Radboud University Nijmegen

Faculty of Science, Computer Science

Fuzzing the data link layer of ZigBee

IEEE 802.15.4

Bachelor Thesis

Author:
Calin Iaru

Supervisors
dr. Erik Poll
Seyed Behnam

Second Examinator
dr. Pol van Aubel

2025

Summary

This thesis tests for vulnerabilities in ZigBee's data link layer by applying fuzzing techniques. ZigBee, a widely used communication protocol for Internet of Things (IoT) devices, is built on the IEEE 802.15.4 standard and is valued for its low power consumption, scalability, and robust mesh networking capabilities. These features have made ZigBee a popular choice for connecting a wide range of smart devices, from simple household appliances to critical safety equipment. However, its increasing adoption introduces security risks that require thorough investigation to safeguard user privacy and safety.

Part of this thesis explores ways to set up a Software-Defined Radio (SDR) to fuzz ZigBee traffic and a CC2531 dongle for sniffing the network. The methodology involved capturing, manipulating, and injecting malformed packets into the protocol to identify weaknesses. This allowed a detailed exploration of potential exploits in everyday devices, such as smart light bulbs, and more critical devices, including smoke and gas detectors.

The ability to manipulate non-critical devices like light bulbs demonstrates the potential for inconvenience or minor disruptions. Although no actual vulnerabilities were found in the tested device (a Philips smart bulb), the experiments demonstrated that low-level fuzzing of ZigBee networks is technically feasible. They also showed that this specific ZigBee device can effectively handle malformed or unexpected packets without crashing or behaving incorrectly.

Contents

1 Introduction

Wireless communication technologies play a critical role in the modern era of Internet of Things (IoT) devices, enabling seamless interaction and connectivity across a wide range of applications. The IoT market is witnessing exponential growth, with projections estimating a compound annual growth rate (CAGR) between 11.4% and 24.3%. Grand View Research forecasts a CAGR of 11.4% through 2030 [1], while Fortune Business Insights anticipates a more aggressive CAGR of 24.3% by 2032. Such rapid adoption underscores the growing reliance on wireless technologies to power smart homes, industrial automation, healthcare, and more[2].

Among these technologies, ZigBee, a protocol built on top of the IEEE 802.15.4 standard, has emerged as a popular choice due to its low-cost, low-power design and robust networking capabilities. Despite its advantages, ZigBee's increasing adoption in IoT devices raises concerns about its security, particularly at the data link layer where foundational communication processes occur. The IEEE 802.15.4 standard serves as the foundation for low-power wireless networks, such as ZigBee, and has become a critical component in Internet of Things (IoT) applications.

This thesis explores the potential vulnerabilities within ZigBee networks by applying *fuzzing* techniques to the *data-link-layer*, composed of the PHY and MAC layers. Fuzzing, a powerful method for uncovering security weaknesses, involves sending malformed or unexpected inputs to a system to evaluate its behavior. By leveraging Software-Defined Radio (SDR) tools and platforms like ZigBee2MQTT, this research aims to systematically analyze ZigBee communications to identify potential weaknesses that attackers could exploit. This thesis tries to answer the following research questions:

Main Research Question

RQ: How can fuzzing techniques be applied to IEEE 802.15.4, the data link layer used by ZigBee, to identify vulnerabilities?

Sub-Questions

RQ1 How can active and passive man-in-the-middle techniques extend the fuzzing coverage of the IEEE 802.15.4 data-link layer in ZigBee networks?

RQ2 How does the presence or absence of a network coordinator affect the success rate of replay-based attacks and other fuzzing payloads?

RQ3 How do ZigBee's network and application-layer security measures influence or constrain the results of data-link-layer fuzzing?

Section 2 surveys earlier ZigBee security literature and sources of inspiration. Section 3 supplies all protocol background, topology, stack layers, frame fields, and security that later chapters can assume. Section 4 documents how a clean Philips-Hue/CC2652P test network was built so that later attacks have a baseline. Section 5.1 and Section 5 detail the SDR hardware, firmware tweaks, and flow-graph modifications that enable raw frame injection. Section 6 describes the setup, replay attacks, and multiple fuzzing configurations across MAC and PHY fields and records their outcomes. Section 7 lists the limitations encountered and logical next steps. Finally, Section 8 ties the experimental results back to the research questions and restates the main contributions and key takeaways.

2 Related work

Several studies have explored different aspects of the security of ZigBee and related IEEE 802.15.4 protocols, including attack strategies, test-bed development, fuzzing techniques, and automated analysis methods.

Jeroen Lammers [3] presents a structured literature survey on ZigBee security, analyzing publications from 2004 to early 2023. His work maps reported vulnerabilities across ZigBee protocol revisions and classify them according to the CIA triad, highlighting key specification changes such as the shift from unsecured to trust-center-based rejoin procedures in ZigBee 3.0. While IEEE 802.15.4 is acknowledged as the foundation of ZigBee, his analysis remains focused on the protocol stack as a whole rather than on the IEEE 802.15.4 MAC/PHY layers in isolation. Building on his findings, this paper aims to identify vulnerabilities specifically in the MAC and PHY layers through fuzzing experimentation.

Sokull et al.[4] conducted a detailed study of attacks on the IEEE 802.15.4 MAC (see Section 3.3 for the protocol stack) layer, highlighting common vulnerabilities such as radio jamming, link-layer jamming, back-off manipulation, replay-protection attacks, and ACK attacks. These attacks exploit protocol features to disrupt communication or gain an unfair advantage. Additionally, the authors proposed two new attack scenarios: the PANId conflict attack and the GTS attack. While these studies relied on simulated environments using ns2.31, their findings provide a solid foundation for further investigation using real-world setups. In contrast to Sokullu's simulation-based approach, our work applies fuzzing techniques in a real-world testbed using actual ZigBee hardware. Rather than reproducing known MAC-layer attacks such as jamming or back-off manipulation, we focus on generating malformed or unexpected frames to uncover new, previously unexplored vulnerabilities.

Using GNU Radio, Bloesel et al.[5] developed a modular framework for the PHY and MAC layers of IEEE 802.15.4. Their implementation enables sending and receiving valid IEEE 802.15.4 frames over the air, but it does not include higher-layer protocols such as ZigBee (e.g., network, application, or security layers). The testbed employs modular components to implement essential features such as frame parsing, encoding/decoding, CRC validation, and channel access mechanisms. This setup allows for real-time manipulation and analysis of protocol behavior, making it an invaluable tool for testing compliance and evaluating the performance of IEEE 802.15.4 implementations. While the testbed's modularity and flexibility provide significant advantages for protocol analysis, it focuses primarily on ensuring compliance rather than systematically uncovering vulnerabilities. Bloessl et al. provided helpful insights into IEEE 802.15.4 behavior by implementing the PHY and MAC layers using GNU Radio. However, we require low-level control over frame construction and transmission to test malformed packets on real ZigBee devices, particularly those with intentionally corrupted fields in the IEEE 802.15.4 data link layer. This is not possible with the existing implementation provided by Bloessl et al. Hence, we adapt or bypass parts of the PHY/MAC stack to allow precise manipulation of lower-layer protocol fields. This is important for fuzzing because we want to send modified frames over the air and see how real ZigBee devices react. In our experiments, we adjust their setup to send raw frames directly.

Olaf van der Kruk applies active automata learning and fuzzing to the upper layer of the ZigBee stack, namely the Network and Application layers (the green and red regions in Figure 3). By sending well-formed and malformed Application Layer (APL) messages into real devices, he discovers logical errors, protocol mismatches, and security weaknesses in processing commands. His methodology highlights flaws in attribute reads/writes, on/off commands, and other high-level interactions rather than in the raw radio transmission itself. Drawing inspiration from his approach, this paper targets the lower layers, specifically the IEEE 802.15.4 MAC and PHY layers (the purple region in Figure 3).

Shang, Garbelini, and Chattopadhyay [6] introduce U-FUZZ, a fuzzing framework designed to test IoT protocols in a stateful and black-box manner. Instead of relying on protocol documentation, U-FUZZ observes real network traffic (from Wireshark captures) and uses it to determine how a device responds to different messages. Based on this, it builds a simple model that shows the different states the device goes through and how it reacts. This model helps guide the fuzzing by choosing the right moments to send modified or unexpected messages. U-FUZZ works across different protocols and was tested on real ZigBee, CoAP, and 5G NR devices. The authors discovered 11 previously unknown vulnerabilities, showing that this method can effectively find issues that depend on how a device reacts over time. While our work focuses on low-level fuzzing of the IEEE 802.15.4 MAC and PHY layers using software-defined radios, U-FUZZ shows how using traffic-based behavior models can help improve fuzzing across many types of devices and protocols.

3 ZigBee

This chapter conducts an in-depth exploration of the ZigBee protocol built on the IEEE 802.15.4 standard for low-power, wireless networks. Section 3.1 presents the advantages of ZigBee such as reliability and channel access, whereas Section 3.2 describes the structure of a ZigBee network, focusing on the terminology and topology. Sections 3.3 and 3.4 present the stack with a look at the data link layer. Section 3.5 presents the security aspects of a ZigBee network. Lastly, Section 3.6 introduces *ZigBee2MQTT*, an open-source project.

ZigBee is a wireless communication standard that addresses the need for very low-cost, low-power communication implementation for low-power devices with low data rates that use wireless communications. It is used by many Internet of Things(IoT) devices and is based on the IEEE 802.15.4 standard, ensuring reliable communication between devices. One of ZigBee's key features is that it can transmit data over long distances by passing data through a mesh network of intermediate devices. They have better penetration through walls than other wireless protocols like Wi-Fi or Bluetooth, but at the cost of bandwidth.

3.1 Key Advantages of ZigBee

ZigBee networks are designed to offer reliable and energy-efficient communication, making them a preferred choice for low-power, short-range wireless applications. Their reliability stems from both robust transmission mechanisms and efficient channel access strategies that help reduce interference, manage network traffic, and ensure seamless connectivity.

In addition to these reliability-enhancing features, ZigBee's channel access mechanisms, including beacon transmission scheduling, play a crucial role in efficient power management and collision reduction.

The following subsections explore how ZigBee ensures reliability through its networking architecture and channel access strategies, detailing the mechanisms that support robust communication even in challenging environments.

3.1.1 Reliability

Different aspects of this type of network make it reliable, as a starting point, we can consider the IEEE standard that it is using, which is a very modern and robust radio technology. IEEE 802.15.4 The IEEE 802.15.4 standard is designed to provide the essential lower network layers for wireless personal area networks (WPANs), emphasizing low-cost, low-speed communication between devices. Its core framework envisions a 10-meter communication range with line-of-sight connectivity and a data transfer rate of 250 kbit/s [7]. Further analysis of the IEEE 802.15.4 frame format is discussed in Section 3.4.

ZigBee uses a Carrier Sense Multiple Access Collision Avoidance(CSMA-CA) to increase reliability. By using this network protocol, ZigBee listens to the channel, waits until it is clear, and then starts to transmit, preventing corruption of data as radios are prevented from taking over one another [8, Section 1.1.1].

What makes ZigBee even more reliable is its mesh network architecture, where data from a node can reach any other node in the network as long as there are enough radios in between to pass the message. This makes it perfect for transmitting even when there are different barriers between points, such as brick walls. Let's assume that the best route for node 1 to communicate with node three is through node two [Fig.2.2.1], but if we add a barrier, like a brick wall between node one and node two [Fig.2.2.2]. Due to the mesh network, this barrier doesn't affect the performance of the network as it gets rerouted through nodes 4 and 5.

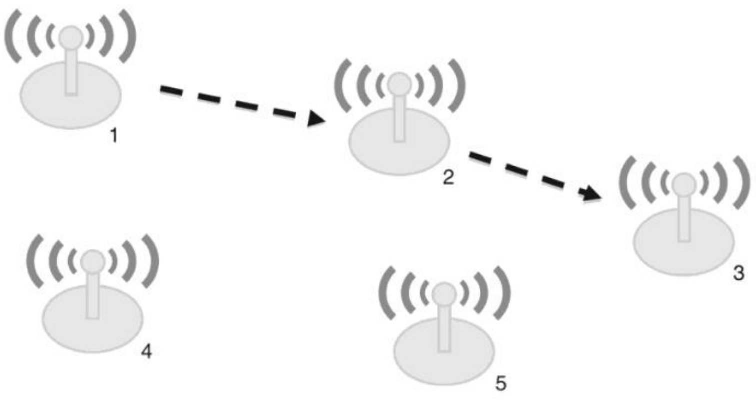


Figure 1: ZigBee mesh network with no barrier, Section 1.1.1 [8]

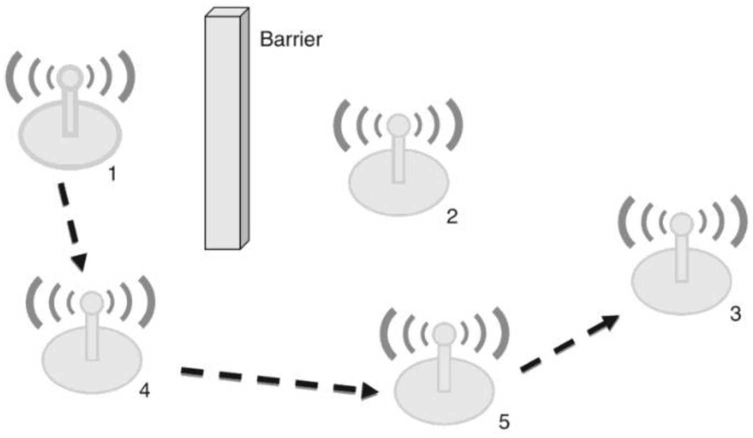


Figure 2: ZigBee mesh network with barrier, Section 1.1.1 [8]

3.1.2 Channel access

ZigBee networks save energy and collisions using *beacon transmission scheduling*. This process allows devices to stay synchronized and communicate effectively by assigning each device a specific time slot. The coordinator sets two parameters for each device: beacon order (BO), which defines how often beacons are sent, and superframe order (SO), which sets the active communication time within each beacon interval. Furthermore, to prevent collision each device is assigned a unique time slot [9].

When a new device joins the network, it scans for existing transmissions to identify available time slots, ensuring its chosen slot does not overlap with those of its neighbors or their parents. If a slot is available, the device includes the offset in its beacon to maintain synchronization, otherwise, it operates as an end device without sending beacons.

3.2 Structure of a ZigBee network

ZigBee networks are organized based on specific device roles and network topologies, which define how devices interact and communicate. Understanding ZigBee's functionality requires first defining the key network components that establish its structure. Additionally, the choice of network topology determines how devices connect and transmit data. The following subsections explore both aspects in detail.

3.2.1 ZigBee terminology

Before going deeper into understanding the ZigBee protocol, some terms need to be defined:

- **Node**: fundamental unit in a ZigBee network and can have multiple roles;
- **Coordinator**: represents the core of the network, acting as the centralized authority and providing a link between the ZigBee network and an IP-based network. Typically, these coordinators come with a LAN port and are powered by an external power adapter;
- **Router**: operates as an intermediary, forwarding data to other devices within the network. These ZigBee devices are typically powered by mains electricity, ensuring they are continuously available on the network;
- **End device**: contains just enough functionality to talk to the parent node and it cannot relay data from other devices. This relationship allows the node to be asleep for a significant amount of time, giving a long battery life.

3.2.2 Topology

ZigBee supports three different network topologies: Star Topology, Tree Topology, and Mesh Network:

- **Star Topology**: Also known as point-to-multipoint, this is the simplest structure. Each end device has a dedicated link only to a central controller called the coordinator.
- **Tree Topology**: This builds on the star topology by retaining the coordinator while adding routers, which help extend communication within the network.
- **Mesh Network**: Also known as peer-to-peer, this topology allows devices to communicate directly with any other device within range, providing redundancy and simplifying device additions or removals.

The flexibility and scalability of the mesh topology make it particularly suitable for larger networks. Additionally, its self-healing mechanisms allow the network to dynamically adapt, maintaining connectivity even when certain devices fail or are added.

3.3 Protocol stack overview

ZigBee, like many network protocols, uses a layered architecture to separate different functions into layers, which can be observed in Figure 3. It is built on the Physical (PHY) layer and Medium Access Control (MAC) sub-layer, both defined by the IEEE 802.15.4 standard, which is discussed in more depth in Section 3.4. These layers handle basic network tasks such as addressing and message transmission(ZigBee purple stack layers [10]).

The ZigBee specification introduces additional layers: the Network (NWK) layer and the Application (APL) layer framework. The Network layer manages the structure, routing, and security of the network, while the Application layer framework includes the Application Support sub-layer (APS), ZigBee Device Objects (ZDO), and user-defined applications that provide specific functionality to devices (ZigBee green and red stack layers [10]).

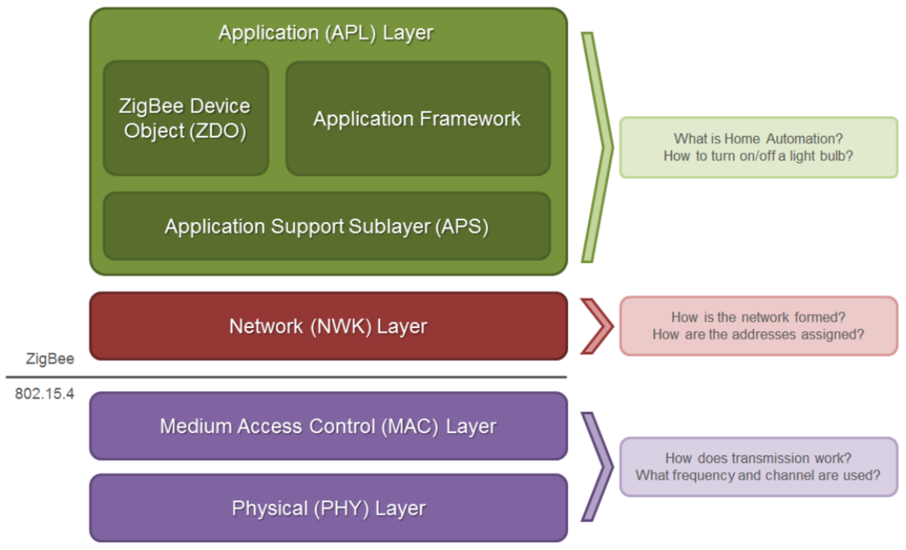

Application (APL) Layer
ZigBee Device Object (ZDO)
Application Framework
Application Support Sublayer (APS)
What is Home Automation?
How to turn on/off a light bulb?
Network (NWK) Layer
How is the network formed?
How are the addresses assigned?
ZigBee
802.15.4
Medium Access Control (MAC) Layer
Physical (PHY) Layer
How does transmission work?
What frequency and channel are used?

Figure 3: ZigBee architecture [10]

3.4 ZigBee on top of IEEE 802.15.4

IEEE 802.15.4 is a technical standard that defines the physical (PHY) and medium access control (MAC) layers for low-rate wireless personal area networks (LR-WPANs). It is designed to support low-power, low-data-rate communication with simple and cost-effective implementations, making it ideal for Internet of Things (IoT) applications. The standard operates in multiple frequency bands, with 2.4 GHz being the most widely used due to its global availability and robust data transmission characteristics[11].

The IEEE 802.15.4(the purple area in Figure 3) standard defines a structured frame format that facilitates data exchange between network nodes. The frame is composed of three main parts:

- MAC Header (MHR): Contains control information, such as the Frame Control Field (FCF), addressing details, and sequence numbers.
- MAC Payload: Holds the actual data being transmitted, including the network layer payload when used with higher-layer protocols like ZigBee.
- MAC Footer (MFR): Includes the Frame Check Sequence (FCS) used for error detection.

Figure 4 illustrates the frame structure, showcasing how IEEE 802.15.4 enables reliable data transmission while maintaining a lightweight overhead suitable for low-power networks.

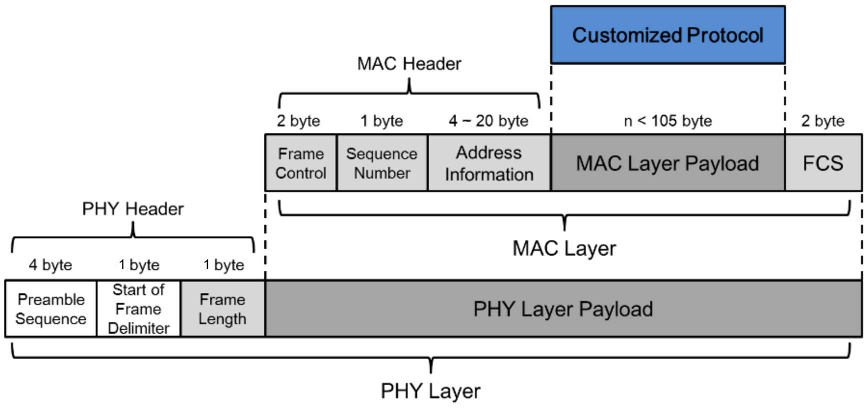


Figure 4: IEEE 802.15.4 Frame Format modified version of [12]

To manage channel access efficiently, IEEE 802.15.4 employs Carrier Sense Multiple Access with Collision Avoidance (CSMA-CA). Before transmitting, a device first listens to the channel to determine whether it is idle. If the channel is busy, the device waits for a random backoff period before attempting to retransmit. This mechanism reduces packet collisions and optimizes network efficiency [11]. Additionally, for time-sensitive applications, IEEE 802.15.4 supports Guaranteed Time Slots (GTS) in beacon-enabled networks, ensuring reliable delivery for delay-sensitive data.

3.4.1 Frame Control Field

The Frame Control Field is a 16-bit header at the beginning of every IEEE 802.15.4 frame. It tells the receiver how to interpret the rest of the frame by specifying key properties like:

- **Frame Type (bits 0–2):** Specifies the type of MAC frame being sent.

Bits	Dec	Type	Description
000	0	Beacon	Used for network discovery/sync
001	1	Data	Carries application/network data
010	2	ACK	Acknowledges successful reception
011	3	MAC Command	Used for control (e.g. association)
100–111	4–7	Reserved	Invalid/undefined

- **Security Enabled (bit 3):** Indicates if security features such as encryption and MIC are enabled. If set, additional security headers must follow;
- **Frame Pending (bit 4):** Informs the receiver that more frames are waiting;
- **Acknowledgment Request (bit 5):** Requests an acknowledgment from the receiver;
- **PAN ID Compression (bit 6):** If set, assumes source and destination share the same PAN ID and only include one field. Otherwise, both PAN IDs are present;
- **Reserved (bit 7):** Reserved for future use; should be set to 0;
- **Sequence Number Suppression (bit 8, optional):** Omits the sequence number field when set;

- **Information Elements Present (bit 9, optional):** Indicates if additional information elements are included in the frame;
- **Frame Version (bits 10–11):** Defines the protocol version;
  - 00 – IEEE 802.15.4-2003
  - 01 – IEEE 802.15.4-2006
  - 10/11 – Reserved
- **Destination Address Mode (bits 12–13)** and **Source Address Mode (bits 14–15)**: Together, these determine the presence and length of address fields.

Bits	Mode	Bytes Added
00	None	0
01	Reserved	Invalid
10	Short (16-bit)	2
11	Extended (64-bit)	8

The visualization of the Frame Control can be observed in Figure 5, where each square represents one bit.

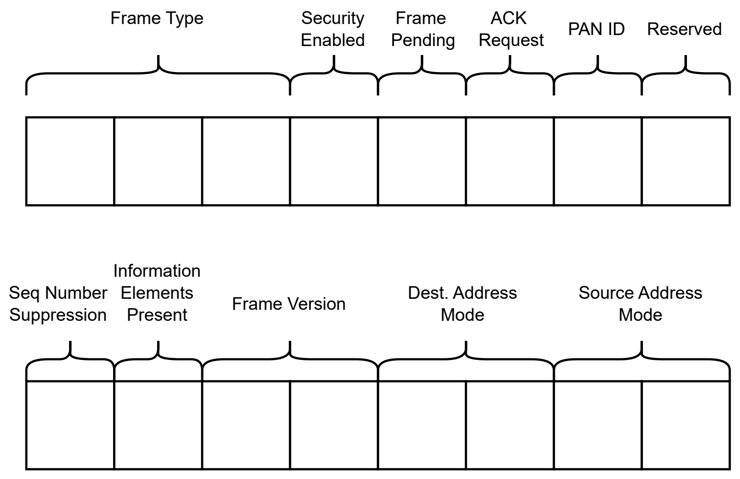


Figure 5: Frame Control Field bits

3.5 Security

The ZigBee security architecture builds upon the foundational security mechanisms of IEEE 802.15.4, ZigBee adds advanced cryptographic algorithms and key management processes. While IEEE 802.15.4 provides basic security features such as AES-128 encryption and frame counters (in the MAC layer payload in Figure 4) for replay protection[11], ZigBee enhances security with AES-CCM for authenticated encryption and AES-MMO for hashing, ensuring robust protection for higher-layer communications (Chapter 4 from [9]).

To verify message authenticity and integrity, ZigBee uses a Message Integrity Code (MIC), which is produced as part of AES-CCM and can be 4, 8, or 16 bytes (32, 64, or

128 bits) long, depending on the configured security level. A valid MIC confirms that the content of a message has not been altered in transit.

To protect against replay attacks, ZigBee maintains dedicated frame counters at multiple protocol layers, they are checked from bottom to the top of the ZigBee stack A simplified version of the stack that illustrates only the counter is in Figure 6. The order is:

1. MAC Sequence Number – used at the MAC layer (8-bit duplicate-frame filter)
2. NWK Security Frame Counter – used at the network layer (32-bit counter)
3. APS Security Frame Counter – used at the application support layer (32-bit counter)

Each of these counters must increment with every transmitted frame. Receivers validate that each incoming frame has a higher counter value than previously observed; otherwise, the frame is discarded.

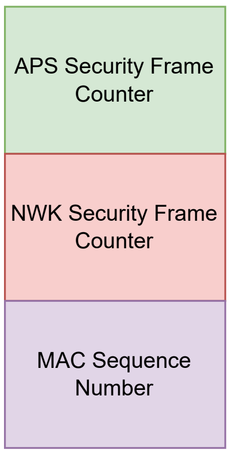


Figure 6: ZigBee stack illustrating only the frame counters

One of the most important aspects of ZigBee's security is the *Trust Center*, which manages device authentication, key distribution, and security models. ZigBee supports two security models:

- Centralized model: it has only one Trust Center, which in most of cases is the coordinator
- Distributed model: it has multiple Trust Centers, and any router can act as one(Chapter 4 from [9]).

A 128-bit common network key is used to secure broadcast and network-wide communications, providing hop-by-hop message authentication and encryption. This key is distributed securely to devices upon joining the network. Devices that can correctly send frames secured with the active network key are considered legitimate by their neighbors.

For end-to-end security, unicast communications between two devices are protected by a 128-bit application link key, which is either manually configured or securely distributed by the Trust Center. Each device also maintains a 128-bit Trust Center link key for securing application-layer messages exchanged with the Trust Center, which may be unique for each device or shared globally(Chapter 4 from [9]).

The ZigBee network is vulnerable to device impersonation and packet injection attacks, especially when an adversary compromises the network key or exploits weaknesses in the Trust Center's authentication process; to combat this, the *PhyAuth* mechanism can be added optionally.

3.5.1 PhyAuth

The PhyAuth mechanism further strengthens ZigBee's initial security presented in Section 3.5 by adding a physical-layer authentication step during device joining, ensuring that only legitimate devices can gain access to the network by combining physical-layer identifiers with cryptographic verification. PhyAuth supports both security models, thus making the ZigBee network more resilient to attack targeting the device authentication process[13].

ZigBee security provides flexible encryption and authentication options for both the network (NWK) and application (APS) layers. These include encryption-only, authentication-only, or both, with configurable Message Integrity Code (MIC) lengths of 32, 64, or 128 bits(Chapter 4 from [9]). This layered security, combined with enhancements like PhyAuth, ensures both hop-by-hop and end-to-end protection, reinforcing ZigBee's ability to operate securely in diverse environments.

3.6 ZigBee2MQTT

ZigBee2MQTT is an open-source software project designed to connect ZigBee devices using MQTT (Message Queuing Telemetry Transport), a lightweight and efficient messaging protocol widely used in Internet of Things (IoT) applications. By using MQTT, ZigBee2MQTT eliminates the need for devices to come from the same manufacturer, enabling compatibility between ZigBee devices from different vendors. For example, the ZigBee coordinator and end devices can be from different manufacturers yet still communicate seamlessly within the same network, providing greater flexibility and choice in building IoT systems [14].

In the context of the ZigBee protocol stack illustrated in Figure 7, ZigBee2MQTT primarily operates at and above the Network (NWK) and Application (APL) layers. It acts as the coordinator, interpreting and managing messages that flow through the NWK layer and the APL layer. Once these messages reach ZigBee2MQTT, the software translates them into the MQTT format used for messaging in broader IoT ecosystems.

One of the key strengths of ZigBee2MQTT lies in its extensive device compatibility. With support for over 4000 ZigBee devices from manufacturers around the world, it provides versatility and scalability for IoT networks. These devices range from smart lights and switches to sensors and smart plugs or switches. The global compatibility ensures that users are not restricted to a single vendor's ecosystem, allowing them to mix and match devices based on specific needs and preferences. Using the list of supported devices provided on the ZigBee2MQTT's website devices, a Philips bulb was selected to conduct the experiments in Section 4 and Section 6.

One of the key conveniences of using ZigBee2MQTT is its optional built-in front-end web interface. Once enabled (usually by adjusting settings in the ZigBee2MQTT configuration file), access to a browser-based dashboard is granted, where different information about the Philips bulb such as series, and the name can be found, also it provides a control panel for the devices. Furthermore, it also provides a section where logs can be seen, providing immediate visual cues about whether a device is connected or requires attention.

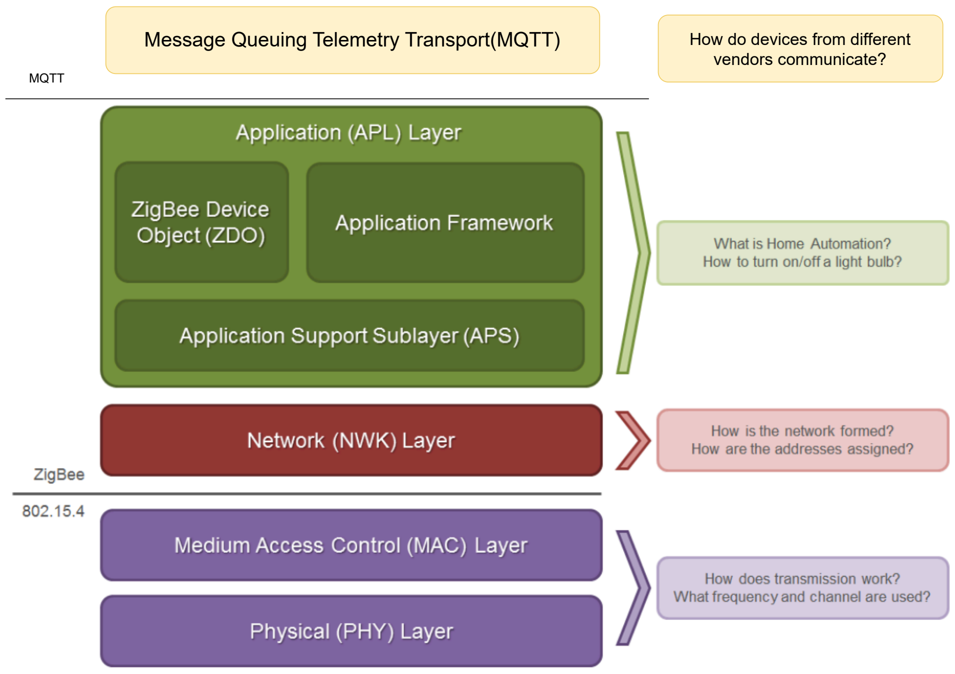

Message Queuing Telemetry Transport(MQTT)
How do devices from different vendors communicate?
MQTT
Application (APL) Layer
ZigBee Device Object (ZDO)
Application Framework
Application Support Sublayer (APS)
What is Home Automation? How to turn on/off a light bulb?
Network (NWK) Layer
How is the network formed? How are the addresses assigned?
ZigBee
802.15.4
Medium Access Control (MAC) Layer
Physical (PHY) Layer
How does transmission work? What frequency and channel are used?

Figure 7: ZigBee with MQTT

4 Experiments for Setting-Up a ZigBee Network

This chapter presents two approaches for establishing a ZigBee network, each highlighting different methods and practical considerations. In Section 4.1, a Philips Hue Bridge is used as the network coordinator. This scenario reflects a standard consumer environment where end users rely on commercially available hubs to manage ZigBee devices. In section 3.6, ZigBee2MQTT is presented and further used in the experiment. In Section 4.2, a CC2652P USB dongle directly connected to a laptop replaces the Hue Bridge, offering more flexibility and control for research purposes such as network monitoring, packet capture, and fuzzing.

4.1 Set-up experiment using a Philips Hue Bridge

The goal of this experiment is to establish a ZigBee network. It uses a Philips Hue Bridge as the network coordinator, which is designed to set up and manage legitimate ZigBee networks. This makes it a suitable choice to observe network communication in a standard ZigBee environment.

To establish this setup, the Hue Bridge (acting as the ZigBee coordinator) needed to connect to a router providing a Wi-Fi network, which includes a mobile device running the Philips Hue app, required to configure and manage the ZigBee network. However, due to the lack of Ethernet ports in the router, a laptop was used as a network gateway to bridge the Hue Bridge Ethernet connection to the Wi-Fi network.

This indirect setup introduced complexities, as configuring the laptop as a gateway to simulate a standard router setup required complicated IP forwarding and network bridging settings. Due to these challenges, an alternative approach was explored, involving the use of a USB dongle as a ZigBee coordinator. This device allowed for more flexible configuration and packet capture capabilities of the ZigBee network. The setup and configuration are presented in Figure 8, where dotted arrows mean over-the-air communication, and normal arrows direct connection between laptop and device. The real setup is in Appendix A.2, Figure 23.

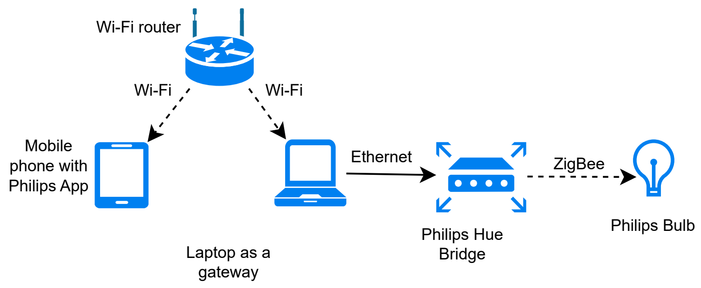


Figure 8: Experiment with Hue Bridge

4.2 Set-up experiment with CC2652P Dongle as ZigBee Coordinator to control a Philips bulb

The goal of this experiment is to leverage the benefits of CC2652P to set up a ZigBee network. Using a CC2652P dongle as a coordinator offers a more compact solution for fuzzing a ZigBee network. By plugging the dongle directly into the laptop, the setup becomes easier to control and monitor. It also makes the set-up easier to manage due to fewer cables and easier to compute with. In this experiment, a ZigBee network is established, where the CC2652P dongle is the coordinator and a Philips Hue white and color ambiance E26/E27/E14 (model 9290012573A) is the end device(it is also referred as the Philips bulb). To configure the CC2652P dongle as a ZigBee coordinator, the `Getting Started` guide on the ZigBee2MQTT website provides detailed instructions on initial setup [14].

To initially set up a device, it should be as close as possible to the coordinator. Next, the ZigBee network should be initialized, ensuring that the necessary services are running in a containerized environment, information can be found in the Installation section on the ZigBee2MQTT website[14]. Additionally, the flag to allow devices to join the network should be enabled on the user interface of the ZigBee2MQTT. When a new device is not recognized, there are two possibilities to make it recognizable:

1. Manual Factory Reset: physically reset the device to its factory settings. This process varies by manufacturer but typically involves a sequence of actions, such as turning the device on and off multiple times. Resetting to factory settings allows the device to rejoin the network as if it were new.

2. Touchlink Factory Reset: ZigBee's Touchlink commissioning feature allows a device to be reset wirelessly without physical access by sending a broadcast message from the coordinator and all compatible devices in range reset.

If any of the factory reset was not successful, and the devices are still not recognized, it might be the case that it is not supported, or something went wrong in the initial pairing. If after a Touchlink reset, the logs don't show the name of the device of the manufacturer there is a slight chance that something went wrong when writing to the database. In this case, just deleting the entry from `database.db` for the device without a name/manufacturer and redoing the factory reset should solve the problem. If the logs show the name and the manufacturer and the device is not supported(can be checked in front end), then there is a section `Support new devices` on the ZigBee2MQTT webpage that has a guideline on how to add a new device[14]. The setup and configuration are presented in Figure 9, where dotted arrows mean over-the-air communication, and normal arrows direct connection between laptop and device. The real setup is in Appendix A.2, Figure 24.

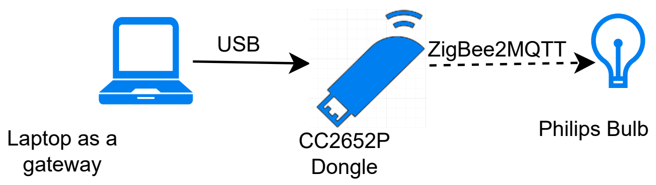


Figure 9: Experiment with CC2652P Dongle

5 PlutoSDR and GNU Radio

This chapter presents the essential tools used for transmitting ZigBee packets over the air. It describes both the hardware and software components necessary for sending and receiving ZigBee communications. Specifically, Section 5.1 introduces PlutoSDR, the software-defined radio (SDR) hardware used in the experiments, while Section 5.2 presents GNU Radio, the software framework responsible for signal processing. The integration of these tools is necessary for conducting the fuzzing experiments detailed in Section 6.

5.1 PlutoSDR

Software-defined radio (SDR) is a radio communication system in which traditional analog hardware components, such as mixers, filters, amplifiers, modulators/demodulators, and detectors, are replaced by software implementations on a computer or embedded system [15]. Although SDR is not a new concept, the rapidly evolving capabilities of digital electronics have made it possible to implement many processes that were previously only unfeasible in practice, such as quickly and efficiently switching between different frequencies and the possibility for real-time signal analysis and monitoring [16].For the research discussed, the SDR used is the PlutoSDR.

The PlutoSDR is utilized to manipulate ZigBee communications, focusing on the PHY and MAC layers of the ZigBee protocol stack, as illustrated in Figure 3. ZigBee operates on the IEEE 802.15.4 standard, which defines these foundational layers.

To facilitate fuzzing, a Python script generates ZigBee messages in which one or more protocol fields are deliberately altered. Each frame remains syntactically valid at the IEEE 802.15.4 level, ensuring that the devices, the SDR and the Philips bulb, can still send/receive the messages. These messages are sent over a UDP socket, where GNU Radio(Section 5.2) listens and reads the messages, this process and setup are discussed in more detail in Section 6.

5.1.1 Set-up

Upon connecting the PlutoSDR, users find a directory containing several documents, with `info.html` being the most significant. This file provides the most important information about the SDR, where to find installation instructions, what frameworks to use, the firmware version, and so on.

This file contains essential information about the SDR, including installation instructions, framework recommendations, and the current firmware version. To ensure optimal performance, it is necessary to have up-to-date firmware, this can be accomplished through two methods: `Mass Storage Update` and `Network Update`. For both methods, the instructions for the installation are presented on the firmware website [17].

5.1.2 Firmware Update Methods

Mass Storage Update: The Mass Storage Update method is the best suited for the initial setup of the PlutoSDR or when physical access to the device is available, as it typically offers a more straightforward process.

If the device is not functioning properly (e.g., due to corrupted firmware), the mass storage method can be used as a recovery option since it resets the firmware more directly. Additionally, the Mass Storage Update gives manual control over the update process, which can be useful if necessary to perform a thorough reset or change the bootloader settings.

Network update: When the PlutoSDR is set up remotely and network access to the device is available, a network update should be performed. This method allows for convenient maintenance and updating of the device without the need for physical access. These updates are generally faster and less disruptive, as they do not require physically disconnecting or manually rebooting the device. For situations requiring frequent updates due to bug fixes, security patches, or feature enhancements, network updates are more efficient.

To use the network update, the SDR should have internet access, and to enable internet access for the PlutoSDR via a host computer, the computer should be an internet gateway for the SDR. To set up this, several steps need to be followed only once unless there are significant changes to network configuration. First, IP forwarding must be enabled on the host system to allow packet forwarding between network interfaces. Then, Network Address Translation (NAT) should be set up to allow the PlutoSDR to share the host's internet connection, which involves masquerading outgoing traffic.

Firewall rules must be configured to permit packet forwarding from the PlutoSDR interface to the internet interface while maintaining security. Furthermore, PlutoSDR's default gateway needs to be assigned to direct its traffic to the host system, which routes it to the Internet. Modifying DNS settings is also essential to ensure the PlutoSDR can resolve domain names and access online resources. However, network access is not required in the current configuration, so this method has not been used.

5.2 GNURadio

As described in Section 5.1, the PlutoSDR hardware provides the necessary interface for capturing radio signals, while GNU Radio acts as the software counterpart for processing and analyzing these signals. This integration is particularly useful for decoding ZigBee communication, which requires precise implementation of the IEEE 802.15.4 PHY layer.

5.2.1 Overview of GNURadio

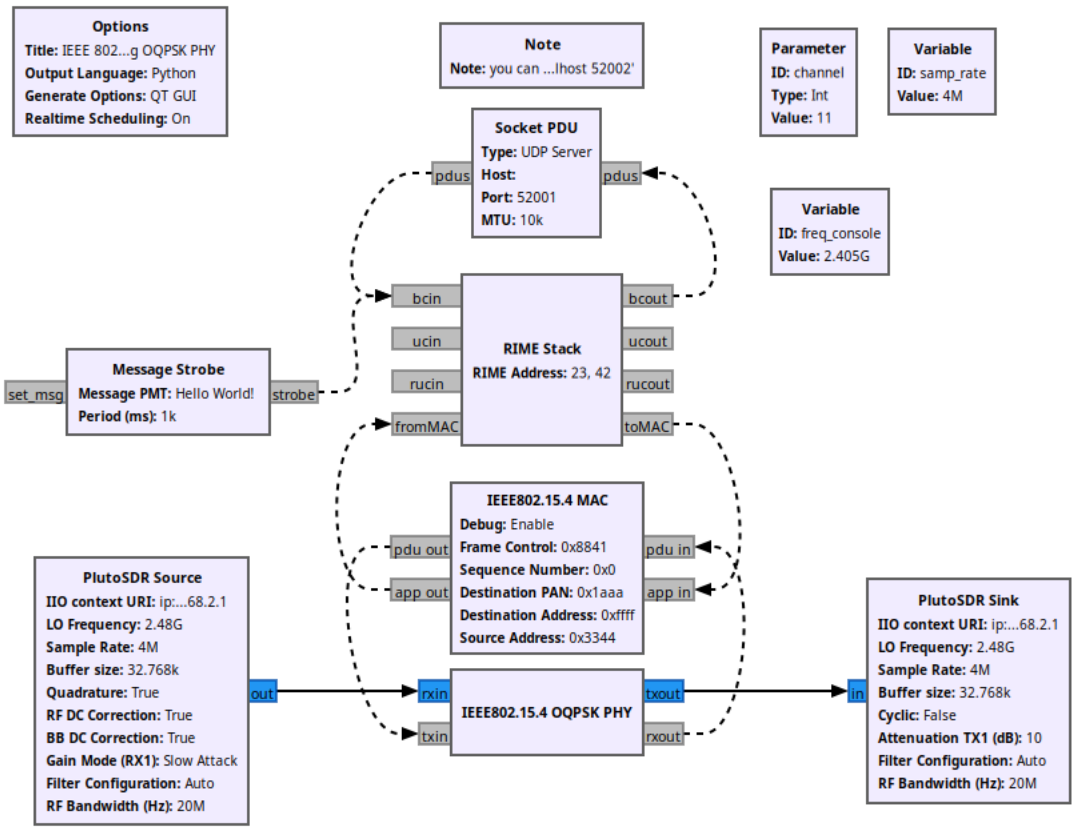
GNURadio is a free and open-source software development toolkit that provides a way to implement the software side of the SDR. Developed by the Free Software Foundation, GNU Radio provides a modular framework that allows users to design, implement, and simulate complex radio communication systems entirely in software, thus reducing the dependency on traditional hardware components such as mixers, filters, modulators, and demodulators. GNU Radio has two main components: `GNURadio Companion`, and `Signal Processing Blocks`.

GNU Radio Companion is a graphical interface that simplifies the design of flow networks for signal processing. Signal Processing Blocks represent the processing steps.

5.2.2 Integrating GNU Radio with PlutoSDR

Integrating GNU Radio with SDR provides a versatile framework for capturing and analyzing ZigBee communications, particularly at the PHY and MAC layer as defined by IEEE 802.15.4. Based on the analysis in Olaf van der Kruk's thesis(Section 5.2.2 [18]), the implementation of the PHY by Bloessl et al. was selected for its suitability in detailed analysis and manipulation of protocols, owing to its modular hierarchical block structure within GNU Radio[19].

Figure 10 illustrates the modified GNU Radio flowgraph for implementing the IEEE 802.15.4 PHY layer, essential to capture and analyze ZigBee communications with the PlutoSDR. This flowgraph is based on an adaptation by Olaf van der Kruk, who modified the `gr-ieee802-15-4` module to support the PlutoSDR [20]. Further adjustments were made to remove unnecessary components and streamline the setup for the specific needs of this application.



Options
Title: IEEE 802...g OQPSK PHY
Output Language: Python
Generate Options: QT GUI
Realtime Scheduling: On

Note
Note: you can ...lhost 52002'

Parameter
ID: channel
Type: Int
Value: 11

Variable
ID: samp_rate
Value: 4M

Socket PDU
Type: UDP Server
Host:
Port: 52001
MTU: 10k

Variable
ID: freq_console
Value: 2.405G

RIME Stack
RIME Address: 23, 42

Message Strobe
Message PMT: Hello World!
Period (ms): 1k

IEEE802.15.4 MAC
Debug: Enable
Frame Control: 0x8841
Sequence Number: 0x0
Destination PAN: 0x1aaa
Destination Address: 0xffff
Source Address: 0x3344

PlutoSDR Source
IIO context URI: ip:...68.2.1
LO Frequency: 2.48G
Sample Rate: 4M
Buffer size: 32.768k
Quadrature: True
RF DC Correction: True
BB DC Correction: True
Gain Mode (RX1): Slow Attack
Filter Configuration: Auto
RF Bandwidth (Hz): 20M

IEEE802.15.4 OQPSK PHY

PlutoSDR Sink
IIO context URI: ip:...68.2.1
LO Frequency: 2.48G
Sample Rate: 4M
Buffer size: 32.768k
Cyclic: False
Attenuation TX1 (dB): 10
Filter Configuration: Auto
RF Bandwidth (Hz): 20M

Figure 10: GNURadio radio blocks for ZigBee

The flowgraph diagram is structured to facilitate both signal transmission and reception, allowing for packet injection, monitoring, and analysis of ZigBee communications. It only implements the lower layer of the ZigBee stack Figure 3, namely the IEEE 802.15.4 part of the picture: PHY and MAC layers. The GNU Radio diagram uses the IEEE 802.15.4 O-QPSK PHY block responsible for modulating and demodulating ZigBee signals.

The Message Strobe block provides a hardcoded message in the flow diagram used for debugging if the diagram is working, e.g., if it sends ZigBee messages over a ZigBee channel but doesn't have utility in the fuzzing process. The PlutoSDR Source is used for receiving packets from a real ZigBee device, and the Socket PDU receives data from an external source over UDP, which is the Python script; these two components take part in the fuzzing experiments, explained in Section 6.

The output of the diagram is represented by the PlutoSDR Sink which transmits processed ZigBee frames over the air, also if PlutoSDR receives a ZigBee packet from a real device, the Socket PDU can be forwarded over UDP to another tool.

These blocks are all part of the original blocks used by Bloessl et al.[5].

6 Experiments with Fuzzing

We use Scapy, a Python-based packet crafting and manipulation tool, to automate the generation of malformed packets and integrate fuzzing capabilities. In contrast to Bloessl et al.'s approach[5], which simulates IEEE 802.15.4 communication in a controlled environment between software radios, our experiments target real commercial ZigBee devices. This real-world setup allows us to more accurately test the complexities of actual network conditions and uncover a broader range of potential vulnerabilities. While Bloessl's implementation can exchange valid IEEE.802.15.4 frames over the air using GNU Radio, it is not designed to transmit or accept malformed frames. In our work, we adapt this stack to remove validation logic and enable low-level frame injection for the purpose of protocol fuzzing.

In Section 6.1, the setup for the experiments is presented, including modifications to the GNU Radio diagram, the sniffer used, diagrams that explain the experiment flows, and how messages are observed and used. In Section 6.2 two replay attacks are presented, one when the message is replayed exactly as it is, and one where different counters are incremented. Section 6.3 and Section 6.4 presents fuzzing experiments in the MAC layer. Section 6.5 tries to go a layer down the stack and fuzz the PHY layer.

6.1 Setup

The experiments involve a Philips bulb, controlled via a physical on/off switch. The bulb is connected to a ZigBee network coordinated by a CC2652P dongle. All ZigBee devices, including the bulb and coordinator, operate on **channel 16**, as ZigBee uses multiple channels in the 2.4 GHz band.

During the experiments, the network was also sniffed. This process is done using A CC2531 dongle(which will be referred to as the sniffer), which forwards captured messages in Wireshark. A guide to setting up the sniffer and the configuration needed for Wireshark can be found on the ZigBee2MQTT website under the section `Sniff Zigbee` traffic[14].

The same messages for turning the bulb on and off were used during the experiments. They are captured using the setup in the Experiment 4.2 together with the sniffer. The first step was setting up a ZigBee2MQT, which gives a user interface that can be used to send messages for turning on and off the bulb. The sniffed messages are in ZigBee format as the sniffer observes only the ZigBee traffic.

The interactions between the CC2652P coordinator and the Philips bulb were analyzed using the sniffer, and messages for turning on/off the device were saved in hex format.

Different scripts are used for each of the experiments, but they have the same principles: take the full IEEE 802.15.4 frame in the hex format, transform it into raw bytes, make changes, and send it continuously every few seconds.

To send fuzzed messages over the air, we used a modified version of the GNU Radio flowgraph shown in Figure 10. Specifically, the RIME Stack and IEEE 802.15.4 MAC blocks were removed to gain direct control over the packet contents. These blocks were designed to construct standards-compliant frames, automatically adding headers, calculating checksums, and enforcing correct structure. While appropriate for normal communication, this behavior interfered with our goal of sending intentionally malformed frames, as it prevented transmission of non-conformant or partially corrupted packets. The modified flowgraph (Figure 12) allows arbitrary IEEE 802.15.4 PSDUs to be injected directly into the PHY layer and transmitted without validation

The initial association process between the bulb and the coordinator happens only once when the bulb is first connected to the network. This process, shown in Figure 11, includes Beacon Request, Beacon, Association Request/Response, Transport Key exchange, and a Device Announcement. After this one-time handshake, the bulb retains its assigned PAN ID and short MAC address, which do not change when turning the bulb on/off. After a successful handshake, the bulb no longer performs the full join process. Instead, it simply sends a ZigBee **Device Announcement**, a broadcast message informing the coordinator that the device is online and ready to receive commands.

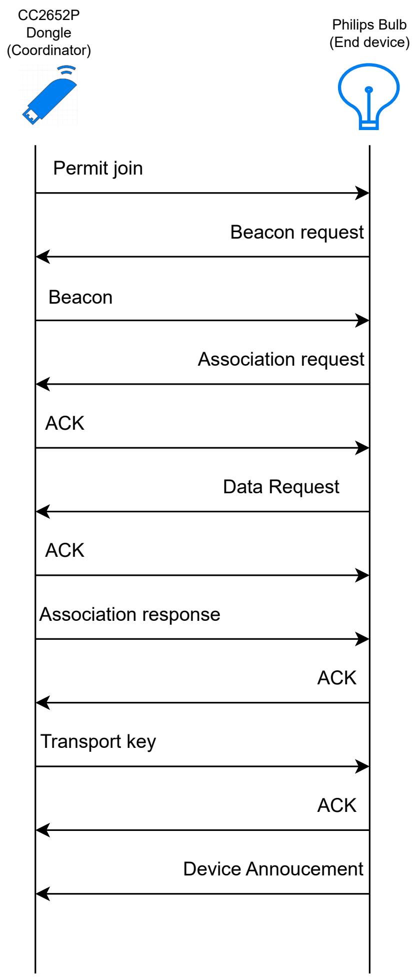
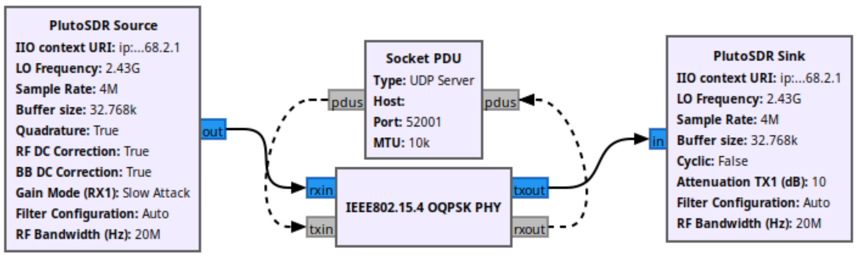


Figure 11: ZigBee one-time handshake between Coordinator and Philips bulb

In Figure 12, the PDU socket is listening on port 520001, the same port as the Python scripts are sending the packet. GNURadio also provides different blocks useful for debugging. One of these blocks is called `Message Debug`, which can be used to see the message format before it gets to the `IEEE802.15.4 OQPSK PHY`, Figure 13.



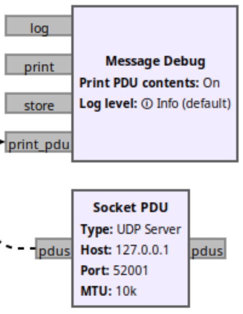
PlutoSDR Source
IIO context URI: ip:...68.2.1
LO Frequency: 2.43G
Sample Rate: 4M
Buffer size: 32.768k
Quadrature: True
RF DC Correction: True
BB DC Correction: True
Gain Mode (RX1): Slow Attack
Filter Configuration: Auto
RF Bandwidth (Hz): 20M

Socket PDU
Type: UDP Server
Host:
Port: 52001
MTU: 10k

IEEE802.15.4 OQPSK PHY

PlutoSDR Sink
IIO context URI: ip:...68.2.1
LO Frequency: 2.43G
Sample Rate: 4M
Buffer size: 32.768k
Cyclic: False
Attenuation TX1 (dB): 10
Filter Configuration: Auto
RF Bandwidth (Hz): 20M

Figure 12: Modified GNU radio diagram

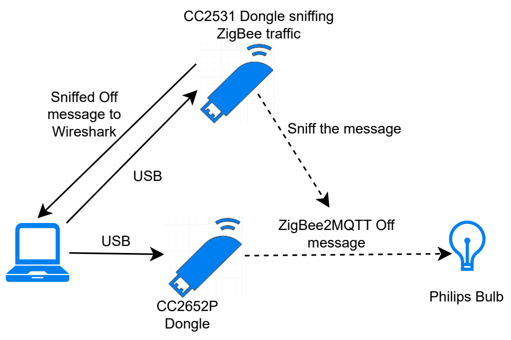


Message Debug
Print PDU contents: On
Log level: ⓘ Info (default)

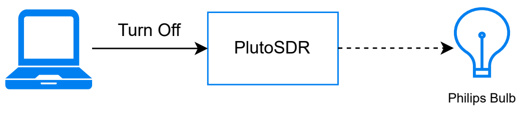
Socket PDU
Type: UDP Server
Host: 127.0.0.1
Port: 52001
MTU: 10k

Figure 13: Debug block connected to the Socket PDU

All the experiments follow the same diagram, including sniffing and transmission paths, shown in Figure 14. The dotted arrow represents communication over the air, and filled lines represent communication by plugging in the device to the computer. The real setup is in Appendix A.2, Figure 25 and Figure 26.



(a) Sniffing Setup



(b) Experiments Setup

Figure 14: Sniffing and replaying a message

The following experiments were carried out under two distinct conditions, illustrated in Figures 15 and 16:

- **Scenario 1**: Only the PlutoSDR and the bulb are powered. With the coordinator offline, no Zigbee network is active. After the bulb is switched on using the physical switch, it broadcasts a `Device Announcement` message indicating that it is ready to join. The SDR immediately sends a previously captured, encrypted Turn Off message, which the bulb accepts and executes (Figure 15). The real setup is the same as the one from Figure 14b, Appendix A.2, Figure 26.
- **Scenario 2**: The CC2652P coordinator running Zigbee2MQTT is online, so a full Zigbee network is present. Once the bulb powers up and announces itself, the coordinator sends a `Read Attributes` message, typically the first secured application-layer message exchanged when the devices have communicated before. This message includes the coordinator's 32-bit NWK frame counter. After this legitimate exchange, the SDR transmits the same `Turn Off` message as in Scenario 1 (Figure 16). The real setup is in Appendix A.2 Figure 27.

For each experiment, the Python script that was used is present in the Appendix. Each script was called in the `run.py` file from Appendix A.1.1.

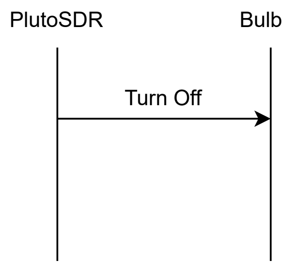


Figure 15: Message Sequence Chart when the SDR and the bulb are communicating

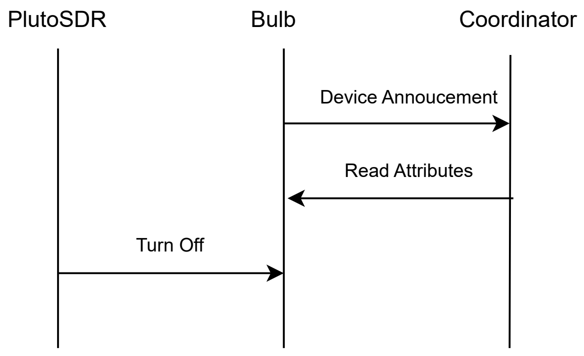


Figure 16: Message Sequence Chart when the SDR and the coordinator are communicating with the bulb

6.2 Replay attacks

This section presents two simple replay attacks: one where a captured ZigBee frame is replayed without modifications, and the second experiment is divided into two sub-experiments. These experiments serve as a baseline for evaluating the target device's resilience to malformed or reused frames and help validate the fuzzing setup before moving on to more complex mutation strategies.

6.2.1 Simple Replay attack - Experiment A (Scenario 1)

The objective of this experiment is to replay a previously captured ZigBee message without any modifications in order to turn off the bulb using an old encrypted message. The message is transmitted over the air using the setup described in Scenario 1.

Since the coordinator was deliberately kept offline during this test, no other secured messages were sent to the bulb upon startup. As a result, the SDR's replayed `Turn Off` message was treated by the bulb as the first valid secured message and was accepted, successfully switching the bulb off. This experiment successfully passes all the sequence numbers (Figure 6).

According to Lammers [3][Section 6.1.2] and the ZigBee Specification [9][Section 4.3.1 version 21], the 32-bit frame counter in the NWK security header must persist across factory resets and power outages and should never reset simply because the coordinator is absent. The fact that the message was accepted suggests this bulb is running an older ZigBee stack that clears its frame counter. This behavior effectively enables replay attacks in the absence of a frame-counter freshness check from the coordinator.

The script used for this experiment is in AppendixA.1.2.

6.2.2 Replay Attack with Modified Counters - Experiments B (Scenario 2)

The goal of this experiment is to assess how minor modifications to protocol-layer counters affect the acceptance of a replayed ZigBee message by a target device. This is done by altering fields in the MAC and after that the MAC and NWK headers, without re-encrypting the payload. The experiment is conducted under Scenario 2, where the ZigBee coordinator is active and has already exchanged legitimate messages with the bulb, establishing valid security frame counters. This setup is used to test ZigBee's replay protection behavior in a live network.

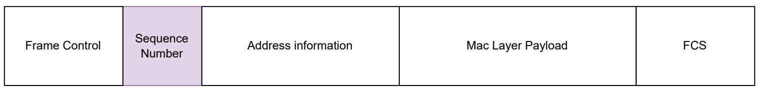


Figure 17: Fuzzed fields in the modified counters experiment

Experiment B.1: Only MAC counter modified

The goal of this experiment is to test whether a message with only the MAC sequence number changed is still accepted by the target device. This is done using a Python script, which modifies the MAC header using Scapy while leaving all upper layer fields, specifically the NWK frame counter. This simulates an attacker attempting to bypass replay protection by changing only the lowest layer counter. The fields that are modified are presented in Figure 17.

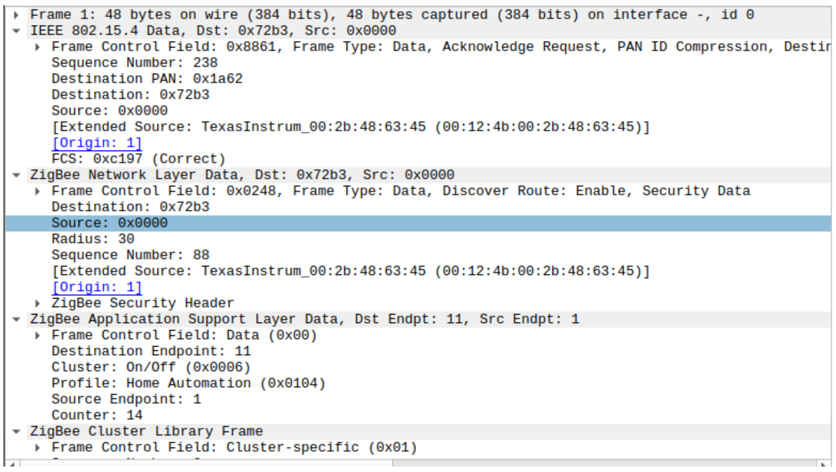
As shown in Figure 18(a), the resulting packet is still structurally valid. Wireshark can parse the frame, and the ZigBee Cluster Library (ZCL) payload remains visible. The ZCL is part of the Application Framework(green area of Figure 3). However, the bulb ignores the message. This is expected behavior: according to the ZigBee Specification [9, Section 4.3.1], devices track the highest received NWK frame counter per sender. Because the reused NWK counter is lower than the one already received from the coordinator, the frame is discarded as a replay. The message stops at the second check (bottom to top) from Figure 6.

Experiment B.2: MAC and NWK counters modified

The goal of this experiment is to determine whether simply increasing both the MAC and NWK frame counters without re-encrypting the payload is a replayed message to bypass ZigBee's integrity checks. In ZigBee, the payload is encrypted and authenticated using AES-CCM*, which combines encryption with a Message Integrity Code (MIC). The MIC is computed over the entire frame, including the NWK frame counter and the encrypted payload(Section 3.5).

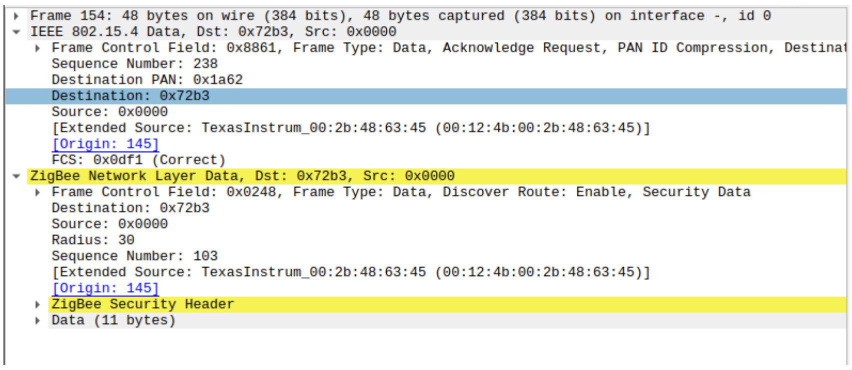
As shown in Figure 18(b), this packet is no longer parsed correctly. Wireshark cannot decode the message, and the target device immediately discards it. This is because the NWK frame counter is used in the AES-CCM* encryption algorithm to compute the MIC. Modifying the counter without updating the MIC breaks integrity, rendering the frame invalid. The NWK counter resides in the network-layer security header(red area in Figure 3), which is transmitted as part of the MAC payload (Figure 17).

The script used for both experiments is in AppendixA.1.3.



```
▸ Frame 1: 48 bytes on wire (384 bits), 48 bytes captured (384 bits) on interface -, id 0
▾ IEEE 802.15.4 Data, Dst: 0x72b3, Src: 0x0000
  ▸ Frame Control Field: 0x8861, Frame Type: Data, Acknowledge Request, PAN ID Compression, Destir
    Sequence Number: 238
    Destination PAN: 0x1a62
    Destination: 0x72b3
    Source: 0x0000
    [Extended Source: TexasInstrum_00:2b:48:63:45 (00:12:4b:00:2b:48:63:45)]
    [Origin: 1]
    FCS: 0xc197 (Correct)
▾ ZigBee Network Layer Data, Dst: 0x72b3, Src: 0x0000
  ▸ Frame Control Field: 0x0248, Frame Type: Data, Discover Route: Enable, Security Data
    Destination: 0x72b3
    Source: 0x0000
    Radius: 30
    Sequence Number: 88
    [Extended Source: TexasInstrum_00:2b:48:63:45 (00:12:4b:00:2b:48:63:45)]
    [Origin: 1]
  ▸ ZigBee Security Header
▾ ZigBee Application Support Layer Data, Dst Endpt: 11, Src Endpt: 1
  ▸ Frame Control Field: Data (0x00)
    Destination Endpoint: 11
    Cluster: On/Off (0x0006)
    Profile: Home Automation (0x0104)
    Source Endpoint: 1
    Counter: 14
▾ ZigBee Cluster Library Frame
  ▸ Frame Control Field: Cluster-specific (0x01)
```

(a) Observed message with only MAC counter increased



```
▸ Frame 154: 48 bytes on wire (384 bits), 48 bytes captured (384 bits) on interface -, id 0
▾ IEEE 802.15.4 Data, Dst: 0x72b3, Src: 0x0000
  ▸ Frame Control Field: 0x8861, Frame Type: Data, Acknowledge Request, PAN ID Compression, Destinat
    Sequence Number: 238
    Destination PAN: 0x1a62
    Destination: 0x72b3
    Source: 0x0000
    [Extended Source: TexasInstrum_00:2b:48:63:45 (00:12:4b:00:2b:48:63:45)]
    [Origin: 145]
    FCS: 0x0df1 (Correct)
▾ ZigBee Network Layer Data, Dst: 0x72b3, Src: 0x0000
  ▸ Frame Control Field: 0x0248, Frame Type: Data, Discover Route: Enable, Security Data
    Destination: 0x72b3
    Source: 0x0000
    Radius: 30
    Sequence Number: 103
    [Extended Source: TexasInstrum_00:2b:48:63:45 (00:12:4b:00:2b:48:63:45)]
    [Origin: 145]
  ▸ ZigBee Security Header
  ▸ Data (11 bytes)
```

(b) Observed message with MAC and NWK counters increased

Figure 18: Comparison of replayed packets with different counter modifications

6.3 Fuzzing the Frame Control field – Experiments C (Scenario 1)

The goal of this experiment is to identify potential vulnerabilities in the Frame Control field of IEEE 802.15.4 by applying fuzzing techniques. This includes both targeted fuzzing of individual subfields and random fuzzing of the entire field. The experiment is conducted under Scenario 1 and aims to determine whether unusual or malformed combinations within the Frame Control field can trigger unexpected behavior in the bulb. The fields of the Frame Control field can be observed in Figure 5. This experiment had two sub-experiments: the first was fuzzing each subfield individually and then fuzzing the entire FCF.

Experiment C.1: Fuzzing each subfield

Despite systematically fuzzing each subfield in the Frame Control field and testing every possible bit combination in isolation, none of the modifications resulted in a crash or notable malfunction of the target device. Even when individual subfields contain unexpected bit patterns, the device simply discards the malformed frame rather than processing it in a way that could cause a crash or undesirable behavior. This test has 34 test cases, and depending on the time between the two frames sent, the experiment should take around 1 minute. The script used for this experiment is in Appendix A.1.4.

Experiment C.2: Fuzzing entire Frame Control field Experiment:

Rather than isolating individual subfields, random and patterned inputs are applied to the full field to uncover vulnerabilities that might arise from interactions between subfields. While this approach is less granular, it can potentially reveal behaviors that do not appear when subfields are fuzzed in isolation. Both random and patterned inputs were used, including incrementing values, completely random values, and values derived by XOR-ing a known valid FCF. This approach aimed to discover issues that may only arise due to interactions between subfields. Figure 19 illustrates the fuzzed fields.

If all the combinations are tested, there are 2^{16} possible combinations. Sending 1000 packets takes around three minutes with 200ms between messages being sent. Sending at the same rate, all the 2^{16} messages take around three and a half hours.

However, no observable side effects, crashes, or abnormal behaviors were triggered during this experiment phase. The device continued to operate normally, discarding malformed frames without impact. While the results did not expose vulnerabilities, they confirm that the MAC layer implementation is resilient even when unpredictable and non-standard combinations of the Frame Control field. In both experiments, the messages were dropped at the MAC layer(purple area in Figure 3). The script used for this experiment is in Appendix A.1.5.

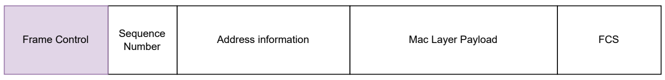


Figure 19: Fuzzed fields Experiment C.2

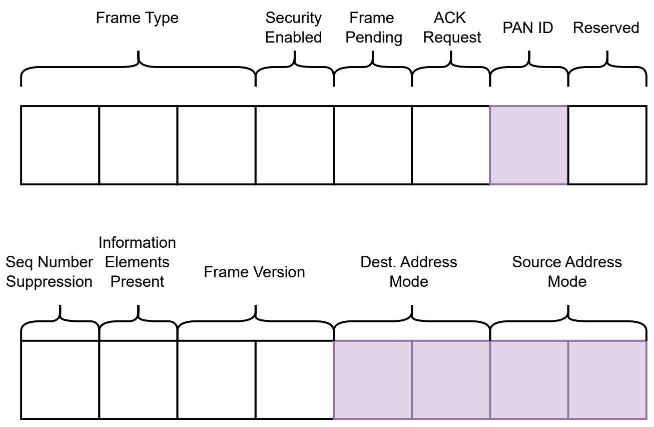
6.4 Experiment D: Address Field Fuzzing (Scenario 1)

The goal of this experiment is to evaluate how the target device handles variations in the Address Information field of the IEEE 802.15.4 MAC header. This is achieved by dynamically modifying the Frame Control Field (FCF) bits related to addressing, specifically the Destination Address Mode, Source Address Mode, and PAN ID Compression bits, across all valid and invalid combinations. The experiment is conducted under Scenario 1 and aims to determine whether unusual or malformed address configurations can lead to parsing errors, inconsistent behavior, or potential vulnerabilities in the frame-handling logic of the device.

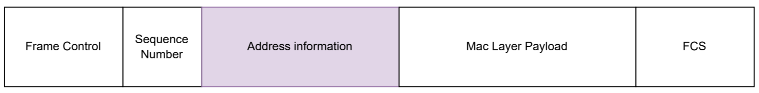
For each possible case, the script sends two different malformed address variants for both destination and source:

- long expected → short address only, short + padded zero bytes
- short expected → short address only, short + padded zero bytes

The experiment tested all possible combinations of address modes: short destination with short source, short destination with long source, long destination with short source, and long destination with long source. For each setup, frames were built by either sending the correct address size or sending an address that was incorrectly sized (either missing padding or with extra padding zeros). The Philips Hue bulb continued to operate normally without any crash or reset. This shows that the bulb properly validates address information in the MAC header and safely discards frames that do not match the expected addressing structure. This experiment has 32 possible combinations, which take a few seconds to send. Figure 20 illustrates the fuzzed fields. The script used for this experiment is in Appendix A.1.6.



(a) Fields in the Grame Control Filed



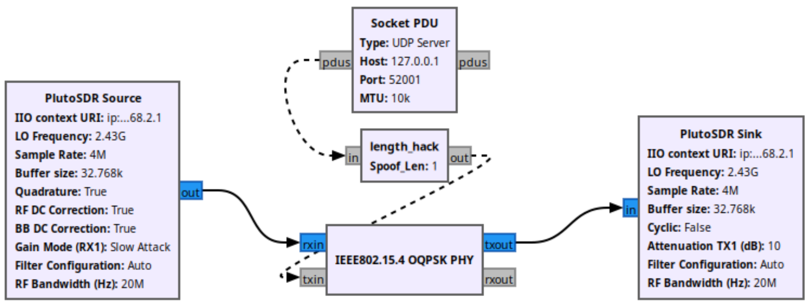
(b) MAC layer fields

Figure 20: Fuzzed fields in Experiment D

6.5 Fuzzing the Frame Length from PHY header Experiment E (Scenario 1)

The goal of this experiment is to manipulate the PHY frame length in order to assess whether spoofing this value affects transmission or device behavior. To enable direct modifications to the PHY header, a custom GNU Radio flow graph was created, as shown in Figure 21. An Embedded Python block was added to the flow graph to overwrite the length field in the outgoing metadata prior to transmission.

Although the script successfully injected a custom value (e.g., 1) into the metadata, the transmitted packet still contained the original length of 48 bytes. This suggests that the PHY layer block recalculates the frame length internally, overriding any external modifications made via metadata. As a result, the spoofed length never reaches the air interface. The specific field targeted for fuzzing is highlighted in Figure 22.



Socket PDU
Type: UDP Server
Host: 127.0.0.1
Port: 52001
MTU: 10k

PlutoSDR Source
IIO context URI: ip:...68.2.1
LO Frequency: 2.43G
Sample Rate: 4M
Buffer size: 32.768k
Quadrature: True
RF DC Correction: True
BB DC Correction: True
Gain Mode (RX1): Slow Attack
Filter Configuration: Auto
RF Bandwidth (Hz): 20M

length_hack
Spoof_Len: 1

IEEE802.15.4 OQPSK PHY

PlutoSDR Sink
IIO context URI: ip:...68.2.1
LO Frequency: 2.43G
Sample Rate: 4M
Buffer size: 32.768k
Cyclic: False
Attenuation TX1 (dB): 10
Filter Configuration: Auto
RF Bandwidth (Hz): 20M

Figure 21: GNU radio diagram modified.

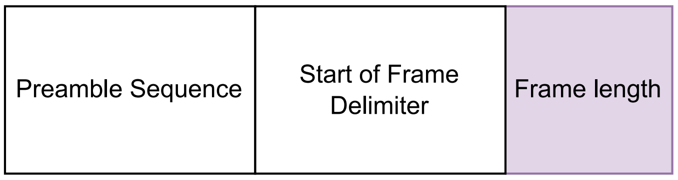


Figure 22: Fuzzed fields in Experiment E

7 Future work

In the experiments from Section 6, the PAN ID and coordinator address used by the SDR are already set to match the original coordinator, allowing the bulb to accept MAC-layer frames without requiring a new handshake. However, performing a clean, legitimate handshake with the SDR acting as the coordinator can still improve the setup. It ensures the device is in a known, fresh state, with properly initialized counters and network parameters, making results more reliable and reproducible. More importantly, it allows fuzzing of the handshake process itself, such as altering Association Request fields, Transport Key contents, or join message timing. This would allow for a more complete exploration of vulnerabilities, both during and immediately after the network join phase.

Another important direction for future work is to perform fuzzing on secured ZigBee messages by directly manipulating the NWK layer. This involves crafting packets with updated frame counters and properly encrypting the payload using the active network key, as described in the ZigBee specification. Since the NWK frame counter is part of the input to the AES-CCM* algorithm, any change to it requires the message to be re-encrypted and the MIC to be recomputed. In the setup from Section 6, only pre-captured encrypted messages are replayed or slightly modified, which breaks the MIC and causes the frame to be rejected. A complete toolchain is needed to generate secured ZigBee frames that pass all integrity checks dynamically. With this in place, fuzzing can be extended to include subtle changes to secured payloads or NWK headers, while still being accepted by the target device. This would allow a deeper investigation of how devices validate incoming frames and handle edge cases in security processing.

Another challenge is to make Section 6.5 work by successfully modifying the PHY header, specifically the frame length field. In the current setup, the final transmitted frame still contains the original length, even when a custom value is set in the metadata using an Embedded Python block in GNU Radio. This is likely because the PHY block in the GNU Radio implementation automatically recalculates the frame length before transmission. To solve this, one option is to modify the PHY block code to skip or disable this recalculation step. If this can be achieved, it would allow fuzzing to go one level deeper and test how devices handle frames with incorrect or edge-case values in the physical layer header, such as lengths that are too short, too long, or inconsistent with the actual payload. This would complete the low-level fuzzing pipeline and help uncover potential weaknesses in the receiver's PHY layer handling.

8 Conclusions

This thesis presents a practical approach for conducting low-level fuzzing of ZigBee networks using a real-world testbed built entirely with open-source tools. It demonstrates the feasibility of applying SDR-based fuzzing at the data link layer and highlights its potential to uncover implementation-specific issues that might remain undetected in simulation-based environments. As ZigBee continues to be widely adopted in both consumer and industrial IoT devices, ensuring its resilience against low-level protocol manipulation remains crucial.

This thesis aimed to apply fuzzing techniques to the IEEE 802.15.4 data link layer of ZigBee, a widely used protocol in IoT devices, to uncover potential vulnerabilities. Two types of experiments were conducted to support this investigation. The first set of experiments (Section 4) focused on establishing a ZigBee network composed of a coordinator and an end device, specifically a Philips Hue smart bulb. Two experiments were conducted to create a ZigBee network. In the first experiment (Section 4.1, Figure 8), a Philips Hue Bridge was used as the coordinator, but the setup proved too complex and ultimately unsuccessful. In the second experiment (Section 4.2, Figure 9), a CC2652P dongle was used instead, offering a more compact and observable setup. This configuration also incorporated ZigBee2MQTT, an open-source platform that supports over 4000 ZigBee devices and enables integration across vendors. The second setup served as the foundation for the subsequent fuzzing experiments described in Section 6.

To transmit messages over the air, this thesis uses a PlutoSDR device (Section 5.1). The PlutoSDR is straightforward to set up; once connected, it opens a directory containing an `info.html` file with all necessary installation instructions. After the initial configuration, no further changes are typically required. GNU Radio was employed as the software framework for processing the radio signals sent and received by the PlutoSDR. GNU Radio is a flexible and powerful tool that enables the design of custom signal-processing graphs. It also offers robust debugging capabilities, which proved especially useful during the fuzzing experiments.

The second set of experiments is the fuzzing ones. They are presented in Section 6 and are categorized into two types: replay attacks (Section 6.2.1) and fuzzing techniques attacks(Sections 6.3–6.5). All experiments utilized a `Turn Off` message that was captured using a CC2531 dongle while monitoring communication between the coordinator and the end device, as described in the setup from Section 4.2. Each experiment was conducted under a specific scenario. These two scenarios indicate if the CC2652P coordinator was present or not in the network.

The replay attack experiments demonstrated that the end device could still accept replayed messages in the absence of a network coordinator. This highlights a potential vulnerability in devices that fail to retain or verify security frame counters across reboots or power cycles. However, when the coordinator was active, replayed messages with outdated or modified counters were consistently rejected, confirming that ZigBee's layered counter-based security checks (as described in Section 6.2.1) are effective when properly enforced.

The fuzzing experiments targeted different components of the data link layer (Medium Access Control and Physical layers), including the Frame Control Field (FCF) of the MAC, address fields, and physical frame length. The fuzzing experiments were all conducted in the absence of the coordinator. In all cases, the Philips Hue bulb demonstrated resilience by safely discarding malformed packets without crashing or exhibiting unexpected behavior. These results indicate a robust implementation of the IEEE 802.15.4 stack in the tested device. However, the inability to fuzz certain PHY-level fields, such as the frame length, due to internal recalculation by the GNU Radio PHY block limited the depth of the low-level testing.

Collectively, these experiments show that while no critical vulnerabilities were discovered in the tested device, the methodology proved effective for exploring potential weaknesses. The real-world setup confirmed that the Philips bulb is generally well-defended against malformed input under normal operating conditions but also revealed areas, such as frame counter resets, where older or less secure devices may still be vulnerable.

References

[1] Grand View Research, *IoT Market Size, Share Trends Analysis Report By Component, By Application, By End-use, By Region, And Segment Forecasts, 2024 - 2030*, 2023. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/iot-market.

[2] Fortune Business Insights. "Internet of Things (IoT) Market Size, Share and COVID-19 Impact Analysis, By Component, By Platform, By Technology, By End-User, and Regional Forecast, 2023-2032." (2023), [Online]. Available: https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307.

[3] J. Lammers, "From larvae to KillerBee: A survey on ZigBee and IEEE 802.15.4 security," Master thesis, University of Groningen, 2023.

[4] R. Sokullu, I. Korkmaz, O. Dagdeviren, A. Mitseva, and N. R. Prasad, "An investigation on IEEE 802.15.4 MAC layer attacks," in *Proc. of WPMC*, vol. 41, 2007, pp. 42–92.

[5] B. Bloessl, C. Leitner, F. Dressler, and C. Sommer, "A GNU radio-based IEEE 802.15. 4 testbed," *12. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (FGSN 2013)*, pp. 37–40, 2013.

[6] Z. Shang, M. E. Garbelini, and S. Chattopadhyay, "U-FUZZ: Stateful Black-box Fuzzing of IoT Protocols on COTS Devices," in *Proceedings of the 45th IEEE Symposium on Security and Privacy*, IEEE S&P, IEEE, 2024.

[7] K. Sohraby, D. Minoli, and T. Znati, *Wireless Sensor Networks: Technology, Protocols, and Applications.* John Wiley & Sons, 2007, ISBN: 978-0-471-74300-2.

[8] D. Gislason, *Zigbee Wireless Networking.* Elsevier, 2008, ISBN: 978-0-7506-8597-9.

[9] C. S. Alliance. "Zigbee Specification." version 05-3474-21. (2015), [Online]. Available: https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf.

[10] Digi International, *XBee/XBee-PRO S2C Zigbee RF Module User Guide.* [Online]. Available: https://www.digi.com/resources/documentation/Digidocs/90001539/#tasks/t_config_zb_comm_s2c.htm.

[11] J. A. Gutierrez, E. H. Callaway, and R. L. Barrett, "An introduction to IEEE STD 802.15.4," *IEEE Communications Magazine*, vol. 42, no. 6, pp. 28–33, 2004. [Online]. Available: https://www.inf.ufes.br/~zegonc/material/Redes%20de%20Sensores%20sem%20Fio/IEEE_802.15.4-ARTIGO-An%20Introduction%20to%20IEEE%20STD%20802.15.4.pdf.

[12] K. Choi, Y. Son, J. Noh, H. Shin, J. Choi, and Y. Kim, "Dissecting customized protocols: Automatic analysis for customized protocols based on IEEE 802.15.4," pp. 183–193, Jul. 2016.

[13] A. Li, J. Li, D. Han, Y. Zhang, T. Li, T. Zhu, and Y. Zhang, "PhyAuth: Physical-Layer message authentication for ZigBee networks," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 1–18, ISBN: 978-1-939133-37-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/li-ang.

[14] Koenkk and Zigbee2MQTT Contributors. "Zigbee2MQTT." (2018), [Online]. Available: https://www.zigbee2mqtt.io/.

[15] M. Dillinger, K. Madani, and N. Alonistioti, *Software Defined Radio: Architectures, Systems and Functions.* Wiley & Sons, 2003, ISBN: 0-470-85164-3.

[16] T. F. Collins, R. Getz, D. Pu, and A. M. Wyglinski, *Software-Defined Radio for Engineers*. Artech House, 2020, ISBN: 978-1-63081-457-1.

[17] Analog Devices. “Pluto/M2k Firmware Updates.” Last modified: 21 Aug 2023. (2023), [Online]. Available: `https://wiki.analog.com/university/tools/pluto/users/firmware`.

[18] Olaf van der Kruk, “Automata learning for ZigBee,” Master’s thesis, Open University, 2025.

[19] B. Bloessl, C. Leitner, F. Dressler, and C. Sommer, “A GNU Radio-based IEEE 802.15.4 Testbed,” in *12. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (FGSN 2013)*, 2013, pp. 37–40.

[20] Olaf van der Kruk, *GNU Radio and Soapy SDR setup for PlutoSDR*, GitHub repository, 2024. [Online]. Available: `https://github.com/olijf/docker-gnuradio-soapy`.

A Appendix

A.1 Python Code

1 run.py

```python
def main():
    off_hex = """
        61 88 f8 62 1a d5 2c 00 00 48 02 d5 2c 00 00 1e
        38 28 f6 df 00 00 45 63 48 2b 00 4b 12 00 00 ae
        51 10 0f 58 c9 73 b6 6c 94 86 db 10 c4 e4 55 6a
        """.strip()

    frame_bytes = bytearray.fromhex(off_hex.replace("\n", " ").strip())

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    addr = ("127.0.0.1", 52001)

    replay_attack(frame_bytes, sock, addr)
    replay_attack_incremented_counters(frame_bytes, sock, addr, both=True)
    fuzz_fcf_each_subfield(frame_bytes, sock, addr)
    fuzz_fcf_all_fields(frame_bytes, sock, addr)
    fuzz_addresses(frame_bytes, sock, addr)
    sock.close()
```

2 replay_attack_incremented_counters.py

```python
def replay_attack(frame_bytes, sock, addr):
    while True:
        sock.sendto(frame_bytes, ("127.0.0.1", 52001))
```

3 replay_attack_incremented_counters.py

```python
def increment_all_counters(pkt, both):
    pkt.seqnum = (pkt.seqnum + 1) % 256
    if both:
        nwk_layer = pkt.getlayer(ZigbeeNWK)
        if nwk_layer is not None:
            nwk_layer.seqnum = (nwk_layer.seqnum + 1) % 256

def replay_attack_incremented_counters(frame_bytes, sock, addr, both):
    pkt = Dot15d4FCS(frame_bytes)

    while True:
        increment_all_counters(pkt, both)
        pkt.fcs = None
        final_bytes = bytes(pkt)
        sock.sendto(final_bytes, addr)
        time.sleep(1)
```

4 fuzz_fcf_each_subfield.py

```python
def get_fcf(frame):
    return frame[0] | (frame[1] << 8)

def set_fcf(frame, new_fcf):
    frame[0] = new_fcf & 0xFF
    frame[1] = (new_fcf >> 8) & 0xFF

def update_fcf_field(frame, mask, shift, value):
    current = get_fcf(frame)
    patched = (current & ~mask) | ((value << shift) & mask)
    set_fcf(frame, patched)

def fuzz_field(frame, sock, addr, mask, shift, value_range):
    original_fcf = get_fcf(frame)
    for val in value_range:
        set_fcf(frame, original_fcf)
        sock.sendto(frame, addr)
        time.sleep(0.1)

        update_fcf_field(frame, mask, shift, val)
        sock.sendto(frame, addr)
        time.sleep(0.1)

        set_fcf(frame, original_fcf)
        sock.sendto(frame, addr)
        time.sleep(0.1)

    set_fcf(frame, original_fcf)

def fuzz_fcf_each_subfield(frame_bytes, sock, addr):
    fields = [
        {"name": "Frame Type", "mask": 0x0007, "shift": 0, "range": range(8)},
        {"name": "Security Enabled", "mask": 0x0008, "shift": 3, "range": range(2)},
        {"name": "Frame Pending", "mask": 0x0010, "shift": 4, "range": range(2)},
        {"name": "Ack Request", "mask": 0x0020, "shift": 5, "range": range(2)},
        {"name": "PAN ID Compression", "mask": 0x0040, "shift": 6, "range": range(2)},
        {"name": "Reserved (bit 7)", "mask": 0x0080, "shift": 7, "range": range(2)},
        {"name": "Frame Version", "mask": 0x0300, "shift": 8, "range": range(4)},
        {"name": "Source Address Mode", "mask": 0x0C00, "shift": 10, "range": range(4)},
        {"name": "Destination Address Mode", "mask": 0x3000, "shift": 12, "range": range(4)},
        {"name": "Reserved (bit 14)", "mask": 0x4000, "shift": 14, "range": range(2)},
        {"name": "Reserved (bit 15)", "mask": 0x8000, "shift": 15, "range": range(2)},
    ]
    for f in fields:
        fuzz_field(frame_bytes, sock, addr, f["mask"], f["shift"], f["range"])
```

5 fuzz_fcf_all_fields.py

```python
def send_frame(pkt, sock, addr):
    fuzz_pkt = Dot15d4FCS(bytes(pkt))
    sock.sendto(bytes(fuzz_pkt), addr)
    fcf = (bytes(pkt)[0] << 8) | bytes(pkt)[1]

def fuzz_fcf_full(base_pkt, sock, addr, mode="random", count=1000):
    if not hasattr(base_pkt, "getlayer"):
        base_pkt = Raw(load=bytes(base_pkt))
    sent = 0
    pkt = base_pkt.copy()

    while True:
        if mode == "random":
            fcf = random.randint(0, 0xFFFF)
        elif mode == "increment":
            fcf = sent % 0x10000
        elif mode == "pattern":
            fcf = 0x8861 ^ sent
        else:
            raise ValueError("Unknown fuzzing mode")

        raw = pkt.getlayer(Raw)
        load = bytearray(raw.load)

        load[0] = fcf & 0xFF
        load[1] = (fcf >> 8) & 0xFF

        raw.load = bytes(load)

        send_frame(pkt, sock, addr)
        time.sleep(0.2)
        sent += 1

        if sent >= count:
            print("Reached end of fuzz range.")
            break


def fuzz_fcf_all_fields(frame_bytes, sock, addr):
    # fuzz_fcf_full(frame_bytes, sock, addr, mode="random", count=1000)
    # fuzz_fcf_full(frame_bytes, sock, addr, mode="increment", count=65536)
    fuzz_fcf_full(frame_bytes, sock, addr, mode="pattern", count=1000)
```

6 fuzz_addresses.py

```python
PAN, SHORT_ADDR = 0x621A, 0x2CD5
EXT_ADDR = SHORT_ADDR << 48

def compute_fcs(data: bytes) -> bytes:
    return Dot15d4FCS().compute_fcs(data)

def build_fcf(dest_mode: int, src_mode: int, pan_comp: int) -> int:
    return (
        0x0001
        | ((pan_comp & 1) << 6)
        | ((dest_mode & 3) << 10)
        | ((src_mode & 3) << 12)
    )

def _short_bytes() -> bytes:
    return struct.pack("<H", SHORT_ADDR)

def _long_bytes(padded: bool = False) -> bytes:
    return _short_bytes() + b"\x00" * 6 if padded else struct.pack("<Q",
        EXT_ADDR)

def build_address_field(dest_mode, src_mode, pan_comp, dest_var=None, src_var=
    None) -> bytes:
    fld = bytearray()
    if dest_mode:
        fld += struct.pack("<H", PAN)
        if dest_var == "short-only":
            fld += _short_bytes()
        elif dest_var == "short-pad":
            fld += _long_bytes(padded=True)
        else:
            fld += _short_bytes() if dest_mode == 2 and dest_var != "long-pad"
                else _long_bytes()
    if src_mode:
        if not pan_comp:
            fld += struct.pack("<H", PAN)
        if src_var == "short-only":
            fld += _short_bytes()
        elif src_var == "short-pad":
            fld += _long_bytes(padded=True)
        else:
            fld += _short_bytes() if src_mode == 2 and src_var != "long-pad" else
                _long_bytes()
    return bytes(fld)

def fuzz_once(frame_bytes: bytes, sock: socket.socket, addr: tuple,
    start_mac_seq: int = 0xBE, delay: float = 0.2):
    PAYLOAD = frame_bytes[15:-2]
    mac_seq = start_mac_seq & 0xFF

    def assemble(mac_seq, d_mode, s_mode, pan_c, d_var=None, s_var=None):
        hdr = struct.pack("<H B", build_fcf(d_mode, s_mode, pan_c), mac_seq)
        return hdr + build_address_field(d_mode, s_mode, pan_c, d_var, s_var) +
            PAYLOAD

    for d_mode, s_mode, pan_c in itertools.product((2, 3), (2, 3), (0, 1)):
        # baseline
        frame = assemble(mac_seq, d_mode, s_mode, pan_c)
        sock.sendto(frame + compute_fcs(frame), addr)
        time.sleep(delay)

        if d_mode == 3:
            for v in ("short-only", "short-pad"):
                frame = assemble(mac_seq, d_mode, s_mode, pan_c, d_var=v)
                sock.sendto(frame + compute_fcs(frame), addr)
                time.sleep(delay)
        else:
            frame = assemble(mac_seq, d_mode, s_mode, pan_c, d_var="long-pad")
            sock.sendto(frame + compute_fcs(frame), addr)
            time.sleep(delay)

        if s_mode == 3:
            for v in ("short-only", "short-pad"):
                frame = assemble(mac_seq, d_mode, s_mode, pan_c, s_var=v)
                sock.sendto(frame + compute_fcs(frame), addr)
                time.sleep(delay)
        else:
            frame = assemble(mac_seq, d_mode, s_mode, pan_c, s_var="long-pad")
            sock.sendto(frame + compute_fcs(frame), addr)
            time.sleep(delay)

        mac_seq = (mac_seq + 1) & 0xFF

def fuzz_addresses(frame_bytes, sock, addr):
    fuzz_once(frame_bytes, sock, addr)
```

A.2 Setup pictures

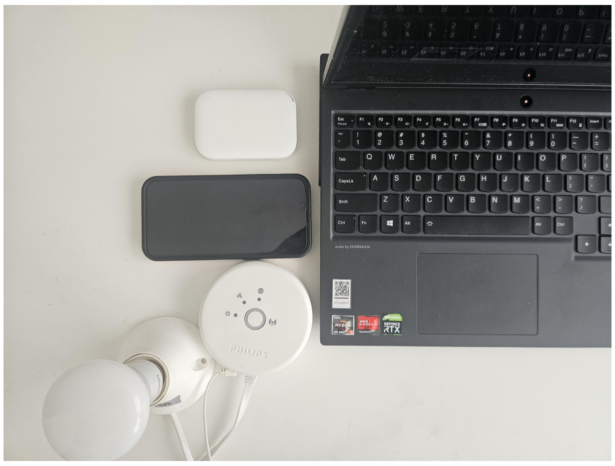


Figure 23: Setup with Philips Hue Bridge

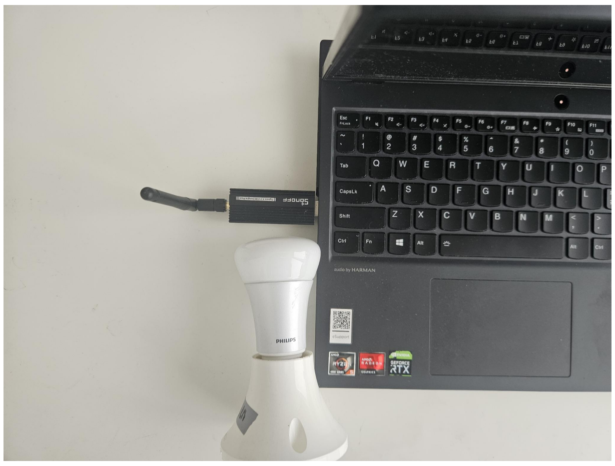


Figure 24: Setup with CC2652P coordinator

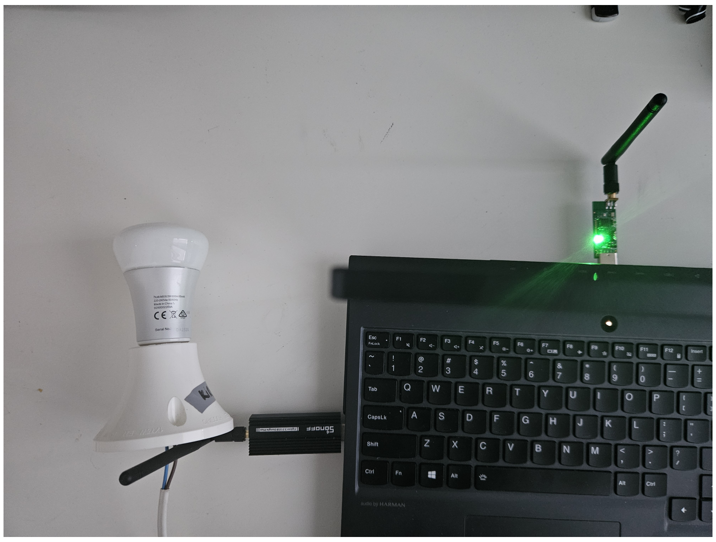


Figure 25: Sniffing Setup

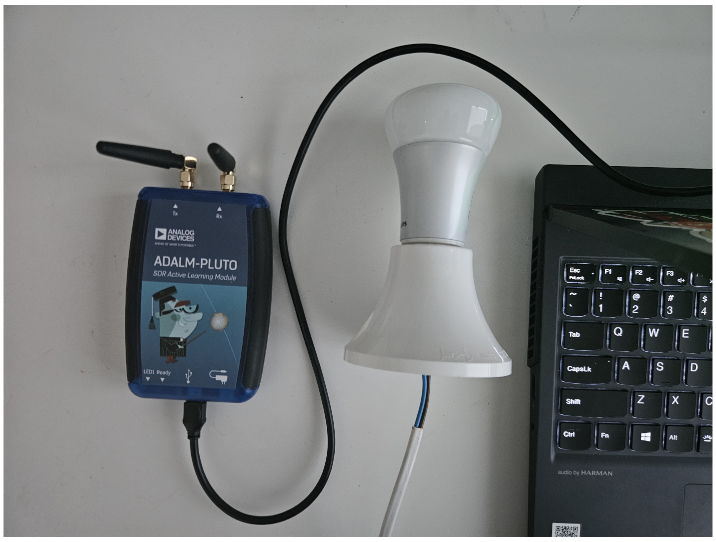


Figure 26: Experiments Setup

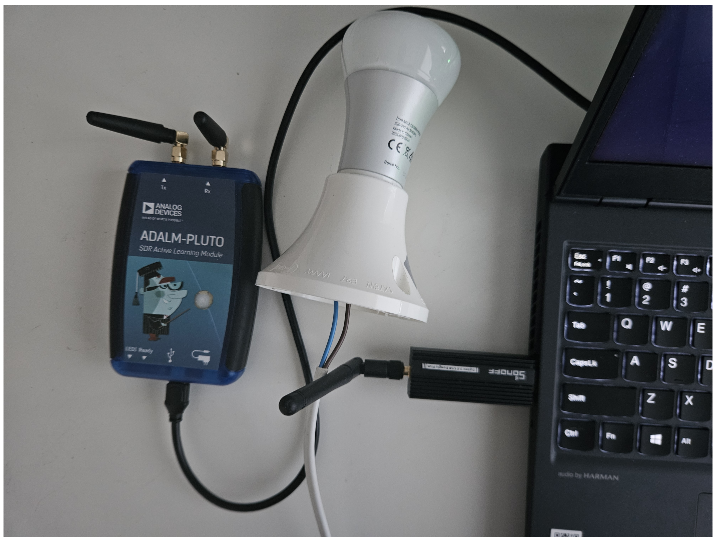


Figure 27: Scenario 2