	yes	yes
12	yes	yes	yes	44	yes	yes	yes
13	no	no	no	45	yes	yes	yes
14	no	no	no	46	yes	no	no
15	no	no	no	47	yes	yes	yes
16	yes	no	no	48	yes	no	no
17	no	no	no	49	yes	no	no
18	yes	no	no	50	yes	no	no
19	yes	sometimes	no auth.	51	yes	yes	no auth.
20	no	no	no	52	yes	no	no
21	yes	yes	yes	53	yes	no	no
22	yes	yes	yes	54	yes	no	no
23	yes	yes	yes	55	no	no	no
24	yes	yes	yes	56	yes	no	no
25	yes	yes	yes	57	yes	no	no
26	yes	no	no	58	yes	no	no
27	yes	no	no	59	yes	no	no
28	yes	no	no	60	no	no	no
29	yes	no	no	61	yes	no	no
30	yes	no	no	62	no	no	no
31	yes	sometimes	no auth.	63	yes	no	no

Table 6.5: Results of manual tests that change only the tail value

In this test there was no difference observed between the 'fosdem.sh' and the 'fosdem-11ag.sh' functions to run the **AP**. There were also no malformed packets reported in Wireshark using either function.

From Table 6.5 three case can be observed:

The first is were the phone was able to connect to the **AP**. In this case either there is error correcting code that is able to reconstruct the normal values or this tail value also gave the correct rate and length fields after decoding is done.

The second is where the phone is able to recognize the **AP** is available but is not able to connect to it. The reason reported by the output of the 'fosdem.sh' function was inability to authenticate. There were also multiple times were the 'openwifi' would not be continuously available. This is indicated by the sometimes used in Table 6.5.

In both the first and second case there was an variability in the time it took to connect to the **AP** of about zero to 30 seconds. The connection was sometimes established instantly and other times it took a long time and/or multiple retries. Disabling and re-enabling the Wi-Fi on the phone also made it connect when it was previously did want to for the same driver. The immediate reconnect could be attributed to internal timers or variables being reset upon disabling the Wi-Fi. When the phone did not want to connect to the Openwifi **AP**, it could still connect to an external Wi-Fi router.

The last case is where neither the phone nor the computer running Wireshark were able to detect that packets were being broadcasted. The board was still transmitting for these tail values because the tx interrupt kept increasing.

In test that were done afterwards, values were found that had a different result. The result was that neither the phone nor the PC running Wireshark could detect the packets using one of the following values: 30, 31, 60, 61, 62. While the Openwifi is transmitting since the tx interrupt kept increasing. The fact that even the computer in monitor mode was unable to detect these packets implies that the decoding of packets encoded with these tail values becomes impossible, at least for the devices used. The Openwifi board either wrongly encodes the packets with the given tail value or the used devices cannot decode packets that were encoded with said tail values and were thrown away or considered noise.

A small number of additional tests were done where the tail values from the first case were combined with a small increase in the length value, which both allow for connection separately. In these test, the **AP** was visible to the phone when both 'fosdem.sh' and 'fosdem-1lag.sh' were used but the phone was only able to connect when 'fosdem-1lag.sh' was used.

Conclusion

This thesis shows that it is possible to fuzz the physical layer of the IEEE 802.11 standard by modifying the implementation made by Openwifi. Automation was shown to be possible using OpenTAP and a combination of different scripts although there is still a problem that needs to be solved which most likely is a result of the spaces in the directory names used. The results from the manual tests as verification show that the physical layer did have an influence.

The results for the driver using the even rate values, which have no meaning and are not defined by the standard, implies that the safety built into the **FPGA** implementation either does not work correctly or only modifies the rate at which the packets are sent and not the rate value in the header that determines the encoding of the following data.

The drivers where an odd value for the rate was used (besides 13) and the **AP** was started using 'fosdem.sh', resulted in malformed **DHCP** and **BOOTP** being sent to the phone, which was not able to acquire an **IP** address and thus no usable connection was established. When the same drivers were used in combination with 'fosdem-11ag.sh' there was no problem connecting to the phone and ping replies were successfully received.

As for the changes in the length value, the only cases where a connection was established occurred when the length was only slightly larger than the original when 'fosdem.sh' was used. When 'fosdem-11ag.sh' was used some increases allowed for a connection while others did not and would require more testing to make a more definite judgement. There was no connection possible when decreases in length were used for both modes. These results could be also be attributed to the degree of successful decoding that was possible of the data that came after the header.

For the drivers using modified tail values there was no difference between the mode used ('fosdem.sh' or 'fosdem-11ag.sh'). The driver could be separated in 3 groups. One where there was no problem connecting, one where the phone was able to see the **AP** was active but was not able to connect to it and one where neither the computer running Wireshark nor the phone was able to sense the packets. These results can presumably be contributed to the degree of decoding was correct.

The test done using a changed reserved field did not differ from regular operation.

Instead of a fully useful fuzzing implementation the can be considered more of a proof of concept that modifying the physical layer is possible. The reason for saying this is that the tests did not have any visible effect on the functionality of the phone that could be attributed to the changed driver with absolute certainty. In conclusion, there is still uncertainty about the impact the changed drivers actually had on the phone. This could be partly solved by using being able to fully understand the contents of the 'dumpsys wifi' and 'dumpsys connectivity' commands as well as doing automatic analysis on logs where these commands and the others mentioned in this thesis are called a lot more frequently instead of just once or twice.

6.4 Possible future work

The first section of this chapter will discuss some optimization that can still be made to the setup, the written code and the used plugins. The second section will discuss the possible interesting parts that could be researched further or otherwise interesting topics that this thesis does not cover.

6.4.1 Possible optimizations

The first obstruction was caused by the use of two laptops instead of one, which made full automation very impractical and complex to implement. The only possibility is using an external router. The router should connect to the Linux laptop via Ethernet since a normal Wi-Fi connection is impossible when configured in 'monitor' mode. Additionally, it should be connected to the Wi-Fi. Of course this problem is due to the used hardware, so if different hardware is used, such a Windows desktop PC that has multiple Ethernet ports, would make things a little easier. The easiest solution for this problem is thus using just one Linux PC for both compiling and testing of the drivers. Then there is no file transfer necessary between multiple devices.

Another problem was due to the fact that the 'pscp' tool used to copy the 'sdr.ko' files to the board did not give an error if the given path name did not exist. The consequence of this is that multiple tests were done using the same driver instead of a different one leading to misleading results.

Reducing test lengths would also prove beneficial. The biggest contributor to this is the 25-second delay used to give the phone time to connect to the AP. Eliminating this is possible by using a custom test step that can initiate a connection with a SSID. It could be possible to connect to the Openwifi SSID by using the app from this source [43], although this would come with the problem that the available SSIDs must be read from somewhere. This could potentially be done from the WifiAssistant section, which shows the last three bytes of the MAC address of the available networks of the 'dumpsys wifi' output if the analysis is fast enough.

Both the bash file as well as the python file could be enhanced with the ability to use (more) arguments to make them more usable and flexible. A consequence of this would be that the whole existing automation process would have to be modified.

Lastly, the biggest problem was that manual analysis of the output files generated by the 'dumpsys' commands was necessary. The information needs to be parsed and cross referenced between multiple test cases to come to an accurate conclusion. The necessary parsing would not be universally applicable because the delimiter is different for different sections of the output. Analysis is further hindered because of the limited information available about the function of certain sections. Further manual analysis will most likely be necessary based on the preliminary analysis done by a computer. To make the further analysis easier the output information should be separated in individual files to make the outputs clearer and easier to cross reference. If the analysis was automatic, then it would also become viable to log the output of 'dumpsys connectivity' frequently to see the state transitions. As well as logging additional information from commands such as 'tcpdump' if the tests are run a Linux machine.

It would also be helpful to be able to limit the amount of information logged using the adb commands, which was not the case. The adb plugin would have to be modified to make this possible.

6.4.2 Future research

It is still possible to do further research using the Openwifi implementation, both on the hardware- and software level.

Multiple possibilities that still exist on a hardware level, meaning changing the **FPGA** implementation. Possible examples of this would be changing the preamble, the encoding or the carriers. A different approach could be disabling the built-in safety for the rate values to be able to test all possibilities, which is currently not possible.

There are multiple facets regarding the software that can still be explored, for instance modifying the guard interval timings to try and get priority. Maybe the **DUT** always looks at the shortest guard interval first in certain situation, which would allow starvation of the device by continuously sending these pockets.

Dynamically changing the channel in real time is another possibility. The execution of the frequency settings is the responsibility of the function `'ad9361_rf_set_channel()'`. This function also handles the Received Signal Strength Indicator (**RSSI**) compensation for the different frequencies as well as the **FPGA** configurations (Short Interframe Space (**SIFS**) and others) for the different bands.[5]

further building on this thesis is also possible by modifying the information given or received by the 'hostapd' service. The final received packet content is sent to the upper layer by the `'ieee80211_rx_irqsafe()'` function and the `'openwifi_tx()'` function gets the necessary information for the packet header from the struct `'ieee80211_hdr'`, which among other things holds information about the packet length and the Modulation Coding Scheme (**MCS**).

Extending on this thesis is also possible by modifying the service, signal tail and padding field, which are part of the data shown in Figure 2.5.

Besides the previously mentioned topics, it is also possible to use fuzzing on the Openwifi implementation, using an Openwifi board as both the client and **AP**.

Since the Openwifi implementation currently only supports the **OFDM** implementation of the standard regarding the physical layer. Since only **OFDM** is implemented means that only this can be tested. This in itself still leaves a lot of room for further research when more parts of the standard are implemented.

Appendix A: How to install the adb-tool

Download and extract the following file: [Android SDK platform ZIP file](https://dl.google.com/android/repository/platform-tools-latest-windows.zip) or use this url: <https://dl.google.com/android/repository/platform-tools-latest-windows.zip>. Once it's extracted it becomes possible to issue adb commands from the directory in which the file was extracted. So a change directory (cd) command will be necessary.

Appendix B: modified 'calc_phy_header()'

```
1  u32 calc_phy_header(u8 rate_hw_value, bool use_ht_rate, bool use_short_gi,
2  ↪ u32 len, u8 *bytes){
3      //u32 signal_word = 0 ;
4      u8  SIG_RATE = 0, HT_SIG_RATE;
5      u8      len_2to0, len_10to3, len_msb,b0,b1,b2, header_parity ;
6      u32 l_len, ht_len, ht_sig1, ht_sig2;
7
8      u8      rate, reserved, tail, rate_changed;
9      u32 length;
10
11     //
12     ↪ printk("rate_hw_value=%u\tuse_ht_rate=%u\tuse_short_gi=%u\tlen=%u\n",
13     ↪ rate_hw_value, use_ht_rate, use_short_gi, len);
14
15     // HT-mixed mode ht signal
16
17     if(use_ht_rate)
18     {
19         SIG_RATE = wifi_mcs_table_11b_force_up[4];
20         HT_SIG_RATE = rate_hw_value;
21         l_len = 24 * len / wifi_n_dbps_ht_table[rate_hw_value];
22         ht_len = len;
23     }
24     else
25     {
26         // rate_hw_value =
27         ↪ (rate_hw_value<=4?0:(rate_hw_value-4));
28         // SIG_RATE = wifi_mcs_table_phy_tx[rate_hw_value];
29         SIG_RATE = wifi_mcs_table_11b_force_up[rate_hw_value];
30         l_len = len;
31     }
32
33     len_2to0 = l_len & 0x07 ;
34     len_10to3 = (l_len >> 3 ) & 0xFF ;
35     len_msb = (l_len >> 11) & 0x01 ;
36
37     b0=SIG_RATE | (len_2to0 << 5) ;
38     b1 = len_10to3 ;
39     header_parity = gen_parity((len_msb << 16)| (b1<<8) | b0) ;
40     b2 = ( len_msb | (header_parity << 1) ) ;
41
42     printk("normal values:");
43     printk("rate=%u\treserved=%u\tlength=%u\ttail=%u\n", SIG_RATE, 0,
44     ↪ l_len, 0);
45     printk("bytes 1-3 in memory:");
```

```

41     printk("byte 0 = %u\t byte 1 = %u\t byte 2 = %u", b0, b1, b2);
42     printk("\n");
43
44     //KEYWORD
45     rate = SIG_RATE;           //0x00 to 0x0f or 0 to 15
46     reserved = 0x00;          //0x10 or 0x00
47     length = l_len;           //0 to 0xFFF
48     tail = 0x00;              //0 to 63 or 0x0 to 0x3f
49     rate_changed = 0;         // 0 or 1
50
51     // mirror tail bits to get expected result in header (tail and
52     ↪ rate need to be mirror to get expected header)
53     tail = (tail & 0xF0) >> 4 | (tail & 0x0F) << 4;
54     tail = (tail & 0xCC) >> 2 | (tail & 0x33) << 2;
55     tail = (tail & 0xAA) >> 1 | (tail & 0x55) << 1;
56
57     if(rate_changed == 1){
58         rate = (rate & 0xF0) >> 4 | (rate & 0x0F) << 4;
59         rate = (rate & 0xCC) >> 2 | (rate & 0x33) << 2;
60         rate = (rate & 0xAA) >> 1 | (rate & 0x55) << 1;
61         rate = rate >> 4;
62     }
63
64     len_2to0 = length & 0x07;
65     len_10to3 = (length >> 3) & 0xFF;
66     len_msb = (length >> 11) & 0x01;
67
68     b0 = rate | reserved | (len_2to0 << 5);
69     b1 = len_10to3;
70     header_parity = gen_parity((len_msb << 16) | (b1 << 8) | b0);
71     b2 = (len_msb | (tail) | (header_parity << 1));
72
73     printk("altered values:");
74     printk("rate=%u\treserved=%u\tlength=%u\ttail=%u\n", rate,
75     ↪ reserved, length, tail);
76     printk("bytes 1-3 in memory:");
77     printk("byte 0 = %u\t byte 1 = %u\t byte 2 = %u", b0, b1, b2);
78     printk("-----");
79     //END_KEYWORD
80
81     memset(bytes,0,16);
82     bytes[0] = b0 ;
83     bytes[1] = b1 ;
84     bytes[2] = b2 ;
85
86     // HT-mixed mode signal
87     if(use_ht_rate)
88     {
89         ht_sig1 = (HT_SIG_RATE & 0x7F) | ((ht_len << 8) &
90         ↪ 0xFFFF00);
91         ht_sig2 = 0x04 | (use_short_gi << 7);
92     }

```



```

89         ht_sig2 = ht_sig2 | (gen_ht_sig_crc(ht_sig1 | ht_sig2 <<
90         ↪ 24) << 10);
91
92         bytes[3] = 1;
93         bytes[8] = (ht_sig1 & 0xFF);
94         bytes[9] = (ht_sig1 >> 8) & 0xFF;
95         bytes[10] = (ht_sig1 >> 16) & 0xFF;
96         bytes[11] = (ht_sig2 & 0xFF);
97         bytes[12] = (ht_sig2 >> 8) & 0xFF;
98         bytes[13] = (ht_sig2 >> 16) & 0xFF;
99
100         return(HT_SIG_RATE);
101     }
102     else
103     {
104         //signal_word = b0+(b1<<8)+(b2<<16) ;
105         //return signal_word;
106         return(SIG_RATE);
107     }
108 }

```

Appendix C: change_sdr.py

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sat May  8 14:56:52 2021
4
5  @author: Steven
6  """
7
8  import sys
9  import argparse
10
11  def mirror_byte(b):
12      return int('{:08b}'.format(b)[::-1], 2)
13
14  def auto_int(x):
15      return int(x, 0)
16  # use type = lambda x: int(x,0) if u don't want to use this function
17
18  parser = argparse.ArgumentParser(description='input parameters Map art')
19
20
21  # allows decimal input, hex input (0xff) and binary input (0b1010)
22
23  parser.add_argument('--path_rd' , type=str , default =
24      ↪ '/home/openwifi/Desktop/MP/sdr-original.c', help='pathname to original
25      ↪ sdr.c that will be used as template but not changed')
26  parser.add_argument('--path_wr' , type=str , default =
27      ↪ '/home/openwifi/openwifi/driver/sdr.c', help='pathname to sdr.c that
28      ↪ will be changed and compiled')
29  parser.add_argument('--rate' , type=auto_int, default = -1,
30      ↪ choices=range(0x00, 0x0f +1) , help='rate field of PLCP header,
31      ↪ value between 0x00 and 0x0f')
32  parser.add_argument('--reserved', type=auto_int, default = -1,
33      ↪ choices=[0x00, 0x10] , help='reserved bit of PLCP header,
34      ↪ value 0x00 or 0x10')
35  parser.add_argument('--length' , type=auto_int, default = -1,
36      ↪ choices=range(0x000, 0xffff +1) , help='length field of PLCP header,
37      ↪ value between 0x000 and 0xffff')
38  parser.add_argument('--length_rel' , type=auto_int, default = 0,
39      ↪ choices=range(-0xffff, 0xffff +1) , help='value added to default l_len')
40  parser.add_argument('--tail' , type=auto_int, default = -1,
41      ↪ choices=range(0x00, 0x3f +1) , help='tail field of PLCP header,
42      ↪ value between 0x00 and 0x3f')
43
44  args = parser.parse_args()
```

```

33 if( args.length != -1 and args.length_rel != 0):
34     raise Exception('cannot change both length variables')
35
36 #fin = open('/home/openwifi/Desktop/MP/sdr-original.c', 'rt')
37 fin = open(args.path_rd, 'rt')
38
39 #read file contents to string (not the most eff. but the easiest)
40 data = fin.read()
41
42 #replace all occurrences of the required string
43 if(args.rate != -1):
44     data = data.replace('rate = SIG_RATE'      , f'rate = {args.rate}')
45     data = data.replace('rate_changed = 0'      , f'rate_changed = 1')
46 if(args.reserved != -1):
47     data = data.replace('reserved = 0x00'      , 'reserved = 0x10') #set
↪ reserved field to 0001000 then it does not have to be mirrored
48 if(args.length != -1):
49     data = data.replace('length = l_len'        , f'length = {args.length}')
50 if(args.tail != -1):
51     data = data.replace('tail = 0x00'          , f'tail = {args.tail}')
52
53 if(args.length != 0):
54     data = data.replace('length = l_len'        , f'length = l_len +
↪ {args.length}')
55
56 #close the input file
57 fin.close()
58
59 #open the input file in write mode
60 #fin = open('/home/openwifi/openwifi/driver/sdr.c', 'wt')
61 fin = open(args.path_wr, 'wt')
62 #override the input file with the resulting data
63 fin.write(data)
64 #close the file
65 fin.close()

```

Appendix D: generate_test_drivers.sh

```
1  #!/bin/bash
2
3
4
5  export OPENWIFI_DIR=/home/openwifi/openwifi
6  export XILINX_DIR=/tools/Xilinx
7  counter = 1
8
9  #make a directory where to store the testdirectories containing the
10 → changed sdr.sh
11 mkdir -p /home/openwifi/Desktop/MP/test_drivers
12 export DIR=/home/openwifi/Desktop/MP/test_drivers
13
14 #default values
15 export rate=-1
16 export reserved=-1
17 export length=-1
18 export tail=-1
19
20 #change to original driver and compile
21 python3 /home/openwifi/Desktop/MP/change_sdr.py
22 $OPENWIFI_DIR/driver/make_all.sh $OPENWIFI_DIR $XILINX_DIR 32
23 mkdir -p $DIR/"rate$rate reserved$reserved length$length tail$tail"
24 cp $OPENWIFI_DIR/driver/sdr.ko $DIR/"rate$rate reserved$reserved
25 → length$length tail$tail"
26 cp $OPENWIFI_DIR/driver/sdr.c $DIR/"rate$rate reserved$reserved
27 → length$length tail$tail"
28
29 #valid rates from lowest to highest
30 for rate in 13 15 5 7 9 11 1 3
31 do
32     let "counter += 1"
33     #change sdr.c
34     python3 /home/openwifi/Desktop/MP/change_sdr.py --rate $rate
35
36     #store the changed field
37     echo "rate$rate reserved$reserved length$length tail$tail">>
38     → /home/openwifi/Desktop/MP/list_subdirectories.txt
39
40     # compile driver using openwifi functionalities (see git)
41     $OPENWIFI_DIR/driver/make_all.sh $OPENWIFI_DIR $XILINX_DIR 32
42
43     #make new dir to contain sdr.ko
```

```

42     mkdir -p $DIR/"rate$rate reserved$reserved length$length
    ↪ tail$tail"
43
44     #copy sdr.ko to a directory:
45     cp $OPENWIFI_DIR/driver/sdr.ko $DIR/"rate$rate
    ↪ reserved$reserved length$length tail$tail"
46     cp $OPENWIFI_DIR/driver/sdr.c $DIR/"rate$rate reserved$reserved
    ↪ length$length tail$tail"
47
48 done
49
50 # back to default values for next loop
51 export rate=-1
52 export reserved=-1
53 export length=-1
54 export tail=-1
55
56 #invalid rates from lowest to highest
57 for rate in 0 2 4 6 8 10 12 14
58 do
59     let "counter += 1"
60     #change sdr.c
61     python3 /home/openwifi/Desktop/MP/change_sdr.py --rate $rate
62
63     #store the changed field
64     echo "rate$rate reserved$reserved length$length tail$tail">>
    ↪ /home/openwifi/Desktop/MP/list_subdirectories.txt
65
66     # compile driver using openwifi functionalities (see git)
67     $OPENWIFI_DIR/driver/make_all.sh $OPENWIFI_DIR $XILINX_DIR 32
68
69     #make new dir to contain sdr.ko
70     mkdir -p $DIR/"rate$rate reserved$reserved length$length
    ↪ tail$tail"
71
72     #copy sdr.ko to a directory:
73     cp $OPENWIFI_DIR/driver/sdr.ko $DIR/"rate$rate
    ↪ reserved$reserved length$length tail$tail"
74     cp $OPENWIFI_DIR/driver/sdr.c $DIR/"rate$rate reserved$reserved
    ↪ length$length tail$tail"
75
76 done
77
78 # back to default values for next loop
79 export rate=-1
80 export reserved=-1
81 export length=-1
82 export tail=-1
83
84 #varying lengths from lowest to highest

```

```

85  for length in {0..4096..64}
86  do
87      let "counter += 1"
88      #change sdr.c
89      python3 /home/openwifi/Desktop/MP/change_sdr.py --length
90          ↪ $length
91
92      #store the changed field
93      echo "rate$rate reserved$reserved length$length tail$tail">>
94          ↪ /home/openwifi/Desktop/MP/list_subdirectories.txt
95
96      # compile driver using openwifi functionalities (see git)
97      $OPENWIFI_DIR/driver/make_all.sh $OPENWIFI_DIR $XILINX_DIR 32
98
99      #make new dir to contain sdr.ko
100      mkdir -p $DIR/"rate$rate reserved$reserved length$length
101          ↪ tail$tail"
102
103      #copy sdr.ko to a directory:
104      cp $OPENWIFI_DIR/driver/sdr.ko $DIR/"rate$rate
105          ↪ reserved$reserved length$length tail$tail"
106      cp $OPENWIFI_DIR/driver/sdr.c $DIR/"rate$rate reserved$reserved
107          ↪ length$length tail$tail"
108
109  done
110
111  export rate=-1
112  export reserved=-1
113  export length=-1
114  export tail=-1
115
116  #invalid tail values from lowest to highest
117  for tail in {0..64..2}
118  do
119      let "counter += 1"
120      #change sdr.c
121      python3 /home/openwifi/Desktop/MP/change_sdr.py --tail $tail
122
123      #store the changed field
124      echo "rate$rate reserved$reserved length$length tail$tail">>
125          ↪ /home/openwifi/Desktop/MP/list_subdirectories.txt
126
127      # compile driver using openwifi functionalities (see git)
128      $OPENWIFI_DIR/driver/make_all.sh $OPENWIFI_DIR $XILINX_DIR 32
129
130      #make new dir to contain sdr.ko
131      mkdir -p $DIR/"rate$rate reserved$reserved length$length
132          ↪ tail$tail"
133
134      #copy sdr.ko to a directory:

```

```

128     cp $OPENWIFI_DIR/driver/sdr.ko $DIR/"rate$rate
    ↪ reserved$reserved length$length tail$tail"
129 cp $OPENWIFI_DIR/driver/sdr.c $DIR/"rate$rate reserved$reserved
    ↪ length$length tail$tail"
130
131 done
132
133 #default values
134 export rate=-1
135 export reserved=-1
136 export length=-1
137 export tail=-1
138
139 #relative lengths
140 for val in -100 -50 -20 -10 10 20 50 100
141 do
142     let "counter += 1"
143     #change sdr.c
144     python3 /home/openwifi/Desktop/MP/change_sdr.py --length_rel
    ↪ $val --tail $tail
145
146     #store the changed field
147     echo "rate$rate reserved$reserved length_rel$val tail$tail">>
    ↪ /home/openwifi/Desktop/MP/list_subdirectories.txt
148
149     # compile driver using openwifi functionalities (see git)
150     $OPENWIFI_DIR/driver/make_all.sh $OPENWIFI_DIR $XILINX_DIR 32
151
152     #make new dir to contain sdr.ko
153     mkdir -p $DIR/"rate$rate reserved$reserved length_rel$val
    ↪ tail$tail"
154
155     #copy sdr.ko to a directory:
156     cp $OPENWIFI_DIR/driver/sdr.ko $DIR/"rate$rate
    ↪ reserved$reserved length_rel$val tail$tail"
157     cp $OPENWIFI_DIR/driver/sdr.c $DIR/"rate$rate reserved$reserved
    ↪ length_rel$val tail$tail"
158
159 done
160
161 echo "$counter files generated"

```

Appendix E: Scp_Driver_To_Board.cs

The following is a C# file of the teststep used to copy the changed driver to the board. It was build using Visual Studio and the generated 'dll' file was transferred to the OpenTAP directory.

```
1  // Author: Steven H
2  // Copyright: Copyright 2021 Keysight Technologies
3  //           You have a royalty-free right to use, modify, reproduce
4  //           and distribute the sample application files (and/or any modified version)
5  //           in any way you find useful, provided that you agree that Keysight
6  //           Technologies has no warranty, obligations or liability for any sample
7  //           application files.
8  using OpenTap;
9  using System;
10 using System.Diagnostics;
11 using System.Collections.Generic;
12 using System.ComponentModel;
13 using System.Linq;
14 using System.Text;
15
16 namespace MyAwesomePlugin
17 {
18     [Display("copy sdr.ko to board", Group: "MyAwesomePlugin",
19     ↪ Description: "execute given Windows shell command")]
20     public class Scp_Driver_To_Board : TestStep
21     {
22         #region Settings
23
24         [Display("path", Description: "The path to the directory
25     ↪ containing the subdirectories")]
26         public string path { get; set; }
27
28         // should be inheritend form sweep loop or set manually
29         [Display("rate", Description: "inherited value from sweep loop")]
30         public int rate { get; set; }
31
32         [Display("reserved", Description: "inherited value from sweep
33     ↪ loop")]
34         public int reserved { get; set; }
35
36         [Display("length", Description: "inherited value from sweep
37     ↪ loop")]
38         public int length{ get; set; }
```



```

34     [Display("tail", Description: "inherited value from sweep loop")]
35     public int tail { get; set; }
36     //
37
38
39     //[Display("Command", Description: "The command to give to the
40     ↪ Windows command prompt")]
41     //public string Command { get; set; }
42
43     #endregion
44
45     public static void test_values(int rate, int reserved, int length,
46     ↪ int tail)
47     {
48         if (rate < -1 || rate > 0x0f)
49         {
50             throw new ArgumentOutOfRangeException("rate value must be
51             ↪ in range [0, 0x0f]");
52         }
53
54         if (reserved < -1 || reserved > 1)
55         {
56             throw new ArgumentOutOfRangeException("reserved value must
57             ↪ be 0 or 1");
58         }
59
60         if (length < -1 || length > 0xffff)
61         {
62             throw new ArgumentOutOfRangeException("length value must
63             ↪ be in range [0, 0xffff]");
64         }
65
66         if (tail < -1 || tail > 0x3f)
67         {
68             throw new ArgumentOutOfRangeException("tail value must be
69             ↪ in range [0, 0x3f]");
70         }
71     }
72
73     public static void ExecuteCommand(string path, int rate, int
74     ↪ reserved, int length, int tail)
75     {
76
77         test_values(rate, reserved, length, tail);
78
79         System.Diagnostics.Process process = new
80         ↪ System.Diagnostics.Process();
81         System.Diagnostics.ProcessStartInfo startInfo = new
82         ↪ System.Diagnostics.ProcessStartInfo();
83         //startInfo.WindowStyle =
84         ↪ System.Diagnostics.ProcessWindowStyle.Hidden;
85         startInfo.FileName = "cmd.exe";

```

```

74         startInfo.Arguments = "/C pscp -scp -pw openwifi \"" + path +
↵         "/rate"+ rate + " reserved" + reserved + " length" + length
↵         + " tail" + tail + "/sdr.ko\"
↵         root@192.168.10.122:openwifi/";
75         startInfo.UseShellExecute = false;
76         startInfo.RedirectStandardOutput = true;
77         process.StartInfo = startInfo;
78         process.Start();
79
80         while (!process.StandardOutput.EndOfStream)
81         {
82             Log.Info(process.StandardOutput.ReadLine());
83         }
84     }
85
86     public Scp_Driver_To_Board()
87     {
88         Name = "Send scp Command";
89         rate = -1;
90         reserved = -1;
91         length = -1;
92         tail = -1;
93         path = "D:/test_drivers";
94     }
95
96     public override void Run()
97     {
98         try{
99             ExecuteCommand(path, rate, reserved, length, tail);
100             UpgradeVerdict(Verdict.Pass);
101         }
102         catch (Exception e)
103         {
104             Log.Warning(e.Message);
105             UpgradeVerdict(Verdict.Error);
106         }
107     }
108 }
109 }
110 }

```

Appendix F: Summary of information from an example log

The following text contains some of the log information that seemed useful for a test using the default driver. From the outputs of 'adb shell dumpsys connectivity' and 'adb shell dumpsys wifi' were only the most recent records included and only the information that seems useful for the application of this thesis. This limitation is used because of the sheer size of the output from these commands as well as the amount of break lines necessary to display it here. The commands of which the output is displayed are at the far left without timestamp. The triple dots are used to indicate that multiple lines were omitted in this section of the output.

```
adb devices:
2021-08-02 14:21:54.775586 ; ADB ; Debug ; List of devices attached
2021-08-02 14:21:54.775586 ; ADB ; Debug ; e75ee026 device

=> adb setup correctly

2021-08-02 14:22:00.325265 ; TestPlan ; Information ; "setup sequence to verify correct
↳ operation of AP \ parallel \ start AP" started.

=> AP started at 14:22:00
=> started broadcasting at 14:22:05 (source: wireshark capture)

assigned IP:

adb shell ip addr show wlan0:

2021-08-02 14:22:46.277002 ; ADB ; Debug ; 30: wlan0:
↳ <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 3000
2021-08-02 14:22:46.277002 ; ADB ; Debug ; link/ether 50:8e:49:db:35:49 brd
↳ ff:ff:ff:ff:ff:ff
2021-08-02 14:22:46.277002 ; ADB ; Debug ; inet 192.168.13.3/24 brd
↳ 192.168.13.255 scope global wlan0
2021-08-02 14:22:46.277002 ; ADB ; Debug ; valid_lft forever
↳ preferred_lft forever
2021-08-02 14:22:46.277002 ; ADB ; Debug ; inet6
↳ fe80::c91c:3af1:c12:a391/64 scope link stable-privacy
2021-08-02 14:22:46.277002 ; ADB ; Debug ; valid_lft forever
↳ preferred_lft forever

adb shell ifconfig wlan0:

2021-08-02 14:22:46.730970 ; ADB ; Debug ; wlan0 Link encap:UNSPEC
↳ Driver icnss
2021-08-02 14:22:46.730970 ; ADB ; Debug ; inet addr:192.168.13.3
↳ Bcast:192.168.13.255 Mask:255.255.255.0
2021-08-02 14:22:46.730970 ; ADB ; Debug ; inet6 addr:
↳ fe80::c91c:3af1:c12:a391/64 Scope: Link
2021-08-02 14:22:46.730970 ; ADB ; Debug ; UP BROADCAST RUNNING
↳ MULTICAST MTU:1500 Metric:1
2021-08-02 14:22:46.730970 ; ADB ; Debug ; RX packets:1146 errors:0
↳ dropped:0 overruns:0 frame:0
2021-08-02 14:22:46.730970 ; ADB ; Debug ; TX packets:2229 errors:0
↳ dropped:0 overruns:0 carrier:0
```

```

2021-08-02 14:22:46.730970 ; ADB ; Debug ; collisions:0
↳ txqueuelen:3000
2021-08-02 14:22:46.730970 ; ADB ; Debug ; RX bytes:756097 TX
↳ bytes:367199

```

adb shell dumpsys connectivity:

Current Networks:

```

2021-08-02 14:22:47.232956 ; ADB ; Debug ; NetworkAgentInfo{ ni{[type:
↳ WIFI[], state: CONNECTED/CONNECTED, reason: (unspecified), extra: , failover: false,
↳ available: true, roaming: false]} network{274} nethandle{1180226736141}
↳ lp{[InterfaceName: wlan0 LinkAddresses: [ fe80::c91c:3af1:c12:a391/64,192.168.13.3/24 ]
↳ DnsAddresses: [ /8.8.8.8,/4.4.4.4 ] Domains: mydomain.example MTU: 0 ServerAddress:
↳ /192.168.13.1 TcpBufferSizes: 524288,1048576,8808040,262144,524288,6710886 Routes: [
↳ fe80::/64 -> :: wlan0 mtu 0,192.168.13.0/24 -> 0.0.0.0 wlan0 mtu 0,0.0.0.0/0 -> 192.168.13.1
↳ wlan0 mtu 0 ]}} nc{[ Transports: WIFI Capabilities:
↳ NOT_METERED&INTERNET&NOT_RESTRICTED&TRUSTED&NOT_VPN&NOT_ROAMING&FOREGROUND&NOT_CONGESTED&NOT_SUSPENDED
↳ LinkUpBandwidth>=58000Kbps LinkDnBandwidth>=58000Kbps SignalStrength: -67 OwnerUid: 1000
↳ SSID: "openwifi" RequestorUid: -1 RequestorPackageName: null]} Score{60}
↳ everValidated{false} lastValidated{false} created{true} lingering{false}
↳ explicitlySelected{false} acceptUnvalidated{false} everCaptivePortalDetected{false}
↳ lastCaptivePortalDetected{false} partialConnectivity{false} acceptPartialConnectivity{true}
↳ clat{mBaseIface: null, mIface: null, mState: IDLE} }

```

adb shell dumpsys wifi:

```

2021-08-02 14:22:48.003894 ; ADB ; Debug ; Wi-Fi is enabled

2021-08-02 14:22:48.003894 ; ADB ; Debug ; WifiController:
2021-08-02 14:22:48.004884 ; ADB ; Debug ; rec[83]: time=08-02 14:21:53.060
↳ processed=DisabledState org=DisabledState dest=EnabledState what=155656(0x26008)
2021-08-02 14:22:48.004884 ; ADB ; Debug ; rec[84]: time=08-02 14:22:18.601
↳ processed=EnabledState org=EnabledState dest=<null> what=155656(0x26008)
2021-08-02 14:22:48.004884 ; ADB ; Debug ; curState=EnabledState

2021-08-02 14:22:48.004884 ; ADB ; Debug ; WifiClientModeManager:
2021-08-02 14:22:48.004884 ; ADB ; Debug ; total records=3
2021-08-02 14:22:48.004884 ; ADB ; Debug ; rec[0]: time=08-02 14:21:53.297
↳ processed=IdleState org=IdleState dest=ScanOnlyModeState what=0(0x0)
2021-08-02 14:22:48.004884 ; ADB ; Debug ; rec[1]: time=08-02 14:21:53.438
↳ processed=StartedState org=ScanOnlyModeState dest=ConnectModeState what=2(0x2)
2021-08-02 14:22:48.004884 ; ADB ; Debug ; rec[2]: time=08-02 14:22:18.601
↳ processed=ConnectModeState org=ConnectModeState dest=<null> what=2(0x2)
2021-08-02 14:22:48.004884 ; ADB ; Debug ; curState=ConnectModeState

2021-08-02 14:22:48.004884 ; ADB ; Debug ; WifiClientModeImpl:
...
2021-08-02 14:22:48.007881 ; ADB ; Debug ; rec[998]: time=08-02 14:22:43.704
↳ processed=L2ConnectedState org=ConnectedState dest=<null> what=CMD_RSSI_POLL screen=on 174 0
↳ "openwifi" 66:55:44:33:22:ef rssi=-67 f=5220 sc=60 link=57 tx=1,6, 1,0, 0,0 rx=0,7 bcn=1
↳ [on:64 tx:1 rx:2 period:805] from screen [on:14002 period:780224] score=60
2021-08-02 14:22:48.007881 ; ADB ; Debug ; rec[999]: time=08-02 14:22:45.489
↳ processed=L2ConnectedState org=ConnectedState dest=<null> what=CMD_ONESHOT_RSSI_POLL
↳ screen=on 0 0 "openwifi" 66:55:44:33:22:ef rssi=-67 f=5220 sc=60 link=57 tx=1,6, 1,0, 0,0
↳ rx=0,7 bcn=1 [on:0 tx:0 rx:0 period:1785] from screen [on:14002 period:782009] score=60
2021-08-02 14:22:48.007881 ; ADB ; Debug ; curState=ConnectedState

2021-08-02 14:22:48.007881 ; ADB ; Debug ; SupplicantStateTracker:
...
2021-08-02 14:22:48.007881 ; ADB ; Debug ; rec[44]: time=08-02 14:21:53.565
↳ processed=DefaultState org=CompletedState dest=UninitializedState what=131183(0x2006f)
2021-08-02 14:22:48.007881 ; ADB ; Debug ; rec[45]: time=08-02 14:22:40.633
↳ processed=DefaultState org=UninitializedState dest=HandshakeState what=147462(0x24006)

```

```

2021-08-02 14:22:48.007881 ; ADB ; Debug ; rec[46]: time=08-02 14:22:40.690
↳ processed=DefaultState org=HandshakeState dest=CompletedState what=147462(0x24006)
2021-08-02 14:22:48.007881 ; ADB ; Debug ; rec[47]: time=08-02 14:22:40.995
↳ processed=CompletedState org=CompletedState dest=DisconnectedState what=147462(0x24006)
2021-08-02 14:22:48.007881 ; ADB ; Debug ; rec[48]: time=08-02 14:22:42.765
↳ processed=DefaultState org=DisconnectedState dest=HandshakeState what=147462(0x24006)
2021-08-02 14:22:48.007881 ; ADB ; Debug ; rec[49]: time=08-02 14:22:42.821
↳ processed=DefaultState org=HandshakeState dest=CompletedState what=147462(0x24006)
2021-08-02 14:22:48.007881 ; ADB ; Debug ; curState=CompletedState

2021-08-02 14:22:48.007881 ; ADB ; Debug ; mLinkProperties {InterfaceName:
↳ wlan0 LinkAddresses: [ fe80::c91c:3af1:c12:a391/64,192.168.13.3/24 ] DnsAddresses: [
↳ /8.8.8.8,/4.4.4.4 ] Domains: mydomain.example MTU: 0 ServerAddress: /192.168.13.1
↳ TcpBufferSizes: 524288,1048576,8808040,262144,524288,6710886 Routes: [ fe80::/64 -> :: wlan0
↳ mtu 0,192.168.13.0/24 -> 0.0.0.0 wlan0 mtu 0,0.0.0.0/0 -> 192.168.13.1 wlan0 mtu 0 ]}
2021-08-02 14:22:48.007881 ; ADB ; Debug ; mWifiInfo SSID: openwifi, BSSID:
↳ 66:55:44:33:22:ef, MAC: 50:8e:49:db:35:49, Supplicant state: COMPLETED, HE Eight Max VHT
↳ Spatial Streams Supported AP: false, Eight Max VHT Spatial streams support: false, Wi-Fi
↳ standard: 4, RSSI: -67, Link speed: 57Mbps, Tx Link speed: 57Mbps, Max Supported Tx Link
↳ speed: 72Mbps, Rx Link speed: 57Mbps, Max Supported Rx Link speed: 72Mbps, Frequency:
↳ 5220MHz, Net ID: 80, Metered hint: false, score: 60
2021-08-02 14:22:48.007881 ; ADB ; Debug ; mDhcpResultsParcelable
↳ baseConfiguration IP address 192.168.13.3/24 Gateway 192.168.13.1 DNS servers: [ 8.8.8.8
↳ 4.4.4.4 ] Domains mydomain.example leaseDuration 600mtu 0serverAddress
↳ 192.168.13.1serverHostName vendorInfo null

2021-08-02 14:22:48.008879 ; ADB ; Debug ; Dump of WifiConfigManager
2021-08-02 14:22:48.008879 ; ADB ; Debug ; 2021-08-02T14:22:38.789 -
↳ addOrUpdateNetworkInternal: added/updated config. netId=80 configKey="openwifi"NONE uid=1000
↳ name=android.uid.system:1000
2021-08-02 14:22:48.008879 ; ADB ; Debug ; 2021-08-02T14:22:38.800 -
↳ setNetworkSelectionEnabled: configKey="openwifi"NONE old
↳ networkStatus=NETWORK_SELECTION_TEMPORARY_DISABLED
↳ disableReason=NETWORK_SELECTION_DISABLED_NO_INTERNET_TEMPORARY
2021-08-02 14:22:48.008879 ; ADB ; Debug ; 2021-08-02T14:22:38.800 -
↳ setNetworkSelectionStatus: configKey="openwifi"NONE networkStatus=NETWORK_SELECTION_ENABLED
↳ disableReason=NETWORK_SELECTION_ENABLE

2021-08-02 14:22:48.008879 ; ADB ; Debug ; WifiConfigManager - Configured
↳ networks
2021-08-02 14:22:48.015885 ; ADB ; Debug ; * ID: 80 SSID: "openwifi"
↳ PROVIDER-NAME: null BSSID: null FQDN: null HOME-PROVIDER-NETWORK: false PRIO: 0 HIDDEN:
↳ false PMF: falseCarrierId: -1
2021-08-02 14:22:48.015885 ; ADB ; Debug ; NetworkSelectionStatus
↳ NETWORK_SELECTION_ENABLED
2021-08-02 14:22:48.015885 ; ADB ; Debug ; hasEverConnected: true
2021-08-02 14:22:48.015885 ; ADB ; Debug ; numAssociation 13
2021-08-02 14:22:48.015885 ; ADB ; Debug ; numNoInternetAccessReports 2
2021-08-02 14:22:48.015885 ; ADB ; Debug ; trusted
2021-08-02 14:22:48.015885 ; ADB ; Debug ; macRandomizationSetting: 0
2021-08-02 14:22:48.015885 ; ADB ; Debug ; mRandomizedMacAddress:
↳ 3a:5d:cd:14:f4:a2
2021-08-02 14:22:48.015885 ; ADB ; Debug ; randomizedMacExpirationTimeMs:
↳ 08-02 14:52:42.870
2021-08-02 14:22:48.015885 ; ADB ; Debug ; KeyMgmt: NONE Protocols: WPA RSN
↳ WAPI
2021-08-02 14:22:48.015885 ; ADB ; Debug ; AuthAlgorithms:
2021-08-02 14:22:48.015885 ; ADB ; Debug ; PairwiseCiphers: TKIP CCMP GCMP_256
2021-08-02 14:22:48.015885 ; ADB ; Debug ; GroupCiphers: WEP40 WEP104 TKIP
↳ CCMP GCMP_256
2021-08-02 14:22:48.015885 ; ADB ; Debug ; GroupMgmtCiphers: BIP_GMAC_256
2021-08-02 14:22:48.015885 ; ADB ; Debug ; SuiteBCiphers: ECDHE_ECDSA
2021-08-02 14:22:48.015885 ; ADB ; Debug ; PSK/SAE:

```

```

2021-08-02 14:22:48.015885 ; ADB ; Debug ; Enterprise config:
2021-08-02 14:22:48.015885 ; ADB ; Debug ; ocsdp: 0
2021-08-02 14:22:48.015885 ; ADB ; Debug ; DPP config:
2021-08-02 14:22:48.015885 ; ADB ; Debug ; IP config:
2021-08-02 14:22:48.015885 ; ADB ; Debug ; IP assignment: DHCP
2021-08-02 14:22:48.015885 ; ADB ; Debug ; Proxy settings: NONE
2021-08-02 14:22:48.015885 ; ADB ; Debug ; cuid=1000
↪ cname=android.uid.system:1000 luid=1000 lname=android.uid.system:1000 lcuid=1000
↪ allowAutojoin=true noInternetAccessExpected=false mostRecentlyConnected=true
2021-08-02 14:22:48.015885 ; ADB ; Debug ; lastConnected: 08-02 14:22:42.883
2021-08-02 14:22:48.015885 ; ADB ; Debug ; recentFailure: Association Rejection
↪ code: 0
2021-08-02 14:22:48.015885 ; ADB ; Debug ; ShareThisAp: false

2021-08-02 14:22:48.039897 ; ADB ; Debug ; WifiCarrierInfoManager:
2021-08-02 14:22:48.039897 ; ADB ; Debug ; Last sweep 0:00:28.907 ago.

2021-08-02 14:22:48.039897 ; ADB ; Debug ;
↪ -----
2021-08-02 14:22:48.039897 ; ADB ; Debug ; ----- Last failed
↪ connection fates -----
2021-08-02 14:22:48.039897 ; ADB ; Debug ; Time usec Walltime
↪ Direction Fate Protocol Type Result
2021-08-02 14:22:48.039897 ; ADB ; Debug ; -----
↪ -----
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4074296531 14:22:40.640 TX
↪ acked 802.11 Mgmt Probe Request N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4074297101 14:22:40.640 RX
↪ success 802.11 Mgmt Probe Response N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4074298098 14:22:40.642 TX
↪ acked 802.11 Mgmt Authentication N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4074298385 14:22:40.641 RX
↪ success 802.11 Mgmt Authentication 0: Success
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4074301750 14:22:40.645 TX
↪ acked 802.11 Mgmt Association Request N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4074302494 14:22:40.645 RX
↪ success 802.11 Mgmt Association Response 0: Success
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4074473502 14:22:40.817 TX
↪ acked DHCP Request N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4074474063 14:22:40.818 TX
↪ acked ICMPv6 Neighbor Solicitation N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4074475344 14:22:40.819 TX
↪ acked ICMPv6 MLDv2 report N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ;
↪ -----
2021-08-02 14:22:48.039897 ; ADB ; Debug ; ----- Latest fates
↪ -----
2021-08-02 14:22:48.039897 ; ADB ; Debug ; Time usec Walltime
↪ Direction Fate Protocol Type Result
2021-08-02 14:22:48.039897 ; ADB ; Debug ; -----
↪ -----
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4076433794 14:22:42.776 TX
↪ acked 802.11 Mgmt Probe Request N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4076437729 14:22:42.780 RX
↪ success 802.11 Mgmt Probe Response N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4076438667 14:22:42.781 TX
↪ acked 802.11 Mgmt Authentication N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4076438969 14:22:42.781 RX
↪ success 802.11 Mgmt Authentication 0: Success
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4076439976 14:22:42.782 TX
↪ acked 802.11 Mgmt Association Request N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4076440509 14:22:42.783 RX
↪ success 802.11 Mgmt Association Response 0: Success

```

2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076491947	14:22:42.834 TX
↪ acked	ICMPv6 Neighbor Solicitation	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076492092	14:22:42.835 RX
↪ success	ICMPv6 Neighbor Solicitation	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076510672	14:22:42.853 TX
↪ acked	ICMPv6 MLDv2 report	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076513580	14:22:42.856 TX
↪ acked	DHCP Request	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076514158	14:22:42.857 RX
↪ success	DHCP Request	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076521226	14:22:42.864 RX
↪ success	DHCP Ack	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076527972	14:22:42.870 RX
↪ success	DHCP Ack	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076541163	14:22:42.884 TX
↪ acked	ICMPv6 MLDv2 report	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076541282	14:22:42.884 TX
↪ acked	IPv4 N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076606686	14:22:42.949 TX
↪ acked	ARP Request	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076606861	14:22:42.949 RX
↪ success	ARP Reply	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076607898	14:22:42.950 TX
↪ acked	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076610838	14:22:42.953 TX
↪ acked	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076677278	14:22:43.020 TX
↪ acked	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076678394	14:22:43.021 TX
↪ acked	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076679388	14:22:43.022 TX
↪ acked	TCP HTTP	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076679578	14:22:43.022 RX
↪ success	TCP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076693538	14:22:43.036 TX
↪ acked	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4076838586	14:22:43.181 TX
↪ acked	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077043637	14:22:43.386 TX
↪ acked	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077043923	14:22:43.386 RX
↪ success	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077044879	14:22:43.387 TX
↪ acked	IPv6 Option/Protocol 17	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077045153	14:22:43.388 RX
↪ success	IPv6 Option/Protocol 17	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077148082	14:22:43.491 TX
↪ acked	ICMPv6 MLDv2 report	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077173362	14:22:43.516 TX
↪ acked	IPv4 N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077279200	14:22:43.622 TX
↪ acked	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077279413	14:22:43.622 RX
↪ success	UDP N/A	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077280463	14:22:43.623 TX
↪ acked	IPv6 Option/Protocol 17	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077280728	14:22:43.623 RX
↪ success	IPv6 Option/Protocol 17	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077287002	14:22:43.630 TX
↪ acked	ICMPv6 MLDv2 report	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077287104	14:22:43.630 TX
↪ acked	ICMPv6 Router Solicitation	N/A
2021-08-02 14:22:48.039897 ; ADB	; Debug ; 4077495408	14:22:43.838 TX
↪ acked	ICMPv6 MLDv2 report	N/A

```

2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4077531670 14:22:43.874 TX
↳ acked UDP N/A N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4077532187 14:22:43.875 RX
↳ success UDP N/A N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4077532303 14:22:43.875 TX
↳ acked IPv6 Option/Protocol 17 N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4077532392 14:22:43.875 RX
↳ success IPv6 Option/Protocol 17 N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4077796384 14:22:44.139 TX
↳ acked UDP N/A N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4077796513 14:22:44.139 TX
↳ acked IPv6 Option/Protocol 17 N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4077797128 14:22:44.140 RX
↳ success UDP N/A N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ; 4077797727 14:22:44.140 RX
↳ success IPv6 Option/Protocol 17 N/A
2021-08-02 14:22:48.039897 ; ADB ; Debug ;
↳ -----

2021-08-02 14:22:48.045894 ; ADB ; Debug ; Dump of WifiConnectivityManager
...
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.751 - About to
↳ run SavedNetworkNominator :
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.752 - About to
↳ run NetworkSuggestionNominator :
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.752 - did not
↳ see any matching auto-join enabled network suggestions.
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.752 - About to
↳ run ScoredNetworkNominator :
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.752 - Skipping
↳ nominateNetworks; Network recommendations disabled.
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.753 - Candidate
↳ { config = 80, bssid = 66:55:44:33:22:ef, freq = 5220, rssi = -69, Mbps = 48, nominator = 0,
↳ pInternet = 50, saved, trusted, noInternet, open }
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.753 -
↳ BubbleFunScorer_v2 would choose 80 score 24.74828010444529+/-8.080870283892395 expid
↳ 42598152
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.753 -
↳ CompatibilityScorer would choose 80 score 103.931+/-10.0 expid 42504592
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.753 -
↳ ScoreCardBasedScorer would choose 80 score 104.0+/-10.0 expid 42902385
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.753 -
↳ ThroughputScorer chooses 80 score 2572.931+/-10.0 expid 42330058
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.754 -
↳ AllSingleScanListener: WNS candidate-"openwifi"
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.754 -
↳ connectToNetwork: Connect to "openwifi":any from Disconnected
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.803 -
↳ handleConnectionStateChanged: state=transitioning
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.803 -
↳ startConnectivityScan: screenOn=true wifiState=transitioning scanImmediately=false
↳ wifiEnabled=true wifiConnectivityManagerEnabled=true
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.879 -
↳ handleConnectionStateChanged: state=connected
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.881 -
↳ startConnectivityScan: screenOn=true wifiState=connected scanImmediately=false
↳ wifiEnabled=true wifiConnectivityManagerEnabled=true
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.881 - Last
↳ periodic single scan started 1887ms ago, defer this new scan request.

2021-08-02 14:22:48.050854 ; ADB ; Debug ; Dump of BssidBlocklistMonitor
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BssidBlocklistMonitor - Bssid
↳ blocklist history begin ----

```



```

2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:6e,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:25:49.730, blocklistEndTimeMs=08-02 13:30:49.730,
↳ logTimeMs=08-02 13:27:25.902, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:6e,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:27:27.174, blocklistEndTimeMs=08-02 13:37:27.174,
↳ logTimeMs=08-02 13:27:32.953, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:6e,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:29:43.728, blocklistEndTimeMs=08-02 14:09:43.728,
↳ logTimeMs=08-02 13:42:32.129, trigger=clearBssidBlocklist
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=36:2c:c4:a7:f9:42,
↳ SSID="TelenetWiFree", isInBlocklist=true, blockReason=REASON_AP_UNABLE_TO_HANDLE_NEW_STA,
↳ blocklistStartTimeMs=08-02 13:41:32.325, blocklistEndTimeMs=08-02 13:46:32.325,
↳ logTimeMs=08-02 13:42:32.130, trigger=clearBssidBlocklist
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:49:42.912, blocklistEndTimeMs=08-02 13:50:12.912,
↳ logTimeMs=08-02 13:49:45.162, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:49:53.878, blocklistEndTimeMs=08-02 13:59:53.878,
↳ logTimeMs=08-02 13:50:17.095, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:50:18.208, blocklistEndTimeMs=08-02 14:10:18.208,
↳ logTimeMs=08-02 13:50:19.797, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:50:22.595, blocklistEndTimeMs=08-02 14:30:22.595,
↳ logTimeMs=08-02 13:50:42.889, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:50:45.797, blocklistEndTimeMs=08-02 15:10:45.797,
↳ logTimeMs=08-02 13:51:02.762, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:51:05.642, blocklistEndTimeMs=08-02 16:31:05.642,
↳ logTimeMs=08-02 13:51:25.722, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:51:26.924, blocklistEndTimeMs=08-02 19:11:26.924,
↳ logTimeMs=08-02 13:52:07.877, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:52:09.088, blocklistEndTimeMs=08-03 00:32:09.088,
↳ logTimeMs=08-02 13:52:31.628, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:52:32.765, blocklistEndTimeMs=08-03 00:32:32.765,
↳ logTimeMs=08-02 13:52:35.477, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:52:36.611, blocklistEndTimeMs=08-03 00:32:36.611,
↳ logTimeMs=08-02 13:53:31.688, trigger=clearBssidBlocklist
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:53:35.411, blocklistEndTimeMs=08-03 00:33:35.411,
↳ logTimeMs=08-02 13:54:44.390, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b8,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:54:45.564, blocklistEndTimeMs=08-02 14:58:45.564,
↳ logTimeMs=08-02 13:58:00.571, trigger=clearBssidBlocklist

```

```

2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:b4,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 13:59:59.639, blocklistEndTimeMs=08-02 14:04:59.639,
↳ logTimeMs=08-02 14:01:56.492, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:ec,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 14:09:58.686, blocklistEndTimeMs=08-02 14:14:58.686,
↳ logTimeMs=08-02 14:11:53.057, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:6e,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_ASSOCIATION_TIMEOUT,
↳ blocklistStartTimeMs=08-02 14:13:41.336, blocklistEndTimeMs=08-02 14:33:41.336,
↳ logTimeMs=08-02 14:13:50.490, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:78,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_NETWORK_VALIDATION_FAILURE,
↳ blocklistStartTimeMs=08-02 14:16:48.985, blocklistEndTimeMs=08-02 14:21:48.985,
↳ logTimeMs=08-02 14:18:10.963, trigger=clearBssidBlocklistForSsid
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BSSID=66:55:44:33:22:8e,
↳ SSID="openwifi", isInBlocklist=true, blockReason=REASON_NETWORK_VALIDATION_FAILURE,
↳ blocklistStartTimeMs=08-02 14:20:53.331, blocklistEndTimeMs=08-02 14:25:53.331,
↳ logTimeMs=08-02 14:21:39.831, trigger=clearBssidBlocklist
2021-08-02 14:22:48.050854 ; ADB ; Debug ; BssidBlocklistMonitor - Bssid
↳ blocklist history end ----

2021-08-02 14:22:48.050854 ; ADB ; Debug ; Dump of WifiLastResortWatchdog
2021-08-02 14:22:48.050854 ; ADB ; Debug ; WifiLastResortWatchdog - Log Begin
↳ ----
...
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:21:39.828 -
↳ connectedStateTransition: isEntering = false
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:40.965 -
↳ updateFailureCountForNetwork: ["openwifi", 66:55:44:33:22:ef, 3]
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:40.966 -
↳ checkTriggerCondition: return = false
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.896 -
↳ connectedStateTransition: isEntering = true
2021-08-02 14:22:48.050854 ; ADB ; Debug ; 2021-08-02T14:22:42.896 -
↳ connectedStateTransition: setWatchdogTriggerEnabled to true
2021-08-02 14:22:48.050854 ; ADB ; Debug ; WifiLastResortWatchdog - Log End
↳ ----

2021-08-02 14:22:48.054851 ; ADB ; Debug ; mConnectionEvents:
...
2021-08-02 14:22:48.055851 ; ADB ; Debug ; startTime=08-02 14:19:33.038,
↳ SSID="openwifi", BSSID=66:55:44:33:22:8e, durationMillis=132, roamType=ROAM_UNRELATED,
↳ connectionResult=1, level2FailureCode=NONE, connectivityLevelFailureCode=NONE,
↳ signalStrength=-66, wifiState=WIFI_DISCONNECTED, screenOn=true,
↳ mRouterFingerprint=mConnectionEvent.roamType=0, mChannelInfo=5220, mDtim=0,
↳ mAuthenticaiton=1, mHidden=false, mRouterTechnology=4, mSupportsIpv6=false, mEapMethod=0,
↳ mAuthPhase2Method=0, mOcspType=0, mPmkCache=false, mMaxSupportedTxLinkSpeedMbps=72,
↳ mMaxSupportedRxLinkSpeedMbps=72, useRandomizedMac=false, useAggressiveMac=false,
↳ connectionNominator=NOMINATOR_SAVED, networkSelectorExperimentId=42330058,
↳ numBssidInBlocklist=0, level2FailureReason=FAILURE_REASON_UNKNOWN, networkType=TYPE_OPEN,
↳ networkCreator=CREATOR_USER, numConsecutiveConnectionFailure=0, isOsuProvisioned=false
2021-08-02 14:22:48.055851 ; ADB ; Debug ; startTime=08-02 14:22:40.615,
↳ SSID="openwifi", BSSID=66:55:44:33:22:ef, durationMillis=334, roamType=ROAM_UNRELATED,
↳ connectionResult=0, level2FailureCode=NETWORK_DISCONNECTION,
↳ connectivityLevelFailureCode=NONE, signalStrength=-69, wifiState=WIFI_DISCONNECTED,
↳ screenOn=true, mRouterFingerprint=mConnectionEvent.roamType=0, mChannelInfo=5220, mDtim=0,
↳ mAuthenticaiton=1, mHidden=false, mRouterTechnology=4, mSupportsIpv6=false, mEapMethod=0,
↳ mAuthPhase2Method=0, mOcspType=0, mPmkCache=false, mMaxSupportedTxLinkSpeedMbps=72,
↳ mMaxSupportedRxLinkSpeedMbps=72, useRandomizedMac=false, useAggressiveMac=false,
↳ connectionNominator=NOMINATOR_SAVED, networkSelectorExperimentId=42330058,
↳ numBssidInBlocklist=0, level2FailureReason=FAILURE_REASON_UNKNOWN, networkType=TYPE_OPEN,
↳ networkCreator=CREATOR_USER, numConsecutiveConnectionFailure=0, isOsuProvisioned=false

```

```

2021-08-02 14:22:48.055851 ; ADB ; Debug ; startTime=08-02 14:22:42.757,
↳ SSID="openwifi", BSSID=66:55:44:33:22:ef, durationMillis=121, roamType=ROAM_UNRELATED,
↳ connectionResult=1, level2FailureCode=NONE, connectivityLevelFailureCode=NONE,
↳ signalStrength=-69, wifiState=WIFI_DISCONNECTED, screenOn=true,
↳ mRouterFingerprint=mConnectionEvent.roamType=0, mChannelInfo=5220, mDtim=0,
↳ mAuthentication=1, mHidden=false, mRouterTechnology=4, mSupportsIpv6=false, mEapMethod=0,
↳ mAuthPhase2Method=0, mOscspType=0, mPmkCache=false, mMaxSupportedTxLinkSpeedMbps=72,
↳ mMaxSupportedRxLinkSpeedMbps=72, useRandomizedMac=false, useAggressiveMac=false,
↳ connectionNominator=NOMINATOR_SAVED, networkSelectorExperimentId=42330058,
↳ numBssidInBlocklist=0, level2FailureReason=FAILURE_REASON_UNKNOWN, networkType=TYPE_OPEN,
↳ networkCreator=CREATOR_USER, numConsecutiveConnectionFailure=0, isOsuProvisioned=false

2021-08-02 14:22:48.055851 ; ADB ; Debug ; StaEventList:
...
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:21:39.833 WIFI_DISABLED
↳ lastRssi=-66 lastFreq=5220 lastLinkSpeed=57 lastScore=60 totalTxBytes=1048253
↳ totalRxBytes=1150008 screenOn=true cellularData=false
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:21:53.555 WIFI_ENABLED
↳ totalTxBytes=1052176 totalRxBytes=1152254 screenOn=true cellularData=false
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:22:40.625 CMD_START_CONNECT
↳ totalTxBytes=1056699 totalRxBytes=1155030 screenOn=true cellularData=false, ConfigInfo:
↳ allowed_key_management=1 allowed_protocols=11 allowed_auth_algorithms=0
↳ allowed_pairwise_ciphers=14 allowed_group_ciphers=47 hidden_ssid=false is_passpoint=false
↳ is_ephemeral=false has_ever_connected=true scan_rssi=-69 scan_freq=5220
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:22:40.660
↳ CMD_ASSOCIATED_BSSID totalTxBytes=1056699 totalRxBytes=1155030 screenOn=true
↳ cellularData=false, supplicantStateChangeEvents: { ASSOCIATING ASSOCIATED }
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:22:40.689
↳ NETWORK_CONNECTION_EVENT totalTxBytes=1056815 totalRxBytes=1155070 screenOn=true
↳ cellularData=false
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:22:40.995
↳ NETWORK_DISCONNECTION_EVENT local_gen=false reason=7:CLASS3_FRAME_FROM_NONASSOC_STA
↳ lastRssi=-70 lastFreq=5220 lastLinkSpeed=43 lastScore=60 totalTxBytes=1057235
↳ totalRxBytes=1155150 screenOn=true cellularData=false, supplicantStateChangeEvents: {
↳ COMPLETED }
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:22:42.761 CMD_START_CONNECT
↳ totalTxBytes=1058739 totalRxBytes=1155998 screenOn=true cellularData=false,
↳ supplicantStateChangeEvents: { DISCONNECTED }, ConfigInfo: allowed_key_management=1
↳ allowed_protocols=11 allowed_auth_algorithms=0 allowed_pairwise_ciphers=14
↳ allowed_group_ciphers=47 hidden_ssid=false is_passpoint=false is_ephemeral=false
↳ has_ever_connected=true scan_rssi=-69 scan_freq=5220
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:22:42.802
↳ CMD_ASSOCIATED_BSSID totalTxBytes=1058739 totalRxBytes=1155998 screenOn=true
↳ cellularData=false, supplicantStateChangeEvents: { ASSOCIATING ASSOCIATED }
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:22:42.820
↳ NETWORK_CONNECTION_EVENT totalTxBytes=1058855 totalRxBytes=1156038 screenOn=true
↳ cellularData=false
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:22:42.876
↳ CMD_IP_CONFIGURATION_SUCCESSFUL lastRssi=-67 lastFreq=5220 lastLinkSpeed=57 lastScore=60
↳ totalTxBytes=1059003 totalRxBytes=1157020 screenOn=true cellularData=false,
↳ supplicantStateChangeEvents: { COMPLETED }

2021-08-02 14:22:48.058849 ; ADB ; Debug ; UserActionEvents:
...
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:21:39.675
↳ eventType=EVENT_TOGGLE_WIFI_OFF startTimeMillis=4013332
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:21:53.060
↳ eventType=EVENT_TOGGLE_WIFI_ON startTimeMillis=4026716
2021-08-02 14:22:48.058849 ; ADB ; Debug ; 08-02 14:22:18.598
↳ eventType=EVENT_TOGGLE_WIFI_ON startTimeMillis=4052255

2021-08-02 14:22:48.059849 ; ADB ; Debug ; Wifi power metrics:
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Logging duration (time on battery):
↳ 320649268

```

```

2021-08-02 14:22:48.059849 ; ADB ; Debug ; Energy consumed by wifi (mAh): 0.0
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Amount of time wifi is in idle (ms):
↳ 3927954
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Amount of time wifi is in rx (ms):
↳ 78540
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Amount of time wifi is in tx (ms):
↳ 49460
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Amount of time kernel is active
↳ because of wifi data (ms): 82819406
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Amount of time wifi is in sleep
↳ (ms): 316593314
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Amount of time wifi is scanning
↳ (ms): 684251
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Number of packets sent (tx): 381929
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Number of bytes sent (tx): 53993813
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Number of packets received (rx):
↳ 1098374
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Number of bytes sent (rx):
↳ 1426761839
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Energy consumed across measured wifi
↳ rails (mAh): 0
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Wifi radio usage metrics:
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Logging duration (time on battery):
↳ 320649268
2021-08-02 14:22:48.059849 ; ADB ; Debug ; Amount of time wifi is in scan mode
↳ while on battery (ms): 684251

2021-08-02 14:22:48.059849 ; ADB ; Debug ; WifiIsUnusableEventList:
...
2021-08-02 14:22:48.059849 ; ADB ; Debug ; 08-02 14:14:22.556 FIRMWARE_ALERT
↳ lastScore=60 txSuccessDelta=0 txRetriesDelta=0 txBadDelta=0 rxSuccessDelta=0
↳ packetUpdateTimeDelta=0ms firmwareAlertCode=12 lastWifiUsabilityScore=-1
↳ lastPredictionHorizonSec=-1 screenOn=true mobileTxBytes=0 mobileRxBytes=0
↳ totalTxBytes=930941 totalRxBytes=1105828
2021-08-02 14:22:48.059849 ; ADB ; Debug ; 08-02 14:22:40.948 FIRMWARE_ALERT
↳ lastScore=60 txSuccessDelta=0 txRetriesDelta=0 txBadDelta=0 rxSuccessDelta=0
↳ packetUpdateTimeDelta=0ms firmwareAlertCode=12 lastWifiUsabilityScore=-1
↳ lastPredictionHorizonSec=-1 screenOn=true mobileTxBytes=0 mobileRxBytes=0
↳ totalTxBytes=1056983 totalRxBytes=1155070

2021-08-02 14:22:48.059849 ; ADB ; Debug ; Hardware Version:
2021-08-02 14:22:48.059849 ; ADB ; Debug ; mWifiUsabilityStatsEntriesList:
...
2021-08-02 14:22:48.059849 ; ADB ; Debug ; label=2
2021-08-02 14:22:48.059849 ; ADB ; Debug ; trigger_type=4
2021-08-02 14:22:48.059849 ; ADB ; Debug ; time_stamp_ms=4074605
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↳ timestamp_ms=3980491,rssi=-66,link_speed_mbps=57,total_tx_success=162,total_tx_retries=62,total_tx_bad=0,total_tx_bytes=1098374,total_rx_bytes=1426761839
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↳ timestamp_ms=3983509,rssi=-66,link_speed_mbps=57,total_tx_success=167,total_tx_retries=63,total_tx_bad=0,total_tx_bytes=1098374,total_rx_bytes=1426761839
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↳ timestamp_ms=3986524,rssi=-66,link_speed_mbps=57,total_tx_success=167,total_tx_retries=63,total_tx_bad=0,total_tx_bytes=1098374,total_rx_bytes=1426761839
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↳ timestamp_ms=3989547,rssi=-66,link_speed_mbps=57,total_tx_success=173,total_tx_retries=66,total_tx_bad=0,total_tx_bytes=1098374,total_rx_bytes=1426761839
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↳ timestamp_ms=3992572,rssi=-66,link_speed_mbps=57,total_tx_success=176,total_tx_retries=67,total_tx_bad=0,total_tx_bytes=1098374,total_rx_bytes=1426761839
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↳ timestamp_ms=3995590,rssi=-68,link_speed_mbps=57,total_tx_success=180,total_tx_retries=70,total_tx_bad=0,total_tx_bytes=1098374,total_rx_bytes=1426761839
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↳ timestamp_ms=3998611,rssi=-67,link_speed_mbps=57,total_tx_success=186,total_tx_retries=72,total_tx_bad=0,total_tx_bytes=1098374,total_rx_bytes=1426761839
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↳ timestamp_ms=4001636,rssi=-66,link_speed_mbps=57,total_tx_success=186,total_tx_retries=72,total_tx_bad=0,total_tx_bytes=1098374,total_rx_bytes=1426761839
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↳ timestamp_ms=4004658,rssi=-66,link_speed_mbps=57,total_tx_success=192,total_tx_retries=73,total_tx_bad=0,total_tx_bytes=1098374,total_rx_bytes=1426761839

```

```

2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↪ timestamp_ms=4007679,rssi=-67,link_speed_mbps=57,total_tx_success=194,total_tx_retries=73,total_tx_bad=0,tot
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↪ timestamp_ms=4010732,rssi=-66,link_speed_mbps=57,total_tx_success=197,total_tx_retries=74,total_tx_bad=0,tot
2021-08-02 14:22:48.059849 ; ADB ; Debug ;
↪ timestamp_ms=4074349,rssi=-70,link_speed_mbps=43,total_tx_success=0,total_tx_retries=0,total_tx_bad=0,totall

2021-08-02 14:22:48.060847 ; ADB ; Debug ; WifiScoreReport:
2021-08-02 14:22:48.060847 ; ADB ; Debug ;
↪ time,session,netid,rssi,filtered_rssi,rssi_threshold,freq,txLinkSpeed,rxLinkSpeed,tx_good,tx_retry,tx_bad,r

...
2021-08-02 14:22:48.061847 ; ADB ; Debug ; 08-02
↪ 14:21:37.081,211,272,-66.0,-66.5,-80.0,5220,57,57,1.03,0.25,0.00,0.10,0,173,54,64,60
2021-08-02 14:22:48.061847 ; ADB ; Debug ; 08-02
↪ 14:22:40.701,213,273,-70.0,-70.0,-80.0,5220,43,43,0.00,0.00,0.00,0.00,0,174,50,60,60
2021-08-02 14:22:48.061847 ; ADB ; Debug ; 08-02
↪ 14:22:42.828,214,274,-67.0,-67.0,-80.0,5220,57,57,0.00,0.00,0.00,0.00,0,175,53,63,60

adb shell ping -c10 192.168.13.1

2021-08-02 14:22:57.560982 ; ADB ; Debug ; ----- Start of adb output -----
2021-08-02 14:22:57.560982 ; ADB ; Debug ; PING 192.168.13.1 (192.168.13.1)
↪ 56(84) bytes of data.
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=1 ttl=64 time=16.8 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=2 ttl=64 time=14.8 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=3 ttl=64 time=2.11 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=4 ttl=64 time=1.21 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=5 ttl=64 time=1.77 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=6 ttl=64 time=28.0 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=7 ttl=64 time=1.87 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=8 ttl=64 time=17.4 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=9 ttl=64 time=16.8 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 64 bytes from 192.168.13.1:
↪ icmp_seq=10 ttl=64 time=16.1 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; --- 192.168.13.1 ping statistics ---
2021-08-02 14:22:57.560982 ; ADB ; Debug ; 10 packets transmitted, 10 received,
↪ 0% packet loss, time 9017ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; rtt min/avg/max/mdev =
↪ 1.213/11.714/28.038/8.831 ms
2021-08-02 14:22:57.560982 ; ADB ; Debug ; ----- End of adb output -----

./fosdem.sh

...
2021-08-02 14:22:57.746881 ; TestStep ; Information ; sdr0: AP-ENABLED
2021-08-02 14:22:57.746881 ; TestStep ; Information ; sdr0: STA 50:8e:49:db:35:49 IEEE
↪ 802.11: authenticated
2021-08-02 14:22:57.746881 ; TestStep ; Information ; sdr0: STA 50:8e:49:db:35:49 IEEE
↪ 802.11: authenticated
2021-08-02 14:22:57.746881 ; TestStep ; Information ; sdr0: STA 50:8e:49:db:35:49 IEEE
↪ 802.11: associated (aid 1)
2021-08-02 14:22:57.746881 ; TestStep ; Information ; sdr0: AP-STA-CONNECTED
↪ 50:8e:49:db:35:49
2021-08-02 14:22:57.746881 ; TestStep ; Information ; sdr0: STA 50:8e:49:db:35:49 RADIUS:
↪ starting accounting session 000002BB-00000000

```

2021-08-02 14:22:57.746881 ; TestStep ; Information ; sdr0: AP-STA-DISCONNECTED
↪ 50:8e:49:db:35:49

Bibliography

- [1] T. Alsop, “Number of wireless local area network (wlan) connected devices worldwide from 2016 to 2021,” 2021, ”Accessed on 2 May 2021”). [Online]. Available: <https://www.statista.com/statistics/802706/world-wlan-connected-device/>
- [2] K. Shaw, “Hoe werkt het osi-model precies?” December 2017, ”Accessed on 3 june 2021”). [Online]. Available: <https://www.computerworld.com/article/3594283/hoe-werkt-het-osi-model-precies.html>
- [3] Cisco Meraki, “802.11 association process explained,” October 2020, ”Accessed on 3 june 2020”). [Online]. Available: https://documentation.meraki.com/MR/WiFi_Basics_and_Best_Practices/802.11_Association_Process_Explained
- [4] M. S. Gast, *802.11 WireLess Networks The definitive Guide*. O’Reilly Media, Inc., 2005. [Online]. Available: https://books.google.be/books/about/802.11_Wireless_Networks.html?id=9rHnRzzMHLIC&redir_esc=y
- [5] J. Xianjun, 2020, ”Accessed on 29 April 2021”). [Online]. Available: <https://github.com/open-sdr/openwifi>
- [6] IEEE Standards Association, “Part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications,” in *IEEE Std 802.11™*, 2016. [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7786995>
- [7] D. Meharchandani, “10 major cyber attacks witnessed globally in q1 2021,” April 2021, ”Accessed on 14 February 2021”). [Online]. Available: <https://securityboulevard.com/2021/04/10-major-cyber-attacks-witnessed-globally-in-q1-2021/>
- [8] Center for Strategic and International Studies, “Significant cyber incidents since 2006,” 2021, ”Accessed on 13 February 2021”). [Online]. Available: <https://www.csis.org/programs/strategic-technologies-program/significant-cyber-incidents>
- [9] D. Tosh, O. Galindo, V. Kreinovich, and O. Kosheleva, “Towards security of cyber-physical systems using quantum computing algorithms,” pp. 313–320, 2020, ”Accessed on 2 May 2021”).
- [10] A. H. Lashkari, F. Towhidi, and R. S. Hosseini, “Wired equivalent privacy (wep),” pp. 492–495, 2009, ”Accessed on 10 February 2021”).
- [11] M. Vanhoef and F. Piessens, “Key reinstallation attacks: Forcing nonce reuse in wpa2,” October 2017, ”Accessed on 10 February 2021”). [Online]. Available: <https://papers.mathyvanhoef.com/ccs2017.pdf>
- [12] —, “Key reinstallation attacks breaking wpa2 by forcing nonce reuse,” 2017, ”Accessed on 10 February 2021”). [Online]. Available: <https://www.krackattacks.com/>
- [13] M. E. Garbelini, C. Wang, and S. Chattopadhyay, “Greyhound: Directed greybox wi-fi fuzzing,” 2020, ”(Accessed on 20 February 2021”). [Online]. Available: <https://asset-group.github.io/papers/Greyhound.pdf>

- [14] wifi-bond, “802.11 association process,” April 2017, ”Accessed on 13 February 2020)”. [Online]. Available: <https://www.escript.com/en/news-events/fuzz-testing>
- [15] R. Dionicio, “802.11 state machine – association and authentication,” September 2015, ”Accessed on 13 February 2020)”. [Online]. Available: <https://packet6.com/802-11-state-machine/>
- [16] escript, “Security testing: The power of fuzzing,” October 2019, ”Accessed on 20 November 2020)”. [Online]. Available: <https://www.escript.com/en/news-events/fuzz-testing>
- [17] guru99, “Fuzz testing(fuzzing) tutorial: What is, types, tools & example,” ”Accessed on 20 November 2020)”. [Online]. Available: <https://www.guru99.com/fuzz-testing.html>
- [18] B. Pleiter, “Fuzzing wi-fi in iot devices,” Ph.D. dissertation, Radboud University, 2020. [Online]. Available: https://www.cs.ru.nl/bachelors-theses/2020/Bart_Pleiter___4752740___Fuzzing_Wi-Fi_in_IoT_devices.pdf
- [19] Analog Devices, “Adrv9361-z7035,” ”Accessed on 6 June 2021)”. [Online]. Available: <https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/adv9361-z7035.html#eb-overview>
- [20] —, “Adrv1crr-bob,” ”Accessed on 6 June 2021)”. [Online]. Available: <https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/ADRV1CRR-BOB.html#eb-overview>
- [21] N. A. Markus Dillinger, Kambiz Madani, *Software defined radio: architectures, systems, and functions*. Wiley, 2003, ”Accessed on 20 July 2021)”.
- [22] V. K. Garg, *Wireless Communications & Networking*. Morgan Kaufmann, 2007, ”Accessed on 20 July 2021)”.
- [23] Mouser, “Analog devices inc. adv9361-z7035 sdr 2x2 system-on-module,” 2018, ”Accessed on 6 June 2021)”. [Online]. Available: <https://www.mouser.be/new/analog-devices/adi-adv9361-z7035>
- [24] Xilinx, “System-on-modules (soms): How and why to use them,” ”Accessed on 20 July 2021)”. [Online]. Available: <https://www.xilinx.com/products/som/what-is-a-som.html>
- [25] B. OLoughlin, “Adi adv936x system on module (som) sdr,” March 2021, ”Accessed on 20 May 2021)”. [Online]. Available: https://wiki.analog.com/resources/eval/user-guides/adv936x_rfsom
- [26] M. C. Peter Loshin, “Secure shell (ssh),” April 2020, ”Accessed on 10 July 2021)”. [Online]. Available: <https://searchsecurity.techtarget.com/definition/Secure-Shell>
- [27] android developers, “Android debug bridge (adb),” July 2021. [Online]. Available: <https://developer.android.com/studio/command-line/adb>
- [28] Keysight Technologies, “Opentap,” 2017, ”(Accessed on 29 April 2021)”. [Online]. Available: <https://www.opentap.io/about.html>
- [29] A. I. Dragos Brezoi, “Sshsteps,” 2020, ”Accessed on 15 February 2021)”. [Online]. Available: <https://gitlab.com/OpenTAP/Plugins/keysight/sshsteps/-/blob/master/OpenTap.Plugins.Ssh/SshCommandStep.cs>
- [30] M. Ang, “Opentap,” May 2020, ”Accessed on 1 April 2021)”. [Online]. Available: <https://doc.opentap.io/>

- [31] G. Combs, “About wireshark,” ”Accessed on 17 January 2021”). [Online]. Available: <https://www.wireshark.org/>
- [32] B. Ghosh, “How to capture wi-fi traffic using wireshark,” 2020, ”Accessed on 20 July 2021”). [Online]. Available: <https://linuxhint.com/capture-wi-fi-traffic-using-wireshark/>
- [33] 0x90, “wifuzz,” ”Accessed on 23 November 2020”). [Online]. Available: <https://github.com/0x90/wifuzz>
- [34] L. Butti, “wifuzzit,” ”Accessed on 23 November 2020”). [Online]. Available: <https://github.com/0xd012/wifuzzit>
- [35] J. Pereyda, “boofuzz,” ”Accessed on 23 November 2020”). [Online]. Available: <https://github.com/jtpereyda/boofuzz#readme>
- [36] S. Krishnan, “Introduction to linux interrupts and cpu smp affinity,” 2014, ”Accessed on 1 august 2021”). [Online]. Available: <https://www.thegeekstuff.com/2014/01/linux-interrupts/>
- [37] Techplayon, “How conventional agc (automatic gain control) works in receiver chain and its important parameters.” 2018, ”Accessed on 2 august 2021”). [Online]. Available: <https://www.techplayon.com/conventional-agc-automatic-gain-control-works-receiver-chain-explain-important-parameters/>
- [38] D. McKay, “How to use the dmesg command on linux,” December 2019, ”Accessed on 16 May 2021”). [Online]. Available: <https://www.howtogeek.com/449335/how-to-use-the-dmesg-command-on-linux>
- [39] P. Romano, “Wifi standards 802.11a/b/g/n vs. 802.11ac: Which is best?” July 2014, ”Accessed on 15 August 2021”). [Online]. Available: <https://www.semiconductorstore.com/blog/2014/WiFi-standards-802-11a-b-g-n-vs-802-11ac-Which-is-Best/806/>
- [40] H. James, “Scp linux – securely copy files using scp examples,” April 2020, ”Accessed on 15 August 2020”). [Online]. Available: <https://packet6.com/802-11-state-machine/>
- [41] L. Phifer, “What is 802.11n ”greenfield” mode?” May 2008, ”Accessed on 15 August 2021”). [Online]. Available: <https://www.computerweekly.com/news/2240101850/What-is-80211n-Greenfield-mode>
- [42] TechDifferences, “Difference between bootp and dhcp,” ”Accessed on 15 August 2021”). [Online]. Available: <https://techdifferences.com/difference-between-bootp-and-dhcp.html>
- [43] J. Pihl, “adb-join-wifi,” February 2019, ”Accessed on 20 July 2021”). [Online]. Available: <https://github.com/steinwurf/adb-join-wifi#usage>

