Drone Security

Ian Hogan, Samuel Arwood, Hari Saran, Hien Nguyen, Bogac Sabuncu

Abstract—As popularity and media attention of unmanned aerial vehicles (UAVs) increases government regulations via the Federal Aviation Administration (FAA) are continuing to change in order to keep people and facilities safe from malicious intentions. While one attack vector is to focus on WiFi access to compromise video streams and flight controls we explore an attack vector of compromising the radio frequency (RF) signals transmitted from the drones controller. We evaluate the difficulty and environment required to compromise the video stream and flight controls of an affordable and popular commercial drone. By sniffing and reconstructing the controllers implementation of Frequency Hopping Spread Spectrum (FHSS), we propose a method to obtain a completed frequency hopping table of any drone, allowing a user to track and spoof flight controls to a targeted drone.

1 Introduction

In recent years the popularity of drones has increased rapidly, with approximately 2.5 million drones being flown in America in 2016 with an estimated 7 million drones projected for 2020 this inadvertently causes security concerns. [1] While a majority of these drones are used by hobbyists, an increasing number of these drones are being used in a commercial setting including one of the largest companies in the U.S.. Amazon has entertained the thought of using unmanned fully autonomous drones in order to complete deliveries with the first Prime Air Delivery occurring back in December of 2016. [2]

As a technology becomes more and more popular and valuable only time will tell when that technology will become valuable enough for someone to spend time to find an unwanted exploit by the manufacturers. Earlier in 2017, UT Dallas Cyber-Physical Systems Security Lab had discovered a Wi-Fi based vulnerability in the Amazon best seller Chinese-made drone. [3] The drone allowed anyone to gain control through an unprotected access point into a file transfer protocol (FTP) server. Once the hacker gains access to the FTP server the lab was able to overwrite system files gaining root access and from there all systems of the drone are compromised. While this is one method of attack, if no access point is available we evaluate how difficult it is to sniff and spoof the control signals being sent over radio frequency (RF) controllers to the drone.

With such a large number of drones or targets at a rapidly increasing number corporations, government, and military facilities are quickly attempting to make their facilities secure to malicious drone attacks. The U.S. Air Force even held an internal Commanders Challenge in 2016 with a goal to develop a layered base defense system capable of warding off or destroying suspicious unmanned aerial vehicles. [4] Locally, one of BUs graduates, Hanna Timberlake, at Hanscom AFB participated on a team consisting of junior engineers and program managers. The team developed a physical defense by implementing a customized shotgun shell fitted with a netting material to physically halt any unwanted UAVs.

2 Drone Camera Application Hacking

The Drone MJX X400W has a android and iOS app that can view the drones video footage.

The android version : https://play.google.com/store/apps/details?id=com.mjx.mjxh&hl=en

The user receives access to the drones video feed through a company provided android application. Any compromising of this app would allow a non intended user to receive the video feed, leaving the privacy and security of the original user no longer safe. Our goal is to receive the video footage from the drone, depriving the user access to the video feed of drone, stopping the drone from emitting video footage, and capturing/recovering the video data stream.

2.1 Reverse Engineering the App

In order to accomplish our goal, the first step is to gather information on how the app actually works. To accomplish this we need to reverse engineer the source code of the app, which can be retrieved by disassembling the app code.

The MJX H android app contains five activities including Main, Control, Monitor, Scan and Base Activity. The Main Activity consists of the opening page which contains two options that are clickable, Monitor and Control. Upon selecting an option the user is redirected to the Monitor and Control activities, respectively. The Monitor Activity is used to monitor the drones camera feed. The Control Activity is generally used to control the drones motor functions, however this feature is not applicable in the drone we are evaluating.

2.2 De-authenticating the User

While investigating the app we noticed that even though multiple phones can connect to the drones WiFi only the initial connection is sent the video stream. From this, our first attack must be to deauthenticate that user from the drone and initialize with the app before the intended user, resulting in taking over the video stream.

This can be achieved by using the aircrack-ng tool. This tool allows the user to locate the broadcast IDs of devices that are connected to a specific WiFi address. Once this is found the tool allows the user to send an infinite amount of deauthentication packets towards the targeted client. This results in preventing them from re-authenticating themselves on that WiFi access point. After doing this we are able to connect to the drones WiFi ourselves and see capture the video feed. However, it is likely that the user will think there is a problem and attempt to reset the drone or the wifi connection.

Process:

1) Start your wifi adapter in monitor mode:
    a) airmon-ng start wlan0
2) Searching for the bssids of the drone wifi and connected devices:
    a) Airodump-ng mon0
    b) Airodump-ng –bssid Drone Bssid mon0
3) Sending the deauthentication packets
    a) aireplay-ng -0 0 -a Drone Bssid -l target bssid mon0

2.3 Denial of Service Attack

Another attack vector we pursued against the drones video stream is a Denial of Service (DoS) attack. This attack causes the video stream to be overwhelmed, rendering the video feed of the drone useless to any user. This can be accomplished by using the following script, Figure 2, with the tool Ettercap.

```
if (ip.src == Drone IP' || ip.dst == Drone IP')
{
        drop();
        kill();
        msg("DoS Attack\n");
}
```

Fig. 2: Denial of Service Script

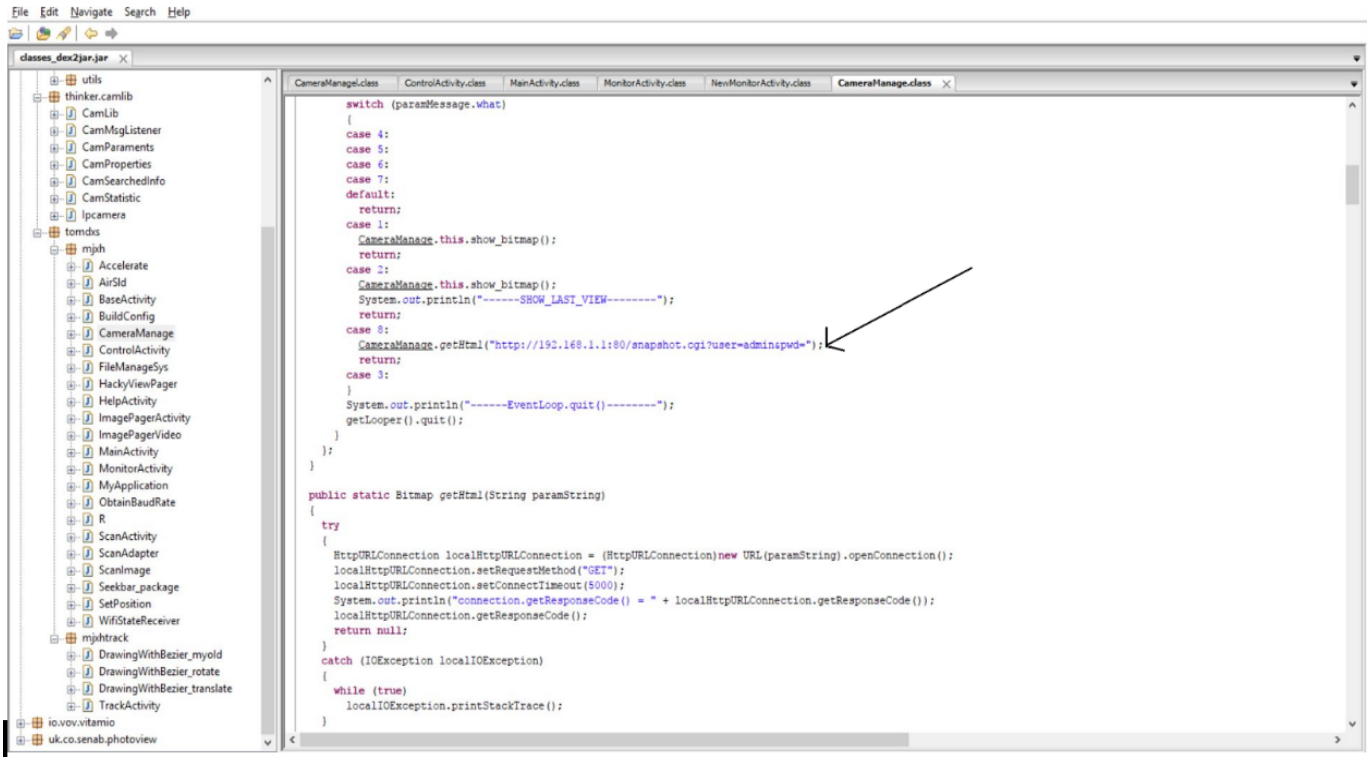

Fig. 1: Decompiled Application Code

This script causes Ettercap to grab any packages coming or going to the drone and destroying them before they arrive to the intended location, meanwhile sending the destination device an RST signal. Which means that the outgoing video stream from the drone is killed before it can reach the user.

2.4 Capturing the Packets Sent From the Drone

Even though the video is streamed to only one device, it does not mean that you cannot retrieve the communication between those two devices. However to successfully acquire the packages we need to be in the middle of the connection.

Man in the middle attacks can be implemented again using Ettercap. The program will let you select the host and the destination MAC address and position you in between the connection. By doing so Ettercap makes the drone think you are the phone and vice versa, while forwarding any communication to the corresponding destination.

Using TCPDump we are able to collect the dump files amongst the devices and these packets can be viewed in WireShark. Each package is separated into header and data sections. The package header contains metadata information such as the source/destination IP address, protocols, port, data length, etc. On the other hand, the data portion solely contains the data to be transmitted from the source to the destination. Since we know that the data sent between the drone and our mobile device is in the form of a video stream, we want to capture all the packages sent from the drone and assemble them into a playable video stream. Wireshark has an option to follow User Datagram Protocol (UDP) stream and this feature can be used to achieve the mentioned task.

This approach has one apparent issue however, as the UDP stream also contains the header portions of all packages. Thus, we need to remove all these header portions to retrieve only the pure data. Another issue has to do with the fact that we are not sure how the

data itself is encrypted. The decompiled MJX H drone APK shows that the developers do indeed use RSA and MD5 techniques from built-in Java libraries to encrypt the data, but it is not clear how and where the methods are used. Thus, our next step is to decrypt the captured packages.

2.5 Decrypting the Captured Packet

In Figure 3 we see the UDP packet , source being the drone with ip 172.16.10.1 and destination is the mobile receiving the data with ip 172.16.10.3 . The data is in Data protocol ,we tried to decode it to a rtp protocol and some other protocols and we found the data does not belong to them , the app has a MD5 decryption class which shows that the data could be encrypted.

2.6 Compiling the Decompiled App

We compiled the decompiled app and visually looks similar to the original, seen in Figure 4, Figure 5, and Figure 6. However, it does not receive the data from the drone which hampered the chances of us corrupting the app to check if we could get the footage along with user getting it.

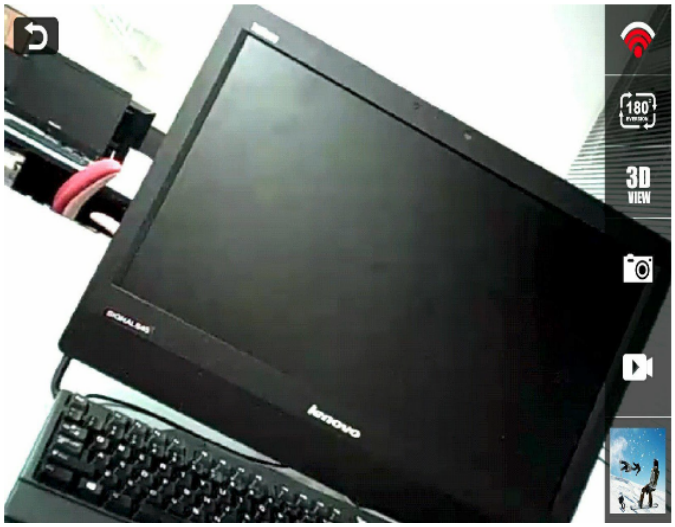

Fig. 4: Working Camera Feed

Fig. 5: Non-Working Camera Feed

Fig. 6: Front Screen of the Application

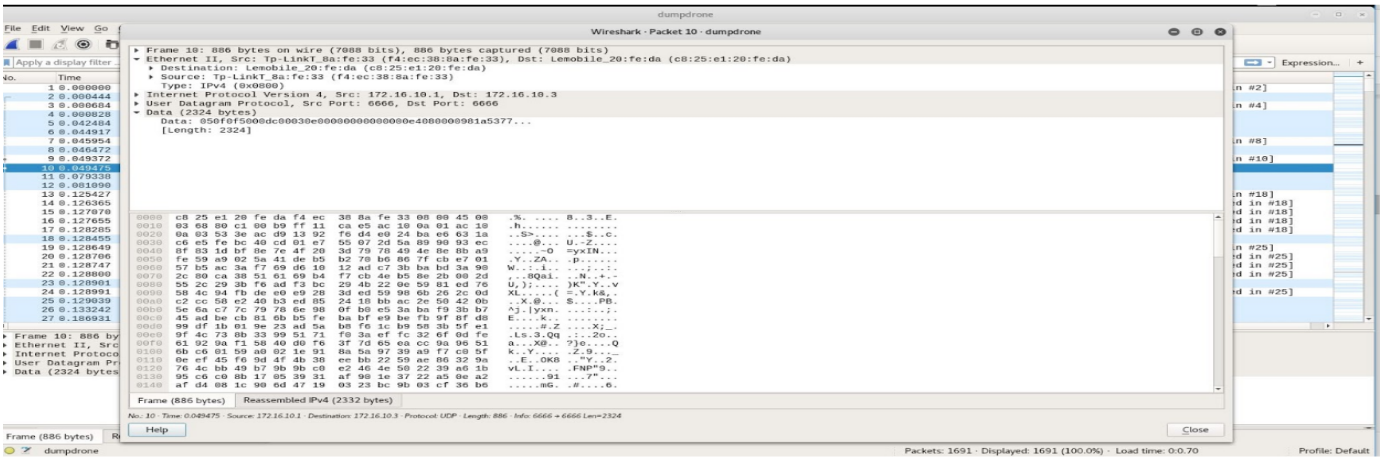

Fig. 3: Captured Packet

3 Drone Controls Hacking

In order to spoof the drones controller we first take a look at what signals the controller is designed to produce. By using the FCCs database we are able to find the family of controllers and narrow in on the specific controller used for the DBPower MJX X400W, from this we are able to pull the following data:

1) Freq Range: 2.4-2.483GHz
2) FHSS Transmitter
3) Data Packets: GFSK Modulated
4) Channel BW: 1.5-2MHz

3.1 Frequency Hopping

With this knowledge, we know due to the controller implementing Frequency Hopping Spread Spectrum (FHSS) we are unable to use only one frequency to sniff and spoof to decode the drones control data. FHSS, similar to its name, utilizes multiple frequencies within one band in order to make the signal more robust to narrow-band interference. While the entire bandwidth is rather large, in our case approximately 80MHz wide, the effective bandwidth is virtually the same since the transmission only occurs for a small portion of time ranging from milliseconds to 100s of microseconds. One of the challenges that comes with FHSS is the synchronization between the transmitter and the receiver. One of the methods is the receiver, or drone in this case, choosing a frequency within the predetermined frequency hopping pattern. The drone will monitor this frequency until it receives known data from the controller, once this occurs the drone is able to follow the controller using the frequency hopping table. Unfortunately, there are two popular methods of creating a frequency hopping table. The first, is that the drone and controller, upon production, will be encoded with the same table and will maintain this table through its lifetime. The second and more difficult to spoof, is that upon reconnection between the drone and the controller, the controller will send a new table every connection to the drone. In the case of the latter, this becomes increasingly more difficult if the drone uses this method especially since the battery life of the drone is approximately 10 minutes of flight time.

From this information we develop several attack vectors to investigate with the primary choices boiling down to analyzing the raw data from the chip or receiving the information over the air. By analyzing the raw data from the chip we would be able to completely ignore one of the hardest factors at first by avoiding the inconsistency and noise of the electromagnetic spectrum. Looking into any documentation for the drone created by the manufacturer however leads to dead ends since all of the manuals for the chips used are in chinese. To view the datastream from the internal chip a digital logic analyzer or equivalent tool would be required in order to capture the bitstream,

however this resource was unavailable. By evaluating the time vs reward of implementing this method viewing the spectrum over air is the direction taken due to fewer initial roadblocks encountered such as resources and background knowledge of the device.

3.2 Wireless Sniffing

In order to receive the information transmitted by the controller we must first be able to view the entirety of the 80MHz spectrum. To do this we explore the use of spectral analyzers, oscilloscopes, and software defined radios (SDRs).

Viewing an 80MHz spectrum at a speed fast enough to capture all of the frequency hops became the two limiting characteristics in our decision of which technology to pursue to visualize the spectrum. First, looking into the spectral analyzer and tuning to a center frequency of 2.44GHz we are able to visualize the entire spectrum ranging from 2.4GHz to 2.8GHz. At an initial glance we are able to detect activity that we were sure was being emitted by the controller. By viewing each hop, the bandwidth matches the expected 1.5-2MHz as opposed to either WiFi or bluetooth technologies which have much larger bandwidths, as much as 20MHz BW.

Unfortunately the spectral analyzers available have a limiting factor in the refresh rate of each 100MHz sweep. With the end goal being able to produce a frequency hopping table, even being able to fully visualize the entirety of the spectrum is non beneficial if we miss high amounts of activity due to slow refresh rates.

3.3 SDR

While a conventional radio or receiver/transmitter contains hardware specifically attuned to a small range or single band of the electromagnetic spectrum, SDRs contains general hardware built to handle a large range of frequencies and data rates. By using GNU Radio and other open source platforms, the SDR can be programmed to change Rx/Tx frequency, modulation schemes, and encoding/decoding using a flowgraph grc file type, an example can be seen in Figure 7. One of the most popular SDRs, used widely by researchers and academia is the USRP N210. The USRP N210 can visualize large portions of the spectrum, up to 40MHz of BW at 2.4GHz, and while using the appropriate FPGA daughterboard has 12-bits of accuracy. The connection speed is one of the highest in the market due to using a 1Gb ethernet connection. While all of these factors would guarantee high accuracy in receiving the signal the USRP N210 is at the upper range of SDRs coming in at approximately $2000. Due to lacking resources and evaluating the likelihood an average hacker in terms of software and/or hardware can cost effectively compromise the controls of an average drone we explore a more affordable option. Using available resources we decide to move forward with the HackRF One, a middle range SDR with a more economical price point of $300, applicable specifications can be seen in Table 1.

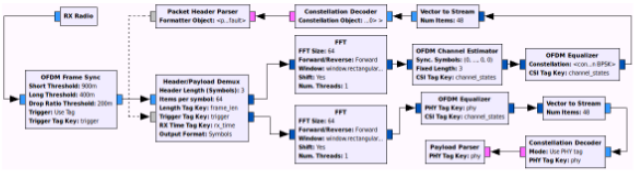

Fig. 7: GNU Radio Flowgraph

3.4 Architecture and Environment

HackRF One	Accuracy	8-bit
	Connection	USB 2.0 (40MBps)
	Rx BW	20MHz
	Rx/Tx	Half Duplex
Data Collecting Computer	OS	Ubuntu 16.04
	CPU	i5 Quad-core
	RAM	8GB
Antenna: HG2403RD-SM	Frequency	2.4GHz ISM Band, IEEE 802.11b/g
Environment	Noiseless	2.4-2.5GHz ISM Band
		Anechoic Chamber/Forest
Data Processing Computer	OS	Windows 10
	CPU	Xeon 6-core
	RAM	32GB

TABLE 1: Architecture and Environment

3.5 Partial Sequencing

Even the high end USRP N210 cannot visualize the entire 80MHz spectrum in the 2.4GHz band so viewing and recording the signal information in one attempt is no longer an option. A technique used to reconstruct the separated spectrum is through partial sequencing noted in a paper attempting a similar analysis of a drone using FHSS, later discussed in the relative work section. [5] By viewing and recording the signal multiple times spanned across the spectrum, theoretically, we would be able to piece together the entirety of the spectrum. Since we are not merely attempting to get an image of the spectrum and require an ordered list of frequencies to produce the hopping table the coverage of the recordings must have some overlap. This can be seen visually in Figure 8, the desired coverage would require at least one active channel of each signal file to exist in another file. With this similarity each partial sequence has a relative time connection with the other signal files, resulting in the possibility to reconstruct the 80MHz spectrum.

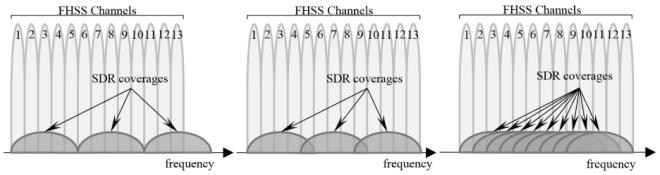

Fig. 8: Partial Sequencing

3.6 Spectrum Gathering and Visualization

Before recording the raw signal data over the spectrum we must first investigate a few questions previously discussed:

1) Does the controller send the drone a new sequence or frequency hopping table as it initializes?
2) How active is the spectrum, how many active channels?
3) How much overlap must be recorded to partial sequence the spectrum?

3.6.1 Frequency Table Generation

By focusing the SDR on several frequencies known to be active (active channels can be seen using the FFT GUI in GNU Radio shown in Figure 10), we proceeded to investigate the behavior when the controller first initializes. Even while keeping the drone off and unconnected with the drone, the controller continues to enter directly into the frequency hopping sequence seen previously while the drone is connected. Doing a few iterations of this test while the drone is off and on we have a high level of confidence that the frequency hopping table between the drone and the controller remains stable and is hard coded from manufacturing, especially since the drone has no manner of transmitting to the drone to acknowledge an initialization sequence.

3.6.2 Channel Activity

Since the total possible spectrum is 80MHz and the bandwidth of each frequency hop is approximately 2MHz this leaves a total of 40 different possible channels in our spectrum. By maximizing the HackRFs receiving bandwidth to 20MHz and shifting across the spectrum

from a center frequency of 2.41GHz to 2.48GHz we are able to visually determine the relative channel activity of the system. From preliminary sweeping we notice that out of all 40 possible channels only around 10 have activity with most activity in the upper band closer to 2.48GHz. Unfortunately, this causes increased difficulties setting up overlapping for partial sequencing.

3.6.3 SDR Overlapping

With the channel activity being so minute and the active channels being so spread out this compromises the success of a complete reconstruction of the partial sequencing. For example, one of the promising looking channels based on strength (power) and bandwidth is located at the lower frequency of 2.41GHz while the next activity seen was located at 2.43GHz, unfortunately based on the strength, shape, and bandwidth of the signal we did not have a high level of confidence that this was an active channel. With 2.43GHz being the edge of the SDRs bandwidth, if this signal is simply noise, 2.41GHz becomes broken in the sequence reconstruction. Resulting from the large spread of the active channels with low levels of confidence between several signals being either noise or harmonics produced from noisy controller circuitry we maximize the effective recording bandwidth and the overlapping of signal files. This ends with each file containing 20MHz bandwidth and shifting each files center frequency by 5MHz, i.e. files at 2410, 2415, 2520, 2425, 2430, 2435, 2440, 2445, 2450, 2455, 2460, 2465, 2470, 2475, and 2480MHz.

3.7 Signal Processing

The input signal from the HackRF is a quantized representation of the current band we are recording. The HackRFs input stream is a series of 8-bit (4-bit real, 4-bit imaginary) values at a rate of 20 Msps (Mega-samples per second). To find the set of all hopped frequencies we first need to convert the input time domain signal to the frequency domain. Figure 9 shows a GNU Radio time domain plot for the frequency recording centered at 2.47GHz. In this domain, the signal does not give us enough, or any, information about the characteristics of the waveform. Figure 10 show the plot of the frequency domain from GNU Radio.

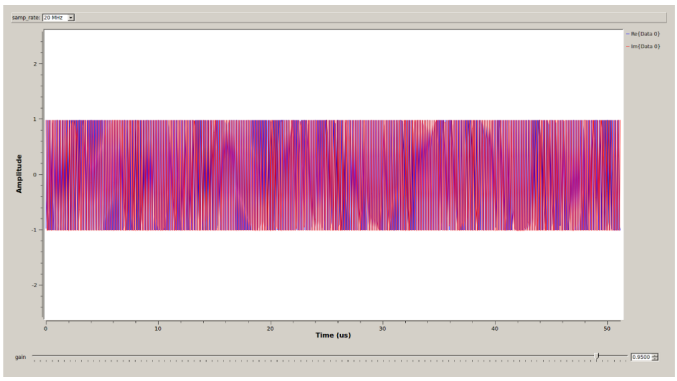

Fig. 9: Time Domain Visualization of 2.470GHz Signal

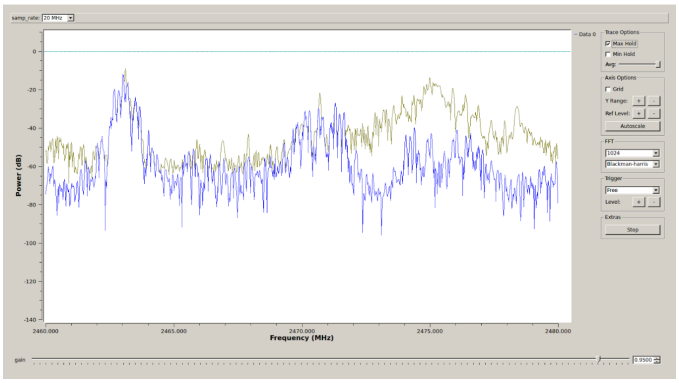

Fig. 10: Frequency Domain Visualization of 2.470 GHz Signal

In the frequency domain we can gather more information about the signal. Figure 10 shows that at this point in the signal we have a signal spike at 2.48 GHz and a spike building at 2.475 GHz. Using this information, spread out over time through the entire signal, we can try and rebuild the original frequency hopping table of the transmitter. We output the original signal from GNU Radio and processed everything in Python, using the numpy module to do any signal processing we needed.

The original signal is a 3-dimensional signal, X=frequency, Y=Power (dB), Z=time

(samples per second). We are using the term frame in the following explanation. Frame is a small window into the signal at a set number of samples. We used a frame size of 1000 samples. This gives us a frame of 50 microseconds in which we need to identify any signal spikes.

$$\frac{20\ \text{Msps}}{1000\ \text{s}} = 50\ \mu\text{s}$$

The specification sheet for the RF chip in the drones remote control told us that each channel will have approximately 1.8 MHz bandwidth, which was also verified through visual inspection. Since we have scaled our frame size from 20 Msps to 1000 samples/frame we also need to scale the bandwidth in which we are seeking a single spike. To do this we use the following equation:

$$\frac{\text{Bandwidth} * \text{Frame Size}}{20\ \text{Msps}} = \frac{1.8\ \text{MHz} * 1000\text{s}}{20\ \text{Msps}} = 90\text{s}$$

The first Python file, files.py, analyzes the raw recorded signal and outputs all of the signal spikes. To find all spikes in the incoming file, we start by taking the FFT of the raw signal. This gives us a signal similar to what is in Figure 10. Now we run that signal through a peak detector to find every possible spike. If you look closer at Figure 10 you will notice that the signal spikes are not smooth. This results in multiple peak detections for each true spike in the signal. To overcome this problem we look for the relative maximums within the peak detectors output. We set the threshold for the peak detector to 10 percent. This means that a signal must rise by at least 10 percent of the maximum value of the signal before a peak can be recorded. Figure 11 shows a graph of the FFT, detected peaks, and relative maximums.

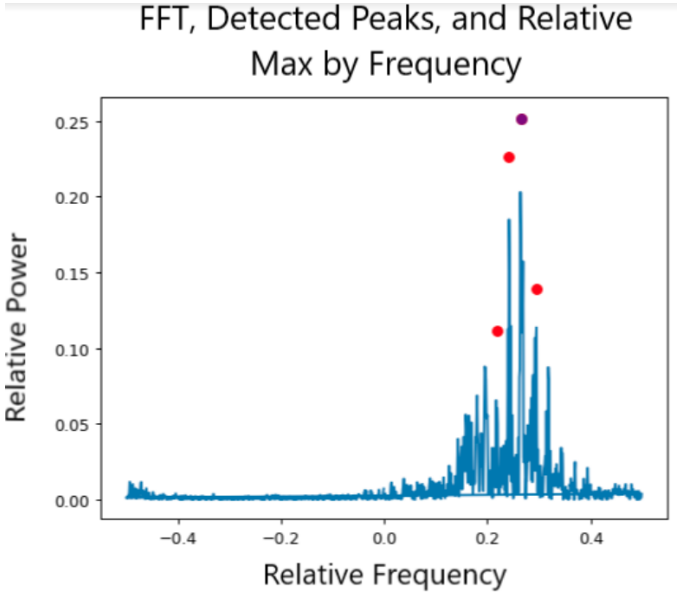

Fig. 11: FFT, Detected Peaks, and Relative Max by frequency

In Figure 11 the blue graph is the FFT of the input signal, red dots show where the peaks were detected, and the purple dot shows the the relative maximum found. The purple dot was found for every frame of every file and output to holding files. Each recorded signal file ranges in size from 3-4 GBs. Because of the large size of the signal information the processing was broken into two more scripts: post_proc.py and sequence.py. After running the 16 raw input files through files.py we were able to reduce the number inputs into post-proc.py because 6 of the input files produced no results. These findings were manually verified to confirm its accuracy.

The post-processing script takes of output of peaks and processes it along the time dimension. There are two parameters that can be tweaked in this script. First, is the frame distance. The fame distance will set how many frames there must be between reported peaks of similar frequency. The second parameter is the similarity frequency. We used 700 frames as our frame differential between spikes and a separation frequency range of 900 MHz (half the bandwidth). After the processing step we noticed that our output had file specific errors. To correct these errors we ran a makeshift switch statement based on the filename to fix

the errors seen in Table 2.

After manual analysis of post-proc.pys output we were able to reduce the number of inputs to sequence.py to 4 files. The last step was to sequence the frequency spikes we found over time. To do this, we created sequence.py. We read in the frame number and frequency found by post-proc.py for each file into a dictionary for sequencing. The dictionarys (key, value) pair was set as the filename and the tuple of (frame number, frequency). We sequenced a single file at a time looking for patterns in the frequency position of the tuples. The pseudocode below shows the overall algorithm used to do the sequencing.

```
for window_size in range(file_length/4):
    for pos_in_file in range(entries_in_data[file] - window_size):
        found[pos_in_file] = tuple(data[pos_in_file:pos_in_file+window_size])
    maybe_item = set(tuple(found))
    distance = tuple(found)
    distance = tuple(x-y) or tuple(x-y, y-z)
    if len_maybe_item != len_found
        max_maybe_items = count(for each in maybe_item)
```

We define a range of window sizes to check for the file. This step takes a while since we have to check for tuples of length 2 through about 900. We take this window if tuples and slide it down the file and store every single possible tuple found. We do this for every window size. If we take the set of each window size we can find out how many unique tuples were found for each window size. Next, we count how many times each of these unique tuples occur in the large found array. The output looks similar to the following:

Pattern found for ws: 2 Max: 1890 List: (2475, 2463)
Pattern found for ws: 3 Max: 1768 List: (2475, 2463, 2475)
Pattern found for ws: 4 Max: 1302 List: (2475, 2463, 2475, 2463)
Pattern found for ws: 5 Max: 734 List: (2463, 2475, 2463, 2475, 2463)

By inspecting the output we noticed there were some file specific tweaks that needed to be added, as well as, we went back and added the ability to print the distances with the results. After fixing the output we manually inspected each output to determine the best output sequence to use for each file. We also noticed that either a file contained a single frequency are the exact same output as another file. We assumed we would get some overlap considering each recording was 5MHz apart, but we were surprised to see so many files with the exact same outputs or only a single frequency. In the end, we used 4 files to create a frequency hopping sequence: 2410, 2460, 2465, and 2470. We decided to include 2410 even though the only information we have is the timing between its own transmissions. Table 3 includes all of the information we used to put together the sequencing.

4 Results

Using this information we were able to put together a possible frequency hopping pattern. That pattern is shown in Table 4.

The only way to test our result would be to design and build a spoofer for the remote control. Unfortunately, the given time for this project doesnt allow for a separate hardware build. Post analysis of our results in reverse engineering the frequency hopping table leaves us with a few questions. Using the parameters we decided on gave us this result, but with tweaks of these parameters we would most likely get a different frequency hopping table. With a longer project we could try each of these tables against the drone to essential brute force the desired result. Another factor to consider if someone were to move forward with a project like this, is that the signal itself still needs to be decoded. We know that the signal used Gaussian Frequency Shift Keying, but deciphering the output of a GFSK demodulator to reverse engineer the actual commands being sent is no small task.

File	Correction
2455	Fix GNU Radio wrap around for signal leakage - 2.455GHz should be 2.475GHz
2467	Ignore this value (Doesn't show up enough to be a reliable data point)
2478	Ignore this value (Doesn't show up enough to be a reliable data point)
2462	Fix rounding error - 2.462GHz should be 2.463GHz
2476	Fix rounding error - 2.476GHz should be 2.475GHz
2474	Fix rounding error - 2.474GHz should be 2.475GHz

TABLE 2: Files and Their Corresponding Corrective Actions

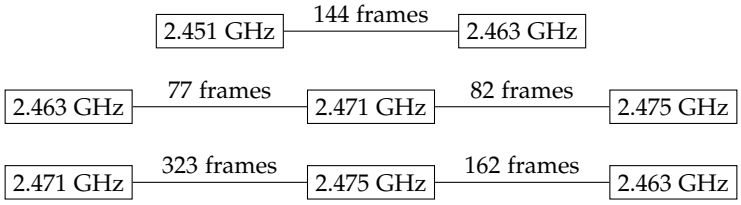

TABLE 3: Most Frequent Frequency Patterns and Their Average Frame Distances

2.463GHz	2.471GHz	2.475GHz	2.410GHz	2.475GHZ

TABLE 4: Our Frequency Hopping Table

5 Related Work

Several groups of academia have proposed and implemented similar attack vectors towards drone controllers. Most notably our work begins its attack vector design based on that of the work of Hocheol Shin et al from the Republic of Korea. [5] Using the USRP N210 they have created an attack model more heavily developed using the open source platform of GNU Radio. By implementing the USRP to follow the hopping sequence of their targeted drone, the raw baseband signal is able to be recorded and constructed over the entirety of the hopping sequence. While our proposed implementation as well as the implementation proposed in [5] are dependent on SDRs to do the bulk of computation and RF tracking researcher Jonathan Andersson has created his own hardware to hijack drones.

Anderssons custom hardware dubbed Icarus abuses the DSMX remote control protocol by discovering the secret key that is transmitted between the drone and the operators controller. By sniffing the non-secure protocol Andersson is able to brute-force the key and is able to impersonate the original operators controller. The resulting key is used with Icarus to perform a timing attack against the targeted drone. Icarus is able to transmit a signal to the targeted drone slightly before the signal of the original operators controller, resulting in Icarus being accepted and the rejection of the original operators request. [6] While this implementation is quick and effective it however is only a vulnerability of drones using the DSMX protocol, designed and licensed by Horizon Hobby.

6 Conclusion

From this project we have evaluated a highly accessible, relatively cheap, and popular drone that an everyday family might purchase. While it has been seen that drones similar to the one evaluated here, the level of understanding of multiple different fields must be utilized including radio frequency, signal processing, coding, and SDRs. Required hardware in order to obtain this information is also not as easy as previously thought, while we are only able to visualize and create a frequency hopping table using (at minimum) a lower end SDR at $300, equipment would be required to use this table to spoof the controller's signal. A quick evaluation using the recommended components from the work done by Hocheol Shin et al estimates

over $150 worth of additional components minimum. While larger drones with higher payoffs would make this endeavour more rewardable, the average $70 drone that uses FHSS is not worth the time or effort of an ”average” hacker.

7 Work Breakdown

Name	Project Sub-assignment	Percent Work
Samuel Arwood	Reverse Engineering FHSS	34%
Ian Hogan	Reverse Engineering FHSS	34%
Hari Saran	Android App & WiFi	20%
Hien Nguyen	Android App & WiFi	4%
Bogac Sabuncu	Android App & WiFi	8%

TABLE 5: Work Distribution by Subtopic and Effort Expended

8 Citation

References

[1] Atherton, Kelsey D. The FAA Says There Will Be 7 Million Drones Flying Over America By 2020. Popular Science, 24 Mar. 2016, https://www.popsci.com/new-faa-report-stares-in-face-drone-filled-future

[2] Amazon Prime Air. Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more, https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011

[3] Fox-Brewster, Thomas. Watch A Very Vulnerable $140 Quadcopter Drone Get Hacked Out Of The Sky. Forbes, Forbes Magazine, 25 Apr. 2017, https://www.forbes.com/sites/thomasbrewster/2017/04/25/vulnerable-quadcopter-drone-hacked-by-ut-dallas-cyber-researchers/#30d987ee1037

[4] Young Commander. Hanscom Air Force Base, 8 June 2017, http://www.hanscom.af.mil/News/Article-Display/Article/1207232/young-commanders-challenge-team-develops-multi-layered-drone-defense/

[5] Shin, Hocheol, et al. Security Analysis of FHSS-Type Drone Controller . 2015, https://syssec.kaist.ac.kr/pub/2015/shin_wisa2015.pdf

[6] Zeljka Zorz - Managing EditorOctober 27, 2016. Icarus takes control of drones by impersonating their operators. Help Net Security, 27 Oct. 2016, https://www.helpnetsecurity.com/2016/10/27/control-drones-icarus/