Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF COMPUTATION, INFORMATION, AND TECHNOLOGY

INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Development of a Framework for IEEE 802.11 Beacon Frame Injection and Response Evaluation

Efe Kamasoglu

Technical University of Munich

School of Computation, Information, and Technology

Informatics

Bachelor's Thesis in Informatics

Development of a Framework for IEEE 802.11 Beacon Frame Injection and Response Evaluation

Entwicklung eines Frameworks für die IEEE 802.11 Beacon Frame Injection und Response Evaluation

Author:	Efe Kamasoglu
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Leander Seidlitz, M. Sc.
Date:	August 22, 2024

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, August 22, 2024	
Location, Date	Signature

Abstract

Beacon frames, a fundamental component of the IEEE 802.11 standard, are essential for establishing and maintaining wireless communication. Access points regularly transmit these frames to announce the presence of a network, helping nearby devices detect and connect to it. However, beacon frames openly broadcast network information and are usually transmitted unauthenticated, without integrity protection. Thus, an adversary can easily manipulate the legitimate beacon frames and inject them into the network to launch disruptive attacks, which can lead to Denial-of-Service issues.

This thesis addresses these risks by developing a beacon fuzzing framework, which enables users to inject malformed beacon frames and simultaneously monitor the wireless traffic to evaluate device responses. It also offers different logging capabilities for the captured traffic, allowing for detailed analysis of the communication during the fuzzing process. Furthermore, the framework was utilized to test various attack scenarios, including Channel Switch, Quiet, and Beacon Flooding Attacks, across the 2.4 GHz and 5 GHz frequency bands, demonstrating its capability in identifying vulnerabilities under different conditions. This testing approach not only validated the effectiveness of our framework in real-world scenarios, but also provided insights into how different devices handle these malicious beacon frames.

Our findings show that devices operating in the 2.4 GHz band are especially prone to beacon injection, as they frequently experience varying disconnection times and unsuccessful reconnection attempts. In the 5 GHz band, however, devices demonstrated greater resilience, with minimal disruptions, indicating that the higher frequency band is less susceptible to the injected beacons. These results emphasize the need for enhanced security protocols, such as WPA 3 [1], which secures beacon frames against manipulation and reduces the risks like those demonstrated in this research.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

With the increasing number of mobile devices in the last decade, wireless networks based on the 802.11 standard, commonly known as Wi-Fi, serve as the backbone for communication and connectivity in everyday life. Since it offers a convenient way of connectivity in contrast to the traditional wired networks, the adoption of wireless LAN has become ubiquitous, spanning homes, offices, and public areas. As a result, WLAN presents a pressing matter in terms of security, with continuous discoveries of vulnerabilities. This thesis project addresses one of the notorious security concerns within the WLAN technology by focusing on exploitation of these vulnerabilities.

The thesis research delves into the specific risks posed by the manipulation of 802.11 beacon frames. Beacon frames are management frames transmitted periodically by access points to announce the presence of a WLAN and also to manage the communication between devices within the network. These frames contain essential information such as the network name, supported data rates, and various other network parameters, allowing nearby devices to identify and connect to the network [2]. However, since they are broadcasted openly and most of the time without encryption, an adversary can spoof legitimate beacons and inject malicious payloads that can trigger harmful behavior in connected devices. By targeting the beacon frames, this thesis project seeks to expose weaknesses of the 802.11 standard that can be exploited for disrupting the network [3].

To achieve this research goal, we developed a dedicated framework for the injection of malformed beacon frames. The framework is designed with flexibility in mind, giving users precise control over the injection process, allowing them to fully customize the structure of the injected frames. In addition to injection, the framework also enables users to capture and log the network traffic simultaneously, offering valuable insights into how different devices react to these malicious beacon frames.

The thesis paper is organized into five chapters: "Background" explains the generic 802.11 protocol, focusing on the Physical and Medium Access Control Layers. "802.11 Fuzzing" outlines the design of a typical 802.11 fuzzer and reviews related work. "Approach" introduces the implemented framework, covering its design, capabilities, and testing methods. "Implementation" covers the details of the framework architecture and provides a guide on how the injection and monitoring features of the framework are implemented and performed. Finally, "Evaluation" discusses the performance and effectiveness of the framework by applying it to various attack scenarios conducted in a real-world setting. This chapter also provides an in-depth analysis of the device responses and the overall impact of these attacks.

Chapter 2

Background

The implementation of WLAN is governed by the IEEE 802.11 standard, which is part of the IEEE 802 family [2]. The standard defines a set of specifications as to Physical Layer (PHY) and Medium Access Control Layer (MAC) of the renowned OSI model, as shown in Figure 2.1. The Physical Layer, which is the lowest layer of the OSI stack, is in charge of the transmission and reception of data across the underlying medium, radio waves in the case of 802.11. The MAC Layer, on the other hand, defines how to access the physical medium and handles the addressing of the frames [2] [4]. In contrast to the traditional Ethernet standard (IEEE 802.3), 802.11 introduces a significantly broader and distinct set of specifications for PHY and MAC, owing to the nature of the medium. This undoubtedly impacts the performance, reliability, and security of the network [2].

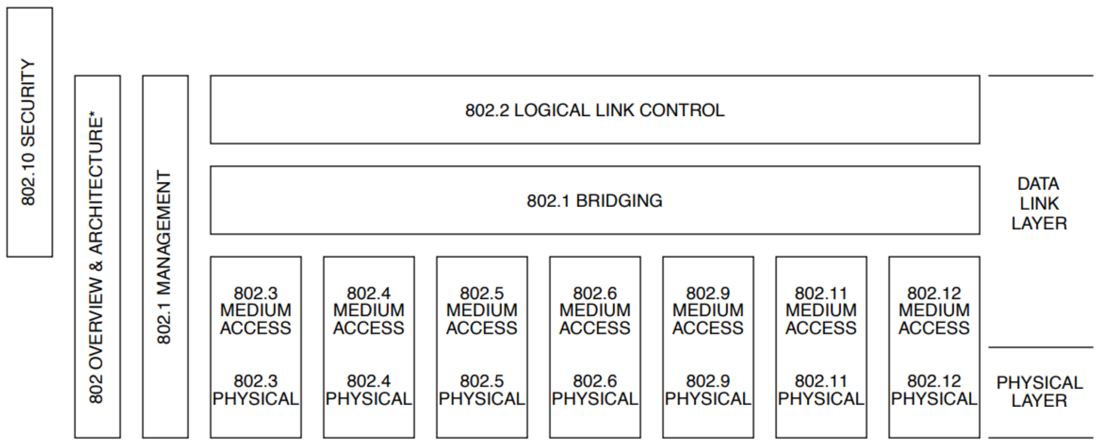


Figure 2.1: Place of 802.11 within the OSI protocol stack [4]

2.1 802.11 PHY

As depicted in Figure 2.2, PHY comprises two sublayers: Physical Layer Convergence Procedure (PLCP) and Physical Medium Dependent (PMD). The PLCP acts as an interface between the PHY and MAC. It is responsible for converting the MAC Protocol Data Unit[1] (MPDU) into a format suitable for sending over the wireless medium [4]. The converted frame is called PLCP Protocol Data Unit (PPDU), which is then transmitted by the PMD sublayer. The PMD sublayer is responsible for determining the characteristics of transmission [2] [4].

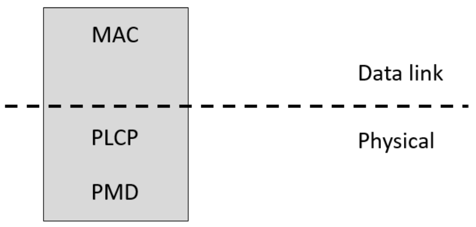


Figure 2.2: PHY component with its sublayers [5]

There are various types of PHY used in the 802.11 standard, where each type is based on a different radio technology. The first release of 802.11 employs three technologies: Frequency Hopping Spread Spectrum (FHSS) PHY, Direct Sequence Spread Spectrum (DSSS) PHY and Infrared (IR) PHY. This thesis does not cover IR PHY, since there has not been any real-world implementation of it [2].

2.1.1 Frequency Hopping Spread Spectrum

The FHSS involves transmitting data by frequently changing the carrier signal frequency. Before frequency hopping starts, the transmitter and receiver must agree on a pattern in order to stay synchronized throughout the entire transmission [2] [6]. One of the benefits of adopting a frequency hopping mechanism is the minimization of sustained (destructive) interference with the primary users, such as military and emergency services. If a specific channel is allocated for a primary user and a secondary user (e.g. a home WLAN) is operating within the same frequency band utilizing FHSS, only a temporary interference occurs instead of a continuous one due to rapid channel switching, as can be seen in the first graph (Time 4-7) of Figure 2.3. Conversely, if two or more secondary users were to use the same band for transmission, they can be configured to

[1] An MPDU is the formal terminology for a MAC frame.

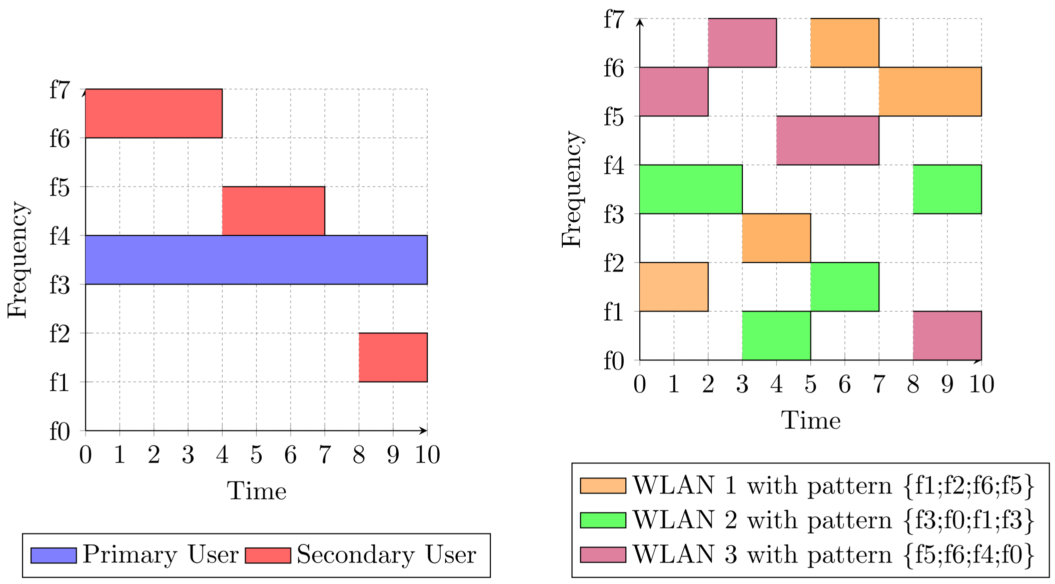


FIGURE 2.3: Occupation of different frequencies in the case of multiple users [2] [6]

hop between channels in a non-overlapping pattern, which would reduce the number of potential network disruptions [2], as depicted in the second graph of Figure 2.3.

2.1.2 Direct Sequence Spread Spectrum

In the case of DSSS, data signal is spread across a broader frequency band. The process involves a transmitter multiplying the original carrier signal with a pseudo-random noise sequence (PN code) and distributing it over the band, as shown in Figure 2.4. A receiver listening to the band recognizes the frequency changes and picks up the widespread signal. The received signal has significantly less power due to spreading and includes the desired data along with a small portion of additional noise and potential interference signals. The receiver then uses the same PN code to reconstruct the original signal by selectively increasing the power of data signals. Any interference signal not correlated with the PN code remains low power and gets filtered out, making the DSSS more resilient to interference [2] [7]. Not only does DSSS mitigate interference, but it also secures communication from eavesdropping. As the PN code is merely known by the transmitter and receiver, a third party monitoring the communication would not be able to reconstruct the original data signal easily. Consequently, DSSS is adopted in many secure communication systems such as military radars [7].

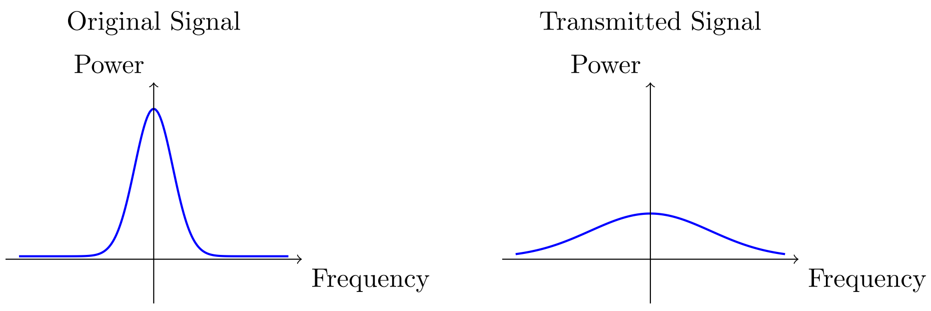


Figure 2.4: Spreading of the original data signal [2]

2.1.3 Orthogonal Frequency Division Multiplexing

Although FHSS and DSSS were the forerunners, they have been overshadowed by another PHY technology that was released as part of the 802.11a, an amendment to the 802.11 standard. In the early days of WLAN, the 2.4 GHz band was widely adopted due to its lower cost and better range. Other devices outside of WLAN, such as Bluetooth devices and microwave ovens, also share the same band. Therefore, it was subjected to significant interference despite the mitigating PHY technologies, FHSS and DSSS. As a result, Orthogonal Frequency Division Multiplexing (OFDM) PHY was introduced, which operated in the 5 GHz band, to increase data rates owing to the higher number of non-overlapping frequency channels [8]. The key idea behind OFDM is to split a wide channel into multiple subchannels, each associated with a carrier signal (subcarrier) that transmits data simultaneously. These subcarriers have the property of orthogonality, which means they are closely spaced in a way that they do not interfere with each other. This way, OFDM can utilize the available band more efficiently compared to FHSS and DHSS [2].

Even though the invention of OFDM traces back to the 1960s, it serves as the foundational framework for today's state-of-the-art Wi-Fi technologies, such as 802.11g/n/ac/ax. Those standards have extended OFDM's capabilities to support even higher data rates, increase network capacity, and improve power efficiency for 2.4 GHz, 5 GHz, and 6 GHz bands, which was officially opened for unlicensed use towards the end of 2020 within Wi-Fi 6E [8] [9].

2.2 802.11 MAC

Expanding on the foundation of the PHY layer, our next topic is the 802.11 MAC, which orchestrates access to the wireless medium in cooperation with PHY, manages frame delivery, and facilitates security to protect WLAN communication [2]. The 802.11 MAC behaves similarly to the Ethernet MAC in terms of fundamental functionalities, including addressing, error detection and access control. For instance, higher layer protocols, such as ARP or IPv4, also apply to 802.11 MAC. This ensures interoperability in the case of a heterogeneous environment where there are both wired and wireless networks and devices. Nevertheless, the 802.11 MAC is inherently broader than the Ethernet MAC, as 802.11 requires complex communication mechanism due to the physical nature of the wireless medium.

2.2.1 Medium Access Control

The 802.11 MAC includes two coordination functions that define different methods of access: Distributed Coordination Function (DCF) and Point Coordination Function (PCF). The DCF, being the fundamental access method, is based on a distributed algorithm, hence it should be implemented by all the devices within the network. Whereas the PCF uses a centralized mechanism that is managed by point coordinators (PCs) residing in access points (APs). The PCs control when each device can transmit by granting them transmission rights, providing a contention-free medium access [4]. However, the way of centrally scheduling the devices comes at the expense of network scalability. As the number of devices increases, an AP must keep track of all the devices in the network, which leads to a heavy polling load on the AP. Besides, its usage is restricted to infrastructure networks, where devices in the network communicate with each other through an AP. On account of these shortcomings, PCF has limited adoption in modern Wi-Fi networks as opposed to DCF [2].

CSMA/CA vs. CSMA/CD

Unlike Ethernet, the DCF utilizes Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA), as collision detection is notably challenging in a wireless environment. One of the critical reasons for that is the Hidden Node Problem. It arises when two devices A and B not in each other's signal range try to send data simultaneously to a third device C that is within range of both, as shown in Figure 2.5. This may lead to a collision at device C, as devices A and B are "hidden", meaning that they cannot detect each other's transmissions.

Moreover, wireless devices typically operate in half-duplex mode, as they cannot transmit and listen to the medium at the same time. Full-duplex support is not preferred in wireless devices since it requires expensive hardware and complicates communication. Plus, half-duplex is more power efficient than full-duplex, which makes it ideal for mobile devices. Apart from that, several other physical phenomena limit the ability to detect collisions as well. One of them is interference, which can cause signal degradation, making it harder to detect the presence of a collision [2]. There is also multipath propagation, where signals take different paths to reach the receiver due to reflection and diffraction caused by obstacles along the path. As a result, signals arrive at different times or might be echoed, negatively affecting collision detection in terms of accuracy [10].

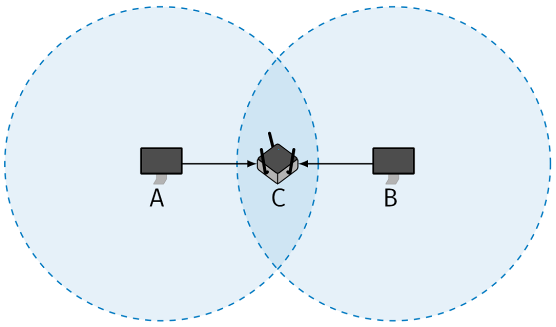


Figure 2.5: Hidden Node Problem [11]

Carrier Sensing and Collision Avoidance

The CSMA/CA procedure begins with carrier sensing, in which a device first listens to the channel to determine whether it is free before transmitting any data. If the channel is free, the device waits for a fixed period called the DCF Inter-frame Space (DIFS), which is required between consecutive transmissions to avoid collisions [4]. It also helps to prioritize different types of frames such that high-priority frames are sent using the minimum IFS, also known as Short Inter-frame Space (SIFS). Those high-priority frames include control frames such as Request to Send (RTS) and Clear to Send (CTS), which are used to notify other devices in the network in the case of a transmission [2].

On the other hand, if the channel is occupied, the device waits for a backoff time randomly selected from a contention window, until the channel becomes free again. The contention window initially has a minimum size, containing a range of backoff periods, which are represented in time slots. Upon successful transmission, the receiver sends back an acknowledgement frame. If the transmitter does not receive an acknowledgement within a specified timeout period, it assumes a collision has occurred. With each "inferred" collision, the contention window is doubled in size until it reaches a predefined maximum. This reduces the probability that multiple devices select the same backoff time and collide again. After a successful transmission, the contention window shrinks back to its minimum size [4]. The collision avoidance process is illustrated in Figure 2.6.

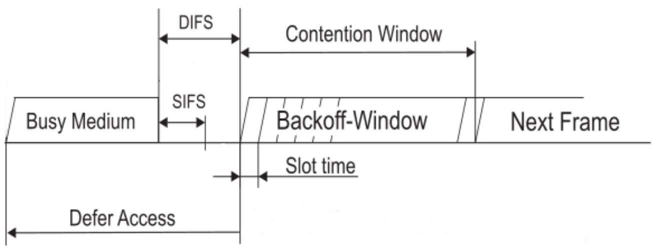


Figure 2.6: Collision Avoidance in CSMA/CA [4]

2.2.2 Network Operations

There are many network operations, also referred to as services, provided by the 802.11 standard. Network operations are procedures for managing the network in which various frames are exchanged between multiple devices. By processing those frames, devices can gather essential information about the network and become aware of the recent network changes. This includes tuning parameters such as channel selection, power levels, and security settings to ensure seamless communication and efficiency within the network [2].

Scanning and Network Announcement

Scanning is one of the fundamental features of a wireless device, enabling user to discover available WLANs. Client devices can perform either passive or active scanning depending on the current situation. At the same time, APs broadcast beacon frames to announce the presence of the network, which will be extensively discussed in the following chapter. In passive scanning, devices listen for incoming beacons and buffer them to later on extract network information. While, in active scanning, devices send out probe request frames to search for a specific WLAN or to identify all nearby WLANs [2].

Authentication & Deauthentication

After selecting a WLAN, authentication phase starts, through which the client device exchange authentication frames with the AP to verify its identity. Authentication prevents unauthorized access and offers encryption, protecting user data from eavesdropping. There are several security protocols, such as WEP and WPA, that implement different authentication methods. Most of them include a Pre-shared Key (PSK) as part of the process, which is commonly known as the WLAN password. Furthermore, both the AP and client can send deauthentication frames to terminate the authentication, which may lead to disconnection from the network [2] [4].

Association & Disassociation & Reassociation

Association refers to the process of joining the network by associating with an AP. Upon successful authentication, client sends an association request frame to the AP. AP then replies with an association response frame, confirming the connection and allowing network access. On the other hand, a client can also send a disassociation frame, informing an AP to leave the network. An AP may also want to dissociate a client, if, for instance, the previous authentication of the client is not valid anymore [2] [4].

In a scenario, where multiple APs are present in a single network, a client can send a reassociation request frame to the currently associated AP to switch to a different AP. This is due to the roaming behavior of mobile clients, in which a client moves from the coverage area of one AP to another within the same network to improve signal quality [2].

Dynamic Frequency Selection (DFS)

Dynamic Frequency Selection is a mechanism for 5 GHz networks, which was released with the amendment 802.11h in 2003. It was implemented with the purpose of avoiding interference with radar systems, hence a primary user, operating within the same band. Before an AP begins with transmission, it first listens to the channel to detect radar signals. If it detects any, it must cease transmission and notify all clients to switch to a different channel or to be quiet for a specified duration. Although DFS provides efficient utilization of the band in the case of multiple users, it can be exploited for disrupting the network. [2] [12].

2.2.3 Addressing and Frame Format

The 802.11 MAC Header is naturally more extensive than the one used for Ethernet. In Ethernet, data transmission is straightforward, with frames typically depending on only source and destination addresses. However, in a WLAN, frames traverse various routes where they may be received by multiple intermediary devices, such as Wi-Fi extenders, and relayed through APs before reaching the final destination. Many wireless devices, e.g. mobile phones or IoT devices, rely significantly on battery power, thus a WLAN must include power-saving features as well [2]. To effectively address those complexities, the 802.11 MAC Header includes additional fields, as depicted in Figure 2.7.

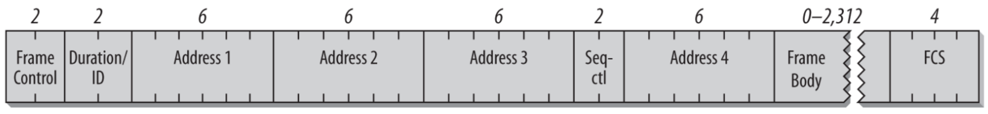


Figure 2.7: 802.11 MAC Header [2]

It consists of multiple fields serving different purposes [2] [4]:

- Frame Control, which contains control information, such as the currently used protocol version, frame type and fragmentation. There are three 802.11 frame types with each type comprising multiple subtypes:
  - **Management Frames (Type 0):** Used to establish and maintain connections between devices and APs, e.g. Beacon (Subtype 8), Probe Request (Subtype 4)
  - **Control Frames (Type 1):** Required for managing access to the medium and ensuring smooth data transmission, e.g. RTS (Subtype 11), ACK (Subtype 13)
  - **Data Frames (Type 2):** Responsible for higher layer protocol encapsulation and delivery of user data, e.g. Data (Subtype 0)
- Duration/ID, which indicates the number of microseconds to complete the current frame transmission or an association identifier that identifies a device associated with an AP in the case of certain control frames.
- Four MAC address fields, but depending on the frame type, the number of address fields may vary:
  - Destination Address: Final destination of the frame
  - Source Address: Original sender of the frame
  - Receiver Address: Intermediate hop to relay the frame; if the destination is beyond the local network, an AP would be the receiver forwarding the frame to its final destination
  - Transmitter Address: Address of the interface that transmits the frame, only used in wireless bridging[1]

[1] A method for connecting two or more wireless network segments by creating a bridge between APs, joining them as if they form a single LAN.

  - Basic Service Set Identifier (BSSID): Refers to the address of the AP within an infrastructure network

- Sequence Control, which includes sequence and fragment numbers to manage the order and integrity of frames
- Frame Body, which represents the higher layer payload and varies in content depending on the frame type
- Frame Check Sequence (FCS) for error detection equivalent to Ethernet FCS

2.3 Beacon Frame

As briefly mentioned in the previous section, beacons are broadcasted periodically by APs, allowing network devices to discover nearby WLANs. However, they also fulfill several other functions, such as synchronization, power management, and security, enabling devices to optimize and maintain their connections. As with other 802.11 frames, each beacon consists of two main components: a MAC header and a Frame Body. Beacon MAC header relies on three address fields: a destination address, a source address, and a BSSID. Since beacons are intended for all surrounding devices, the destination address is always the broadcast MAC. The source address is typically identical to the BSSID, which is the MAC address of the AP. Additionally, beacon frame body comprises two further components: a mandatory Beacon Body and a number of optional Information Elements (IEs) [2] [4], as shown in Figure 2.8.

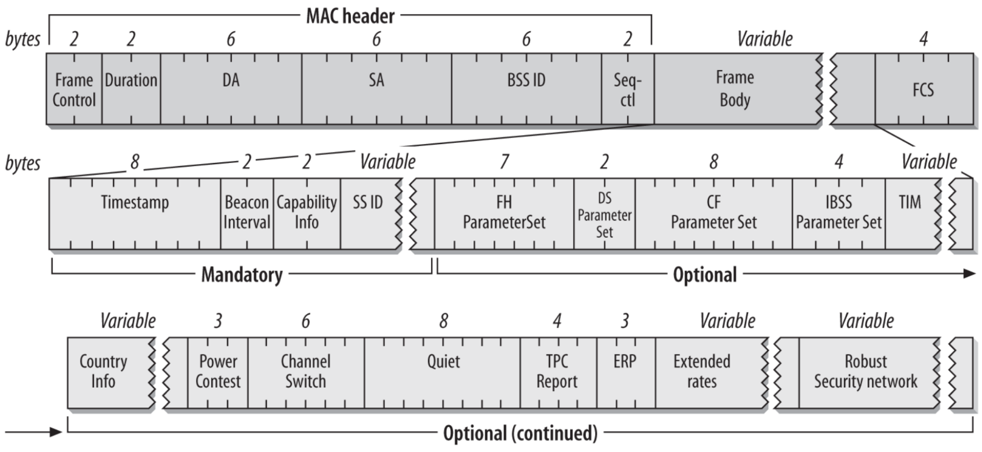


Figure 2.8: Beacon Frame [2]

For network announcement, beacons contain Service Set Identifier (SSID), a string ranging from 0 to 32 bytes that uniquely identifies a WLAN, generally known as the network name. The null-byte SSID, also called the broadcast or wildcard SSID, serves a particular purpose in active scanning, where devices include it in their probe requests to discover all networks rather than probing for a specific one [4].

Another crucial aspect of the network announcement operation is presenting the network capabilities using the 16-bit Capability Information field, with each bit representing a single capability. After receiving a beacon frame, a device can compare its capabilities to those of network's. If the device does not support all the capabilities, it is not allowed access to the network. In addition, beacons provide a number of data rates that the network supports. Although some data rates are optional, a device must verify that it supports all the mandatory data rates, as it does for capabilities, to ensure compatibility [2].

Furthermore, beacons include a Timestamp field to perform Timing Synchronization Function (TSF), which helps synchronize the clocks of all devices in the network. Timestamp represents the TSF timer, which an AP maintains throughout the network's lifetime. It indicates the number of microseconds the AP has been active [2]. When a device receives a beacon, it adjusts its own TSF timer to match the timestamp provided by the associated AP. With a synchronized clock, the device may operate in power-saving mode, knowing precisely when to wake up and check for active transmissions, which ensures efficient battery consumption [13]. Further synchronization is achieved through the Beacon Interval field. This field indicates the interval between consecutive beacon transmissions and is measured in Time Units (TUs), with 1 TU being 1.024 microseconds [2] [4].

Following the mandatory Beacon Body, there are optional Information Elements providing additional information about the network such as extended capabilities and vendor-specific information. As devices buffer the beacons, they extract this information, which helps them to configure themselves accordingly to take advantage of these extended features [2]. However, Information Elements are highly prone to exploitation by unauthorized parties and can pose security risks [3]. This vulnerability serves as the central research focus of this thesis project.

Chapter 3

802.11 Fuzzing

Fuzzing is a technique that discovers security vulnerabilities by systematically testing a target system with manipulated inputs and observing its behavior [14] [15]. In the context of 802.11 fuzzing, the target system typically refers to a WLAN with multiple devices connected to it, into which different types and subtypes of malformed frames are injected.

The generic fuzzing process starts with the generation of input data using methodologies such as mutation-based or generation-based fuzzing. Generation-based fuzzing involves creating data from scratch based on the understanding of the protocol or data format, whereas, in mutation-based fuzzing, the initial data is slightly modified in each fuzzing iteration. The data is then fed into the system and continuously monitored to capture its responses, including crashes, errors, or other deviations from expected behavior. After gathering the responses, an output data set is created. The fuzzing process then moves to the analysis stage, where the output data is examined to identify potential bugs of the target system. If the analysis indicates that the fuzzing is complete, a detailed bug report is generated, in which all the findings are documented. However, if the analysis suggests that further testing is needed, the process loops back to the data generation stage. A refined test data is then created based on the insights gained from the previous fuzzing iterations [14] [16].

Fuzzers, pieces of software designed to perform fuzzing, are categorized into three types. White box fuzzers have complete knowledge of target system's internal workings, allowing them to generate targeted test cases. Grey box fuzzers operate with partial knowledge, typically combining some internal insights with observed behaviors to guide the fuzzing process. Black box fuzzers, on the other hand, have zero knowledge of the target system, thus they generate test cases purely based on the protocol specifications and observed behaviors [14].

3.1 Design of a 802.11 Fuzzer

The 802.11 fuzzers produce frames as input data according to the 802.11 specifications, yet they lack the knowledge of the internal mechanisms by which devices in the target network process these frames. Network devices often incorporate proprietary wireless drivers and firmware developed by different vendors, leading to variations in how the underlying 802.11 network stack is implemented. As reverse engineering of such drivers is arbitrarily complex, given the multitude of device brands, developing a fuzzing tool stands out as an inexpensive and efficient alternative for identifying vulnerabilities. However, fuzzing is generally effective at detecting rather obvious vulnerabilities, while subtle ones require an in-depth understanding of the driver code [17].

An 802.11 fuzzer typically consists of two primary modules: an injection and a monitor module. Depending on the design of the fuzzer, additional modules and features can be implemented to enhance its functionality. These features may include the categorization of discovered bugs to understand the severity and impact of fuzzing for future analysis [15] [17] [18].

Injection Module

Injection module is responsible for crafting and injecting malformed 802.11 frames into the target WLAN. It can create frames based on predefined templates or dynamically on the fly by randomly modifying the frame contents or following an algorithmic approach, in which previous responses are analyzed to mutate the frame. By introducing anomalies in the frame structure, timing, or content; this module can be utilized to carry out several attacks to test the network through its handling of unexpected frames [15] [17] [18].

Monitor Module

Monitor module captures and analyzes the responses from the target network to the injected frames. This module listens to the wireless traffic, recording any abnormal device behavior or signs of vulnerabilities, such as unexpected deauthentication attempts from the network, which may imply Denial-of-service conditions. It may also include automated reporting capabilities for generating comprehensive logs and summaries of the fuzzing session. These reports are crucial for further investigation and mitigation efforts, as they offer insights into the weaknesses of the target devices [15] [18].

3.2 Related Work

Several notable fuzzing tools and methods have been developed in the field of Wi-Fi security, each contributing significant findings. These tools vary widely in design, functionality, and scope, each adopting unique approaches. Most of them support a wide range of 802.11 frame types, including management, control, and data frames, providing a broad attack surface for testing.

WPAxFuzz [15], one of the fully-fledged fuzzing tools, begins with generating frames based on a user-defined configuration file that includes information about the target AP and device. It utilizes random test seed generation to produce the malformed frames, where the generated seeds are injected into respective frame fields. These seeds are either compliant with the 802.11 standard or completely random. During the injection phase, *WPAxFuzz* follows an effective strategy: It transmits batches of frames for a selected frame subtype, initially sending only the MAC header without a payload. In subsequent batches, each odd-numbered frame is varied in a specific field using different seeds, with pairs of frames sharing the same seed (e.g., frames 1 and 2 have seed x, frames 3 and 4 have seed y, and so on) for easier monitoring. This continues until all fields are tested or a connection disruption is observed. Moreover, monitoring involves two parallel processes: tracking deauthentication or disassociation frames and checking the device's connectivity through ICMP echo requests. This helps detect connection disruptions or "zombie" states where the device is connected but not communicating effectively. This comprehensive approach enabled *WPAxFuzz* to identify a range of vulnerabilities in networks employing different security protocols, such as WPA 2 and WPA 3.

Another multi-faceted tool that targets all the frame types is *Owfuzz* [18]. It utilizes the open-source 802.11 protocol stack *Openwifi* to generate and inject frames at a more granular level, which allows modifying fields in PLCP Protocol Data Unit header as well. Due to the diverse structures of different frame types, *Owfuzz* employs a more sophisticated generation method, where a specific generator handles each frame type. This generator then calls a generic header generator to complete the frame header, employing different strategies for creating the frame body based on its type. For injection, *Owfuzz* leverages the *osdep* library of *aircrack-ng*, a full-featured set of tools for assessing the security of Wi-Fi networks, where it can transmit frames across multiple channels simultaneously. Similar to *WPAxFuzz*, monitoring of *Owfuzz* involves detecting anomalies through continuous frame transmission checks and interruption of higher-layer protocol connection, such as TCP/IP.

In contrast, some research studies rely on plain *Scapy* scripts rather than implementing a dedicated fuzzing tool. *Scapy* [19], a packet manipulation library written in Python, provides an API for crafting and injecting frames of any protocol, enabling users to execute specific attack scenarios. This approach is used in targeted attack studies, where the focus is narrowed to a single frame type or a specific frame field. For instance, Koenings et al. [20] presented the Channel Switch Announcement and Quiet attacks, examining the impact of receiving malicious beacon frames. By utilizing *Scapy*, they crafted beacons with malformed Channel Switch Announcement and Quiet Information Elements and transmitted them to various devices. These specially designed frames targeted weaknesses within the CSA and Quiet mechanism, in which devices are falsely signaled to switch channels or remain silent, thereby causing communication disruptions. Similarly, Raj and Sankaran [21] focus on battery drain attacks in their work. They exploited the power-saving mechanism by injecting beacons with manipulated Traffic Indication Map Information Element to trick the target devices into staying awake.

On top of that, Ferreri et al. [22] and Lee and Bahk [23] conducted experiments regarding the effects of flooding on wireless networks. They investigated various forms of flooding attacks with management frames, such as probe request frames. They demonstrated how these attacks can exploit vulnerabilities in APs as well as in client devices by overwhelming the network with an excessive amount of frames. These studies also opted for targeted frame injection and external monitoring through packet capturing tools like *Wireshark* for evaluating the outcome.

Inspired by the aforementioned works, this thesis project presents a black box fuzzing framework specifically targeting management frames. The framework combines the targeted approach of *Scapy*-based tools for frame generation and injection with a more robust monitoring capabilities akin to those found in comprehensive fuzzers. Plus, it was designed for ease of configuration and use, allowing users to effortlessly set up the wireless interface for injection by offering a Python API and execute the desired beacon-based attacks.

Chapter 4

Approach

Wireless networks face ongoing threats that compromise their security and reliability. Specifically, beacon-based vulnerabilities are a significant concern, allowing malicious parties to broadcast deceptive information and cause network disruption, as mentioned in the previous chapter. Despite the recent advancements in security protocols, the beacon-based threats are still relevant to this day, necessitating ongoing research to extensively address and mitigate the risks they pose. This chapter details the methodological approach used in this thesis project, covering the overall research design and procedural steps followed to further investigate the security issues regarding beacon frames.

4.1 A 802.11 Fuzzing Framework: Beaconfuzz

With the purpose of addressing these vulnerabilities, we implemented a standalone multi-faceted fuzzing framework called *Beaconfuzz*, focusing solely on beacon frames. The framework is developed using Python by leveraging the powerful capabilities of the *Scapy* library, which allows for frame crafting and manipulation of any protocol. Additionally, it integrates other network tools available in the Linux environment to efficiently monitor the responses. It is available in two distinct versions, each designed to meet different user needs and preferences. The first version is deployed as a package, which can be extended and easily integrated into Python projects. The package version provides an API that allows users to have full control over the system. This makes it flexible for users who prefer to develop scripts or applications that directly interact with the framework. The second version is a CLI application, which offers a streamlined out-of-the box functionality, allowing users to perform essential tasks without the need for extensive coding or configuration. It enables users to quickly set up and get started with injection and monitoring processes through command line parameters, making it efficient and easy to use for minimal tasks and automation scripts operating in a terminal environment.

4.2 Design and Capabilities of Beaconfuzz

Beaconfuzz follows the typical design of a 802.11 fuzzer, where an injection and a monitor module is implemented, as presented in Chapter 3.

4.2.1 Injection Module

The injection module of *Beaconfuzz* is further divided into two components: the crafters module and the senders module. The CLI app version of *Beaconfuzz* relies on only one crafter, namely the *ConfigCrafter*. This is because the *ConfigCrafter* class provides the functionality to parse a user-defined configuration file, inspired by *WPAxFuzz* [15], and craft beacons based on it. *Beaconfuzz* adopts a "smart fuzzing" method [17] [24] rather than employing complete randomness for crafting, meaning that the crafted beacon must align with the 802.11 standard for the injection to be effective. Therefore, the *ConfigCrafter* class validates each input parameter defined within the configuration file before crafting, ensuring that the file is correctly interpreted, and any inconsistencies are promptly identified and handled.

On the other hand, the package version implements two other crafters. The *DefaultCrafter* provides methods and generators for crafting various 802.11 frame layers. In the context of *Beaconfuzz*, the 802.11 frame layer refers to a partial frame object that encapsulates information. Crafted beacon frames consist of different frame layers; for instance the MAC Header as well the Channel Switch Announcement Information Element are both frame layers containing multiple fields. By using the methods of *DefaultCrafter*, users can craft beacons based on the default parameters defined in the settings file. With the *DefaultCrafter* being the only class that is strictly dependent on the settings file, users can incorporate their own settings file independently before installing the package. This design feature ensures that the default settings does not impose restrictions on the overall setup of *Beaconfuzz*. What's more, there is the *RandomCrafter* class, which utilizes the *DefaultCrafter* under the hood to craft beacons with randomized fields without violating the 802.11 standard. In addition, the *RandomCrafter* implements the *Spawner* class, which lets user to spawn multiple rouge networks by spreading out beacons with fake SSIDs.

The senders module includes two sender classes that, each serving distinct purposes. The *ConfigSender* class, as the name suggests, is responsible for parsing of injection parameters, such as the number of beacons to be sent or the time between consecutive transmissions, which are also defined within the configuration file. Users can craft beacons, let alone implement their own beacon generators outside of the configuration file and yet still define a transmission pattern using *ConfigSender*. However, the *ConfigSender* is not completely decoupled from the *ConfigCrafter*. It also provides methods that facilitate an all-in-one function that handles both crafting and sending of beacons based on the configuration file, unless the user specifies a particular beacon to send. The other class is the generic *Sender* class, differing from the *ConfigSender* primarily in how it obtains its input parameters. In comparison to *ConfigSender*, the generic *Sender* accepts parameters directly through the methods provided by the Python API.

Furthermore, both senders support two operational modes: synchronous and asynchronous. In synchronous mode, the sender component executes in a blocking manner, ensuring that the injection process is completed before moving on to the next task. This mode is ideal for scenarios where the surrounding devices are monitored externally, and the injection task requires strict order and precise timing. In contrast, asynchronous mode employs multi-threading, allowing beacon transmission to be concurrently executed alongside other tasks. By leveraging concurrency, users can perform injection while simultaneously observing the device behavior by utilizing the monitor module of *Beaconfuzz*.

4.2.2 Monitor Module

The monitor module of *Beaconfuzz* implements the *Sniffer* class, which also relies on *Scapy* for its core functionality. *Sniffer* provides essential methods for the real-time capturing of wireless traffic, offering monitoring and network analysis capabilities. By using *Sniffer*, users can capture the exchanged management frames between the devices within the target network. Similar to the injection module, *Sniffer* incorporates synchronous and asynchronous modes. While the synchronous mode is intended merely for monitoring, the asynchronous mode is designed to be used alongside with the injection module for capturing device responses to the injected beacons. In addition, *Sniffer* includes different filtering methods based on certain fields of the frame, such as the subtype or a specific destination address.

On top of that, it offers three logging mechanisms, each being capable of outputting the captured frame information in different formats. The first logging mechanism can output frame information to stdout in a simple text format, making it easy to read and understand for quick reviews and basic analysis. The second logger also writes the frame information to stdout but enhances readability by formatting the output in a more visually appealing manner. Instead of plain text, this logger presents the data in a well-organized table format, where each row corresponds to a captured management frame, with columns detailing key attributes such as timestamp and frame subtype. In contrast to these two loggers, the third logger outputs the data to a CSV file. Through logging in CSV format, users can import the captured frames into spreadsheet applications or data analysis software to perform detailed investigations, generate reports, and visualize network traffic trends over time.

One other notable feature of *Beaconfuzz*'s monitor module is its API for setting the wireless network interface card (WNIC) into monitor mode on Linux. This is crucial for *Beaconfuzz* to function properly, as the default managed mode of a WNIC does not support frame injection. The Python API provided by *Beaconfuzz* simplifies the process of configuring a WNIC into monitor mode by utilizing external network tools, such as *aircrack-ng* [25] and *iw* [26]. Additionally, it offers methods to seamlessly switch between channels in different frequency bands. It also provides a channel hopping method that can be used concurrently with injection and monitoring, allowing users to conduct distributed attacks across multiple channels. Through the integration of such an API for WNIC configuration, *Beaconfuzz* aims to eliminate the need for manual intervention by the user, as it can dynamically hop between different channels during the attack.

4.3 Testing with Beaconfuzz

To comprehensively assess the capabilities of *Beaconfuzz*, we conducted series of real-world experiments within both 2.4 GHz and 5 GHz network environments. Our tests involved a wide range of devices running various operating systems, providing a thorough evaluation of how *Beaconfuzz* interacts with diverse hardware and software setups. This approach allowed us to understand *Beaconfuzz*'s performance in practical scenarios, as we adopted the perspective of an attacker to closely monitor network traffic. Thus, we evaluated the impact of our fuzzing activities on the targeted devices.

During these experiments, we employed numerous attack strategies to rigorously test the resilience of the devices. These attacks were carried out by injecting beacons with malformed Information Elements to exploit the 802.11 network operations, attempting to induce unexpected behaviors in devices. In the end, we successfully identified multiple vulnerabilities based on the response analysis of every target device. Each vulnerability was documented, and our findings were used to produce evaluation reports. These reports include descriptions of the vulnerabilities and the specific conditions under which they were identified. The experiments yielded valuable insights into *Beaconfuzz*'s practical applications, showcasing its ability to uncover vulnerabilities.

Chapter 5

Implementation

This chapter covers the specifics of *Beaconfuzz* architecture, and provides a guide on how to set up and effectively use it. For this purpose, we are going to focus on the CLI app version of *Beaconfuzz*, as the package version operates in a similar fashion but with additional API capabilities for advanced use cases.

As the initial step, users must ensure that all Python dependencies and required packages are installed. *Beaconfuzz* has been tested with Python version 3.10.12, and a *requirements.txt* file is provided in the repository for installing the requirements:

```
# Package Dependencies (e.g. apt)
apt update && apt install -y wireless-tools aircrack-ng iw
# Python Dependencies
pip install -r requirements.txt
```

In addition to meeting the software requirements, users must also verify that their WNIC supports monitor mode. This verification is crucial, as monitor mode is necessary for *Beaconfuzz* to properly function.

5.1 Identifying the Target

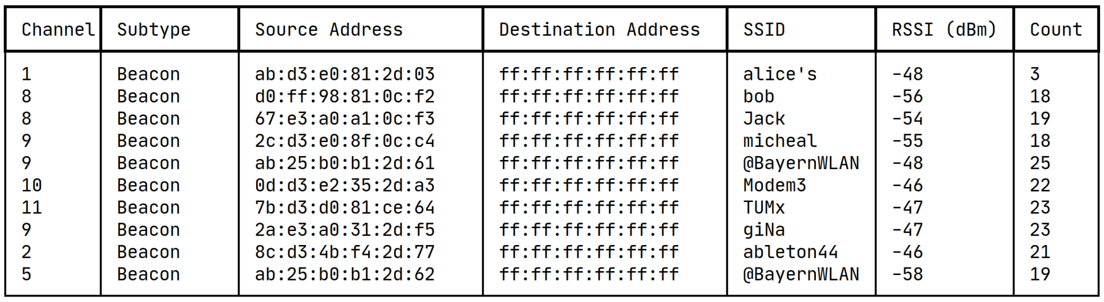
Before proceeding with the injection, users must first identify the target WLAN and extract essential parameters, such as SSID. This involves determining the channel on which the network operates by capturing the legitimate beacon frames transmitted by the AP. Thanks to the monitor module of *Beaconfuzz*, users can listen to multiple channels to detect the target network by running the app with the necessary arguments:

```
# 2.4 GHz band
./beaconfuzz -i wlan0 -m monitor -c 1 2 3 4 5 6 7 8 9 10 11 12 13 \
-t 20 -d 1 -f subtype_filter 8 -t 20 -o table

# 5 GHz band
./beaconfuzz -i wlan0 -m monitor -c 36 40 44 48 52 56 60 64 100 \
104 108 112 116 120 124 128 132 136 140 144 149 153 157 161 165 \
-t 20 -d 1 -f subtype_filter 8 -o table
```

The *-i* argument specifies the name of the interface used for listening to the wireless traffic. The second argument *-m* defines the execution mode, which is monitor mode in our case. For each different frequency band, we provide all the channel numbers using the *-c* argument. If the supported channels of the WNIC are unknown, *Beaconfuzz* offers a method to list all the available channel numbers. To specify the duration of the monitoring operation, we use the timeout *-t* argument and set it to '20' seconds.

Along with the timeout, the dwell time *-d* argument is used to indicate the number of seconds *Beaconfuzz* remains on each channel. The total timeout duration is divided into equal intervals of dwell time, ensuring balanced coverage across all channels. Since we are interested in extracting the network information, we filter out all other frames, retaining only beacons by applying a filter on the subtype '8'. Moreover, we utilize the table logger via *-o* argument, which offers a quick network analysis. An example output is shown in Figure 5.1.



```
Channel  Subtype  Source Address     Destination Address  SSID         RSSI (dBm)  Count

1        Beacon   ab:d3:e0:81:2d:03  ff:ff:ff:ff:ff:ff    alice's      -48         3
8        Beacon   d0:ff:98:81:0c:f2  ff:ff:ff:ff:ff:ff    bob          -56         18
8        Beacon   67:e3:a0:a1:0c:f3  ff:ff:ff:ff:ff:ff    Jack         -54         19
9        Beacon   2c:d3:e0:8f:0c:c4  ff:ff:ff:ff:ff:ff    micheal      -55         18
9        Beacon   ab:25:b0:b1:2d:61  ff:ff:ff:ff:ff:ff    @BayernWLAN  -48         25
10       Beacon   0d:d3:e2:35:2d:a3  ff:ff:ff:ff:ff:ff    Modem3       -46         22
11       Beacon   7b:d3:d0:81:ce:64  ff:ff:ff:ff:ff:ff    TUMx         -47         23
9        Beacon   2a:e3:a0:31:2d:f5  ff:ff:ff:ff:ff:ff    giNa         -47         23
2        Beacon   8c:d3:4b:f4:2d:77  ff:ff:ff:ff:ff:ff    ableton44    -46         21
5        Beacon   ab:25:b0:b1:2d:62  ff:ff:ff:ff:ff:ff    @BayernWLAN  -58         19
```

Figure 5.1: Running *Beaconfuzz* to monitor 2.4 GHz networks

5.2 Crafting the First Beacon

Creating a configuration file is the very first step for injection. This file must define the content of all the necessary fields related to the structure of the beacon. Additionally, the file should specify the sending parameters, such as transmission intervals, and the number of frames to be injected. The crafting process starts by creating a MAC Header, as presented in Listing 1.

```ini
# beaconfuzz/configs.ini

[MAC Header]
# Destination MAC address, optional
DA = ff:ff:ff:ff:ff:ff
# Source MAC address, mandatory
SA = d0:ff:98:81:0c:f2
# Basic Service Set ID, mandatory
BSSID = d0:ff:98:81:0c:f2
# Fragment Offset, optional
FOffset = 0
# Sequence Number, optional
SNumber = 0
```

Listing 1: MAC Header section in configuration file

The MAC Header section is mandatory, and if users omit it, *Beaconfuzz* will throw a *SectionNotFoundError* exception, which applies to all the other mandatory sections as well. The source address and BSSID options in the MAC Header are also mandatory; if not provided, an *OptionNotFoundError* exception will be raised. These options must be provided with caution, as the fuzzing process will not be effective if these addresses do not correspond to the AP of the target network. The destination address, on the other hand, is optional. If it is not defined, it is set to the broadcast MAC address by default. *Beaconfuzz* also allows fuzzing the Sequence Control field, where users can input custom values for Fragment Offset and Sequence Number. To keep track of the injected beacons, the Sequence Number is incremented with each beacon. This feature ensures that each beacon is uniquely identifiable, facilitating precise monitoring and analysis of the network's response to the fuzzing activities.

Following the MAC Header, users must also define the mandatory Beacon Body, as presented in Listing 2. Devices rely on the data within the Beacon Body to recognize the networks, assess their capabilities, and make informed decisions about which network to join [2]. The Timestamp field specifies the number of microseconds the AP has been active, and by fuzzing this value, users can assess how the network handles synchronization anomalies. The Interval field denotes the beacon interval in Time Units

(TUs), set to '100' in this example, defining how frequently the beacon frames should be transmitted [2].

```
[Beacon Body]
# Timestamp, optional
Timestamp = 124435
# Beacon Interval in TUs, optional
Interval = 100
# Capability Information, optional
Capability = privacy
# Service Set ID, mandatory
SSID = bob
# Supported Rates in Mbps, optional
Rates = 12, 18, 24, 36, 48, 72, 96, 108
```

Listing 2: Beacon Body section in configuration file

The Capability field is crucial for defining the network's functionalities. For instance, setting this field to 'privacy' indicates the use of encryption for secure communication. Users can adjust this field to announce a free, open network by setting the appropriate flags, which might attract devices looking to connect to an unsecured network [2]. The SSID field is used to set the network name, such as 'bob' in this case. Users must pay close attention when providing both the Capability and SSID options, ensuring that the injected beacons are correctly associated with the target network. Additionally, the Rates field lists the supported data rates for the network. In this example, the rates include '12, 18, 24, 36, 48, 72, 96, and 108', corresponding to standard Wi-Fi data rates. By configuring these parameters in the Beacon Body section, users can manipulate the network characteristics and evaluate how different configurations affect the network during fuzzing [2].

In addition to the mandatory Beacon Body, users can define a range of optional Information Elements (IEs) in the Beacon Extension section. These optional IEs offer enhanced customization and precise control over the beacon frames, enabling users to exploit various network operations [2] [3]. *Beaconfuzz* currently supports the fuzzing of 13 different IEs, as shown in Listing 3, where each IE has a maximum length of 255 bytes as defined by the 802.11 standard.

```
# A number of optional Information Elements (B: Bytes)
[Beacon Ext]
```

```
# FH Parameter Set IE: Dwell Time (2B), Hop Set (1B),
# Hop Pattern (1B), Hop Index (1B)
FHSet = 500, 1, 2, 0
# DS Parameter Set IE: Current Channel (1B)
DSSet = 6
# CF Set IE: Count (1B), Period (1B), Max Duration in TUs (2B),
# Duration Remaining in TUs (2B)
CFSet = 1, 2, 500, 0
# IBSS IE: ATIM Window (2B)
IBSS = 0
# TIM IE: Count (1B), Period (1B), Bitmap Control(1B),
# Partial Virtual Bitmap (1-251B)
TIM = 2, 1, 0, 0x12003
# Country IE with Country Constraint Triple(s):
# Country Code (3B), [First Channel Number (1B);
# Number of Channels (1B); Maximum Transmit Power (1B)]...
Country = DE, [1; 13; 20]
# Power Constraint IE: Local Power Constraint (1B)
PConstraint = 30
# Channel Switch Announcement IE: Mode (1B), New Channel Number (1B),
# Count (1B)
CSA = 0, 11, 1
# Quiet IE: Count (1B), Period (1B), Duration (2B), Offset (2B)
Quiet = 1, 2, 100, 0
# TPC Report IE: Transmit Power (1B), Link Margin (1B)
TPCReport = 20, 0
# ERP IE in hex string (1B):
# Non-ERP present (1 Bit), Use Protection (1 Bit),
# Barker Preamble (1 Bit), Reserved (5 Bits)
ERP = 0x00
# Extended Rates IE in Mbps
ERates = 2, 4, 11, 22
```

Listing 3: Beacon Ext section in configuration file

Frequency Hopping (FH) Parameter Set IE

FHSet = 500, 1, 2, 0: This option includes the Dwell Time, Hop Set, Hop Pattern, and Hop Index. The Dwell Time specifies the duration in TUs that the AP stays on a channel before hopping onto another. The Hop Set and Pattern define the hopping sequence used in networks adopting the FHSS PHY. The Hop Set contains various hopping patterns defined by the 802.11 standard. The Hop Index is used to indicate the current channel within the hopping sequence [2] [4].

Direct Sequence (DS) Parameter Set IE

DSSet = 6: Networks utilizing the DSSS PHY include the DS Parameter Set IE, which consists of only one-byte field that specifies the current operating channel of the AP [2].

Contention-Free (CF) Parameter Set IE

CFSet = 1, 2, 500, 0: This option includes the CFP Count, CFP Period, Maximum Duration in Time Units (TUs), and Duration Remaining in TUs. These parameters control the Contention-Free Period (CFP) within the network, during which certain devices have exclusive access to the medium, preventing others from transmitting and thus avoiding collisions in an enforced way. The CFP is often used in real-time applications where timely and reliable transmission is crucial [2].

The CFP Count field specifies the number of beacon frames to be sent before the next CFP starts, whereas the CFP Period indicates the period between the consecutive CFPs. The Maximum Duration field sets the maximum length of each CFP in TUs and the Duration Remaining is used to inform devices about the remaining TUs of the current CFP [2].

Independent Basic Service Set (IBSS) IE

IBSS = 0: Independent Basic Service Set (IBSS) or an ad-hoc network is a type of WLAN where devices communicate directly with each other without the need for an AP. Devices in an IBSS are responsible for transmitting beacons, including the IBSS IE, which contains the Announcement Traffic Indication Map (ATIM) window field. The ATIM mechanism facilitates efficient power management via ATIM frames by notifying other devices about pending transmissions, informing them when to wake up. The ATIM window field refers to the number of TUs between these ATIM frames [2].

Traffic Indication Map (TIM) IE

TIM = 2, 1, 0, 0x12003: This option includes the DTIM Count, DTIM Period, Bitmap Control, and Partial Virtual Bitmap. In an infrastructure network, APs buffer frames for the devices operating in low-power mode, allowing for reduced power consumption by minimizing the need for staying awake, similar to ATIM in IBSS. Devices are represented within the Partial Virtual Bitmap, with each bit corresponding to a specific device. If an AP wants to inform a device about a buffered frame, it sets the bit associated with it and include this in its beacon to wake up the device. The Partial Virtual Bitmap field must be provided as a hex string; otherwise, *Beaconfuzz* will raise a *ValueError* exception. Moreover, the Bitmap Control field consists of two subfields: Bit 0 is set if the buffered frames are multicast, while the remaining bits specify the Bitmap Offset, marking the start of the Partial Virtual Bitmap. This allows a portion of the Bitmap Control to be used for the Partial Virtual Bitmap [2] [21].

The delivery of the buffered frames is managed via DTIM frames, which are special beacons including a DTIM count of 0. The DTIM period defines the number of non-DTIM beacon intervals between these frames, and the DTIM count decreases with each beacon, starting from the period value and counting down to 0. The actual delivery occurs immediately after each DTIM frame [2] [21].

Country IE

Country = DE, [1; 13; 20]: This option includes the Country Code and a variable list of Country Constraint Triples. Each triple includes the First Channel Number, Number of Channels, and Maximum Transmit Power in dBm for that country. This indicates the starting channel number and the subsequent channels that are subjected to the specified transmit power limit [2].

Power Constraint IE

PConstraint = 30: This option includes the Local Power Constraint in dBm, which is used alongside with the regulatory power constraint in the Country IE. The upper limit for the transmit power is determined by subtracting the Local Power Constraint from the Maximum Transmit Power [2].

Channel Switch Announcement (CSA) IE

CSA = 0, 11, 1: This option includes the Channel Switch Mode, New Channel Number, and Channel Switch Count. To facilitate Dynamic Frequency Selection, APs include the CSA IE in beacon frames to inform devices of an impending channel switch. The Channel Switch Mode indicates whether devices should cease packet transmission immediately until the switch occurs (mode set to 1) or continue transmitting (mode set to 0). The New Channel Number, as the name suggests, indicates the new channel for the switch. The Channel Switch Count specifies the number of beacons to be transmitted before the switch. After the beacon with a count value of 0 is delivered, the channel switch can happen without any additional notice [2].

Quiet IE

Quiet = 1, 2, 100, 0: This option includes the Quiet Count, Quiet Period, Quiet Duration, and Quiet Offset, which manage quiet periods to the minimize interference with external radar technologies. Similar to the Channel Switch Count in CSA IE, the Quiet Count specifies the number of beacon intervals until the quiet period starts. The Quiet Period indicates how often the quiet periods occur, in terms of beacon intervals. The Quiet Duration defines the length of the quiet period in TUs, ensuring no transmissions occur during this time. Lastly, the Quiet Offset determines the start time of the quiet period relative to a beacon interval, indicating the number of TUs after the beacon interval [2].

Transmit Power Control (TPC) Report IE

TPCReport = 20, 0: This option includes the Transmit Power and Link Margin. The Transmit Power field indicates the power level, in dBm, at which the frame containing when this IE was sent. The Link Margin field represents the safety margin, in decibels, needed by the device. These fields are included in management frames, helping devices understand the link attenuation and adjust their transmission power accordingly [2].

Extended Rate PHY (ERP) IE

ERP = 0x00: This option includes the bit fields Non-ERP present, Use Protection, Barker Preamble, Reserved within a single one-byte hex string. The Non-ERP Present bit is set when older, non-802.11g devices join the network. The Use Protection bit is set when there are devices in the network that do not support 802.11g data rates. The Barker Preamble Mode bit is set if associated devices are unable to utilize the short preamble mode. These fields collectively ensure smooth operation and compatibility in mixed-mode environments [2].

Extended Rates IE

ERates = 2, 4, 11, 22: This option contains the additional supported data rates, extending the rates defined in the Beacon Body [2].

Apart from the aforementioned Information Elements, *Beaconfuzz* includes the Robust Security Network (RSN) Information Element, which is given its own dedicated section due to its extensive nature.

```
# RSN Information Element, also part of Beacon Ext (B: Bytes)
[RSN Info]
# RSN Version (2B)
Version = 1
```

```
# Group Cipher Suite (4B)
GroupCS = 0x000fac04
# Pairwise Cipher Suite Count (2B)
PairwiseCSC = 2
# Pairwise Cipher Suite (4B * Pairwise Cipher Suite Count)
PairwiseCS = 0x000fac04, 0x000fac05
# Authentication Suite Count (2B)
AKMCSC = 2
# Authentication Suite (4B * Authentication Suite Count)
AKMCS = 0x000fac02, 0x01020a0b
# RSN Capabilities (2B)
RSNCaps = 0x0000
# Pairwise Master Key Count (2B)
PMKC = 3
# Pairwise Master Key List (16B * Pairwise Master Key Count)
PMKL = 0x00ca, 0x123456789a, 0xffffff
```

Listing 4: RSN Info section in configuration file

Robust Security Network (RSN) IE

RSN is fundamental to the enhanced security features introduced with the 802.11i amendment, providing secure communication by protecting the data frames. This is achieved through the transmission of beacons including the RSN IE to announce several RSN parameters. It begins with the Version field, specifying the RSN protocol version to ensure compatibility. The Group Cipher Suite identifies the cipher used for protecting broadcast and multicast frames, whereas the Pairwise Cipher Suite fields incorporate the ciphers used for unicast frame protection, with the number of ciphers determined by the count parameter [2] [4].

The Authentication and Key Management (AKM) Suite fields detail the authentication methods. The RSN Capabilities field includes flags for extended features supported by the AP such as pre-authentication. Finally, the Pairwise Master Key (PMK) List fields are included for facilitating faster hand-offs, which are the transitions of a device's connection from one AP to another. When a device associates with an AP, it can provide a list of PMKs, enabling a quicker reconnection without the need for full authentication [2].

5.3 Injection & Monitoring

Having systematically crafted our beacon frame, we can now proceed with the injection. To achieve this, it is necessary to define two additional parameters within the same configuration file utilized for crafting, as shown in Listing 5.

```
# Injection parameters
[Packet Send]
# Actual interval in seconds between consecutive beacon transmissions
inter = 0.1
# Number of beacons to be transmitted
count = 10
```

Listing 5: Packet Send section in configuration file

These parameters pertain to the transmission behavior of the beacons and are specified in a separate section called Packet Send. Specifically, the first option, *inter*, determines the actual interval in seconds between consecutive beacon transmissions, set to 0.1 seconds in this instance. The second parameter, *count*, specifies the total number of beacons to be sent, which is set to 10. If the *count* option is omitted, *Beaconfuzz* defaults to a continuous injection mode, sending beacon frames until the end of the defined timeout period. By configuring these parameters, users can control the frequency of the injected beacon frames, which offers a systematic injection process.

Beaconfuzz supports two modes for injection. One of them is the send mode, designed to craft and transmit beacon frames by utilizing the injection module. It relies on the asynchronous *ConfigSender*, which initiates a daemon thread that runs in the background and lives until the main program exits. It also allows for concurrent channel hopping in the main thread, which ensures seamless transmission of beacon frames across multiple frequencies.

Assuming we have successfully identified our target network and determined its operating channel, we fire up *Beaconfuzz* with the following parameters to inject beacons using send mode:

```
./beaconfuzz -i wlan0 -m send -p tests/configs.ini -c 8 -t 10
```

```
# tests/configs.ini
[MAC Header]
SA = d0:ff:98:81:0c:f2
```

```
BSSID = d0:ff:98:81:0c:f2
FOffset = 0
SNumber = 0

[Beacon Body]
Timestamp = 124435
Interval = 100
Capability = privacy
SSID = bob
Rates = 12, 18, 24, 36, 48, 72, 96, 108

[Beacon Ext]
DSSet = 1

[Packet Send]
inter = 0.3
```

Listing 6: Configuration file for manipulating DS Parameter Set IE

In the example presented in Listing 6, we are targeting the network with the SSID 'bob', which is operating on the 8th channel (refer to 'bob' in Figure 5.1). The path *-p* argument points to the configuration file residing in the 'tests' folder, which defines the structure of the beacon to be injected. To ensure our crafted beacon frame accurately mimic the target network, we carefully configure the MAC Header and Beacon Body sections. The source address and BSSID are both set to 'd0:ff:98:81:0c:f2', making the frames appear as if they are coming from the authorized AP. Since we did not specify the destination address, it is automatically set to the broadcast MAC. Additionally, the Capability option is configured to privacy, reflecting the network's use of security features.

As our primary fuzzing element, we selected the DS Parameter Set Information Element, in which we manipulate the 'current channel' field to be set to 1, as it differs from the actual channel on which the network is operating. Finally, we set the interval value to 0.3 seconds and leave out the 'count' option, as we want to constantly inject during entire timeout period, which is set to 10 seconds. The output contains the injected beacon frame with the respective fields and should look similar to the following:

```
$ ./beaconfuzz -i wlan0 -m send -p tests/configs.ini -c 8 -t 10
Searching for processes that may interfere with the monitor mode: airmon-ng check
Killing the interfering processes via systemctl: airmon-ng check kill
Setting interface to monitor mode: airmon-ng start wlan0
Changing channel to 8: iw dev wlan0mon set channel 8
```

```
Sending beacons...
###[ RadioTap ]###
  version   = 0
  pad       = 0
  len       = 9
  present   = Flags
  Flags     = FCS
  notdecoded= ''
###[ 802.11-FCS ]###
 subtype   = Beacon
 type      = Management
 proto     = 0
 FCfield   =
 ID        = 0
 addr1     = ff:ff:ff:ff:ff:ff (RA=DA)
 addr2     = d0:ff:98:81:0c:f2 (TA=SA)
 addr3     = d0:ff:98:81:0c:f2 (BSSID/STA)
 SC        = 0
 fcs       = 0x4023f1f0
###[ 802.11 Beacon ]###
timestamp = 124435
beacon_interval= 100
cap       = privacy
###[ 802.11 Information Element ]###
   ID        = SSID
   len       = 3
   info      = 'bob'
###[ 802.11 Rates ]###
   ID        = Supported Rates
   len       = 8
   rates     = [12.0 Mbps, 18.0 Mbps, 24.0 Mbps, 36.0 Mbps, 48.0 Mbps, 8.0(B) Mbps,
32.0(B) Mbps, 44.0(B) Mbps]
###[ 802.11 DSSS Parameter Set ]###
   ID        = DSSS Set
   len       = 1
   channel   = 1

......................................Unsetting monitor mode: airmon-ng stop wlan0mon
Restarting processes: ['avahi-daemon', 'NetworkManager', 'wpa_supplicant']
```

Before every injection or monitoring process, *Beaconfuzz* prepares the network interface via *aircrack-ng*, ensuring that no processes interfere with the monitor mode. This is done by executing the command 'airmon-ng check', which searches for any processes that might disrupt the monitor mode operation. Once identified, these processes are terminated via 'airmon-ng check kill'. After that, a new interface called 'wlan0mon' is generated via 'airmon-ng start wlan0', which can operate in monitor mode [25]. Finally, the channel is switched to the specified channel number by utilizing *iw* [26].

Right after the respective process is complete, it is essential to properly unset the monitor mode and restart the necessary network processes. *Beaconfuzz* achieves this by running the command 'airmon-ng stop wlan0mon', which stops the monitor mode on the specified interface. Following this, it is crucial to restart several key processes to restore the normal network functionality [25].

The other execution mode of *Beaconfuzz* is called the combined mode, which allows for sending and monitoring of the management frames concurrently. This dual functionality allows for the real-time capture of device responses, offering a thorough evaluation of the impact of the injected beacons. For this purpose, the combined mode leverages both the asynchronous *ConfigSender* and the asynchronous *Sniffer*. The *Sniffer* class extends the *AsyncSniffer* class by *Scapy* [19], which provides the out-of-the-box functionality for asynchronous packet capturing. As mentioned in Chapter 4, the *Sniffer* offers three loggers; one of which, the table logger, was showcased in the very first example for basic frame capture (refer to Figure 5.1). The other two loggers, stdout and CSV, will now be presented in detail.

To adapt the above example for monitoring the device responses as well, we must configure *Beaconfuzz* to operate in combined mode[1]:

```
$ ./beaconfuzz -i wlan0 -m combined -p tests/configs.ini -c 8 -t 10 -o stdout
...

Channel 8 RadioTap / Dot11FCS / Dot11Beacon / SSID='House' / Dot11EltRates / Dot11EltDSSSet /
Dot11EltERP / Dot11EltRates / Dot11EltHTCapabilities / Dot11Elt / Dot11EltMicrosoftWPA

Channel 8 RadioTap / Dot11FCS / Dot11Beacon / SSID='bob' / Dot11EltRates / Dot11EltDSSSet

Channel 8 RadioTap / Dot11FCS / Dot11ProbeResp / SSID='bob' / Dot11EltRates / Dot11EltDSSSet /
Dot11EltHTCapabilities / Dot11Elt / Dot11EltRSN / Dot11Elt / Dot11Elt / Dot11EltVendorSpecific

Channel 8 RadioTap / Dot11FCS / Dot11ProbeReq / SSID='' / Dot11EltRates / Dot11EltRates /
Dot11EltHTCapabilities

Channel 8 RadioTap / Dot11FCS / Dot11Beacon / SSID='bob' / Dot11EltRates / Dot11EltDSSSet

Unsetting monitor mode: airmon-ng stop wlan0mon
Restarting processes: ['avahi-daemon', 'NetworkManager', 'wpa_supplicant']
```

The stdout logger outputs the captured frame data as well as the current channel number directly to the terminal, allowing for immediate feedback during the fuzzing process. The captured frame data includes various frame layers, providing insights into the structure and contents of each frame. In the sample output above, each line beginning with the channel number signifies a captured frame. For instance, a probe response frame intended for our target network can be identified in the third line. Notably, we can also observe the beacons we injected, as indicated by the second and fifth line.

In contrast, the CSV logger records the captured data into a structured CSV file, offering a detailed evaluation and easier integration with other data analysis tools. To make use

[1] Informational messages regarding the monitor mode and the structure of the injected beacon are omitted in the output and represented as '. . . ' for clarity.

of the CSV logger, we simply have to modify the value of the *-o* argument by specifying a CSV file name with its extension. A sample output is presented below:

```
$ cat output.csv
channel=8,timestamp=1722973908.7414858,subtype="Beacon",direction=tx,
source_address="d0:ff:98:81:0c:f2",destination_address="ff:ff:ff:ff:ff:ff",
ssid="bob",rssi=""
channel=8,timestamp=1722973912.8650205,subtype="Action",direction=rx,
source_address="2e:b9:36:8c:72:e6",destination_address="ff:ff:ff:ff:ff:ff",
ssid="",rssi="-52dBm"
```

In contrast to the stdout logger, the CSV logger includes additional fields such as Received Signal Strength Indicator (RSSI) and timestamp. RSSI measures the power level of the received signal and the timestamp indicates the exact system time each frame was captured, which is essential for creating a real-time data record. It also specifies the direction of a frame, marking those transmitted by *Beaconfuzz* as 'tx' and the monitored ones as 'rx'.

Chapter 6

Evaluation

In this chapter, we will evaluate the performance of *Beaconfuzz*. The evaluation aims to assess the framework's ability to craft specific beacon frames, inject them into the target network, and capture and analyze the device responses. To accomplish this, we carried out an extensive series of real-world experiments, which spanned both the 2.4 GHz and 5 GHz frequency bands, targeting an array of devices. This chapter details the experimental setup, results, and a discussion of the findings.

6.1 Test Environment

The experiments were conducted in a residential place, providing a realistic environment to simulate a typical Wi-Fi network encountered in everyday use. The target networks of 2.4 and 5 GHz bands were both configured in infrastructure mode, with a TP-Link Archer C50 router as the AP. The network on the 2.4 GHz band, identified by the SSID "Bowser", was set on channel 2 utilizing WPA2-PSK. The 5 GHz network "Bowseros" operated on channel 36, also secured with WPA2-PSK. The supported modes and BSSID for these target networks are listed in Table 6.1.

As for the target devices, the setup includes wireless devices such as smartphones, laptops, and IoT devices, all connected to the AP. Each target device operates on a different operating system or version, utilizing various wireless drivers and firmware, as shown in Table 6.2. Apart from the targets, the Lenovo ThinkPad 13 G2, running Ubuntu 22.04.4 LTS with the iwlwifi 6.5.0-45-generic wireless driver, served as the adversary device in our experiments. Equipped with a dual-band Intel Wireless 8265 / 8275 WNIC, this laptop supported monitor mode for performing packet injection and monitoring in its vicinity. Its hardware reliability and performance ensured the device could handle the

Target Networks					
Band	**Channel**	**SSID**	**BSSID**	**Mode**	**Security**
2.4 GHz	2	Bowser	40:ae:30:c7:bf:e1	802.11bgn mixed	WPA2-PSK
5 GHz	36	Bowseros	40:ae:30:c7:bf:e0	802.11a/n/ac mixed	WPA2-PSK

Table 6.1: An overview of the target networks

Adversary			
Type	**Model**	**OS**	**Wireless Driver**
Laptop	Lenovo ThinkPad 13 G2	Ubuntu 22.04.4 LTS	iwlwifi 6.5.0-45-generic
Target Devices			
Type	**Model**	**OS**	**Wireless Driver**
Router	TP-Link Archer C50	-	-
Smartphone	iPhone 15	iOS 17.5.1	-
Smartphone	Google Pixel 8	Android 14	Google WiFi ES
Tablet	iPad Air 4th Gen	iPadOS 17.5.1	-
Laptop	Lenovo Ideapad L340	Windows 11 Pro	Realtek 8821CE WLAN
Laptop	MacBook Pro 2019	macOS Sonoma 14.5	Broadcom WLAN
IoT	Raspberry Pi Zero W	Raspberry Pi OS	brcmfmac 7.45.98
IoT	Raspberry Pi 3 B+	Raspberry Pi OS	brcmfmac 7.45.234

Table 6.2: An overview of the test devices

frame capture and processing required for our experiments, providing consistent and accurate results across multiple test scenarios.

6.2 Performed Attacks

Our initial approach was to fuzz all the fields within a beacon without being selective. This approach was intended to ensure that no aspect of the beacon frame was overlooked and that any potential weaknesses could be identified. Nevertheless, as the experiments progressed and based on the insights gained from the existing studies [3] [20] [21] [23], it became evident that focusing specifically on the Information Elements was advantageous. By targeting IEs, we were able to exploit the 802.11 network operations, taking advantage of the fact that the devices configure themselves based on the information contained within these elements. Despite our efforts on fuzzing each IE, the majority of the IEs did not yield significant responses from the devices. However, we identified a small but critical subset of IEs that produced notable results. Specifically, the Channel

Switch Announcement IE and Quiet IE emerged as particularly sensitive to fuzzing. Therefore, we designed and carried out targeted attacks focusing on these IEs.

6.2.1 Initial Setup

Before initiating each attack, it is necessary to create a *Beaconfuzz* configuration file. In this file, we have to adapt certain sections and options to ensure that our malformed beacon blends seamlessly with the target network communication, as shown in Listings 7 and 8.

```
# For the 2.4 GHz target network
[MAC Header]
# Default DA, broadcast MAC
SA = 40:ae:30:c7:bf:e1
BSSID = 40:ae:30:c7:bf:e1

[Beacon Body]
Timestamp = 1284435
Interval = 100
Capability = privacy
SSID = Bowser
# Default Rates, [12, 18, 24, 36, 48, 72, 96, 108] Mbps
```

Listing 7: Mandatory sections for the 2.4 GHz network

```
# For the 5 GHz target network
[MAC Header]
SA = 40:ae:30:c7:bf:e0
BSSID = 40:ae:30:c7:bf:e0

[Beacon Body]
Timestamp = 1284435
Interval = 100
Capability = privacy
SSID = Bowseros
```

Listing 8: Mandatory sections for the 5 GHz network

For each target network, specific modifications are made to both the MAC Header and the Beacon Body section of the frame. The destination address is assigned the default broadcast MAC address, whereas the source address and BSSID fields are set to the respective MAC address of the AP, as presented in Table 6.1. Furthermore, the SSID field is adapted to match the legitimate network name. For the capability field, we provide ’privacy’ to align with each network’s security configuration. Other parameters, such as the timestamp and beacon interval, can be set to any value, as it turns out they did not influence the fuzzing process.

6.2.2 Channel Switch Attack

The first attack we performed was the Channel Switch Attack, which exploits the Channel Switch Announcement (CSA) Information Element (IE). The attack targets the mechanism by which networks manage and adapt to interference caused by primary users operating within the same band, such as weather radars. APs include this CSA IE within the beacon frames, instructing all devices to switch to a different channel, which can disrupt the communication for a specified duration [2] [3] [20].

The custom CSA IE crafted for this attack includes three fields: *Channel Switch Mode*, *New Channel Number* and *Channel Switch Count*. These fields were tailored differently for each target network to reflect the appropriate frequency parameters. Within the same configuration file, we started by setting the *Channel Switch Mode* to 1, which requires all devices receiving the beacon frame to immediately cease transmission until the channel switch occurs. Conversely, we specified the *New Channel Number* as ’13’ and ’100’ for 2.4 GHz and 5GHz bands respectively, ensuring that the devices were directed to switch to appropriate channels based on the frequency band in use [2] [20] (refer to the channel used by each target network in Table 6.1).

Finally, the *Channel Switch Count* was set to its maximum value of 255, meaning that the devices have to wait for 255 beacon intervals before actually switching to the new channel. Together with the mode and count, we could control the behavior of the channel switch operation and exploited it to disrupt the communication for 255 Time Units, which is approximately 26 seconds, considering the beacon interval value of each target network is 100 TUs [2] [20]. The configuration parameters are outlined in Listings 9 and 10.

```
# For the 2.4 GHz target network
[Beacon Ext]
# CSA IE: Mode (1B), New Channel Number (1B), Count (1B)
CSA = 1, 13, 255
```

Listing 9: Channel Switch Announcement IE option for the 2.4 GHz network

```
# For the 5 GHz target network
[Beacon Ext]
CSA = 1, 100, 255
```

Listing 10: Channel Switch Announcement IE option for the 5 GHz network

6.2.3 Quiet Attack

The second attack focuses on the Quiet Information Element, a critical component of the Dynamic Frequency Selection network operation for the 5 GHz band. It is also defined by the 802.11 standard particularly for avoiding interference with primary users. To accurately detect the signals of primary users, APs transmit beacons with the Quiet IE to inhibit transmissions by other devices in the network for a specified duration. The Quiet IE comprises four fields to manage the quiet periods: *Quiet Count*, *Quiet Period*, *Quiet Duration*, and *Quiet Offset* [2].

```
# For both 2.4 GHz and 5 GHz target networks
[Beacon Ext]
# Quiet IE: Count (1B), Period (1B), Duration (2B), Offset (2B)
Quiet = 1, 1, 65535, 0
```

Listing 11: Quiet IE option for both target networks

In this attack setup, demonstrated in Listing 11, the *Quiet Count* was set to 1, indicating that the quiet period should begin immediately after the beacon is received, equivalent to the count field in CSA IE. The *Quiet Period* was configured to 1, indicating that there is a scheduled quiet period. The *Quiet Duration* specified the number of TUs during which devices should remain silent. It was set to the maximum value of 65535, which is approximately 67 seconds, effectively forcing devices to cease the transmission for the longest period allowed by the standard. On the other hand, the *Quiet Offset* was set to 0 to indicate that the quiet period would begin without any further delay. By setting these parameters, we intended to achieve 67 seconds of uninterrupted silence [2] [3] [20].

6.2.4 Beacon Flooding Attack

Unlike the previous attacks that focused on manipulating Information Elements, the Beacon Flooding Attack overwhelms the network by transmitting an excessive number of beacon frames. To achieve an effective outcome, it was essential to forge beacons that have the same BSSID as the AP while randomizing the SSIDs with each transmitted beacon. This method created the appearance of numerous networks all seemingly originating from the same source, making it quite difficult for devices to differentiate between legitimate and fake beacons [23].

Since the CLI app version of *Beaconfuzz* does not provide the flexibility to transmit beacons with constantly changing SSIDs, we opted to use the package version for this attack, which includes the *Spawner* class as part of the *RandomCrafter* module, as shown in Listing 12.

```python
# beaconfuzz/crafters/random_crafter

class Spawner:
    """Provides methods for spawning networks with arbitrary SSIDs"""

    @staticmethod
    def spawn(ssid_len: int = 8, ssid_num: int = 8) -> BeaconGenerator:
        """Generates Beacon frames with random SSIDs given ssid_num.

        The first frame starts with the Sequence Number 0, incrementing
            with each new packet.

        Args:
            ssid_len (int): Max Length of each SSID. By default, `8`.
            ssid_num (int): Number of frames to be generated
                with distinct ssids. By default, `8`.

        Yields:
            Beacon: Next beacon consisting of multiple 802.11
                layers.
        """
        ...
```

Listing 12: *Spawner* class defined in *RandomCrafter*

The key method within this class is *spawn*, which takes two parameters: *ssid_len* and *ssid_num*. The *ssid_len* parameter indicates the maximum length of each SSID, with a default value of 8 characters. The *ssid_num* parameter, on the other hand, determines the number of distinct SSIDs to be generated. This allows for the creation of a specified number of beacons, each with a different SSID, ensuring a dynamic set of network names. The method begins by initializing the sequence number for the first beacon frame of each distinct SSID to 0 and then increments it with each subsequent packet. This implies that each SSID has its own sequence number that increases independently, ensuring each beacon is perceived as unique by the receiving devices, even though they all share the same BSSID. As a result, method then returns a *BeaconGenerator*, which yields a beacon frame consisting of only the MAC Header and Beacon Body. The BSSID and other frame layers are defined in a global 'settings.py' file, which is used for crafting the beacons, ensuring consistency across all generated frames. The details of transmission and monitoring are discussed in Subsection 6.3.3.

6.3 Results

This section presents the results of the attacks performed, including the Channel Switch, Quiet, and Beacon Flooding attacks. We identified several vulnerabilities in the devices, revealing weaknesses in their handling of different aspects of 802.11 network operations. In these experiments, we used a consistent beacon structure and content, as presented in Section 6.2, across all devices to ensure a controlled and comparable testing environment. However, it became evident that not all target devices responded uniformly to the injection patterns employed in these attacks. Therefore, while the beacon content remained the same, we had to adjust the transmission parameters, such as the frequency and the intervals between each transmission, tailoring the attack strategies to each device.

To detect the impact of our attacks on network connectivity, we configured the tested devices to actively use the ping utility, inspired by the approach taken in *WPAxFuzz* [15] and *Owfuzz* [18]. The ping utility relies on the Internet Control Message Protocol (ICMP) to send echo request packets to a specified IP address. Each device was set to continuously ping 8.8.8.8, the IP address of Google's public DNS servers, chosen for its consistent uptime and reachability. In addition, each ping was recorded with a timestamp to accurately track the timing and impact of network disruptions. As some devices lacked a built-in ping utility, we relied on terminal emulators that implement this functionality. This allowed us to run ping commands across all devices, ensuring that even devices without native support for network diagnostics could be included in the testing process. The failure of ICMP replies serves as a critical indicator of Wi-Fi disruption. ICMP operates at the network layer, and if these echo requests fail or the replies are interrupted, it points to underlying issues with the lower layer connection. This is because device is either losing communication with the AP, hindering its ability to manage both incoming and outgoing traffic.

To enhance the precision of our testing, each attack was conducted with only one target device associated with the AP. This approach was essential for isolating the effects of the attack on the specific device, as the presence of additional devices could generate competing traffic, or trigger unintended responses from the AP.

Device	Beacon Count	Interval
iPhone 15	10	0.2s
Google Pixel 8	1	-
iPad Air 4th Gen	10	0.2s
Lenovo Ideapad L340	40	0.5s
MacBook Pro 2019	1	-
Raspberry Pi Zero W	30	0.2s
Raspberry Pi 3 B+	30	0.2s

TABLE 6.3: Transmission parameters for Channel Switch Attack

6.3.1 CHANNEL SWITCH ATTACK

Target device reactions varied significantly depending on the specific configuration used. To address this, we experimented with various combinations of transmission parameters, to determine which settings were most effective in eliciting a response from each device. Through this iterative process, we were able to identify the respective transmission parameters, as shown in Table 6.3.

ATTACK EXECUTION

To carry out the attack, we integrated the selected transmission parameters into our aforementioned configuration file, an example of which is detailed in Listing 13.

```ini
# tests/csa_lenovo_2_4.ini

[MAC Header]
# 2.4 GHz
SA = 40:ae:30:c7:bf:e1
BSSID = 40:ae:30:c7:bf:e1
# 5 GHz
# SA = 40:ae:30:c7:bf:e0
# BSSID = 40:ae:30:c7:bf:e0

[Beacon Body]
Timestamp = 1284435
Interval = 100
Capability = privacy
# 2.4 GHz
SSID = Bowser

```

```
# 5 GHz
# SSID = Bowseros

[Beacon Ext]
# 2.4 GHz
CSA = 1, 13, 255
# 5 GHz
# CSA = 1, 100, 255

# Transmission Parameters for Lenovo Ideapad L340
[Packet Send]
inter = 0.5
count = 40
```

Listing 13: Configuration file of Lenovo Ideapad L340 for Channel Switch Attack

In the configuration file for Lenovo Ideapad L340, two critical parameters define the injection pattern: *inter* and *count*. The *inter* parameter specifies the actual time interval between consecutive beacon transmissions, set to 0.5 seconds in this case. On the other hand, the *count* parameter defines the total number of beacon frames to be sent, which is set to 40. Having crafted our beacon frame, the next step was to inject it. We decided to run the injection for a duration of 70 seconds, as it aligns with the chosen injection pattern and ensures sufficient time to observe the impact on the target device. During this period, we not only injected the malformed beacons into the network but also simultaneously monitored exchanged frames between the AP and the target device in real-time. To achieve this, we executed the application with the specific parameters:

```
# 2.4 GHz band
./beaconfuzz -i wlp3s0 -m combined -p tests/csa_lenovo_2_4.ini -c 2 \
-t 70 -o output.csv

# 5 GHz band
./beaconfuzz -i wlp3s0 -m combined -p tests/csa_lenovo_5.ini -c 36 \
-t 70 -o output.csv
```

We started by specifying the wireless interface to be put into monitor mode via *-i*. Following the interface, we set the execution mode to 'combined', as it allows for concurrent injection and monitoring. For each different frequency band, we provided the respective configuration file path that contains the tailored settings. We then set the channel using the *-c* argument to target appropriate network. Finally, we set the duration of the attack to 70 seconds via *-t* and specified an output CSV file, where the captured frames will be logged for further analysis.

As a result, we recorded a variety of management frames exchanged between AP and the target devices. These frames included probe and reassociation requests sent by the target device, and action frames that are transmitted by the AP to maintain data integrity, even in the face of network disruptions. The specific frame subtypes and their frequency varied depending on the target device, reflecting the different ways each device responded to the attack.

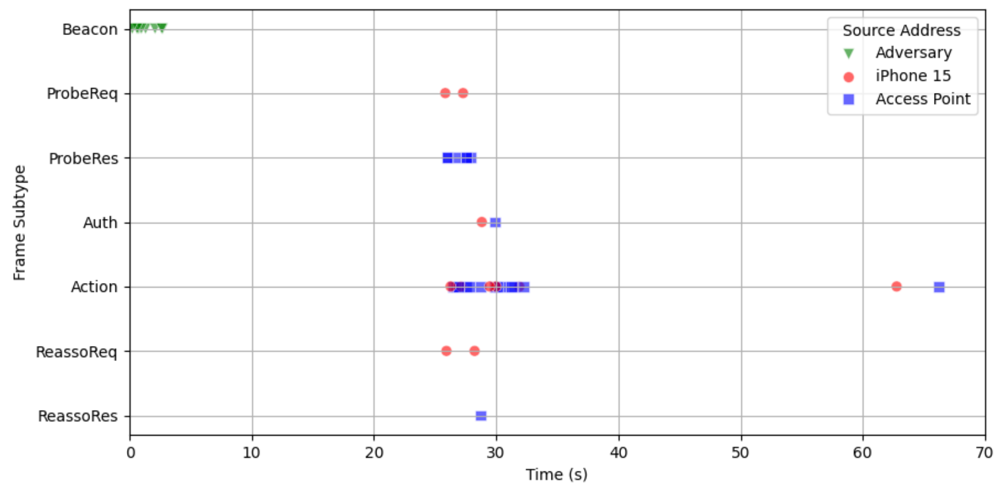
Findings

We initially assumed that the 5 GHz band would be vulnerable to the attack due to the implementation of channel switching as part of Dynamic Frequency Selection (DFS). However, when testing these devices in the 5 GHz target network, a notable difference was observed. Despite using the same attack configuration as with the 2.4 GHz band, the devices operating on the 5 GHz band ignored the injected beacon frames, resulting in little to no impact from the attack. We also tested out different injection patterns, but they were similarly ineffective. Contrary to our expectations, the attack had significant effects within the 2.4 GHz band, leading to Denial-of-Service (DoS) vulnerabilities of varying durations, as demonstrated in Table 6.4. This is why our focus had shifted to the 2.4 GHz band throughout the analysis.

Device	**DoS Duration (2.4 GHz)**	**DoS Duration (5 GHz)**
iPhone 15	30.2s	0s
Google Pixel 8	34.3s	2.6s
iPad Air 4th Gen	29.1s	0s
Lenovo Ideapad L340	5.7s	0s
MacBook Pro 2019	17s	0s
Raspberry Pi Zero W	10.3s	-
Raspberry Pi 3 B+	7.2s	2.3s

Table 6.4: Denial-of-Service durations for Channel Switch Attack

When analyzing the responses of the Apple devices, several similarities were observed in the way they handled the injected beacon frames. The iPhone 15 and iPad Air 4th Gen, both configured with 10 beacon frames transmitted at 0.2-second intervals, experienced DoS events lasting about 30 seconds. Similarly, the Google Pixel 8, despite being tested with only one beacon, also encountered a DoS event with a slightly longer duration. During the DoS period, devices failed to receive incoming ICMP replies, resulting in a timeout. In response, they attempted to send probe requests on both channels 2 and 13, but received a probe response only on channel 2, where the AP was operating.



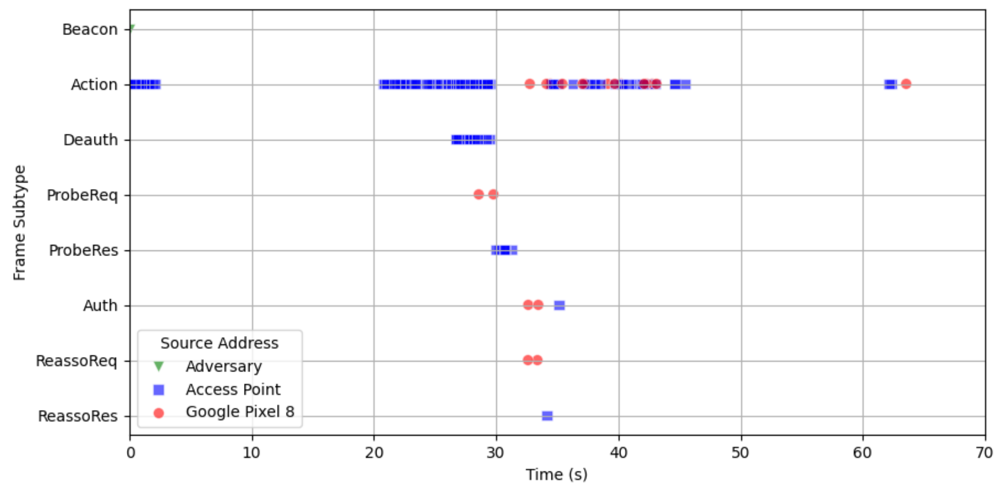


Figure 6.1: Communication timeline of Channel Switch Attack for iPhone 15 and Google Pixel 8

Following this, the devices also exchanged authentication and reassociation frames, indicating that they were attempting to re-establish the connection with the AP after the disruption. The exchanged frames captured on channel 2 are illustrated in Figure 6.1, where triangles indicate injected beacons, circles represent frames sent by the target device, and squares represent frames sent by the AP.

In contrast, the MacBook Pro 2019, which was subjected to a more immediate attack with only 1 beacon frame, experienced a shorter DoS duration of 17 seconds. This shorter duration may indicate a quicker recovery mechanism in the MacBook, possibly due to more robust network capabilities inherent to its hardware or macOS software. The MacBook could have also potentially scanned more frequently during the attack due to less battery constraints, allowing it to quickly receive the legitimate management frames from the AP to maintain its connection.

Furthermore, both Raspberry Pi models (Zero W and 3 B+), were tested with 30 beacon frames transmitted at 0.2-second intervals. As evident in Figure 6.2, these devices exhibited a frame exchange pattern similar to that of the iPhone 15. The DoS durations were 10.3 seconds and 7.2 seconds, respectively. These results indicate that while these low-powered devices are more susceptible to network disruptions, their response times vary, likely due to differences in processing power.

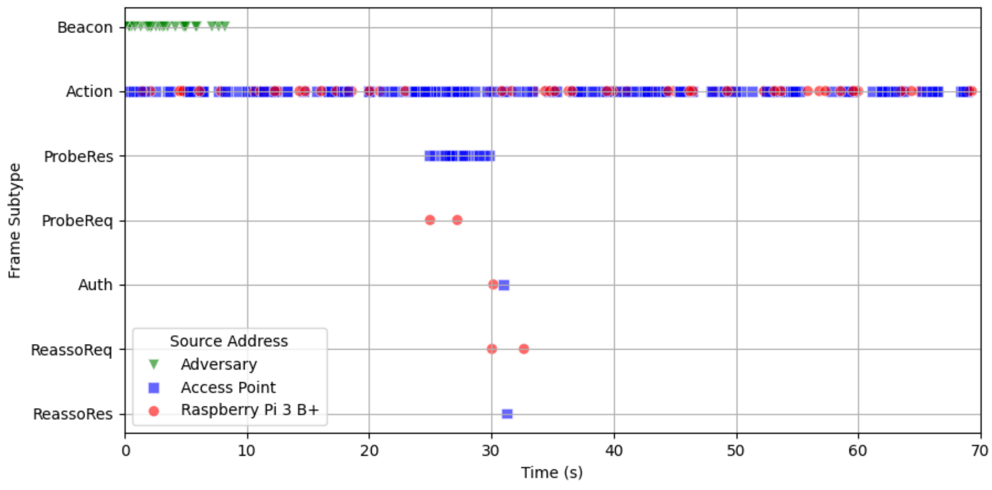


Figure 6.2: Communication timeline of Channel Switch Attack for Raspberry Pi 3 B+

In comparison to the other targets, the Lenovo Ideapad L340, demonstrated remarkable resilience, with the shortest DoS duration among all the tested devices at only 5.7 seconds. During this time, the Lenovo device transmitted a substantial number of various management frames, as can be seen in Figure 6.3. These included repeated probe requests, reassociation frames, and many others, indicating the device's aggressive attempts to maintain connectivity. Rather than experiencing a significant connectivity loss, it showed an increase in the time taken for ping responses during the attack. This suggests that while the device was able to maintain its connection to the network, it experienced only temporary latency issues, as a result of the injected beacon frames.

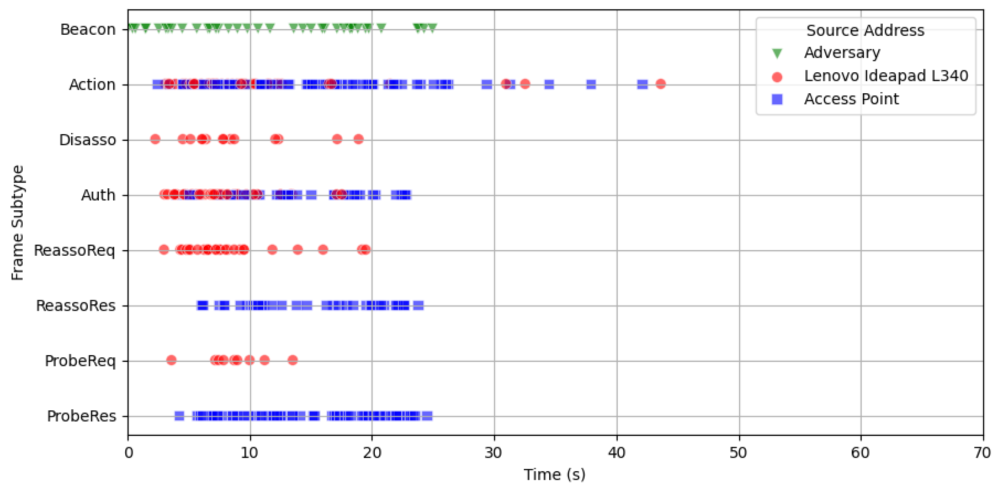


Figure 6.3: Communication timeline of Channel Switch Attack for Lenovo Ideapad L340

6.3.2 Quiet Attack

In the Quiet attack, fewer beacons were injected compared to the Channel Switch attack. After trying out several different transmission parameters, we found that the Quiet attack required less intensity, as introducing quiet periods with less amount of beacons was apparently sufficient to disrupt the target devices. The parameters are listed in Table 6.5.

Device	Beacon Count	Interval
iPhone 15	1	-
Google Pixel 8	3	0.4s
iPad Air 4th Gen	1	-
Lenovo Ideapad L340	40	0.5s
MacBook Pro 2019	1	-
Raspberry Pi Zero W	1	-
Raspberry Pi 3 B+	1	-

Table 6.5: Transmission parameters for Quiet Attack

Attack Execution

After including these transmission parameters, we finalized our configuration file, an example of which is detailed in Listing 14.

```ini
# tests/quiet_lenovo_2_4.ini

[MAC Header]
# 2.4 GHz
SA = 40:ae:30:c7:bf:e1
BSSID = 40:ae:30:c7:bf:e1
# 5 GHz
# SA = 40:ae:30:c7:bf:e0
# BSSID = 40:ae:30:c7:bf:e0

[Beacon Body]
Timestamp = 1284435
Interval = 100
Capability = privacy
# 2.4 GHz
SSID = Bowser
# 5 GHz
# SSID = Bowseros

[Beacon Ext]
Quiet = 1, 1, 65535, 0

# Transmission Parameters for Lenovo Ideapad L340
[Packet Send]
inter = 0.5
count = 40
```

Listing 14: Configuration file of Lenovo Ideapad L340 for Quiet Attack

We proceeded to inject the malformed beacon into the network using the CLI version of *Beaconfuzz*. The path to our configuration file was specified once again with the *-p* argument, and the execution mode was set to 'combined', enabling us to simultaneously monitor the exchanged frames, which were then outputted to a CSV file. To ensure that the attack has sufficient time to impact the network and targets, we set the timeout to 110 seconds. This duration was intentionally longer than the one used in the Channel Switch attack, as our objective was to induce a longer Denial-of-Service effect, allowing additional time to capture all the device responses:

```
# 2.4 GHz band
./beaconfuzz -i wlp3s0 -m combined -p tests/quiet_lenovo_2_4.ini -c 2 \
-t 110 -o output.csv

# 5 GHz band
./beaconfuzz -i wlp3s0 -m combined -p tests/quiet_lenovo_5.ini -c 36 \
-t 110 -o output.csv
```

Findings

Having already determined that the Channel Switch Attack was ineffective in the 5 GHz band, we proceeded to test the Quiet Attack in the same frequency range. However, similar to the previous findings, the Quiet Attack also failed to produce actual results in the 5 GHz band as well. Although we adjusted the transmission parameters and fields within the Quiet IE, such as reducing the Quiet Duration or adding an Offset, these adjustments also proved to be ineffective in the current test setup. Nevertheless, *Beaconfuzz* performed quite well in the 2.4 GHz band, successfully inducing many DoS effects across various devices, as presented in Table 6.6.

Device	DoS Duration (2.4 GHz)	DoS Duration (5 GHz)
iPhone 15	67.2s	0s
Google Pixel 8	30.8s	0s
iPad Air 4th Gen	67.8s	0s
Lenovo Ideapad L340	12.9s	0s
MacBook Pro 2019	17s	0s
Raspberry Pi Zero W	76.2s	-
Raspberry Pi 3 B+	68.3s	0s

Table 6.6: Denial-of-Service durations for Quiet Attack

The Quiet attack led to a significant overall increase in ICMP reply times. During the quiet period, many target devices, including the iPhone, iPad, Lenovo Ideapad, Raspberry Pi 3 B+, and Zero W, were observed actively exchanging action frames with the AP. The exchange of these action frames, specifically Block Ack Requests, indicates that both the target devices and the AP were actively managing the disruption, working to ensure that communication could resume smoothly once the quiet period concluded. However, despite these devices exhibiting similar responses in terms of frame exchanges, they experienced varying DoS durations For instance, the iPhone and iPad had DoS durations of approximately 67 seconds, matching the Quiet Duration announced in the injected beacon frames. The Lenovo Ideapad L340, on the other hand, experienced a much shorter disruption of 12.9 seconds. The Raspberry Pi Zero W, whose exchanged frames are depicted in Figure 6.4, and the Raspberry Pi 3 B+ were among the most affected, experiencing DoS durations of 76.2 seconds and 68.3 seconds, respectively.

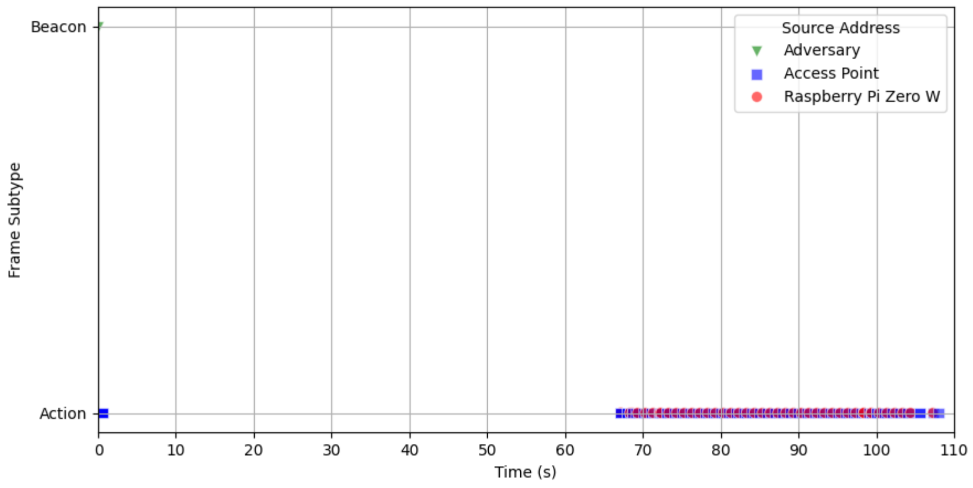


Figure 6.4: Communication timeline of Quiet Attack for Raspberry Zero W

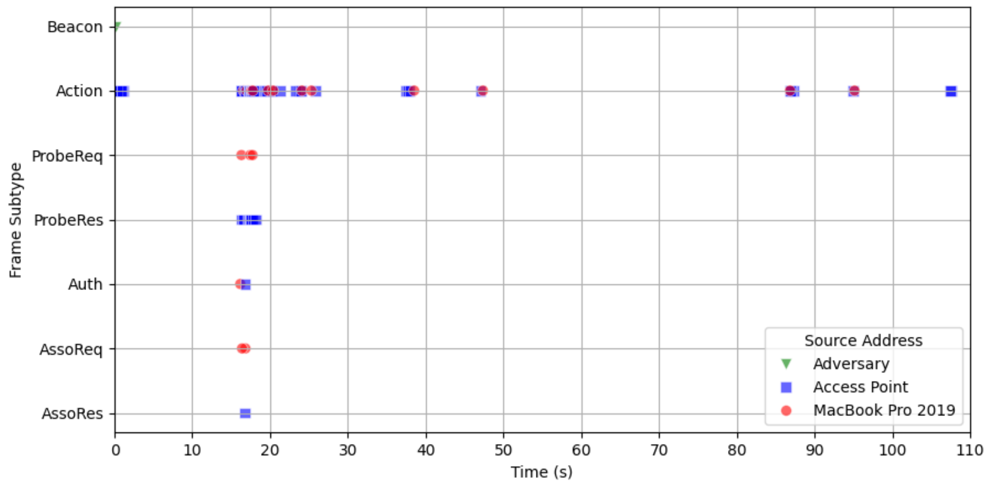


Figure 6.5: Communication timeline of Quiet Attack for Macbook Pro 2019

In contrast to the devices that primarily exchanged action frames, some devices also attempted to probe the network during the Quiet attack as part of their strategy. The MacBook and Google Pixel initiated probe requests for the target network's SSID during the quiet period and received probe responses from the AP, as shown in Figure 6.5 and 6.6. This behavior indicates that these devices were seeking to confirm the presence of the network. Additionally, unlike the Google Pixel, the MacBook also sent authentication and association frames after receiving the probe response, likely as a measure to fully re-establish its connection with the AP. This possibly ensures that any potential disruption caused by the attack did not result in a complete disassociation from the network. In terms of DoS effects, the Google Pixel experienced a DoS duration of 30.8 seconds, while the MacBook suffered a shorter disruption of 17 seconds, which reflects the varying resilience of respective devices.

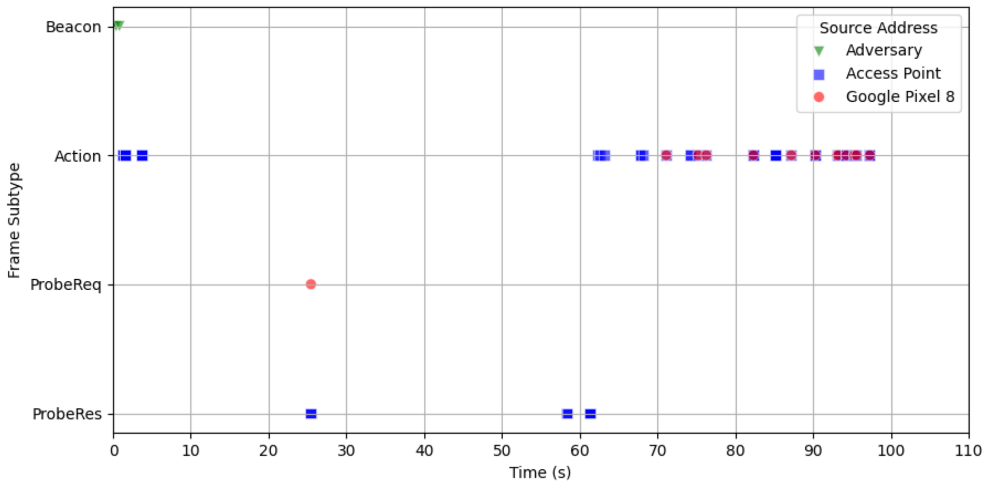


Figure 6.6: Communication timeline of Quiet Attack for Google Pixel 8

6.3.3 Beacon Flooding Attack

Attack Execution

To carry out the beacon flooding attack, we started by importing all the necessary functions and classes from the injection and monitoring modules of the *Beaconfuzz* package. The next step was to configure the wireless interface using the 'set_monitor' function to operate in monitor mode, which is crucial for allowing the interface to inject and capture frames on the specified channel. This function returns the name of the new interface and any processes that might interfere with the monitor mode, which are then terminated.

With the interface prepared, we had to craft the beacons. As discussed in Subsection 6.2.4, the 'spawn' method of the *Spawner* class was used to create an infinite beacon generator, with each generated beacon containing the BSSID of the AP and a randomized SSID. The parameters specified that 8 beacons with distinct SSIDs, each up to 8 characters long, were crafted.

To monitor the impact of the attack, a *Sniffer* thread was started using the *sniff_to_csv* function. This thread captured all frames exchanged on the target network during the attack and outputted them to a CSV file. The *Sniffer* operated asynchronously, allowing it to run in parallel with the beacon injection, and it was set to capture frames for 100 seconds, which also defines the total duration of the attack. Additionally, a subtype filter was applied to exclude the injected beacon frames from the capture, which enabled *Sniffer* to focus on other management frames. The beacons were then transmitted via the *send_async* function, which started a *Sender* daemon thread. The *inter=0* parameter ensured that the beacons were sent with no delay between transmissions, hence maximizing the flooding effect. The *verbose=True* option provided a detailed output during the transmission, which was useful for monitoring the progress of the attack in real-time.

Finally, after the transmission and capture processes are complete, the code performs cleanup operations. The *Sniffer* thread is joined, and the output file is closed to ensure all captured data is properly saved. The interface is then restored to its previous state by exiting monitor mode using the 'unset_monitor' function, which also restarts the processes that were initially terminated. The complete code snippet is provided in Listing 15.

```python
# tests/beacon_flooding.py
from beaconfuzz import *

# channel=36 for 5 GHz
new_iface, procs = set_monitor(
    iface="wlp3s0", mode="airmon-ng", naive=False, channel=2
)

# Craft beacons with randomized SSIDs with the known BSSID
beacon_generator = Spawner.spawn(ssid_len=8, ssid_num=8)

# Start the Sniffer thread that outputs captured frames to "output.csv"
sniffer, file = sniff_to_csv(
                    output="output.csv",
                    mode="async",
                    iface=new_iface,
                    timeout=100,
                    filter=subtype_filter(0, 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12,
                    13, 14, 15),
                )

# Start the Sender daemon
send_async(beacon_generator, iface=new_iface, inter=0, verbose=True)

# Perform the clean-up
sniffer.join()
```

```
file.close()
unset_monitor(iface=new_iface, mode="airmon-ng", procs=procs)
```

Listing 15: Complete code to execute Beacon Flooding Attack

Device	DoS Duration (2.4 GHz)	DoS Duration (5 GHz)
iPhone 15	124.8s	0s
Google Pixel 8	128.1s	0s
iPad Air 4th Gen	134.3s	0s
Lenovo Ideapad L340	103.9s	0s
MacBook Pro 2019	142.5s	0s
Raspberry Pi Zero W	123.2s	-
Raspberry Pi 3 B+	136.2s	0s

Table 6.7: Denial-of-Service durations for Beacon Flooding Attack

Findings

The beacon flooding attack had a pronounced impact on devices operating within the 2.4 GHz band, similar to the effects observed in the Channel Switch and Quiet Attacks. When subjected to the flood of beacon frames, all target devices were forcefully deauthenticated and disassociated from the network. In an attempt to recover their connections, the devices followed the typical reconnection sequence, as shown in Figure 6.7: they first sent probe requests, received probe responses from the AP, and then proceeded with authentication and association or reassociation request frames.

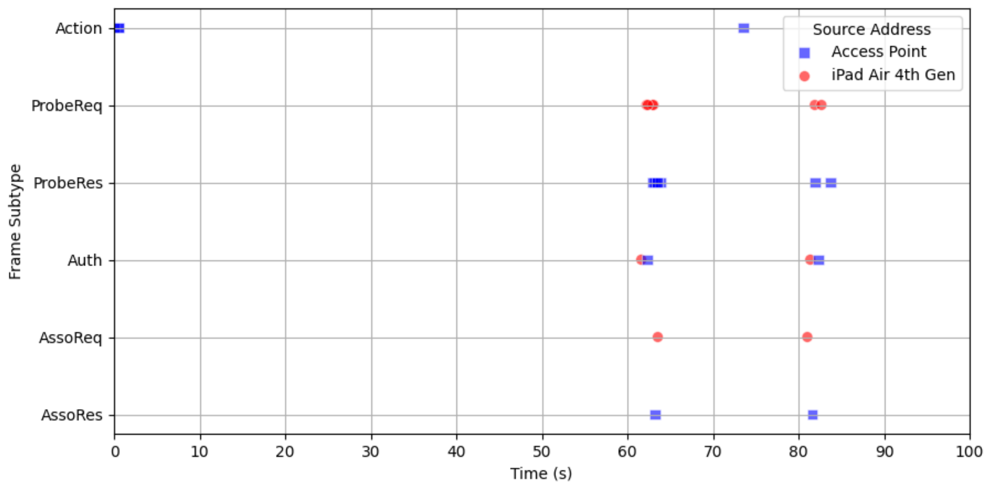


Figure 6.7: Communication timeline of Beacon Flooding Attack for iPad Air 4th Gen

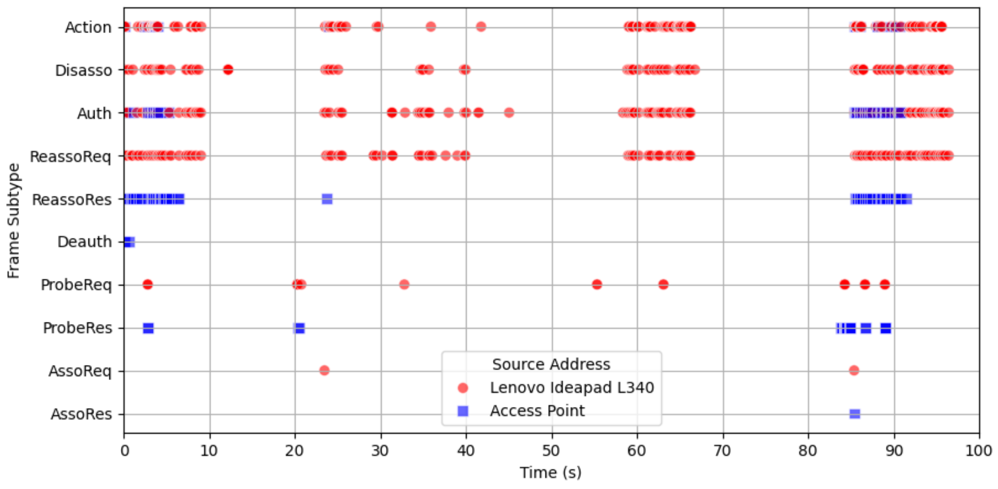


Figure 6.8: Communication timeline of Beacon Flooding Attack for Lenovo Ideapad L340

Similar to the Channel Switch Attack, the Lenovo device maintained its aggressive behavior during the flooding as well, whose frame exchange process is demonstrated in Figure 6.8. Despite these efforts, the overwhelming number of injected beacons continued to saturate the channel, rendering the devices' reconnection attempts unsuccessful. The continuous announcement of randomized SSIDs from the same BSSID effectively disrupted normal communication processes, preventing devices from successfully re-establishing their connections to the network even after the attack had ended, as evident from the longer DoS durations compared to the 100s timeout presented in Table 6.7.

Chapter 7

Conclusion

In this thesis project, we explored the vulnerabilities of WLANs defined by the IEEE 802.11 standard, focusing on beacon frame injection and its impact on different target devices. Our research began with an extensive review of the existing literature in the field of 802.11 fuzzing, where we examined various methodologies and evaluated fuzzing tools in terms of their capabilities. Building on this foundation, we developed a lightweight yet multi-functional fuzzing framework named *Beaconfuzz*, capable of crafting and injecting custom beacon frames while monitoring wireless traffic to evaluate device responses. We deployed the framework in two versions: a package version and a CLI application version. The package version offers the flexibility for integration with custom scripts. It allows for detailed configuration, making it ideal for more advanced users who need granular control over the fuzzing process. On the other hand, the CLI app version provides a more user-friendly interface, streamlining the execution of fuzzing tasks through predefined arguments.

Furthermore, we conducted a multitude of experiments to analyze how target devices respond to manipulated beacon frames under real-world conditions, which also allowed us to evaluate the performance of our framework. Our test setup included a diverse range of devices with varying operating systems and hardware configurations, all connected to a standard home router operating within both the 2.4 GHz and 5 GHz frequency bands. To conduct the testing, we employed different attack strategies, particularly focusing on fuzzing the Information Elements (IEs) embedded within the injected beacon frames. By targeting specific IEs, such as the Channel Switch Announcement and Quiet IEs, we aimed to exploit the 802.11 network operations that could lead to network disruptions or reveal unexpected device behavior.

The first attack we performed targeting the IEs was the Channel Switch Attack, where we forced devices to move to a different wireless channel, disrupting network communication. The second attack was the Quiet Attack, which announced silent periods, during which devices cease frame transmission, leading to increased communication latency and even disassociation from the network [2] [3] [20]. Apart from fuzzing the different IEs, we also implemented a Beacon Flooding Attack [23] as part of our testing strategy. In this attack, we injected an excessive amount of beacons with randomized SSIDs originating from the same BSSID MAC address, which enabled us to overwhelm the wireless channel with fake network advertisements. The flexibility of our framework allowed us to systematically create multiple test cases for each attack strategy. In the end, we successfully identified a range of Denial-of-Service vulnerabilities and captured the management frames exchanged between the target devices and the access point to analyze their distinct responses during the attacks. Our findings revealed that the 2.4 GHz band was particularly prone to these attacks, as devices consistently exhibited DoS effects in response to the injected beacon frames, despite the fact that the exploited mechanisms were primarily designed for the 5 GHz band.

Future Work

There are several opportunities for expanding the functionality and scope of our framework in future iterations. One key enhancement would be the ability to fuzz the "Radiotap" header, allowing for more granular control over low-level parameters such as transmission power, data rates, and antenna configuration. This would provide deeper insights into how devices respond to subtle variations in frame characteristics. Additionally, expanding the monitoring capabilities to include control and data frames, alongside the currently monitored management frames would offer a more comprehensive view of network behavior under attack, allowing for a full-spectrum analysis of wireless communication.

Further development could also enhance platform compatibility by adding functionality to put the wireless network interface card (WNIC) into monitor mode on Windows and macOS systems. This would broaden the framework's applicability, opening it up to various users. Lastly, introducing support for utilizing multiple WNICs would enable simultaneous monitoring of different channels. This feature would be important for accurately analyzing multi-channel environments and tracking how devices respond to attacks across varying channels. Implementing these improvements would significantly enhance the versatility of the framework, paving the way for more advanced research.

Bibliography

[1] T. Derham and N. Bhandaru, *Wi-Fi CERTIFIED WPA3™ December 2020 update brings new protections against active attacks: Operating Channel Validation and Beacon Protection*, https://www.wi-fi.org/beacon/thomas-derham-nehru-bhandaru/wi-fi-certified-wpa3-december-2020-update-brings-new.

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

[3] M. Vanhoef, P. Adhikari, and C. Pöpper, "Protecting Wi-Fi Beacons from Outsider Forgeries", WiSec '20, 155–160, 2020. DOI: 10.1145/3395351.3399442. [Online]. Available: https://doi.org/10.1145/3395351.3399442.

[4] IEEE Computer Society, "IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications", *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*, pp. 1–1076, 2007. DOI: 10.1109/IEEESTD.2007.373646.

[5] Hujian1, *What is Received Signal Strength Indication (RSSI)*, https://forum.huawei.com/enterprise/en/what-is-received-signal-strength-indication-rssi/thread/667281913221627904-667213855346012160, last accessed on 05.07.2024, Feb. 2023.

[6] F. Fund, *Frequency hopping spread spectrum*, https://witestlab.poly.edu/blog/frequency-hopping-spread-spectrum/, last accessed on 09.07.2024, Feb. 2017.

[7] A. Kumar and Y. Zhu, "Generalizing direct sequence spread-spectrum to mitigate autocorrelation-based attacks", in *2022 International Symposium on Networks, Computers and Communications (ISNCC)*, 2022, pp. 1–6. DOI: 10.1109/ISNCC55209.2022.9851804.

[8] C. Links, *The Evolution of Wi-Fi networks: from IEEE 802.11 to Wi-Fi 6E*, `https://www.wevolver.com/article/the-evolution-of-wi-fi-networks-from-ieee-80211-to-wi-fi-6e`, last accessed on 11.07.2024, May 2022.

[9] W.-F. Alliance, *Wi-Fi CERTIFIED 6*, `https://www.wi-fi.org/discover-wi-fi/wi-fi-certified-6`, last accessed on 11.07.2024.

[10] S. Boualleg and B. Haraoubia, "Influence of multipath radio propagation on wide-band channel transmission", in *International Multi-Conference on Systems, Signals & Devices*, 2012, pp. 1–6. DOI: `10.1109/SSD.2012.6197993`.

[11] Carle, G., Günther, S., Simon, M., Sosnowski, M., Lachnit, S., *Lecture: Grundlagen Rechnernetze und Verteilte Systeme (GRNVS) IN0010 – SoSe 2024, Lehrstuhl für Netzarchitekturen und Netzdienste, School for Computation, Information and Technology, Technische Universität München*, `https://www.net.in.tum.de/teaching/ss24/grnvs.html`, Apr. 2024.

[12] IEEE Computer Society, "IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications - Spectrum and Transmit Power Management Extensions in the 5 GHz Band in Europe", *IEEE Std 802.11h-2003 (Amendment to IEEE Std 802.11, 1999 Edn. (Reaff 2003))*, pp. 1–75, 2003. DOI: `10.1109/IEEESTD.2003.94393`.

[13] A. Mahmood, R. Exel, and T. Bigler, "On clock synchronization over wireless lan using timing advertisement mechanism and tsf timers", in *2014 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS)*, 2014, pp. 42–46. DOI: `10.1109/ISPCS.2014.6948529`.

[14] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey", *Cybersecurity*, vol. 1, no. 1, p. 6, 2018, ISSN: 2523-3246. DOI: `10.1186/s42400-018-0002-y`. [Online]. Available: `https://doi.org/10.1186/s42400-018-0002-y`.

[15] V. Kampourakis, E. Chatzoglou, G. Kambourakis, A. Dolmes, and C. Zaroliagis, "WPAxFuzz: Sniffing Out Vulnerabilities in Wi-Fi Implementations", *Cryptography*, vol. 6, no. 4, 2022, ISSN: 2410-387X. DOI: `10.3390/cryptography6040053`. [Online]. Available: `https://www.mdpi.com/2410-387X/6/4/53`.

[16] I. Siros, *Wireless Network Protocol Fuzzing*, `https://www.esat.kuleuven.be/cosic/blog/wireless-network-protocol-fuzzing/`, last accessed on 21.07.2024, Mar. 2022.

[17] L. Butti, *Wi-Fi Advanced Fuzzing*, Presented at the Black Hat Europe 2007, Amsterdam, Netherlands, `https://www.blackhat.com/presentations/bh-europe-07/Butti/Whitepaper/bh-eu-07-butti-handouts-apr19.pdf`, last accessed on 22.07.2024, Mar. 2007.

[18] H. Cao, L. Huang, S. Hu, S. Shi, and Y. Liu, "Owfuzz: Discovering Wi-Fi Flaws in Modern Devices through Over-The-Air Fuzzing", in *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '23, Guildford, United Kingdom: Association for Computing Machinery, 2023, 263–273, ISBN: 9781450398596. DOI: `10.1145/3558482.3590174`. [Online]. Available: `https://doi.org/10.1145/3558482.3590174`.

[19] P. Biondi and the Scapy community, *Scapy Documentation*, `https://scapy.readthedocs.io/en/latest/`, last accessed on 30.07.2024, 2008-2024.

[20] B. Könings, F. Schaub, F. Kargl, and S. Dietzel, "Channel Switch and Quiet attack: New DoS Attacks Exploiting the 802.11 Standard", in *2009 IEEE 34th Conference on Local Computer Networks*, 2009, pp. 14–21. DOI: `10.1109/LCN.2009.5355149`.

[21] A. Raj and D. S. Sankaran, "Battery Drain using WiFi Beacons", in *2023 11th International Symposium on Digital Forensics and Security (ISDFS)*, 2023, pp. 1–6. DOI: `10.1109/ISDFS58141.2023.10131769`.

[22] F. Ferreri, M. Bernaschi, and L. Valcamonici, "Access points vulnerabilities to DoS attacks in 802.11 networks", vol. 14, Apr. 2004, 634 –638 Vol.1, ISBN: 0-7803-8344-3. DOI: `10.1109/WCNC.2004.1311620`.

[23] H. Lee and S. Bahk, "Beacon Flooding Problem in High-Density WLANs", in *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, 2017, pp. 595–597. DOI: `10.1109/ICTC.2017.8191048`.

[24] L. Li and N. Xui, *802.11 Smart Fuzzing*, Presented at CyberPeace@AD-LAB, `https://powerofcommunity.net/poc2018/lidong.pdf`, last accessed on 28.07.2024, 2018.

[25] *Aircrack-ng*, `https://www.aircrack-ng.org/`, last accessed on 30.07.2024, 2009-2023.

[26] J. Carpenter, *iw Documentation*, `https://wireless.wiki.kernel.org/en/users/documentation/iw`, last accessed on 30.07.2024, May 2024.