JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Ing. Thomas Christof BSc

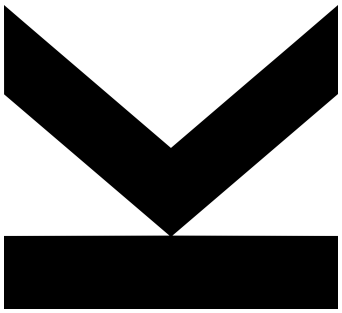
Submitted at
Institute of Networks and Security

Thesis Supervisor
Univ.-Prof. DI Dr. René Mayrhofer

Assistant Thesis Supervisor
Dr. Michael Roland

November 2021

DJI Wi-Fi Protocol Reverse Engineering



Master Thesis

to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

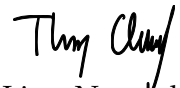
Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.



Linz, November 2021

Abstract

This master's thesis engages the reverse-engineering of the Da Jian Innovation low level Wi-Fi protocol. With deductive reasoning we try to establish logical connections between drone control instructions and their corresponding sent network packets. We further cluster UDP packets based on their payload length and execute bit-precise reasoning for payloads of interest. We unveil the protocol's core structure which enables pixel-perfect camera-feed and telemetry data extraction. Finally, we introduce a proprietary software solution to capture, analyse and post-process drone operation relevant network packets.

Kurzfassung

Diese Masterarbeit beschäftigt sich mit der Rekonstruktion des Da Jian Innovation Wi-Fi Protokolls. Mit deduktivem Vorgehen wird versucht logische Verbindungen zwischen den Instruktionsbefehlen der Drohne und deren resultierenden Netzwerkpaketen herzustellen. Weiteres werden relevante UDP Pakete anhand ihrer Länge gruppiert und Bit für Bit analysiert. Wir präsentieren die Struktur des Kernprotokolls welches die Extraktion einer pixelgenauen Kamera-Bildübertragung und der Telemetriedaten ermöglicht. Schließlich stellen wir eine proprietäre Softwarelösung vor, mit der steuerungsrelevante Netzwerkpakete gesammelt, analysiert und weiterverarbeitet werden.

Contents

Abstract ii

Kurzfassung ii

1 Introduction 1
- 1.1 Unmanned Aerial Vehicle . . . 1
- 1.2 Civilian Unmanned Systems . . . 2
- 1.3 Da Jian Innovation . . . 2
- 1.4 Wireless Communication . . . 3
- 1.5 Thesis Objective . . . 5

2 Target Specification 6
- 2.1 Software Defined Radio . . . 8
- 2.2 Ocusync . . . 9
- 2.3 Wi-Fi . . . 10

3 Man-in-the-Middle 12
- 3.1 Access Point Configuration . . . 13
- 3.2 Unstable Network Connection . . . 15
  - 3.2.1 Wi-Fi Extender . . . 15
  - 3.2.2 External Network Adapter . . . 16
- 3.3 Intercommunication Eavesdropping . . . 18

4 DJI Wi-Fi Protocol 19
- 4.1 Ethernet Frame . . . 20
- 4.2 Payload Encryption . . . 21

4.3 Deductive Reasoning . . . 24
4.3.1 Operator-to-Drone . . . 24
4.3.2 Drone-to-Operator . . . 37
4.3.3 Concluding Findings . . . 41
4.4 DJI Universal Markup Language . . . 42
4.4.1 Wireshark Dissector . . . 43
4.4.2 Payload Delimiter . . . 44
4.4.3 Protocol Architecture . . . 45

5 DJI Wi-Fi Tools 52
5.1 Framework and Libraries . . . 53
5.2 Application Architecture . . . 54
5.2.1 DJI Network Packets . . . 55
5.2.2 DJI Network . . . 57
5.2.3 DJI Camera . . . 59
5.2.4 DJI UI . . . 60
5.3 Information Gathering . . . 63

6 Conclusion 65

Bibliography 66

A Software Design UML Diagrams 72

B DJI Source Code Snippets 75

1 Introduction

Drones are of high popularity among private individuals and public institutions. Its wide variety of application pushes its demand of affordable and easy to use systems, satisfied by Da Jian Innovation (cf. Section 1.3), one of the major drone manufacturers. Their drone market dominance derives not only from their technical advance, but also from the mobile-applications ease of use. Despite the focus on usability, the applications are suitable for beginners and technically advised operators, as our personal experience throughout the entire reverse-engineering process denotes. However, as usability often contradicts security, we are curious what kind of information and in which granularity is extractable from a mid-air drone operation by unauthorised third party individuals.

1.1 Unmanned Aerial Vehicle

Unmanned Arial Vehicles (UAV), also known as drones, are remotely piloted aircrafts. Dalmagkidis et al. [4] classify UAVs based on various sources, distinguishing between: ownership, autonomy, altitude and mid-air collision risk, ground impact (i.e. expected kinetic energy at ground impact), and military classifications. One major drawback of this classification is its specificness in certain fields of operations. Accordingly, we select a more generalized alternative instead [58]: fixed and rotary winged. Fixed winged UAVs operate at a high velocity speed and carry heavy payloads. A continuous forward motion is required to stay airborne. Rotary winged UAVs (e.g. quadcopters) are suitable for stationary operations while providing the flexibility to move in any direction.

1.2 Civilian Unmanned Systems

Early UAV technology was not reliable, expensive and has been adopted very slowly in Europe. However, within the last decades, the rotary winged quadcopter grew of public interest among Civilian Unmanned Systems (CUS) as technology has been improved and became much more reliable [50]. Besides hobby pilots shooting pictures of structures, nature or sports events, CUS applications include [50]: scientific research, search and rescue, emergency response, traffic control tasks, infrastructure support, aerial photography, forest protection and wildfire monitoring, electrical facility monitoring, etc. Its variety of applications led to the establishment of small and medium enterprises, predominantly selling low-cost systems for civilian applications. In order to keep the development and production cost at a minimum, some major companies avoid proprietary hardware and leverage existing off-the-shelf items instead (cf. Section 2.2).

1.3 Da Jian Innovation

In 2015, five out of the top ten quadcopter drone manufacturers (cf. Table 1.1) are Chinese companies, mostly imitating existing products and selling at a lower price [57]. One of them, Da Jian Innovation (DJI), holds over 70% [57] of the global market share. Unlike its Chinese competitors, DJI did not start their business with imitation, but with their own research and technology. According to State Intellectual Property Office of P.R.C. (SIPO), DJI submitted 679 patent applications by December 2015 [57], leaving little to no chance for other companies to compete. Their technological advance solved many modern problems, such as obstacle avoidance, object tracking, prevention of image blurs at high-velocity speeds, remote photo and video access, and self-aware return to takeoff or operator location. Its superior technology shows in sales: With the release of the Phantom in 2013, sales reached USD 130 million and USD 500 million in the following year [57]. Predictions estimated more than USD 1 billion in 2015 and more than USD 1.5 billion in 2016 [57]. As of 2021, eight out of ten hobby UAV pilots own a DJI product. Therefore, while choosing our reverse-engineering target, we want to select a non-domestic drone manufacturer

Table 1.1: Global drone company market share ranking in 2015 [57]

Ranking	Name	Establishment	Country
1	DJI	2006	China
2	GoPro	2002	USA
3	Robotics	2009	USA
4	Parrot	1994	France
5	Zerotech	2007	China
6	AscTec	2002	Germany
7	Xaircraft	2007	China
8	Microdrones	2005	Germany
9	PowerVision	2012	China
10	Beihang University's Institute	1964	China

with the highest market share and units sold, in order to maximize any security vulnerability impact in a real-world scenario. Thus, we decided to focus on one of the most famous DJI families, the DJI mavic series.

1.4 Wireless Communication

Mid-air drone operations presume a radio link between the drone and the operator. Compared to a wired network scenario, radio links do not require attackers to compromise a physical system, nor to gain access to a wired ether network itself. Wireless communication is fairly easy interceptable within the infrastructural range, or by using high sensitive directional antennas [15]. Attackers are potentially capable of passive eavesdropping, message modification, replay attacks, or masquerading [15]. In order to comply to the same security standard of a wired connection, wireless communication protocols are required to implement the *CIA* triad [46]:

- *Confidentiality:* As passive eavesdropping is unavoidable, message encryption confines data access to only authorized parties. Based on the cryptosystem

and its corresponding cipher, not even traffic analysis will lead to any in-depth knowledge about the data itself.

- *Integrity:* prevents improper information modification or destruction by unauthorized third parties, specifically including authenticity and non-repudiation. *Authenticity* implies a successful authentication in first place and denotes being original and genuine, leading to *non-repudiation*, the inability to defy communication. Hence, in case of broken confidentiality, integrity still guarantees the ability to verify the legitimacy of a received message.

- *Availability:* ensures a reliable access to a network and its resources. The related term *usability* is tightly coupled, as access to business-critical information has a direct impact on productivity. *"Usability is a battle between security and productivity, as security measures can neither be so restrictive that they affect business processes and the flow of information, nor too relaxed, thereby causing harm"* (Samonas et al. [46]). Recompiling the statement for the UAV field of application, productivity represents the operator's ability to send and receive live status information, cohering with mid-air flight safety.

Drones require a reliable, high-throughput wireless communication for a live camera feed and remote command execution. Moreover, as drones operate in civilian airspace, it is important to ensure a certain level of flight safety, such that operating a drone will not violate local regulations, nor interfere with other aerial vehicles. As an example, in Austria in the year of 2016, a collision between a drone and an emergency helicopter (carrying a car crash victim to the hospital) could be avoided only in last second [24]. The helicopter pilot dodged the drone at a flight velocity of 250 km/h at 1500 meters altitude. Drones are a risk to human life, either by a direct collision with other aircrafts or to non-involved people on the ground [54]. Therefore, regardless of its security, a highly available and reliable wireless connection is a necessity to avoid potential air disasters. Drone manufacturers face the challenge to find the optimum balance between fly safety, usability, and network security.

1.5 Thesis Objective

Even though drones offer a wide variety of potential attack surfaces (e.g. GPS spoofing [53], Wi-Fi deauthentication or a distributed denial-of-service attack [27]) reverse-engineering and understanding the wireless communication protocol yields the potential to observe and monitor foreign-owned drones by unauthorized third party individuals. Furthermore, based on our findings, a future attack surface might comprise the development of proprietary software, capable of overtaking a mid-air operating drone, by replacing its ground station communication counterpart. As a proof of concept we try to obtain the following information from a passive observation:

- *UAV status:* Mid-air or grounded
- *Battery:* Power-level and remaining flight time expectation
- *Telemetry:* UAV altitude and velocity
- *Camera:* Live-feed images

We focus on the Wi-Fi mode of operation (cf. Section 2.3) and setup an artificial man-in-the-middle network scenario (cf. Section 3) to passively eavesdrop the intercommunication between the operator and the drone. Via deductive reasoning (cf. Section 4.3) we try to establish logical connections between drone control instructions and their sent network packets. In order to obtain in-depth knowledge about the protocol structure itself, we exert bit-precise reasoning (cf. Section 4.3.1.1) which enables pixel-perfect camera-feed image extraction and video streaming. As a final step we utilise pre-existing dissectors (cf. Section 4.4.1) to unveil the communication core protocol (cf. Section 4.4.3) and exemplify telemetry data extraction. With existing network traffic inspection tools not satisfying our needs, we develop a proprietary software solution (cf. Chapter 5) to enable live monitoring, flight simulation, bitwise network packet comparision, and fully automated post-processing for dissector compatibility. The thesis excludes the simulation of a ground station and does not cover mid-air communication hijacking.

2 Target Specification

The *Institute of Networks and Security* at JKU provided us a DJI Mavic Pro 1 (MP1) with following specifications:

- *Release:* 2016
- *Generation:* 3
- *Firmware:* 01.03.1000
- *App:* DJI Go 4 version 4.1.42

In order to operate the drone, a mobile application, which is available for iOS and Android, may be used. Additionally, DJI offers several Software Development Kits (SDK) to enable custom software solutions and task automation:

- The *mobile* SDK [7] supports iOS (9.0+) and Android (5.0+) as a target platform. Besides the basic flight control functionality, it supports telemetry sensor data extraction, obstacle avoidance, camera and gimbal control, live-video streaming, remote storage access, traversal of a pre-defined path, and various drone and remote controller state information retrieval. The SDK either supplements the DJI default mobile application, or represents a full replacement for more complex scenarios.
- The *User Experience* (UX) SDK [10] complements the mobile SDK by providing default user interface (UI) controls. This speeds up the development process and reduces the lines-of-code and complexity throughout custom applications.

- Complementary to the mobile SDK, *the Windows* SDK [11] facilitates the same feature-set on a Windows runtime-environment. The Windows Universal application (Creators Update 1.0; Build 16299) runs on Desktop, mobile and XBOX.
- Compared to other SDKs, the *Payload* SDK [9] is not publicly available and its access must be requested via the DJI developer portal. It offers, besides the on-board sensor data extraction API, the option to mount, operate, and integrate custom hardware components. Therefore, DJI not only covers daily use-cases, but also edge scenarios requiring special hardware equipment.
- The *Onboard* SDK [8] (OSDK) runs on embedded systems and Linux machines, as it not only aims to execute complex and high computation based tasks (e.g. object recognition), but also to run on custom hardware configurations, directly equipped to the drone itself or to a secondary ground-station. This SDK allows an autonomous mode of operation and covers every imaginable use-case scenario.

One limitation of the MP1 is its only support for the *mobile* and *UX* SDKs, due to its lack of compatibility for custom hardware mounting. Thus, for our research, we have the option to analyse the mobile application as well as the mobile and UX SDKs as a last resort. Independently of the drone's ground-station counterpart (native or custom mobile application), three communication channels exist in order to operate the drone: serial, software defined radio and Wi-Fi. As serial communication presupposes a wired connection, which is unfavourable while flying and indicates the permission to physically access the drone, it contradicts our final goal – unauthorized data extraction. Therefore, candidates for our reverse-engineering process and final target goal are the software defined radio and Wi-Fi modes only (cf. Table 2.1).

2.1 Software Defined Radio

Software Defined Radio (SDR) represents an umbrella term, as it leaves space for interpretation and implementation details. Other varieties of related terms include Software Based Radio [48] or Flexible Architecture Radio [21]. However, the main purpose throughout various interpretations and related terms is one specific requirement: Adjusting the waveform-output via software adaptation, instead of hardware redesign. The optimal goal is to communicate on any desired frequency, bandwidth, modulation and different data-transmission rates, simply by loading the appropriate software module [49]. SDR has a more practical interpretation by implementing the waveform mostly in software, comprising [19]:

- A *multi-band* system which is supporting more than one frequency band.
- A *multi-standard* system that is supporting more than one standard. Multi-standard systems can work within one standard family or across different networks.
- A *multi-service* system which provides different services, e.g. video stream, command stream, heartbeat stream, etc.
- A *multi-channel* system that supports two or more independent transmission and reception channels at the same time.

The motivation behind designing a proprietary SDR protocol, compared to the usage of an existing communication standard, yields some advantages. As communication standards evolve rapidly, backwards-compatibility on newer hardware may exceed previous hardware-specifications or newly introduced state-regulations. Moreover, long-term projects likely require certain hardware updates throughout the years while guaranteeing the same mode of operation [49]. SDR enables the required flexibility to span its functionality across different hardware configurations and software iterations.

DJI's SDR implementation offers three different modes of operation: Federal Communication Commission (FCC) for drone operations within the US air space, State

Radio Regulation of China (SRRC), and European Conformity (CE) for drone operations within the EU air space. Based on the chosen mode and frequency (cf. Table 2.1) the drone's maximum range varies. Presupposed the frequency 2.4 GHz and flying in an area free of interference, the maximum operation range settles between 4 and 7 kilometres. As DJI does not offer any 5 GHz range information within their product specification, only assumptions can be made, since the drone's operating mode within Europe is software-locked to comply to local regulations.

Table 2.1: DJI Mavic Pro 1 wireless transmission specifications [6]

Operation Mode	Frequency [GHz]	Trans. Power [dBm]	Range [km]
FCC	2.4 – 2.48	≤ 26	≤ 7
	5.15 – 5.25	≤ 23	n.a.
	5.72 – 5.85	≤ 23	n.a.
CE	2.4 – 2.48	≤ 20	≤ 4
	5.72 – 5.85	≤ 13	n.a.
SRRC	2.4 – 2.48	≤ 20	≤ 4
	5.72 – 5.85	≤ 23	n.a.
WiFi	2.4 – 2.49	≤ 20	≤ 0.08
	5.47 – 5.72	≤ 30	≤ 0.08

2.2 Ocusync

DJI's proprietary SDR protocol is called Ocusync [12, 13] and has been introduced with the MP1. It is part of the predecessor's Lightbridge family and allows up to 7 kilometers high definition video streaming [12]. As Lightbridge was originally a designed field programmable gate array [12], the hardware and software development was quite expensive, convincing DJI to cancel their custom silicon production. Contrarily, Ocusync uses generic "off-the-shelf" Wi-Fi hardware, while realizing their proprietary SDR protocol entirely in software [12]. As most smartphones already ship with a radio transmission hardware onboard, DJI was capable of removing

their custom silicons from the drone's remote control, leveraging existing hardware. Besides reducing development and maintenance costs, DJI is capable of introducing new feature and software updates anytime, without being limited by pre-shipped hardware specifications.

Ocusync comprises *multi-band*, *multi-service* and *multi-channel* in order to deliver the best stability and data-throughput. Its *multi-service* system provides a video-, control- and telemetry-signal [52]. Encoded with orthogonal frequency-division multiplexing (OFDM), the video-signal endures packet-loss by interference or attenuation, delivering acceptable results over large distances. Additionally, upon a certain interference threshold, the protocol automatically switches to a less occupied channel. Hence, the video-signal only changes channels if required, while the control- and telemetry-signal uses frequency hopping spread spectrum (FHSS). Packets are sent on random, regularly changing frequencies, adding tolerance for packet-loss [12].

2.3 Wi-Fi

Additionally to SDR, the MP1 is capable of operating in Wi-Fi mode, without the need of an extra remote-control counterpart. Wi-Fi mode facilitates short flight operations (cf. Table 2.1) whereas tedious setup and pairing can be avoided, by a peer-to-peer Wi-Fi connection between the smartphone and the drone, ideally for spontaneous camera-shots and recordings. However, this convenience factor introduces several disadvantages over the SDR mode, as the wireless encrypted communication protocol follows a public standard, unsuitable for a high-throughput in a one-way communication scenario.

Encrypted with WPA2-PSK in CCMP mode (AES in counter mode and CBC MAC for integrity checks [36]) a four-way connection handshake [23] is required in order to establish or re-establish (after losing the connection to the drone due to environmental signal obstruction) a secure channel between both peers. Each sent packet equals 128 bytes and needs to be confirmed by a two-way handshake, not only confirming the packet receival, but also its integrity status. As the Wi-Fi protocol

guarantees reliability, a lost or damaged packet requires a re-transmission, easily caused by signal attenuation or interference.

Besides the high-latency and zero transmission error tolerance, WPA2 is prone to several security threats, including: Authentication, association, deauthentication, and disassociation request flooding or a distributed denial of service attack [27]. As Wi-Fi does not require any special transmitter hardware and several tools are publicly available for exploitation, the Wi-Fi mode offers an easier and wider attack-surface compared to the proprietary SDR protocol. However, as we do not target to reverse-engineer the communication's cipher algorithm nor to takeover the communication itself, weaknesses are more likely to facilitate our reverse-engineering process. Furthermore, following arguments support our reasoning of choosing the Wi-Fi protocol as our reverse-engineering target:

- *Encryption:* We assume a secure communication between the drone and its operator. Due to the proprietary Ocusync protocol, deciphering may be a challenge by its own, as packets arrive on random frequencies, potentially out-of-order and may include corrupt packets, on top of being encrypted by an unknown cipher. Wi-Fi, on the other hand, guarantees an in-order, integrity checked and decrypted stream of data, as we are capable of performing a man-in-the-middle attack.
- *Transmitter:* Unlike Wi-Fi, sending and receiving SDR signals requires special-purpose hardware. While a transmitter is available at the university, access is quite tedious during the global Covid-19 pandemic. Moreover, flying the drone and monitoring the transceiver in-door bears yet another challenge.
- *Relevance:* The MP1's Ocusync protocol is obsolete, as its successor MP2 has been shipped with Ocusync 2.0. As of 2021, drones ship with Ocusync 3.0. In other words, potential findings within Ocusync 1.0 are of questionable relevance nowadays.
- *Compatibility:* The SDR and Wi-Fi protocol may comprise a slightly different implementation. However, we assume some kind of consistency and similarities throughout both implementations as both modes support the same feature-set.

3 Man-in-the-Middle

A Wi-Fi connection between the operator and its drone fulfils the CIA triad (cf. Section 1.4) by using the Wireless Protected Authentication (WPA2) protocol. The operator authenticates via a Pre-Shared Key (PSK) [26] and further encrypts the traffic with the Advanced Encryption Standard (AES) in Counter Mode (CTR). To ensure integrity and authentication next to confidentiality, AES in Cipher Block Chaining (CBC) adds a Message Authentication Code (MAC) [44] – both in combination: Counter with CBC-MAC (CCM) mode Protocol (CCMP) [36]. In conclusion, passive eavesdropping the wireless intercommunication will not yield any reverse-engineerable result. As we do not focus on WPA2-PSK security vulnerabilities (e.g. key recovery [26]) we instead setup an artificial, eavesdrop-friendly network configuration, such that the wireless intercommunication is inspectable and interceptable in a decrypted manner. For that reason we setup a Man-in-the-Middle (MITM) network scenario (cf. Figure 3.1):

(a) The *Drone* establishes an Access Point (AP) after its initialization phase, usually used for a direct connection between both network peers.

(b) As our *MITM-Machine* does not feature two network interface cards (NIC), we do require to setup a second virtual network interface[1] acting as an AP *(MITM-AP)* for our mobile device.

(c) The *MITM-Client* directly connects to the drone, forwarding the *MITM-AP's* network packets to the drone and vice versa.

(d) Our *MITM-Machine* represents the eavesdropping device, connecting via Network Address Translation (NAT) the virtual *MITM-AP's* with the internal NIC

[1] [Online; accessed 14-July-2021] `https://github.com/oblique/create_ap`

(MITM-Client). The favourable *bridge* network aggregation mode is, due to hardware limitations (cf. Section 3.1), unavailable. As our *MITM-Machine* physically owns the *MITM-AP*, we can passively eavesdrop all unencrypted packets between the *MITM-AP* and the *MITM-Client*.

(e) The *Operator* represents either an iOS or Android device, running the DJI Go application version 4 (cf. Chapter 2) without any hardware nor software modifications. The mobile device connects via Wi-Fi to our drone representative *(MITM-AP)*.

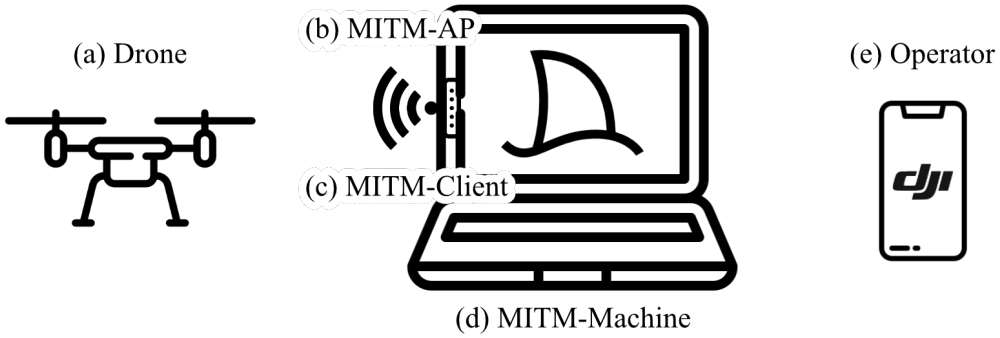


Figure 3.1: MITM network configuration setup

3.1 Access Point Configuration

The Jumper EZbook X3 Air, our *MITM-Machine*, ships with the Intel Dual Band Wireless-AC 3165 [17] wireless chipset. Besides missing a second NIC (favourable for a MITM scenario) the chipset features only one antenna for sending and receiving data, while further sharing its available bandwidth with active Bluetooth connections [17]. With those limitations in mind, we create a virtual *MITM-AP* over the Command Line Interface (CLI) (cf. Listing 3.1).

Listing 3.1: Virtual Access Point creation via Command Line Interface

```
1 sudo create_ap wlp1s0 wlp1s0 Operator 12345678 -w 2 -c 2
          -m nat --freq-band 2.4 --country AT
```

In order to eliminate as many disturbance factors as possible, we once connect our *Operator* directly to the *Drone's AP*, disable the 5 GHz frequency band and pre-configure an arbitrary but fixed unoccupied Wi-Fi channel within the DJI Go mobile application. Consequently, the `create_ap` (cf. Listing 3.1) command parameters mirror the *Drone's AP* configuration:

- `-w 2`: WPA2-PSK in CCMP cipher mode
- `-c 2`: Usage of an arbitrary but fixed unoccupied channel
- `-m nat`: NAT between the virtual AP and physical NIC
- `--freq-band 2.4`: Disable the 5 GHz frequency band
- `--country AT`: Comply to local regulations

Connecting the *Operator* to the *MITM-AP* launches the drone's UI and the camera stream starts immediately (cf. Figure 3.2). Hence, no additional security measures, for a MITM attack detection and prevention, have been considered.

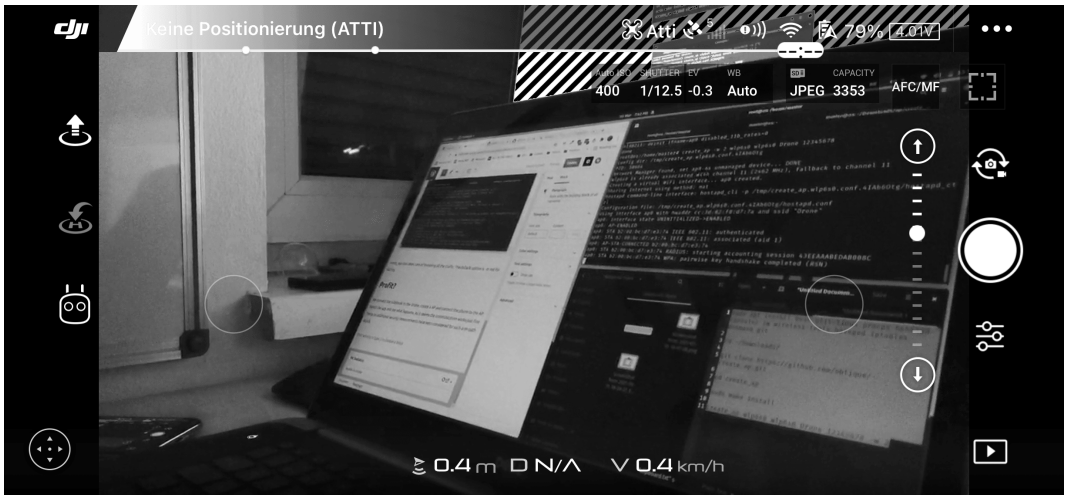


Figure 3.2: Operating the drone within the MITM network scenario

3.2 Unstable Network Connection

With the initial setup in place, we started a trip into a wide and empty area, ready to capture live packets between the operator and the drone. Besides the functional MITM scenario tests (obtaining the live camera feed) no additional effort has been made to determine any drawbacks or limits. Consequentially, our first "testflight" did not last long. After takeoff and hovering approximately 1.5 meters above the ground, the drone lost connection to our operator. As the connection did not recover, the drone initiated an emergency landing procedure, descending slowly to the ground. Various Wi-Fi channel tweaks had no different outcome on the experiment. Since the connection shows no degradation in a non-airborne scenario, we assume that the issue might be Wi-Fi signal strength or network throughput related; in regards to our notebook's wireless network chipset, both assumptions seem quite reasonable. For root cause determination we consider two initiatives: A Wi-Fi extender and an external network adapter.

3.2.1 Wi-Fi Extender

As the notebook's wireless network chipset shows strong signal attenuation in WLANs within a relatively small radius, an airborne drone might lead to an unstable or broken network connection, as obviously the drone will not operate within close proximity. Therefore, we consider a portable USB Wi-Fi extender for the *drone's AP*. As the operator always operates within close range, extending the *MITM-AP* would not yield a different result. The requirements for the Wi-Fi USB dongle comprises: WPA2-PSK support, multiple antennas for transmitting and receiving (minimum 2T2R) and a decent throughput to prevent network congestion. Such dongles are widely available[2] and fairly cheap.

The improvement partially confirmed our assumption, as the connection was not immediately degraded after takeoff. However, a noticeable unusual live camera feed delay started to occur, followed by several frame skips. Approximately after 30 seconds, the connection's stability started to throttle and the signal had been lost

[2][Online; accessed 17-July-2021] `https://www.amazon.de/gp/product/B07PRVVV29/`

completely. Even though the operator was able to re-connect to the drone once more, the delays aggregated up to a certain level whereas the drone did not even accept a landing command instruction anymore. We had no choice but to cut the drone's connection to initiate the emergency landing procedure once more. Improving the Wi-Fi signal did help in certain aspects, but the root cause seems to co-relate with the *MITM-Client's* throughput.

3.2.2 External Network Adapter

We started the experiment with certain hardware limitations in mind. With the notebook's internal NIC struggling to offer a virtual AP and a stable connection to the drone, an External Wi-Fi Network Adapter (EWNA) might facilitate the overall MITM network scenario data throughput. Equivalent to the Wi-Fi extender's requirements, the EWNA has to comprise at least the same hardware specifications. With a second installed NIC[3] we have several possible network configurations available to choose from, as both are candidates for the *MITM-AP* establishment. A benchmark tool[4] determines, based on various factors, the most suitable setup, while connecting the mobile phone to four different *MITM-AP* configurations (cf. Listing 3.2). The first `create_ap` command parameter represents the NIC used for the *MITM-AP*, while the second parameter defines the NAT target. To avoid arbitrary spikes falsifying the test result, each benchmark has been run three times, while ensuring the same environmental conditions.

Listing 3.2: Access Point configuration scenario bash commands

```
1 create_ap internal internal Operator 12345678 -c 2 -w 2
2 create_ap external internal Operator 12345678 -c 2 -w 2
3 create_ap internal external Operator 12345678 -c 2 -w 2
4 create_ap external external Operator 12345678 -c 2 -w 2
```

The mobile benchmark tool defines five quality attributes: down- and upload speed, ping, jitter, and packet loss. Even though a high data transmission rate is favourable,

[3] [Online; accessed 17-July-2021] `https://www.amazon.de/gp/product/B00LLIOT34/`
[4] [Online; accessed 17-July-2021] `https://apps.apple.com/us/app/speedtest-by-ookla/id300704847`

the drone operates on a pre-definable bandwidth between 1 Mbps and 4 Mbps. Transmission rates over 1 Mbps may facilitate the video stream quality, but had no further impact on the drone's operability. All *MITM-AP* configuration scenarios (cf. Table 3.1) feature more, or equal to, 1 Mbps. Hence, the original network instability derives from jitter and packet loss, reflected within the first scenario's test results. Scenario 3 offers the highest transmission rate, nevertheless including immense jitter and packet loss. Claypool and Tanner [3] investigated the impact of jitter and packet loss on perceptual video quality. Their study, incorporating over 40 participants, concluded a perceptual quality drop by more than 50% with low amounts of jitter and packet loss being introduced. Contrarily, scenario 2 comprises less jitter and almost no packet loss, although settling in-between scenarios 1 and 3 in terms of the transmission rate. The fairly acceptable mixture of all five quality attributes determines our final MITM network configuration setup. The *EWNA* (cf. Figure 3.3) replaces the virtual *MITM-AP* (cf. Figure 3.1), allowing the *MITM-Client* to maintain an exclusive connection to the *Drone*. Field tests have confirmed our presumptions with a stable and responsive connection, although the drone requires leastwise a 2 Mbps predefined bandwidth limitation.

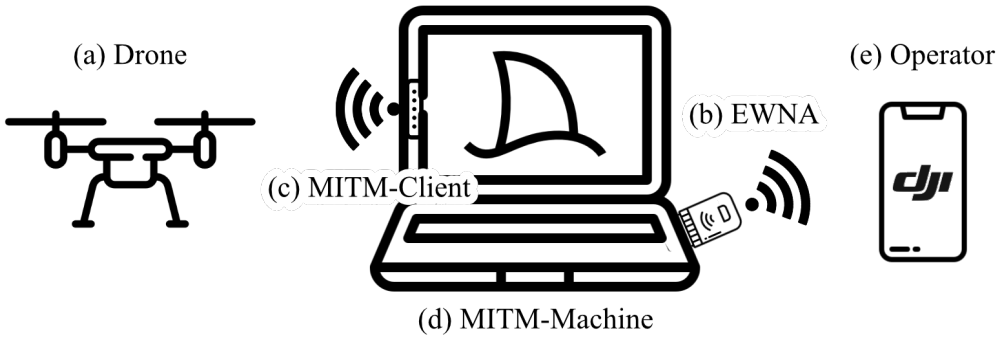


Figure 3.3: Improved MITM network configuration setup

Table 3.1: Access Point configuration scenario benchmarks

Scenario	↑ [Mbps]	↓ [Mbps]	Ping [ms]	Jitter [ms]	Packet loss [%]
1	1.0	3.4	84	0.2	28
	1.7	3.5	85	0.5	29
	1.0	3.4	99	0.7	28
2	3.9	6.9	11	4	0.4
	3.3	6.2	12	2	0.0
	3.2	8.9	11	2	3.9
3	12.1	9.4	9	66	7.2
	16.6	9.7	10	21	7.1
	13.0	8.5	11	50	7.7
4	NS				

3.3 Intercommunication Eavesdropping

Even though desktop applications exist to capture network packets, we instead use a CLI based approach to avoid any serious performance impact on our MITM network device. With `tcpdump` (cf. Listing 3.3) network packets captured at the *EWNA-AP* (cf. Figure 3.3) will be stored in a `pcap` file, readable and inspectable by various network analytic programs. Additional filters reduce the captured output to a bare minimum, only comprising relevant network packets between the operator and drone. In particular, `-n` avoids Domain Name Service (DNS) resolutions, `udp` ignores any TCP packet and `host 192.168.2.1` (cf. Section 5.3) drops any non-drone related packet, where neither the source nor destination equals the specified IP address.

Listing 3.3: Packet capture via Command Line Interface

```
1 sudo tcpdump --interface=ap0 -n udp host 192.168.2.1
    -w /tmp/dumps/capture.pcap
```

4 DJI Wi-Fi Protocol

Via deductive reasoning (cf. Section 4.3) we try reverse engineer the DJI Wi-Fi protocol by identifying logical connections between sent network packets and their corresponding operator input commands. For the exact protocol structure determination, we exert bit-precise reasoning to classify each individual bit. Wireshark is a free and open-source[1] network traffic and packet analysis application, offering live monitoring, in-depth packet inspection and a visual user interface; altogether the perfect software solution to build our strategy upon. Moreover, Wireshark supports to restore a previous session from a `.pcap` file, generated by the `tcpdump` command execution (cf. Section 3.3). In other words, previous session recordings do not require additional conversion or compatibility adjustments. Besides the visual network packet inspection, Wireshark offers to apply filter predicates, reducing – based on the current investigation task – thousands of network packets to a bare minimum. Throughout the entire investigation several useful filter predicates did facilitate our reverse-engineering process:

- `!dns && !mdns && !icmp`: `tcpdump` includes DNS, Internet Control Message Protocol (ICMP) and Multicast DNS (MDNS). The negation and conjunction of all three protocols remove their corresponding network packets, as only application level network packets are of relevance.
- `ip.src == 192.168.2.1`: Not only the source *(src)*, but also the destination *(dst)* may be filtered by IP address. I.e. `192.168.2.1` represents the drone (cf. Section 5.3), while some other Dynamic Host Configuration Protocol (DHCP) assigned IP address represents the operator.

[1] [Online; accessed 06-August-2021] `https://gitlab.com/wireshark/wireshark`

- `data.len == 1472`: With length constraints network packets may be exclusively in- or excluded from the result-set. In the provided example, only payloads of size `1472` bytes are of interest. Network packets, whereas the payload size equals, might share even more similarities and constitute to our very first deductive reasoning attempt: The presumption of a direct correlation between the payload size and its representing drone control instruction.
- `data.data[0] == 0x00`: Compares a static hex value against the payload content at the provided index. Mid-flight packet inspection focuses on co-related changes between the protocol and the operator's drone control instructions. Therefore, comparing bytes throughout several network packets may reveal valuable instruction indicators.

Predicate negation may comprise the replacement of the *equal* with the *not equal* comparator, although, the entire negation of the predicate is recommended [55]. I.e. `!(data.len == 1472)` yields all network packets with a data-length not equal to `1472`.

4.1 Ethernet Frame

Unlike our custom software solution (cf. Section 5), Wireshark visualizes the network packets corresponding to the Open Systems Interconnection (OSI) model. Layer 2, Layer 3 and Layer 4 usually carry bits of importance (cf. Table 4.1), but due to our filter predicates, they are not of any relevance. Consequently, we only focus on the datagram payload, henceforth referred to as payload. To be more precise, from byte index `0x2A` to Y, whereas Y represents the Ethernet frame length.

Table 4.1: Ethernet Frame OSI model layers

Layer	Begin [byte]	End [byte]	Length [bytes]	Contents*
2	0x00	0x0D	14	• Src MAC Address • Dst MAC Address
3	0x0E	0x21	20	• Src IP-Address • Dst IP-Address • Protocol-Type
4	0x22	0x29	8	• Src Port • Dst Port • Checksum

* Each layer does facilitate more information than provided within the table. The content has been reduced to our needs.

4.2 Payload Encryption

Even though the whole communication itself is encrypted, the decrypted stream of data might still comprise payload encryption. Such an additional security measures would render any passive-eavesdrop scenario meaningless, as the payload content itself would not yield any reverse-engineerable result in first place. For that reason, our first investigation task is to determine whether any kind of application layer encryption has been applied to the payload.

AES encryption, particularly in CBC, PTR, CCM or CCMP cipher mode, results in an arbitrary data stream, even if the input blocks comprise the same content. Contrarily, the Electronic Codebook Mode (ECB) generates a deterministic stream of data by applying the very same pseudorandom permutation to each plaintext block [20]. Therefore, repeated plaintext blocks will produce repeated ciphertext blocks, as the encryption process features a deterministic behaviour. In conclusion, we either encounter a pseudo-random payload throughout the whole session recording or some identical payloads. Our investigation data-set comprises more than 20.000 UDP packets, recorded over a 1-minute time period while providing zero drone

control instruction. No single payload duplication (drone-to-operator and operator-to-drone) has been detected. Either no duplicate packet has been sent, or ECB is not our candidate, presupposed any kind of encryption. For more detailed reasoning we compare two different payloads (cf. Figure 4.1, #6823 and #6824) with the same payload size of 56 bytes. Only 5 bytes at index `0x06`, `0x07`, `0x24`, `0x36` and `0x37` (indicated by red colour) vary. Most remarkably, at index `0x24`, the value increments by one, confirmed across the entire session recording. The payloads are too similar and even comprehend some sort of incrementation, all indicators for an unencrypted stream of data. To confirm our assumption, we additionally compare five other payloads (cf. Figure 4.2, #439, #440, #441, #442 and #443) with the same payload size of 33 bytes. Besides yet another incrementing byte at index `0x10` and `0x1A`, `0x04` alongside with `0x0A` increments by eight and reflect the same value. The value at `0x1D` either equals zero or six throughout the entire session, while `0x1E` to `0x20` seem to follow no obvious pattern. Even though we can not reason any incrementing or alternating behaviour yet, it is safe to assume that our provided example payloads do not facilitate any kind of encryption at first glance.

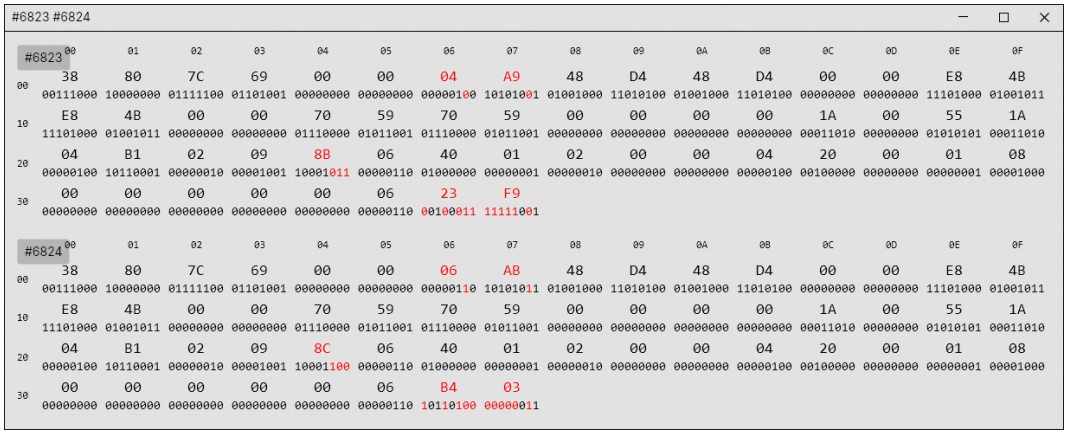


Figure 4.1: Binary comparison between 2 different payloads with a size of 38 bytes

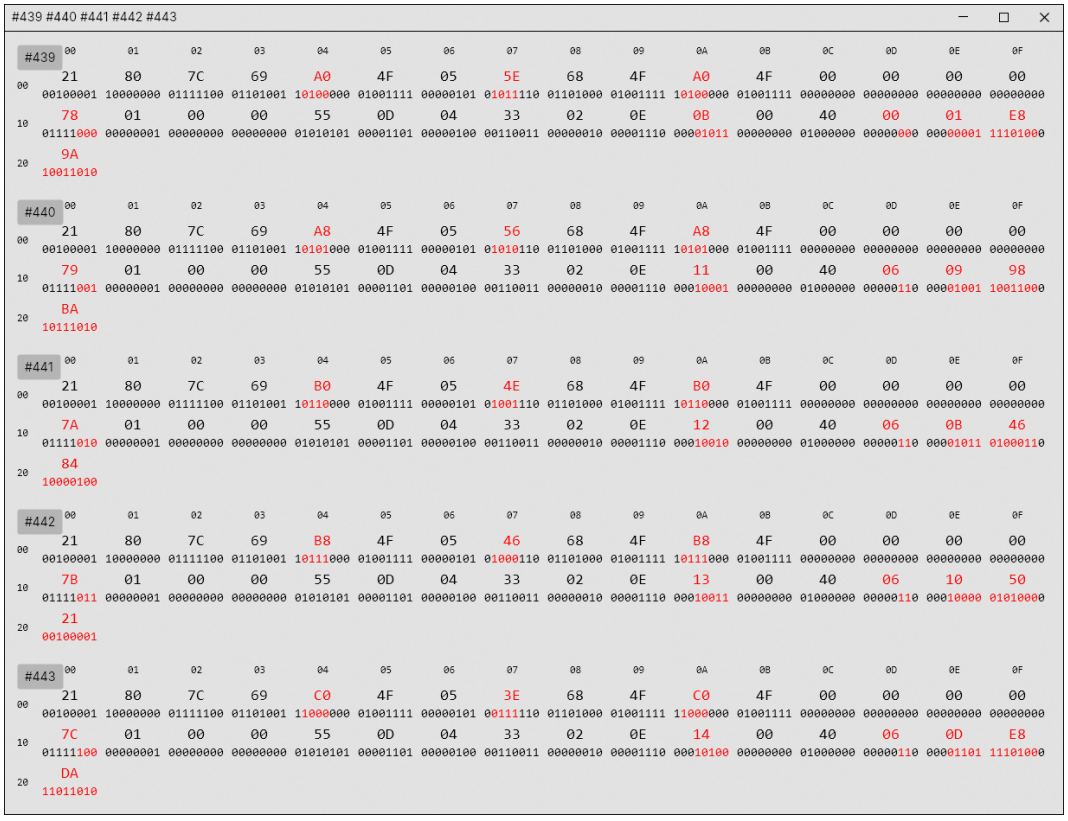


Figure 4.2: Binary comparison between 5 different payloads with a size of 33 bytes

4.3 Deductive Reasoning

Our first approach to reverse-engineer the entire DJI Wi-Fi-Protocol builds upon deductive reasoning, whereas we try to interpret datagrams based on correlating operator inputs. In other words, with direct drone control instructions (e.g. take-off, landing, accelerate, rotate, etc.) we force the operator to send the command's representative network packet. With only one type of control instruction per recording, we are capable of compiling a list of properties, comprising: command-type, datagram count, payload length and content. Subsequently, packets of interest will be compared bit by bit, revealing the command structure and its parameter values — in theory. First insights, during our payload encryption research, reviles a chatty protocol, predominated by drone-to-operator packets, fully utilizing the Maximum Transmission Unit (MTU). Consequently, we separate our reasoning into two unrelated phases: Operator-to-Drone and Drone-to-Operator phase. Analysing fewer and smaller payloads might lead to faster conclusions and further utilises the reasoning of even larger ones.

4.3.1 Operator-to-Drone

The selection of the first drone control instruction to be analysed is not quite obvious, due to the fact that no instruction payload size is known. Focusing on small payloads presupposes the awareness of all different kinds of payload sizes, including their corresponding drone control instruction. For that reason, we record five different drone control instruction scenarios, each with a total duration of 20 seconds. The instructions have been chosen semi-randomly, as we can only assume that basic input instructions might not comprise a lot of parameters. Our selection comprises (cf. Table 4.2):

- *Idle:* Providing zero control instructions while grounded.
- *Gimbal:* The camera's rotational orientation manipulation while airborne.
- *Rotation:* Clock- and counterclockwise rotational acceleration around its axis.
- *Altitude:* Increasing and decreasing its altitude level via vertical acceleration.

- *Velocity:* Horizontal acceleration on both axis.

With the idle scenario as a baseline, correlations between Ethernet frame occurrences and the payload size could reveal the control instruction's affiliation. Surprisingly, no significant correlation has been found between the instructions and payload sizes. Upon close inspection (cf. Table 4.2), the *Gimbal* column leads – in terms of packet occurrence – three times. Considering the low occurrence count, we can eliminate a possible correlation, as the gimbal's orientation has been manipulated throughout the entire 20 seconds course. Similar behaviours are perceivable throughout all other mid-air observations. What additionally stands out are the logical contradictions while comparing mid-air against the idle sample recordings. The drone in idle mode did neither rotate nor accelerate on any axis, but inheres packet occurrences close to all mid-air observations. As an example, the *Rotation* column is leading in 14 cases; none of them offers a zero cell entry within the *Idle* column. Logically, the drone can not rotate while being grounded; thus, should feature at least one zero cell entry within the *Idle* column, whereas the corresponding *Rotation* cell offers the most packet occurrences. The only exceptional and coherent correlation has been found within the *Velocity* column. Although 8 packet occurrences with a payload size of 76 bytes are questionable, we can not determine any other objection. In conclusion, we can not identify control instructions based on the payload size, as not only the peak occurrences are close to other observations, but also due to the fact of logical contradictions. The protocol might aggregate several instructions into one Ethernet frame, or the instruction's binary representation is of variable length. Another possible explanation comprises a steady stream of data, even if no input has been provided; reinforced by the insignificant occurrence difference between our *Idle* and mid-air (*Gimbal*, *Rotation*, *Alititude* and *Velocity*) sample recordings.

4.3.1.1 Bit-Precise Reasoning for Packets of length `0x38`

Without any pre-classification in place, we change directions and start to investigate payloads with 56 bytes of size, as we expect the most sent packet to be of equivalent importance. For simplicity reasons, we introduce clusters which aggregate payloads of identical size, enabling straightforward bit-precise reasoning without payload

Table 4.2: Idle and mid-flight Ethernet frame occurrences

Payload size [bytes]	Idle	Gimbal	Rotation	Altitude	Velocity
33	214	222	219	223	223
34	108	118	134	128	105
35	6	9	11	10	5
36	4	3	3	4	4
37	29	28	37	29	30
38	3	3	7	4	3
39	1	1	3	1	1
40	7	7	8	7	7
41	4	4	8	4	4
42	23	22	30	24	22
43	6	5	14	8	10
45	2	2	3	3	2
46	4	4	4	4	4
47	0	2	4	3	0
48	1	2	1	2	1
49	0	1	1	0	0
50	12	10	15	26	13
56	1412	1401	1385	1362	1341
58	4	1	10	10	8
60	1	1	2	7	1
61	1	2	2	1	0
62	0	1	0	4	0
76	0	0	0	0	8
77	0	1	1	1	0

length discrepancies. I.e. the cluster `0x38` represents all payloads with a size of 56 bytes. To avoid danger of confusion between hexadecimal numbers and their field of application throughout the thesis, we introduce further notations whereas `H` reflects an arbitrary hexadecimal number:

- `col(H)`: Represents clustered payloads with a size of `H`.
- `val(H)`: Portrays the numeric value of an array element at index `H`.
- `con(H)`: Describes the numeric constant value `H`.
- `idx(H)`: Refers to an array index at `H`.
- `len(H)`: `H` represents the length of an array.
- `col(`H_1`)[val(`H_2`)]`: Refers to the cluster H_1 and its numeric value at array index H_2. This notation additionally supports range parametrisation: `col(`H_1`)[val(`H_2`) - val(`H_3`)]` with H_2 and H_3 delimiting the range boundary.

Our bit-precise reasoning of `col(0x38)` does not exclude the analysis of any other clusters in parallel, as we do require to confirm or falsify certain findings across the entire recording. Nevertheless, we concentrate on the *Rotation* recording data-set (cf. Table 4.2), comprising 1385 entries, leading to a few findings (cf. Table 4.3):

`idx(0x00) - idx(0x01)`: An endless stream of data requires some sort of End of Message (EOM) identification. Otherwise, neither the drone nor the operator would be capable of composing valid messages from the stream. Such an EOM identifier may be implemented with a special purpose delimiter (e.g. null terminator) or with a prefixed message length definition. As no single payload starts nor ends with a special purpose delimiter, the size information has to be encoded within the payload itself. Unsurprisingly, the payload's first byte reflects the same value as our cluster definition. I.e. the cluster `col(0x38)` prefixes every payload with `con(0x38)`. Besides the payload size indicator at `idx(0x00)`, we originally thought of a protocol version identifier at `idx(0x01)`, as the value, throughout all recordings, equals to `con(0x80)`. Even though a protocol version identifier is not a requirement per se, it would enable the possibility to connect and operate various drone firmware or

operator application versions concurrently. However, the payload size of operator-to-drone packets did never exceed 255 bytes, representable within `idx(0x00)`. On the other hand, the drone-to-operator packets exceed the 255 byte size limitation by predominantly sending payloads up to the MTU. Obviously, one byte is not capable of mapping a payload size of 1472 bytes. At this point, we started to reconsider the protocol version identifier's value. Investigating drone-to-operator packets only, the wrongly assumed protocol version identifier proved to correlate with the payload's size (cf. Table 4.4). The final formula to extract the payload size from `idx(0x00)` and `idx(0x01)` is defined by: `((val(0x01) & con(0x0F)) << 8) + val(0x00)`.

`idx(0x02)` - `idx(0x03)`: Presupposed an operational MITM network configuration setup (cf. Figure 3.3), our sample recordings (cf. Table 4.2) comply to a strict procedure: (i) start a new instance of `tcpdump` (cf. Listing 3.3), (ii) launch the mobile application (cf. Chapter 2) and generate network traffic, (iii) and close the mobile application to enforce a broken and unrecoverable connection state. Contrarily to an unstable or recoverable connection status, a broken connection discards any previously set configuration and state information, imposing a cold start after application relaunch, perceivable at `idx(0x02)` and `idx(0x03)`. In other words, `idx(0x02)` and `idx(0x03)` seem to adhere an arbitrary but fixed session identifier within the same recording. Behind the scene, we assume a random value generation as we were not capable of deducting a deterministic behaviour.

`idx(0x0A)` - `idx(0x0B)`: Unfortunately, neither their purpose nor seemingly random change behaviour lead to any conclusion. Nevertheless, many other bytes (cf. Table 4.3) seem to mirror their state: `idx(0x08)`, `idx(0x09)`, `idx(0x0E)`, `idx(0x0F)`, `idx(0x10)`, `idx(0x11)`, `idx(0x14)`, `idx(0x15)`, `idx(0x16)`, and `idx(0x17)`. The operator obtains its initial value from the very first drone-to-operator packet (cf. Section 4.3.1.2) followed by occasional changes, triggered by arbitrary drone-to-operator messages.

`idx(0x24)` - `idx(0x25)`: As our observations suggest, a payload may contain several exclusive or non-exclusive packet-counters, distinguishable by their incremental behaviour. A non-exclusive packet counter comprehends the entire clusters, whereas an exclusive packet counter is bound onto one specific cluster only. The execution order of certain commands might be of relevance, considering a certain level of

Table 4.3: `col(0x38)` bit-precise reasoning

Offset [byte]	Size [bits]	Function	Description
`0x00 - 0x01`	16	Payload Length	`val(0x01) & 15 << 8 + val(0x00)`
`0x02 - 0x03`	16	Session Identifier	`col(0x30)[val(0x02) - val(0x03)]`
`0x04 - 0x05`	16	Padding	Zero bits
`0x06 - 0x07`	16	Unknown	
`0x08`	8	Mirror	`val(0x0A)`
`0x09`	8	Mirror	`val(0x0B)`
`0x0A - 0x0B`	16	Unknown	`col(0x30)[val(0x08) - val(0x09)]`
`0x0C - 0x0D`	16	Padding	Zero bits
`0x0E`	8	Unknown	Initial value `val(0x0A)`
`0x0F`	8	Unknown	Initial value `val(0x0B)`
`0x10`	8	Unknown	Initial value `val(0x0A)`
`0x11`	8	Unknown	Initial value `val(0x0B)`
`0x12 - 0x13`	16	Padding	Zero bits
`0x14`	8	Unknown	Initial value `val(0x0A)`
`0x15`	8	Unknown	Initial value `val(0x0B)`
`0x16`	8	Unknown	Initial value `val(0x0A)`
`0x17`	8	Unknown	Initial value `val(0x0B)`
`0x18 - 0x23`	96	Unknown	Constants
`0x24 - 0x25`	16	Packet counter	Little-Endian
`0x26 - 0x2D`	64	Unknown	Constants
`0x3E - 0x3F`	16	Rotation	`con(0x0D) con(0x05)` – Left `con(0xF3) con(0x0A)` – Right `con(0x01) con(0x08)` – Idle
`0x30 - 0x34`	40	Padding	Zero bits
`0x35`	8	Unknown	
`0x36 - 0x37`	16	CRC-16 Kermit	`val(0x24) - val(0x35)`

Table 4.4: `idx(0x01)` payload size decoding

From [bytes]	To [bytes]	Value [byte]
0	255	`0x80`
256	511	`0x81`
512	767	`0x82`
768	1023	`0x83`
1024	1279	`0x84`
1280	1535	`0x85`

security in mid-flight scenarios. Composed by two bytes, encoded in little-endian byte order, a non-exclusive packet counter is located at `idx(0x24)` and `idx(0x25)`, enabling sequential command execution.

`idx(0x3E)` - `idx(0x3F)`: Our hypothesis (the most sent packet being of equivalent importance) confirms with `idx(0x3E)` and `idx(0x3F)` being correlated to rotational operator requests:

- `idx(0x3E)` = `con(0x0D)` ; `idx(0x3F)` = `con(0x05)` — Counterclockwise
- `idx(0x3E)` = `con(0xF3)` ; `idx(0x3F)` = `con(0x0A)` — Clockwise
- `idx(0x3E)` = `con(0x01)` ; `idx(0x3F)` = `con(0x08)` — Idle

Our *Rotation* session recording does not comprise velocity or altitude related entries, such that we can not conclude whether `idx(0x3E)` and `idx(0x3F)` includes additional parameters. Although the rotational parameter values are consistent throughout the entire session, we suspect a different value occurrence for future recordings, as `idx(0x3E)` and `idx(0x3F)` could be the result of a calibration process. However, counterclockwise and clockwise exhibit following invertible relation:

- `idx(0x3E)`= (`con(0x0D)` & `con(0xFE)`) = $\sim$(`con(0xF3)` & `con(0xFE)`)
- `idx(0x3F)`= (`con(0x05)` & `con(0x0F)`) = $\sim$(`con(0x0A)` & `con(0x0F)`)

`idx(0x36)` - `idx(0x37)`: UDP requires neither reliability nor in-order packet delivery, consequently preferring data-throughput over packet-loss without any congestion control in place [42], suitable for wireless drone operations. In addition, UDP offers an optional Cyclic Redundancy Check (CRC) within its header (cf. Table 4.1) covering the pseudo-IP header, UDP header and payload [42]. With a non-zero populated checksum field within the datagram, we did not expect the presence of an additional CRC checksum. However, unique values per message at `idx(0x36)` and `idx(0x37)`, regardless of predominant payload congruency (cf. Figure 4.3), forced us to evaluate a CRC checksum appendix candidate. With two bytes serving as a CRC value, we focus on CRC-16 algorithms and utilised an online CRC calculation tool [28] to obtain further details. No match across all 23 available algorithms suggests either a custom CRC algorithm or the usage of non-standard parametrisation of the polynomial function, initial or final xor value. Too many variables within the equation leaves us no choice but to execute a brute-force[2] against several payloads (cf. Listing 4.1). Brute-force attempts on individual messages were successful, but non-identical throughout several messages. Hence, the CRC checksum does not comprise the entire payload. Systematically in- and excluding sub-sets of the payload within the online tool [28] exposed the payload's CRC checksum algorithm and coverage: CRC-16 Kermit in little-endian Byte order, spanning `idx(0x24)` - `idx(0x35)`. Retrospectively, the checksum's payload coverage spans the most important bytes, including the packet-counter and the rotational input parameters. Moreover, the CRC checksum appendix is our first indicator of a common base between the Wi-Fi and the Ocusync protocol, as the SDR is likely to lack the UDP header alongside its checksum.

Listing 4.1: CRC-16 parameter value brute-forcing

```
1 echo "3880ec2a...1c00400102...00061619" >> data.txt
2 echo "3880ec2a...1d00400102...0006f1e1" >> data.txt
3 echo "3880ec2a...1f00400102...00062e18" >> data.txt
4
5 ./bruteforce-crc --verbose 1 --file data.txt --start 0
     --end 49 --width 10 --offs-crc 49
```

[2][Online; accessed 19-August-2021] `https://github.com/nitram2342/bruteforce-crc`

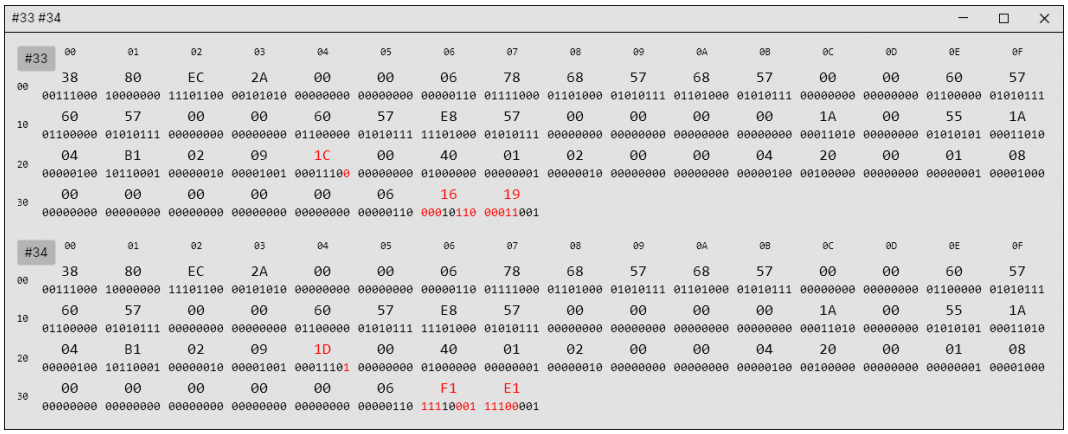


Figure 4.3: CRC-16 KERMIT payload error detection code

4.3.1.2 Bit-Precise Reasoning for packets of length `0x30`

Appropriate determination of the previously encountered unknown values might lead to additional conclusions; a necessity to understand the protocol to its full extend. Descendingly peeking at the communication's packet flow, starting at the first occurrence of `col(0x38)`, unfolds the value's origination: The connection handshake. The operator sends its first packet `col(0x30)`, comprising 48 bytes, to the drone, introducing various persistent and/or session pervading values (cf. Table 4.5):

`idx(0x02) - idx(0x03)`: An operator side, random generated session identifier (cf. Section 4.3.1.1), unexceptional encoded into every sent and received message at position `idx(0x02) - idx(0x03)`.

`idx(0x07)`: The previously reverse-engineered CRC-16 checksum raised our first suspicion of a common implementation base between the Ocusync and Wi-Fi protocol. Due to the presence of a CRC-16 checksum within the payload, regardless of the layer 3 checksum byte, we further argue the Ocusync protocol being of higher importance. In theory, a Ocusync based encoded message, wrapped inside a UDP packet, instantly enables the same feature-set without any additional implementation effort. Thus, the presence of the UDP header's checksum may be neglected with the application layer being responsible for error detection. The session identifier's importance, in combination with no error detection out of the box, requires an

additional safety handle, present at `idx(0x07)`: a CRC-8 or another Block Check Code (BCC) error detection byte. Similar to our previous determination attempt, we utilised the online CRC calculation tool [28] without success. Besides the brute-force option, we decided to construct an equation to solve the problem in a mathematical fashion: `val(0x02)` $\oplus$ `val(0x03)` $\oplus$ X = `val(0x07)`, whereas X represents the final XOR value and `val(0x07)` has been obtained from our session recording samples (cf. Figure 4.4 for idle, and Figure 4.5 for the gimbal session recording). Constant congruent results acknowledge our finding of the final XOR byte value `con(0xB0)`:

- `con(0x7C)` $\oplus$ `con(0x69)` $\oplus$ `con(0xB0)` = `con(0xA5)` — Idle session
- `con(0xD0)` $\oplus$ `con(0x50)` $\oplus$ `con(0xB0)` = `con(0x30)` — Gimbal session
- `con(0xEC)` $\oplus$ `con(0x2A)` $\oplus$ `con(0xB0)` = `con(0x64)` — Rotation session
- `con(0xBB)` $\oplus$ `con(0x67)` $\oplus$ `con(0xB0)` = `con(0x6C)` — Altitude session
- `con(0x67)` $\oplus$ `con(0x27)` $\oplus$ `con(0xB0)` = `con(0xF0)` — Velocity session

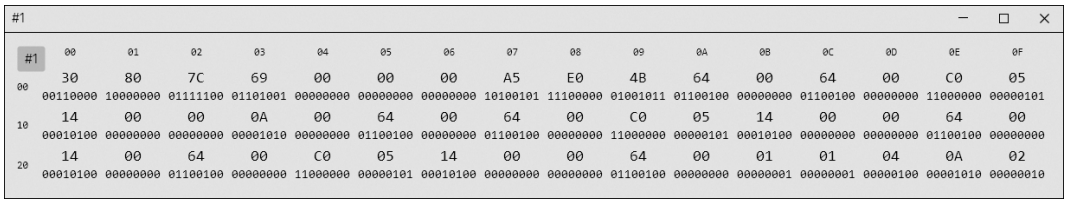


Figure 4.4: `col(0x30)` error detection byte reference — idle session recording

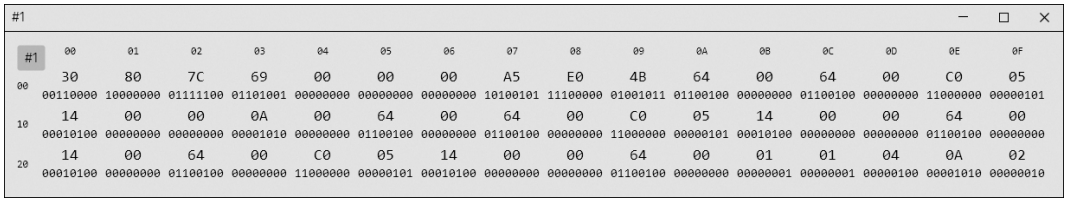


Figure 4.5: `col(0x30)` error detection byte reference — gimbal session recording

`idx(0x08) - idx(0x09)`: Represents the start value for various packet counters or other, yet unknown, fields. The bit-precise reasoning of `col(0x38)` and `col(0x21)` identified five direct dependencies:

- `col(0x38)[val(0x0A)]` = `val(0x08)`
- `col(0x38)[val(0x0B)]` = `val(0x09)`
- `col(0x21)[val(0x04)]` = `val(0x08)`
- `col(0x21)[val(0x05)]` = `val(0x09)`
- `col(0x21)[val(0x08)]` = `val(0x09)`

`idx(0x0A) - idx(0x2F)`: The mobile application offers a wide variety of variable settings for the thumb-stick assignment, image quality, acceleration modifier, and much more. Constant 304 bits indicate the transmission of essential takeoff parameters, as some mid-flight configuration adjustments (e.g. transmission bandwidth) require a drone reboot. Non-equal application settings were not part of our study, such that we can not deduce any further correlations.

Table 4.5: `col(0x30)` bit-precise reasoning

Offset [byte]	Size [bits]	Function	Description
`0x00 - 0x01`	16	Payload Length	`val(0x01) & 15 << 8 + val(0x00)`
`0x02 - 0x03`	16	Session Identifier	
`0x04 - 0x06`	24	Padding	Zero bits
`0x07`	8	BCC	`val(0x02)` $\oplus$ `val(0x03)` $\oplus$ `con(0xB0)`
`0x08 - 0x09`	16	Unknown	
`0x0A - 0x2F`	304	Unknown	Constants

4.3.1.3 Bit-Precise Reasoning for packets of length 0x21

Several col(0x21) initialization packets follow after the connection handshake, classifiable by val(0x1D) - val(0x1E) and their alternation between <con(0x00), con(0x01)> and <con(0x00), con(0xFF)>. Most other bytes either mirror (idx(0x09) - idx(0x0B)) or feature constant values (idx(0x06), idx(0x11) - idx(0x18) and idx(0x1C)). Furthermore, col(0x21) facilitates three independent counters (cf. Table 4.6):

idx(0x04): The start value equals col(0x30)[val(0x08)] + con(0x08) and its incrementation behaviour alternates between con(0x02) and con(0x08). It represents a non-exclusive packet-counter.

idx(0x10): A non-exclusive packet-counter starting at con(0x01) and incrementing by con(0x01).

idx(0x1A) - idx(0x1B): Starts at either <con(0xA0), con(0x00)> or <con(0xB0), con(0x00)>. It increments by con(0x08) in little-endian Byte order and is exclusively used within col(0x21).

The col(0x38) CRC-16 Kermit checksum application does not apply for col(0x21). It either implies the implementation of a proprietary algorithm or a sole exception for col(0x21). Similar to our previous final XOR determination attempt, we could elicit a deterministic formula to calculate the checksum value: kermit(val(0x18) - val(0x1E)) $\oplus$ con(0x75) = val(0x1F) - val(0x20). Moreover, idx(0x07) is likely to embrace a CRC-8 checksum, as no composition candidate nor a mathematical solvable equation has been found. We suspect idx(0x04) to be part of the equation as idx(0x07) varies upon counter incrementation. Other than that, we were not capable of extracting more precise information, which we could not already conclude from previous reasonings. Thus, we further continue our deductive reasoning with drone-to-operator packets, although we doubt to obtain new findings in respect to numerous unexplained fields within col(0x38), col(0x30) and col(0x21).

Table 4.6: `col(0x21)` bit-precise reasoning

Offset [byte]	Size [bits]	Function	Description
`0x00 - 0x01`	16	Payload Length	`val(0x01) & 15 << 8 + val(0x00)`
`0x02 - 0x03`	16	Session Identifier	`col(0x30)[val(0x02) - val(0x03)]`
`0x04`	8	Packet counter	`col(0x30)[val(0x08)] + val(0x08)`
`0x05`	8	Unknown	`col(0x30)[val(0x09)]`
`0x06`	8	Unknown	Constant
`0x07`	8	CRC-8	`val(0x04)` $\oplus$ `con(?) = val(0x07)`
`0x08`	8	Unknown	`col(0x30)[val(0x08)]`
`0x09`	8	Mirror	`val(0x05)`
`0x0A`	8	Mirror	`val(0x04)`
`0x0B`	8	Mirror	`val(0x05)`
`0x0C - 0x0F`	32	Padding	Zero bits
`0x10`	8	Packet counter	
`0x11 - 0x18`	64	Unknown	Constants
`0x19`	8	Unknown	
`0x1A - 0x1B`	16	Packet counter	Little-Endian
`0x1C`	8	Unknown	Constant
`0x1D - 0x1E`	16	Unknown	Initialization Sequence
`0x1F - 0x20`	16	CRC-16 Kermit	final XOR value `con(0x75)`

4.3.2 Drone-to-Operator

Contrarily to the operator, the drone tends to send Ethernet frames up to 1514 bytes (MTU without frame check sequence) with an effective UDP payload size of 1472 bytes. One remarkable benefit with rather large Ethernet frames incorporates a greater efficiency in data transmission, since the payload carries more user data while the protocol overhead remains the same [1]. In combination with a live-camera stream, it comes to no surprise that we predominantly observe rather large frames. However, its efficiency impedes our bit-precise reasoning due to non-congruent payloads, unfeasible for operator instruction response monitoring and value change-tracking. Further, the focus on extensive frames sizes hardens the suspicion of instruction response aggregation, backed by the fact of only two Ethernet frames being smaller than 100 bytes, in contrast to the remaining 4986 frames of the *Altitude* (cf. Section 4.3.1) session recording. In other words, we cannot pursue the same reverse-engineering strategy. Instead, we have to count on external resources and tools in order to extract as much information as possible, thereby shifting our previous structural related focus to raw data extraction itself.

4.3.2.1 Bit-Precise Reasoning for packets of length `0x08`

Even though bit-precise reasoning is not applicable for drone-to-operator packets, one payload, with a size of 8 bytes, stands out in particular: The connection handshake response (cf. Table 4.7). Similar to `col(0x30)`, the BCC final XOR has been determined by solving the equation: `val(0x02)` $\oplus$ `val(0x03)` $\oplus$ X = `val(0x07)`. (cf. Figure 4.6 as an example for the idle session recording checksum calculation)

- `con(0x7C)` $\oplus$ `con(0x69)` $\oplus$ `con(0x88)` = `con(0x9D)` — Idle session
- `con(0xD0)` $\oplus$ `con(0x50)` $\oplus$ `con(0x88)` = `con(0x08)` — Gimbal session
- `con(0xEC)` $\oplus$ `con(0x2A)` $\oplus$ `con(0x88)` = `con(0x4E)` — Rotation session
- `con(0xBB)` $\oplus$ `con(0x67)` $\oplus$ `con(0x88)` = `con(0x54)` — Altitude session
- `con(0x67)` $\oplus$ `con(0x27)` $\oplus$ `con(0x88)` = `con(0xC8)` — Velocity session

Table 4.7: `col(0x08)` bit-precise reasoning

Offset [byte]	Size [bits]	Function	Description
`0x00 - 0x01`	16	Payload Length	`val(0x01) & 15 << 8 + val(0x00)`
`0x02 - 0x03`	16	Session Identifier	`col(0x30)[val(0x02) - val(0x03)]`
`0x04 - 0x06`	24	Padding	Zero bits
`0x07`	8	BCC	`val(0x02)` $\oplus$ `val(0x03)` $\oplus$ `con(0x88)`

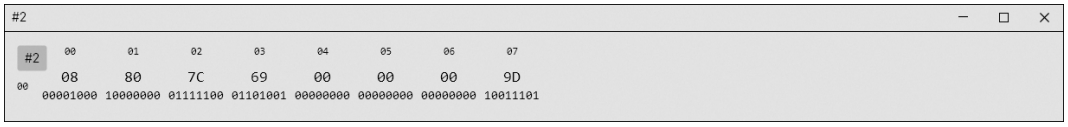


Figure 4.6: `col(0x08)` error detection byte reference for the idle session recording

4.3.2.2 Camera stream

Operator-to-drone Ethernet frames never draw near the MTU size limitation. Consequently, operator payloads do not require any kind of post-receival payload aggregation, in order to compose a valid message from the stream. On the other hand, drone-to-operator Ethernet frames predominantly utilise the payload to its full capacity, introducing the necessity of a message delimiter in regards to the previous deducted upper bound payload size limitation (cf. Table 4.4). A subset of our *Velocity* recording concludes and reinforces our reasoning (cf. Table 4.8) as frames smaller than the MTU represent EOM. As an example, the first five Ethernet frames (No. 2819 - 2823) assemble message #1, quantifiable by the fifth Ethernet frame size not being equal to 1514 bytes. Message #2 (No. 2826 - 2831) comprises six Ethernet frames with an aggregated payload size of 8249 bytes. Message #3 (No. 2834), with no MTU sized ancestor frame present, denotes a standalone message. According to our logic, the existence of a delimiter is per se optional, but obligatory within a Ethernet frame smaller than 1514 bytes. The presumption partially confirms with a delimiter series of <`val(0x00)`, `val(0x00)`, `val(0x00)`, `val(0x01)`, `val(0x09)`, `val(0x10)`> being mostly present at the end of such frames. Either the delimiter series incorporates only aggregation related frames, or it is part of the camera-feed image encoding [38].

With an endless stream of data in a mid-flight scenario, the message composition highly depends on the delimiter series. However, in our post-analysis, not only the Ethernet frame size is known, but also their corresponding message affiliation. Thus, we remove all delimiter series occurrences from our *Velocity* recording and export the drone's UDP data stream as a binary file, fully aware of intermixing camera-feed updates alongside drone control instruction responses. To obtain hints about the video encoding, we utilise FFprobe[3], an open-source application to gather meta-information from multimedia streams. It is part of the FFmpeg multimedia framework and supports not only cutting edge, but also most ancient and obscure formats [14]. Probing our binary file produced, next to missing frames and decoding errors, an extensive estimation:

- *Codec:* H.264
- *FPS:* 29.42
- *Resolution:* 1280x720
- *Color encoding:* yuv420p
- *Interlacing:* progressive

Next to probing, the multimedia framework additionally supports file format conversion. During its conversion process, FFmpeg is capable of repairing the video stream up to a certain degree, compensating, in our case, the mixture of video stream and instruction response data. Hence, we utilise FFmpeg to obtain a repaired, human-viewable video. Far from being perfect, the result contains enough information to get a clear picture of the surroundings, although the video accompanies stutter, frame skips and missing batch updates (cf. Figure 4.7). Accordingly, once we are capable of camera-stream and instruction response differentiation, FFmpeg would be competent enough to reconstruct an image-perfect video stream.

[3][Online; accessed 25-August-2021] https://ffmpeg.org/ffprobe.html

Table 4.8: Drone message compositions out of aggregated Ethernet frames

Message No.	Frame No.	Frame size [bytes]	Payload size [bytes]
1	2819	1514	1472
	2820	1514	1472
	2821	1514	1472
	2822	1514	1472
	2823	1000	958
2	2826	1514	1472
	2827	1514	1472
	2828	1514	1472
	2829	1514	1472
	2830	1514	1472
	2831	931	889
3	2834	470	428

Figure 4.7: FFmpeg UDP stream to .mp4 conversion

4.3.3 Concluding Findings

The bit-precise reasoning of various payloads turned out to be more difficult than expected. Even though we were capable to wrap our heads around several bytes and their purpose, most values still remain unknown. The constant stream of data additionally introduces complexity, as a direct correlation between a cluster and its drone control instruction can not be determined statistically. Moreover, due to the fact of shared values throughout various clusters, it's not safe to assume a direct correlation between the payload size and its purpose either. CRC checksum(s) cover a subset of each payload, an indicator for a multi-layered protocol structure, reinforced by the drone's instruction response aggregation which we can not refute on the operator side. The overall ambiguous protocol characteristics pose more questions as we are capable to answer. Nevertheless, our deductive reasoning results in a few findings:

- Neither full nor partial payload encryption has been implemented.
- Payloads start with their size encoded at `idx(0x00) - idx(0x01)`.
- Payloads carry a session identifier at `idx(0x02) - idx(0x03)`.
- CRC-16 checksums ensure the integrity of data-subsets.
- CRC-8 checksums complements uncovered CRC-16 data-subsets.
- Packet counters guarantee sequential command execution.
- The operator establishes a connection and dictates session parameters.
- The drone aggregates control instruction responses.
- The camera-feed is H.264 encoded.

In order to restrict potential protocol characteristics to a bare minimum, we do require to rely on supplementary external inputs. Besides the decomposition of the mobile application, we first focus on pre-existing online resources, favourable directly related to the MP1's Wi-Fi communication protocol, or, leastwise, another drone from the DJI Mavic series product lineup.

4.4 DJI Universal Markup Language

The extensive online research leaves a lot to be desired; besides some vague, surface leveled online articles [5, 12, 16, 47] discussing vendor-published facts and release notes, just a few forum posts [22, 40, 41, 43] explicitly engage the Wi-Fi or SDR protocol in a technical manner, more or less without any usable outcome or direct relation to a DJI product. However, most discussion threads share one common similarity: a reference to a public GitHub repository[4] "Original Gangsters". A community of drone enthusiasts and IT experts, sharing their knowledge about various DJI technologies, obtained via reverse-engineering, and their implementation details. Their findings and tools, although work in progress, cover [35]: firmware extraction and decryption, module partition patches, diverse Executable Linkable Format (ELF) converters, hard-coded value editors, flight-log parser, serial bus sniffer, container stream parser, etc. As the repository lacks usage instructions and in-depth documentation, our description and conclusions, throughout the entire chapter, purely build upon our personal gained experience while contributing (cf. Section 5.3) to the project. However, the serial bus sniffer and the container stream parser grabbed our attention in particular. The serial bus sniffer is capable of passively eavesdropping messages between the operator and the drone and outputs the received binary stream in DJI Universal Markup Language (DUML) format, a community derived acronym invention for the close-source DJI protocol implementation. On the other hand, the container stream parser accepts a binary file as an input parameter and tries to extract a valid DUML formatted packet. DUML is the core communication format used throughout all DJI products, including their latest releases; henceforth referred to as DUML protocol, as the format comprehends rules, syntax and semantics. In other words, regardless of the drone's mode of operation and its communication channel, the corresponding protocol contains DUML conformant data structures within its payload. As an example, a direct physical serial connection to the drone generates no overhead and serves raw DUML packets only, while the Wi-Fi and SDR mode of operation introduce some additional, non-identical headers. Therefore, we use OGs' preparatory work to familiarize ourselves with their findings and eventually compile the Wi-Fi's protocol architecture along the way.

[4] [Online; accessed 27-August-2021] https://github.com/o-gs/dji-firmware-tools

4.4.1 Wireshark Dissector

Wireshark supports dissectors via a self-registering plugin system. Custom dissectors parse a pre-defined portion of a frame and pass their remaining payload onto the next lowest-level data dissector [56]. Each state decodes and displays its corresponding values, enabling a convenient visual representation of the data. Fortunately, OG offer various DUML dissectors for an easy in-depth packet inspection [30] (cf. Figure 4.8). Using those dissectors on our recordings will not be of any help, as our `.pcap` files contain, next to the DUML payload, the entire protocol stack, comprising the data link, network and transport layer. Thus, for appropriate dissector application, we require to eliminate everything except for the DUML payload, presuming knowledge about its boundaries. By removing Ethernet frame bytes from `idx(0x00)` to `idx(0x29)` (cf. Table 4.1) we get rid of the data link, network and transport layer, but the Wi-Fi-Header within the payload still remains. Hence, we are obliged to find its length in the first place.

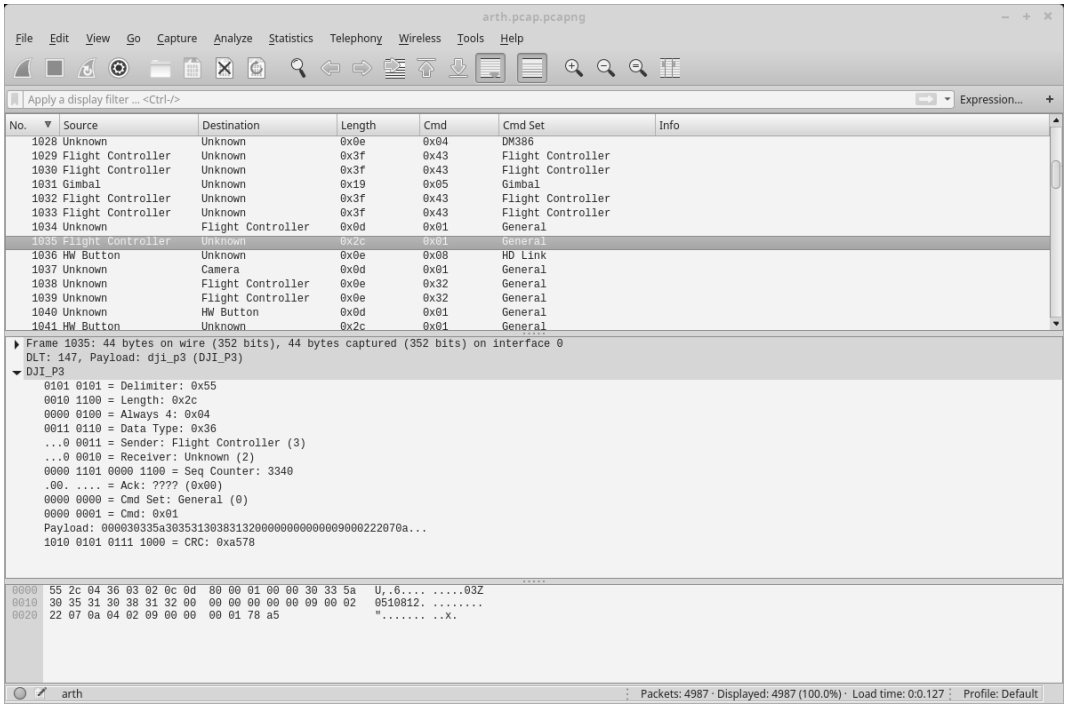


Figure 4.8: Wireshark DUML packet dissector utilisation example [29]

4.4.2 Payload Delimiter

The OG container stream parser python script `comm_dat2pcap.py` [31] accepts a binary file as an input parameter and attempts to identify valid DUML packets therein. Upon successful probing, the script exports the discovered packet into a dissector compatible `.pcap` file. If the parser is able to extract a DUML from an arbitrary payload within our *Altitude* recording, we can then investigate the functionality of the python script. Consequentially, we store the payload into a binary file and start the conversion process (cf. Listing 4.2), with `-n` to avoid trailing whitespaces and `-e` to enable the interpretation of backslash escapes. Next to a found DUML packet, the script yields a damaged section, no surprise in regards to the provided Wi-Fi-Header.

Listing 4.2: Probing for a valid DUML packet

```
1 echo -n -e '\x21\x80\xbb...\x01\xe8\x9a' > 3.bin
2 ./comm_dat2pcap.py -d 3.bin -p 3.pcap -vvv
3 3.bin: Packets encountered: 1 valid, 1 damaged
```

The script's source [31] revealed a drone model dependent variable payload delimiter. In case of our target specification: `con(0x55)`. However, several discrepancies emerge when applying the same delimiter logic onto all payloads within the recording. The delimiter is non-exclusive, as not only the header, but also the DUML may contain `con(0x55)` values. Especially drone-to-operator camera-feed updates are overpopulated by `con(0x55)` occurrences. Moreover, the header is of variable size, leading to inapplicable static index separation. The script tries to circumvent that issue by validating a CRC-16 checksum (cf. Table 4.3 and 4.6), which is part of each DUML packet (cf. Section 4.4.3) but accidentally identifies false positives within drone-to-operator camera-feed updates. The variable Wi-Fi-Header size leads us to believe in yet another header encoded indicator.

The preparatory execution of the delimiter logic on all packets, accompanied by manual interventions on false positives, provides differently sized headers and their payload. Classified and descendingly sorted by their size, only a small data subset requires an in-depth investigation in order to find the significant Wi-Fi-Header size

Table 4.9: Wi-Fi-Header encoded payload offset and content-type at `idx(0x06)`

Indicator [byte]	Sender	Purpose	Payload Offset [byte]
`0x00`[1]	Drone	Unknown	No DUML
`0x01`[3]	Drone	Command	`(val(0x1C) << 1) + con(0x20)`
`0x02`[1]	Drone	Camera-stream	`con(0x14)`
`0x03`[3]	Drone	Command	`con(0x14)`
`0x04`[4]	Operator	Command	`(val(0x0C) << 1) + con(0x1E)`
`0x05`[2]	Operator	Command	`con(0x14)`
`0x06`[4]	Operator	Command	`(val(0x0C) << 1) + con(0x1E)`

The payload contains: zero DUML entries[1], one DUML entry[2], *N* DUML entries[3], zero or one DUML entries[4]

indicator. Unfortunately, several bytes qualify as a candidate with a little help of bit shifting and/or constant value addition or subtraction. However, one candidate stands out in particular. The byte at `idx(0x06)` provides an indirect calculable Wi-Fi-Header size. Moreover, it provides further information about the payload's origin and content-type (cf. Table 4.9). Byte values between `con(0x00)` and `con(0x03)` relate to drone packets and values between `con(0x04)` and `con(0x06)` to operator ones. The deterministic content determinable protocol characteristic (e.g. `con(0x02)` accommodates only camera-stream updates) enables precise camera-feed update extraction and therefore, an image-perfect video encoding (cf. Figure 4.9).

4.4.3 Protocol Architecture

The protocol architecture explicitly excludes the OSI model layers (cf. Table 4.1), due to their non-relevant property reflections. Furthermore, we subdivided the self-contained DUML format into 3 logical layers, improving the comprehensibility in our architectural interpretation; a composition of our deductive reasoning results and the OG Wireshark dissector implementation. It consists of 4 layers (cf. Figure 4.10), whereas solely the Wi-Fi-Header remains mostly unknown, being the last obstacle for a complete self-contained communication counterpart replacement.

Figure 4.9: Image-perfect FFmpeg UDP stream to .mp4 conversion

4.4.3.1 Wi-Fi-Header

We suspect yet another protocol behind the Wi-Fi-Header, as its content has no direct influence on sent and received DUML packets. While some meta-information is essential for the data transmission and interpretation (e.g. payload length and content type), other bytes serve no obvious purpose. Other DJI products or 3rd party software solutions may take advantage of other values (e.g. session identifier), although we did not find any related API documentation entry [7, 8, 9, 10]. The Wi-Fi-Head is of variable size and carries mostly zero bits and value duplication besides its partially reverse-engineered structure (cf. Table 4.10).

Table 4.10: Protocol Architecture: Wi-Fi-Header

Offset [byte]	Size [bits]	Function	Description
`0x00 - 0x01`	16	Payload Length	`val(0x01) & 15 << 8 + val(0x00)`
`0x02 - 0x03`	16	Session Identifier	Arbitrary but fixed
`0x06`	8	Content type	cf. Table 4.9

4.4.3.2 DUML-Header

Based on the received Wi-Fi-Header's content-type property, the DUML-Header and its payload may be nonexistent, reinforcing our suspicion of a self-contained protocol. In particular, the content-type `con(0x04)` and `con(0x06)` indicates such a behaviour. Thus, if the layer 4 payload size equals the DUML-Header offset calculation, no additional payload has been provided. Moreover, in contrast to the drone, the operator never aggregates DUML packets; instead, sends each individual DUML packet as a standalone datagram. Nevertheless, in regards to message compilation, we still require to consider concatenated DUML packets (payloads of content-type `con(0x01)` or `con(0x03)`), whereas the DUML-Trailer strings together with the adjacent DUML-Header. This logical layer embraces relevant meta-information for further payload delegation and processing, by the coverage of the following fields (cf. Table 4.11):

`idx(0x00)`: The DUML-Header starts with a drone model dependent, variable delimiter. The MP1, Phantom 3, Phantom 4, etc. encode `con(0x55)`, while Phantom 1, Phantom 2, Naza M, etc. encode `con(0xAB)`. Although the delimiter is a necessity in the serial mode of operation, it is a futile piece of information in Wi-Fi mode. We justify its existence with the DUML protocol's diverse field of application.

`idx(0x01) - idx(0x02)`: The DUML protocol's optimized design leaves no bit unassigned. Therefore, not only the protocol version, but also various other fields compose their values via bit shifting. In other words, the first 6 bits of `idx(0x02)` represents the protocol version (`val(0x02) & con(0xFC) >> 2`) which equals, in our case, `con(0x01)`. The remaining 2 bits, plus the entire byte at `idx(0x01)`, compose the payload length in little-endian byte order (`val(0x01) - val(0x02) & con(0x3FF)`). The payload length includes the Header, Body and Trailer of the DUML packet, described by the formula: `con(0x0B) +` Y `+ con(0x02)`, whilst Y represents the arbitrary but fixed DUML-Body length.

`idx(0x03)`: A standard CRC-8 checksum algorithm [31] covers the delimiter, protocol version and the payload length. The algorithm uses a custom initial value `val(0x77)` and a non-standard hexadecimal lookup table [33].

`idx(0x04) - idx(0x05)`: The drone and operator inheres several logical software and hardware component abstractions, each individually addressable. The modular design allows not only flexible hardware extensions and independent software updates, but also direct DUML-Body payload delegation to its responsible processing module. For appropriate delegation, the sender/receiver index/type reflects its origin and destination. The concrete values are visible within the Dissectors implementation [34]. Even though the Wi-Fi mode of operation exposes its Ethernet frame origin at layer 2 and 3, the serial mode of operation highly depends on those values for source and destination differentiation.

`idx(0x06) - idx(0x07)`: On the operator side, the little-endian packet counter starts at `con(0x00)` and increments by `con(0x01)` per DUML packet. On the other hand, drone-to-operator packets facilitate a random packet counter value (for each DUML message), avoiding interference or duplicates.

`idx(0x08)`: Operator-to-drone packets predominantly require no command execution acknowledgement, represented by `con(0x00)`. Alternatively, modules may confirm the command before `con(0x02)` or after `con(0x03)` its execution. The last 4 bits of `idx(0x08)` indicate the DUML-Body's encryption mode [34]: None, AES 128, Self Def, XOR, DES 56, DES 112, AES 192 or AES256. Due to the Wi-Fi AES CCMP encryption, no other value than `con(0x00)` (none) occurs within our recordings. However, the presence of an encryption flag raises the suspicion of an application-level DUML-Body encryption, potentially applied upon unencrypted communication channel usage or with other modes of operation.

`idx(0x09) - idx(0x0A)`: A command [34] `idx(0x0A)` is an action to be performed (e.g. get parameter), whilst the command-set [34] `idx(0x09)` specifies the target component (e.g. camera). Hence, a command may be valid and executable in various components, such that a second constraint is needed; an indicator for a shared code-base across several self-contained component implementations.

4.4.3.3 DUML-Body

The body is of variable length and its content embraces the command's parameters or response data. Even though the structural information of the wast majority

Layer 2 - 4	Payload			
`len(0x2A)`	Wi-Fi-Header	Payload		
	`len(0x14)` + *N**	DUML-Header	DUML-Body	DUML-Trailer
		`len(0x0B)`	*N***	`len(0x02)`

* Length depends on the content-type
** Length depends on the command-set and command

Figure 4.10: Protocol Architecture

Table 4.11: Protocol Architecture: DUML-Header

Offset [byte]	Size [bits]	Function	Description
`0x00`	8	Delimiter	`con(0x55)`
`0x01 - 0x02`	6	Protocol Version	`val(0x02) & con(0xFC) >> 2`
`0x01 - 0x02`	10	Payload length	`val(0x01) - val(0x02) & con(0x3FF)`
`0x03`	8	CRC-8	`val(0x00) - val(0x02)`
`0x04`	3	Sender Index	
`0x04`	5	Sender Type	
`0x05`	3	Receiver Index	
`0x05`	5	Receiver Type	
`0x06 - 0x07`	16	Packet counter	Little-Endian
`0x08`	1	Request type	Req. `con(0x00)` Res. `con(0x01)`
`0x08`	3	Acknowledgement	`con(0x00)`, `con(0x02)`, `con(0x03)`
`0x08`	4	Encryption	`con(0x00)`
`0x09`	8	Command-set	
`0x0A`	8	Command	

has been covered by the dissectors [30], some particular commands still require further investigation. With uncountable available commands, we focus on thesis relevant objectives, such as: Mid-air status indication, UAV altitude and velocity extraction (cf. Table 4.12 and Figure 4.11), Power-level and remaining flight time expectation (cf. Table 4.13). The appropriate payload structures and their values

are inspectable with the OGs' Wireshark dissectors and their corresponding filter predicates (`dji_dumlv1.cmd` and `dji_dumlv1.cmdset`) applied (cf. Figure 4.11).

The remaining flight time expectation is not an absolute retrievable number. Instead, the command `con(0x03)` within the command-set `con(0x0D)` retrieves the batteries power-level. According to the MP1 specification, a single battery contains a power-level of 3830 mAh [6]. Thus, the remaining flight-time is computable.

Table 4.12: DUML-Body: Cmd `con(0x43)` Cmd-Set `con(0x03)`

Offset [byte]	Size [bits]	Function	Description
`0x00 - 0x07`	64	Longitude	
`0x08 - 0x0F`	64	Latitude	
`0x10 - 0x11`	16	Relative Height	
`0x12 - 0x13`	16	Velocity X	
`0x14 - 0x15`	16	Velocity Y	
`0x16 - 0x17`	16	Velocity Z	
`0x18 - 0x1A`	16	Pitch	
`0x1B - 0x1C`	16	Roll	
`0x1D - 0x1E`	16	Yaw	
`0x24`	4	On Ground	`val(0x24) & con(0x02)`
`0x24`	4	In Air	`val(0x24) & con(0x04)`

Table 4.13: DUML-Body: Cmd `con(0x03)` Cmd-Set `con(0x0D)`

Offset [byte]	Size [bits]	Function	Description
`0x01`	8	Cell count	`con(0x03)`
`0x02 - 0x03`	16	Voltage Cell 1	
`0x04 - 0x05`	16	Voltage Cell 2	
`0x06 - 0x07`	16	Voltage Cell 3	

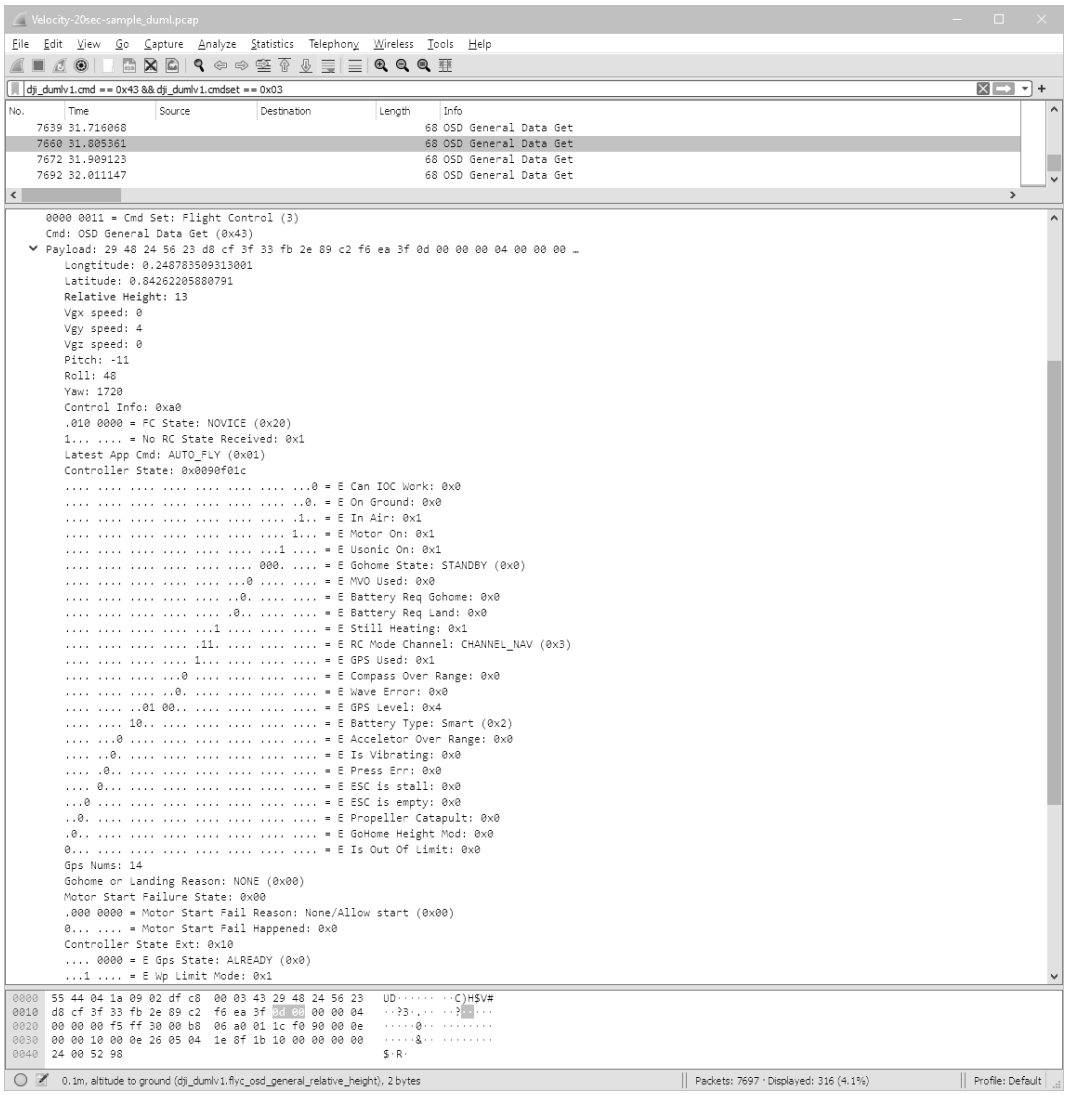


Figure 4.11: UAV altitude and velocity inspection with Wireshark and OG dissectors

4.4.3.4 DUML-Trailer

The trailer comprehends a CRC-16 checksum, calculated with a standard algorithm [31] and populated with a custom initial value `con(0x3692)` and a non-standard hexadecimal lookup table [32]. It covers a portion of the DUML-Header (`idx(0x04)` - `idx(0x0A)`) and the entire DUML-Body. The trailer poses the last layer within our architectural illustration (cf. Figure 4.10) and concludes our investigative research.

5 DJI Wi-Fi Tools

Wireshark lacks various implementation details for an unsophisticated reverse-engineering process, demanding additional tool-sets to overcome its limitations. As an example, payload truncation, a requirement for the DUML dissector compatibility, requires the general-purpose utility editcap[1] just for the simple task of data-link to transport layer data removal. With the variable-sized Wi-Fi-Header still being part of the payload, editcap would require a dynamic rule-set to pursue data truncation, yet another uncovered functionality. A dissector extension might be a potential alternative, but even optimistic time estimates (syntax familiarisation, read up OG's code and the actual implementation) clearly surpass its value in exchange. Truncation aside, the hexadecimal payload representation is non-ideal, in terms of bit-precise reasoning, as conspicuous bit-differences are not visible at first glance. Moreover, the importance of a side-by-side payload comparison, within or throughout various `.pcap` files, is a necessity, but non-existent. Multiple concurrent filter predicates, a necessity to deal with numerous data sets, are too tortuous with their concatenation behaviour. Chains of clauses cause confusion and immense adaption effort upon investigation target realignment. Altogether, Wireshark is too cumbersome and inappropriate for our use-case scenario. After utilising Wireshark for a while, we decided to develop a proprietary software solution to assist our investigation process and to enable task automatisation up to a certain degree for tedious intermediate workflow steps. The application should not pose a direct alternative to Wireshark; instead, it incorporates a collection of highly specialised feature sets, only applicable for the DJI low-level Wi-Fi protocol reverse-engineering process. We define its requirements as follows:

[1] [Online; accessed 27-September-2021] `https://www.wireshark.org/docs/wsug_html_chunked/`

- *Automatic drone and operator detection:* The application detects the drone and operator automatically within the same network and starts to listen to their network traffic only. Non-relevant traffic, such as ICMP, DNS, and MDNS packets should be ignored.
- *Session recording:* For post-flight analysis in Wireshark the application exports a session into two different `.pcap` files: The raw Ethernet frame captures (equivalent to Wireshark's passive eavesdrop capability) and prefabricated DUML dissector compatible packets.
- *Flight simulation:* Time discrepancies between the actual mid-flight session recording and the post-analysis process devastate any drone control instruction and its sent network packet coherence. In other words, one might not be able to interlink Ethernet frames and their corresponding operator inputs. Therefore, the application can load previous *session recordings* and simulate the flight mission in soft real-time. Furthermore, it supports step-by-step or bulk import operations.
- *Inspection and comparison:* Within the application a data-grid contains all recorded Ethernet frames and offers in-depth binary payload inspection. Arbitrary frames are selectable and comparable whereas each bit difference is visible at first glance with appropriate colour highlighting.
- *Filter predicates:* The data-grid offers various pre-defined or custom filter predicates, including custom binary sequences, DUML fields (e.g. command, command-set, sender, receiver, etc.), payload length and time-based constraints.
- *Video-stream rendering:* The application directly embeds FFmpeg to export the current *session recording* into a human-viewable file format.

5.1 Framework and Libraries

To support the vast majority of operating systems and due to the fact of using Windows and Linux concurrently, we focus on platform-independent technologies

only. Furthermore, based on personal preferences, we select the Windows Presentation Foundation (WPF) derivative Avalonia UI[2], as all essential UI controls are built-in and the widely-used Extensible Application Markup Language (XAML) is supported to its full extend [18]. Moreover, the Model–View–ViewModel (MVVM) pattern allows a clear separation of UI code and business logic, facilitating reusability, modularity, and maintainability [45]. With .NET 5 we may compile the code base for x86 and ARM processors, enabling support for more exotic system configurations. To avoid reinvention and unnecessary reimplementation of certain functionalities, we further include the following open-source project references:

- *FFMpegCore:* A .NET Standard wrapper for the non-managed FFmpeg/FFprobe open-source library, supporting synchronous and asynchronous media analysis and video file format conversion [39].
- *LibVLCSharp:* An audio playback and video rendering library for .NET applications. Based on the VideoLAN's LibVLC library, *LibVLCSharp* offers a comprehensive API for various tasks, such as stream-based video encoding and rendering. [51].
- *PcapNet:* A .NET Standard wrapper for the non-managed WinPcap library [37], utilised to import `.pcap` files for further processing.
- *SharpPcap:* A fully managed, cross platform .NET library for network traffic inspection and intersection [25].

5.2 Application Architecture

We split our application into four class libraries (cf. Figure 5.1) to assign and restrict their functional scope, allowing library independent development progress and simplified mockup provisioning during the early development phase. We define their field of responsibility as follows (cf. Figure 5.1): (i) *Dji.Network.Packets* reflects the DUML protocol structure in separated classes, (ii) *Dji.Network* embeds the passive eavesdropping implementation alongside a network traffic simulation derivative,

[2][Online; accessed 27-September-2021] `https://avaloniaui.net/`

(iii) *Dji.Camera* provides a custom video-player control with the ability to playback arbitrary byte sequences, (iv) *Dji.UI* defines UI windows and components for the entire application.

The following implementation details portray a small portion of the overall software architecture. For presentability and understandability reasons the UML diagrams and code illustrations have been adapted up to a certain degree and deviate from the actual implementation. The software solution is open-source and available on GitHub[3] for an accurate in-depth investigation.

5.2.1 DJI Network Packets

Due to the Wi-Fi protocol architecture (cf. Section 4.4.3.1) one could argue a single class implementation for all sent and received payloads. However, our generalised approach (cf. Figure A.2) allows the instantiation of particular content-type (cf. Table 4.9) implementations and forms the foundation of a type-based event-driven message delegation within the application. To be more precise, we choose the concrete object type based on the Wi-Fi-Header's content-type property (cf. Table 4.9):

- `con(0x00)`: `DjiEmptyPacket`
- `con(0x02)`: `DjiFramePacket`
- `<con(0x01),con(0x03),con(0x05)>`: `DjiCmdPacket`
- `<con(0x04), con(0x06)>`: `DjiEmptyPacket` or `DjiCmdPacket`

Although we did not abstract any further, one could extend the library with more concrete command and command-set related object types. For instance, `DjiBatteryPowerLevel` (cf. Table 4.13) would inherit `DjiCmdPacket` and overwrite its base class building methods (cf. Listing B.1). Additionally, the library features CRC lookup tables and their algorithm implementations [32, 33] alongside various enumeration constants [34] (cf. Figure A.2).

[3] `https://github.com/Toemsel/dji-wifi-tools/`

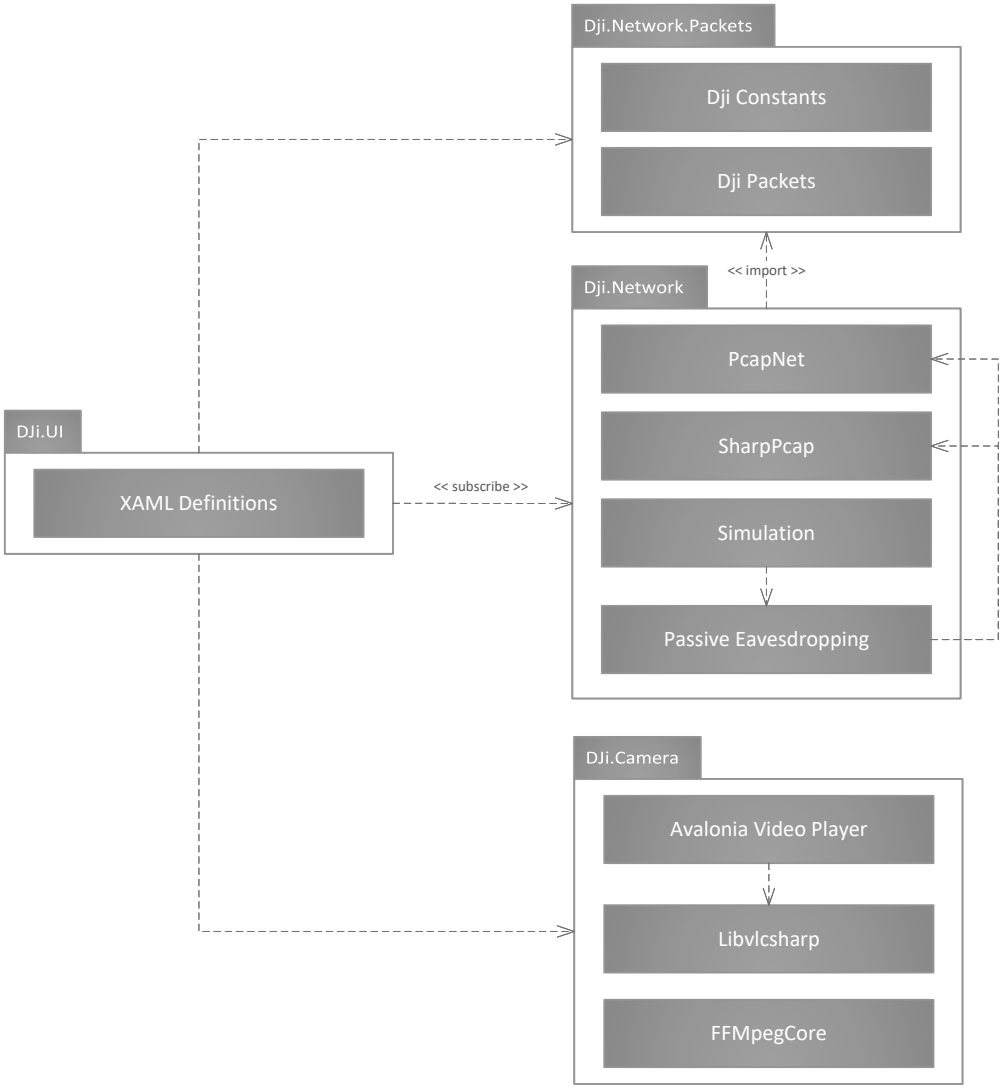


Figure 5.1: DJI Wi-Fi Tools Package Diagram

5.2.2 DJI Network

The `DjiPacketSniffer` (cf. Figure A.3) utilises the open-source *SharpPcap* library in order to passively eavesdrop Ethernet frames. Within our artificial MITM network scenario (cf. Chapter 3) the relevant network interface to listen on is pre-defined, but remains unknown on foreign systems. Thus, in order to guarantee the same functional behaviour on diverging network configurations, we require to configure *SharpPcap* to listen on all available interfaces till the very first drone-to-operator or operator-to-drone message has been encountered, identifiable by the following predicate compliance:

- The Ethernet frame has a minimum length of 43 bytes.
- The packet's protocol-type at `idx(0x17)` equals `con(0x11)` (cf. Table 4.1).
- The source or destination IP address equals `192.168.2.1` (cf. Section 5.3).
- Neither the source nor the destination IP address equals a local loopback or broadcast address.

With all conditions met, the `DjiPacketSniffer` perceives the IP addresses and henceforth only delivers appropriate Ethernet frames to its `NetworkPacketReceived` subscribers, as further processing is beyond the scope of its responsibility. Two subscribers, `DjiOperatorResolver` and `DjiDroneResolver` (both implementations of `DjiPacketResolver`), continue to instantiate the payload's representative `DjiPacket` (cf. Section 5.2.1) and forward the concrete instance to their, via `AddDjiPacketListener` subscribed, consumers. Hence, a consumer may subscribe to a particular `DjiPacket` implementation (cf. Listing 5.1).

Listing 5.1: Type based DjiPacket subscription

```
1 var _sniffer = new DjiPacketSniffer();
2 var _droneResolver = new DjiDronePacketResolver();
3
4 _sniffer.NetworkPacketReceived += (data) => _droneResolver.Feed(data);
5 _droneResolver.AddDjiPacketListener<DjiFramePacket>(FrameUpdate);
6
7 void FrameUpdate(DjiFramePacket framePacket) { } // process frame update
```

The *Simulation* of previous mid-flight scenarios requires some sort of persistent flight-log documentation. The most straightforward and relatively easy to implement solution comprises a direct Ethernet frame dump, such that each sent and received frame will end up on a persistent storage in form of a `.pcap` file. The only necessity for a simulation replay is the playback of the `.pcap` file; or alternatively, feeding the `DjiPacketSniffer` directly by its public packet retrieval processing method `OnPacketArrival`. In other words, `OnPacketArrival` either receives an Ethernet frame via eavesdropping or from a previous session recording supplied by the `DjiDroneSimulator`. The architectural advantage of our approach involves no additional implementation effort or existing code base adjustments within the `DjiPacketSniffer`, while the application's overall functional behaviour remains the same. Consequently, we implement the pcap writer `DjiPacketPcapWriter` and subscribe to `NetworkPacketReceived` (cf. Listing 5.2). Alongside raw Ethernet frames, the `DjiPacketPcapWriter` accepts concrete `DjiPacket` instances and stores the DUML-Header, DUML-Body, and DUML-Trailer into a second independent `.pcap` file, automatically enabling dissector compatibility by receiving all `DjiPackets` from both `DjiPacketResolvers` (cf. Listing 5.2).

Listing 5.2: DjiPacketPcapWriter Ethernet frame and DjiPacket subscription flow

```
1  var _sniffer = new DjiPacketSniffer();
2  var _pcapWriter = new DjiPacketPcapWriter();
3  var _droneResolver = new DjiDronePacketResolver();
4  var _operatorResolver = new DjiOperatorPacketResolver();
5
6  _sniffer.NetworkPacketReceived += (data) => _droneResolver.Feed(data);
7  _sniffer.NetworkPacketReceived += (data) => _operatorResolver.Feed(data);
8
9  // 1. subscribe to raw Ethernet frames
10 _sniffer.NetworkPacketReceived += ( data) => _pcapWriter.Write(data);
11 // 2. subscribe to concrete DjiPackets
12 _droneResolver.AddDjiPacketListener<DjiPacket>(_pcapWriter.Write);
13 _operatorResolver.AddDjiPacketListener<DjiPacket>(_pcapWriter.Write);
```

`DjiDroneSimulator`, responsible for the actual simulation, loads a previous `.pcap` session recording with *PcapNet*, and passes, based on the current mode of simulation (step-by-step, bulk, or soft realtime), the Ethernet frame(s) directly to the `DjiPacketSniffer`.

5.2.3 DJI Camera

To convert the `DjiFramePacket`, received via `DjiDronePacketResolver`, into a human viewable video format, the `DjiCamera` (cf. Figure A.4) passes their aggregated payloads to the referenced *FFMpegCore* library (cf. Listing 5.3), whereas the underlying FFmpeg binary starts the conversion process and stores the encoded video stream to the predefined target file location. For video playback within the application (cf. Figure 5.2), the `VideoPlayer` embeds an external `VideoView` control from *Libvlcsharp*; a straightforward solution for a file based video playback of the previously encoded video stream.

Listing 5.3: DjiCamera video export

```
1  public void FrameUpdate(DjiFramePacket framePacket) =>
2      File.AppendBytes("buffer", framePacket.FrameData);
3
4  private async Task<bool> ExportVideo(string targetFile) => await
5      await FFMpegCore.FFMpegArguments.FromFileInput("buffer")
6      .OutputToFile(targetFile).ProcessAsynchronously(false);
```

Figure 5.2: VideoPlayer control showcase

5.2.4 DJI UI

The last library within our architectural design (cf. Figure 5.1) strings the independent packages together (cf. Listing 5.1 and Listing 5.2). Moreover, it contains all XAML definitions for the entire application, comprising (cf. Figure 5.3):

(a) Menu-bar items to (i) load a previous session recording, (ii) start or stop a new session recording, (iii) save the current session as a video-stream

(b) Individual panels for `DjiCmdPacket`, `DjiEmptyPacket`, and payloads

(c) A data-grid per panel with the most essential information visible at first glance

(d) Filter predicates for the data-grid's content

(e) A Simulation window, supporting soft real-time, step-by-step or bulk import

#	src-module	dest-module	comm	cmd-set	cmd	ack	enc	ctr	f-size	p-siz
762	App	Gimbal	Request	General	Version Inquiry	AfterExe	None	63	75	33
755	App	DroneHDMCU	Request	Special	Old Special Remote Control	AfterExe	None	280	98	56
754	App	DroneHDMCU	Request	Special	Old Special Remote Control	AfterExe	None	279	98	56
752	Battery	App	Response	General	Activate Config	AfterExe	None	149	985	943
752	FC	App	Response	General	Version Inquiry	NoAck	None	62	985	943
752	FC	App	Request	FlightCont	ead Params By Hash	NoAck	None	26733	985	943
752	Battery	App	Response	General	Activate Config	AfterExe	None	47	985	943
752	FC	App	Response	FlightController	Ver Phone Set	NoAck	None	150	985	943
752	Camera	App	Response	General	Version Inquiry	AfterExe	None	61	985	943
752	DroneHDMCU	App	Request	MBINO	Eye Com Status Update	NoAck	None	21957	985	943
752	DroneDM36X	App	Response	General	Set Date/Time	AfterExe	None	151	985	943
752	DroneHDMCU	App	Request	FlightController	FlyC Vis Avoidance Param Get	NoAck	None	21956	985	943
752	DroneHDMCU	App	Request	FlightController	OSD General Data Get	NoAck	None	21955	985	943
752	FC	App	Request	FlightController	FlyC RTK Location Data Get	NoAck	None	26704	985	943

Figure 5.3: DJI.UI main window showcase

Each panel's viewmodel base-class `DjiNetworkPacketPool` (cf. Figure 5.3) holds a collection of `DjiPacket`, whereas new elements are addable via the internal `Store` method. Concrete implementations of the abstract base-class (e.g `DjiTrafficDock`, `DjiEmptyDock`, etc.) provide their elements by a static `DjiPacketResolver` type subscription (cf. Listing 5.4). The data-grid, a child control of the panel, obtains its content from the parent's viewmodel and defines its visual representation as referenced (cf. Listing 5.5). With the concrete `DjiPacket` type known within each grid, the object's properties (e.g. Sender, Receiver, Comms, etc.) are direct accessible and bind-able to the corresponding column definition.

Listing 5.4: DjiTrafficDockViewModel packet-type subscription

```
1 public class DjiTrafficDockViewModel : TrafficDockViewModel
2 {
3     public DjiTrafficDockViewModel()
4     {
5         DjiContentViewModel.Instance.OperatorPacketResolver.
              AddDjiPacketListener<DjiCmdPacket(Store);
6
7         DjiContentViewModel.Instance.DronePacketResolver.
              AddDjiPacketListener<DjiCmdPacket>(Store);
8     }
9 }
```

Listing 5.5: XAML definition clip of DjiTrafficDock

```
1  <WrapPanel Orientation="Horizontal">
2      ...
3      <filters:IpAddressFilter DjiNetworkPacketPool="{Binding $parent[
           UserControl].DataContext}" />
4  </WrapPanel>
5
6  ...
7
8  <DataGrid Source="{Binding DjiPackets}">
9      <DataGrid.Columns>
10         <DataGridTextColumn Header="src" Binding="{Binding Sender}" />
11         <DataGridTextColumn Header="dest" Binding="{Binding Receiver}" />
12         <DataGridTextColumn Header="comm" Binding="{Binding Comms}" />
13         ...
14     </DataGrid.Columns>
15 </DataGrid>
```

Additionally, the `DjiNetworkPacketPool` holds a filter expression property called `baseFilter`, applied to newly added elements or to the entire `DjiPackets` collection upon `EvaluateFilterOnPackets` method invocation or predicate redefinition. Similar to the data-grid having access to the parent viewmodel, each filter control within the panel may concatenate its own predicate definition to the `baseFilter`, received via relative binding (cf. Listing 5.5), enabling control reusability among different panels and a dynamic UI configuration, whereas filter controls may be added or removed on demand. As an example, the `IpAddressFilter` (cf. Listing 5.6) provides two toggle buttons bound to their corresponding `Drone` and `Operator` viewmodel property (cf. Listing B.2). Both properties feature a state-change listener causing the `DjiNetworkPacketPool` to re-evaluate the `baseFilter` expression upon `RaiseAndSetIfChanged`. In order to concatenate the `FilterExpression` to the `baseFilter`, its parent class implementation `FilterControlViewModel` (cf. Listing B.3) adds its own abstract property definition to the `DjiNetworkPacketPool`.

Listing 5.6: XAML definition clip of IpAddressFilter

```
1  <UserControl.DataContext>
2      <model:IpAddressFilterViewModel />
3  </UserControl.DataContext>
4
5  <StackPanel Orientation="Horizontal">
6
7      <ToggleButton IsChecked="{Binding Drone}" Image="Drone.png" />
8      <ToggleButton IsChecked="{Binding Operator}" Image="Operator.png" />
9
10 </StackPanel>
```

Each data-grid entry facilitates a context-menu, whereas its underlying datagram's payload is inspectable in a separate window in detail (cf. Figure 4.5, Figure 4.4, and Figure 4.6), or directly comparable to other datagram payloads in their window instance (cf. Figure 4.2 and Figure 4.3). Tooltips for all hexadecimal numbers offer a convenient denary representation.

5.3 Information Gathering

We manually transcompiled the dissector's available commands and command-sets into a C# representative attribute annotation (cf. Listing 5.7 and Listing B.4 for the attribute implementation). The application pre-loads all hard-coded attributes to create a dictionary mapping between the hexadecimal command representation and the attribute's object reference, enabling a fast lookup during the `DjiPacket` instantiation. Each attribute holds a description which will be displayed within the data-grid – to facilitate human readability and understandability – and may serve as an optional filter predicate condition. The first live monitoring observation with our software tool received various unknown commands and command-sets, proving the dissectors incomplete. With no further information available, we have no choice but to decompile the mobile Android application (cf. Chapter 2) in order to identify the unknown commands and command-sets corresponding descriptional denotation.

Dex to Java Decompiler (JADX), an open-source Android Application Package (APK) decompiler available on GitHub[4], offers a convenient GUI for automated bytecode decompiliation, deobfuscation, and Java file exportation. Instead of manual code investigation throughout 12.747 decompiled Java files, we further utilise Agent Ransack[5], a free file content search utility for Windows, to obtain, via regular expression, only files of interest. Many hexadecimal and denary expression building attempts did not deliver any feasible search result, thereby leading us to rephrase the expression to a more generalised and file name oriented predicate. A command related file name notation affirms our assumption with the regular expression `.*Cmd.*` yielding various command and command-set associated Java files, including: CmdSet, CmdIdCamera, CmdIdGimbal, CmdIdWifi, etc, similar to the dissector's command-set naming convention, and therefore, similar to our attribute transcompilation (cf. Listing 5.7). Thus, in order to obtain the missing attribute definitions, we inspect `CmdSet.java` which describes the available command-set enum values (cf. Listing B.5). As an example, the missing command-set `con(0x11)` represents the enum `ADS_B` and further dictates its specific command values within `CmdIdADS_B.java` (cf. Listing B.6). It might relate to Automatic Dependent Surveil-

[4][Online; accessed 02-October-2021] `https://github.com/skylot/jadx`
[5][Online; accessed 02-October-2021] `https://www.mythicsoft.com/agentransack/`

lance–Broadcast (ADSB), a technology to determine the UAV's position via satellite navigation [2]. Besides the amendment of missing attribute definitions within our application, we correspondingly contributed[6] to the OGs' dissectors to add support for uncovered command and command-set derivatives.

Listing 5.7: Command and command-set attribute notation

```
1 [Cmd(513, 0x02, "Camera", 0x01, "Do Capture Photo")]
2 Camera_DoCapturePhoto = 0x0201,
3 [Cmd(1025, 0x04, "Gimbal", 0x01, "Gimbal Control")]
4 Gimbal_GimbalControl = 0x0401,
5 [Cmd(1799, 0x07, "Wifi", 0x07, "WiFi Ap SSID Get")]
6 Wifi_WiFiApSSIDGet = 0x0707
```

Crucial for *automatic drone and operator detection* is the identification of a potential hard-coded and pre-defined drone IP address. Hence, we construct another regular expression (`"([0-9]{1,3}.){3}[0-9]{1,3}"`) to search for any IP address occurrence within the source-code. Among other services with a static IP assignment, the `SwUdpService.java` confirms, with a static IP declaration of `192.168.2.1`, our prior MITM observations.

[6] `https://github.com/o-gs/dji-firmware-tools/pull/206`

6 Conclusion

With deductive and bit-precise reasoning we were capable to determine various fields and their values alongside a rough picture of the protocol's functional behaviour. The OGs' DUML dissectors allowed a precise protocol structure characterisation and led us to the final Wi-Fi protocol structure. Dissector unknown commands have been obtained by decompilation and investigation of the mobile Android application. We developed a proprietary open-source software solution to passively eavesdrop the intercommunication between a drone and its operator, facilitating live-monitoring, post-flight simulation, in-depth analysis, DUML compatible Wireshark dissector export, and camera-feed image extraction.

Although we did not investigate any other mode of operation, it is safe to assume an application-level payload encryption in SDR mode, rendering – in terms of information extraction or manipulation – any active or passive eavesdrop operation infeasible.

The development of a ground station communication counterpart is within the realms of possibility and would enable a DJI software product independent remote control. Furthermore, drones operating in Wi-Fi mode, in combination with Wi-Fi deauthentication and a passphrase brute-force or dictionary attack, are vulnerable to unauthorised third-party hijacking. Thus, we recommend to avoid the Wi-Fi mode of operation and suggest the usage of its superior SDR mode.

Bibliography

[1] Ethernet Alliance. *Ethernet Jumbo Frames*. Nov. 2009. URL: http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf.

[2] Budroweit J., Jaksch M. P., Delovski T. "Design of a multi-channel ADS-B receiver for small satellite-based aircraft surveillance". In: *2019 IEEE Radio and Wireless Symposium (RWS)*. ISSN: 2164-2974. IEEE, Jan. 2019, pp. 1–4. DOI: 10.1109/RWS.2019.8714514.

[3] Claypool M., Tanner J. "The effects of jitter on the peceptual quality of video". In: *MULTIMEDIA '99: Proceedings of the seventh ACM International Conference on Multimedia (Part 2)*. New York, NY, USA: Association for Computing Machinery, Oct. 1999, pp. 115–118. DOI: 10.1145/319878.319909.

[4] Dalamagkidis K. "Definitions and Terminology". In: *Handbook of Unmanned Aerial Vehicles*. Ed. by Kimon P. Valavanis and George J. Vachtsevanos. Dordrecht: Springer Netherlands, 2015, pp. 43–55. DOI: 10.1007/978-90-481-9707-1_92.

[5] DeepSig. *Introduction to Commercial Drone Signals*. [Online; accessed 23-September-2021]. URL: https://www.deepsig.ai/news/introduction-to-commercial-drone-signals.

[6] DJI. *DJI Mavic Pro 1 hardware specifications*. [Online; accessed 09-June-2021]. URL: https://www.dji.com/at/mavic.

[7] DJI. *DJI Mobile SDK*. [Online; accessed 14-June-2021]. URL: https://developer.dji.com/mobile-sdk/documentation/introduction/mobile_sdk_introduction.html.

[8] DJI. *DJI Onboard SDK*. [Online; accessed 14-June-2021]. URL: https://developer.dji.com/onboard-sdk/.

[9] DJI. *DJI Payload SDK*. [Online; accessed 14-June-2021]. URL: https://developer.dji.com/payload-sdk/.

[10] DJI. *DJI User-Experience SDK*. [Online; accessed 14-June-2021]. URL: https://developer.dji.com/mobile-sdk/documentation/introduction/ux_sdk_introduction.html.

[11] DJI. *DJI Windows SDK*. [Online; accessed 14-June-2021]. URL: https://developer.dji.com/document/900519f4-f89d-4458-a8a3-0f38f289f7ad.

[12] Djibestdrones. *DJI OcuSync 2.0: What You Need to Know About This FPV Transmission System*. [Online; accessed 14-June-2021]. URL: http://djibestdrones.com/dji-ocusync-2-0/.

[13] Edström V., Zeynalli E. "Penetration testing a civilian drone". http://kth.diva-portal.org/smash/get/diva2:1463784/FULLTEXT01.pdf. Bachelor's Thesis. KTH Royal Institute of Technology, July 2020.

[14] FFmpeg. *FFmpeg Multimedia Framework*. [Online; accessed 25-August-2021]. URL: https://ffmpeg.org/about.html.

[15] Frankel E. S., Eydt B., Owens L., Scarfone K. K. "Establishing wireless robust security networks: A guide to IEEE 802.11 i". In: *NIST Special Publication* 800–97 (2007).

[16] Heliguy. *DJI Transmission Systems - Wi-Fi, Ocusync & Lightbridge*. [Online; accessed 23-September-2021]. URL: https://www.heliguy.com/blogs/posts/dji-transmission-systems-wi-fi-ocusync-lightbridge.

[17] Intel. *Intel Dual Band Wireless-AC 3165*. [Online; accessed 16-July-2021]. URL: https://ark.intel.com/content/www/us/en/ark/products/89450/intel-dual-band-wireless-ac-3165.html.

[18] James M., Walmsley D. *WPF Developers Tips*. [Online; accessed 27-September-2021]. URL: https://docs.avaloniaui.net/misc/wpf.

[19] Jondral F. K. "Software-defined radio—basics and evolution to cognitive radio". In: *EURASIP Journal on Wireless Communications and Networking* 2005.3 (Dec. 2005), pp. 1–9. DOI: 10.1155/WCN.2005.275.

[20] Katz J., Lindell Y. *Introduction to modern cryptography*. CRC press, 2020. ISBN: 978-1-46657027-6.

[21] Kenington P. B. *RF and baseband techniques for software defined radio*. Artech House, 2005. ISBN: 978-1-58053793-3.

[22] Kostao, Quaddamage. *Lightbridge and OcuSync protocol description*. [Online; accessed 23-September-2021]. URL: https://phantompilots.com/threads/lightbridge-and-ocusync-protocol-description.147896/.

[23] Krishnam R. K. V., Vallikumari V., Raju K. "Modeling and analysis of IEEE 802.11i WPA-PSK authentication protocol". In: *2011 3rd International Conference on Electronics Computer Technology*. Vol. 5. IEEE, Apr. 2011, pp. 72–76. DOI: 10.1109/ICECTECH.2011.5941959.

[24] The Local. "Drone nearly collides with Austrian rescue helicopter". In: *Local Austria* (Aug. 2016). URL: https://www.thelocal.at/20160816/drone-nearly-collides-with-rescue-helicopter-in-austria.

[25] Morgan C. *SharpPcap*. [Online; accessed 27-September-2021]. URL: https://github.com/chmorgan/sharppcap.

[26] Noh J., Kim J., Kwon G., Cho S. "Secure key exchange scheme for WPA/WPA2-PSK using public key cryptography". In: *2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. IEEE, Oct. 2016, pp. 1–4. DOI: 10.1109/ICCE-Asia.2016.7804782.

[27] Ondiwa N. O., Biermann E., Noel G. "An integrated security model for WLAN". In: *AFRICON 2009*. IEEE, Sept. 2009, pp. 1–6. DOI: 10.1109/AFRCON.2009.5308183.

[28] *Online CRC-8 CRC-16 CRC-32 Calculator*. [Online; accessed 19-August-2021]. URL: https://crccalc.com/.

[29] Original Gangsters. *Dji-firmware-tools - Communication Dissector Example*. Aug. 2021. URL: https://github.com/o-gs/dji-firmware-tools/blob/37d63d9d9be548f368f5c4a0d9f438bbd7b22161/comm_dissector/img/wireshark-using-dji-dissector.png.

[30] Original Gangsters. *Dji-firmware-tools - Communication Dissectors*. Aug. 2021. URL: https://github.com/o-gs/dji-firmware-tools/blob/bd901aff74871d1ef7362c232afb8c56ea854c45/comm_dissector/README.md.

[31] Original Gangsters. *Dji-firmware-tools - Container Stream Parser script*. Aug. 2021. URL: https://github.com/o-gs/dji-firmware-tools/blob/6570ecc8ffd6b8c396365e870a23ab6fd0c2ec2b/comm_dat2pcap.py.

[32] Original Gangsters. *Dji-firmware-tools - CRC-16 Hextable*. Aug. 2021. URL: https://github.com/o-gs/dji-firmware-tools/blob/533c4abb09544cf56ecdbb3176079d31685ed7a1/comm_dat2pcap.py#L70.

[33] Original Gangsters. *Dji-firmware-tools - CRC-8 Hextable*. Aug. 2021. URL: https://github.com/o-gs/dji-firmware-tools/blob/533c4abb09544cf56ecdbb3176079d31685ed7a1/comm_dat2pcap.py#L116.

[34] Original Gangsters. *Dji-firmware-tools - DUMLv1 Protocol*. Aug. 2021. URL: https://github.com/o-gs/dji-firmware-tools/blob/05e24cb12803943f63ac5ae1574e517e59a2dd0a/comm_dissector/wireshark/dji-dumlv1-proto.lua.

[35] Original Gangsters. *Dji-firmware-tools - Readme*. Aug. 2021. URL: https://github.com/o-gs/dji-firmware-tools/blob/40aeaca6bf7302f409de18c795927fcc12a17372/README.md.

[36] Pacheco de Carvalho J. A. R., Veiga H., Marques N., Ribeiro Pacheco C. F. F., A. Reis D. "Performance measurements of IEEE 802.11 b, g laboratory WEP and WPA point-to-point links using TCP, UDP and FTP". In: *2011 International Conference on Applied Electronics*. ISSN: 1803-7232. IEEE, Sept. 2011, pp. 1–6.

[37] PcapDotNet. *Pcap.Net*. [Online; accessed 27-September-2021]. URL: https://github.com/PcapDotNet/Pcap.Net.

[38] Wenger S., Hannuksela M. M., Stockhammer T., Westerlund M., Singer D. *RTP Payload Format for H.264 Video*. RFC 3984. IETF, Feb. 2005.

[39] Rosenbjerg M. *FFMpegCore*. [Online; accessed 27-September-2021]. URL: https://github.com/rosenbjerg/FFMpegCore.

[40] Samlaf. *Low-Level Protocol*. [Online; accessed 23-September-2021]. URL: https://tellopilots.com/wiki/protocol/#MessageIDs.

[41] Sandsmark. *Communication protocol*. [Online; accessed 23-September-2021]. URL: https://tellopilots.com/threads/communication-protocol.3093/.

[42] Santosh K., Sonam R. “Survey on transport layer protocols: TCP & UDP”. In: *International Journal of Computer Applications* 46.7 (2012), pp. 20–25. DOI: 10.5120/6920-9285.

[43] SevreNniarB. *S1 App - Advanced Debugging Mechanism*. [Online; accessed 23-September-2021]. URL: https://forum.dji.com/forum.php?mod=viewthread&tid=202546.

[44] Sivakumar C., Velmurugan A. “High Speed VLSI Design CCMP AES Cipher for WLAN (IEEE 802.11i)”. In: *2007 International Conference on Signal Processing, Communications and Networking*. IEEE, Feb. 2007, pp. 398–403. DOI: 10.1109/ICSCN.2007.350770.

[45] Sorensen E., Mikailesc M. “Model-view-ViewModel (MVVM) design pattern using Windows Presentation Foundation (WPF) technology”. In: *MegaByte Journal* 9.4 (2010), pp. 1–19.

[46] Spyridon S., David C. “The CIA Strikes Back: Redefining Confidentiality, Integrity and Availability in Security”. In: *Journal of Information System Security* 10.3 (2014), pp. 21–45.

[47] Timbrook R. *Lightbridge and OcuSync protocol description*. [Online; accessed 23-September-2021]. URL: https://expertworldtravel.com/what-is-dji-ocusync/.

[48] Tuttlebee W. *Software defined radio: enabling technologies*. John Wiley & Sons, Apr. 2003. ISBN: 978-0-47085263-7.

[49] Ulversoy, Tore. “Software Defined Radio: Challenges and Opportunities”. In: *IEEE Communications Surveys & Tutorials* 12.4 (2010), pp. 531–550. DOI: 10.1109/SURV.2010.032910.00019.

[50] Vachtsevanos, George J., Valavanis, Kimon P. “Military and Civilian Unmanned Aircraft”. In: *Handbook of Unmanned Aerial Vehicles*. Ed. by Valavanis, Kimon P., Vachtsevanos, George J. Dordrecht: Springer Netherlands, 2015, pp. 93–103. DOI: 10.1007/978-90-481-9707-1_96.

[51] Videolan. *Libvlcsharp*. [Online; accessed 27-September-2021]. URL: https://github.com/videolan/libvlcsharp.

[52] Waldmann N. *DJI OcuSync 2.0 / 3.0: Der ultimative OcuSync & Lightbridge Guide*. [Online; accessed 14-June-2021]. URL: https://www.drone-zone.de/dji-ocusync-2-0-der-ultimative-ocusync-lightbridge-guide/.

[53] Warner J. S., Johnston R. G. "GPS spoofing countermeasures". In: *Homeland Security Journal* 25.2 (2003), pp. 19–27.

[54] Wild G., Murray J., Baxter G. "Exploring civil drone accidents and incidents to help prevent potential air disasters". In: *Aerospace* 3.3 (2016), p. 22. DOI: DOI:10.3390/aerospace3030022.

[55] Wireshark. *Building Display Filter Expressions*. [Online; accessed 06-August-2021]. URL: https://www.wireshark.org/docs/wsug_html_chunked/ChWorkBuildDisplayFilterSection.html#ChWorkBuildDisplayFilterMistake.

[56] Wireshark. *Wireshark Dissectors*. [Online; accessed 27-August-2021]. URL: https://www.wireshark.org/docs/wsdg_html_chunked/ChapterDissection.html.

[57] Xu F., Muneyoshi H. "A Case Study of DJI, the Top Drone Maker in the World". In: *Kindai Manag. Rev* 5 (2017), pp. 97–104.

[58] Zeng Y., Zhang R., Lim T. J. "Wireless communications with unmanned aerial vehicles: Opportunities and challenges". In: *IEEE Communications Magazine* 54.5 (2016), pp. 36–42. DOI: 10.1109/MCOM.2016.7470933.

A Software Design UML Diagrams

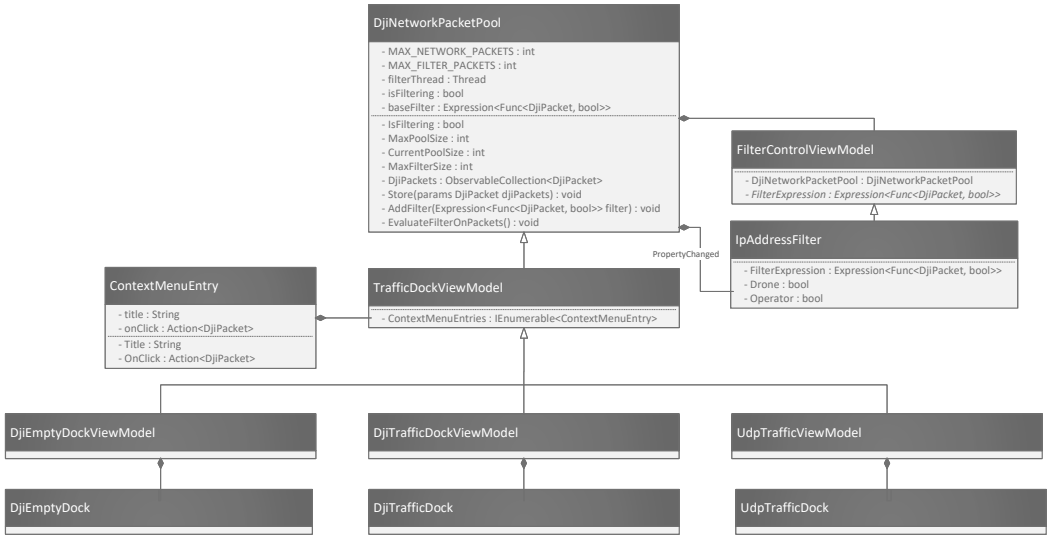


Figure A.1: Dji UI Class Diagram

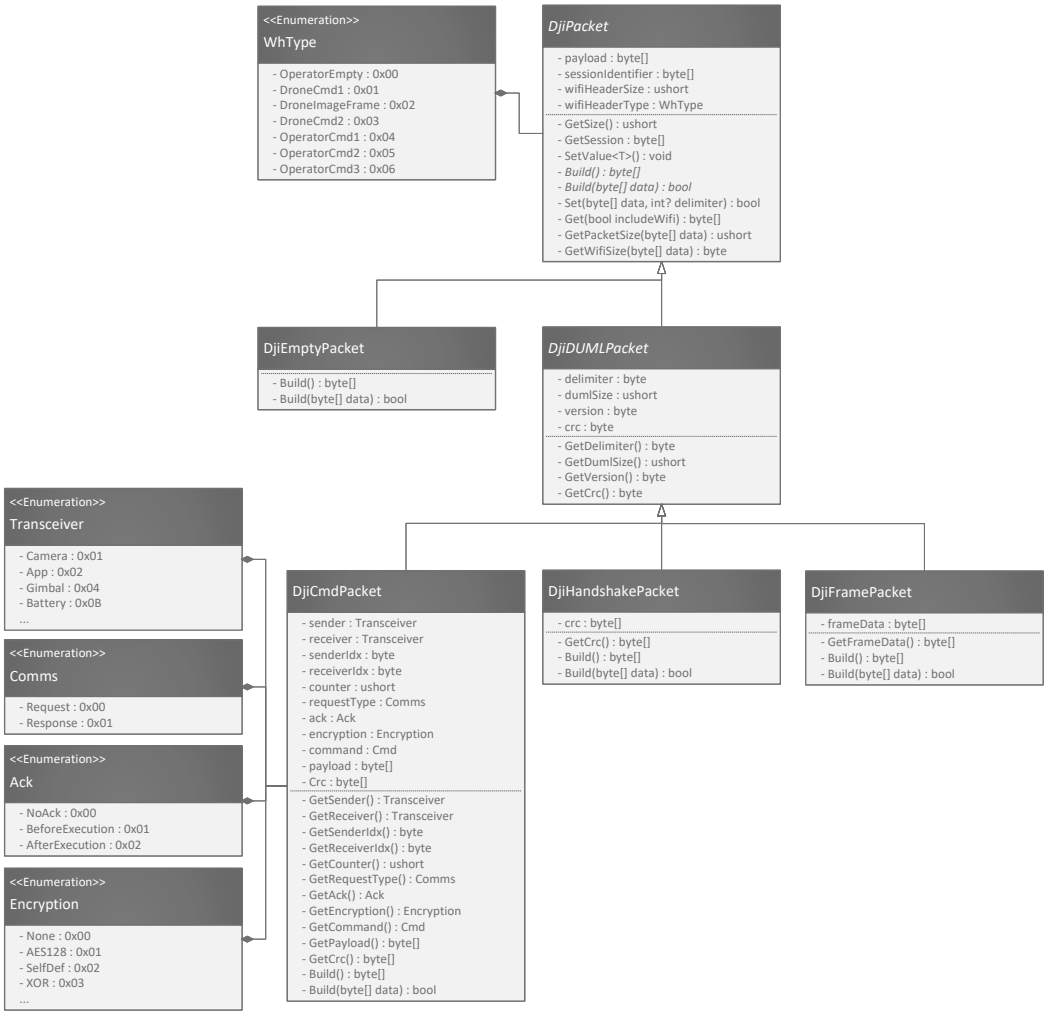


Figure A.2: Dji Network Packet Class Diagram

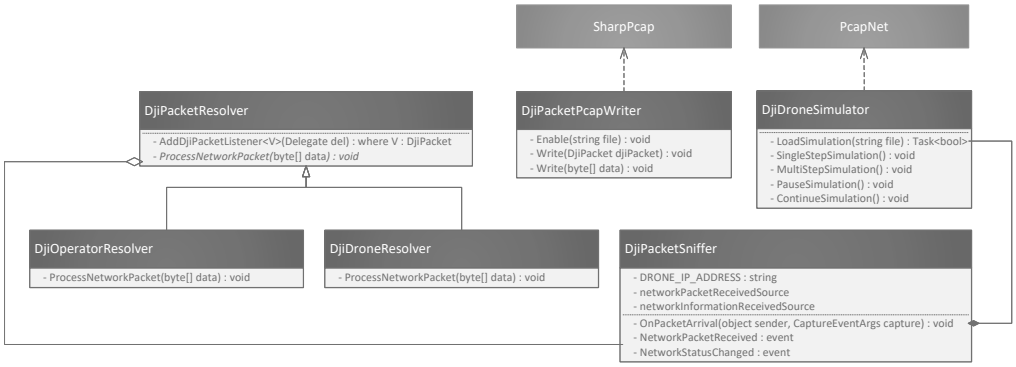


Figure A.3: Dji Network Class Diagram

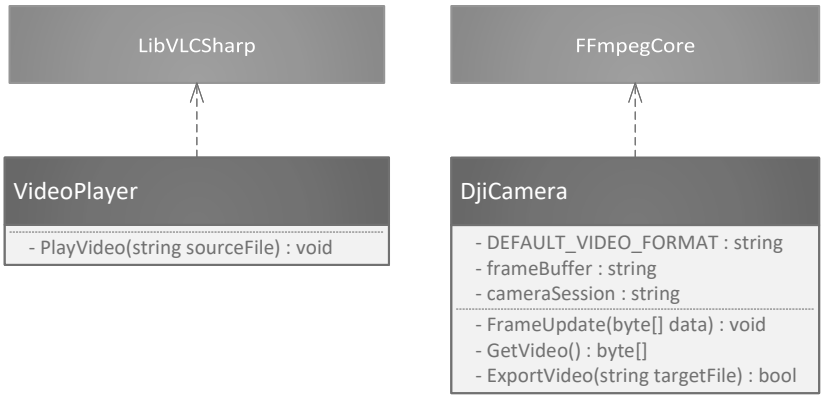


Figure A.4: Dji Camera Class Diagram

B DJI Source Code Snippets

Listing B.1: Cmd `con(0x03)` Cmd-Set `con(0x0D)` concrete implementation example

```
1  public class DjiBatteryPowerLevel : DjiCmdPacket
2  {
3      private byte[] _cellCount, _cell1, _cell2, _cell3;
4
5      public float BatteryLevel =>
6          (BitConverter.ToInt16(_cell1) +
7          BitConverter.ToInt16(_cell2) +
8          BitConverter.ToInt16(_cell3)) / (3830f * 3);
9
10     protected override byte[] Build()
11     {
12         byte[] data = new byte[7];
13         data[0] = _cellCount;
14         data[1] = _cell1[0];
15         data[2] = _cell1[1];
16         data[3] = _cell2[0];
17         data[4] = _cell2[1];
18         data[5] = _cell3[0];
19         data[6] = _cell3[1];
20
21         return base.Build().Append(data);
22     }
23
24     protected override bool Build(byte[] data)
25     {
26         if (!base.Build(data))
27             return false;
28         if (data.Length != 7)
29             return false;
30
31         _cellCount = data[0];
32         _cell1 = data[1..3];
33         _cell2 = data[3..5];
34         _cell3 = data[5..7];
35
36         return true;
37     }
38 }
```

Listing B.2: IpAddressFilterViewModel implementation

```
1  public class IpAddressFilterViewModel : FilterControlViewModel
2  {
3      private bool _operator;
4      private bool _drone;
5
6      public IpAddressFilterViewModel()
7      {
8          this.WhenAnyValue(instance => instance.Drone).Subscribe(drone =>
               DjiNetworkPacketPool?.EvaluateFilterOnPackets());
9          this.WhenAnyValue(instance => instance.Operator).Subscribe(op =>
               DjiNetworkPacketPool?.EvaluateFilterOnPackets());
10     }
11
12     protected override Expression<Func<DjiPacket, bool>> FilterExpression
           => (djiPacket) =>
13         (!Drone && !Operator) ||
14         (djiPacket.Participant == Participant.Drone && Drone) ||
15         (djiPacket.Participant == Participant.Operator && Operator);
16
17     public bool Drone
18     {
19         get => _drone;
20         set => this.RaiseAndSetIfChanged(ref _drone, value);
21     }
22
23     public bool Operator
24     {
25         get => _operator;
26         set => this.RaiseAndSetIfChanged(ref _operator, value);
27     }
28 }
```

Listing B.3: FilterControlViewModel implementation

```
1  public abstract class FilterControlViewModel : ReactiveObject
2  {
3      private DjiNetworkPacketPool _networkPool;
4
5      public FilterControlViewModel()
6      {
7          this.WhenAnyValue(instance => instance.DjiNetworkPacketPool).
               Subscribe(networkPool => networkPool?.AddFilter(
               FilterExpression));
8      }
9
10     public DjiNetworkPacketPool DjiNetworkPacketPool
11     {
12         get => _networkPool;
13         set => this.RaiseAndSetIfChanged(ref _networkPool, value);
14     }
15
16     protected abstract Expression<Func<DjiPacket, bool>> FilterExpression
           { get; }
17 }
```

Listing B.4: CmdAttribute implementation

```
1  public class CmdAttribute : Attribute
2  {
3      public CmdAttribute(ushort data, byte cmdSet, string cmdSetDescription
           , byte cmd, string cmdDescription) =>
4          (Data, CmdSet, CmdSetDescription, Cmd, CmdDescription) = (data,
               cmdSet, cmdSetDescription, cmd, cmdDescription);
5
6      public ushort Data { get; init; }
7
8      public byte Cmd { get; init; }
9
10     public byte CmdSet { get; init; }
11
12     public string CmdDescription { get; init; }
13
14     public string CmdSetDescription { get; init; }
15 }
```

Listing B.5: CmdSet.java: Command-set enum values

```
public enum CmdSet
{
    COMMON(0, new CmdIdCommon()),
    SPECIAL(1, new CmdIdSpecial()),
    CAMERA(2, new CmdIdCamera()),
    FLYC(3, new CmdIdFlyc()),
    GIMBAL(4, new CmdIdGimbal()),
    CENTER(5, new CmdIdCenter()),
    RC(6, new CmdIdRc()),
    WIFI(7, new CmdIdWifi()),
    DM368(8, new CmdIdDm368()),
    OSD(9, new CmdIdOsd()),
    EYE(10, new CmdIdEYE()),
    SIMULATOR(11, new CmdIdSimulator()),
    BATTERY(12),
    SMARTBATTERY(13, new CmdIdSmartBattery()),
    ADS_B(17, new CmdIdADS_B()),
    Glass(21, new CmdIdGlass()),
    Flight(31, new CmdIdFlight()),
    RTK(15, new CmdIdRTK()),
    Module4G(24, new CmdIdModule4G()),
    OnboardSDK(25, new CmdIdOnBoardSDK()),
    NarrowBand(32, new CmdIdNarrowBand()),
    FLYC2(34, new CmdIdFlyc2()),
    PayloadSDK(60, new CmdIdPayloadSDK()),
    RECOGNIZE(238, new CmdIdRecognize()),
    OTHER(100);
}
```

Listing B.6: CmdIdADS_B.java: Command enum values

```
public enum CmdIdType implements CmdIdInterface
{
    GetPushData(2, false, DataADSBGetPushData.class),
    GetPushWarning(8, false, DataADSBGetPushWarning.class),
    GetPushOriginal(9, false, DataADSBGetPushOriginal.class),
    SendWhiteList(16),
    RequestLicense(17),
    SetLicenseEnabled(18),
    GetLicenseId(19),
    GetPushUnlockInfo(20, false, DataADSBGetPushUnlockInfo.class),
    SetUserId(21),
    GetKeyVersion(22),
    GetPushAvoidanceAction(23, false, DataADSBAvoidanceAction.class),
    Other(FrameMetricsAggregator.EVERY_DURATION);
}
```