FACULTY OF SCIENCES

Developing IEEE 802.11 PHY fuzzing capabilities using the open source Openwifi project

Jasper Devreker
Student number: 01702823

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

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de informatica

Academic year 2021-2022

GHENT UNIVERSITY

Acknowledgment

In first place, I would like to thank Prof. Dr. Ir. Ingrid Moerman and Dr. Xianjun Jiao for their support, help and assistance they provided during the thesis. Not only for offering this thesis, but also for all the effort and time they put into it. I would like to thank Rafael Cavalcanti for dealing with all details concerning Keysight. I also wish to show my gratitude to the OpenOFDM authors and Keysight for their excellent (open!) documentation on the 802.11 standard.

Finally, I would like to extend my deepest gratitude to the members of Zeus WPI, who always made sure there was something to do in the Zeus basement other than working on the thesis. In the past 5 years, we accomplished some great achievements together, I'm very thankful to have been a part of the group.

Permission for use

"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."

"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."

"Deze masterproef vormt een onderdeel van een examen. Eventuele opmerkingen die door de beoordelingscommissie tijdens de mondelinge uiteenzetting van de masterproef werden geformuleerd, werden niet verwerkt in deze tekst."

"This master's dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master's dissertation are not included in this text."

Abstract

One of the major challenges in computer security is the shear amount of different software and hardware implementations that need to have their security evaluated. One way to alleviate this problem is to automate certain parts of security testing, for example with fuzz testing: this method consists of sending mutated input to a device or process and then detecting anomalies. For Wi-Fi, there already exist numerous fuzzers, but they all work on the MAC layer.

This thesis goes a layer deeper and enables fuzzing of the PHY layer, more specifically the legacy and high-throughput signal header. The open source Openwifi project implements the whole 802.11 standard on an FPGA; this project was modified to allow setting arbitrary values in the signal headers. This means that the on-air values can differ from the hardware parameters used to send the frame. The FPGA modifications were then tested in simulation, using the Openwifi self loopback feature and with a wireless adapter in monitor mode. A Proof-Of-Concept denial-of-service attack was then carried out by sending frames with a modified length value in their legacy signal header, reducing the throughput of other clients by 75%. These modified packets are not detectable with a wireless adapter in monitor mode, making this attack hard to detect. A similar attack with the high-throughput signal header was also tried, but proved to be ineffective.

Keywords: Openwifi, IEEE802.11, PHY, FPGA

Developing IEEE 802.11 PHY Fuzzing Capabilities Using the Open Source Openwifi Project

Jasper Devreker
Ghent University
Ghent, Belgium
Jasper.Devreker@UGent.be

***Abstract*—One of the major challenges in computer security is the shear amount of different software and hardware implementations that need to have their security evaluated. One way to alleviate this problem is to automate certain parts of security testing, for example with fuzz testing: this method consists of sending mutated input to a device or process and then detecting anomalies. For Wi-Fi, there already exist numerous fuzzers, but they all work on the MAC layer. This thesis goes a layer deeper and enables fuzzing of the PHY layer, more specifically the legacy and high-throughput signal header. The open source Openwifi project implements the whole 802.11 standard on an FPGA; this project was modified to allow setting arbitrary values in the signal headers. The modifications were then tested in simulation, using the Openwifi self loopback feature and with a wireless adapter in monitor mode. A Proof-Of-Concept denial-of-service attack was then carried out by sending frames with a modified length value in their signal header, reducing the throughput of other clients by 75%. These modified packets are not detectable with a wireless adapter in monitor mode, making this attack hard to detect.**

***Index Terms*—Wi-Fi, IEEE802.11, FPGA, security, fuzzing**

I. Introduction

According to the Wi-Fi Alliance, nearly 18 billion Wi-Fi devices will be in use in 2022 [1]. It's important that these are secure, so over the years, more and more security mitigations have been introduced in the Wi-Fi standards: WEP, WPA, WPA2, protected management frames, Opportunistic Wireless Encryption, Simultaneous Authentication of Equals, ... [2] However, even if the protocol is secure, the implementation might not always be, as shown by for example CVE-2020-26145, where plaintext frames were accepted as full frames in an encrypted network [3]. Detecting implementation errors manually is a lot of work, made worse by the fact that there are a lot of implementations: there are multiple vendors, most of whom make multiple chipsets, each having a potentially different implementation of the standards. To automate assessing the security of these implementations security, fuzz testing (fuzzing) can be used: with fuzzing, a lot of packets are sent to the device under test, some of which might be invalid or malformed. If the device reacts unexpectedly, the fuzzing software logs the packet flow for a human to take a closer look at. This allows for much quicker security testing compared to manually auditing. The aim of this thesis is to research the possibility of modifying the Openwifi project [4] to be able to fuzz the physical layer of 802.11. This thesis will change the FPGA implementation in order to be able to send packets where the PHY parameters used in the transmission hardware don't match with the parameters sent over the air in the signal field of the packet. This thesis will additionally provide support for fuzzing the PHY layer of packets in HT mode. Finally, using the newly developed features, a hard to detect denial-of-service attack will be carried out: frames where the length value in the signal header is set to the maximum value will be continually sent, preventing other stations from transmitting their own frames. The attack is hard to detect because most commercial Wi-Fi adapters don't pass invalid or incorrectly received frames to the operating system.

II. Design

A. FPGA modifications

Thanks to the modularity and separation of concerns in the Openwifi FPGA code, only one module needed to be modified to allow signal field modification: `openofdm_tx`. This module is responsible for modulating the bits in the 802.11 frames into a modulated signal; it implements the PHY layer [5]. The `openofdm_tx` module consists of several state machines: the data collection state machine, the IQ sample generation state machine and the IQ sample forwarding state machine. Since we only need to modify what's sent and not the modulation of the signal itself, we only need to modify code in the first state machine. The FPGA design uses a 64 bit data word to encode the 24 bit legacy signal header. This means that there are still 40 bits left over to put the hardware parameters in. This modification was very minimal, only some offsets into the FPGA data words needed to be incremented by 32: the transmitted header is in `bram_din[24:0]`, the hardware parameters are, in the same format, in `bram_din[24+32:32]`. For the High Throughput header, the modifications weren't as trivial: since the HT signal header is 48 bits long, there's not enough bits left over in the 64 bit FPGA word to squeeze in the HT hardware parameters. Another FPGA word thus needed to be used to hold the HT hardware parameters. In order for the FPGA to still write one bit per clock cycle for the bit source selection and scrambling, while transmitting the first bits of the HT signal header, we save the current block RAM word and pre-read the next block RAM word into a register. This saved register can then be used to read the hardware parameters when they are needed.

B. Driver changes

The driver also needed to be changed to account for the changed hardware: for the legacy signal modifications, the block RAM word that contains the legacy signal header needs to also contain the hardware parameters. For the High Throughput change, the size of the block RAM allocated for the PHY header needed to increased from 16 to 24 bytes and these bytes need to be filled in. In the master's thesis of another student [6], it's described how to pass the hardware parameters to the driver by using the 8 byte timestamp field in the Radiotap header. That way, the additional signal header data can be passed from user space to kernel space without having to modify the driver API.

III. Verification

Once the modifications to Openwifi were finished, the design could be validated by testing. We used three methods to verify that the implementation is correct: functional simulation (simulating the circuit and checking that the output for a given input matches the expectation), in-system testing (by transmitting a frame on the FPGA, capturing it and decoding it in a receiver simulation) and by transmitting a frame on the FPGA, then capturing it with a commercial Wi-Fi adapter.

A. Functional verification

For the functional simulation, two Openwifi modules were used: `openofdm_tx` (this is the module we modified to be able to send the modified signal field) and `openofdm_rx` (this module takes IQ samples and tries to decode those into 802.11 frames). Both modules have a Verilog test bench that allows developers to run a simulation. The `openofdm_tx` module's testbench (`dot11_tx_tb.v` connects a `dot11_tx` instance to the contents of a file instead of to block RAM. The IQ samples generated in the simulation are then written to a file. Special care needed to be taken to keep the correct memory layout of the input file: the modifications for the HT field increased the size of BRAM for the physical header from one 64-bit word to two 64-bit words; the test vectors in the Openwifi repository thus need to be modified to account for this.

The file with IQ samples can then be loaded into the `openofdm_rx` testbench, where they are decoded into 802.11 frames. After this, the textttopenofdm_rx internal simulation waveforms can be checked to confirm we see the expected malformed parameters in the `openofdm_rx` receiver module.

B. Openwifi self-loopback

After the testbenches showed that the modifications worked in simulation, the Verilog code was compiled into a bitstream and the patched driver was compiled. The `openwifi-1.2.0-leuven- 2-64bit.img` image was written to the SD-card, after which the existing bitstream and driver files were overwritten with the newly compiled ones.

The Openwifi project is full duplex; this means that the FPGA can transmit and receive at the same time. This makes it possible to receive the a signal that was transmitted from the FPGA and then save the IQ samples or decode the sent signal [7]. This feature was used to test the implementation by sending a frame with a modified signal header, capturing the IQ samples and then decoding those samples.

There is a program called `inject_80211` in the Openwifi repository that allows sending frames with specific rates, lengths, ... This program was modified to take an additional two command line flags: `-c` sets the legacy signal header, `-f` sets the HT signal header. The modified program was then used to send frames with a modified signal header.

The existing Python script that received the IQ sample UDP packet originally wrote the IQ samples to a file in a format that makes it easy to load into MATLAB for further analysis. MATLAB has a WLAN Toolbox™ software package that has support for decoding 802.11 packets from IQ samples. We however chose not to use MATLAB for this thesis since it is closed-source and one has to have a (paid) license that's behind a 'Request a quote' page to use it. Instead, we opted to use the OpenOFDM 802.11 OFDM decoder that is included in the openwifi-hw repository. The decoder was used in the Vivado simulator. The file format needed to be changed slightly (there was a bug in older versions of Openwifi that swapped the I and Q component).

C. Commercial Wi-Fi adapter

A commercial, off-the-shelf Wi-Fi adapter (TP-Link TL-WN722N) was put into monitor mode and Wireshark was started on that interface. The modified `inject_80211` program was then used to send both legacy and HT frames. Unmodified frames were correctly captured by the adapter, while frames with a modified signal header were not shown (the adapter does not pass corrupted or incorrect frames to the host PC).

IV. Results

The modifications to the signal field make it possible to send frames with an arbitrary signal header. It's for example possible to send a frame that's very short, but with the signal header indicating the slowest possible rate ($6\,\mathrm{Mbit/s}$) and the longest possible length: the size of the length field is 12 bits, so the length field could indicate that the length of the sent message is 4095 bytes long. Each OFDM symbol contains 24 bits, so in total the biggest frame could contain 1367 OFDM symbols [8].

These 1367 symbols would take $(1367\,\mathrm{symbol})/(6\,\mathrm{Mbit/s}) * (24\,\mathrm{bit/symbol}) = 5.468\,\mathrm{ms}$ to transmit. When receiving such a signal header, the receiver would be tied up for 5.468 ms and thus be unable to receive any other signals. With our new capabilities of modifying the L-SIG header, it's possible to continually send a very short packet with a spoofed L-SIG header that indicates the maximum possible size and slowest possible rate and in that way jam other transmitters.

This attack was experimentally tested by setting up an access point and measuring the throughput under different

conditions. The access point is an off-the-shelf ASUS AC1300 running OpenWrt 18.06, configured to run a single WPA2 access point in 802.11a mode on channel 1 (2412 MHz). This access point is connected to a local area network (LAN) in which an iperf3 server is running. iperf3 is a software package that can test throughput [9]. Channel 1 was chosen because this was the least busy channel, with only sporadic beacon packets from another access point that had no clients connected to it at the moment of the experiment.

Additionally, a TP-LINK TL-WN722N dongle was connected to the laptop and put into monitor mode. Wireshark, a network sniffer, was started and configured to listen on the monitor interface of the dongle. Display filters were then added to filter out background packets of nearby devices by adding the source MAC-addresses of packets sent before the test to a deny-list. This allows us to see the packets sent by the FPGA that get accepted by a wireless card. It must be noted that these are not necessarily the same packets that get accepted by the wireless card of the laptop the test is conducted on: the chipsets are different, so the underlying implementation might be different, leading to possible different behaviour.

The experiment started by taking a baseline measurement of the throughput bitrate of the laptop to the iperf3 server over under no special conditions; it was measured to be a stable $20\,\mathrm{Mbit/s}$. The FPGA was then started and configured to transmit a minimal frame with the `inject_80211` command that makes the FPGA transmit in mode 802.11a, with payload size 1, rate index 0 ($6\,\mathrm{Mbit/s}$), for 180 packets with a delay of $500\,\mu\mathrm{s}$ between packets. This command is executed continually in a loop to work around an apparent bug where the `inject_80211` program would stop transmitting after 180 packets. The packets that are sent can be seen in the Wireshark packet dump (originating from MAC address `66:55:44:33:22:11`, the default source MAC address of the `inject_80211` program). The iPerf3 test was run again and no noticeable effect was seen on the throughput, it remained around $20\,\mathrm{Mbit/s}$.

The FPGA was then configured to send a minimal packet, but with a modified legacy signal header indicating the maximum length (4095 bytes), again using the `inject_80211` program; all options options are the same as in the previous setup, except that the signal header is now set to a value of `0xebff03`. This corresponds to `1101 0 1111111111111 0 00000` (in order of transmission, spaces added for clarity): the rate is `1101` (=$6\,\mathrm{Mbit/s}$), the reserved bit is 0, the length is 4095 and the parity bit is correct. This command is once again executed in a loop. The packets that are sent are not passed from the wireless card to the operating system anymore, most likely because there are no (valid) symbols after the small packet. However, a significant drop in throughput is observable in the iPerf3 test: it was reduced to about $5\,\mathrm{Mbit/s}$ (25% of the baseline speed).

A better attack result might be possible if the timing between the packets sent on the FPGA were more accurate: the `inject_80211` program is implemented in user-space and thus is subject to the non-deterministic job scheduling of the kernel. If the wireless card of the laptop does not hear the signal header at the correct time, it will begin transmitting. Since the TX power of that wireless card in the laptop is much larger that that of the FPGA (the SDR board does not have an amplifier), packets sent by the laptop will not be corrupted by packets sent by the FPGA at the same time (the FPGA could not reach the access point)

This attack was also tried with the HT-SIG header: the AP mode was changed from 802.11a to 802.11n. The injected frames are mixed-mode, with an unchanged legacy signal header, but a modified HT signal header: the length is $2^{16}-1$, with MCS index 0 and short guard interval turned off. This would theoretically take $(65\,535\,\mathrm{B})/6\,\mathrm{Mbit/s} = 80.66\,\mathrm{ms}$, which is longer than the maximum duration of a 802.11a frame. This attack however did have no effect on the throughput. This is likely because the packets are sent in mixed-mode: mixed-mode packets include both a legacy and HT signal header. The wireless card used in the experiment likely looks at the length field in the legacy header to determine how long to block sending. Setting the length field in the legacy signal header would not thus result in a significantly different attack from the one previously described. A way to work around this would be to send greenfield frames; these don't include the legacy signal header, receivers thus have to look at the HT signal header. Unfortunately, Openwifi does not have support yet for sending greenfield frames yet, so this could not be tested.

V. Conclusion

The Openwifi project was modified to have support for modifying the legacy and high-throughput signal header. These modifications are both to the gateware and to the driver software. The changes were tested with a simulation testbench, with the loopback feature of Openwifi and by sending packets to a wireless card in monitor mode. With these modifications, a hard-to-detect denial of service attack is possible by sending packets that have a very long length value in their signal header.

A. Future work

The modifications are based on the 1.2.0 leuven release (the latest release at the time of starting this thesis); the current latest release is 1.3.0. In this newer version, the calculation of the PHY header bits moved from the driver to the FPGA, making it hard to implement the injection code in the same way it is implemented now. Future work could include implementing PHY header injection in the latest version and getting it merged into the upstream Openwifi repository. This has several advantages over maintaining a separate source tree: when other work to Openwifi gets done and merged into the upstream repository, we also benefit from that work. It also makes maintaining the injection feature easier since it will be clear for developers refactoring that code that they might break the feature.

Currently, the Openwifi project only supports 802.11 a, g and n. 802.11n introduced the HT signal header. This

thesis supports fuzzing both the legacy signal header and the HT signal header. 802.11ax, also known as Wi-Fi 5, is the successor of 802.11n. It introduced the creatively named VHT (Very High Throughput) signal header. Once 802.11ac is implemented in Openwifi, fuzzing this header should also be possible. This also applies for the `HE-SIG-A` (High Efficiency) field in 802.11ax (Wi-Fi 6) and the `EHT-SIG` (Extremely High Throughput) field in 802.11be (Wi-Fi 7) [10].

Openwifi also doesn't have support yet for sending HT frames in greenfield mode. This might be a useful feature to have: devices under test might take a different code path in greenfield mode compared to mixed mode; with this feature we might have better coverage of the implementation while fuzzing.

References

[1] Wi-Fi Alliance. "Wi-Fi Alliance 2022 Wi-Fi trends." (2022), [Online]. Available: https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-2022-wi-fi-trends (visited on Aug. 4, 2022).

[2] Wi-Fi Alliance. "Security Wi-Fi Alliance." (2022), [Online]. Available: https://www.wi-fi.org/discover-wi-fi/security (visited on Aug. 4, 2022).

[3] M. Vanhoef, "Fragment and forge: Breaking Wi-Fi through frame aggregation and fragmentation," in *Proceedings of the 30th USENIX Security Symposium*, USENIX Association, Aug. 2021.

[4] X. Jiao, W. Liu, M. Mehari, M. Aslam, and I. Moerman, "Openwifi: A free and open-source IEEE802.11 SDR implementation on SoC," in *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, IEEE, 2020, pp. 1–2.

[5] X. Jiao, W. Liu, and M. Mehari. "Open-source IEEE 802.11 WiFi baseband FPGA (chip) design: FPGA, hardware." (2022), [Online]. Available: https://github.com/open-sdr/openwifi-hw (visited on Aug. 1, 2022).

[6] T. Schuddinck, "Cybersecurity: Breaking IEEE 802.11 devices at the physical layer," M.S. thesis, Ghent University, 2022.

[7] X. Jiao, W. Liu, and M. Mehari. "Open-source IEEE 802.11 WiFi baseband FPGA (chip) design: Driver, software." (2022), [Online]. Available: https://github.com/open-sdr/openwifi (visited on Aug. 1, 2022).

[8] X. Jiao and M. Mehari. "Open-source IEEE 802.11 WiFi baseband FPGA (chip) design: FPGA, hardware: dot11_tx.v." (2022), [Online]. Available: https://github.com/open-sdr/openwifi-hw/blob/dc6fc84bd8b1816460b143f70c435cf2b9ca4fda/ip/openofdm_tx/src/dot11_tx.v#L78 (visited on Aug. 1, 2022).

[9] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. "iPerf - the ultimate speed test tool for TCP, UDP and SCTP." (2022), [Online]. Available: https://iperf.fr/ (visited on Aug. 12, 2022).

[10] Keysight Technologies, Inc. "OFDM SIG info (802.11n/ac/ax/be)." (2022), [Online]. Available: https://rfmw.em.keysight.com/wireless/helpfiles/89600b/webhelp/subsystems/wlan-mimo/Content/trc_ofdm_ht-sig_info.htm (visited on Aug. 1, 2022).

Contents

1 Introduction **13**

1.1 Problem statement 13

1.2 Wi-Fi (IEEE 802.11) 14

1.2.1 Physical layer 14

1.2.2 Data link layer 17

1.3 Openwifi 19

1.4 802.11 fuzzers 20

1.4.1 Greyhound 20

1.4.2 Owfuzz 21

1.4.3 Wifuzzit, wifuzz 21

1.5 Shortcomings of current fuzzers 21

2 Design **22**

2.1 FPGA programming 22

2.2 Setup 23

2.3 Modifications to Openwifi 24

2.3.1 FPGA changes 24

2.3.2 Driver changes 25

2.3.3 Injection program changes 26
2.4 Verification 26
2.4.1 Functional simulation 26
2.4.2 In-system testing: FPGA loopback 27
2.4.3 Decoding received signal in simulation 28

3 Results **30**
3.1 Low-energy jamming attack 30

4 Conclusion **34**
4.1 Future work 35
4.1.1 Porting the work to the latest Openwifi version 35
4.1.2 Enabling longer fuzzing runs 35
4.1.3 Widening attack surface 35
4.1.4 Testing devices in the field 36

Bibliography **37**

Appendices **40**

List of Figures

1.1 Diagram showing the layers in the OSI model [5] 15

1.2 Diagram of the legacy signal header introduced in 802.11a [6] 16

1.3 Diagram showing the types of frames possible with 802.11n: a legacy 802.11a frame without HT-SIG header, an 802.11n frame that includes a legacy signal field and an HT-SIG field for backwards compatibility; and a greenfield frame that only includes the HT-SIG field which is used when there are no legacy stations [6]. 17

1.4 Diagram of the HT-SIG header introduced in 802.11n [6] 18

1.5 Diagram of an 802.11 MAC frame [11] 20

2.1 Photo of the FPGA and RF daughter board 23

2.2 Diagram of first dot11 `openofdm_tx` state machine. Starts in `S1_WAIT_PKT` state 25

2.3 The loopback feature of openwifi, showing both FPGA internal loopback and over-the-air loopback 27

2.4 OpenOFDM decoder simulation running on IQ samples captured using the Openwifi loopback feature. The legacy signal header was modified to have the length value be 1337. 29

1 Introduction

1.1 Problem statement

According to the Wi-Fi Alliance, nearly 18 billion Wi-Fi devices will be in use in 2022 [1]. Over the years, more and more security mitigations have been introduced in the Wi-Fi standards: WEP, WPA, WPA2, protected management frames, Opportunistic Wireless Encryption, Simultaneous Authentication of Equals, …[2] However, even if the protocol is secure, the implementation might not always be, as shown by for example CVE-2020-26145, where plaintext frames were accepted as full frames in an encrypted network [3]. Detecting implementation errors manually is a lot of work, made worse by the fact that there are a lot of implementations: there are multiple vendors, most of whom make multiple chipsets, each having a potentially different implementation of the standards. To automate assessing the security of these implementations security, fuzz testing (fuzzing) can be used: with fuzzing, a lot of packets are sent to the device under test, some of which might be invalid or malformed. If the device reacts unexpectedly, the fuzzing software logs the packet flow for a human to take a closer look at. This allows for much quicker security testing compared to manually auditing.

The aim of this thesis is to research the possibility of modifying the Openwifi project to be able to fuzz the physical layer of 802.11. Unlike in [4], this thesis will also change the FPGA implementation, in order to be able to send frames where the PHY hardware parameters used in

the transmission hardware don't match with the parameters sent over the air in the signal field of the frame. This thesis will additionally provide support for fuzzing the PHY layer of frames in HT mode. Finally, using the newly developed features, a hard to detect denial-of-service attack will be carried out.

1.2 Wi-Fi (IEEE 802.11)

Computers can be networked so they can communicate with each other, but doing this in a reliable way is not simple: there are multiple physical ways to communicate data (for example, over radio waves, using copper cables, using fiberglass cable, …) and there's different content you could want to communicate (for example, transferring files, communicating the time, transferring real-time video, …). It would be very costly in engineering time if for every combination of physical medium and application a new protocol needed to be designed from the ground up. To solve this, the way computers communicate is split up into multiple layers, where every layer is responsible for a specific aspect of the communication. These different layers can then each have multiple protocols that can be standardized in order to make devices interoperable with each other. The most common way to describe this protocol stack is the Open Systems Interconnection (OSI) model, a product of the International Organization for Standardization (ISO), shown in Figure 1.1. This way, a combination of protocols can be chosen for every layer, where every layer is responsible for its own aspects.

Wi-Fi is a system that allows nearby computing devices to wirelessly communicate with each other over radio waves. Wi-Fi consists of multiple standards provided by IEEE, namely the 802.11 family of standards. It implements the physical and data link layers of the OSI model.

1.2.1 Physical layer

The physical layer concerns itself with how bits are physically transmitted over the medium. In the case of Wi-Fi (also referred to as the 802.11 standards), it's about how the bits are encoded in radio waves. There are 3 main ways to encode (also called modulate) information in radio waves: in the amplitude, frequency and phase of the waves; it's also possible to use a combination of these three. Wi-Fi is a set of standards that have had new versions through the years, and as such also has multiple physical layers and ways to modulate data. The modulation that will be used in the rest of this thesis is orthogonal frequency-division multiplexing (OFDM), where multiple conventionally modulated signals are modulated using multiple orthogonal subcarriers.

In the case of 802.11, the physical layer itself is divided into two sublayers: the Physical Layer Convergence Procedure (PLCP) layer and the Physical Medium Dependent (PMD) layer. The

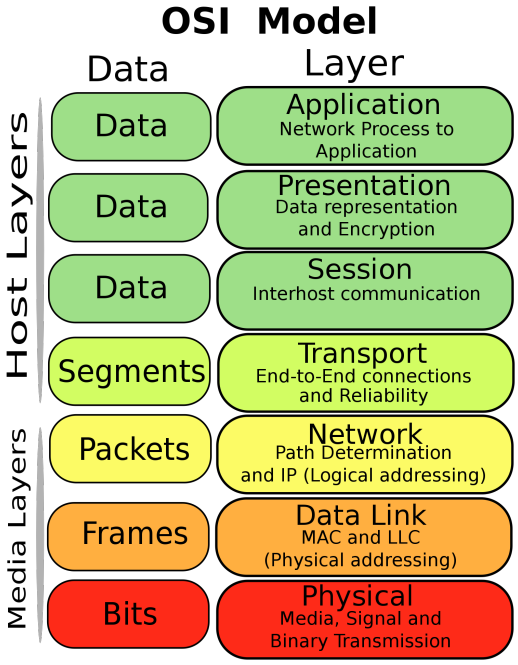

Figure 1.1: Diagram showing the layers in the OSI model [5]

PMD layer is the 'bottom-most' layer in the OSI model and is responsible for the modulation details. The PLCP layer is responsible for adding a preamble (if the modulation requires it) and to add its own header that holds information needed for decoding the signal (data rate, length, …). For OFDM, every physical frame starts with a preamble, followed by the PLCP header or headers, then the data, finished with a trailer/padding.

In 802.11a, the first 802.11 standard to use OFDM, the PLCP header contains the (legacy) signal field, shown in Figure 1.2. The data in the field is protected with a convolutional code for error correction. The field contains 5 properties (in order):

- rate (4 bits): this describes the modulation and coding rate of the following data. This can be found in section 4.1.4)
- reserved (1 bit): this bit is reserved for future use and should for now always be 0; no standard as of yet uses it
- length (12 bits, least significant bit first): this subfield contains the length, in bytes, of the following MAC data part
- parity (1 bit): should be even parity with the preceding 17 bits to detect errors
- tail (6 bits): should be all zeroes, used in the convolutional code when decoding the signal field to correct errors [6] [7]

In order to improve throughput, an amendment to the 802.11 standards called 802.11n was

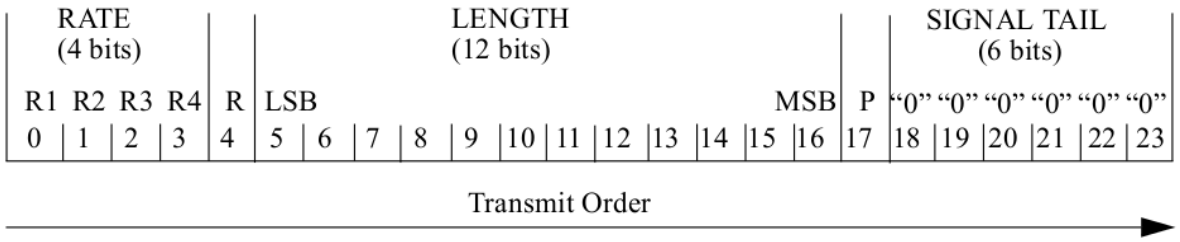

Figure 1.2: Diagram of the legacy signal header introduced in 802.11a [6]

introduced in 2009. It has more and faster data rates, an optional shorter guard interval (the interval between two symbols that is used to avoid inter-symbol interference) and other features with the goal of increasing the transfer rate with a factor 10. [8]. One of the changes is to the PCLP header, as shown in Figure 1.3, is to have another signal field, called the High-Throughput Signal field (HT-SIG). This HT signal field contains information that holds the parameters needed for new features in 802.11n, shown in Figure 1.4. This signal header now takes up 2 24-bits words, where the legacy signal header only used 1. It contains the following subfields:

- Modulation and Coding Scheme (MCS) (7 bits): this is a better name for the 'rate' parameter as described in the legacy 802.11a signal header since in addition to the rate, it also describes the modulation and how much error correcting redundancy there is in the coding. The values can be found in section 4.1.4.
- Channel Bandwidth (1 bit): indicates if the channel bandwidth is 20 or 40 MHz
- HT length (16 bits): number of bytes in the PSDU
- Some single-bit and 2 bit fields that indicate various new 802.11 features
- CRC (8 bits): this is a checksum of the previous bits in the HT-SIG header, used for error detection
- Tail: 6 all-zero bits, used in the convolutional code when decoding the HT-SIG field to correct errors [9] [6]

Since 802.11n has to be compatible with previous versions, there's a mode called 'HT-mixed format PPDU' for stations that don't understand 802.11n. In this mode, the HT-SIG header is prefixed with the legacy preamble and signal field. The rate in that signal field is fixed to 6 Mbps (rate bits 1101); the length field then no longer indicates the length of the PSDU, but is set in such a way that the duration of the following data matches what would be the length. This ensures that stations that receive this legacy header don't start transmitting halfway into the signal and thus corrupt it. Stations that do support 802.11n detect this HT-SIG field by using two conditions: the rate field should be 1101 (6 Mbps). If this is the case, then the next symbol will be Binary Phase Shift Keyed (BPSK). The legacy signal is modulated using the I-component, the HT-SIG will be modulated using the Q-component. The receiver can thus

compare the amount of samples in which the quadrature component is bigger than the in-phase component and then determine if the HT-SIG header is present.

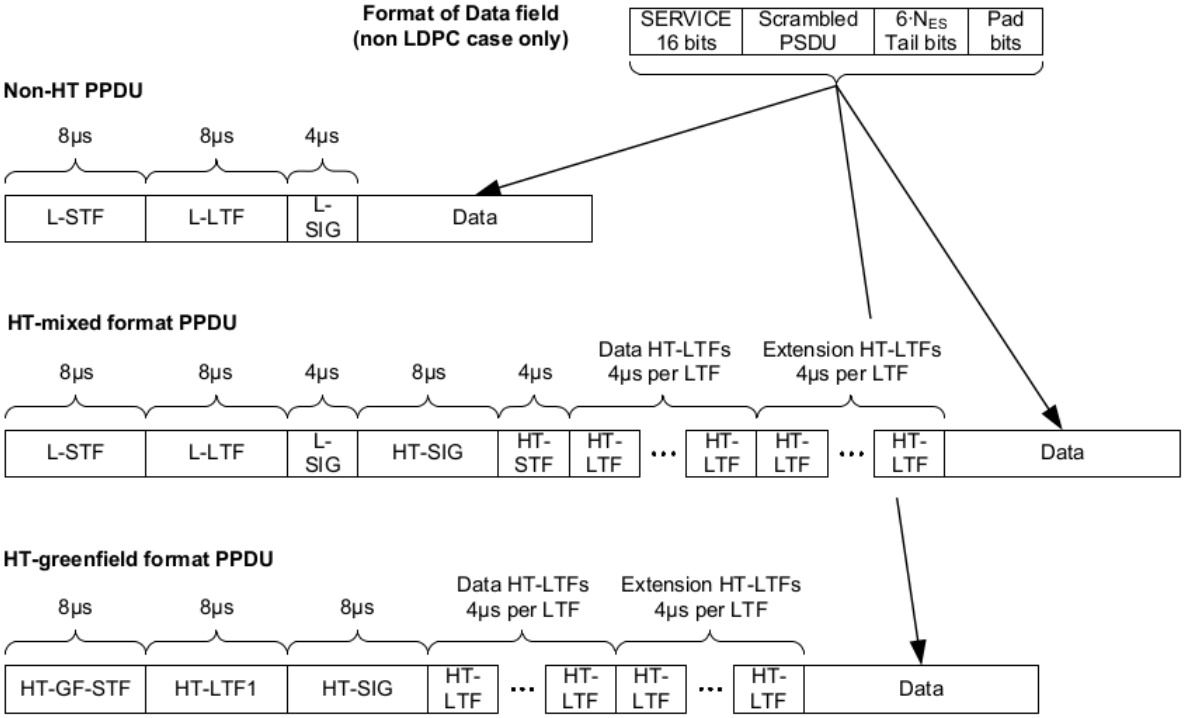

Figure 1.3: Diagram showing the types of frames possible with 802.11n: a legacy 802.11a frame without HT-SIG header, an 802.11n frame that includes a legacy signal field and an HT-SIG field for backwards compatibility; and a greenfield frame that only includes the HT-SIG field which is used when there are no legacy stations [6].

802.11n is followed by 802.11ac, 802.11be and 802.11ax which define their own, new signal fields. Since those will not be used in this thesis, we won't go into further detail about them. Referring to the different amendments to the 802.11 standard by their letter code got confusing for consumers, especially since they are non-alphabetic, so starting from 802.11n, the amendments were referred to as 'Wi-Fi **x**' with **x** a number. See Table 1.1 for a list of Wi-Fi versions and their associated signal header.

1.2.2 Data link layer

The physical layer makes it possible to send encoded bits over the air, but receiving stations don't yet know if the data in the frame is addressed to them. This is handled by the data link layer. This layer is also split into two sublayers: the Media Access Control (MAC) layer and the Logical Link Layer (LLC). The MAC layer specification is unique for 802.11, the LLC specification (802.2) is shared between multiple standards.

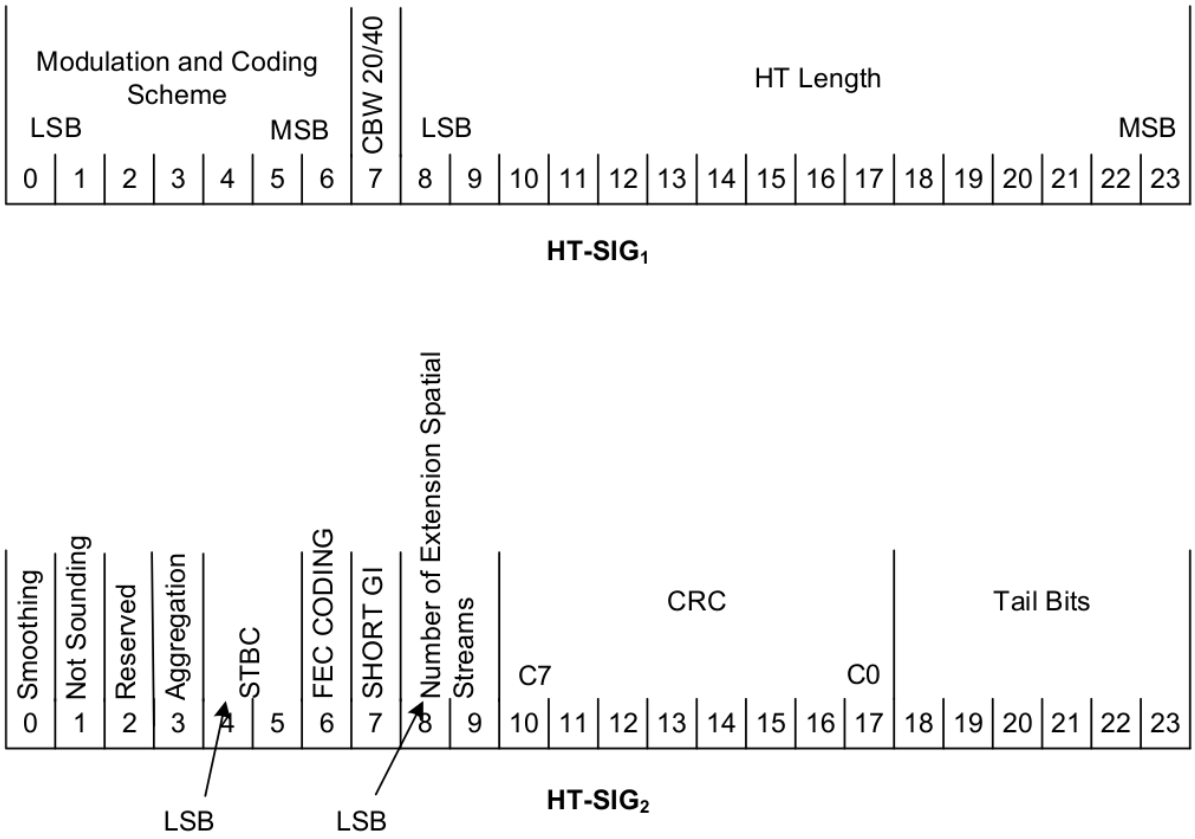

Figure 1.4: Diagram of the HT-SIG header introduced in 802.11n [6]

The MAC layer is the lowest layer in the data link layer, directly above the physical layer. It's responsible for collision avoidance/detection. Since the transmission capacity is very limited, 802.11 has multiple ways to avoid collisions in order not to waste the valuable transmission capacity.

802.11 uses both physical and virtual carrier sensing: physical carrier sensing looks at the activity of the RF medium. However, this is not enough: it does not solve the "hidden node probem": if node B is in reach of node A and C but node A and C are not in range of each other, it could happen that A and C transmit at the same time, thus corrupting the frame. This can be avoided by using virtual carrier sensing. Here, stations send a Request To Send frame before transmitting a frame. After the receiving station confirms that the medium will be free by sending a Clear To Send (CTS) response, the original station sends it frame. Other stations that hear the RTS or CTS will not transmit for a certain time (indicated in the duration field of the RTS and CTS packets).

The MAC frame (as can be seen in Figure 1.5) includes space for up to 4 MAC addresses: the destination, source, but also the receiver address and, in case of bridging, the transmitter

Version number	IEEE 802.11 amendment	Adopted	Signal header
Wi-Fi 2	802.11a	1999	L-SIG
Wi-Fi 4	802.11n	2008	HT-SIG
Wi-Fi 5	802.11ac	2014	VHT-SIG
Wi-Fi 6/6E	802.11ax	2019	HE-SIG
Wi-Fi 7	802.11be	not yet	EHT-SIG

Table 1.1: Evolution of 802.11 signal header over time. [10] [9]

address. After the data, it includes a checksum (FCS) to ensure the data is valid (or to at least have a very small probability that it was corrupted). The type and subtype fields identify the type of of frame; there are 3 basic types of frames: data, control and management. The data frames (evidently) carry data. Control frames are short messages (they don't have a body) to aid with reliable delivery of data and management frames (for example, CTS and RTS). Management frames are related to managing the BSS, for example association request, authentication, de-authentication, ...[7]

This thesis will stay within the physical layer, so no detailed explanation of the higher layers will be given.

1.3 Openwifi

Openwifi is an open-source full-stack implementation of IEEE802.11 a/g/n on an FPGA (Field Programmable Gate Array) based on software defined radio [12]. The Openwifi project was created to solve the problem that there was no open-source reference implementation for researchers to test their ideas against: they would have to use simulations, hacked commercial chips or expensive reference designs to test their ideas. It is based on Xilinx Zynq System-on-Chip (SoC) that includes FPGA and ARM processor. The Openwifi design is based on FPGA technology to keep the latency low enough: the Wi-Fi Short Inter-Frame Space (SIFS) requires acknowledgement (ACK) frame being sent out in 10µs/16µs (2.4GHz/5GHz) after receiving a frame successfully. The critical SIFS timing is achieved by implementing the Physical layer (PHY) and low level Media Access Control (low MAC) in FPGA. This could not be done on PC-based Software-Defined Radio (SDR), because of the latency ($\geq$ 100µs) between the PC and the SDR front-end.

The Openwifi hardware consists of 2 main parts, the ARM SoC and the FPGA. The physical layer and low level Media Access Control is implemented on the FPGA (which is connected to the RF frontend); everything above that in the 802.11 stack is implemented on the ARM SoC. The driver implements the standard Linux mac80211 API.

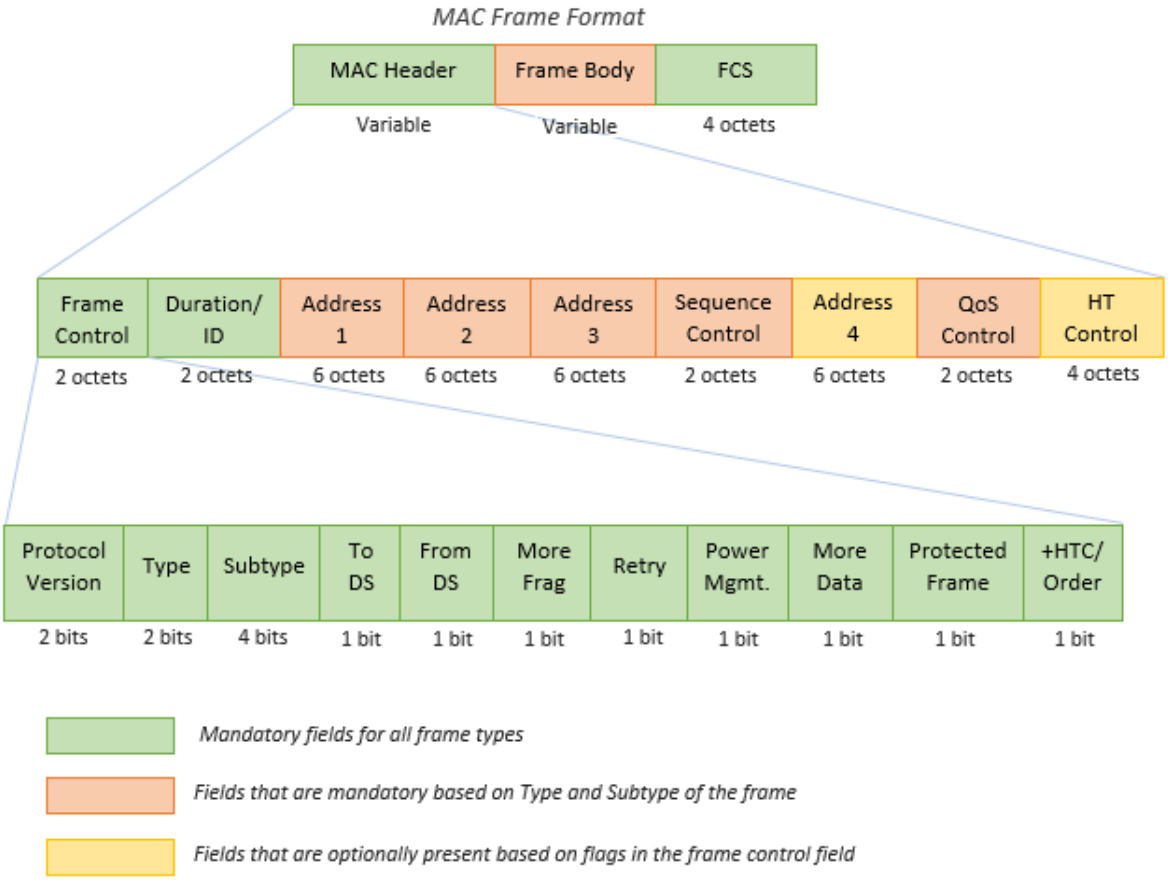

Figure 1.5: Diagram of an 802.11 MAC frame [11]

1.4 802.11 fuzzers

There are multiple existing fuzzers for Wi-Fi, some of them under open-source licenses, some of them under closed-source licenses.

1.4.1 Greyhound

Greyhound [13] is a closed-source grey-box Wi-Fi fuzzer developed by Keysight that uses a manually built Finite State Machine of the Wi-Fi protocol to detect invalid state transitions of the device under test (DUT). It fuzzes the DUT by both mutating fields in packets and by sending valid packets that are incorrect in the current state of the FSM. The fuzzer only uses the response of the DUT to detect misbehaviour: if the received packet does not match the expected packet (or no packet is received), this is marked as misbehaviour.

1.4.2 Owfuzz

Owfuzz [14] is an open-source Wi-Fi fuzzer (licensed under the GNU General Public License v3.0, although some components are under other open-source licenses) that uses Openwifi as a platform. It also supports using other wireless network cards, as long as they support monitor mode and frame injection.

1.4.3 Wifuzzit, wifuzz

Wifuzzit and wifuzz are open-source Wi-Fi fuzzers (licensed under the GNU General Public License v3.0) that are not actively developed anymore: the last commit to their software repository happened about 10 years ago for both.

1.5 Shortcomings of current fuzzers

Current (publicly known) fuzzers don't support fuzzing the PHY layer and although Owfuzz uses Openwifi as the platform, it does not modify any FPGA code, it only uses the standard Linux API's to send packets.

2

Design

2.1 FPGA programming

In this section, we will explain what an FPGA is and why programming it is different from imperative, sequential programming.

An FPGA, short for Field Programmable Gate Array, is an integrated circuit that is a re-programmable (Field Programmable) network of logic gates (Gate Array). It enables developers to create custom circuits without the expensive and time-consuming process of creating an ASIC (application-specific integrated circuit).

FPGA programming is different from imperative programming in several ways: instead of a program being evaluated sequentially, everything happens everywhere, all at once. A major advantage of using FPGAs over traditional microcontrollers is their superior processing speed and parallellism, which makes them suitable for real-time, low latency applications and signal processing.

Internally, FPGAs work by having programmable logic blocks that are in turn connected by programmable routing. These configurable logic blocks (CLB) are commonly implemented by the use of a LookUp Table (LUT) and a flip-flop. The LUT has n input bits and 1 output bit and can realise any n bit boolean function. The function of the flip-flop is to prevent glitches

caused by intermediate outputs while the input bits change: the flip-flop guarantees that the output can only change on a clock edge. In addition to CLBs, FPGAs also contain specific purpose blocks like block RAM, multipliers, …. These would be very expensive to implement in CLBs because of their complexity, so they are included as 'hard blocks' in the FPGA. This is a trade-off: their function is fixed, but they come at a much lower area cost [15].

FPGAs are programmed in a Hardware Description Language (HDL) like Verilog. HDLs abstract some the internal details of the FPGA away, making it possible to program with higher constructs like numbers, conditionals, …instead of having to program the LUTs and how they are connected. The code can be compiled into a 'bitstream', a file that contains the configuration of the FPGA.

2.2 Setup

The Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit was used as FPGA, with the Analog Devices AD-FMCOMMS4-EBZ as RF daughter board. The RF daughter board was connected via the FPGA Mezzanine Card (FMC) interface of the FPGA.

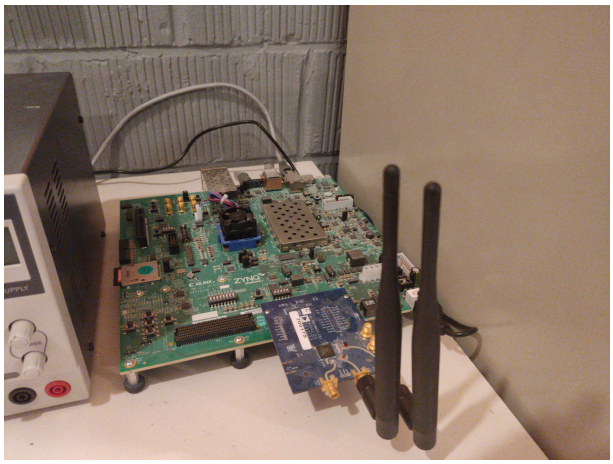

Figure 2.1: Photo of the FPGA and RF daughter board

Since Vivado (the toolchain used to build the HDL files into a bitstream) is rather particular of the environment it runs on, a build server was used to build the FPGA code. This server ran Ubuntu 18.04, with `en_US.UTF-8` as locale: if for example `nl_BE.UTF-8` is used, the TCL interpreter used in the build scripts will no longer interpret dots (`.`) as the floating point separating character, causing scripts to fail. Since Vivado is GUI-based, VNC was set up to not have to be physically present at the build server. SSH and SCP were used to transfer the output of the build process to the development computer. The remote editing feature of Visual Studio Code was used to develop code on the build server.

2.3 Modifications to Openwifi

The modifications to Openwifi consist of two parts: the modifications to the FPGA Verilog code, and the modifications to the driver, written in C. The Openwifi codebase is split over two git repositories: the openwifi repository, containing the documentation, driver and Linux user space; and the openwifi-hw repository, containing the FPGA source code. The modifications have been written on top of the 1.2.0 (codename leuven) release, since that was the latest public release at the moment of starting this thesis. For the openwifi repository, this is on top of commit `d692cb8dcc40b8ce4844ede037dfde88c60534a8`, for the openwifi-hw repository on top of commit `968156843568777dabaf49d07b01d4746fed9ba9`. Note that this is not the latest version that's currently available at the time of writing this thesis: the next release, 1.3.0 (codename wilsele), significantly restructured the signal header code, both in the FPGA and driver code.

2.3.1 FPGA changes

Thanks to the modularity and separation of concerns in the Openwifi FPGA code, only one module needed to be modified to allow signal field modification: `openofdm_tx`. This module is responsible for modulating the bits in the 802.11 frames into a modulated signal; it implements the PHY and lower MAC layer [16].

The `openofdm_tx` module consists of several state machines: the data collection state machine, the IQ sample generation state machine and the IQ sample forwarding state machine. Since we only need to modify what's sent and not the modulation of the signal itself, we only need to modify code in the first state machine.

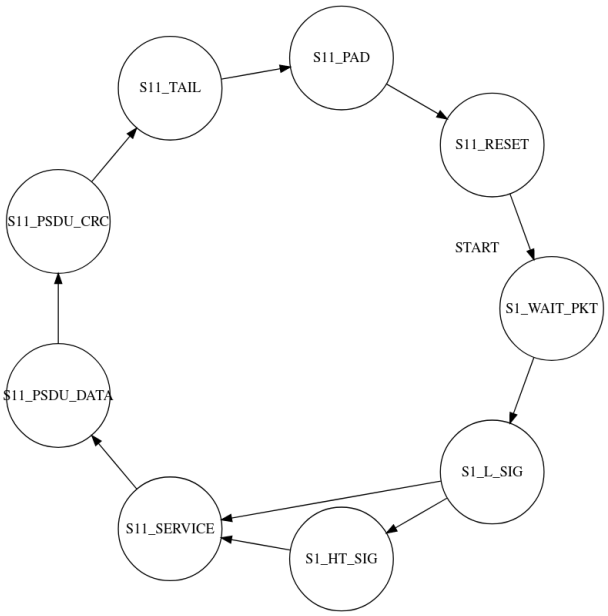

Figure 2.2: Diagram of first dot11 `openofdm_tx` state machine. Starts in `S1_WAIT_PKT` state

.

In particular, the code in the `S1_L_SIG` and `S1_HT_SIG` states need to be modified. The FPGA design uses a 64 bit data word to encode the 24 bit legacy signal header. This means that there are still 40 bits left over to put the hardware parameters in. This modification was very minimal, only some offsets into the FPGA data words needed to be incremented by 32: the transmitted header is in `bram_din[24:0]`, the hardware parameters are, in the same format, in `bram_din[24+32:32]`.

For the High Throughput header, the modifications weren't as trivial: since the HT signal header is 48 bits long, there's not enough bits left over in the 64 bit FPGA word to squeeze in the HT hardware parameters. Another FPGA word thus needed to be used to hold the HT hardware parameters. In order for the FPGA to still write one bit per clock cycle for the bit source selection and scrambling, while transmitting the first bits of the HT signal header, we save the current block RAM word and pre-read the next block RAM word into a register. This saved register can then be used to read the hardware parameters when they are needed.

2.3.2 Driver changes

The driver also needed to be changed to account for the changed hardware: for the legacy signal modifications, the block ram word that contains the legacy signal header needs to also contain the hardware parameters. For the High Throughput change, the size of the block RAM allocated for the PHY header needed to increased from 16 to 24 bytes and these bytes need to be

filled in. In the master's thesis of another student [17], it's described how to pass the hardware parameters to the driver by using the 8 byte timestamp field in the Radiotap header. That way, the additional signal header data can be passed from user space to kernel space without having to modify the driver API. This is not a particularly stable approach however: programs that are not aware of this feature might have an actual timestamp in the timestamp header. The implementation also assumes the timestamp field is at a fixed offset in the variable-size Radiotap header, causing it to only work correctly for one program.

2.3.3 Injection program changes

There is a program called `inject_80211` in the Openwifi repository that allows sending frames with specific rates, lengths, …This program was modified to take an additional two command line flags: `-c` sets the legacy signal header, `-f` sets the HT signal header. These are then passed via the timestamp field to the driver.

2.4 Verification

An important part of writing code is making sure that the code that was written actually works and does what it's supposed to do. There are multiple ways to do this for hardware description code: among others formal verification (proving the correctness of the implementation compared to a formal specification), functional simulating (simulating the circuit and checking that the output for a given input matches the expectation) and in-system testing (uploading the generated bitstream to real-life hardware and then checking it passes certain tests) [18].

We decided not to use formal verification to verify the modifications work since it is a very time-consuming process, no other parts of Openwifi are formally verified and even if the result is formally verified, there might be errors in the formal model. Instead of formal verification, we first did a functional simulation. Once these tests passed, we verified that the system also works in real life on the board.

2.4.1 Functional simulation

For the functional simulation, two Openwifi modules were used: `openofdm_tx` (this is the module we modified to be able to send the modified signal field) and `openofdm_rx` (this module takes IQ samples and tries to decode those into 802.11 frames).

Both modules have a Verilog test bench that allows developers to run a simulation. The

`openofdm_tx` module's testbench (`dot11_tx_tb.v` connects a `dot11_tx` instance to the contents of a file instead of to block RAM. The IQ samples generated in the simulation are then written to a file. Special care needed to be taken to keep the correct memory layout of the input file: the modifications for the HT field increased the size of BRAM for the physical header from one 64-bit word to two 64-bit words; the test vectors in the Openwifi repository thus need to be modified to account for this.

The `openofdm_rx` module's testbench (`dot11_tb.v`) reads from a file containing IQ samples (where every sample has I and Q as 16 bit integers, formatted as decimal numbers separated by a space and terminated with a newline) and writes the decoded frames to multiple text files. The files that are interesting for verification are `signal_out.txt`, containing the decoded legacy signal fields and `byte_out.txt`, containing the decoded data. The HT signal fields were not logged to a file yet, but a small modification to the test bench made it possible to also log those to a file.

2.4.2 In-system testing: FPGA loopback

After the testbenches showed that the modifications worked in simulation, the Verilog code was compiled into a bitstream and the patched driver was compiled. The `openwifi-1.2.0-leuven-2-64bit.img` image was written to the SD-card, after which the existing bitstream and driver files were overwritten with the newly compiled ones.

The Openwifi project is full duplex; this means that the FPGA can transmit and receive at the same time. This makes it possible to receive the a signal that was transmitted from the FPGA and then save the IQ samples or decode the sent signal. As shown in Figure 2.3, this can be done both internally in the FPGA (so there is no interference) or over the air (for a more realistic test scenario) [19].

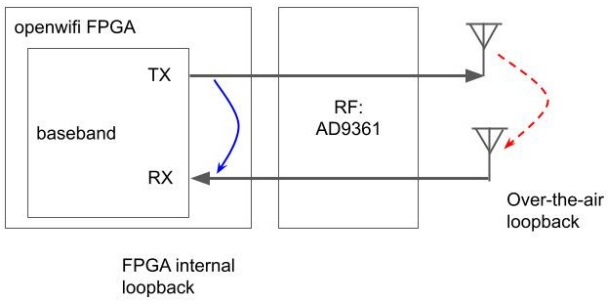

Figure 2.3: The loopback feature of openwifi, showing both FPGA internal loopback and over-the-air loopback

The loopback feature did not work out of the box; several modifications to the provided instruc-

tions were needed to get the FPGA to transmit the modified signal.

If two 802.11 stations transmit at the same time on the same channel and they are both received by a receiver, neither of the frames will be decodeable. To prevent stations from transmitting when the channel is in use, a feature called CCA (Clear Channel Assessment) is used. The CCA is composed of carrier sensing to detect if there is another 802.11 transmission in progress and energy detection to detect if the channel is in use by another transmitter. With carrier sensing, the receiver can then calculate when the next opportunity to send a frame will be. The purpose of energy detection is not obvious at first: if carrier sensing is already implemented and it can predict when the channel will be idle again, why then have another feature that can only detect if the channel is in use now? The answer lies in the fact that 802.11 is commonly used in frequency bands that are licensed as ISM (Industrial, Scientific, Medical) bands. This means that transmitters that use another protocol than 802.11 are also allowed to send in those bands. As an example, the 2.4GHz band is shared with radio amateurs, Bluetooth, Zigbee, …[20], [21]

When the loopback feature was used, the FPGA did not transmit. It eventually was determined that this was because of the CCA feature: the CSMA (Carrier Sense Multiple Access) engine always detected that the channel was in use. The CCA feature was then turned off by writing a very high value into the CCA threshold configuration register. This causes the CSMA engine to always behave as if the channel was idle, since there won't be any incoming signal that's larger than this threshold.

The board continually captures the IQ samples into a circular buffer. When the (configurable) trigger condition is met, it sends the content of this buffer over UDP to a Python program that will then show those IQ samples in several graphs and save them to a file for later analysis. In the older 1.2.0 version of Openwifi, there was an edge condition where if the pre-trigger-lenght (the amount of IQ samples that are captured before the trigger condition happens) is 0, no capture would happen. This was fixed by setting the pre-trigger-length to 100. This still leaves plenty of space for the actual transmission, the buffer holds 8187 samples.

2.4.3 Decoding received signal in simulation

The existing Python script that received the IQ sample UDP packet originally wrote the IQ samples to a file in a format that makes it easy to load into MATLAB for further analysis. MATLAB has a WLAN Toolbox™ software package that has support for decoding 802.11 frames from IQ samples. We however chose not to use MATLAB for this thesis since it is closed-source and one has to have a (paid) license that's behind a 'Request a quote' page to use it. Instead, we opted to use the OpenOFDM 802.11 OFDM decoder that is included in the openwifi-hw repository. This forked OFDM decoder does not functionally differ significantly from the upstream OpenOFDM repository; the main changes consist of restructuring the code

to centralize the tests. The OFDM decoder was used in the Vivado simulator. The file format needed to be changed slightly, as previously described. The I and Q components also needed to be swapped to work around an error in the implementation where the I and Q components were accidentally swapped.

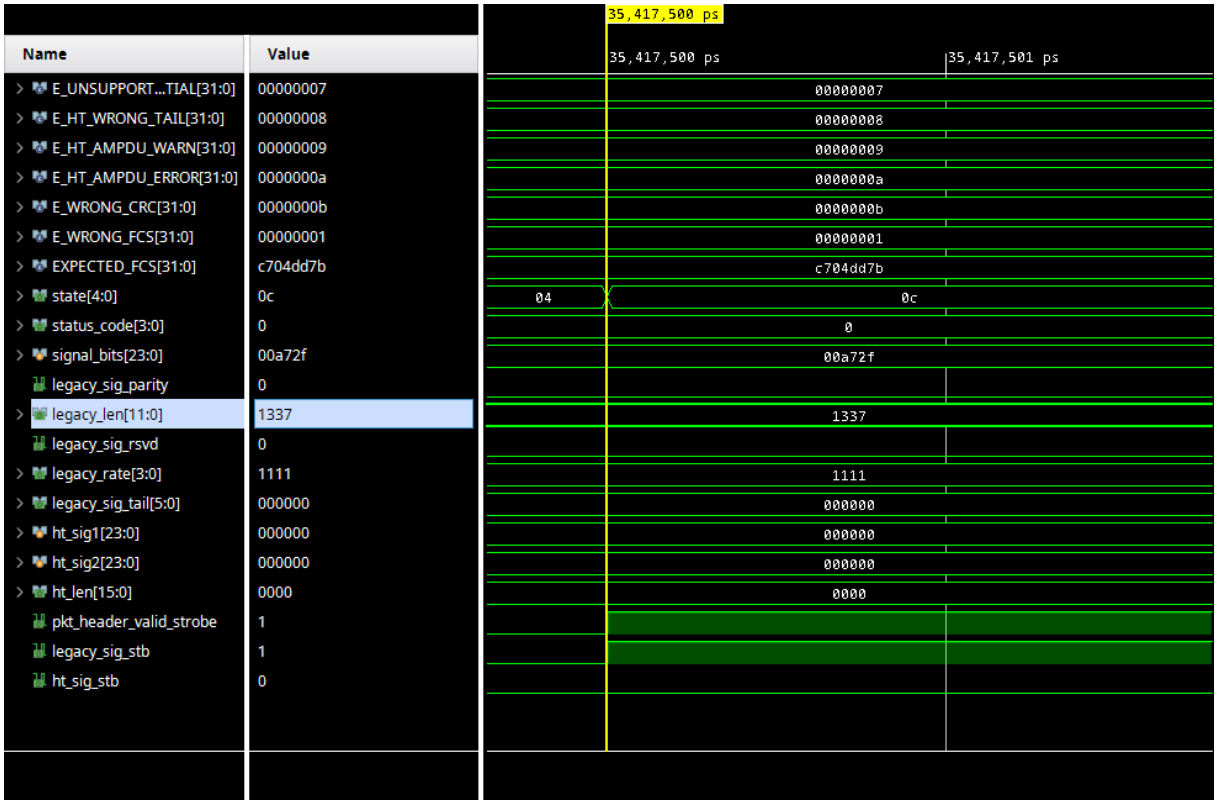

Figure 2.4: OpenOFDM decoder simulation running on IQ samples captured using the Openwifi loopback feature. The legacy signal header was modified to have the length value be 1337.

3 Results

The functional and in-system tests showed that the header modification mechanism is both functional and correct. Loopback tests show that both L-SIG and HT-SIG spoofing is possible. Unmodified frames sent by the FPGA are still decodable by a wireless card in monitor mode.

3.1 Low-energy jamming attack

In [22], it is described how the P25 digital two-way radio protocol that is primarily used by public safety users can be efficiently jammed by jamming certain bits in the voice frames. These 64 jammed bits then cause the entire 1728 bit frame to be discarded by the receiver. This makes a very low power jamming attack possible: only 3.7% of the bits sent by the transmitter need to be jammed.

Unlike the narrowband P25 protocol, Wi-Fi is a spread spectrum signal. This makes it much harder to jam: for a narrow band signal, the jammer needs to be 3 to 6 dB more powerful than the legitimate transmitter; a spread spectrum signal jammer needs to have 30 dB more power than the legitimate transmitter if the spectrum is sufficiently wide [22].

However, the modifications to the signal field make it possible to send frames with an arbitrary signal header. It's for example possible to send a frame that's very short, but with the signal

header indicating the slowest possible rate (6 Mbit/s) and the longest possible length: the size of the length field is 12 bits, so the length field could indicate that the length of the sent message is 4095 bytes long. Each OFDM symbol contains 24 bits, so in total the biggest frame could contain 1367 OFDM symbols [23].

These 1367 symbols would take $(1367\,\text{symbol})/(6\,\text{Mbit/s}) * (24\,\text{bit/symbol}) = 5.468\,\text{ms}$ to transmit. When receiving such a signal header, the receiver would be tied up for 5.468 ms and thus be unable to receive any other signals. With our new capabilities of modifying the L-SIG header, it's possible to continually send a very short frame with a spoofed L-SIG header that indicates the maximum possible size and slowest possible rate and in that way jam other transmitters.

This attack was experimentally tested by setting up an access point and measuring the throughput under different conditions. The access point is an off-the-shelf ASUS AC1300 running OpenWrt 18.06, configured to run a single WPA2 access point in 802.11a mode on channel 1 (2412 MHz). This access point is connected to a local area network (LAN) in which an iPerf3 server is running; iPerf3 is a software package that can test throughput [24]. Channel 1 was chosen because this was the least busy channel, with only sporadic beacon packets from another access point that had no clients connected to it at the moment of the experiment. The length value in the unmodified legacy signal was 33.

Additionally, a TP-LINK TL-WN722N dongle was connected to the laptop and put into monitor mode. Wireshark, a network sniffer, was started and configured to listen on the monitor interface of the dongle. Display filters were then added to filter out background packets of nearby devices by adding the source MAC-addresses of packets sent before the test to a deny-list. This allows us to see the packets sent by the FPGA that get accepted by a wireless card. It must be noted that these are not necessarily the same packets that get accepted by the wireless card of the laptop the test is conducted on: the chipsets are different, so the underlying implementation might be different, leading to possible different behaviour.

The experiment started by taking a baseline measurement of the throughput bitrate under no special conditions; it was measured to be a stable 20 Mbit/s. The FPGA was then started and configured to transmit a minimal frame with the following command:

```
./inject_80211 -m a -s 1 -r 0 -n 180 -d 500 sdr0
```

This command makes the FPGA transmit in mode 802.11a, with payload size 1, rate index 0 (6 Mbit/s), for 180 frames with a delay of 500 µs between frames. This command is executed continually in a loop to work around an apparent bug where the `inject_80211` program would stop transmitting after 180 frames. The frames that are sent can be seen in the Wireshark packet dump (originating from MAC address `66:55:44:33:22:11`, the default source MAC address of

the `inject_80211` program). The iPerf3 test was run again and no noticeable effect was seen on the throughput, it remained around 20 Mbit/s.

The FPGA was then configured to send a minimal frame, but with a modified legacy signal header indicating the maximum length (4095 bytes) with the following command:

```
./inject_80211 -m a -s 1 -r 0 -n 180 -d 500 -c 0xebff03 sdr0
```

All options are the same as in the previous command, except that the signal header is now set to a value of `0xebff03`. This corresponds to `1101 0 1111111111111 0 00000` (in order of transmission, spaces added for clarity): the rate is `1101` (=6 Mbit/s), the reserved bit is 0, the length is 4095 and the parity bit is correct. The actual length of the packet is only 33 bytes. This command is once again executed in a loop. The frames that are sent are not passed from the wireless card to the operating system anymore, most likely because there are no (valid) symbols after the small frame. However, a significant drop in throughput is observable in the iPerf3 test: it was reduced to about 5 Mbit/s (25% of the baseline speed). An interval of 500 500 µs was chosen over the longer theoretical interval of approximately 5000 µs to increase the chances the adapter under attack will be in the correct state to receive the modified frame and block transmitting its own frame.

A better attack result might be possible if the timing between the frames sent on the FPGA were more accurate: the `inject_80211` program is implemented in user-space and thus is subject to the non-deterministic job scheduling of the kernel. If the wireless card of the laptop does not hear the signal header at the correct time, it will begin transmitting. Since the TX power of that wireless card in the laptop is much larger than that of the FPGA (the SDR board does not have an amplifier), frames sent by the laptop will not be corrupted by frames sent by the FPGA at the same time.

Note that this attack is hard to detect: most commercial Wi-Fi chipsets don't pass invalid frames up to the next layer in the network stack, not even in monitor mode. Detecting this attack would thus require special hardware; the Openwifi code could be modified to detect this kind of attack.

This attack was also tried with the HT-SIG header: the AP mode was changed from 802.11a to 802.11n. The baseline iPerf3 benchmark then indicated the throughput was 100 Mbit/s. The following command was used to send modified frames:

```
./inject_80211 -m n -f 0x00ffff040c01 -s 1 -r 0 -n 180 -i 0 -d 500 sdr0
```

These frames are mixed-mode, with an unchanged legacy signal header, but a modified HT

signal header: the length is $2^{16} - 1$, with MCS index 0 and short guard interval turned off. This would theoretically take $(65\,535\,\mathrm{B})/6\,\mathrm{Mbit/s} = 80.66\,\mathrm{ms}$, which is longer than the maximum duration of a 802.11a frame. This attack however did have no effect on the throughput. This is likely because the frames are sent in mixed-mode: mixed-mode frames include both a legacy and HT signal header. The wireless card used in the experiment probably looks at the length field in the legacy header to determine how long to block. Setting the length field in the legacy signal header would not thus result in a significantly different attack from the one previously described. A way to work around this would be to send greenfield frames; these don't include the legacy signal header, receivers thus have to look at the HT signal header. Unfortunately, Openwifi does not have support yet for sending greenfield frames, so this could not be tested.

Modifying other fields in the legacy and signal headers is also possible and was verified to work, but was not used in experiments due to time constraints.

4

Conclusion

The Openwifi project was modified to have support for modifying the legacy and high-throughput signal header. These modifications are both to the gateware and to the driver software. The changes were tested with a simulation testbench, with the loopback feature of Openwifi and by sending packets to a wireless card in monitor mode.

This is the first thesis in a series of master's theses on physical layer fuzzing that succeeded in modifying the physical layer [4], [17], [25]. The other theses modified only software, which makes their fuzzing limited to the internals of the Openwifi project. This thesis enables security research into the 802.11 physical layer of other devices.

A novel denial-of-service attack that is hard to detect by traditional means was found. This attack needs significantly less transmit power than a 'dumb' jammer and is harder to detect than a deauth-attack since wireless adapters don't pass on the modified frames to the host computer.

4.1 Future work

4.1.1 Porting the work to the latest Openwifi version

The modifications are based on the 1.2.0 leuven release; the current last release is 1.3.0. In this newer version, the calculation of the PHY header bits moved from the driver to the FPGA, making it hard to implement the injection code in the same way it is implemented now. Future work could include implementing PHY header injection in the latest version and getting it merged into the upstream Openwifi repositories. This has several advantages over maintaining a separate source tree: when other work to Openwifi gets done and merged into the upstream repository, we also benefit from that work. It also makes maintaining the injection feature easier since it will be clear for developers refactoring that code that they might break the injection feature.

4.1.2 Enabling longer fuzzing runs

Since the Xilinx Viterbi Decoder IP core was used with the evaluation licence, fuzzing runs would be limited in time. Per [26], this core will be fully functional for two to three hours, after which the core will cease to function and the FPGA needs to be power cycled. This makes it hard to do longer fuzzing runs: if the board needs to be power cycled every two hours, the fuzzer state needs to be saved and restored every time. Buying the full license for the Xilinx Viterbi Decoder is very expensive, on the order of multiple thousands of US dollars [27] and since one of the stated goals of the Openwifi project is to avoid expensive reference designs [12], this also does not seem like the ideal solution. Since the Xilinx Viterbi Decoder is the only IP core that needs a separate license, replacing it with an open-source viterbi decoder solves two problems: it removes the evaluation time out and makes managing licenses easier, since all other needed licenses are included in the (free) license of the Vivado IDE. Work has started on implementing an open-source Viterbi decoder in [28]; the simulation results show that the open-source decoder can correctly decode, the integration into the Openwifi project did not succeed yet. Future work could thus include integrating this open-source Viterbi decoder into the Openwifi project, replacing the limited commercial Xilinx Viterbi Decoder core.

4.1.3 Widening attack surface

Currently, the Openwifi project only supports 802.11 a, g and n. 802.11n introduced the HT signal header. This thesis supports fuzzing both the legacy signal header and the HT signal header. 802.11ac, also known as Wi-Fi 5, is the successor of 802.11n. It introduced the creatively

named VHT (Very High Throughput) signal header. Once 802.11ac is implemented in Openwifi, fuzzing this header should also be possible. This also applies for the `HE-SIG-A` (High Efficiency) field in 802.11ax (Wi-Fi 6) and the `EHT-SIG` (Extremely High Throughput) field in 802.11be (Wi-Fi 7) [9].

Openwifi also doesn't have support yet for sending HT frames in greenfield mode. This might be a useful feature to have: devices under test might take a different code path in greenfield mode compared to mixed mode; with this feature we might have better coverage of the implementation. This will also enable the more impactful HT-SIG denial-of-service attack outlined in the previous chapter.

4.1.4 Testing devices in the field

This thesis was mainly focused on modifying the Openwifi project to enable modifying the signal header, and less on thoroughly fuzzing a great number of devices. Combining the code from [17] and this thesis, should allow automated fuzzing, making it feasible to test numerous different Wi-Fi chipsets efficiently.

Bibliography

[1] Wi-Fi Alliance. "Wi-Fi Alliance 2022 Wi-Fi trends." (2022), [Online]. Available: https://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-2022-wi-fi-trends (visited on Aug. 4, 2022).

[2] ——, "Security Wi-Fi Alliance." (2022), [Online]. Available: https://www.wi-fi.org/discover-wi-fi/security (visited on Aug. 4, 2022).

[3] M. Vanhoef, "Fragment and forge: Breaking Wi-Fi through frame aggregation and fragmentation," in *Proceedings of the 30th USENIX Security Symposium*, USENIX Association, Aug. 2021.

[4] S. Heijse, "IEEE 802.11 physical layer fuzzing using openwifi," M.S. thesis, Ghent University, 2021.

[5] G. Rivero. "OSI RM model." (2015), [Online]. Available: https://commons.wikimedia.org/wiki/File:Osi-model-jb.svg (visited on Aug. 8, 2022).

[6] J. Shi. "SIGNAL and HT-SIG, OpenOFDM documentation." (2017), [Online]. Available: https://openofdm.readthedocs.io/en/latest/sig.html (visited on Aug. 8, 2022).

[7] M. S. Gast, *802.11 Wireless Networks: The Definitive Guide, Second Edition*. O'Reilly Media, Inc., 2005, ISBN: 0596100523.

[8] Keysight Technologies, Inc. "802.11n HT OFDM overview." (2022), [Online]. Available: https://rfmw.em.keysight.com/wireless/helpfiles/89600b/webhelp/subsystems/wlan-mimo/Content/mimo_80211n_overview.htm (visited on Aug. 8, 2022).

[9] ——, "OFDM SIG info (802.11n/ac/ax/be)." (2022), [Online]. Available: https://rfmw.em.keysight.com/wireless/helpfiles/89600b/webhelp/subsystems/wlan-mimo/Content/trc_ofdm_ht-sig_info.htm (visited on Aug. 1, 2022).

[10] Intel Corporation. "Different wi-fi protocols and data rates." (2021), [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000005725/wireless/legacy-intel-wireless-products.html (visited on Aug. 11, 2022).

[11] The MathWorks, Inc. "802.11 MAC frame decoding." (2022), [Online]. Available: https://nl.mathworks.com/help/wlan/ug/802-11-mac-frame-decoding.html (visited on Aug. 15, 2022).

[12] X. Jiao, W. Liu, M. Mehari, M. Aslam, and I. Moerman, "Openwifi: A free and open-source IEEE802.11 SDR implementation on SoC," in *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, IEEE, 2020, pp. 1–2.

[13] M. E. Garbelini, C. Wang, and S. Chattopadhyay, "Greyhound: Directed greybox wi-fi fuzzing," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[14] E7mer. "Owfuzz: A WiFi protocol fuzzing tool." (2021), [Online]. Available: `https://github.com/alipay/Owfuzz` (visited on Aug. 18, 2022).

[15] U. Farooq, Z. Marrakchi, and H. Mehrez, "Fpga architectures: An overview," in *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. New York, NY: Springer New York, 2012, pp. 7–48, ISBN: 978-1-4614-3594-5. DOI: `10.1007/978-1-4614-3594-5_2`. [Online]. Available: `https://doi.org/10.1007/978-1-4614-3594-5_2`.

[16] X. Jiao, W. Liu, and M. Mehari. "Open-source IEEE 802.11 WiFi baseband FPGA (chip) design: FPGA, hardware." (2022), [Online]. Available: `https://github.com/open-sdr/openwifi-hw` (visited on Aug. 1, 2022).

[17] T. Schuddinck, "Cybersecurity: Breaking IEEE 802.11 devices at the physical layer," M.S. thesis, Ghent University, 2022.

[18] P. Simpson, *FPGA Design: Best Practices for Team-based Design*. Springer New York, 2010, ISBN: 9781441963390. DOI: `10.1007/978-3-319-17924-7`.

[19] X. Jiao, W. Liu, and M. Mehari. "Open-source IEEE 802.11 WiFi baseband FPGA (chip) design: Driver, software." (2022), [Online]. Available: `https://github.com/open-sdr/openwifi` (visited on Aug. 1, 2022).

[20] Belgian Institute for Postal Services and Telecommunications. "Frequency plan BIPT." (2022), [Online]. Available: `https://www.bipt.be/operators/frequency-plan` (visited on Aug. 4, 2022).

[21] P. Vrancken. "IEEE 802.11 medium access control (MAC)." (2004), [Online]. Available: `http://www.wirelesscommunication.nl/reference/chaptr01/wrlslans/80211_page2.htm` (visited on Aug. 1, 2022).

[22] S. Clark, P. Metzger, Z. Wasserman, K. Xu, and M. Blaze, "Security weaknesses in the APCO Project 25 two-way radio system," *Technical Reports (CIS)*, Nov. 2010.

[23] X. Jiao and M. Mehari. "Open-source IEEE 802.11 WiFi baseband FPGA (chip) design: FPGA, hardware: dot11_tx.v." (2022), [Online]. Available: `https://github.com/open-sdr/openwifi-hw/blob/dc6fc84bd8b1816460b143f70c435cf2b9ca4fda/ip/openofdm_tx/src/dot11_tx.v#L78` (visited on Aug. 1, 2022).

[24] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. "iPerf - the ultimate speed test tool for TCP, UDP and SCTP." (2022), [Online]. Available: `https://iperf.fr/` (visited on Aug. 12, 2022).

[25] S. Pletinck, "Greybox Wi-Fi fuzzing on the openwifi platform," M.S. thesis, Ghent University, 2021.

[26] AMD Xilinx division. "Viterbi decoder offerings and system requirements." (2022), [Online]. Available: `https://www.xilinx.com/products/intellectual-property/viterbi_decoder/viterbi_decoder-software-requirements.html` (visited on Aug. 1, 2022).

[27] Avnet Americas. "EF-DI-VITERBI-SITE." (2022), [Online]. Available: `https://www.avnet.com/shop/us/products/amd-xilinx/ef-di-viterbi-site-3074457345625581960/` (visited on Aug. 1, 2022).

[28] Y. Xi and B. Zhang, "High-throughput open source viterbi decoder for openwifi," M.S. thesis, KU Leuven, 2022.

[29] Keysight Technologies, Inc. "OFDM SIG info (802.11n/ac/ax/be)." (2022), [Online]. Available: `https://rfmw.em.keysight.com/wireless/helpfiles/n7617a/ofdm_signal_structure.htm` (visited on Aug. 12, 2022).

[30] F. Vergès. "MCS index table, modulation and coding scheme index 11n, 11ac, and 11ax." (2022), [Online]. Available: `https://mcsindex.com/` (visited on Aug. 12, 2022).

Appendices

Appendix A - Rate / MCS values

The rate values for the legacy signal header are described in [29] and [7].

Data Rate (Mbit/s)	Modulation	Encoding rate	Rate bits (in transmission order)
6	BPSK	1/2	1101
9	BPSK	3/4	1111
12	QPSK	1/2	0101
18	QPSK	3/4	0111
24	QAM-16	1/2	1001
36	QAM-16	3/4	1011
48	QAM-64	2/3	0001
54	QAM-64	3/4	0011

For the MCS indices, I would like to refer to [30].

Appendix B - Self loopback setup script

```
# ssh into the SDR board, password: openwifi
ssh root@192.168.10.122
cd openwifi
# Bring up the openwifi NIC sdr0
./wgd.sh

# Setup monitor mode in WiFi channel 44. You should find a channel as
# clean as possible in your location.
#Note that some channels don't work, so stick to 44 or 48 for now.
./monitor_ch.sh sdr0 44

# Turn off CCA by setting a very high threshold that make the CSMA engine
# always think the channel is idle (no incoming signal is higher than this threshold)
./sdrctl dev sdr0 set reg xpu 8 1000

# Load side channel kernel module with buffer lenght of 8187
# (replace this with 4095 when using low end FPGA board)
insmod side_ch.ko iq_len_init=8187

# Set 100 to register 11. It means the pre trigger length is 100, so we mainly capture
# IQ samples after trigger condition is met
./side_ch_ctl wh11d100

# Set 16 to register 8 -- set trigger condition to phy_tx_started signal from
# openofdm tx core
./side_ch_ctl wh8d16

# Unmute the baseband self-receiving to receive openwifi own
# TX signal/packet -- important for self loopback!
./sdrctl dev sdr0 set reg xpu 1 1

# Set the loopback mode to over-the-air
./side_ch_ctl wh5h0

# Relay the FPGA IQ capture to the host computer that will show the captured IQ later on)
./side_ch_ctl g0
```

Appendix C - Quality of life development scripts

These scripts enabled two key parts of development: rapid iteration and reproducibility. They ensure that every version of the output of the build process is saved. This makes it easy to switch between different versions and ensures that the research can be reproduced. Note that the build is done on another host because the Vivado toolchain is very particular about its environment (operating system, version, locale, …)

This script copies copies the built driver kernel objects and the FPGA bitstream to a local folder.

```
#!/bin/bash
if [ $# -lt 1 ]; then
    echo "./copy.sh <name_of_experiment>"
    exit 1
fi
set -e -x
DEST="./$1"
mkdir -p "$DEST"
scp
↪  thesispc:~/openwifi/kernel_boot/boards/zcu102_fmcs2/output_boot_bin/BOOT.BIN
↪  "$DEST/BOOT.BIN"
scp `ssh thesispc 'find openwifi/driver -name \*.ko -printf "thesispc:%h/%f "
↪  '` "$DEST/"
```

This script overwrites the driver files and bitstream on the inserted SD card with those of the selected experiment.

```
#!/bin/bash
if [ $# -lt 1 ]; then
    echo "./copy_to_sdcard.sh <name_of_experiment>"
    exit 1
fi
set -e -x
sudo mount /dev/mmcblk0p1 _boot_partition
sudo mount /dev/mmcblk0p2 _root_partition
sudo cp "$1/BOOT.BIN" _boot_partition/
sudo bash -c "cp $1/*.ko _root_partition/root/openwifi/"
sudo umount _boot_partition
sudo umount _root_partition
sudo sync
```

Appendix D - Python script to decode and calculate modified L-SIG

This script was used to calculate the values used with the `inject_80211` program. Note that it does not have any user input handling code; it's intended to be used for the proof-of-concept denial of service attack. The original signal header included at the top was taken from the kernel log.

```
#!/usr/bin/env python3
signal_header = [0x2b, 0x04, 0x02] # size 33, rate 0 (original packet)

def reverse_bits(n):
    return int('{:08b}'.format(n)[::-1], 2)


bits = ''.join('{:08b}'.format(reverse_bits(byte)) for byte in signal_header)
print(bits)
rate = bits[0:4]
reserved = bits[4]
length = bits[5:17]
parity = bits[17]
signal = bits[18:24]
parsed_length = int(length[::-1], 2)
print(f'{rate=} {reserved=} {parity=} {parsed_length=}')

# modify length
parsed_length = 2**12 - 1

new = rate + reserved + '{:012b}'.format(parsed_length)[::-1]
parity = str(sum(int(c) for c in new) % 2)
new_header = new + parity + '00000'

parts = [new_header[i*8:(i+1)*8] for i in range(3)]
print('0x' +  ''.join(["{:02x}".format((int(part[::-1], 2))) for part in
↪  parts]))
```

Appendix E - Python script to calculate modified HT-SIG

This script was used to calculate the values used with the `inject_80211` program. Note that it does not have any user input handling code; it's intended to be used for the proof-of-concept denial of service attack.

```
#!/usr/bin/env python3

# http://jhshi.me/2016/10/18/calculating-crc-for-ht-sig-in-80211n-
↪  preamble/index.html
def calc_crc(bits):
    c = [1] * 8
    for b in bits:
        next_c = [0] * 8
        next_c[0] = b ^ c[7]
        next_c[1] = b ^ c[7] ^ c[0]
        next_c[2] = b ^ c[7] ^ c[1]
        next_c[3] = c[2]
        next_c[4] = c[3]
        next_c[5] = c[4]
        next_c[6] = c[5]
        next_c[7] = c[6]
        c = next_c
    return [1-b for b in c[::-1]]

def chunks(lst, n):
    """Yield successive n-sized chunks from lst."""
    for i in range(0, len(lst), n):
        yield lst[i:i + n]

def format_ht_header(mcs_index, length, short_gi):
    """Formats HT header in ./inject_80211 format"""
    length_field = [int(c) for c in '{:016b}'.format(length)[::-1]]
    mcs_field = [int(c) for c in '{:07b}'.format(mcs_index)[::-1]]

    bits = (
        mcs_field                   + # MCS
        [0]                         + # CBW20/40
        length_field                + # lenght
```

```
        [0,0,1,0,0,0,0]            + # misc parameters ('reserved' should be 1)
        [1 if short_gi else 0]     + # short GI
        [0, 0]                       # misc parameters
    )
    crc = calc_crc(bits)
    ht_sig = bits + crc + [0]*6
    # split sig into list of strings (8 bits per string)
    parts = [''.join(str(c) for c in ht_sig[i*8:(i+1)*8]) for i in range(6)]
    return '0x' +  ''.join(["{:02x}".format((int(part[::-1], 2))) for part in
    ↪  parts])

assert format_ht_header(5, 33, False) == '0x052100040800'
print(format_ht_header(0, int('1'*16, 2), False))
```