Drone Security and the Mysterious Case of DJI's DroneID

Nico Schiller*, Merlin Chlosta‡, Moritz Schloegel*, Nils Bars*, Thorsten Eisenhofer*,
Tobias Scharnowski*, Felix Domke§, Lea Schönherr‡, Thorsten Holz‡

**Ruhr University Bochum*
§*Independent*
‡*CISPA Helmholtz Center for Information Security*

***Abstract*—Consumer drones enable high-class aerial video photography, promise to reform the logistics industry, and are already used for humanitarian rescue operations and during armed conflicts. Contrasting their widespread adoption and high popularity, the low entry barrier for air mobility—a traditionally heavily regulated sector—poses many risks to safety, security, and privacy. Malicious parties could, for example, (mis-)use drones for surveillance, transportation of illegal goods, or cause economic damage by intruding the closed airspace above airports. To prevent harm, drone manufacturers employ several countermeasures to enforce safe and secure use of drones, e. g., they impose software limits regarding speed and altitude, or use geofencing to implement no-fly zones around airports or prisons. Complementing traditional countermeasures, drones from the market leader DJI implement a tracking protocol called DroneID, which is designed to transmit the position of both the drone and its operator to authorized entities such as law enforcement or operators of critical infrastructures.**

In this paper, we analyze security and privacy claims for drones, focusing on the leading manufacturer DJI with a market share of 94%. We first systemize the drone attack surface and investigate an attacker capable of eavesdropping on the drone's over-the-air data traffic. Based on reverse engineering of DJI firmware, we design and implement a decoder for DJI's proprietary tracking protocol DroneID, using only cheap COTS hardware. We show that the transmitted data is *not* encrypted, but accessible to anyone, compromising the drone operator's privacy. Second, we conduct a comprehensive analysis of drone security: Using a combination of reverse engineering, a novel fuzzing approach tailored to DJI's communication protocol, and hardware analysis, we uncover several critical flaws in drone firmware that allow attackers to gain elevated privileges on two different DJI drones and their remote control. Such root access paves the way to disable or bypass countermeasures and abuse drones. In total, we found 16 vulnerabilities, ranging from denial of service to arbitrary code execution. 14 of these bugs can be triggered remotely via the operator's smartphone, allowing us to crash the drone mid-flight.

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24217
www.ndss-symposium.org

I. Introduction

Civil drones are a defining element of the 21st century. Their low-cost and high-quality image capabilities enable innovations in aerial photography for journalism and film, geographic mapping of inaccessible terrain and locations [64], information gathering of disaster situations [35], and humanitarian rescue operations with thermal sensors [50]. Companies such as Amazon plan to deliver packages from the air [34], and events like the Tokyo Olympics opening ceremony featured a drone-based light show [59].

At the same time and in stark contrast to these various opportunities, consumer drones pose a significant threat of (mis-)use in criminally motivated scenarios. For example, drones can easily bypass physical boundaries such as prison walls or national border fences [53]. Thus, they open up a new vector for the delivery of illegal goods such as drugs [45], [29] with high mobility and low risk of detection. Flying within or near restricted airspace can also cause significant economic damage. For instance, regulations require airports to cease operations as soon as a drone sighting is reported, causing high financial costs and potentially affecting thousands of people [40]. To mitigate the risks of these abuse scenarios, drone vendors commonly employ several countermeasures: *Geofencing* prevents drones from entering restricted airspace, while software limits prevent operators from going too fast or too high. Beyond actively preventing harmful actions, drones from market leader DJI regularly transmit their position and the operator's position via a proprietary protocol called *DroneID*, allowing law enforcement or critical infrastructure operators to identify and track drones as well as their pilots. While this allows to locate and apprehend malicious drone operators effectively, transmitting sensitive location data can be problematic, e. g., if law enforcement uses drones for surveillance purposes. Additionally, recent European conflicts have seen armed forces and civilians use consumer drones to track the movement and positioning of hostile forces or guide artillery strikes. In both cases, drone users rely on the privacy of sensitive location data not being accessible.

Despite these potential security, safety, and privacy risks, and the existing abuse of consumer drones, only a few scientific works assess the security and privacy of civil drones.

Nassi et al. [43] systematically analyze the academic state-of-the-art in this regard. Various aspects such as Global Positioning System (GPS) spoofing [1] and de-authentication attacks [10], [49], [46] were analyzed previously. Critically, however, system security aspects of existing drone systems have not yet been comprehensively assessed. Most existing research uses either low-cost or open-source drones, which are not as widely used as commercial high-end drones. The few works that address such widespread drones, like DJI products, are limited to the DJI Go 4 app [63] or they perform only a basic analysis of the Phantom 3 [65], [6] or Phantom 4 [16] drones. To the best of our knowledge, no further attempts were made to study the software or hardware level in detail.

In this work, we present a comprehensive analysis of security and privacy claims of state-of-the-art consumer drones from DJI, the market leader of consumer drones in terms of sales [58], [60] and security measures employed [20]. We consider two threat models where (i) a passive attacker eavesdrops on traffic transmitted over-the-air and (ii) a scenario where an active attacker has physical access to the drone or a device connected to the drone. For the former, we especially focus on DJI's *DroneID* protocol, and its impact on the operator's privacy. Widespread public belief – nurtured by only recently withdrawn vendor statements [30] – is that the transmitted position data is encrypted and can only be decrypted and decoded by law enforcement or operators of critical infrastructure. We reverse-engineered both the firmware and the software related to the wireless physical layer to examine how packets using DroneID are modulated and encoded. With the insights gained from reverse engineering, we built the complete pipeline required to receive, demodulate, and decode DroneID packets and reveal that this protocol is *not* encrypted. Based only on standard components such as a Software Defined Radio (SDR), our findings show that the packets accurately disclose the location of the drone and, more importantly, the home point and the location of the pilot. We identify a way of disabling the transmission of this data and show that it is also possible to spoof the pilot's position with an off-the-shelf GPS spoofing app.

Beyond a passive attacker eavesdropping on the data transmitted via a radio interface, we consider an active attacker with physical access to the drone, remote control, or the connected smartphone. We analyze the full system stack, including the hardware and software of the drone itself and the remote control. To dynamically analyze the drone's firmware, we design and implement a black-box fuzzer with a customized grammar of the drone's proprietary communication protocol called DUML. Fuzzing a complex cyber-physical system such as a drone is challenging because we typically do not have access to the complete firmware and thus cannot use any existing fuzzer that relies on instrumentation. In addition, faults do not necessarily manifest in a complete crash, but often only in an inconsistent state. We present a novel bug oracle that monitors DJI's Android app connected to the remote control for subliminal but unexpected changes. This allows us to effectively identify erroneous patterns and uncover faults that lead to an inconsistent state, which conventional fuzzers cannot detect. Using this approach, we successfully identify several security-critical bugs that allow an attacker to gain privileged access, a pre-condition for disabling many countermeasures, to execute arbitrary code, or to spoof their identity by forging the drone's serial number. Our analysis further uncovers a safety-critical remote attack vector, where drones can be crashed mid-flight. Static analysis of the firmware yields further critical bugs, which allow arbitrary command execution. Overall, we find 16 vulnerabilities, with a severity ranging from *low* to *critical*, in four out of five tested DJI devices.

Contributions. In summary, our main contributions are

- *Security Analysis.* We present a comprehensive security analysis of market leader DJI's consumer drones. This analysis includes the hardware, software, and wireless physical layer of the drone itself and the remote control. Our analysis uncovers multiple critical security vulnerabilities that allow for arbitrary command execution and bypassing of countermeasures, or even remotely crashing drones mid-flight.
- *DroneID.* We reverse engineer both the firmware and wireless physical layer of current DJI drones to analyze DJI's proprietary transmission protocol called *OcuSync*. Based on these findings, we present a combination of a receiver, demodulator, and decoder for DJI's DroneID packages to reveal sensitive information, including the live positions of both the drone and the remote pilot.
- *Drone Fuzzing.* We design and implement a custom generational black-box fuzzer that combines a DJI-specific grammar with a novel bug oracle to identify faults in drones and their remote controls. Our fuzzer uncovers security-critical crashes that can be used to gain root access or crash drones in the air.

To foster future research on this topic, we open-source our fuzzing framework and our DroneID receiver at https://github.com/RUB-SysSec/DroneSecurity. Following a responsible disclosure process, DJI has fixed all bugs.

II. Primer on DJI drones

As a first step, we provide an overview of the most important aspects of modern consumer drones, focusing specifically on drones from the manufacturer DJI. This company is the world's leading manufacturer of commercial drones in 2021, with a total market share of approximately 54 % by units (as of late 2021) and 94% in the consumer drones segment [58]. Furthermore, DJI provides a whitepaper on their security and safety mechanisms [20], thus establishing a solid baseline in terms of security and safety. Given the practical importance of DJI, in this work we focus on consumer drones from this vendor with a weight between 200g and 1kg. In this work, we use three different DJI drone models and their corresponding Remote Controls (RCs) for our security analysis:

- DJI Mini 2, RC: RC231
- Mavic Air 2, RC: RC231
- Mavic 2 Pro / Zoom, RC: RC1B

Furthermore, we reproduce our findings on the latest DJI drone, the Mavic 3 (which was unavailable during initial analysis). The results presented in this section are based on our analysis of several recent DJI drones and reverse engineering of different firmware images.

A. Communication Interfaces and Protocols

In the following, we provide an overview of a typical drone's interfaces and communication protocols. These aspects are particularly interesting because they are directly exposed to attackers.

Figure 1 provides a broad overview of the different interfaces of a DJI drone and how these are used to communicate between the drone, the Remote Control (RC), and a computer. While the drone and the RC communicate during the operation of the drone, the computer is only used to analyze, update, or access files on the drone or the RC.

a) ***USB****:* Both drones and RCs typically have a USB interface, which is used for various device classes and use cases. The intended use case is to download data, such as media files from the internal storage or flight logs from the flight recorder. For DJI devices, the USB interface is also used to send *DJI Universal Markup Language (DUML)* commands—a proprietary communication protocol designed by DJI—to control the internal settings of the device or to initiate firmware updates. In addition, a bootloader is exposed during the device's boot process, which can be triggered via specific USB packets. Furthermore, on the RCs (RC231) there are two USB interfaces, one to connect the smartphone with the RC for access with the *DJI Fly* app and a second one to charge the remote control. The charging port can also be used to connect the remote control to a computer.

b) ***UART****:* A Universal Asynchronous Receiver-Transmitter (UART) interface is primarily used in microcontrollers for wired communication. Such UART interfaces can also be found in DJI drones, e. g., on the *Sparrow S1 transceiver* (called S1 System on a Chip (SoC), see Section C2 for details). During the hardware analysis process, we found that the interface is enabled for two seconds after startup and displays the boot screen of the bootloader.

c) ***Wireless Physical Layer: Bluetooth, WiFi, and OcuSync****:* The latest DJI Drones support different wireless protocols like Bluetooth and WiFi, e. g., to transfer photos taken by the camera of the drone to the smartphone using the *DJI Fly* app. A WiFi connection is also available on older drones like the Mavic Pro or Mavic Air, but is used for the Command & Control (C2) link to control drones via smartphones. This communication link has been replaced on newer DJI drones by the proprietary *OcuSync* radio protocol, which performs significantly better and is less prone to interference.

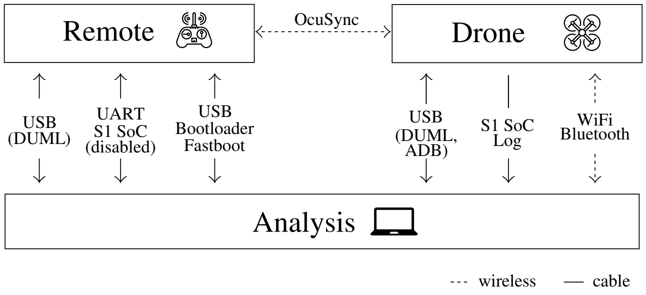


Fig. 1. Generic overview of the different interfaces of a DJI Mini 2 and the corresponding remote control RC231 as an example of a typical drone. USB ADB requires root privileges on the drone, which are disabled by default. UART is hidden and requires reverse engineering of the hardware pins.

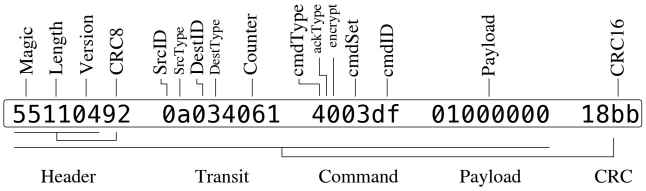


Fig. 2. The structure of a DUML packet. This example shows a request for the DJI Assistant unlock command.

d) ***DUML****:* The DJI Universal Markup Language format is a proprietary communication protocol used by DJI, for example, to send commands and data between internal modules and from the RC to the drone [36]. DUML is used to set and change parameters of the drone, such as the flight parameters for maximum altitude or different speed parameters like the maximum ascent and descent speed. Since the parsing process of this protocol is prone to errors, DUML is an interesting target for further security analysis. Therefore, we provide a detailed overview of this custom protocol below.

DJI distinguishes between two DUML protocol versions: *V1* and *Logic*. The Logic protocol is used for internal communication between the individual modules of the drone, while V1 is used for communication between the computer and the drone via USB. DJI does not publicly document the possible commands, but many of them have been documented by the DJI reversing community [37] We also reverse engineered the different parts of the firmware to confirm the structure.

The structure of a DUML packet, as depicted in Figure 2, can be divided into four parts: *header*, *transit*, *command*, and *payload*. The header contains a magic byte set to the fixed value of `0x55`, followed by the length of the packet, the version number, and a CRC-8 header checksum. The header is followed by the transit data, which is used to set the sender/receiver of the DUML command. According to previous work [37], there are 32 known sender and receiver types (e. g., `PC`, `Mobile App`, `Center Board`, and `WiFi`). Besides the source and destination, the transit part of the packet contains a unique sequence number to maintain the order of multiple packets.

The third part of a DUML packet is the command itself. Here the fields `commandtype`, `commandset`, and the `commandid` are defined. Based on the `commandtype`, the acknowledgment type, the packet type, and the encryption type are set. The `commandset` specifies which set of commands are used for the `commandid`. According to [37], there are 17 known `commandsets` that can be used, e. g., `general`, `flight controller`, `WiFi`, or `battery`. We confirmed these findings based on our reverse engineering results. Finally, the last part of the packet is the payload and a CRC-16 checksum calculated over the entire packet. For efficiency reasons, a custom encoding is used where the information is tightly packed. This encoding and decoding is summarized in Listings 14 and 15 in the appendix, respectively. Later, we will base our dynamic security tests on the recovered DUML format and use it to build a custom fuzzing framework (see Section D for details).

e) ***DJI Fly / DJI Go 4 App:*** DJI provides an app for iOS and Android to display the live video feed and additional controls. The latest drones use the *DJI Fly app*, while legacy drones use the *DJI Go 4* app. Both apps offer essentially the same functionality, in particular, both expose the live video feed of the drone, allow to take pictures and record videos, change different drone settings, update the firmware, or check the general status of the drone.

A 2020 security analysis by Synacktiv concluded that the DJI Go 4 app contains questionable features, such as an auto-updater that is allegedly breaking Play Store terms of service [63]. DJI responded that their auto-updating feature aids the integrity of no-fly zones [19]. The app has been unlisted from the Google Play Store in 2021 without official explanation, with the DJI website now functioning as the main distribution channel. Installing or updating the application requires users to accept installations from untrusted third-parties, a feature that is disabled by default for security reasons. In particular, this evades any security and privacy measures enforced by the app store.

B. Drone Firmware

Depending on their complexity, the drones we analyzed use different operating systems (OSes). For most DJI drones, the OS is based on Android which is also used on older versions of their remote controls (e. g., Mavic Pro RC GL200). The DJI Mini series uses a 32-bit ARM Linux based on Buildroot [4]. For time- and performance-critical operations, real-time operating systems (RTOS) are used, e. g., the flight controller and newer RCs such as the RC231 use two RTOSes in the transceiver to manage the Radio Frequency (RF) connection — an application and a communication processor. The firmware of the RTOSes in the transceiver is available as encrypted binaries for both processors. The OSes (except for the RTOS firmware) are based on the BusyBox software suite, which is commonly used in embedded systems. DJI encodes its firmware in a proprietary file format encrypted with AES and signed with RSA. Different modules, files, and use cases, such as firmware updates or transfers, use different encryption keys. Some of these encryption keys have been leaked by individuals from the DJI community to decrypt parts of the firmware; we confirmed that these keys are valid.

C. Drone Hardware

Next, we present an overview of the typical hardware components used in the drones we analyzed. Figure 3 shows a general overview of the essential components of such a drone, using the DJI Mini 2 as an example. The figure also illustrates how the individual components are linked together and how they communicate with each other. This overview and the following description are transferable to other DJI drone models that only differ in the components' performance or feature additional sensors.

*a) **Flight Controller:*** The flight controller is the most critical part of a drone and must be reliable, predictable, and deterministic under all circumstances. To ensure these properties, the flight controller uses an RTOS. The controller supervises the flight behavior by gathering inputs from sensors like the Inertial Measurement Unit (IMU), compass, GPS, or Visual Positioning System (VPS). It uses this information to maintain a stable flight by sending instructions to the Electronic Speed Control (ESC) controllers, which regulate the speed of the four motors. Furthermore, it is responsible for approving the flight and allowing the drone to takeoff by checking the status of ESCs, battery, and other modules. In addition, it also checks the internal No-Fly-Zone (NFZ) database to determine if the drone is prohibited from entering specific areas of nearby airspace. The flight controller receives the control commands through the transceiver, here the S1 SoC, and reacts to the user inputs received from this SoC. Moreover, the flight controller sends the information from the different modules and sensors to the transceiver.

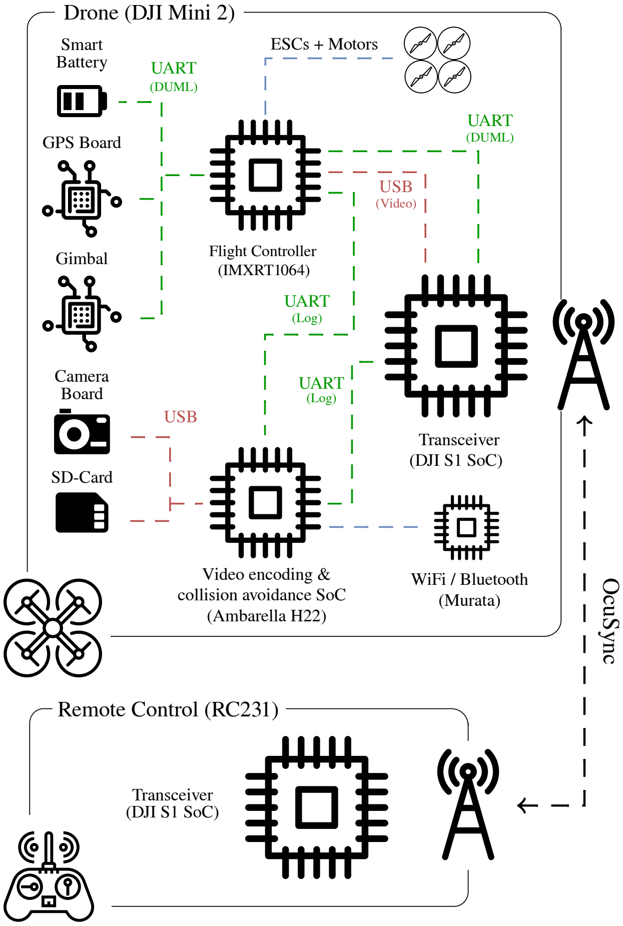


Fig. 3. Schematic overview of a typical DJI drone and its components. This overview shows the basic internal components of a DJI drone, here using a DJI Mini 2 as an example.

*b) **Video encoding & collision avoidance SoC:*** This SoC handles the image sensor and video encoding. It receives the video and image data from the camera, processes it, and forwards it to the transceiver. If additional sensors are installed for collision avoidance, their data is also processed in this SoC. Besides sending the video data to the transceiver, the SoC uses USB interfaces to store the pictures or videos on the internal storage or SD-Card. Finally, it handles the drone's firmware update process. This SoC uses a Linux or an Android-based operating system for DJI drones.

*c) **Remote Control:*** The RC controls the drone and the camera via the C2 link. The user input from the controller and smartphone is sent to the transceiver, which modulates the signal and transmits it via the *OcuSync* protocol. Furthermore, the RC receives the drone's downlink data, which includes telemetry data and the video feed, which is then passed to the smartphone. The RC231, which also uses the S1 SoC as transceiver, is supported by the latest DJI drones like the DJI Mini 2, Mavic Air 2, Mavic Air 2s, and Mavic 3. These drones use the OcuSync transmission protocol for C2 and downlink. The OcuSync version selected depends on the actual drone model; this can be OcuSync 2.0, 3.0, or 3+.

d) ***Transceiver:*** The transceiver combines a transmitter and receiver for radio communication and is an essential part of the drone. It works via a proprietary protocol or wireless standards such as Bluetooth or WiFi. Several of the latest DJI drones use the so-called *Sparrow S1 transceiver* for OcuSync transmissions. This transceiver is a proprietary SoC based on an ARM Cortex-M CPU and can be found in the Mini 2 and Mavic Air 2. The Mavic Air 2s and the Mavic 3 use as transceiver the so-called P1 (Pigeon) SoC. We are focusing on the S1 SoC in this work.

This SoC is used as a transceiver for OcuSync and contains two RTOSes for the application processor and the communication processor. The firmware for the RTOS is called *Sparrow Firmware* and can be found in firmware updates for the RC as well as in the drone's filesystem. The transceiver in the drone receives telemetry data and other reports from the flight controller and video data from the video encoding SoC. All of this data is then RF modulated and transmitted through the antennas. Furthermore, the transceiver receives the C2 link from the remote controller and passes this data to the flight controller.

e) ***WiFi / Bluetooth Chip:*** WiFi or Bluetooth chips can also be used as a transceiver, allowing the connection to an RC or a smartphone to control the drone. However, this setup has the disadvantage that range and reliability are not as good as with a proprietary radio protocol. The DJI Mini 2 and Mavic 3 have an additional WiFi and Bluetooth chip that allows users to access media files stored on the drone easily.

f) ***Additional Sensors and Other Hardware:*** Most current consumer drones have a high-resolution camera attached to a gimbal. This gimbal compensates the drone's movements and ensures a steady image. Besides the primary camera, drones can also use additional cameras for collision avoidance. Other sensors can include infrared or ultrasonic sensors to measure the drone's altitude, or accelerometers and gyroscopic sensors to measure acceleration and tilt, respectively.

D. Wireless Physical Layer

The current generation of DJI drones uses the proprietary OcuSync protocol for wireless transmission in the unregulated 2.4 GHz and 5 GHz ISM bands. The drone transmits a high-bandwidth, live video feed to the RC and subsequently to the connected smartphone. The RC controls the drone via C2 signals. According to DJI, both uplink and downlink are AES-256 encrypted [20]. OcuSync transmissions in the 2.4 GHz and 5 GHz ISM bands have a range of about 15km [18].

According to the FCC ID database, the devices use 20 MHz wide channels as the downlink (from drone to remote) and Orthogonal Frequency Division Multiplexing (OFDM) signals. The control uplink uses frequency hopping with narrower signals [15]. Earlier drones used "Enhanced WiFi" that could be sniffed using WiFi cards. In contrast, there are no openly available OcuSync-compatible receivers that allow decoding the signals from a drone or a RC.

III. Security Analysis without Physical Access

Drones are complex cyber-physical systems composed of many modules that expose various interfaces for an attacker as depicted in Figure 3. The interplay of components and interfaces requires the systematization of access capabilities to provide a comprehensive security analysis of drones. These interfaces can be divided into wireless interfaces and those that require physical access.

A. Threat Model 1: Passive Attacker

In the first part of our security analysis, we focus on a scenario where the attacker has *no* physical access to either the drone itself or the RC. Consequently, an adversary is restricted to the wireless link and broadcast signals. Here, we envision an adversary that passively listens to the wireless physical layer to detect the whereabouts of the drone as well as the accompanying RC (i. e., the pilot). In Section IV, we relax this assumption and consider adversaries with physical access to the drone and therefore a much broader capability to exploit functionalities and access critical information.

B. Wireless Link

The wireless link is a crucial attack vector because it controls the drone and can be accessed remotely. DJI drones use the proprietary *OcuSync* protocol to control the drone and to transmit the video stream to the remote control. DJI itself sells *Aeroscope*, a device that allows law enforcement to locate DJI drones and the pilots operating them. For this localization, *Aeroscope* listens for a special signal, called DroneID, which is broadcast by all DJI drones. Unfortunately, neither OcuSync nor DroneID are publicly documented and no open receivers are available. A media report [30] stated that current-generation DroneID packets are encrypted, which has been confirmed by DJI twice. Even though this has been corrected in a later version of the article (also based on our research results), it shows that the assumption that DroneIDand the transmitted Aeroscope data is encrypted is widespread in practice. DJI's security whitepaper [20] does not explicitly mention DroneID transmissions, their features, or contents. In this section, we present the results of our reverse engineering efforts of the DroneID broadcasts and show how we can successfully extract sensitive information such as the location of the drone and remote pilot. Most importantly, we debunk the claim that the protocol is encrypted.

By scanning frequency bands, we identified radio broadcasts that were detached from the high-bandwidth video feeds and the uplink control channel, with very small packet sizes and periodic occurrence. We suspected these are *DroneID* broadcasts for localization and identification of drones. DJI has maintained this feature at least since 2017. Drone regulations that require location broadcasts through *open* standards (based on WiFi or Bluetooth) are currently being drafted but have not yet been finalized (see Section V b for a discussion). Therefore, we assume that the latest drones use the OcuSync

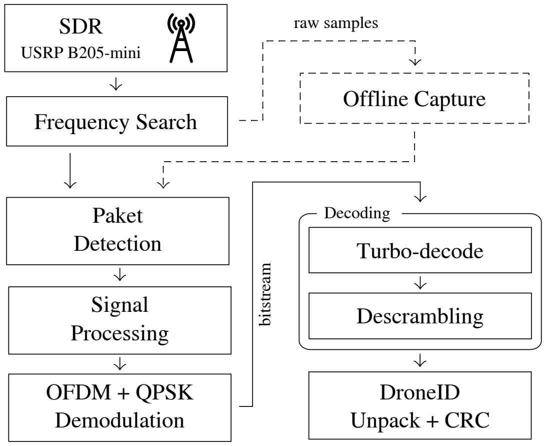


Fig. 4. A schematic overview of our DroneID receiver: Raw samples from the Software-Defined Radio are fed into the receiver chain, resulting in JSON-formatted DroneID output (see Figure 8). We describe the different steps in Section III.

protocol for such broadcasts. We analyze the packets to learn (i) what data is included in current-generation DroneID and (ii) if there are (unexpected) security features. As noted above, the wireless protocol is undocumented and requires reverse engineering. The analysis of OcuSync's DroneID is the first step towards enabling further analysis of the OcuSync protocol.

a) ***DroneID Receiver:*** We have implemented a live receiver for DroneID, using a Software-Defined Radio (SDR) for signal capture and a Python-based processing chain. Our prototype implementation contains the full pipeline of signal acquisition, demodulation, decoding, and verification via a CRC checksum. We primarily perform a step-by-step analysis to derive unknown parameters for a proprietary protocol from the radio signals, which is a challenging problem in itself.

Our implementation can receive and decode packages live. Additionally, it solves a separate challenge: Drones dynamically switch between the 2.4 GHz and 5.7 GHz band; hence, we continuously scan both bands for candidate frames and feed these candidate frames into the demodulation and decoding stages. Figure 4 shows the structure and components of our receiver and the internal processing of the received data, including frequency search, signal processing, demodulation, and decoding. The details of the system are explained in more detail below.

Our setup uses a USRP B200mini SDR that we connect to a laptop. The SDR records up to 56 MHz simultaneously in frequency ranges between 70 MHz to 6 GHz. The receiver scans every band for 1.3 seconds at 50 MHz bandwidth, resulting in 496 Mb batch sizes per band. Once frames are found, we lock the band to consecutively record DroneID frames and return to scanning only when the drone switches the channel and we have not received new packets for several cycles.

b) ***Spectrum Analysis:*** As noted above, the OcuSync and DroneID specifications are not public. Based on our analysis, it became clear that these protocols use similar modulation techniques and parameters as LTE. We reverse-engineered all subsequent parameters step-by-step. Figure 5 shows the spectrum of a single DroneID radio frame. One packet contains nine symbols, including two Zadoff-Chu (ZC) synchronization symbols (Symbol 4 and 6). The other symbols are OFDM data symbols with 601 sub-carriers (600 data and 1 DC), and a subcarrier spacing of 15 kHz. The carriers are padded to the next power of two to apply the Fast Fourier Transform (FFT) in the next step; this gives a total of 1024 sub-carriers with a total bandwidth of 15.36 MHz (including the guard bands).

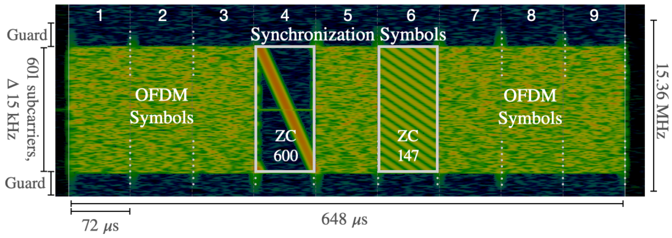


Fig. 5. Spectrum view of a DroneID packet with highlighted Zadoff-Chu synchronization symbols. The signal is the same for the 2.4 GHz and the 5.7 GHz band.

Our recordings showed that the packets are repeatedly broadcast every 640 ms. We noticed that some drones (Mavic 2 and older OcuSync drones) do not send the first symbol (Symbol 1, Figure 5), which results in a shorter frame duration of 576 μs. The other parameters remain unchanged.

c) ***Demodulation:*** The translation of radio signals into bits and bytes requires multiple steps: (i) synchronization in time to find the boundaries of the OFDM symbols and frequency synchronization to align with the OFDM subcarriers that carry the payload, (ii) channel estimation to account for distortions during the radio transmission, and (iii) demodulation of subcarriers (i. e., mapping OFDM subcarriers to bits). This section briefly outlines the steps we took for demodulation.

Time Synchronization via Cyclic Prefixes: Symbols cannot be appended directly one after another, but padding is required between them to reduce *Inter-Symbol-Interference*. For DroneID, the gap between symbols is filled with a cyclic prefix (CP): A copy of the end of each symbol is attached in the beginning of the respective symbol. This enables us to apply a Schmidl-Cox time synchronization: We correlate two blocks that are shifted by one symbol length and have the width of the cyclic prefix. The principle is shown in Figure 6. The cyclic prefix lengths are 72 samples, except for symbols 1 and 9, which have an extended cyclic prefix of 80 samples. The peaks in the correlation in Figure 6 show the symbol start in the time domain. With the information about the exact symbol start, the symbols can be transferred to the frequency domain using the FFT: 1024 samples from the time domain result in 1024 subcarriers in the frequency domain. After the synchronization, the cyclic prefix is no longer required and discarded.

Frequency Offset Correction via Zadoff-Chu Sequences: We found that the symbols 4 and 6 always contain the ZC sequences with roots 600 and 147. We correlate the locally-generated ZC sequences with the actual symbol, yielding any carrier-frequency offset, and apply a frequency shift for correction.

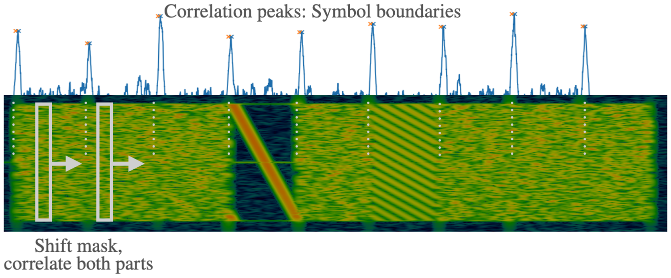


Fig. 6. Symbols contain cyclic prefixes (matching symbol start and end) that we find by shifting a mask over the frame and calculating the correlation. Note that the actual calculation takes place in the time domain.

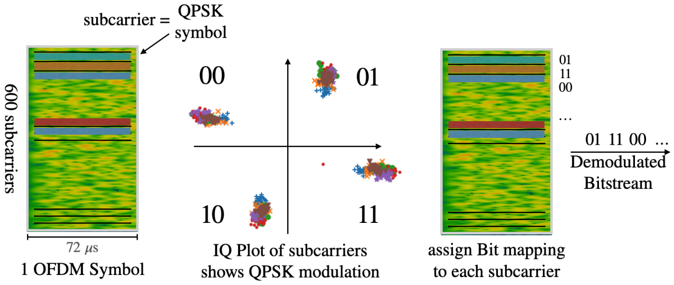


Fig. 7. Subcarrier demodulation and bit assignment: Each subcarrier (i. e., a ”row” within the OFDM symbol) is QPSK-modulated and belongs to one of four groups, visible in the IQ plot. Thereby we assign bits to the subcarriers, yielding the bitstream payload for the OFDM symbol.

Subcarrier Demodulation and Bit Assignment: The OFDM subcarriers are Quadrature Phase Shift Keying (QPSK)-modulated; i. e., the carrier signals are phase-shifted by one of four possible angles to modulate the signal's message in two bits. Figure 7 shows the four different phase shifts, resulting in four clusters—if plotted as complex number representation—and their respective bit representation. The better the synchronization and error correction, the sharper the grouping. The following decoding steps will show if the bit assignment was correct.

d) ***Decoding:*** The previous steps convert the radio signal into a bitstream, which can be decoded to retrieve the actual DroneID payload. By analyzing the S1 firmware, we found the data is (i) scrambled with a Gold sequence, and we identified the seed for the underlying Linear-Feedback Shift Register (LFSR). Also, it is (ii) encoded with a turbo-encoder using the same permutation table for sub-block interleaving as in the LTE specification [14], [22].

We apply the de-scrambling and turbo-decoding of the bitstream and map the resulting data to the DroneID structure (cf. Listing 13 in Appendix A), which we found in the firmware of the drone via reverse engineering. The CRC checksum included in each packet matches our calculation, showing that we recovered the data correctly. Figure 8 shows an example of a successfully recovered DroneID payload.

e) ***Performance:*** We examined the DroneID broadcasts from various DJI drones such as the Mini 2, Mavic Air 2, and, Mavic 2 Pro / Zoom; these models all use the OcuSync 2.0 protocol. We tested the receiver outdoors with drones in-flight. We kept both drone and operator moving to generate non-static DroneID packets. We were able to successfully decode broadcasts from all these models. The decoded broadcasts show that all location information is accurate and updated within each packet, allowing a full flight path reconstruction. Only a single DroneID packet is required to locate both the drone and the pilot. The more packets we can decode, the higher our flight path resolution is. We calculate the percentage of detected frames, decoding attempts, and successful decodes—with matching CRC—and compare it to the *expected* number (based on the DroneID interval of 640 ms). Our initial measurements show that for all tested drones, about 37% of the expected packets are correctly decoded – meaning that we receive location updates every two seconds. That shows that even our prototype implementation of a receiver is an efficient method for live recording of a full flight path.

```
{ "pkt_len": 88,                      "v_up": 0,
  "unk": 16,                          "d_1_angle": 1.27,
  "version": 2,                       "gps_time": 1650894651295,
  "sequence_number": 627,             "app_lat": "43.2682071",
  "state_info": 8179,                 "app_lon": "6.6401597",
  "serial_number": "spoofed",         "lon_home": "7.2679321",
  "lon":  "7.2679607",                "lat_home": "51.4468438",
  "lat":  "51.4468610",               "device_type": "Mini 2",
  "altitude": 40.54,                  "uuid_len": 0,
  "height": 3.05,                     "uuid": "",
  "v_north": 29,                      "crc-packet": "22f5",
  "v_east": 84,                       "crc-calculated": "22f5"    }
```

Fig. 8. A decoded DroneID packet with the manipulated serial number `spoofed` and a spoofed operator location (`app_lat, app_lon`).

f) ***Results:*** Our receiver range is limited to a range of roughly 10 m and sensitive to interference that occurs, e. g., due to neighboring WiFi stations. This effect can be attributed to our receiver chain focusing on reverse-engineering the signal, while not being optimized for performance or high range. Our tests show that—within the receiver range—we can successfully and reliably receive, demodulate, and decode DJI's proprietary DroneID transmission. Most importantly, we find that the signals are not encrypted. Our observations show that DroneID accurately discloses the locations of the drone, the home point, and the remote pilot.

The received DroneID packets showed that our assumptions about the structure of the packets on the wireless physical layer were correct and matched the structures we recovered from the drone's firmware. Furthermore, the decoded packets confirmed that we successfully modified the drone's serial number (see Section IV D e), which allows a malicious user to obfuscate their identity. Figure 8 shows an example of the full DroneID information.

Now that we can observe and decode DroneID packets—which was previously not possible without access to DJI's Aeroscope receiver—we want to test if it is possible to disable DroneID or fake the transmitted position. We conduct two experiments: $i)$ Spoofing the pilot's location, and $ii)$ disabling DroneID through undocumented DUML commands.

For $i)$ spoofing the operator's location, we used a non-rooted Android phone and the app called "Fake GPS" from the Google Playstore [33]. After the fake position was set in the spoofing app, we switched to the DJI app, started the drone, and turned on the receiver. The receiver starts to find and decode valid DroneID packets where the drone's position is set correctly, but the transmitted coordinates of the remote pilot are set to the spoofed position. The app allows the simulation of random movements, which we could also verify in the received

DroneID packets. The distance between the drone, its home point, and the pilot's faked GPS position did not play a role. Thus, we conclude that no check is performed regarding the consistency of the coordinates, and the location of the remote pilot can be successfully spoofed.

Further, during our security analysis, we found a publicly unknown DUML command that *seemed* to allow configuring and disabling the different DroneID values. According to DJI, this command is part of an internal API that should not be available externally. This has been fixed in the latest model. Using our live DroneID decoder, we could confirm that this command does not disable the DroneID packets but replaces the respective values within the packet with the value 'fake'.

In summary, we reverse-engineered the proprietary DJI DroneID broadcasts, confirmed that they are unencrypted, and discussed how common off-the-shelf tools suffice to decode the transmitted information, including the location of both the drone and the remote pilot. This contradicts the public perception, which assumes that such privacy-relevant information are transmitted in a protected manner [30], making it a privacy-risk for people using the drone without being aware of this behavior.

When studying whether the location data is protected against spoofing attempts, we found that no such protections are in place.

IV. Security Analysis with Physical Access

So far, we have focused solely on a passive attacker monitoring the wireless physical layer. In this section, we assume an *active* attacker.

A. Threat Model 2: Active Attacker

In the following, we consider scenarios where an attacker has physical access to the drone, its RC, and the smartphone connected to the RC. We assume the goal of this attacker is to deny its service or to bypass the countermeasures enforced by DJI on the drone. For example, they may desire to crash the drone during flight, disable DroneID, geofencing, or other software limits, such that they can fly over restricted areas. A first step towards achieving this goal is the ability to extract the firmware running on the drones and then to elevate privileges.

We stress that despite assuming physical access to the drone, this is mainly for the analysis process itself. As we show, once interesting commands are uncovered, almost all of them can be sent to the drone over the air. In other words, we also analyze a variant of this active attacker model: an attacker that has only compromised the user's smartphone connected to the RC (which is normal during flight) but has *no* physical access to the drone itself.

B. Overview

Because drones are complex cyber-physical systems with multiple interfaces, different firmware files, and diverse architectures, several types of analysis methods need to be applied to them. Nassi et al. [43] identified six unique targets that can be attacked: Drone hardware, drone chassis and package, ground control station, radio communication channel, pilot, and cloud services. Nassi et al. use the term "drone hardware" to refer to the drone itself, its components, and the firmware. In this work, we distinguish between hardware and firmware, and we analyze both in more detail.

Before we can use automated dynamic analysis methods such as fuzzing, which have proved invaluable towards effectively finding bugs in software [24] and firmware [23], [55], we must understand the intricacies of the target under test. For example, we need to identify communication interfaces and protocols. To do so, we manually conduct a static analysis of the firmware and analyze the drone's hardware. Beyond yielding essential information for our automated dynamic analysis, we uncover several critical security vulnerabilities this way. Building on the identified DUML communication protocol, we then devise a fuzzer to automatically search for further vulnerabilities and uncover a significant number of them. All findings have been responsibly disclosed to and acknowledged by the vendor DJI.

For our detailed analysis, we investigate the following DJI hardware devices: DJI Mini 2, the Mavic Air 2, and the Mavic 2 Pro / Zoom. Furthermore, we studied the remote control RC231 / RC-N1 (used by the latest DJI drones, including DJI Mini 2, Mavic Air 2, and Mavic 3) and replicated found bugs on the latest DJI drone, the Mavic 3.

C. Initial Manual Analysis and Interactive Access

Before we can mount any automated analyses techniques, such as fuzzing, we need to understand our target.

1) Manual Static Analysis: We apply manual static analysis to different parts of the drone firmware, such as the flight controller, and uncover crucial information on various features and drone functionality. For our analysis, we use standard tools, in particular Ghidra [44], Binwalk [52], a hex editor, and the dji-firmware-tools [38]. Most importantly, our manual static analysis reveals information on the structure of DUML, laying the foundation of our dynamic analysis described in the subsequent section. Beyond providing essential information for the understanding and analysis of drones, our manual static analysis unveils a vulnerability that allows us to bypass DJI's firmware signing and gain a privileged shell on the device.

a) ***S1 SoC Firmware Signature Verification Bypass:*** The firmware of the S1 SoC can be found on the drone as `sparrow_firmware` and in firmware updates, which can be found online; it consists of different binary files. Part of this firmware package are so-called "SDRH" configuration files. These files are by design neither signed nor encrypted, and our manual static analysis of the bootloader reveals that these files specify memory addresses and their values. We assume that DJI uses them to install patches in the transceiver firmware without having to reflash the entire firmware. However, DJI does not disclose any details nor mentions these files in the security whitepaper [20]. The bootloader reads these files and patches the defined memory addresses of the RTOS's RAM with the specified values. Using hand-crafted SDRH files allows us to patch the firmware during boot, which occurs after the verification of the firmware signature. This approach completely bypasses the firmware signing process and enables an attacker, who is able to provide crafted SDRH files, to make arbitrary changes to firmware code.

TABLE I. DRONES AFFECTED BY SIGNATURE BYPASS AND ARBITRARY CODE EXECUTION FOUND BY MANUAL STATIC ANALYSIS

Drone	**Firmware**[a]	**Sig. Bypass**		**Code Exec**
		Drone	RC	
Mavic 2 Pro	01.00.0770	✗	✗	✗
Mavic 3	01.00.0600	✗	✓	✓
Mavic Air 2	01.01.0920	(✓)[b]	✓	✗
DJI Mini 2	01.05.0000	✓	✓	✓
RC231 / RC-N1	04.11.0034	✓		✓

[a] We verified the bugs on the latest firmware version available during writing

[b] We could not verify this as we have no root access on this model. However, the underlying vulnerable hardware is the same (S1 SoC) as in the DJI Mini 2.

Table I lists drones and RCs affected by this signature bypass (column **Sig. Bypass**). The Mavic 3 and Mavic 2 Pro are not affected as they use a different transceiver than the vulnerable S1. Following our responsible disclosure process, DJI has assigned this vulnerability the severity *critical*.

*b) **SDRH File Delivery via Fastboot**:* In search of a delivery mechanism for SDRH files to the drone, we investigate different binaries of the drone. We discover that the firmware is flashed to the RAM of the SoC via a fastboot-like system. This requires the bootloader to be in the correct state to accept these fastboot commands. By reverse engineering the system initialization binaries from the drone's filesystem, we find that the bootloader of the transceiver can be triggered by sending different correctly-formatted packets. Sending such packets to the RC during boot via USB enables the fastboot mode. By first unlocking the fastboot mode, we can use it to upload (signed) transceiver firmware files alongside (unsigned) SDRH files. This allows an attacker to include a malicious SDRH file alongside original firmware files and divert the control flow of the transceiver firmware. Uploading these SDHR files does not require authorization but only physical access to the drone or RC.

*c) **Backdooring the Sparrow Transceiver Firmware**:* In the following step, we analyze the Sparrow transceiver firmware and discover an UART interface. The transceiver firmware exposes a shell via this interface. However, we find that this shell is closed on production drones. Hardware indicates its production state to the firmware via what likely is a fuse bit. We locate the code in the drone firmware that checks this bit. By uploading a custom SDRH file that patches firmware code to disable this check, we re-enable the UART shell in the firmware logic.

2) Hardware Analysis: To match the findings from our manual static analysis, we inspect the drone hardware for communication interfaces, including the previously mentioned UART interface. Our goal is to verify our software finding on the hardware level and obtain an interactive shell on the transceiver firmware. We use a custom PCB workstation for our hardware analysis (cf. Figure 11) that allows us to probe different connectors. We use a logic analyzer and an oscilloscope to identify the UART interface exposed in the firmware.

Since the sparrow firmware is shared by the latest DJI drones like the DJI Mini 2, the Mavic Air 2, and also their remote controls, we choose these devices as our target. Here, the hardware analysis of the RC proved to be the most accessible and could be tested even while the device was switched on. In our investigation of the RC231 remote controller hardware, we find the UART port used by the transceiver firmware to be active for the first two seconds after turning on the device.

Using our previous findings from the manual static analysis, i. e., the patching of firmware files, we caused the transceiver firmware to keep the UART connection open indefinitely. This allows us to connect to the transceiver UART shell. On this shell we identified commands to perform arbitrary memory reads, writes, and a command to spawn a shell, granting us elevated privileges – the prerequisite towards disabling countermeasures and software-enforced limits.

D. Dynamic Analysis – Fuzzing

Given the understanding and insight from the previous manual static analysis, we can now set up an automated approach to uncover bugs. One of the most effective techniques in effectively uncovering bugs is *fuzzing*, an automated approach to identify software faults by providing the tested target with (potentially invalid) inputs. Its great success has spurred many different research directions to improve all aspects of fuzzing. One of the most important innovations was the introduction of *coverage feedback* [68], where the target is instrumented to report which code was exercised by a particular input. This allows the fuzzer to observe how individual inputs influence the control flow and thus guide the fuzzing process. Modern fuzzers usually rely on instrumentation injected during the compilation phase, thereby requiring access to source code. If source code is not available, fuzzers can instrument the binary executable by using Dynamic Binary Instrumentation (DBI) [27], [28], binary rewriting [17], [41], or hardware features such as Intel PT [62], [56], [2]. However, they either require specific hardware, e. g., Intel processors, or make assumptions about the execution environment, e. g., by relying on defined interfaces provided by the OS. Unfortunately, firmware running on embedded devices usually does not expose such an interface. For fuzzing such firmware blobs (which typically require specific hardware configurations), state-of-the-art fuzzing relies on *re-hosting* [23], [55], i. e., (partial) emulation of the firmware's environment, since fuzzing on the actual hardware is often complex to set up, provides insufficient feedback, and does not scale well. Still, it is mandatory to have access to the firmware.

*a) **Drone Fuzzer Design**:* While fuzz testing of a given drone seems like a natural approach to analyzing its security, we face the aforementioned obstacle: We have neither access to the source code nor to the complete firmware. Consequently, concurrent fuzzing approaches do not apply to DJI drones: We must fuzz using the actual hardware and have no access to coverage information.

As a suitable fuzz target, we identify the DUML protocol, which is used to enable configuration of the entire drone. Moreover, the protocol is used by all drone components for internal communication, and—by the means of the USB interface—readily available to adversaries with physical access. Due to its nature of gluing different components together, its deep embedding in the core of the drone, and its resulting high complexity, this protocol represents an attractive target for potential attacks: A vulnerability in this system provides an attacker with a powerful primitive to exploit the whole drone.

DUML Protocol. Based on our analysis of DUML (Section II A d), we designed a generational black-box fuzzer that uses a custom grammar, i. e., a fuzzer that knows the specification for which it generates inputs but is not guided by coverage. As we do not have access to source code, complete firmware, or an emulator capable of mimicking DJI-specific hardware, we need to perform fuzzing on the drone itself. Since the DJI drones and remote controls provide a USB interface where communication via the DUML protocol is possible, we choose that interface for our fuzzer and observe its behavior. As we have no coverage feedback, our fuzzer works in a *black-box* scenario, where we can only observe the drone's behavior from the outside. More precisely, we can identify whether the drone crashes, which resets the connection. To improve detection of bugs not resulting in a crash, we propose a novel bug oracle based on differences in the smartphone app's UI, which is connected to the RC and serves as a screen for drone configuration, sensor readings, and camera feed. For example, when the fuzzer changes the serial number, our oracle can automatically flag this behavior.

Communicating via DUML requires us to adhere to the DUML protocol. Fuzzing input not heeding the specification is likely rejected during early parsing without testing actual program logic. Thus, we design our fuzzer as a grammar fuzzer, which is aware of the underlying protocol specification. Recall that each DUML command (see Section II A d) consists of the fields, *src*, *dest*, *cmdType*, *cmdSet*, and *cmdID* — all these fields are represented by one byte each. One possibility for a fuzzer is to iterate over all 256 values for each byte, which would take too long and test many unnecessary and non-existent commands. To avoid this, we narrow down the possible values for these fields to the already known sources, destinations, and command sets. This allows us to test the remaining commands exhaustively in a deterministic way: We generate all possible DUML commands from our set of commands and test all 256 values for the field `commandid`. Only the command's payload may consist of arbitrary bytes, limited only by the length of the DUML packet. We generate (and record to allow deterministically replaying tested commands) a random payload for this field. Before sending the packet, we calculate the correct checksum for the packet such that the program logic beyond the checksum parsing is tested. With this approach, we can identify any DUML command that causes the drone to crash, disconnect, or otherwise physically malfunction. Note that not all errors manifest in crashes or physically observable malfunctioning. Instead, some commands may cause internal errors or corrupt other data on the device.

UI Oracle. To overcome this limitation, we introduce a more precise bug oracle: the UI oracle. DJI drones are closely interconnected with a smartphone, which is connected to the remote control. The smartphone is used to show the camera feed of the drone and allows the user to inspect values, such as sensor readings, or change settings (cf. Section II A e). In the spirit of widely deployed test strategies, e. g., as implemented by frameworks such as `selenium`, we can automatically navigate the user interface of the drone during a fuzzing campaign to identify subliminally diverging but non-crashing behavior. For example, we can detect when the fuzzer changes settings or triggers parser errors, which are reported in the UI as warning messages.

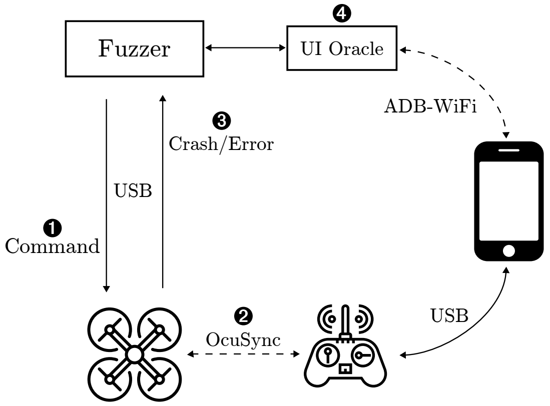


Fig. 9. Overview of the fuzzer and its components. The fuzzer sends multiple commands to the drone ❶, which processes them and synchronizes itself with the RC ❷ for the sake of displaying, e. g., status information. During fuzzing, the drone is monitored for crashes ❸ or changes in the UI ❹. If a crash or UI change was found, we use Algorithm 1 to identify the offending command.

Input Blocks. As communication (request-response) with the drone is slow, we send fuzzing inputs, i. e., commands, without waiting for a response. Compared to a regular fuzzing iteration, a UI oracle cycle (walking the UI and identifying non-crashing errors) is slow. Thus, we opt for reducing the number of queries to the UI oracle to a minimum. Therefore, instead of considering inputs on an individual basis, the fuzzer splits the input space of all possible commands into *input blocks*, where each input block contains $130,000$ commands.

Fuzzing Loop. The fuzzer then sends these inputs one-by-one over the USB serial device to the fuzz target (❶ in Figure 9). The drone processes each command and synchronizes with the RC (❷); for example, it updates status data. If the fuzzer observes the drone's firmware crashing (❸), we can not associate this crash with the last command as we do not wait for the response to each command; instead, we re-test the last $5,000$ commands to identify the offending command.

To optimize this process, we perform a binary-search algorithm as outlined in Algorithm 1. Given a group of commands C, we successively split the commands into two equally sized groups C_A and C_B. We execute all commands from the first group and verify whether the group contains the crashing command. If it does, we repeat this process for the first group. If it does not, we discard this group and continue with the second group. This is continued until only one command—the trigger command—is left and returned. This way, we can identify the command causing the crash in $log(n)$ steps.

If no crash is observed and all commands in an input block have been sent, we still have to check for unexpected behavior. For this, the fuzzer queries the *UI oracle* for deviations (❹). In case the UI oracle finds a deviation, we, again, cannot directly associate this finding with a particular command sent. Instead, we need to re-test the last input block using Algorithm 1; instead of checking for crashes, we query the UI oracle and check whether the UI deviation can be observed.

In summary, depending on the identified fault, we either use the traditional crash oracle to verify if a crash has occurred or the UI oracle to verify whether a deviation within the UI can be observed.

```
Algorithm 1: FindOffendingCmd
Input: Set of commands $C$
Result: Error causing command $c$
# Exit when $C$ contains only one command
if $|C| == 1$ then
    Execute($c := C[0]$)
    if Oracle() == “error observed” then
        return $c$
    else
        return $\perp$

# Split commands into equal-sized subsets
$C_A \cup C_B := C$ with $|C_A| - |C_B| \leq 1$
# Execute all commands in $C_A$
forall $c \in C_A$ do
    Execute($c$)

# Continue with subset $C_A$ or $C_B$
if Oracle() == “error observed” then
    return FindOffendingCmd($C_A$)
else
    return FindOffendingCmd($C_B$)
```

*b) **Implementation:*** The fuzzer and UI oracle are implemented in $\sim 4,000$ lines of Python code. Our fuzzer features two modus operandi: It can either fuzz the drone via its serial connection or over-the-air while connected to the serial interface of the RC. The latter allows us to identify crashes that can be triggered remotely. When fuzzing over-the-air, the fuzzer checks periodically if the drone is still alive, using a specific command with a known return value. Additionally, our fuzzer checks if there are any Android Debug Bridge (ADB) devices except the Android phone that is used for the UI oracle. Any other device is a strong indicator that the ADB service on the drone has been started by one of the commands the fuzzer sent; an available ADB service grants *root access* to the drone.

Our UI oracle uses Android-internal tools such as ADB and `uiautomator` to interact with the DJI app in an automatic fashion to determine whether the app's state and the data reported in the app's interface match the expectations; deviations indicate that the fuzzer managed to identify an interesting command. We designed the UI oracle such that all values to be checked are defined in advance. This prevents regularly changing values, such as the battery level, leading to false positives.

*c) **Experiment Setup:*** We tested our fuzzer with three different DJI drones and one remote control, which is used by the latest DJI drones:

- Mavic Air 2, firmware: 01.01.0610
- DJI Mini 2, firmware: 01.03.0000
- Mavic 2 Pro, firmware: 01.00.0770
- RC231 (RC) + Mavic Air 2, firmware: 01.01.0610

The host computer running the fuzzer must be in the same WiFi network as the Android phone, and the drone must be connected to that computer. We used the smartphone's WiFi hotspot for our test. On the first start, the fuzzer needs to initialize the smartphone, start the ADB server, and configure the ADB WiFi connection. To use the phone with the app as a UI oracle, the phone needs to be connected to the RC, and the connection between RC and drone (OcuSync) needs to be established. We use a rooted OnePlus 8 Android phone with Android 11 running the DJI Fly app (v1.6.6), which is used by both Mavic Air 2 and DJI Mini 2.

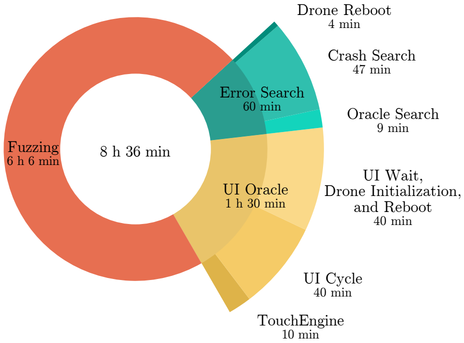


Fig. 10. Distribution of time spent by the fuzzer during fuzzing of the Mavic Air 2 drone.

We run four independent fuzzing campaigns:

1) Mavic Air 2 + connected RC231 with UI orcale
2) DJI Mini 2 + connected RC231 with UI oracle
3) Mavic 2 Pro *without* UI oracle
4) RC231 + Mavic Air 2 in a reverse manner, where the RC is fuzzed and the drone is connected via OcuSync

We test the Mavic Air 2 and DJI Mini 2 drones using our UI oracle. As the Mavic 2 Pro uses a different legacy app, we currently can not use our UI oracle there, as it is specifically tailored to the modern DJI app. To reduce engineering burden, we refrain from adapting the oracle to the old application, which is deprecated. The fourth fuzzing campaign investigates whether fuzzing the drone over-the-air via the RC is feasible.

It is also possible that the controller loses the drone's signal because an invalid state is reached or the transceiver of the drone crashes. Such a state, which continues to accept commands on the serial interface but without a connection to the controller, can only be detected when the UI oracle checks the app. If the UI oracle detects that no drone is connected, our fuzzer attempts to reboot the fuzzing target. Finally, the UI oracle itself also has limitations and requirements: We need to define all interesting values that should be checked during the fuzzing campaign. We need to further account for common, non-critical error messages, such as those that occur when the GPS signal is weak.

*d) **Results:*** We implemented the proposed approach, which allows us to systematically test different DJI devices, i. e., both drones and RCs, via the USB interface for potential vulnerabilities. Noteworthy, our approach works on all current DJI drones and RCs. We test roughly 7.8 million commands, which are generated by enumerating all commands as described above. On average, we can test 400 commands per second if the drone itself is fuzzed, resulting in a raw fuzzing time of roughly 5.5 hours if we do not account for crashes or UI cycles. However, each time the drone crashes, this time is prolonged because the drone must be restarted first. Also, the UI oracle interruptions (and potentially searching for the command that caused an observed bug) take additional time. In practice, we found that an entire fuzzing run takes around 9 hours to complete. In Figure 10, we outline the time spent by the fuzzer on the individual steps during fuzzing the Mavic

TABLE II. DEDUPLICATED FINDINGS IDENTIFIED DURING FUZZING

ID	Oracle	Component	Observable Behavior	Classification[a]	Severity[a]	Remote[b]	Vulnerable Devices
#1	ADB check	`dji_sys` binary	ADB started (root access)	arbitrary code exec	mid	✗	Mini 2
#2	crash	flight controller	critical error (drone reboot)	buffer overflow	mid	✓	Mavic Air 2
#3	crash	flight controller	critical error (drone reboot)	buffer overflow	mid	✓	Mavic Air 2
#4	crash	flight controller	critical error (drone reboot)	buffer overflow	mid	✓	Mavic Air 2
#5	crash	flight controller	critical error (drone reboot)	buffer overflow	mid	✓	Mavic Air 2
#6	crash	flight controller	critical error (drone reboot)	buffer overflow	mid	✓	Mavic Air 2
#7	crash	flight controller	critical error (drone reboot)	denial of service	mid	✓	Mini 2
#8	crash	flight controller	critical error (drone reboot)	denial of service	mid	✓	Mini 2
#9	crash	unknown[c]	critical error (drone reboot)	denial of service	mid	✓	Mini 2
#10	crash	unknown[c]	critical error (drone reboot)	denial of service	mid	✓	Mini 2
#11	crash	unknown[c]	critical error (drone reboot)	denial of service	low	✓	Mini 2
#12	crash	unknown[c]	critical error (drone reboot)	denial of service	low	✓	Mini 2
#13	crash	flight controller	critical error (drone reboot)	denial of service	low	✓	Mavic Air 2
#14	UI change	WiFi chip	change SSID	arbitrary code exec	mid	✓	Mini 2, Mavic 3
#15	UI change	flight controller	change serial number	identity spoofing	mid	✓	Mini 2
#16	UI change	flight controller	change drone name[d]	—	—	✓	Mavic Air 2, Mini 2

[a] as assigned by DJI during the responsible disclosure process
[b] attack can be carried out over-the-air with the RC functioning as relay
[c] component is undocumented or finding is not directly associated with a specific component
[d] benign finding that does not appear to be exploitable

Air 2 drone. The UI oracle takes about 25% of the total runtime. Interestingly, only about half of this time is spent in the actual UI cycle (i. e., inspecting values or navigating to UI fields), resetting the app, and checking if the phone is properly connected (TouchEngine). The other half is spent on rebooting the drone and waiting for initialization to happen. In general, searching for a command causing a UI deviation is more costly than searching a command leading to a crash, as the drone and app have to be restarted multiple times and the UI must be searched for the deviation.

For our fuzzing run 4), where we fuzzed the RC231 instead of the drone, we received a much lower execution rate with around 21 commands per second. With such an execution speed, the fuzzing run would take around 104 hours to complete. Therefore, we reduced the commands for the fuzzer to the sources, destinations, and command sets that proved bug-ridden during the other fuzzing campaigns, limiting the number of tested commands to $1,073,111$. While this fuzzing run is incomplete, it serves as a proof of concept that the execution of the commands is also possible over-the-air as the RC relays commands. When issuing a command that causes a segmentation fault within the flight controller, an attacker can crash the drone, which means that it falls to the ground during flight.

Overall, we found 15 software faults (after manual deduplication) that lead to a (software) crash or other kinds of unexpected behavior. A detailed summary is listed in Table II, including the classification and severity assigned by DJI during the responsible disclosure process. Interestingly, most of the crashes occur within the flight controller, which is critical with respect to the drone's flight operation. Beyond the ability to crash the drone, our fuzzer uncovers a way to modify the presumably immutable serial number (bug #15), an arbitrary code execution (bug #14), and a backdoor launching an ADB service with root privileges (bug #1). The first two bugs (#14 and #15) can only be uncovered using our UI oracle as they do not cause an imminent crash. This is also the case for #16, where the fuzzer changes the drone name. However, this information is not considered immutable by DJI, and no other harmful behavior is apparent from such a change, making it a benign finding. Bug #1 is special in the fact that it requires the fuzzer to check for active ADB services after processing an input block. As this requires physical access, it is also the only finding that can not be exploited remotely.

For all other findings, we can issue the associated commands on the RC, which are then transmitted to the drone via the wireless protocol OcuSync. This is possible since DUML acts as a bus protocol interconnecting all components of a DJI drone. Consequently, this allows us to send commands to the drone without being physically connected. This has several implications and opens up new attack vectors: Bugs found during fuzzing via USB, e. g., a command that crashes the firmware of the drone, can be issued remotely by sending it to the RC. Previously harmless bugs simply crashing the drones' firmware when directly connected to the drone on the ground now have significantly worse consequences if triggered mid-flight remotely, leading to at least a *Denial of Service (DoS)* and possibly damaging or destroying the drone. As DJI classified some of these bugs as *buffer overflow*, they may potentially be exploitable. The only pre-requisite for this is that the attacker has access to the RC via a device under their control. Notably, while our fuzzing happened over the RC's communication port, DUML messages can also be sent via the USB port that the pilot's smartphone is connected to during regular flight operation. Here, the pilot's Android smartphone is connected to the USB-C port at the top of the RC231 and, using the Android Open Accessory (AOA) protocol, acts as a USB accessory while the RC is the USB host. This is normal behavior that allows the DJI app to display flight status information; at the same time, however, this enables an attacker who has compromised the user's smartphone to control or crash the drone. Since DJI distributes their app only via the official DJI website but not the Google Play Store, users need to override Google's security settings and allow app installations from other sources than the Google Play Store, thereby increasing the attack surface of their devices.

*e) **Case Studies:*** We now highlight three exemplary findings to show how a crash or a change in the drone's internal values triggered by the fuzzer can result in security-relevant problems.

(1) Arbitrary Command Execution (#14). We first discuss a bug that, at first glance, seems harmless, but upon further analysis, turns out to be a fully-fledged *OS command injection* vulnerability (CWE-78 [39]) that grants an attacker elevated privileges on the drone. This was found by using the UI oracle that reported a deviation in the app's user interface: The SSID name of the WiFi, which is used to transfer videos and photos between drone and smartphone, was replaced with random bytes. This implies that the fuzzer found a command that allowed it to replace the SSID name with a user-controlled input. Using our aforementioned approach, the fuzzer identified the specific DUML command responsible for this action in an automated way. Subsequent manual root-cause analysis revealed that the drone processes this string in multiple functions. Since the location where this string is placed within a shell command is not properly sanitized, an attacker can inject arbitrary shell commands. One caveat is that the vulnerable function contains a length check, such that the injected command's length is limited. To overcome this limitation, an attacker can simply create an exploit script containing their desired commands and transfer this script chunkwise to the drone. Once transferred, the attacker marks the script as executable, granting them an arbitrary command execution vulnerability without length restrictions. DJI has acknowledged this bug as an *arbitrary code execution vulnerability*. Table I shows the affected devices (column **Code Exec**).

(2) Arbitrary Serial Number (#15). DJI devices have various serial numbers for the different hardware modules such as the camera, battery, or flight controller. The serial number of the flight controller is used to authenticate and identify the aircraft. The security whitepaper of DJI states that this serial number must be unique, *immutable*, and should be stored in a secure storage [20]. While fuzzing the DJI Mini 2 drone, the UI oracle discovered that the fuzzer managed to change the serial number of the flight controller (cf. Figure 12 in Appendix A). An attacker can abuse this to *spoof their identity*, as this serial number is also the one that is transmitted in DroneID.

(3) Unlocking ADB Root shell (#1). While fuzzing the DJI Mini 2, the fuzzer's periodical ADB check was triggered, and the error search by our algorithm reported a command that starts an ADB root shell on the drone. Further investigation reveals that actually two commands are needed: the so-called 'DJI Assistant Unlock command' followed by the command found by the fuzzer. The fuzzer always sends the former command to unlock full DUML communication.

Manually analyzing the bug, we find it is a logical flaw in the *challenge-and-response*-like system used to activate a DJI debugging feature in the `dji_sys` binary. The flight controller, which is responsible for validating the challenge, seems to accept any value, thus always unlocking this debugging mode.

*f) **Roadblocks and Limitations of Fuzzing:*** Fuzzing embedded devices involves various roadblocks and limitations. One limitation is running out of battery if the device cannot be supplied with power during the fuzzing campaign. This leads to numerous interruptions for replacing and charging the batteries and limits the automation of the fuzzing process. Since the devices may reach invalid states triggered by the fuzzing inputs, the drone or the remote controller firmware can brick and no longer respond to the fuzzer. In such a state, the fuzzer assumes a crash and waits until the device returns online, which may never happen. If such a software brick occurs and the device cannot be reset by, e. g., removing the battery, we must flash the firmware to "un-brick" the drone. This happened once during our initial implementation of the fuzzer.

V. Discussion and Lessons Learned

In the previous sections, we have examined the security and privacy risks associated with consumer drones. We focused on drones from DJI, as this company is the leading drone manufacturer. In addition, DJI's whitepaper on security establishes a solid foundation. In the following, we discuss our main findings and elaborate on their implications.

*a) **Current State of Drone Security:*** We found that the deployed hardening mechanisms make analysis of the drone more difficult, but not impossible. With static analysis, we were able to uncover bugs that led to the execution of arbitrary code on the S1 SoC via specifically crafted configuration files. The bugs could be observed for both the drone and RC. Using our DUML fuzzer, we found additional vulnerabilities compromising the device security for various drones. More concerning, we found safety-critical faults and managed to crash the drone mid-flight when fuzzing the RC. Furthermore, our security assessment reveals that we can flash arbitrary firmware to the transceiver, which implies that the secure updates using signed and encrypted firmware are not implemented correctly. We recommend that vendors take our findings into account and perform additional testing, e. g., in the form of fuzzing, to reduce the number of (exploitable) bugs in their code. We plan to collaborate with DJI to improve security and safety properties of the studied drones.

*b) **EU and US Regulations for Drone Identification:*** Regulatory bodies consider identification and localization as a safety feature for consumer drones that outweighs the potential privacy risks for remote pilots. Two compatible standards were in draft status during our analysis: EN 4709 (EU) and F3411-19 (US) [5], [3]. Both standards will require drones to broadcast drone *and* pilot location, the drone's trajectory, and its identification number. The standards foresee using either Bluetooth advertisements or WiFi Neighborhood-Aware-Network features and, by requirement, they will operate without any kind of encryption. These standards are to be introduced on a mandatory basis from mid-2023 in Europe and on September 16, 2023, in the US. Contrary to DJI's proprietary solution, every compatible smartphone will be able to receive these broadcasts via WiFi or Bluetooth, albeit at a significantly lower distance. Open implementations are already available as library or Android application [48], [47].

Although we are aware of these upcoming standards, we found it surprisingly difficult to identify what *current* drones—where open standards do not yet apply—actually transmit as part of their proprietary protocol. Our detailed analysis confirms that DJI's DroneID protocol transmits unencrypted information, such as the drone's and the remote pilot's position, while the official statements of DJI claimed that this information is encrypted [30], [20]. DJI has corrected this statement since then [31]. Regardless, we strongly believe that such features should be transparently communicated to users and be part of a security whitepaper, especially due to inherent privacy risks related to a broadcast of the pilot's location.

c) ***Active Attacker without Physical Access:*** Our analysis of DroneID represents a first step towards understanding the communication protocols used by DJI drones. Given the findings from our analysis, we believe that an active attacker could inject their own DroneID packets to spoof the location of a drone. However, this is beyond the scope of our work; instead, we leave it as an interesting approach for future research.

d) ***Data Integrity:*** With AeroScope, DJI sells tracking equipment for drones and remote pilots; consequently, we expected that the integrity of the GPS location data would be enforced by some countermeasure. However, our analysis reveals that GPS data can be trivially disabled or spoofed, rendering the position data reported by these systems to authorities questionable and non-actionable. A proper integrity protection should be enforced on all devices.

e) ***Applicability to Other Vendors:*** Although we have focused on drones manufactured by DJI in this paper, there are other vendors on the market (e. g., Autel with a market share of 7% [58]). Given their comparably low adoption and the fact that, to the best of our knowledge, no public documentation regarding security and safety measures exists, they are less interesting targets for analysis. Regardless, our principled approach can be applied to other drones: Their hardware, firmware, and software can be analyzed both statically and dynamically, as well as their wireless physical layer. Depending on the protocols used, it may be worthwhile to adapt our fuzzing approach, potentially with a UI oracle similar to our method. As DroneID is unique to DJI, we do not expect interesting results in this direction with respect to other drone vendors, but other proprietary protocols could be worthwhile targets.

f) ***Ethical Considerations and Research Artifacts:*** Drones, by their very nature, raise a number of ethical and moral considerations, especially when considering their misuse in conflicts or their potential use as surveillance devices. We have attempted to avoid discussing these facets in our work, instead focusing on a technical analysis of security and privacy aspects. We argue that regardless of who uses a drone, it should adhere to the security standards promised by the vendor and ensure that the integrity of the countermeasures is not compromised. We responsibly shared the identified vulnerabilities in a coordinated way. Furthermore, we plan to publish all research artifacts related to this work once the vulnerabilities have been fixed.

g) ***Lessons Learned:*** To further support future research, we outline the procedural and methodological lessons learned with regard to analyzing drones. In particular, we would like to emphasize the importance of a holistic, interdisciplinary analysis of firmware, signal processing, and RF signal reverse engineering. This is a necessity to gain deeper insights and understanding of the drone as a system. Focusing only on a single aspect results in limited context and hinders applicability of potential findings. For example, to receive the final decoded DroneID information, the underlying byte structure has to be known. This structure can only be uncovered by reverse engineering the firmware. Combining the analysis of both allows to recover the protocol structure and thereby decode the packets.

Similarly, we strongly recommend combining different techniques: While fuzzing proved fruitful to uncover various software faults, it required a strong understanding of the DUML protocol to have a reasonable target for the fuzzer. Random inputs sent to a drone are highly unlikely to resemble an actual packet and thus discarded immediately by the drone. In the same vein, crashing the drone or observing differing UI behavior requires manual static analysis to uncover the impact and implications of the finding. In our case, the fuzzer managed to change the WiFi SSID – manually investigating this odd behavior allowed us to discover that this enables arbitrary code execution on the drone. On the contrary, without running a fuzzer we would not have detected most of the vulnerabilities given the huge amount of possible DUML commands and their complex interactions with different system components.

Finally, analyzing complex, unknown, closed-source targets such as drones is currently impossible without *manual* analysis. No tool or technique is adequate to automatically reverse engineer and "understand" the complexity posed by drones that expose different hardware components, firmware, and communication protocols. Still, as our fuzzer shows, once a basic understanding is obtained, known techniques can be adapted and ported forward to automate parts of the analysis.

In summary, our lessons learned are that analyzing complex systems such as drones requires (1) holistic, interdisciplinary research that (2) uses multiple analysis techniques but also employs (3) manual work where appropriate.

VI. Related Work

Our work focuses on several different system levels, and we briefly review how it relates to previous work below.

a) ***Drone Research:*** Previous research addressing drone security mostly dealt with aspects such as Denial of Service (DoS), GPS spoofing [1], de-authentication attacks [10], [49], [46], or identifying privacy invasion attacks by drones [42], [7]. For an excellent overview of different targets of drones, attack methods, and countermeasures used, we refer the interested reader to the work by Nassi et al. [43]. However, there is little work so far that addresses system security aspects. For example, fuzzing has only been used to find vulnerabilities in (open source) protocols [21] or to test open ports in WiFi-controlled drones [54]. In contrast, our work analyzes the hardened software that powers DJI drones, and we presented a custom fuzzing approach that takes drone-specific requirements into account. Other research proposes spectrum analysis using an SDR to evaluate the security of the link between drones and RC [57]. We also use an SDR to receive the proprietary DroneID packets so that we can reverse-engineer them.

Note that most current research targets low-cost drones, which typically offer fewer interfaces to control and often employ open-source software and protocols, making them less interesting targets. Nevertheless, there are some prior works that analyze DJI products: For example, one security analysis examines the DJI Go 4 app [63], while others investigate the Phantom 3 [65], [6] or Phantom 4 [16]. These analyses are rather basic, focus only on a particular drone, and make no attempt to assess the security of the full system.

*b) **Fuzzing:*** In the field of fuzzing, there is already a large body of work, e. g., grey-box fuzzers like AFL [68] and its derivatives [8], [24], [25], [9], which, despite their supposed simplicity, have found a large number of different bugs in all kinds of software. From this line of research, a multitude of different techniques emerged. Until recently, most techniques have focused on improving the bug-finding capabilities by using more involved techniques. Among these techniques, the most frequently used ones are symbolic execution [66], [61], [11] and taint tracking [13], which use SMT solvers and data flow analyses to improve the fuzzers' performance.

However, most of these techniques assume the existence of an operating system that provides the fuzzers with a unified interface to the target under test. To overcome this limitation, a considerable effort was made to apply "classic" fuzzing techniques to targets that do not provide a unified interface, such as embedded systems with virtually unlimited configurations. While some approaches propose to fuzz on the devices themselves [32], [67], it quickly became apparent that emulation of (parts of) the embedded system during fuzzing is required to make testing of such systems scalable. This so-called *re-hosting*, i. e., executing (parts of) the firmware inside an emulation environment [55], [23], [26], is currently considered state of the art in this area. Due to the complex interaction of a drone and the RC, we cannot use such techniques but developed a custom fuzzing engine that takes reverse engineering results related to the DUML protocol into account.

Two recent works, IoTFuzzer [12] and Diane [51], also use smartphone apps to benefit fuzzing. Similar to our work, the insight is that the smartphone app associated with a device allows to gain insights on the device itself. In the case of IoTFuzzer, Chen et al. extract information on the protocol used to communicate with the device, allowing them to generate effective fuzzing inputs without having to manually specify a protocol. Redini et al. observe that these apps often sanitize inputs, limiting IoTFuzzer's approach of generating inputs to producing *valid* inputs. They overcome this by extracting code locations to produce valid but *under-constrained* inputs. In contrast, our approach does not utilize the app to extract information useful for input generation, but instead uses the smartphone app as a *bug oracle*, allowing us to detect subtle, non-crashing problems.

VII. Conclusion

In this paper, we analyzed the security and privacy of modern drones from the market leader DJI. We gave an overview of the attack surface of drones and considered two attacker models: A passive attacker without physical access, who is capable of eavesdropping on the drones' over-the-air traffic, and an active attacker who has physical access to the drones' hardware. Reverse engineering the proprietary DroneID packets from DJI's tracking protocol, we found that this protocol broadcasts the drone's position, its home point, and the location of the operator. On the other hand, we uncovered how to disable DroneID or spoof the transmitted location, rendering its reliability for law enforcement questionable. Assuming an active attacker, we devised a fuzzer with a new UI oracle, both tailored specifically to DJI drones, that uncovered multiple critical security vulnerabilities in three different DJI devices. A closer examination of the bugs revealed that our findings can be used to execute arbitrary code or change the serial number of the device, which was presumed immutable. Furthermore, we found out that an attacker can crash the drone mid-flight by sending the payload remotely over-the-air.

Acknowledgements

We would like to thank our shepherd Daniele Antonioli and our anonymous reviewers for their valuable comments and suggestions. We also thank Daniel Klischies for his feedback. Furthermore, we would like to thank DJI for the cooperative responsible disclosure process and swiftly fixing reported vulnerabilities. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972 and by the German Federal Ministry of Education and Research (BMBF, projects CPSec – 16KIS1564K and KMU-Fuzz – 16KIS1523).

References

[1] J. Aru Saputro, E. Egistian Hartadi, and M. Syahral, "Implementation of GPS Attacks on DJI Phantom 3 Standard Drone as a Security Vulnerability Test," in *International Conference on Information Technology, Advanced Mechanical and Electrical Engineering (ICITAMEE)*, 2020.

[2] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "RedQueen: Fuzzing with Input-to-State Correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[3] ASD-STAN, "ASD-STAN prEN 4709-002 P1," https://asd-stan.org/downloads/asd-stan-pren-4709-002-p1/.

[4] B. Association, "Buildroot Making Embedded Linux Easy," https://buildroot.org/.

[5] ASTM, "Standard Specification for Remote ID and Tracking," https://www.astm.org/f3411-19.html.

[6] T. E. A. Barton and M. A. H. B. Azhar, "Forensic analysis of popular UAV systems," in *International Conference on Emerging Security Technologies (EST)*, 2017.

[7] R. Ben Netanel, B. Nassi, A. Shamir, and Y. Elovici, "Detecting Spying Drones," *IEEE Security & Privacy*, vol. 19, no. 1, pp. 65–73, 2021.

[8] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, 2017.

[10] C. A. T. Bonilla, O. J. S. Parra, and J. H. D. Forero, "Common Security Attacks on Drones," *International Journal of Applied Engineering Research*, vol. 13, no. 7, 2018.

[11] C. Cadar, D. Dunbar, and D. E. Klee, "Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[12] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[13] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[14] J.-F. Cheng, A. Nimbalker, Y. W. Blankenship, B. K. Classon, and K. T. Blankenship, “Analysis of Circular Buffer Rate Matching for LTE Turbo Code,” *IEEE Vehicular Technology Conference*, 2008.

[15] F. C. Commission, “FCC ID SS3-MT2WD2007,” https://fccid.io/SS3-MT2WD2007.

[16] V. Dey, V. Pudi, A. Chattopadhyay, and Y. Elovici, “Security Vulnerabilities of Unmanned Aerial Vehicles and Countermeasures: An Experimental Study,” in *International Conference on VLSI Design and International Conference on Embedded Systems*, 2018.

[17] S. Dinesh, N. Burow, D. Xu, and M. Payer, “RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization,” in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[18] DJI, “DJI Mavic 3 - Specs,” https://www.dji.com/mavic-3/specs.

[19] DJI, “DJI Statement On Recent Reports From Security Researchers,” https://www.dji.com/newsroom/news/dji-statement-on-recent-reports-from-security-researchers, 2020.

[20] DJI, “System Security, A DJI Technology White Paper V2.0,” https://security.dji.com/data/resources/, 2022.

[21] K. Domin, I. Symeonidis, and E. Marin, “Security Analysis of the Drone Communication Protocol: Fuzzing the MAVLink Protocol,” in *Symposium on Information Theory*, 2016.

[22] etsi.org, “ETSI TS 136 212 V10.0.0,” https://www.etsi.org/deliver/etsi_ts/136200_136299/136212/10.00.00_60/ts_136212v100000p.pdf, 2011.

[23] B. Feng, A. Mera, and L. Lu, “P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling,” in *USENIX Security Symposium*, 2020.

[24] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining Incremental Steps of Fuzzing Research,” in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[25] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, “Collafl: Path Sensitive Fuzzing,” in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[26] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *et al.*, “Toward the Analysis of Embedded Firmware through Automated Re-hosting,” in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019.

[27] M. Heuse, “AFL-DynamoRIO,” https://github.com/vanhauser-thc/afl-dynamorio, 2018.

[28] ——, “AFL-PIN,” https://github.com/vanhauser-thc/afl-pin, 2018.

[29] S. Hollister, “A Tiny DJI Drone Smuggled its Own Weight in Drugs over the US border Wall,” https://www.theverge.com/2022/2/3/22916246/dji-mini-2-drone-smuggle-meth-us-mexico-border-wall, 2022.

[30] ——, “DJI drones, Ukraine, and Russia — what we know about AeroScope,” https://www.theverge.com/22985101/dji-aeroscope-ukraine-russia-drone-tracking, 2022.

[31] ——, “DJI insisted drone-tracking AeroScope signals were encrypted — now it admits they aren’t,” https://www.theverge.com/2022/4/28/23046916/dji-aeroscope-signals-not-encrypted-drone-tracking, 2022.

[32] K. Koscher, T. Kohno, and D. Molnar, “SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems,” in *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[33] Lexa, “Google Play – Fake GPS Location,” https://play.google.com/store/apps/details?id=com.lexa.fakegps, 2022.

[34] R. Mac, “Amazon Proposes Drone Highway As It Readies For Flying Package Delivery,” https://www.forbes.com/sites/ryanmac/2015/07/28/amazon-proposes-drone-highway-as-it-readies-for-flying-package-delivery/, 2015.

[35] M. McFarland, “In Nepal, a Model for Using Drones for Humanitarianism Emerges,” https://www.washingtonpost.com/news/innovations/wp/2015/10/07/in-nepal-a-model-for-using-drones-for-humanitarianism-emerges/, 2015.

[36] Mefistotelis, “DJI-Firmware-Tools,” https://github.com/o-gs/dji-firmware-tools/wiki/Abbreviations, 2020.

[37] O. G. Mefistotelis, “comm_mkdupc.py,” https://github.com/o-gs/dji-firmware-tools/blob/master/comm_mkdupc.py, 2019.

[38] ——, “DJI-Firmware-Tools,” https://github.com/o-gs/dji-firmware-tools, 2021.

[39] Mitre, “CWE-78: Improper Neutralization of Special Elements used in an OS Command (OS Command Injection),” https://cwe.mitre.org/data/definitions/78.html.

[40] B. Mueller and A. Tsang, “Gatwick Airport Shut Down by ‘Deliberate’ Drone Incursions,” https://www.nytimes.com/2018/12/20/world/europe/gatwick-airport-drones.html, 2018.

[41] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, “Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing,” in *USENIX Security Symposium*, 2021.

[42] B. Nassi, R. Ben-Netanel, A. Shamir, and Y. Elovici, “Drones’ Cryptanalysis - Smashing Cryptography with a Flicker,” in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[43] B. Nassi, R. Bitton, R. Masuoka, A. Shabtai, and Y. Elovici, “SoK: Security and Privacy in the Age of Commercial Drones,” in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[44] National Security Agency, “Ghidra,” https://github.com/NationalSecurityAgency/ghidra/releases.

[45] B. News, “Drugs, weapons ’smuggled to prisoners by drone’,” https://www.bbc.com/news/world-us-canada-60262715, 2022.

[46] J. Noh, Y. Kwon, Y. Son, H. Shin, D. Kim, J. Choi, and Y. Kim, “Tractor Beam: Safe-Hijacking of Consumer Drones with Adaptive GPS Spoofing,” *ACM Trans. Priv. Secur.*, vol. 22, no. 2, 2019.

[47] opendroneid, “Open Drone ID Core C Library,” https://github.com/opendroneid/opendroneid-core-c.

[48] ——, “OpenDroneID Android receiver application,” https://github.com/opendroneid/receiver-android.

[49] N. Pojsomphong, V. Visoottiviseth, W. Sawangphol, A. Khurat, S. Kashihara, and D. Fall, “Investigation of Drone Vulnerability and its Countermeasure,” in *Symposium on Computer Applications Industrial Electronics (ISCAIE)*, 2020.

[50] M. Półka, S. Ptak, and Łukasz Kuziora, “The Use of UAV’s for Search and Rescue Operations,” *Procedia Engineering*, vol. 192, pp. 748–752, 2017.

[51] N. Redini, A. Continella, D. Das, G. D. Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, “Diane: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices,” in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[52] ReFirmLabs, “Binwalk,” https://github.com/ReFirmLabs/binwalk.

[53] M. S. Rosenwald, “Prisons Try to Stop Drones from Delivering Drugs, Porn and Cellphones to Inmates,” https://www.washingtonpost.com/local/prisons-try-to-stop-drones-from-delivering-drugs-porn-and-cellphones-to-inmates/2016/10/12/645fb102-800c-11e6-8d0c-fb6c00c90481_story.html, 2016.

[54] D. Rudo, D. Zeng *et al.*, “Consumer UAV Cybersecurity Vulnerability Assessment Using Fuzzing Tests,” *arXiv preprint arXiv:2008.03621*, 2020.

[55] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, “Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing,” in *USENIX Security Symposium*, 2022.

[56] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels,” in *USENIX Security Symposium*, 2017.

[57] H. Shin, K. Choi, Y. Park, J. Choi, and Y. Kim, “Security Analysis of FHSS-type Drone Controller,” in *International Workshop on Information Security Applications*, 2015.

[58] I. Singh, “DroneAnalyst report reveals dramatic drop in DJI’s commercial drone market share ,” https://dronedj.com/2021/09/14/droneanalyst-dji-market-share-2021/, 2021.

[59] A. Slodkowski, E. Lies, and K. Takenakac, “Olympics-Superstar Osaka lights Flame as Japan’s COVID-hit Games Open,” https://www.reuters.com/lifestyle/sports/slimmed-down-ceremony-open-pandemic-hit-tokyo-games-2021-07-23/, 2021.

[60] D. Slotta, “China’s Thriving Drone Industry,” https://www.asiaperspective.com/china-thriving-drone-industry/, 2022.

[61] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting

fuzzing through selective symbolic execution," in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[62] R. Swiecki, "Security-oriented Fuzzer with Powerful Analysis Options," https://github.com/google/honggfuzz.

[63] T. T. synacktiv.com, "DJI Android GO 4 application security analysis," https://www.synacktiv.com/en/publications/dji-android-go-4-application-security-analysis.html, 2020.

[64] S. Thornton, "Data Drones," https://www.nationalgeographic.org/article/data-drones/, 2014.

[65] F. Trujano, B. Chan, and R. R. May, "Security Analysis of DJI Phantom 3 Standard," Massachusetts Institute of Technology, Tech. Rep., 2016.

[66] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *USENIX Security Symposium*, 2018.

[67] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[68] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/, 2013.

APPENDIX

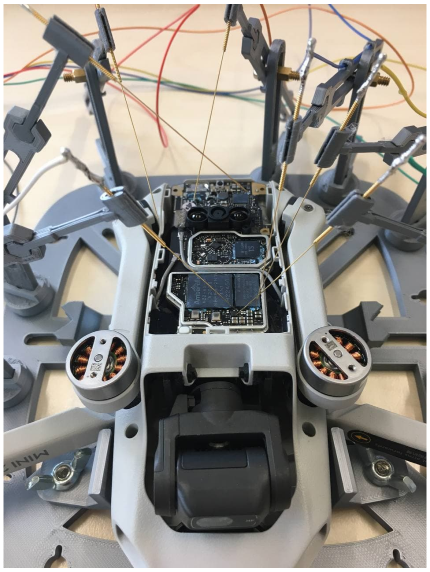
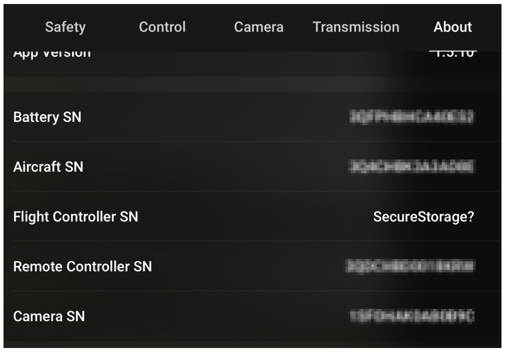


Fig. 11. The 3D printed PCB workstation with a fixed DJI Mini 2, for examining the PCB test pads.



App version 1.5.10

Battery SN [redacted]

Aircraft SN [redacted]

Flight Controller SN SecureStorage?

Remote Controller SN [redacted]

Camera SN [redacted]

Fig. 12. An arbitrary serial number of the flight controller, displayed by the DJI Fly App.

```
#  -------- drone_info_s    struc ; sizeof=0x5b
#1  00000000 len_pack          byte
#1  00000001 zero_byte         byte
#1  00000002 version           byte
#2  00000003 sequence_num      unsigned short
#2  00000005 state_info        unsigned short
#16 00000007 serial_num        char[16]
#4  00000017 longitude         int
#4  0000001b latitude          int
#2  0000001f altitude          short
#2  00000021 height            short
#2  00000023 v_north           short
#2  00000025 v_east            short
#2  00000027 v_up              short
#2  00000029 yaw               short
#8  0000002b gps_time          unsigned long long
#4  00000033 rc_latitude       int
#4  00000037 rc_longitude      int
#4  0000003b home_longitude    int
#4  0000003f home_latitude     int
#1  00000043 device_type       byte
#1  00000044 uuid_len          byte
#20 00000045 uuid              char[20]
#2  00000059 crc               unsigned short
```

Fig. 13. The structure of a DroneID packet

```
// Header
Magic:   packet[0] = 0x55
         pktlen = len(packet)
Length   packet[1] = (pktlen + 2) & 0xff
Version  packet[2] = (((pktlen + 2) >> 8 & 0x03) | 0x04)
CRC8     packet[3] = calc8([Magic, Length, Version])

// Transit
SrcID    packet[4] = (0 & 31) | ((sender_index & 7) << 5)
SrcType  packet[4] = (packet[4] & 0xe0) | (sender_type & 31)
DestID   packet[5] = (0 & 31) | ((receiver_index & 7) << 5)
DestType packet[5] = (packet[5] & 0xe0) | (receiver_type & 31)

Counter  seqno = randrange(0,32767)
         packet[6] = seqno & 0xff
         packet[7] = (seqno >> 8) & 0xff

// Command
cmdType  packet[8] = (0 & 127) | ((packet_type & 1) << 7)
 ackType packet[8] = (packet[8] & 159) | ((ack_type & 3) << 5)
 Encrypt packet[8] = (packet[8] & 248) | (enc_type & 7)
cmdSet   packet[9] = cmd_set
cmdID    packet[10] = cmd_id

// PL & CRC
Payload  packet[11:len(payload)]payload
CRC16    crc16 = calc16(packet)
         crc[0] = crc16 & 0xff
         crc[1] = (crc16 >> 8) & 0xff
         packet = packet + crc
```

Fig. 14. Calculation of the different fields of a DUML command

```
// Header
Magic:   packet[0]
Length   packet[1] | ((packet[2] & 0x03) << 8)
Version  packet[2] >> 2
CRC8     packet[3]

// Transit
SrcID    (packet[4] & 0xe0) >> 5
SrcType  packet[4] & 31
DestID   (packet[5] & 0xe0) >> 5
DestType packet[5] & 31
Counter  ((packet[7] & 0xFF) << 8) | packet[6] & 0xFF

// Command
cmdType  packet[8] >> 7
 ackType packet[8] >> 5 & 3
 Encrypt packet[8] & 7
cmdSet   packet[9]
cmdID    packet[10]

// PL & CRC
Payload  packet[11:pktlen-2]
CRC16    [packet[-1], packet[-2]]
```

Fig. 15. Parsing of the different fields of a DUML command