Bachelor thesis
Computing Science

Radboud University

Fuzzing Wi-Fi in IoT devices

Author:
Bart Pleiter
S4752740

First supervisor/assessor:
Dr. Ir. Erik Poll
e.poll@cs.ru.nl

Second assessor:
Dr. Peter Schwabe
p.schwabe@cs.ru.nl

January 16, 2020

Abstract

The growing Internet of Things is bringing more connected devices to consumers every day. These devices, like smart thermostats and smart speakers, are designed to make life easier. Usually, these devices are connected to the internet or your smartphone in some way. For this reason, security of these devices is very important.

This thesis investigates the security of the interface protocol implementation of IoT devices using black box fuzzing. With fuzzing, we send many different inputs that are most of the time slightly out of spec, to the device in an automated way to see if the parser of those inputs is robust enough to handle those inputs. We chose to fuzz Wi-Fi, because it is relatively easy to fuzz and does not require special hardware.

Surprisingly, there are only a few articles published regarding Wi-Fi fuzzing and these are over ten years old. These articles describe that a common problem with fuzzing certain frames (messages) is the small time frame for sending back an acknowledgment, for example the frames used for authentication and association. However, the setup described in this thesis offers a solution to this problem, which allows fuzzing of all types of frames, including the frames used to authenticate and associate.

We built our fuzzer in C using Libpcap. While this is more complex than using Scapy with Python, it is about ten times faster than Scapy and allows us to verify if our sent frames are received by the system under test (SUT).

While fuzzing, it is important that the sent Wi-Fi frames are being parsed by the SUT. Therefore, we made the decision to only fuzz the types of frames that the SUT expects. For example, we will only fuzz Probe response frames when the SUT sends Probe requests frames. This approach increased our success rate, since it allowed us to crash a Nintendo DSI XL when it scans for Wi-Fi networks.

Our fuzzer fuzzed the scanning, authentication and association process of several IoT devices and non-IoT devices. Most of the tested devices did not show any problems or anomalies when fuzzed. Only the Nintendo DSI XL that we mentioned before did crash when specific Probe response frames were sent, and a Wi-Fi smart plug showed strange behavior when certain Association response frames were sent.

Contents

Chapter 1

Introduction

The IoT (Internet of Things) market is growing each year. In 2016, already more than 6 billion IoT devices were in use and every year more devices are added to the market [18]. Most of these devices are designed for the consumer market and include sensors like smart thermostats and gadgets like smart speakers with personal assistants to make life a bit more convenient. Most of these devices are connected to the internet in some way, so the user can connect to them using their smartphone. However, this also makes these devices very attractive to hackers. The IoT devices can be used as an entry to your home network, or can even become part of a botnet and be used for DDOS attacks. A real world example of this is the botnet Mirai, a very large botnet consisting of hundreds of thousands devices, which was used in DDOS attacks. There are many different devices ranging from cheap Bluetooth trackers to expensive smart TV's and smart cars. Especially for the cheaper devices, security might have been less of a priority. Therefore, doing research on security of IoT devices is very important.

There are many ways to hack IoT devices. For example, one could search for security holes in the user software or search for weak/default passwords. However, in this thesis we will research the security of the interface protocol implementation, using a technique called fuzz testing. Therefore, our research question is: *How secure is the interface protocol implementation of IoT devices?*. With fuzz testing we can automatically generate and send semi-random data to the target device and monitor if the device crashes. To do this, we first choose an interface and protocol to fuzz. Then we made a selection of IoT devices to test. After that we built a fuzz tester. Finally we tested the selected devices using this fuzz tester and observed the results. These results show which potential vulnerabilities were found for the test devices, and therefore indicate how secure their interface implementation is.

As stated in section 2.3, we chose to fuzz Wi-Fi. While research has already been done on this protocol using fuzzing [16][23][22], this research is very old and has not yet been done on IoT devices specifically.

Chapter 2 gives background information on IoT and fuzzing, and explains the different interfaces used in IoT and why we chose to fuzz Wi-Fi. Chapter 3 gives the relevant information about how Wi-Fi works. Chapter 4 discusses the related work that is already done on this subject. Chapter 5 discusses the devices that we will fuzz. Chapter 6 discusses how our fuzzer works and what decisions we made. Chapter 7 discusses the test results we obtained with our fuzzer. Chapter 8 discusses future work and chapter 9 concludes our research.

1.1 Recurring problems

In this thesis, we found some recurring problems. Some of these problems are very similar and therefore can be confusing. For this reason we give a list of these problems. We will refer back to this list when we encounter one of the problems. Problem 1, 4 and 5 will be referred to by a lot, since these are also encountered in related work. Problem 4 and 5 are timing problems.

Problem 1. When we send a frame to the SUT, how do we know if the sent frame is correctly *received* by the SUT?

Problem 2. When we send a frame to the SUT, which is correctly received by the SUT, how do we know if the frame is *parsed* by the SUT?

Problem 3. At which moment should we send frames to the SUT so that the frame will most likely be *parsed*?

Problem 4. To reply to a Probe request frame from the SUT with a Probe response, we have to respond within a few milliseconds before the SUT moves on to the next channel, otherwise the sent Probe request will be ignored.

Problem 5. When we receive a directed frame from the SUT, we have to send an Acknowledgment frame back within a very small time frame, to notify that the frame was received correctly.

Chapter 2

Background

In this chapter we will explain some background information about the definition of IoT and fuzz testing. Here we will also discuss interfaces used in IoT devices and why we chose to focus on Wi-Fi.

2.1 Definition of IoT

Before doing research on the security of IoT devices, we first need to know what IoT actually means and what qualifies as an IoT device. It is hard to find an exact definition of IoT and IoT devices. Many sources have different definitions ranging from very broad to very specific definitions [6][29][10][27][31][30][12]. Because of these different definitions, there is a gray area of devices that are seen as IoT devices according to one definition, but not according to another definition. For example, devices that use a wired connection, do not have a sensor or use a non-IP based protocol are sometimes considered to not be an IoT device.

In this thesis we give our own definition of IoT and IoT devices. We define *IoT* as a network of IoT devices, and we define an *IoT device* as a non-standard computing device (no smartphone, laptop, pc, server, etc.) that can be connected to using a standard computing device (using an app, program, website, etc.) over a computer network (a PAN over Bluetooth, over the internet using Wi-Fi or using Ethernet, etc.). We exclude devices that are only used for making a computer network available, like network switches or routers.

2.2 Fuzz testing

Fuzz testing or fuzzing is a way of automatically testing software by giving random or semi-random data to the inputs of the software, in order to find implementation bugs [16]. It is usually applied on structured inputs of

software, like a .png file for an image viewer program, but it can also be used on protocols, which are essentially also inputs for software.

There are different kind of fuzzers. There is a distinction between dumb and smart fuzzers, and between generation based and mutational based fuzzers. A dumb fuzzer has no knowledge of the input that it fuzzes and makes the input completely random. This is very easy to implement, but might not be very efficient. A smart fuzzer tries to create a semi-random input that is just good enough for the input parser to accept, but crashes the application using the input. In order to do this, a smart fuzzer must know about the input it is fuzzing. A generation based fuzzer creates inputs from scratch, while a mutation based fuzzer takes an existing input as base and modifies that input. A black box fuzzer generates inputs without knowing about the program, therefore treating the program as a black box. This makes it very well suited for testing firmware or other software where the source code is not available. A white box fuzzer generates inputs using the code that is running on the program. While this might increase the effectiveness of the fuzzer, it also increases the complexity of the fuzzer. Furthermore, it might take longer to generate an input using a white box fuzzer. A grey box fuzzer tries to combine these two types of fuzzers.

While fuzzing, it is important that the software is monitored, so you know when the software fails and which input caused that failure. For this thesis, we use mutation based smart black box fuzzing.

2.3 Choosing an interface

IoT devices can have very different interfaces [28] with each technology having its own strengths and weaknesses in terms of data rate, range, power usage and cost. For example, 3G has a relatively high data rate and range, but the power usage and cost is also high. Some interfaces like DigiMesh and ANT are proprietary and therefore more difficult to fuzz. Commonly mentioned interfaces used in IoT devices that we found were Bluetooth Low Energy (BLE), Wi-Fi (802.11), cellular (GSM), LoRaWAN and ZigBee (802.15.4). Because these are used in many devices, they are the most interesting to fuzz. In order to fuzz GSM, LoRaWAN and ZigBee, special hardware is needed, which would require a lot of work to setup. For BLE, it seems a device like an Ubertooth One [1] would be needed to inject frames. In order to fuzz Wi-Fi, only an Wi-Fi dongle that supports monitor mode and package injection is needed. This makes Wi-Fi a well suited interface to fuzz. Because of this, we chose Wi-Fi as interface to fuzz.

[1] See Ubertooth Github page `https://www.github.com/greatscottgadgets/ubertooth`

2.4 Wi-Fi or 802.11?

Wi-Fi and 802.11 are two related terms. Wi-Fi is based on the IEEE 802.11 standard [2] which specifies the Medium Access Control (MAC) and the Physical Layer (PHY). To make sure a device meets a certain standard for interoperability and security, the device can be certified by the Wi-Fi Alliance which allows it to have a Wi-Fi Certified logo [3]. It can be very confusing to know when to use Wi-Fi and when to use 802.11. Because these terms are very similar, we will use them interchangeably in this thesis.

[2] See `https://www.wi-fi.org/certification`
[3] See `https://www.wi-fi.org/discover-wi-fi`

Chapter 3

The IEEE 802.11 protocol

In this chapter, we describe the important parts of the IEEE 802.11 protocol that relates to our research.

802.11 is a collection of specifications for wireless local area networks. It specifies a medium access control (MAC) and physical layer (PHY) for wireless devices. There are many extensions for 802.11 released, indicated by the letters after 802.11. For example 802.11b allows for faster data rates and 802.11a allows the use of the 5GHz band [24]. Different extensions of 802.11 uses different bands [26], but the mayor extensions use 2.4GHz and since the 802.11a extension also 5GHz. Since 2.4GHz is the most widely used band within Wi-Fi, we will focus only on this band. Within the 2.4GHz band, there are 14 different channels (not all channels are available depending on the country), which are all slightly different frequencies with a 5MHz difference (except between the last two channels) to avoid interference. However, the bandwidth within a channel is 22MHz (or sometimes even 40MHz for 802.11n), and therefore there is some overlap and interference between certain channels. A device can only listen and send to one channel at the same time. While we got most of our information about 802.11 from the 802.11 specification [13], it might be useful to read the book "802.11 Wireless Networks: The Definitive Guide" by M. Gast [19] to read more about the 802.11 protocol, since the book is easier to read than the lengthy specification.

3.1 Networks and Operation modes

There are two types of networks used in Wi-Fi: *infrastructure networks* and *ad hoc networks*.

Within an infrastructure network, there are two types of devices:

- *client devices*. This device connects to an access point so it can access the wired network connected to the access point.
- *access points*. This device can allow connected client devices to access

Figure 3.1: General 802.11 MAC frame (size in bytes)

2	2	6	6	6	2	6	0–2312	4
Frame Control	Duration/ ID	Address 1	Address 2	Address 3	Sequence Control	Address 4	Frame Body	FSC

the wired network connected to the access point.

In ad hoc mode there are only client devices. These devices are then connected to each other directly. This usually means that the device cannot connect to the internet. Therefore, we choose to focus on IoT devices that are part of an infrastructure networks.

When we look at an infrastructure network, a Wi-Fi chip can operate in multiple modes [17][34]:

- Client mode. In this mode the Wi-Fi chip acts as a client device.
- Access point mode. In this mode the Wi-Fi chip acts as a access point.
- Promiscuous mode. A mode used for sniffing network packets. In this mode received packages from the connected network are passed over to the host, even if the destination address does not correspond with the address of the Wi-Fi chip. The host can then see the data going over the connected network at the data link layer.
- Monitor mode. Also a mode used for sniffing network packets. In this mode all valid (at the physical layer) received packets are passed over to the host, without having to connect to a network. Even the frames from adjacent channels are passed over. The host can then see the packets at the physical layer and data link layer. If the Wi-Fi chip and driver both support packet injection, one can use a Wi-Fi chip in monitor mode to send arbitrary frames.

While it is common for an IoT device in an infrastructure network to be in access point mode during the setup of the device, most of the time it will be in client mode so it can access or be accessed over the internet. Therefore, we will focus on client mode, leaving access point mode for further research. This also means that our fuzzer will have to act as an access point.

3.2 Frame types and format

According to the 802.11 specification [13], a general 802.11 MAC frame consists of the fields shown in Figure 3.1. However, not all fields are used in each frame type. The frame type is specified in the Type and Subtype

Figure 3.2: Frame Control header (size in bits)

2	2	4	1	1	1	1	1	1	1	1
Protocol Version	Type	Subtype	To DS	From DS	More Frag-ments	Retry	Power Manage-ment	More Data	WEP	Order

fields within the Frame Control header, as illustrated in Figure 3.2. There are three main frame types, specified by the Type field:

- Control frames. These frames are used help with frame delivery. They always have the To DS, From DS, More Frag, Retry, More Data, WEP and Order fields set to 0 within the Frame Control header. Also, they do not have a Frame Body field. There are different frame formats for the different subtypes.
- Management frames. These frames are used to connect to and disconnect from a device. Aside from not having an Address 4 field, they have the same format as the general 802.11 MAC frame.
- Data frames. These frames are used to send data. They have the same format as the general 802.11 MAC frame.

We will discuss the subtypes of these types in section 3.3.

3.3 connection states

Within 802.11 there are three states, which we will call connection states, between a client device and an access point:

connection state 1. Unauthenticated and unassociated. This is when the client device is not connected to any access point. During this state the client device will most likely scan for access points to connect to.

connection state 2. Authenticated and unassociated. This is when the client device is authenticated with the access point, but not yet associated and therefore cannot transfer data yet by sending data frames.

connection state 3. Authenticated and associated. This is when the client device can send and receive data frames over the network.

During each of these connection states, only certain types of frames are allowed. The frames allowed in connection state 1 are called class 1 frames, the new frames allowed in connection state 2 are called class 2 frames, and the new frames allowed in connection state 3 are called class 3 frames. Furthermore, all frames from the lower connection states are allowed as well. According to the 802.11 specification, the subtypes of frames that are allowed in connection state 1 are the following[1]:

- Control frames
  - Request to send (RTS)
  - Clear to send (CTS)
  - Acknowledgment (ACK)
  - Contention-Free (CF)-End+ACK
  - CF-End
- Management frames
  - Probe request/response
  - Beacon
  - Authentication
  - Deauthentication
  - Announcement traffic indication message (ATIM)
- Data frames
  - Data frames with frame control (FC) bits "To DS" and "From DS" both false

The extra frames allowed in connection state 2 are (Re)Association request/response frames and disassociation frames (both are management frames).

The extra frames allowed in connection state 3 are Data frames with no limitation on the "To DS" and "From DS" bits and PS-Poll control frames.

The current connection state can change when there is a successful authentication or association (see section 3.7) between the client device and access point, or when a deauthentication or disassociation frame is sent to the client device. See figure 3.3

[1]There are more frames added in several extensions of the 802.11 standard. However, we will keep our focus on the main specification and therefore ignore the other frames

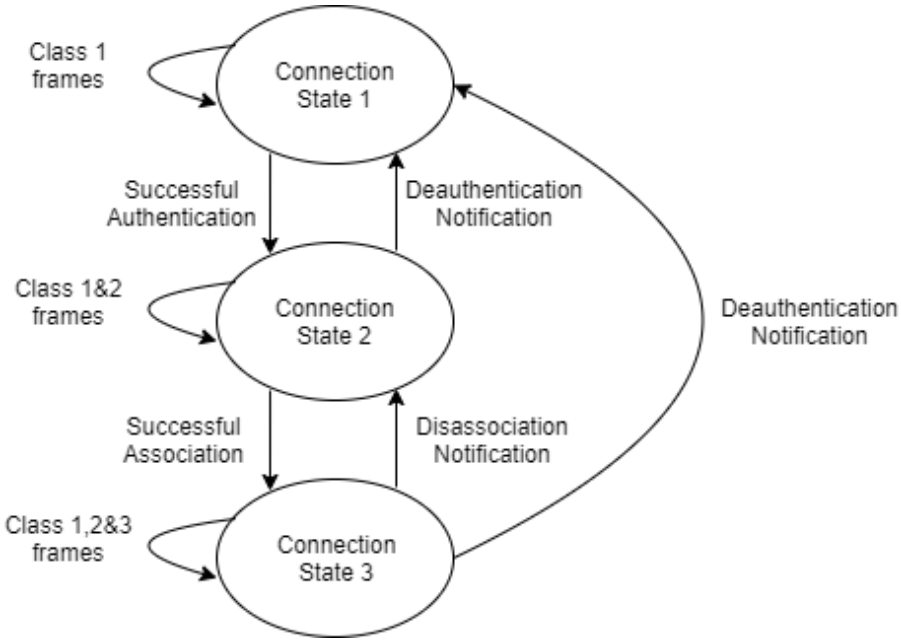


Figure 3.3: Wi-Fi connection states

3.3.1 Fuzzing in which connection state?

It might seem logical to do all the fuzzing in state 3, since in this state all types of frames are allowed. Earlier research has also done this [23]. However, we choose not to do this. For example, if we fuzz Probe response frames, we want to send these Probe response frames as an answer to Probe request frames from the system under test (SUT), since this is the only moment that the SUT expects the Probe response frame (see section 3.6). We expect that this increases the chance that the frame is parsed, and therefore increases the chance that we find some kind of implementation error in the SUT. If we send a Probe response frame while the SUT has not sent a Probe request frame, we expect that the SUT will most likely drop the frame early on without parsing all data of the frame. This corresponds with problem 3 of section 1.1. The same holds for other frames, like Authentication and Association response frames. We can confirm that this design decision to only send frames when the SUT expects them, indeed increases the chance of finding an implementation error, since only this way, we were able to crash a Nintendo DSI XL with Probe response frames (see section 7.1).

3.4 Interesting frames to fuzz

The most interesting frames to fuzz would be the frames allowed in connection state 1, because in case a vulnerability is found, the targeted client

device would be vulnerable from the moment Wi-Fi is turned on without having to connect to a specific access point and therefore minimizing the user interaction. For example, when someone turns on Wi-Fi on their smartphone, it might listen for Beacon frames, a frame allowed in connection state 1 used during the scanning for networks (see section 3.6). If the parser of the Beacon frames has some kind of vulnerability, and attacker can craft a specific Beacon frame and send this to the smartphone to exploit this vulnerability. If the smartphone is already connected to a network, it will probably not listen to Beacon frames anymore, which gives the attacker a small attack window. However, some devices like the Google Chromecast (see section 5.2.1) continue to scan for networks while they are in connection state 3. This gives a huge attack window.

Furthermore, the frames used to authenticate and associate with the access point would also be interesting. For example, in case of an open network, if a device has an exploit in the parser of the authentication or association frames, an attacker could disconnect this device from a network using a deauthentication attack [32] which can cause the targeted device to reconnect to the network. The attacker then might respond to the authentication and association frames with specially crafted frames to exploit the vulnerability. In this case, the attack window would be very large.

When we look at just the types of frames, we can say the following: For the control frames, the RTS, CTS, ACK and PS-Poll frames only contain a Frame Control, Duration (Association ID for PS-Poll frames), Receiver Address (and Transmitter Address for RTS and PS-Poll frames) and FSC field. These fields do not seem interesting to fuzz. The CF-End and CF-End+ACK frames are only used in Point Coordination Function (PCF) mode [25] and PCF is rarely implemented. Because these frames are rare, they might not have been tested and are therefore interesting to fuzz. However, they do not contain a frame body and have no fields with bounds or variable length.

As for the management frames, the Probe response and Beacon frames are the most interesting to fuzz, because these frames are parsed by the device during scanning, and can contain many information fields. See section 3.6.1 for a more detailed explanation and see section 3.6 for a detailed explanation about scanning. The Authentication and (Re)Association response frames could also be interesting, because these frames also contain certain information elements. Furthermore, the Deauthentication and Disassociation frames contain a reason code element that is expected to be two bytes long, which might be interesting to fuzz.

However, the ATIM frame is only used in ad-hoc mode, and the (Re)Association request and Probe request are usually only sent by the client device, and therefore are these frames not that interesting. Though it might be interesting to find out how a client device would react to a Probe request and (Re)Association request frame.

The data frames are not interesting to fuzz, since the frame body will not be parsed by the firmware or driver.

3.5 Acknowledgment frames

Since all frames are sent wireless susceptible for interference and devices can move away from each other, not all sent frames will arrive at destination device. The 802.11 standard specifies that all directed traffic uses ACK frames to verify if a frame is received. This means that when device A sends a directed frame to device B, device B will have to send an ACK frame back to A if it successfully received the frame. If device A does not receive an ACK frame within a small amount of time, it will retransmit the frame until it received an ACK or until it has retransmitted for a certain maximum of times. Undirected frames like Beacon frames do not require an ACK frame and are therefore harder to verify, which corresponds to problem 1 of section 1.1. For this reason, we decide not to fuzz Beacon frames.

3.6 Scanning

In order for a client device to connect to an access point, it has to know which access points it can connect to. This can by done by a process called *scanning*. Scanning is usually done in state 1. It can be done for a short time, or continuously with some interval, depending on the application that controls the Wi-Fi chip. There are two ways a client device can scan for access points:

- Passive scanning. In this mode the client device listens to Beacon frames that are periodically sent by access points. In these Beacon frames, all required info to connect to the access point is provided. The client device self does not send any frames. This way of scanning is the slowest one, because the client device has to wait for Beacon frames to be sent.

- Active scanning. In this mode the client device sends a Probe request frame in each channel that it wants to scan to the broadcast address. The Probe request also contains the SSID of the access point that should respond (this SSID is usually empty, since this requires all access points to respond). After an access point receives a Probe request with a matching SSID, it returns a Probe response to the address of the sender, containing all required information. The client device sending the Probe request only listens for Probe response frames for some time before moving on to the next channel. This timeout is not specified by the specification and therefore can vary between devices. When a Probe response is successfully received, the client

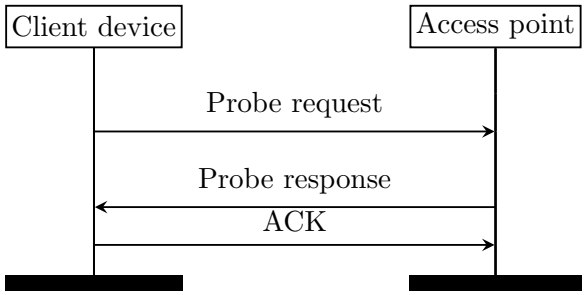


Figure 3.4: Active scanning sequence diagram

device sends back an ACK frame to the access point. See figure 3.4 for the diagram showing the active scanning process.

3.6.1 Beacon and Probe response frames

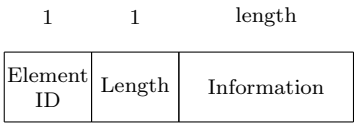
Beacon and Probe response frames have some required and optional fields within the Frame Body. The first three fields are of fixed size, while all other fields are of variable size. The order of these fields is also specified. According to the specification, this is how the frame body should look like:

1. Timestamp (8 bytes)
2. Beacon interval (2 bytes)
3. Capability information (2 bytes)
4. Information elements (variable size)

The format of an information element can be seen in Figure 3.5. All information elements have an unique ID and the variable length is defined by the length field. Since these fields are probably parsed, it is very interesting to fuzz these fields. With newer 802.11 extensions, more information elements were added, giving us more fields to test. Originally, there were only eight information elements. In the 2012 version there were already more than one hundred elements defined. For this reason, we choose to use the 2012 revision of the 802.11 standard [14] to find the information elements that we want to fuzz (see appendix B.1)

The Beacon and Probe response frames are very similar, however there are a few differences between them. The Beacon frame can have a TIM information element, which should only be present within Beacon frames

Figure 3.5: Information Elements (size in bytes)



generated by access points. Furthermore, when the scanning client device successfully receives a Probe response frame as an answer on a Probe request, it sends an ACK frame back. This ACK frame can be used to verify if the Probe response is received, solving problem 1 of section 1.1. An ACK frame is not sent when a scanning client device receives a Beacon frame, which is the reason why we will not fuzz Beacon frames.

3.7 Authentication and Association

When a client device wants to connect to an access point, it has to authenticate and then associate with the access point. To keep things simple, we will assume the access point serves an open network. The process of connecting is shown in figure 3.6. Note that for each sent directed frame, an ACK frame is expected to be received (see section 3.5).

Initially, the client device and access point are in connection state 1, which means they are unauthenticated and unassociated with each other. To authenticate with an access point, the client device has to initiate with an Authentication frame with 1 as sequence number. The access point then responds with the same frame, but with 2 as sequence number. At this point, the client device and access point are in connection state 2. To get in connection state 3, the client device has to send an Association request frame, to which the access point has to reply to with an Association response frame.

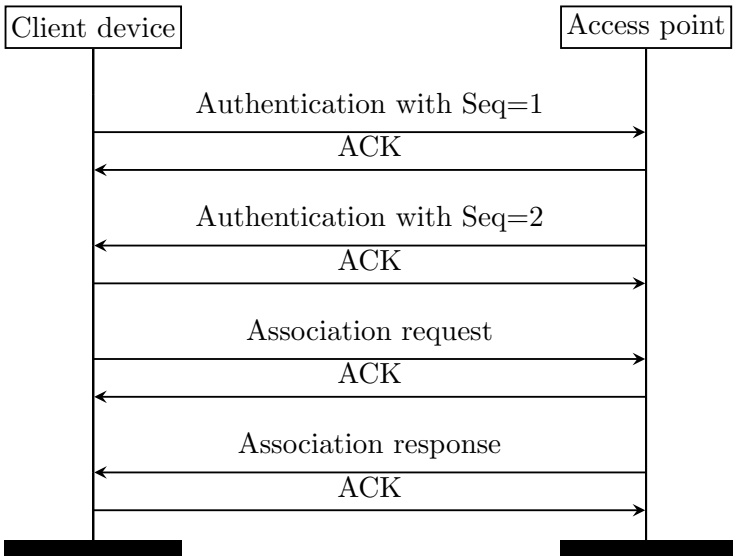


Figure 3.6: Authentication and association sequence diagram with an unprotected network

Chapter 4

Related Work

This chapter discusses the related work on 802.11 fuzzing and show the problems that were encountered in this work and the solutions that were proposed. For these problems we refer back to section 1.1.

When looking at 802.11 fuzzing, some research already has been done. However, this research is already more than ten years old and we could not find any research done on 802.11 fuzzing for IoT devices (as defined in section 2.1).

Butti and Tinnès [16] show how to fuzz 802.11 in state 1 (see chapter 3.3) using Scapy or Lorcon. They mainly focus on the the Information elements of Beacon and Probe response frames. With their fuzzer they fuzzed several 802.11 driver implementations in what appears to be PCs running Windows or Linux. They found four vulnerabilities in certain Wi-Fi drivers using their fuzzer [2][3][5][4]. These vulnerabilities were found by overflowing the rates, SSID, TIM, and RSN information element. Their approach was to send many Beacon and Probe response frames for several seconds, to increase the chance that the frames were received by the SUT. The main problem with this approach is that they still could not verify if the sent frames were being received or not, which corresponds to problem 1 and 4 from section 1.1. Furthermore, they say that in order to fuzz other states, one need to reply to ACK incoming frames within a short time frame that is not easily achievable for user-land applications, which is problem 5 of section 1.1.

These three problems (problem 1, 4 and 5) are also addressed by Keil and Kolbitsch [22]. They propose a solution where they emulate the wireless environment and therefore eliminate the physical wireless device. While this does give more control and the ability to fuzz other states, one could only test the driver of a wireless device. Furthermore, the code of the driver has to be available in order for this to work. Since there usually is no source code available for IoT devices and no way to emulate those devices as far as we know, this solution would not work in our case. They used their method to fuzz the Atheros Windows XP driver and the MADWifi drivers.

As described by Keil and Kolbitsch, a common way to fuzz 802.11 devices is as follows: First the fuzzer listens to probe request frames to make sure the SUT is listening to frames and only then the SUT is flooded with frames. This is used as solution to problem 1 of section 1.1. Usually there also is some kind of second connection to the SUT via Ethernet or USB to monitor the device in some way, for example by parsing logs, detecting OS crashes or running a monitoring program on the SUT. This way of monitoring is usually not possible for IoT devices, as there is probably no easy way to obtain information about the state of the device using another interface.

Another research on 802.11 fuzzing by Mendonça and Neves [23] uses a smart trick to fuzz other states than just state 1 by avoiding problem 5 of section 1.1. They use a real access point to manage the state of the SUT. Using a second Wi-Fi device, they then send custom frames to the SUT with presumably the same MAC address of the real access point. While this lowers the complexity of the fuzzer, it does not allow for proper fuzzing of the authentication and association frames, because these frame will be sent by the real access point. When these frames are sent by the fuzzer, the SUT does not expect these frames and will probably be dropped. This problem corresponds to problem 3 of section 1.1. Furthermore, their setup has no method to indicate whether a frame is actually received by the SUT, which corresponds to problem 1 of section 1.1, and therefore they had to send each frame multiple times. They used their fuzzer to fuzz an HP iPAQ hw6915 PDA running Windows Mobile 5. There is also an article from 2012 by Wang and Zhou which describes the same research with as difference that the SUT is an Android smartphone [33]. However, this research appears to be invalid, since it uses mostly copied or slightly modified text, images and results from the paper by Mendonça and Neves, without any citation.

In contrast to the methods of fuzzing we described, our setup (see chapter 6) does allow fuzzing all states without emulating the wireless device or using a second access point. It solves all three problems that the other research encountered (problem 1, 4 and 5 of section 1.1). This allows us to fuzz the authentication and association process.

Chapter 5

Selecting devices to fuzz

In this chapter we discuss the requirements for a device to be fuzzed by our fuzzer, and discuss potential devices to see if they are a suitable test device.

5.1 Requirements

In order to fuzz a certain IoT device with our fuzzer, it has to meet the following requirements:

1. Since we chose to focus on IoT devices in client mode in an infrastructure network (see section 3.1), the test devices also need to support this mode.

2. It is important that the device scans for networks at some point (see section 3.6), so it can communicate with our fuzzer. This allows us to fuzz Probe response frames.

3. The device has to scan repeatedly for networks, preferably without timeout and without requiring any user interaction between each scan, so we can automate the fuzzing process. However, a long timeout of at least a minute would also be acceptable, as long as the device sends many frames per minute.

4. (optional) In order to fuzz the authentication and association frames, the device has to repeatedly try to authenticate and associate to our fuzzer, preferably without a timeout and without requiring any user interaction between each attempt.

5.2 Potential devices to fuzz

We already had a number of IoT devices available that are potentially suited for fuzzing. Using Wireshark [11] and a Wi-Fi dongle in monitor mode, we

monitored the Wi-Fi traffic from and to these devices to see if they are suited for our fuzzer.

5.2.1 Google Chromecast

The Google Chromecast (H2G2-42, first generation from 2013) is a dongle that can be plugged in a TV using HDMI. It connects to an access point using Wi-Fi. A smartphone or computer that is connected to the same network can then cast videos to this device. According to iFixit [21] the first generation Chromecast uses an AzureWave AW-NH387 chip.

When the device is not setup yet, it starts in access point mode so a smartphone can connect to it and set it up using the Google Home app. During this setup the Chromecast, in client mode while using a different MAC address, starts to scan continuously for access points until a network is selected on the smartphone. This means that requirement 1, 2 and 3 from section 5.1 are met.

When a network is selected on the smartphone, the Chromecast will continuously try to connect to the network. When it cannot obtain an IP address, it will disconnect and retry connecting to the access point, redoing the authentication and association phase. While this only happens every 7 seconds, it will go on forever until the device cannot connect to the access point anymore. This means that also requirement 4 is met.

Also, when setup is done, the device still sends Probe requests every minute with the SSID parameter of the connected network, making it always vurlnerable in case a vulnerability is found in the the Probe response parser.

5.2.2 Google Chromecast Audio

The Google Chromecast Audio (RUX-J42, from 2015) is much like the original Chromecast (see section 5.2.1), except it only outputs audio. It was released two years after the original Chromecast and alongside the second generation Chromecast. According to iFixit [20] it uses a different and newer Wi-Fi chip, the Marvell Avastar 88W8887. This means that the driver implementation would be different from the original Chromecast and therefore we find that this device is also interesting.

As for Wi-Fi, the device works exactly as the original Chromecast. This means that all four requirements from section 5.1 are met as well.

5.2.3 Raspberry Pi 3

The Raspberry pi 3 is a single-board computer with Wi-Fi. While it is basically a very small PC, it has a GPIO header to allow external devices like sensors to be connected. It can be used for many IoT projects and usually runs on Linux. One could even install the Windows 10 IoT OS on it. Therefore, we see it as an IoT device. This also means that this device is

not a black box, giving us more options in terms of monitoring an controlling the Wi-Fi chip.

Since we can run our own applications on the OS, we can build an application that sets the Wi-Fi chip in client mode, scans for access points with a certain interval and connects/disconnects to our fuzzer. Therefore all four requirements from section 5.1 are met, making it a suited device for fuzzing.

5.2.4 Orange Pi Zero

An alternative to the Raspberry Pi 3 (see section 5.2.3) is the Orange Pi Zero, which also has onboard Wi-Fi. This device is a cheaper and smaller alternative to the Raspberry Pi 3. It has the same purpose and also runs Linux, which allows us to create our own application that controls the Wi-Fi chip. Therefore all four requirements from section 5.1 are met, making it a suited device for fuzzing.

5.2.5 Unfit devices

There are a number of devices that were not suited for fuzzing with our fuzzer. These devices will be described here.

The Microsoft wireless display adapter is a dongle that plugs into a TV or monitor using HDMI. The difference between this devices and a Chromecast (see section 5.2.1) is that the Microsoft adapter directly displays the data sent over Wi-Fi on the display. In order to this, it uses Wi-Fi in ad hoc mode. Therefore requirement 1 from section 5.1 is not met and we cannot fuzz it with our fuzzer. The same goes for the Sony DSC-QX10 camera. This device can also only operate in ad hoc mode.

The Panasonic Lumix DMC-LF1 is a compact camera with Wi-Fi functionality. It can operate in either ad hoc mode or in client mode, which meets requirement 1. While the device does scan for access points, it requires pressing a button each time it should scan. Therefore, requirement 3 is not met, making the device unsuited for fuzzing. The same goes for the Brother MFC-J4510DW Wi-Fi enabled printer. It cannot scan continuously without using the touch screen between each scan.

5.2.6 New devices

We also looked at other types of Wi-Fi enabled IoT devices that we did not already have around. The downside of using these devices is that we cannot verify using Wireshark if the device meets all requirements from section 5.1 without buying the device.

We looked at the following three types of devices:

- Wi-Fi Smart Plug. With this device one can turn on or off any device that can be plugged into a wall outlet over the internet

- Wi-Fi Doorbell with camera. With this device one can see who rang the doorbell from over the internet
- Wi-Fi Smart security camera. With this device one can look live from the security camera over the internet without the need for a wired internet connection between the camera and the router

Because of the cheap price of the Wi-Fi Smart Plug, which sometimes can be bought for just under twenty euros, we decided to buy a Wi-Fi Smart Plug. We chose the Caliber HWP101E, because it was the cheapest.

Caliber Wi-Fi smart plug

The Caliber HWP101E is a smart plug that lets you turn on or off any device that is plugged in the smart plug using an app. The device can be setup in two modes. Since we could not figure out how the first mode works, we chose to use the second mode, which is called AP mode. In this mode the smart plug goes into access point mode so a smartphone with the Caliber app can connect to it. The network configuration will be sent from the smartphone to the smart plug using this app. Then the smart plug disables AP mode and tries to connect to the network. The device itself does not scan for networks. However, it does send Probe requests with SSID from the network configuration when it tries to connect to the network. While it might decrease the chance that most information elements from our fuzzer are parsed, it does meet the requirement 1 and 2 from section 5.1. Because the device does not have any timeout, requirements 3 and 4 are also met, making it a good test candidate.

5.2.7 Final selection of IoT test devices

We chose to use the following devices, which we described above, as our main test devices (a picture of the devices can be seen in figure 5.1):

1. Orange Pi Zero
2. Chromecast
3. Chromecast audio
4. Raspberry Pi 3
5. Caliber HWP101E Smart Plug

Figure 5.1: Main test devices

5.2.8 More test devices

Our fuzzer is not limited to test only Wi-Fi enabled IoT devices, since it should be able to test any Wi-Fi enabled devices that meets the requirements from section 5.1. Modifying our fuzzer to test other devices costs practically no effort, since we only have to change one line of code to select the test device. Because of this, we also want to test other devices to see how robust they are. These devices include smartphones and game computers with Wi-Fi. More specifically, we want to run our fuzzer on the following devices:

- Samsung Galaxy S6
- LG Optimus G
- Samsung Galaxy Ace
- Nintendo DS
- Nintendo DSI XL

Chapter 6

Fuzzer

In order to fuzz the devices we chose in chapter 5, we have to use a fuzzer. In this chapter we will discuss the structure of our fuzzer and what options we have with the decisions we made for each part of our fuzzer. In the end we also give our thoughts about writing a fuzzer in C compared to writing it in Python.

6.1 Structure

There are three main parts of our fuzzer:

- A part that handles sending and receiving frames using our Wi-Fi dongle. We will call this part the *Wi-Fi Controller*.
- A part that generates the fuzzed frames which also decides which part of the frame to fuzz. We will call this part the *Fuzz Controller*.
- A part that monitors the system under test (SUT) to verify if a frame

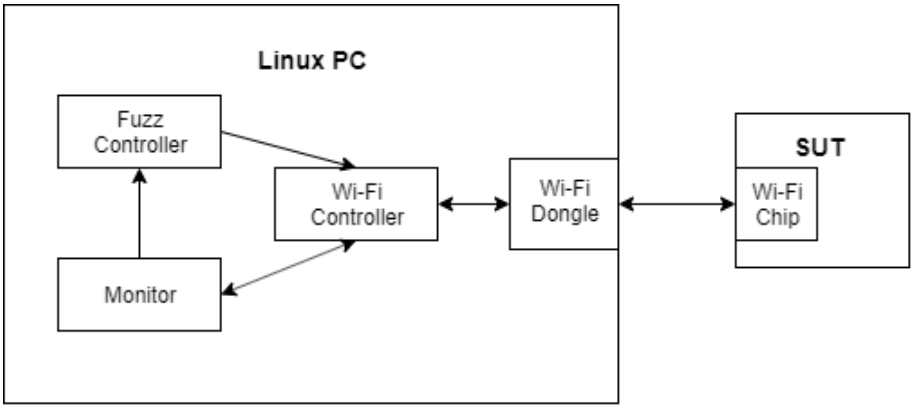


Figure 6.1: Fuzzing setup diagram

is received by the SUT and to verify if the SUT is still up and running. We will call this part the *Monitor*.

Our fuzzing setup using these parts is illustrated in figure 6.1

6.2 Wi-Fi Controller

The Wi-Fi Controller is a very important part of our fuzzer, since it handles sending and receiving frames using our Wi-Fi dongle. Because this is a very fundamental part, we decided to start with creating the Wi-Fi Controller instead of the Fuzz Controller or Monitor.

We found several options to communicate with our Wi-Fi dongle:

- Scapy. Scapy is a library for Python. With it one can easily send and receive 802.11 frames. While it is very easy to use, it might not be the fastest option, because it runs on Python.
- Libpcap. Libpcap is a library for C in Linux. Since it runs on C, it might be a very fast option. However, it is less easy to use than Scapy.
- RAW socket programming. RAW socket programming in C on Linux seems the most direct way to access a network interface in a user-land program. However, it is also a very complex.
- Lorcon. Lorcon is a Metasploit module for 802.11 packet crafting. These libraries are not maintained anymore and the official website is down [7].

Because support for Lorcon is lacking and RAW socket programming seems very complicated, we had to choose between Scapy and libpcap. Scapy looks very easy compared to libpcap, and therefore we decided to build our fuzzer with Scapy. However, as we will describe in section 6.2.1, we were unable to achieve high enough performance using Scapy which resulted in timeouts in some of our test devices. Therefore, decided to use libpcap for our final design.

6.2.1 Scapy

To make sure we could send and receive frames with our setup using Scapy, we did some tests first (see appendix A.2). Our first experiment shows that we are able to send Beacon frames. Our second experiment shows that we can also listen to Probe request and respond to them with Probe responses. When a Probe response is successfully received by a scanning device, an ACK is sent back. However, not all devices that we tested with received the Probe response within their timeout. This problem corresponds to problem 2 of section 1.1. We measured the time it took to respond using Wireshark

and found out that it took 0.03 seconds, which is quite long. Because we want to make sure a Probe response is received by the SUT by listening to ACK responses, we cannot solve this issue by flooding frames. Therefore, we decided to try libpcap to see if that would reduce this delay. The reason for this is that Python (used with Scapy) uses an interpreter to translate python code to machine code at runtime, while C (used with libpcap), a lower level language, is compiled directly to machine code. In theory this means that C could be faster than Python.

6.2.2 libpcap

With libpcap, we did some tests to verify if the delay between receiving and sending a frame would be less than with Scapy (see appendix A.3 experiment 3). We found out that we could respond about ten time faster with libpcap, solving problem 4 of section 1.1. Furthermore, we were also able to consistently verify frames by listening to ACK frames, solving problem 1 of section section 1.1. When we tried to simulate the authentication and association phase (see appendix A.3 experiment 4), we found out that the firmware of our Wi-Fi dongle takes care of sending ACK frames, solving problem 5 of section section 1.1. This allows us to fuzz the authentication an association phase.

6.3 Fuzz Controller

The Fuzz Controller has to provide fuzzed frames and needs to manage what parts of the frame needs to be fuzzed. Using existing fuzzer for this might save some time, since we do not have to build one ourselves. However, an existing fuzzer might be less customizable than a self-built one. Furthermore, understanding the code of a fuzzer created by someone else might take quite some time.

We looked at the following options for a Fuzz Controller:

- Sulley. Sulley is a fuzzing framework written in Python [9]. There exists an 802.11 fuzzer that uses this framework [15]. Because all of the code is written in Python, it would be hard to link the code with our Wi-Fi Controller without taking a significant performance hit.
- BeSTORM. BeSTORM is an enterprise smart fuzzer [1]. It does not seem to be a good option, because it is closed source.
- Peach Fuzzer. Peach Fuzzer is a general fuzzer which has modules including one for Wi-Fi [8]. It states that it only works with certain Ralink chipsets and does not seem very customizable.
- Build our own Fuzz Controller. This would allow us to create a fast Fuzz Controller in the same language as our Wi-Fi Controller.

The existing fuzzers that we found do not really look easy to customize to our needs. Also, the fuzz controller would need to be very fast, since we do not want to increase the delay between receiving and sending a frame by much. This is not easy to test without using the fuzz controller, which could take a lot of time. Furthermore, a Fuzz Controller should not have to be a very complex piece of software, so we decided to build one ourselves.

6.3.1 Authentication and Association Beacon frames

We found that certain devices like the LG Optimus G have trouble with authenticating and associating with our fuzzer, because we do not send out Beacon frames. To counter this, we wrote a simple second program[1] using a second Wi-Fi dongle in monitor mode to send out Beacon frames every 100 milliseconds. We use a second program and dongle for this, since libpcap blocks the program when it is listening for frames, which makes it hard to send out Beacon frames.

6.3.2 Fuzzed frames

We decided to mainly focus on the Probe response frames because of the large number of information elements, and if possible also the authentication and association response frames (see section 3.4). This means that our fuzzer will have to reply to the frames sent by the SUT. Because the SUT usually is a black box, we cannot control its interval between sending. This means our fuzzer can only respond at the same speed as the SUT sends requests. Because of the limited number of frames that we can send per minute, we decided to not use a random number generator. Instead, we manually give a range of number to try. This way we can test the most interesting numbers for each field that we fuzz. While this lowers coverage, it increases the efficiency of the fuzzer.

We created a fuzz controller for the following frames:

- Probe response frames. We created sub-fuzzers for a selection of twenty information elements that we found the most interesting, listed in section B.1. Furthermore, we did some general fuzzing on the Probe response frames.

- Authentication frames and Association response frames. Since we need to send an Authentication frame before sending an Association response frame, we made a single fuzzer for these two frames with two variations. The first variation fuzzes the Authentication frames and sends a default Association response frame. The second variation sends a default Authentication frame, and fuzzes the Association response frame. For the Authentication frame, we only fuzzed the

[1] See BeaconSender on `https://b4rt.nl/git/bart/cfuzz`

Challenge text information element aside from general fuzzing, since this is the only element that is somewhat expected in an Authentication frame. For the Association response frame, we fuzzed aside from general fuzzing six different information elements that can be included in Association response frames. These elements are listed in section B.1.

6.4 Monitor

The monitor verifies if a sent frame is received by the SUT and checks if the SUT has crashed to a certain frame. This part of the fuzzer is specific for our fuzzer only, so building our own monitor is our only option. As seen in earlier research [16][16], one could monitor locally on the SUT and inform the fuzzer using a secondary interface. While in our case this is possible to do on the Raspberry Pi or Orange Pi Zero (see section 5.2.3 and 5.2.4), this approach would not be possible on the other devices that we will test, because those devices are a black box and do not allow for custom applications and have no secondary interface. Therefore, we will monitor the SUT by monitoring its Wi-Fi traffic.

6.4.1 Acknowledgment of receipt

We can verify certain directed frames by listening for ACK responses (See section 3.5). However, an ACK frame does not contain a source address. This means that an ACK frame might have come from another device. To counter this, we make sure to only send frames to and receive frames from the MAC address of the SUT. This way we can listen for ACK frames after sending a frame to verify if the sent frame was arrived correctly, solving problem 1 on section 1.1 (see appendix A.3 experiment 3). We first did this by checking if the first received frame after sending our own frame was an ACK frame. While this worked most of the time, it did not if the system under test was sending two of the same frames directly after each other. Therefore, we changed our Monitor to listen for ACK frames for a certain amount of time instead. This solved the issue. As seen in the experiment described in section A.3.4, not all acknowledged frames are always parsed, which corresponds to problem 2 of section 1.1. Therefore, we decided to have each frame be ACKed multiple times before moving on to the next frame. By doing this, we significantly decrease the chance that an acknowledged frame is not parsed by the SUT, solving the problem.

6.4.2 Crash monitoring

Since we do not always have a way to connect to the SUT over a second interface to monitor the device, we need another way of determining if the

SUT is still running. We decided that the SUT has crashed when it would not reply with an ACK frame for more than ten times in a row. Furthermore we also check if the SUT has stopped sending out frames for a long period of time.

Crashing firmware, driver or application

We expect three different types of crashes that can occur during the fuzz testing:

- Firmware crashes. The firmware is the software that runs on the Wi-Fi chip itself.
- Driver crashes. The driver is the software on the SUT that handles communication with the Wi-Fi chip.
- Application crashes. The application is the software on the SUT that uses the Wi-Fi functionality.

6.5 Writing the fuzzer in C

Since we decided to use libpcap for our Wi-Fi controller, we had to write it in C. To maximize the performance of our fuzzer, and to prevent issues with combining two programs written in different languages, we decided to build our entire fuzzer in C. Compared to Scapy and Python in general, it would have been way easier to write the fuzzer in Python. A program like a fuzzer is a lot easier to make when you can use an object oriented programming language. With Python you also do not have to worry about allocation and deallocation of variables on the stack or the heap, and if a returned variable still exist when the program leaves the function that defined the variable. Also, with Python one does not have to worry about pointers. For these reasons, we would recommend to use Scapy in Python, as long as relatively slow performance is not an issue.

Chapter 7

Results

Here we describe the results of our fuzzing tests on our selection of devices and discuss them. We combined the results and discussion of the Authentication fuzzing and the Association response fuzzing, because we fuzzed these two frames in a very similar manner (as described in section 6.3.2).

7.1 Nintendo DSI XL ERP element crash

During the creation of the Probe response fuzzer, we found that the Nintendo DSI XL, a game computer with Wi-Fi, crashed when it was fuzzed by our fuzzer. This happens with the following setup.

On the DSI, the System Settings application should be opened. In this settings menu, the user has to select the Internet option, then go to Connection Settings and select any of the six connection slots. In this connection setting, the user has to select the AOSS option (since the Scan Networks option uses passive scanning and therefore does not send Probe request frames). When this option is selected, the display should look the same as in figure 7.1 so the DSI will start sending Probe request frames. When these Probe request frames are answered by first a valid frame (or valid enough for the DSI) and then a frame with an ERP information element with a length between 131 and 253 bytes (including 131 and 253), then the system freezes. Most of the time, this only happens after multiple responses are sent. Sending the same frames to the DSI when it is not in the AOSS menu will not have any impact on the system. When the system freezes, all buttons and touch screen are unresponsive. The audio should also freeze. The only way to revert the system to a normal state is to do a hard reset by either reinserting the battery or by holding the power button for multiple seconds. In some rare cases the DSI XL does not freeze, even after a few seconds. We suspect that a number of other access points have to be in range in order for the device to crash, since the device does not crash at certain locations where the DSI does not find several networks. The reason why the

Figure 7.1: Nintendo DSI XL active scanning

system only crashes under these conditions is unknown, since information about the software running on the DSI is not available. We think that the crash is caused in the application and not in the firmware or driver of the Wi-Fi chip, since we have to send two different frames to make the device crash, and it only crashes under certain circumstances. However, the driver probably should have dropped the frame with the invalid RSN information element, since it contains an invalid information element. If that happened, the DSI would not have crashed.

7.1.1 Conclusions

We can draw two conclusions from this result:

1. Because the DSI XL only crashes to Probe response frames as a reply to Probe request frames, we can conclude that our design decision to only send frames when the system under test expects them (see section 3.3.1) increases the chance to find a weakness in the system under test, solving problem 3 of section 1.1.

2. Because it usually takes multiple frames before the device crashes, it might be the case, even though we cannot be sure without an addition experiment, that while all fuzzed frames are acknowledged by the DSI,

not all frames are processed by the application. This problem corresponds to problem 2 of section 1.1. A possible explanation might be that during active scanning, the device sends multiple Probe requests over different channels, as described in section 3.6. When we reply with a Probe response on different channels, it might be the case that the driver parses or returns only one of these Probe responses because it expected just one Probe response, since an access point usually only listens and replies on one channel.

To verify this hypothesis, we did an experiment with client devices scanning for access points, which we describe in full detail in section A.3.4. We saw that while one client device parsed all acknowledged frames, the other client device did not. This means that at least for some devices, not all acknowledged Probe response frames are parsed up until the application that uses those frames. Therefore, it might be useful to modify the fuzzer so it will only send the next frame after multiple confirmations of the same frame, based on the number of confirmations or based on a timer. This is not useful to do on Authentication and Association response frames, since these are definitely parsed, and because the SUT only expects one frame anyway.

7.2 Probe response fuzzing results

For the Probe response frames, we fuzzed a selection of twenty information elements and some more generic parts of the Probe response frame (see appendix B). The results of the fuzzer can be seen in table 7.1.

Device	Problems found	Time (Hours)
Orange Pi Zero	No problems detected	1
Chromecast	No problems detected	10
Chromecast Audio	No problems detected	10
Raspberry Pi 3	No problems detected	1
Caliber Smart Plug	No problems detected	1
Samsung Galaxy S6	No problems detected	2
LG Optimus G	No problems detected	1
Samsung Galaxy Ace	No problems detected	10
Nintendo DS	No problems detected	1
Nintendo DSI XL	Entire system freezes	1

Table 7.1: Probe response fuzzing results

7.2.1 Probe response fuzzing results discussion

As seen in table 7.1, only the Nintendo DSI XL showed problems when we fuzzed it. To be more specific, the entire system freezes when during the AOSS setup as described in section 7.1, a Probe request is answered with an Probe response with an 253 bytes long ERP information element. This also happens when an Extended Supported Rates or Vendor Specific: WPS information element of the same length of 253 bytes is sent. Finally, the device also crashes if a RSN information element is sent with a very long cipher suite length, such that the frame is also 253 bytes long.

7.2.2 Problems with Probe response fuzzing

While fuzzing the Probe response frames, we encountered several problems:

- The small time frame for responding a Probe request frame with a Probe response frame (problem 4 of section 1.1). As stated in section 6.2.2, we solved this by using libpcap in C.
- As stated in section 6.3.2, some devices do not send Probe request frames that often and this sending frequency cannot be altered because they are black box devices. To limit the time needed for fuzzing the devices, we had to also limit the amount of frames that we send. As seen in table 7.1, the two Chromecasts still took about ten hours to fuzz.
- Some devices do not parse all received Probe response frames, even though they have received them correctly (problem 3 of section 1.1). As stated in section 6.4.1, we solved this by sending the same frame multiple times, to reduce the chance that this frame is not parsed.
- Some devices like the Orange Pi Zero would send two Probe requests very fast after each other, which caused our Fuzz monitor to report unacknowledged frames. We solved this by using a timer for listening for acknowledgments, instead of just listening to the next received frame.

7.3 Authentication and Association response fuzzing results

For the Authentication and Association response frames, we fuzzed several parts as described in appendix B. The results of the fuzzer can be seen in table 7.2 and in in table 7.3.

Device	Problems found	Time (Hours)
Orange Pi Zero	Unable to test	na
Chromecast	No problems detected	3
Chromecast Audio	No problems detected	3
Raspberry Pi 3	No problems detected	2
Caliber Smart Plug	No problems detected	<1
Samsung Galaxy S6	No problems detected	4
LG Optimus G	No problems detected	2
Samsung Galaxy Ace	No problems detected	<1
Nintendo DS	Unable to test	na
Nintendo DSI XL	Unable to test	na

Table 7.2: Authentication fuzzing results

Device	Problems found	Time (Hours)
Orange Pi Zero	Unable to test	na
Chromecast	No problems detected	3
Chromecast Audio	No problems detected	3
Raspberry Pi 3	No problems detected	2
Caliber Smart Plug	"Clicks"	<1
Samsung Galaxy S6	No problems detected	4
LG Optimus G	No problems detected	3
Samsung Galaxy Ace	No problems detected	<1
Nintendo DS	Unable to test	na
Nintendo DSI XL	Unable to test	na

Table 7.3: Association response fuzzing results

7.3.1 Authentication and Association response fuzzing results discussion

As seen in tables 7.2 and 7.3, all devices except the Caliber Smart Plug did not show any anomalies during fuzzing. The Caliber Smart Plug, however, started making a clicking noise when we sent all information elements at the same time in an Association response frame. The noise itself is not exceptional, since the device clicks when the device is plugged in or out, when it changes connection mode and when the device receives a command from the smartphone app that instructs it to switch the connected device on or off. However, we did not expect that the device would click when we include

all information elements at the same time in an Association response frame. Especially since the device does now show this behavior single information elements. Therefore, we wonder if this is expected behavior or if this could be used for some kind of attack.

7.3.2 Problems with Authentication and Association response fuzzing

We had several problems while fuzzing the Authentication and Association response frames:

- We were unable to fuzz the Nintendo DS and DSI XL, because we could not find a way to make those devices automatically reconnect to our fuzzer.
- We were unable to test Orange Pi Zero, since for some unknown reason it refuses to initiate a connection to our fuzzer.
- We had some trouble to configure the Raspberry Pi 3 to automatically reconnect to our fuzzer. Eventually we solved this in the following way. We first created a real access point with the same SSID as our fuzzer. Then we connected the Raspberry Pi to the real access point. Then we turned off the real access point and started our fuzzer. Finally we ran the following command to automatically connect and disconnect the Raspberry Pi 3: "while true: do wpa_cli i wlan0 disconnect && sleep 3 && wpa_cli i wlan0 reconnect && sleep 3; done".
- While testing the LG Optimus G, the device would not automatically disconnect from our fuzzer when it cannot obtain an IP address. To solve this, we sent a Deauthentication frame after the Association response frame. However, this caused the device to not see our fuzzer anymore. Therefore we used a second Wi-Fi dongle to send Beacon frames, which partially solved the problem. To make the device automatically reconnect to our fuzzer, we had to delay sending of the Deauthentication frame by one second.
- For fuzzing Authentication and Association response frames, we have to reply to the Authentication and Association request frames from the SUT with an Acknowledgment frame within a very small time frame (problem 5 of section 1.1). As stated in section 6.2.2, we solved this by using libpcap in C and using a Wi-Fi dongle that handles acknowledgments in firmware.

Chapter 8

Future Work

In this chapter we present questions and topics that we encountered during our research that have not been answered yet. These questions and topics might be interesting for future research.

More research on IoT definitions While doing research about IoT devices we could not find any clear definition about what IoT is and what counts as an IoT device, as described in section 2.1. It might be interesting to research what really counts as an IoT device and what IoT is.

Fuzzing other interfaces There are many different interfaces that are used in IoT devices, each with their own protocol. In this thesis we only discussed the Wi-Fi interface of IoT devices. One might also want to research how robust the interface implementation of other interfaces like Bluetooth Low Energy and LoRaWan are.

Fuzzing other devices In this thesis we mainly focused on IoT devices. However, there are many more devices with Wi-Fi, like smartphones. These could also be fuzzed in future research.

More fuzzing of Wi-Fi There are many parts of the Wi-Fi specification that we did not fuzz. We list the parts that we recommend for future research here.

- As discussed in section 3.1, there are two kinds of Wi-Fi network types. For this research, we only focused on infrastructure networks. In a future work, fuzzing could also be applied in ad hoc networks.
- Within an Wi-Fi infrastructure network, there are client devices and access points (see section 3.1). For this research, we only focused on client devices. However, it is also interesting to fuzz access points.

Especially since many IoT devices also have some kind of access point mode for setup.

- For this research, we made a selection of Wi-Fi frames and information elements that we fuzzed (see appendix B). In a future research, it might be interesting to fuzz some of the frames and fields that we did not cover, like frame types that were added in later revisions of the 802.11 standard.

Chapter 9

Conclusions

Wi-Fi is a very complicated protocol. There are five confusing problems, which we described in section 1.1, that need to be solved in order to properly fuzz Wi-Fi on IoT devices. We refer back to these five problems in this chapter.

In chapter 3, we described how Wi-Fi enabled devices connect to each other and what frames they use. From all the different Wi-Fi modes, we decided to focus on client devices in an infrastructure network. We found that Probe response frames, which are used during scanning for access points, are very interesting to fuzz. This is because Probe response frames can contain many variable length Information Elements. For the same reasons, Authentication and Association response frames can also be very interesting, as the also can contain variable length information elements.

As stated in section 3.3.1, to increase the probability that a fuzzed frame is being parsed by the SUT, we made the decision to only send the types of frames that the SUT expects (this corresponds to problem 3; when to send frames such that the frames are most likely parsed). In case of Probe response frames, this means that we will only send a Probe response frames when the SUT sends a Probe request frame. We can conclude that this decision indeed increased our success rates, since it allowed us to crash a Nintendo DSI XL when it scans for Wi-Fi networks.

In chapter 4, we observed that very little research has been conducted on Wi-Fi fuzzing. This research [16][23][22] from many years ago described a number of problems that occur when fuzzing Wi-Fi. These problems are knowing that sent frames are received (problem 1), the small time frame when sending back Acknowledgment frames (problem 4), and the small time frame for responding to a Probe request frame (problem 5). In this earlier research multiple solutions were given to these problems, like flooding the SUT with frames, using a real access point to change Wi-Fi connection states, or to emulate the entire environment. However, none of these solutions seemed optimal for our situation. In chapter 6, we showed that when using Scapy

in Python, we had the same problems as the related work. However, when we used libpcap in C, we were able to solve problem 1 and 4, even though programming became a lot more difficult and time consuming in C than in Python. With libpcap, we could respond about ten times faster than with Scapy, solving problem 4. Also, our Wi-Fi dongle's firmware automatically sends Acknowledgment frames, solving problem 5 and allowing us to fuzz frames used in the authentication and association phase. Finally, were able to receive Acknowledgment frames to our own sent frames, which allows us to monitor if frames are received by the SUT (solving problem 1), and if the SUT has crashed. With all of these parts, we created a fuzzer that automatically communicates with the SUT, monitors it and generates fuzzed frames to send.

While creating and testing our fuzzer, it also appeared that some frames were not parsed by one of our test devices, even though these frames were acknowledged (this corresponds to problem 2; how to know that a received frame is parsed). We confirmed that this is the case for some devices with an experiment described in section A.3.4. To make sure that each generated frame by our fuzzer is at least parsed once by the SUT, we decided to modify our fuzzer so it requires each frame to be acknowledged multiple times before moving on to the next frame.

The finished fuzzer contains three separate C programs with each of those three programs fuzzing a single type of frame. Each of these three programs consist of about 15 to 25 .C files, excluding headers. Coding the fuzzers took a big part of the time we had for writing this thesis. It is built in such a way that changing test devices (MAC address) can be done by changing a single line, and new 'sub-fuzzers' (module that fuzzes a single information element) can be added relatively easily. However, a big part of the code could be optimized by reducing the amount of required duplicate code. This is important if the fuzzer will be extended to fuzz more parts of the Wi-Fi specification.

In chapter 7, the results showed that the Nintendo DSI XL and Caliber Smart Plug showed unexpected behavior when fuzzed with our fuzzer. The DSI XL crashed when it received a Probe response with an overflow in the ERP, extended supported rates, RSN and vendor specific WPS information element. While an possible attack using these Probe response frames appear to be completely useless because of its tiny attack window, it does show how important it is to have a robust input parser. The Caliber Smart Plug started making clicking noises when we included all possible information elements in an Association response frame. We do not know why this happens and if it is expected behavior of the device, or if this might be used for some kind of exploit. From the ten tested devices, the DSI XL and the Caliber Smart Plug were the only devices that showed unexpected behavior. The time needed to fuzz a single device ranged from about an hour, to more than ten hours.

We can conclude from these results, at least for the devices and parts of the Wi-Fi specification that we tested, that IoT devices appear to have a robust and therefore secure Wi-Fi interface implementation. However, there are still many parts of the Wi-Fi specification and more Wi-Fi devices that need to be fuzzed, since we only looked at three different frames used by client devices within an infrastructure network. In chapter 8 we note other parts of the Wi-Fi specification that are interesting to fuzz.

Bibliography

[1] Bestorm. https://www.beyondsecurity.com/dynamic_fuzzing_testing_wifi_protocol.html.

[2] CVE-2006-6059. National Vulnerability Database.

[3] CVE-2006-6125. National Vulnerability Database.

[4] CVE-2006-6332. National Vulnerability Database.

[5] CVE-2007-0933. National Vulnerability Database.

[6] Internet of Things: A survey on the security of IoT frameworks. http://iranarze.ir/wp-content/uploads/2018/02/E5779-IranArze.pdf.

[7] Lorcon website. http://802.11ninja.net/lorcon.

[8] Peach fuzzer wi-fi pit. https://www.peach.tech/wp-content/uploads/WIFI.pdf.

[9] Sulley github page. https://github.com/OpenRCE/sulley.

[10] What is an IoT device? https://www.hcltech.com/technology-qa/what-is-an-iot-device.

[11] Wireshark. https://www.wireshark.org/.

[12] Overview of the Internet of Things. http://handle.itu.int/11.1002/1000/11559, June 2012. ITU-T Recommendation Y.4000/Y.2060.

[13] IEEE 802.11. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999.

[14] IEEE 802.11. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 2012.

[15] Laurent Butti. Wifuzzit github page. https://github.com/0xd012/wifuzzit.

[16] Laurent Butti and Julien Tinnes. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology*, 4(1):25–37, 2008.

[17] Debookee. Promiscuous vs Monitoring mode. https://medium.com/@debookee/promiscuous-vs-monitoring-mode-d603601f5fa.

[18] Gartner. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-3 2 2017.

[19] M. Gast. *802.11 Wireless Networks: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, 2005.

[20] iFixit. Chromecast 2015 teardown. https://www.ifixit.com/Teardown/Chromecast+2015+Teardown/50189.

[21] iFixit. Chromecast teardown. https://www.ifixit.com/Teardown/Chromecast+Teardown/16069.

[22] Sylvester Keil and Clemens Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[23] Manuel Mendonça and Nuno Neves. Fuzzing wi-fi drivers to locate security vulnerabilities. In *2008 Seventh European Dependable Computing Conference*, pages 110–119. IEEE, 2008.

[24] Bradley Mitchell. 802.11 standards explained: 802.11ac, 802.11b/g/n, 802.11a. https://www.lifewire.com/wireless-standards-802-11a-802-11b-g-n-and-802-11ac-816553, 7 2019.

[25] Nayarasi. CWAP – 802.11 Control Frame Types. https://mrncciew.com/2014/10/02/cwap-802-11-control-frame-types/, 10 2014. Accessed: 2019-09-19.

[26] Electronics Notes. Wi-Fi Channels, Frequencies, Bands & Bandwidths. https://www.electronics-notes.com/articles/connectivity/wifi-ieee-802-11/channels-frequencies-bands-bandwidth.php. Accessed: 2019-09-18.

[27] Keyur K Patel, Sunil M Patel, et al. Internet of things-IOT: definition, characteristics, architecture, enabling technologies, application & future challenges. *International journal of engineering science and computing*, 6(5), 2016.

[28] Postscapes. IoT Standards & Protocols Guide: 2019 Comparisons on Network, Wireless Comms, Security, Industrial. https://www.postscapes.com/internet-of-things-protocols. Accessed: 2019-09-01.

[29] Partha Pratim Ray. A survey on Internet of Things architectures. *Journal of King Saud University-Computer and Information Sciences*, 30(3):291–319, 2018.

[30] Margaret Rouse. Definition IoT devices (internet of things devices). https://internetofthingsagenda.techtarget.com/definition/IoT-device, 3 2018. Accessed: 2019-09-03.

[31] Margaret Rouse. Definition Internet of Things (IoT). https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT, 7 2019. Accessed: 2019-09-03.

[32] Brandon Skerritt. Forcing a device to disconnect from WIFI using a deauthentication attack. https://hackernoon.com/forcing-a-device-to-disconnect-from-wifi-using-a-deauthentication-attack-f664b9

[33] Dong Wang and Ming Zhou. A framework to test reliability and security of Wi-Fi device. In *2014 15th International Conference on Electronic Packaging Technology*, pages 953–958. IEEE, 2014.

[34] Wireshark.org. Wi-Fi (WLAN, IEEE 802.11). https://wiki.wireshark.org/Wi-Fi.

Appendix A

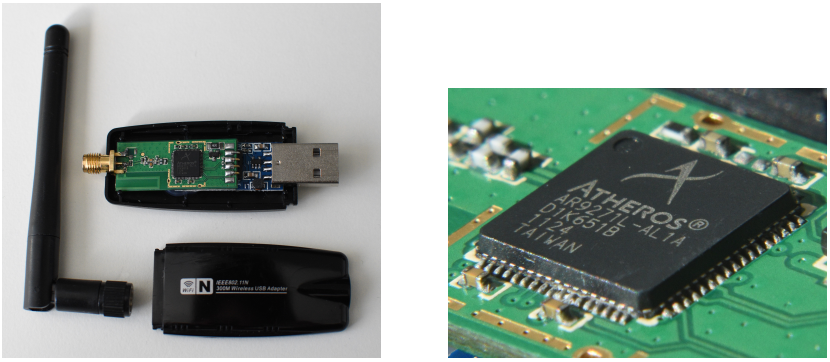
Experiments

In this appendix we give more details about the experiments discussed in chapter 6 and chapter 7. For our experiments, we are using a desktop PC running Ubuntu 18.04 and an Atheros USB Wi-Fi dongle with an Atheros AR9271L chipset (See Figure A.1).

We describe the following experiments:

Experiment 1. Here we test our setup with our Wi-Fi dongle to see if we are able to send our own made frames. For this we used Scapy in Python to send Beacon frames (used in passive scanning). Using a smartphone that uses passive scanning, we were able to verify that our created frames were sent correctly.

Experiment 2. Here we test if we can reply to Probe request frames (used in active scanning) with Probe response frames using Scapy in Python. We measured that the time it took to respond was around 0.03 seconds, which was not fast enough for



(a) Atheros dongle with antenna (b) AR9271L chipset

Figure A.1: Atheros AR9271 USB Wi-Fi dongle

all devices to accept the Probe response frame. This corresponds to problem 4 of section 1.1.

Experiment 3. Here we test if we can reply faster to Probe request frames (used in active scanning) with Probe response frames using libpcap in C instead of Scapy in Python. We measured that our code using libpcap could reply in around 0.003 seconds, which is ten times faster than with Scapy, solving problem 4 of section 1.1 We also found out that we could successfully listen to Acknowledgment frames to our sent frames with libpcap, solving problem 1 of section 1.1.

Experiment 4. Here we test if we can simulate the authentication and association phase of open networks using libpcap in C. The results show that the firmware of our Wi-Fi dongle handles ACKing frames to our MAC address, solving problem 5 of section 1.1, which allowed us to simulate the authentication and association phase correctly.

Experiment 5. Here we test if on some client devices some Probe response frames do not show up at the application that lists the scanning results, even though these frames are acknowledged by the client device, which would mean that not all acknowledged Probe response frames are always parsed. This corresponds to problem 2 of section 1.1.

A.1 Monitor mode and Wireshark

To setup our Wi-Fi dongle in monitor mode (see section 3.1), we use the aircrack-ng suite. For monitoring sent and received 802.11 frames, we use Wireshark. Wireshark can use the Wi-Fi dongle in monitor mode to listen to frames over the air. We can install these packages using the following command:

```
sudo apt install aircrack-ng, wireshark
```

To enable monitor mode on the Atheros AR9271, we used the following commands:

```
sudo airmon-ng start wlx000aeb2d7255
sudo ip link set wlan0mon up
```

In our case wlx000aeb2d7255 was the interface name of our Wi-Fi dongle. This could be a different name depending on the machine and dongle. wlan0mon was the name that airmon-ng gave to the dongle in monitor mode. This name could also change.

A.2 Scapy

Here we discuss our experiments using Scapy.

A.2.1 Setup Scapy

To setup Scapy, we first need to install the following packages:

```
sudo apt install python3, python3-pip, aircrack-ng
pip3 install scapy
```

This should install the latest version of Scapy. Only versions 2.4.0 and above work with python3.

A.2.2 Experiment 1: Test setup by sending Beacon frames

To make sure the setup works, we modified an example from 4armed.com[1] to send Beacon frames. We created a python file called ap.py with the following content:

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

netSSID = 'testSSID'       #Network name here
iface = 'wlan0mon'         #Interface name here

dot11 = Dot11(type=0, subtype=8, addr1='ff:ff:ff:ff:ff:ff',
addr2='22:22:22:22:22:22', addr3='33:33:33:33:33:33')
beacon = Dot11Beacon()
ssid = Dot11Elt(ID='SSID',info=netSSID, len=len(netSSID))

frame = RadioTap()/dot11/beacon/ssid

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

When we ran this code on our PC with the Wi-Fi dongle using Python (with root permissions), this code basically creates a 802.11 beacon frame with an information element containing testSSID as SSID and sends it every 100ms over the wlan0mon interface (our Wi-Fi dongle in monitor mode). This means that our dongle now acts as an access point. While running the script, we enabled Wi-Fi on a Samsung Galaxy S6 (which uses passive scanning in addition to active scanning, see section 3.6) in client mode and saw a new access point called testSSID. This shows that the setup works.

A.2.3 Experiment 2: Respond to Probe request frames without flooding Probe response frames

As described in chapter 3, the scanning device should return an ACK frame after receiving the Probe response frame within a certain time frame. If we

[1]The original example can be found on `https://www.4armed.com/blog/forging-wifi-beacon-frames-using-scapy/`

can successfully listen for these ACK frames, we have a good way to make sure a frame is received by the system under test while fuzzing. Therefore it might be useful to be able to respond to Probe request frames without flooding Probe responses, since we then can listen for ACK frames.

Using Scapy, we can listen for packets using sniff(). We can pass a function which will be called each time a packet is received. In this function we check if the type was a Probe request, then we read the source address, followed by creating a Probe response frame and sending the frame over the interface. The code we created is shown here:

```
from scapy.all import Dot11,Dot11Beacon,Dot11Elt,Dot11ProbeResp
                                ,RadioTap,sendp,RandMAC,sniff

netSSID = "testSSID2"
iface = 'wlan0mon'
myMAC = "00:0a:eb:2d:72:55"

def send_probe_response(source):
    radioTap    = RadioTap()
    dot11       = Dot11(type=0, subtype=5, addr1=source, addr2=
                                myMAC, addr3=myMAC)
    probeResp   = Dot11ProbeResp(cap=0x0114, beacon_interval=
                                0x64, timestamp=12345)
    ssid        = Dot11Elt(ID=0, len=len(netSSID), info=netSSID
                                )
    capability  = Dot11Elt(ID=1, info='\x96\x18\x24\x30\x48\x60
                                \x6c')
    ds          = Dot11Elt(ID=3, len=1, info='\x01')

    frame = radioTap/dot11/probeResp/ssid/capability/ds

    sendp(frame, iface=iface,verbose=False)

def recv_packet(packet):
    if packet.type == 0 and packet.subtype == 4:  # Probe
                                request
        send_probe_response(packet.addr2)

#listen for packets
sniff(iface=iface, prn=recv_packet, store=0)
```

We ran this code on our PC with the Wi-Fi dongle in monitor mode. This way our PC would act as an access point, responding to Probe requests from client devices. We tested this with four smartphones as client devices, a Samsung Galaxy S6, a LG Optimus G, a Samsung Galaxy Ace and a Samsung GT-S3350. All phones except the Galaxy Ace showed the access point called testSSID2 in the Wi-Fi list. Using Wireshark with a second Wi-Fi dongle in monitor mode, we found out that on average it took our PC 0.03 seconds to respond to a Probe request. The smartphone that did not show the SSID of our Probe response did show the SSID after flooding

the device with Probe responses. This shows that the problem is the delay before the packet arrives, which is longer than the time that the device waits for a Probe response, which describes problem 4 of section 1.1.

A.3 Libpcap

Here we discuss our experiments using libpcap.

A.3.1 Setup libpcap

To setup libpcap, we first need to install the following packages:

```
sudo apt-get install libpcap-dev
```

This should install the libpcap library.
To compile a C program using libpcap, we used the following command:

```
gcc -o programName code.c -lpcap
```

Here it is important to include the -lpcap flag.

A.3.2 Experiment 3: Send and verify Probe response

This experiment shows that with libpcap, we were able to receive a Probe request from the system under test (SUT), respond with a Probe response within a very small timeframe using our Wi-Fi dongle, and verify the reception of the Probe response frame by listening to ACK frames from the SUT. This solves both problem 4 and 1 of section 1.1.

For this experiment we used the same four smartphones as in Experiment 2. We ran our code[2] on our PC with the Wi-Fi dongle in monitor mode. This made our PC listen to Probe request frames. When it receive such frame, it creates a Probe response frame and sends it back to the MAC address from the Probe request frame. After the frame is sent, it listens for the next frame and checks the type of this frame. If this frame is an ACK frame to our MAC address, then this means that the frame sent from our Wi-Fi dongle was successfully received by the SUT. If it is not an ACK frame to our MAC address, then the frame might not be received successfully, or there was a frame sent in between the Probe response and the ACK frame.

On all smartphones we were able to see our access point we created with our code. Using Wireshark and a second Wi-Fi dongle in monitor mode, we found out that with libpcap it only takes 0.003 seconds on average to respond to a Probe request frame. This is ten times faster than when using Scapy (See Experiment 2) and is therefore fast enough to reliably use. Furthermore, we were also able to receive most of the ACK frames sent by

[2]For the full code of Experiment 3, see experiment3.c on `https://b4rt.nl/git/bart/cfuzz`

the test devices, allowing us to verify whether the Probe response frame was successfully received by the SUT. Since the delay between a frame and an ACK response is really small, it is important to not recompile and apply the libpcap filter before receiving each frame, since this causes libpcap to miss the ACK frame most of the time.

As a side note, we found during this experiment that the LG Optimus G prints a new line in the access point list when an SSID contains the ASCII character 0x0A. When an SSID consists of 32 of those characters the access point entry will be as long as the entire screen. By creating multiple access points with the same SSID of 32 0x0A characters, one could make it difficult to see any other access point on the list.

A.3.3 Experiment 4: Successful authentication and association

This experiment shows that with libpcap, we were able to simulate the authentication and association phase of connecting a an open network while correctly ACKing frames, solving problem 5 of section 1.1, which means that we can use this to fuzz the authentication and association phase.

For this experiment we only used a LG Optimus G smartphone as system under test (SUT). When ran on our PC with the Wi-Fi dongle in monitor mode, our code[3] uses the same way of responding to Probe request frames as in Experiment 3. In addition it also listens to Authentication frames and Association request frames. For the Authentication frames it responds with an Authentication frame with sequence number 2. For the Association request frames it responds with an Association response frame. We modified the filter for incoming packages to only allow from the MAC address of the SUT (except for ACK frames, since these have no source address). We did not code any functionality to send back the required ACK frame for each incoming frame.

When we ran our code, our created access point showed up on the smartphone. When we tried to connect to the access point, the smartphone gave a message that it could not obtain an IP address. Using Wireshark with a second Wi-Fi dongle in monitor mode, we found out that the smartphone successfully authenticated and associated with our fake access point, and that it could not obtain an IP address because our access point does not handle DHCP requests. More interesting is that the Authentication and Association request frames of the smartphone were ACKed, while we did not code any ACK frames. This is probably because the firmware of our AR9271 Wi-Fi dongle handles the ACKing of frames. We verified this by sending an ACK frame manually using libpcap and it did indeed not show up in Wireshark, which probably means that these frames are handled by

[3]For the full code of Experiment 4, see experiment4.c on `https://b4rt.nl/git/bart/cfuzz`

the firmware. Furthermore, our dongle only ACKs frames when the destination MAC address of the incoming message corresponds with the hardware MAC address of our Wi-Fi dongle. Luckily, while we cannot not send any ACK frame, we are able to receive ACK frames using libpcap, which enables us to verify frames.

A.3.4 Experiment 5: Parsing of Probe response frames

This experiment shows that on some client devices some received Probe response frames do not show up at the application that lists the scanning results, even though these Probe response frames are acknowledged by the client device. This means that not all acknowledged Probe response frames are always parsed by the client device. This corresponds to problem 2 of section 1.1.

Since we want to test if all acknowledged frames are parsed, we decided we can test this by responding to the Probe requests frames of the client device with a valid Probe response frame with the number 0 as SSID. For each acknowledged Probe response frame, we increase this number 0 by one. This way, when the client device parsed all the acknowledged frame, it should display a list of SSIDs with increasing numbers without missing any number in between the highest and lowest number. If the client device does not parse all acknowledged frames, it should have missing numbers in the list of SSIDs. To create the code to test this, we created a copy of our fuzzer and modified it to do exactly as described above[4].

For this experiment, we used an LG Optimus G smartphone and an Orange Pi Zero as test devices. We first ran our code to test the LG Optimus G smartphone. On the smartphone, we opened the settings application and turned on Wi-Fi. After a few seconds of scanning, we saw that we got a list of SSIDs of numbers without any gaps in between, as seen in figure A.2. This means that all acknowledged Probe response frames were parsed.

We then ran our code to test the Orange Pi Zero. Since the device runs on a headless Linux OS, we do not have a settings menu that lists all scanned Wi-Fi access points. However, we were able to use a program called wavemon (installed with the command: sudo apt install wavemon) which scans for access points and prints the results on the terminal. After a few seconds of scanning, we saw that there were a few numbers missing from the list, as seen in figure A.3. From the 28 acknowledged frames, 7 numbers were missing: 4, 6, 10, 12, 14, 20 and 25. The frames that contained these SSIDs were acknowledged by the Orange Pi Zero.

We can conclude from this that not always all frames are parsed, even though they are acknowledged by the system under test. However, this does not mean that these frames are not parsed at all. Maybe they are parsed

[4]For the full code of Experiment 5, see the folder experiment5 on `https://b4rt.nl/git/bart/cfuzz`

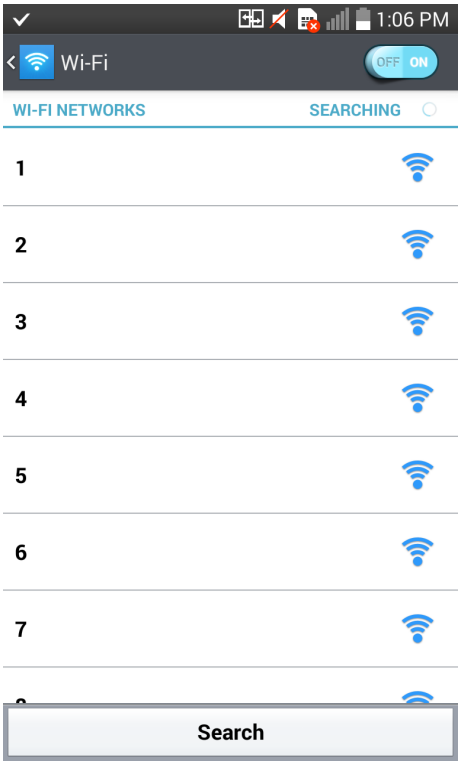


Figure A.2: Wi-Fi connection states

```
-Scan window-

CaveSpot                [redacted]  49%, -76 dBm, ch  11, 2462 MHz ESS
HZN242216699            [redacted]  41%, -81 dBm, ch  11, 2462 MHz 3% chan
HutSpot                 [redacted] 100%, -30 dBm, ch   6, 2437 MHz ESS
Netwerk van YIELD       [redacted]  46%, -78 dBm, ch   6, 2437 MHz ESS, Radio Measure, Spectrum Mgmt
Gastnetwerk van YIELD   [redacted]  46%, -78 dBm, ch   6, 2437 MHz ESS, Radio Measure, Spectrum Mgmt
YUZ082424D              [redacted]  36%, -85 dBm, ch   6, 2437 MHz 23% chan
UPC241210609            [redacted]  34%, -86 dBm, ch   6, 2437 MHz 11% chan
ZiggoA1C92CD            [redacted]  33%, -87 dBm, ch   6, 2437 MHz 6 sta, 9% chan
26                      00:0A:EB:2D:72:55  81%, -53 dBm, ch   1, 2412 MHz ESS
27                      00:0A:EB:2D:72:55  80%, -54 dBm, ch   1, 2412 MHz ESS
24                      00:0A:EB:2D:72:55  80%, -54 dBm, ch   1, 2412 MHz ESS
21                      00:0A:EB:2D:72:55  80%, -54 dBm, ch   1, 2412 MHz ESS
19                      00:0A:EB:2D:72:55  80%, -54 dBm, ch   1, 2412 MHz ESS
16                      00:0A:EB:2D:72:55  80%, -54 dBm, ch   1, 2412 MHz ESS
9                       00:0A:EB:2D:72:55  80%, -54 dBm, ch   1, 2412 MHz ESS
7                       00:0A:EB:2D:72:55  80%, -54 dBm, ch   1, 2412 MHz ESS
23                      00:0A:EB:2D:72:55  79%, -55 dBm, ch   1, 2412 MHz ESS
22                      00:0A:EB:2D:72:55  79%, -55 dBm, ch   1, 2412 MHz ESS
18                      00:0A:EB:2D:72:55  79%, -55 dBm, ch   1, 2412 MHz ESS
17                      00:0A:EB:2D:72:55  79%, -55 dBm, ch   1, 2412 MHz ESS
15                      00:0A:EB:2D:72:55  79%, -55 dBm, ch   1, 2412 MHz ESS
11                      00:0A:EB:2D:72:55  79%, -55 dBm, ch   1, 2412 MHz ESS
0                       00:0A:EB:2D:72:55  79%, -55 dBm, ch   1, 2412 MHz ESS
13                      00:0A:EB:2D:72:55  77%, -56 dBm, ch   1, 2412 MHz ESS
8                       00:0A:EB:2D:72:55  77%, -56 dBm, ch   1, 2412 MHz ESS
5                       00:0A:EB:2D:72:55  77%, -56 dBm, ch   1, 2412 MHz ESS
3                       00:0A:EB:2D:72:55  77%, -56 dBm, ch   1, 2412 MHz ESS
2                       00:0A:EB:2D:72:55  77%, -56 dBm, ch   1, 2412 MHz ESS
1                       00:0A:EB:2D:72:55  77%, -56 dBm, ch   1, 2412 MHz ESS
Ziggo                   [redacted]  67%, -63 dBm, ch   1, 2412 MHz 29% chan, Radio Measure
PrutSpot                [redacted]  67%, -63 dBm, ch   1, 2412 MHz 29% chan, Radio Measure
Ziggo                   [redacted]  46%, -78 dBm, ch   1, 2412 MHz 18% chan, Radio Measure
Liedy                   [redacted]  40%, -82 dBm, ch   1, 2412 MHz 5 sta, 22% chan, Radio Measure
```

Figure A.3: Wi-Fi connection states

by the firmware and driver, but just not by the application. However, we cannot easily verify this. To minimize the probability of having unparsed frames, we can send the same frame multiple times before moving on to the next one.

Appendix B

Fuzzed Fields

In this appendix we describe the specific fields that we fuzzed. We selected most of these fields after reading the 802.11 specification, chapter 4 of the book 802.11 framing [19, Chapter 4], which describes the function of most fields in a more readable manner, and by sniffing access points around us to see what information elements they include in their frames.

B.1 Information elements

There are many information elements (see section 3.6.1) that can be added in the frame body of management frames. These information elements are usually expected to meet some kind of rules, like an SSID length should not be longer than 32 characters. This makes it very interesting to see what the parser of the management frames would do when an SSID is longer than 32 characters. Therefore, we will fuzz these kind of information elements.

B.1.1 Information element selection strategy

In the original 802.11 specification, there were only eight information elements available. Since these are very basic information elements, we expect most of them to be parsed by a client device. Therefore, we decided to fuzz all of these eight elements in our Probe response frames, except the Challenge text element, since these are used in Authentication frames (we will fuzz this element in our Authentication frame fuzzer).

In the many extensions of the 802.11 specification, many new information elements were added. In the 2012 version, a list of more than one hundred information elements are specified. We also wanted to fuzz these information elements. Since we did not have the time to implement fuzzers for all these elements, we chose only the ones that seemed the most likely to be parsed by a client device, because we do not expect that a client device parses so many information elements for each frame it receives. To make a selection of

elements, we sniffed for Probe response frames that were sent by our access point at home to see what information elements it contains. Since these elements are used in practice, we think these have the most chance of being parsed.

B.1.2 List of fuzzed Information elements in Probe response frames

Here we give a list of the twenty information elements that we fuzzed in Probe response frames and what we exactly tried for each information element. While not explicitly stated in the list, we added a case for each information element where we set the data field to 255 times 0xFF bytes. For this section we will abbreviate information element to *IE*. Researching and implementing all these twenty information elements did cost about two weeks of time.

- SSID IE. It contains the SSID or network name of the network. The length is specified to be not longer than 32.
  - Set SSIDs longer than 32 characters
  - Set non standard ASCII values as characters
- Supported rates IE. It should only contain a maximum of 8 data rates.
  - Set more than 8 data rates.
  - Set non-existing data rates.
  - Use duplicated data rates
  - Do not set any rate, or just do not include the information element at all
- FH parameter set IE. It should have a length of 5.
  - Set other lengths than 5.
- DS parameter set IE. It should have a length of 1
  - Set other lengths than 1.
  - Set very high channel numbers
- TIM IE. It should only be present in Beacon frames.
  - Try DTIM Period of 0 and 255
  - Set lengths of 3 and lowere
  - Use this IE in Probe response frames

- CF parameter set IE. It should have a length of 6 and is only used in beacon frames from access points that support PCF
  - Set other lengths than 6.
  - Use this IE in Probe response frames
  - Set all data bits to 1 or 0
- IBSS parameter set IE. It should have a length of 2
  - Set other lengths than 2 with data that is all zeros or all ones.
  - Set all data bits to 1 or 0
- Country IE. Must be an even number of bytes.
  - Try lengths lower than 6.
  - Set Country String to non-ASCII values
  - Set first channel number to 255 or 0
  - Set number of channels to 255 or 0 (in combination with first channel number)
  - Set maximum transmit power to 0 or 255.
  - Try duplicate triplets.
  - Try odd lengths without padding.
- Hopping Pattern Parameters IE. It should have a length of 4
  - Set other lengths than 4.
  - Set all data bits to 1 or 0
- Hopping Pattern Table IE. It should have a minimum length of 6
  - Set lengths lower than 6.
  - Set all data bits to 1 or 0
- Request IE. Is only used in Probe request frames to request certain information elements from the access point.
  - Use in Probe response frame.
  - Set all data bits to 1 or 0.

The following information elements were found by listening to frames sent by access points in range of our Wi-Fi dongle. Because these elements are used by real access points, there is a high probability that these are parsed by scanning devices.

- ERP IE. Has a length of 1

  - Set all data bits to 1 or 0.
  - Try other lengths than 1.
- Extended supported rates
  - Set very large number of rates.
  - Set non-existing data rates.
  - Use duplicated data rates
  - Do not set any rate.
- HT capabilities IE. Has a length of 26
  - Set all data bits to 1 or 0.
  - Try other lengths than 26.
- HT operation IE. Has a length of 22
  - Set all data bits to 1 or 0.
  - Try other lengths than 22.
- AP channel report IE. Has a minimum length of 1
  - Set all data bits to 1 or 0 at large sizes.
  - Set length to 0.
- Extended capabilities IE
  - Set all data bits to 1 or 0 at large sizes.
- BSS load IE. Has a length of 5
  - Set all data bits to 1 or 0.
  - Try other lengths than 5.
- RSN IE. Has a minimum length of 18
  - Set all data bits to 1 or 0, except the version field, which should be 1.
  - Fuzz version field
  - Fuzz Pairwise Cipher suite count field
  - Fuzz AKM Suite count field
  - Fuzz PMKIDcount field
  - Try smaller lengths than 18.
  - Fuzz all List fields with correct lengths
  - Try very large lengths in List fields.

- Vendor specific IE.
  - Fuzz Microsoft WPA type with sub elements.
  - Fuzz Microsoft WPS type
  - Fuzz Microsoft WMM type

B.1.3 List of fuzzed Information elements in Authentication frames

For the Authentication frames, we only fuzzed the Challenge text IE, since this is the only IE that is used in open or WEP networks. For this element, we tried to set invalid lengths. Since the challenge text itself is allowed to be anything, we did not find it interesting enough to fuzz the challenge text.

B.1.4 List of fuzzed Information elements in Association response frames

We fuzzed the following information elements for Association response frames, since these are used in this type of frame.

- Supported rates IE. It should only contain a maximum of 8 data rates.
  - Set more than 8 data rates.
  - Set non-existing data rates.
  - Use duplicated data rates
  - Do not set any rate, or just do not include the information element at all
- Extended supported rates
  - Set very large number of rates.
  - Set non-existing data rates.
  - Use duplicated data rates
  - Do not set any rate.
- HT capabilities IE. Has a length of 26
  - Set all data bits to 1 or 0.
  - Try other lengths than 26.
- HT operation IE. Has a length of 22
  - Set all data bits to 1 or 0.
  - Try other lengths than 22.

- Extended capabilities IE
  - Set all data bits to 1 or 0 at large sizes.
- EDCA parameter set IE. Has a length of 18
  - Try other lengths than 18
  - Set all data bits to 1 or 0

B.2 Generic fuzzing

In this section, we will list the generic fuzzing we did on certain frames.

B.2.1 Generic Probe response fuzzing

Aside from specific information elements, we also did some generic fuzzing on Probe response frames. We tried the following things:

- Large lengths for all 256 possible information elements IDs
- All 256 combinations of flags in the frame control header
- Send frames with a body around 2312 bytes long (it appears that dongle or libpcap will not send frames for body lengths larger than or equal to 1450 bytes, since this appears to be the maximum length of a MAC frame. Therefore, we cannot test sizes longer than 1450 bytes. As alternative, we tested frames of just below and equal to the maximum size our setup could send)
- Duplicates of the following IEs by varying large amounts:
  - SSID
  - RSN
  - Vendor specific
  - HT capabilities
- Send all 256 IEs at the same time
- Do not send any IEs at all

B.2.2 Generic Authentication fuzzing

Aside from the specific information elements, we tried the following things in Authentication frames:

- Use invalid sizes, smaller than 6, since the static elements make up for 6 bytes

- Invalid algorithm number and sequence numbers
- Try all 256 IEs with overflow and underflow
- Send all 256 IEs at same time
- Send duplicate IEs
- Try large frame body sizes

B.2.3 Generic Association response fuzzing

Aside from the specific information elements, we tried the following things in Association response frames:

- Set the Association ID to 0xFFFF
- The Association ID field has two bits always set to 1, so we set it to zero
- Fuzz the static Capability info field
- Try all 256 IEs with overflow and underflow
- Send all 256 IEs at same time
- Do not use IEs at all
- Send duplicate IEs
- Try large frame body sizes