Wyze Camera Report

Makiyah Dee, Marshall Yuan, Tegan Kapadia, Andrew Verzino
Embedded System Cyber Security VIP
Georgia Institute of Technology
Atlanta, GA

Abstract— **This paper details the current progress in research into the Wyze Camera by the Embedded Systems Cyber Security VIP team of the Georgia Institute of Technology. This document covers the information the team has discovered regarding the camera's capabilities, vulnerabilities, and firmware. The goal of the research is to discover the Over the Air protocol used by the Wyze camera so that we can use it as a test case for the development of an RF fuzzing testbed.**

I. INTRODUCTION

The Wyze Camera V2 is an Internet of Things (IoT) Device. It allows for wireless connection to multiple devices, such as cameras, motion sensors, and contact sensors, that together provide the user with surveillance over many locations. When placed on a door or a window, contact sensors tell users if the object they are placed on is open or closed. Motion sensors add to detection capabilities and when triggered, can even serve as precursor events to something coming into the camera's view The "Wyze - Make your Home Smarter" mobile application provides real-time status updates for the locations being surveilled by these devices.

Recently, there has been an increase in the use of wireless cameras, cloud access, and mobile applications [1]. As a result, the need for enhanced security in these devices has risen. As the number of devices using wireless and cloud-based technology increases, so too has the risk of attacks from individuals with malicious intent [2]. Companies often show a lack of priority regarding security by conducting security testing of their IoT devices in the production phase when it is often too late for major changes. Consumer apathy regarding security (E.g., not changing default credentials on devices) couples with this developmental idleness to further security problems. [3].

An example of this trend would be the CloudPets toys released by the company Spiral Toys. The toys acted as IoT devices by allowing parents to communicate with their children via the internet. From late 2016 to early 2017, it was found that the CloudPets toys had some massive security vulnerabilities, which led to the data of over 800,000 users being leaked. However, even when the company was notified and took notice of the vulnerabilities, hardly anything was actually done to fix these vulnerabilities. Information about the companies' stocks at the time implies that they may not have had enough money to do much about the security issues. Therefore, they made the simplest of patches and continued to sell the toys as if nothing was wrong [4].

An example involving the Wyze camera occurred in 2019. Wyze confirmed that from December 4th to December 26th of 2019, a large amount of personal data was leaked to more than 2.4 million users [5]. Both of these examples should clarify just how important it is to have enhanced security on IoT devices is growing.

The main goal of this team is to decode the Over the Air (OTA) protocol of the Wyze camera since this will enable the construction of a fuzzing testbed. Fuzzing is a technique in which malformed data is fed into computer programs. Monitoring the program's output while fuzzing helps find crashes, memory leaks, and other issues, which would present a way to discover security flaws in the IoT system. Sections II and III will detail known information about the Wyze camera, the following sections will document the progress the team has made regarding the OTA protocol.

II. FUNCTIONAL DESCRIPTIONS

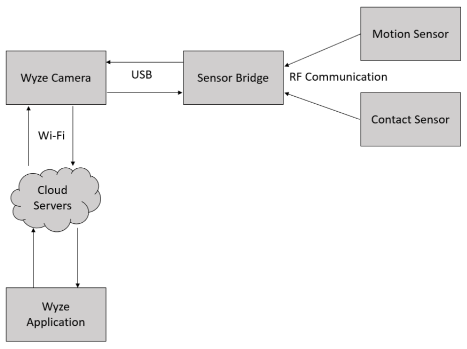


Fig. 1. Overview of System Communications

The motion and contact sensors, the sensor bridge, the camera, and the Wyze application make up most of the configuration for the Wyze IP Camera V2, as depicted in Figure 1. Through radio frequency (RF) communication, the sensors and sensor bridge exchange information [6]. Through a USB connection, the sensor bridge transmits that information to the camera. Wi-Fi is then used to connect the camera to the Wyze cloud servers, transmitting data to the Wyze app. Up to 100 sensors can communicate with the sensor bridge [6]. We have Joint Test Action Group (JTAG) access to the sensor bridge and sensors, enabling dynamic analysis and memory snapshot-taking.

A. Mainboard

Three printed circuit boards (PCBs) are sandwiched together to make up the physical Wyze Camera. However, the main camera system is made up of the main PCB containing the SoC (System on a Chip) (Figure 2), a microSD board (Figure 3), and a sensor board (Figure 4).

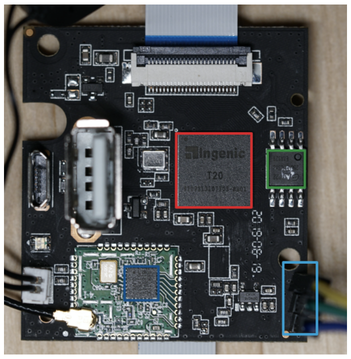


Fig. 2. PCB of the Main Board

Label Color	Red	Green	Light Blue	Dark Blue
Component	T20 SoC	Flash Mem	Serial Port	WiFi Board

TABLE I
COMPONENTS FOR MAIN BOARD PCB

One of the circuit boards within the Wyze camera is represented by the PCB in Figure 2. A T20 processor [7] using the MIPS (Million Instructions per Second) ISA (Instruction Set Architecture) powers the board. As seen in the diagram, the main board also includes flash memory, Wi-Fi, and serial access. An interactive root shell for the given Linux OS is made available by the open serial access highlighted in light blue in Table 1. This was exploited by soldering wires onto the board, giving us direct access. Additionally, a password-protected account was present but this was easily cracked to bypass it.

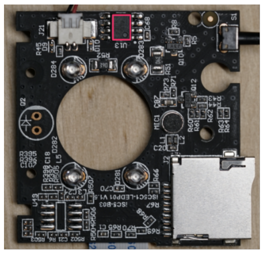


Fig. 3. MicroSD PCB for the Main Board with motor driver in red

The PCB with an SD card slot and a motor driver to move the camera is shown in Figure 3.

B. Sensor Bridge

The motion and contact sensors of the camera are wirelessly linked to the primary camera by the sensor bridge. The CC1310 microcontroller, which handles RF packet transmission and reception and packet-to-data conversion using a Sub-1GHz frequency, controls the sensor application code and messaging logic. For wireless communication, there is an antenna that is tuned to or transmits at a frequency less than 1 GHz.

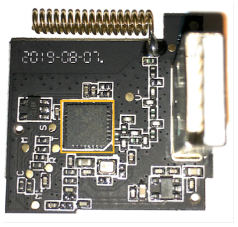


Fig. 4. Front of Sensor Bridge PCB with CC1310 highlighted in Orange

A WCH CH554T chip on the back of the PCB controls the USB-A connection that the sensor bridge uses to connect to the primary camera [8].

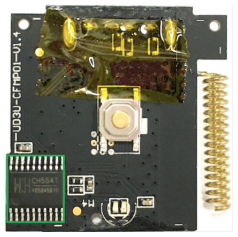


Fig. 5. Back of Sensor Bridge PCB with WCH CH554T chip highlighted in green

C. Motion Sensor and Contact Sensor

The sensor bridge, contact sensors, and motion sensors communicate wirelessly. A CC1310 microcontroller manages both sensors.

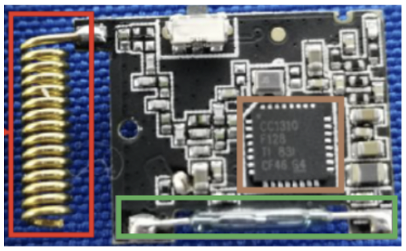


Fig. 6. Contact Sensor PCB

Label Color	Red	Brown	Green
Component	Antenna	CC1310	Magnetic Switch

TABLE II
COMPONENTS FOR THE CONTACT SENSOR

The CC1310 receives data from the contact sensor's magnetic switch over its GPIO and constructs a packet to send to the camera when the state of the switch changes.

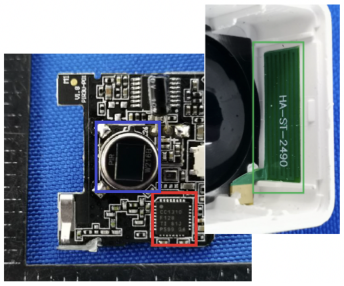


Fig. 7. Motion Sensor PCB

Label Color	Blue	Red	Green
Component	PIR Motion Sensor	CC1310	Antenna

TABLE III
COMPONENTS FOR THE MOTION SENSOR

A PIR (passive infrared) sensor on the motion sensor gives the microcontroller feedback, which is then used to construct and send wireless messages that are sent to the sensor bridge. All of this communication is handled by the microcontroller.

D. CC1310 Micro controller

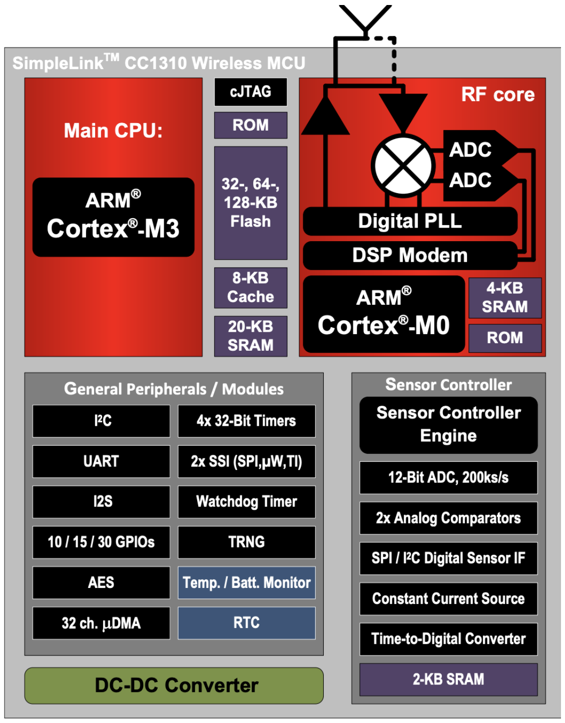


Fig. 8. CC1310 functional block diagram

Figure 8 displays the CC1310 TI Simplelink Wireless MCU that powers the sensor bridge for the Wyze Camera. Our research focused on this processor, particularly the RF core, its application, and its interactions with the primary CPU. This CC1310 radio peripheral can be configured to support a variety of protocol standards [9]. These radio protocols include ZigBee®, Bluetooth® low energy, and 802.15.4 RF4CE [10].

The main CPU in Figure 8 — also identified as the ARM® Cortex®-M3 — is known as the system CPU. The application layer and the high-level protocol stack are all handled by the main system processor [10]. It executes programs from the system flash and boot ROM. A boot sequence, low-level protocol stack, device driver functions, and a serial bootloader are all executed on this processor [10]. The system flash is a nonvolatile memory that stores configuration data and code that is executed when the device is off so that it can be accessed again after a restart [10].

Figure 8 also depicts the ARM® Cortex®-M0, which is known as the radio CPU and found inside the RF core. The radio CPU receives commands from the system CPU and schedules them into different parts of the RF core. Additionally, the radio CPU interfaces the analog RF and baseband circuitries and assembles the information bits in a given packet structure [10].The RF core operates nearly entirely from a separate read-only memory (ROM) [10] and has a dedicated 4-KB static random access memory (SRAM). [10]. Currently, the way the RF core interacts with the main CPU is what interests us the most, as it can provide information about how data is arranged in the communicated packets.

The radio doorbell module (RFC_DBELL), also known as the command and packet engine, serves as the primary means of communication between the system CPU and radio CPU (CPE) [10]. Dedicated registers, parameters stored in any of the device's SRAMs, and a set of interrupts connected at both the radio CPU and the system CPU are all components of the RFC_DBELL [10]. This means that by modifying the RFC_DBELL's parameters and interrupts, data and instructions can be sent between the system CPU and radio CPU. This module provides us with key insight into the code and operations of the CC1310 in our reverse engineering with its use and outline of the packet structures.

The SRAM stores the various parameters for specific data transactions along with packet information (TX and RX payloads) [10].

By utilizing the exposed JTAG test ports on the PCB, we soldered wires connected to a TI debugger tool (XDS110) which allows us to control the processor through the JTAG connections. This also allows us to capture the Wyze sensors' memory while in use. This capture will hold a copy of the SRAM as it was at the time of the dump. It is anticipated that informational fragments like packets and other volatile structures will be present when the SRAM is captured.

Before transmission, these structures might be encrypted or encoded by the CC1310 as part of the Over the Air (OTA) radio protocol [10]. One of our previous objectives was to determine whether the packets were encrypted or not. We determined that the packets were not encrypted due to our ability to capture and decode packets using SmartRF without difficulty, something that would not be true if encryption had been used.

The RF protocol employed by the RF Core of the CC1310 is not well understood; nevertheless, sections V and VI go into more detail about what was learned about the RF Protocol through the team's study.

E. Technical Documents

The TI CC13x0, CC26x0 SimpleLink™ Wireless MCU Technical Reference Manual [10] is the technical manual for the CC1310 MCU. Understanding how the chip is used by the Wyze camera can be made easier with the aid of the manual, which offers crucial information on how the chip functions.

For learning about Wyze firmware, we can also utilize the Texas Instruments CC13x0 and CC26x0 Software Development Kit (SDK) [?]. For developers wishing to construct applications utilizing this chip, the SDK serves as an abstract representation of the hardware. It enables programmers to communicate with the CC1310 through the application programming interface (API) provided by the SDK. The CC13x0 SDK also includes example code for the API being utilized, which is crucial for this team as it provides example use cases for the team to cross-reference. The SDK enables the team to become familiar with potential memory-based procedures, structures, and constants and their usage in applications. Ghidra is a suite of software reverse engineering (SRE) tools developed by NSA's Research Directorate in support of the Cybersecurity mission[11]. The Ghidra disassembly has already turned up certain methods that are available in the SDK. Additionally, the SDK includes sample implementations of many protocols and programs that can be used with the API offered by the SDK. There are examples of RF protocol implementations that include packet transmission and reception in particular.

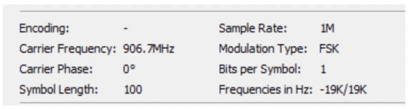
III. EXISTING VULNERABILITIES

The Wyze Camera system contains vulnerabilities that could expose sensitive information if they are exploited. The Wyze Camera is susceptible to a replay attack. Another potential issue is that the Wyze Camera's firmware is not signed, which will be explained in detail in section III-B of this paper.

A. Replay Attacks

Previously captured packets from a contact and motion sensor were replayed to the dongle (alias for the Wyze Sense Bridge, which is a hub that allows your sensor to connect the internet) by a USRP N210. Note: More information on how replay attacks were conducted can be found in the Packet Captures subsection. The USRP N210 is a software defined radio used for RF applications, and we used it to transmit and receive RF signals [12]. The packets were successfully received by the dongle as verified by the chosen input displayed in the Wyze application. This means that the dongle does not adequately verify if it is receiving a previously received message and is vulnerable to replay attacks. The first packet associated with an alert contains a 4-digit hexadecimal character value that increments upwards with each alert and resets when a sensor is powered off. When captured packets were replayed, this 4-character field returned to the value that was captured. It is unknown what these 4-characters represent, but given that they increment when an event happens, it is possible they are some kind of sequence counter. This 16-bit sequence counter increments each time there is an event (open/close, motion/no motion). Additionally, it resets to 0 every time the sensor is powered down. Recorded packets can be sent in any order, as long as the event types alternate between open and closed, and will be processed successfully by the dongle.

Expanding on the replay attack, Universal Radio Hacker (URH) modulated packets so that arbitrary changes could be made. The settings used to successfully modulate packets can be seen in Figure 10:



Encoding:	-	Sample Rate:	1M
Carrier Frequency:	906.7MHz	Modulation Type:	FSK
Carrier Phase:	0°	Bits per Symbol:	1
Symbol Length:	100	Frequencies in Hz:	-19K/19K

Fig. 9. Setting used to modulate packets

Using URH, arbitrary packets were able to be captured by the dongle without being discarded. Using a recorded contact sensor open alert, nibbles and then bytes were zeroed out sequentially and then transmitted with a contact sensor close alert in between.

B. Unsigned Firmware

In embedded systems, firmware authors can choose to sign their firmware. By signing the firmware, the author can prevent the firmware from being modified or corrupted and flashed onto a device. Signing a firmware entails generating a hash value, encrypting it with the private key of a private/public key pair, and attaching it to the firmware [13]. The firmware on the devices of the Wyze system is not signed, allowing unauthorized firmware to be flashed onto the devices [14].

IV. SENSOR LOGS

The sensors and sensor bridge communicate through radio frequency (RF) messages. The packets received from the sensors are decoded by the sensor bridge. The type of packet being sent, will contain information about the camera and state of the sensors and other devices in the Wyze network. Some commands for packets are included here, focusing on ones that we believe are relevant to the work we are doing [15]:

Name	Type	Cmd
HD_Inquiry	0x43	0x27
HD_GetENR	0x43	0x02
HD_GetMac	0x43	0x04
HD_GetSensorList	0x53	0x30
HD_GetSensorCount	0x53	0x2E
DH_AddSensor	0x53	0x20
HD_StartStopNetwork	0x53	0x1C

TABLE IV
OTA PACKET COMMANDS

By observing communication during startup and movement in front of the camera and sensors, we gathered log files for various communication scenarios between the sensors and the sensor bridge. This gave us a chance to observe various device communication transmissions. Looking through some of the RTSP (real time streaming protocol) log files, there are certain parts of the communication that directly match with the packet information, such as the type of event that it was.

A. *Packet Decomposition*

Through over-the-air (OTA) packets, the camera and dongle can communicate with each other. After reviewing the OTA protocol, we can see the communication channels and logs that reveal details about the metadata of the camera, sensors, and other associated Wyze system devices. In Ghidra [16], we have developed a serial packet register (struct) that separates the crucial portions of the OTA packets. The header of the packets is depicted in the following figure. Each packet's header structure is the same, only the contents vary, but its size is always 5 bytes.

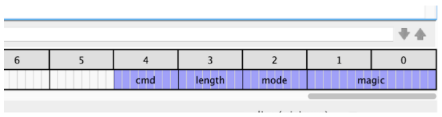
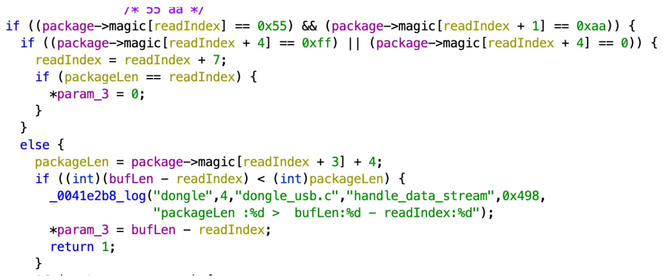


Fig. 10. Ghidra Serial Packet Struct Header

Depending on the contents and command type of the packet being transmitted, the serial packets that are sent between the camera and the sensor bridge have different lengths following the header. They will be anywhere from 7 bytes to 64 bytes in size. Checking the two magic bytes and the command byte is the first step in packet interpretation from the sensor bridge: The "mode" byte is the third byte in the packet, while the magic bytes are the package's first two indices. The command byte comes after the fourth byte, which is the length byte. The payload takes up the remaining space in the packet. The first five sections of the payload are constant and present in all packages even though the Wyze camera system employs packets of varying lengths for communication between the devices [**?**]. Since this connection occurs between the camera and the dongle, any information about the serial packet protocol could be useful in figuring out the RF protocol.

- Magic bytes: [55] [aa]: is a packet received from the dongle. Flipping the order, [aa] [55] refers to a packet from the camera sent to the dongle. These messages include commands which give information on the system.
    1) Finding the length of the packet
        - The packet's length byte is translated into a decimal value (see Figure 12 below). 3 bytes apart from the initial magic byte is and (the 4th byte). This number plus four is the total length of the packet. The 4 bytes before the packet length index can be interpreted to be accounted for by this +4. In other words, this byte truly indicates how many bytes are left in the packet length.
          To distinguish between two successive packets, one needs to know the duration of each packet. The researchers verified that the Wyze system uses packets with variable-length payloads for communication. As a result, since the number of bytes in each packet can vary, it would be very challenging to determine where one packet ended and another one began without knowing the packet length.



```
                  /* 55 aa */
if ((package->magic[readIndex] == 0x55) && (package->magic[readIndex + 1] == 0xaa)) {
  if ((package->magic[readIndex + 4] == 0xff) || (package->magic[readIndex + 4] == 0)) {
    readIndex = readIndex + 7;
    if (packageLen == readIndex) {
      *param_3 = 0;
    }
  }
  else {
    packageLen = package->magic[readIndex + 3] + 4;
    if ((int)(bufLen - readIndex) < (int)packageLen) {
      _0041e2b8_log("dongle",4,"dongle_usb.c","handle_data_stream",0x498,
                    "packageLen :%d >  bufLen:%d - readIndex:%d");
      *param_3 = bufLen - readIndex;
      return 1;
    }
```

Fig. 11. Ghidra interpretation of packet length in decimal

    2) Finding the sub_mac value of the camera
        - The camera's sub_mac value can be discovered in bytes [16:23] of the packet if it is not an ACK packet. These bytes' values, like the packet's length, are interpreted as decimal, and the sub_mac value corresponds to their ASCII values.
    3) Simply acknowledging that the previous packet is received
        - This simple ACK packet is 7 bytes long.
        - There are only two alternatives for the type byte (3rd byte): 0x43 or 0x53. Between synchronous (0x53) and asynchronous, there is this distinction (0x43). When the type is 0x43, the other device will instantly send a response packet with any payload data that the current instruction requires. The responding device will send packets corresponding to the command

byte and an ACK packet if the type is 0x53. [15]

V. Signal Analysis of the RF Protocol

Reverse engineering Wyze's proprietary RF protocol between the touch sensors, motion sensors, and the sensor bridge is the main goal of the team's present research. To proceed, the team must first determine whether the Over-The-Air packets are whitened and/or encrypted. Analyzing OTA captures of records between the sensor bridge and the sensors can help the team understand the issue.

A. Packet Captures

When a data packet is captured during transit over a data network, it can be kept and examined. To record packets moving between the dongle and sensors, an Ettus N210 SDR was employed. The packets were recorded with GNU Radio. The block diagram for data flow and the GNU Radio flowgraph used to record the packets are shown below:

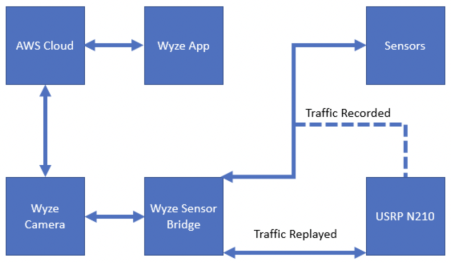
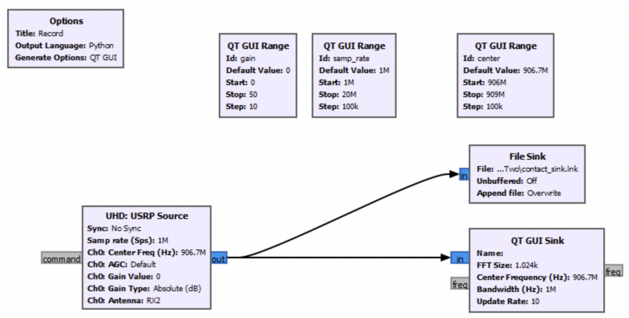


Fig. 12. Block Diagram showing where Packets and Logs were captured



```
Options
Title: Record
Output Language: Python
Generate Options: QT GUI

QT GUI Range
Id: gain
Default Value: 0
Start: 0
Stop: 50
Step: 10

QT GUI Range
Id: samp_rate
Default Value: 1M
Start: 1M
Stop: 20M
Step: 100k

QT GUI Range
Id: center
Default Value: 906.7M
Start: 906M
Stop: 909M
Step: 100k

File Sink
File: ...Two\contact_sink.lnk
Unbuffered: Off
Append file: Overwrite

UHD: USRP Source
Sync: No Sync
Samp rate (Sps): 1M
Ch0: Center Freq (Hz): 906.7M
Ch0: AGC: Default
Ch0: Gain Value: 0
Ch0: Gain Type: Absolute (dB)
Ch0: Antenna: RX2

QT GUI Sink
Name:
FFT Size: 1.024k
Center Frequency (Hz): 906.7M
Bandwidth (Hz): 1M
Update Rate: 10
```

Fig. 13. GNU Radio flow graph used to record packets for playback

WyzeSensePy, a raspberry pi, and a USB connection to the dongle—which implements the communication protocol between the dongle and camera—can all be used to directly record serial logs [17]. The WyzeSensePy printed the serial logs that contained the serial data once the RF data (data sent over-the-air) had reached its destination. The dongle served as an interface to send/receive messages to/from the contact sensor.

B. RF Overview

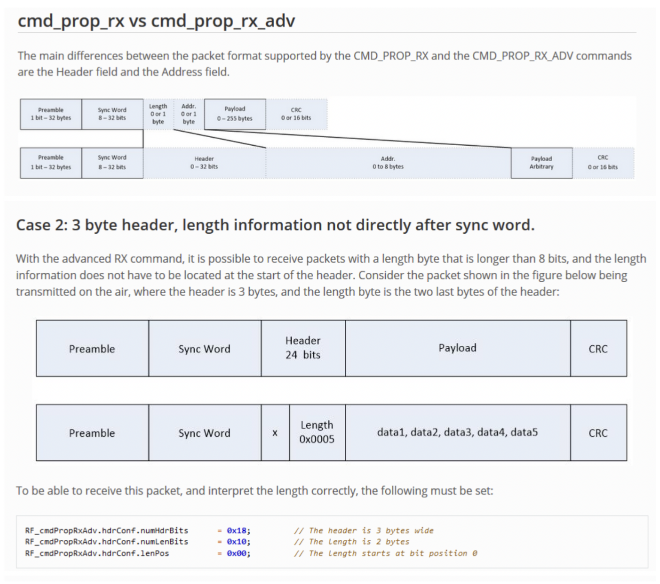
Information from the contact and motion sensor is read by their microcontrollers and modulated through Gaussian Frequency Shift Keying (GFSK). GFSK filters the data pulses and makes transmissions smoother [18]. URH [19] assists in analyzing OTA packets to understand the protocol between the camera and the dongle.

C. Sensor Over-The-Air Packet

URH can be used to gain a clearer understanding at what is happening during the time between the motion and contact sensor of the camera before the OTA packet is received. This will allow the team to gain a better understanding of the communication protocol as we can understand what happens when the system is in close to an idle position. Seeing what happens before an OTA packet is received can assist the team in seeing the changes that occurs as it is received.

D. Packet Contents

Currently, not all data fields transmitted via OTA packets are known. We suspect that OTA packets are composed of a proprietary protocol packet with a payload that contains the application level payload, which may consist of the: MAC, Battery, Counter, and Event Type. It is likely that the OTA packets from sensor-to-dongle contain the MAC of the sensor, which is used for identification, so when messages want to be received by the sensors, the sensor will look for its MAC address to see first if it is compatible to accept the message. As seen in the WyzeSensePy debug information with a marker reading "battery =...." sensors transmit how much battery they have left to the dongle via the sensor bridge. Finally, the type of event is transmitted (open/close, motion/no motion). Given that we know that the packets are largely the same (the different open and closed packets only differed with variation for battery/MAC/counter/event type) in terms of formatting due to the findings in previous semesters, we suspected that the packets were encrypted and/or whitened, but we learned later that this suspicion was false.The TI SDK has provided insight into the physical structure of the packets. The diagram below provides more information:



cmd_prop_rx vs cmd_prop_rx_adv

The main differences between the packet format supported by the CMD_PROP_RX and the CMD_PROP_RX_ADV commands are the Header field and the Address field.

Case 2: 3 byte header, length information not directly after sync word.

With the advanced RX command, it is possible to receive packets with a length byte that is longer than 8 bits, and the length information does not have to be located at the start of the header. Consider the packet shown in the figure below being transmitted on the air, where the header is 3 bytes, and the length byte is the two last bytes of the header:

To be able to receive this packet, and interpret the length correctly, the following must be set:

```
RF_cmdPropRxAdv.hdrConf.numHdrBits      = 0x18;        // The header is 3 bytes wide
RF_cmdPropRxAdv.hdrConf.numLenBits      = 0x10;        // The length is 2 bytes
RF_cmdPropRxAdv.hdrConf.lenPos          = 0x00;        // The length starts at bit position 0
```

Fig. 14. RX information

We used smartRF to inquire more information about the packet structure. In the following figures, we have listed one open, and one closed packet, respectively. When we tried to generate more packets, we believe that a possible whitening issue occurred since we could only read the data of the packets. However, the packets below, when compared to the other packets of the same type that we captured, it remained clear that a certain portion of the data could not be edited, and we believe that this could be a header to distinguish the type of event of the packet. This assumption would need further research to conclude the structure of the packets.

```
ba 5c 70 17 4f b8 69 5a 98 21 d6 81 cd b3 b3 96 92 e8 55 2b 17 34 0c 99 84 82 fe 80 c4 bf 48
c5 9f 80 47 1e 73 11 bf 83 29 a2 58 fb 66 fc 20 76 51 dc 5e 76 a2 f8 f3 6a fb da 38 91 48 28 16
7b c7 46 59 7c be d3 60 0d 08 b4 99 45 01 ed 71 7d cb 3c 61 24 62 40 84 98 ed f2 b7 de d2 09
b4 dc a0 81 db 94 b7 50 00 0a ad bd 30 27 29 2b 63 ac ff db e6 97 b4 4c 2f 32 3b 24 13 3f af
d6 d6 b1 0e 69 61 85 e1 6e 43 1b 96 63 4d 8a 9b 68 38 16 be f9 53 07 c7 7b 45 57 a4 73 39 5e
ea a2 02 74 53 33 2c d9 c7 3c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

Fig. 15. Open Event Packet

```
ba 5c 70 18 4f b8 69 5a 98 21 7d 74 3a 6a 51 ed f1 18 0e 4f 7f 20 03 e5 10 7a 23 7b f8 2d 2f ac
fd 57 9e 7a 31 76 bb e1 d4 60 c0 d3 30 db 20 5c 54 dc 5c a2 82 fc f3 6a fb da d8 70 d1 2b e5
6b c7 57 07 bb c3 18 17 b4 f5 45 47 3d 6c 66 11 fd c3 9c ae 3c 91 d8 c3 69 92 e5 bb a3 a3 35
2a 10 31 57 e4 fc 20 c9 e0 b0 24 7f 24 54 47 8c 12 52 df 5e ea 87 7e e5 11 e8 ae 75 92 1f 3e
85 94 8b 23 7f 68 b7 4d 48 a5 b8 b6 2d 83 43 41 99 d4 44 1e 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

Fig. 16. Closed Event Packet

E. Communication Protocol

When an event is detected, the sensor transmits a packet to the dongle. Once the dongle has received the packet, it replies with an acknowledgment, or "ACK", to indicate that it successfully received the event alert from the sensor. The dongle keeps the state of sensors in memory. For a contact sensor, you cannot send two events of the same type back-to-back since the second packet will result in an error rather than another event. The state-keeping appears only to be related to the state the sensor is in (open/close, motion/no/motion) and not related to the sequence counter embedded in the message.

F. Transmissions

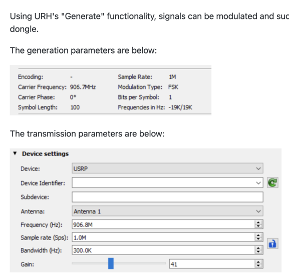
Packets were transmitted at a frequency of 906.8MHz with a modulation type of GFSK. URH, alongside SmartRF, allowed packets to be sent to the dongle and from the dongle to the host. These recordings were taken from three states of the motion sensor: motion, connect, and delete.

G. Motion Sensor

URH was used to conduct an analysis of the motion sensor captures as well as analyze the log files by parsing the contents to find commonalities. The goal of this was to gain more insight into the data being sent in the OTA packets.

H. OTA Protocol

The most critical part of our research relies on answering whether the OTA packets are whitened and/or encrypted. If one knows this information, one could manipulate and spoof messages to the Wyze Camera. To uncover the details of the OTA packets, the team reviewed work done in previous semesters and continued along the outline depicted in Figure 16.



Using URH's "Generate" functionality, signals can be modulated and successfully received by the dongle.

The generation parameters are below:

Encoding:	-	Sample Rate:	1M
Carrier Frequency:	906.7MHz	Modulation Type:	FSK
Carrier Phase:	0°	Bits per Symbol:	1
Symbol Length:	100	Frequencies in Hz:	-19K/19K

The transmission parameters are below:

Device settings

Device: USRP
Device Identifier:
Subdevice:
Antenna: Antenna 1
Frequency (Hz): 906.8M
Sample rate (Sps): 1.0M
Bandwidth (Hz): 300.0K
Gain: 41

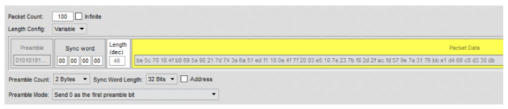
Fig. 17. Outline of Continuing Research in the OTA Protocol

Using URH, the team would create new packets by taking previously successful transmissions and arbitrarily editing some of the bits. After we downloaded the edited replay attack from URH, we uploaded it to the flowgraph to GNU-radio companion (Figure 14). In the GNUradio companion, we would see if a valid replay attack occurred with the 'new' packet. Success occurred when the edited message was recognized and captured by the Wyze Camera's sensors. Once the flow graph was executed, and WyzeSensePy was running, the newly created replay attack was tested to see if they were still valid. The results of the experiment were inconclusive because the edited replay attack was not able to be received by the dongle. There are many possibilities as to why the edited message did not go through: the cyclic redundancy check (CRC) [20] could have been incorrect, or possibly the edited piece of the packet was indistinguishable and couldn't be unencrypted or encrypted. One way to overcome this setback would be to correctly recalculate the CRC after bits are edited and/or to find the sync word. The sync word, as mentioned later, will allow the team to see where the data begins in the packet, which would allow us to directly modify the data bits rather than bits that may be important for validation, and we can try to replay it to Wyze.

I. The Sync Word

A critical part of reverse engineering the OTA protocol depends on finding the sync word. The sync word, as mentioned before, indicates the start of the actual data in OTA packets. The sync word, 0x5555904E, was discovered through RF Command structures, which will be covered later in the paper. Through checking XREFs on the sync word in the structures, it was found that the sync word could also have been 0x55557A0E. Although the team did not look further into how the sync word is chosen, the knowledge of the potential sync word is useful. In the following paragraphs, the team explains how they used TI's SmartRF studio to gain further insight into the sync word's connection to replay attacks.

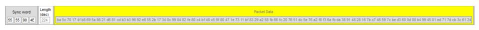
Given that the team uncovered the sync word, we wanted to answer the following question: can we locate the beginning of the packet's data if we have the correct sync word? The team learned that even if the incorrect sync word was inputted that SmartRF studio would still reveal the data, as is seen in the figures below:


Packet Count: 100 ☐ Infinite
Length Config: Variable

Preamble	Sync word	Length (dec)	Packet Data
01010101...	00 00 00 00	46	ba 5c 70 18 4f b8 69 5a 98 21 7d 74 3a 6a 51 ed f1 18 0e 4f 7f 20 03 e5 10 7a 23 7b f8 2d 2f ac fd 57 9e 7a 31 76 bb e1 d4 60 c0 d3 30 db

Preamble Count: 2 Bytes Sync Word Length: 32 Bits ☐ Address
Preamble Mode: Send 0 as the first preamble bit

Fig. 18. Packet Data is Revealed with the incorrect sync word



Sync word	Length (dec)	Packet Data
55 55 90 4E	224	ba 5c 70 17 4f b8 69 5a 98 21 d6 81 cd b3 b3 96 92 e8 55 2b 17 34 0c 99 84 82 fe 80 c4 bf 48 c5 9f 80 47 1e 73 11 bf 83 29 a2 58 fb 66 fc 20 76 51 dc 5e 76 a2 f8 f3 6a fb da 38 91 48 28 16 7b c7 46 59 7c be d3 60 0d 08 b4 99 45 01 ed 71 7d cb 3c 61 24

Fig. 19. Packet Data is Revealed with the correct sync word

However, the team was able to modify the packet data and directly replay the message to the Wyze camera, given the correct sync word. The team made this discovery by arbitrarily modifying bytes starting at the end of the packet data and replaying it to the Wyze camera's application; the application would change from open or close or vice versa. The team also learned that the most significant 46 bytes could not be modified or the message would become invalidated and not be received by the camera. The figure below depicts the most significant bytes of the packet (not highlighted) that could not be modified to still have a valid message to replay:

Recently the team has discovered that there are two possible values for the sync word, 0x5555904E as it is now or 0x55557A0E. This is decided based on the input to the function FUN_00012188, which sets up some values in the data structure mentioned before that contain the sync word. Not much effort has gone into finding how the sync word is decided between the two. However, it is useful information to know the two possible values of the sync word.

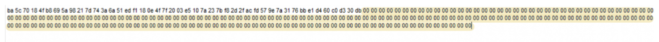

```
ba 5c 70 18 4f b8 69 5a 98 21 7d 74 3a 6a 51 ed f1 18 0e 4f 7f 20 03 e5 10 7a 23 7b f8 2d 2f ac fd 57 9e 7a 31 76 bb e1 d4 60 c0 d3 30 db 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Fig. 20. Packet Data is Revealed with the correct sync word

By learning that all the bytes, except the most significant 46 bytes, can be modified, the team learned that the packets are not whitened or encrypted since those types of messages cannot be modified and replayed because modifying them would invalidate the message [21].

VI. REVERSE ENGINEERING THE RF PROTOCOL

A. Ghidra

Ghidra is the program used by the team to reverse engineer the Wyze Camera's binary files. The team used a TI debugger to dump all of the memory of the Wyze camera into binary files while it was running, obtaining the firmware on the camera as well as other information, such as the SRAM. Those binary files are loaded into Ghidra, which disassembles the binary files into their corresponding assembly instructions and displays the byte memory for memory values that don't correspond to assembly instructions. A useful feature of Ghidra is that it can further decompile the assembly instructions into source code, displayed as code similar to the C programming language. However, the raw decompiled source code may still be hard to read as Ghidra doesn't have knowledge of different memory regions, structs, and registers leading to code that just has a bunch of references to different parts of the memory.

A helpful tool used with Ghidra is the System View Description (SVD) loader. SVD files contain information like the memory map, memory address names, memory registers locations. The SVD file can be loaded into Ghidra with the SVD Loader [22], and Ghidra will automatically create names in the memory map and replace constant memory references in the decompiled source code with variables and function calls.

B. Texas Instruments CC13x0, CC26x0 Software Development Kit (SDK)

To begin reverse engineering the disassembled code in Ghidra, the team studied the CC13x0 and CC26x0 SDK by Texas Instruments, which was introduced in Section II.

During our research, several files in the SDK were investigated. One file that was focused on was the rfPacketRx.c example file. Not only does this file include the main thread containing the setup of the RF protocol, it also demonstrates the use of a callback() function. This function is believed to be the primary handler of packet intake through the data queue.

C. RFC_DBELL

The RFC_DBELL starting at memory location 0x40041000 is crucial for narrowing down functions related to the RF protocol [10]. The RFC_DBELL is the primary means of communication between the system CPU and the radio CPU. It contains a set of dedicated registers and a set of interrupts for both the radio and system CPU [10]. The system and radio use the RFC_DBELL registers and interrupts to send information and commands to each other, respectively. As such, all functions that reference a memory location within the RFC_DBELL are likely to be related to the RF protocol and are a high priority for reverse engineering. We can find these functions through their cross references thanks to the SVD loader, which labeled this memory region while still keeping the Ghidra-created cross-references [22].

The function 00008bc0_radio_one, hereafter referred to as Radio1, is one of the functions that interacts with the interrupts stored in the RFC_DBELL section of memory. Specifically, it modifies the values of the RFCPEIEN/RFCPEIFG, and RFHWIEN/RFHWIFG interrupt register pairs in the RFC_DBELL. All interrupt registers in RFC_DBELL are 32 bits. For the pairs that Radio1 interacts with, each bit represents a different interrupt that is only activated when the corresponding bit is set to 1 in both registers. The RFCPEIEN/RFCPEIFG pair of registers contains Command and Packet Engine Generated Interrupts, and the RFHWIEN/RFHWIFG pair contains interrupts from RF Hardware Modules [10]. Radio1 overall appears to be more of a setup function, as it just modifies certain values memory and the interrupt registers, and it always calls the 0000cfdc_radio_two and 0000ee60_radio_three functions before it concludes.

The 0000cfdc_radio_two function, hereafter referred to as Radio2, also interacts with the interrupts in the RFC_DBELL. Like Radio1, Radio2 modifies the values in the RFCPEIEN/RFCPEIFG and RFHWIEN/RFHWIFG interrupt register pairs. However, Radio2 also calls the 00013b34_RFCDoorbellSendTo function, which sets values in the CMDR interrupt register. The CMDR interrupt register is what is used to pass commands from the system CPU to the radio CPU [10]. This clearly marks Radio2 as a transmission function that is meant to send information and commands from the system CPU to the radio CPU.

Another function found through references to the RFC_DBELL is the RFCRTrimRead() function. This function is in the TI SDK within the rfc.c file. This function references an important struct called rfc_CMD_PROP_RADIO_DIV_SETUP_t. This is one of several important structs containing information relating to the setup of the radio. More on these structs is covered in RF Command Structures (VI.G).

D. dongle_app

The dongle app is one of the program running on the Wyze camera and its binary files were obtained with the Wyze firmware. The dongle app contains multiple areas of importance in the reverse engineering area. Specifically, functions were found that were crucial for error handling. One of the found functions was msgsnd. This was found to be important as it is responsible for returning several important values, such as message_queue_id, message_pointer, message_size, and message_flag, all of which are utilized in various ways across the dongle_app.

Another found function was msg_success_checker, which checks whether a message was sent successfully or not. The function would return a value of 0 if the message was sent successfully and return a value of less than 0 if it was unsuccessful. There is a loop that makes three attempts to get a successful message, and if not, it is assumed to be unsuccessful. This function is used in many places where error messages occur, such as when verifying camera info, setting the camera to play an audio file, or updating dongle information. The 0 or less than 0 return value allows for easy programming of error messages. This function is crucial for designing error message protocols and programming error messages, as seen in the add_to_msg_queue function.

The add_to_msg_queue function adds messages to a queue and utilizes msg_success_checker to determine whether a message was successfully added to the queue. If it is unsuccessful, the function sends an error message containing the error number. This process depicts a typical example of how the msg_success_checker is used across multiple functions.

E. Important Data Structures

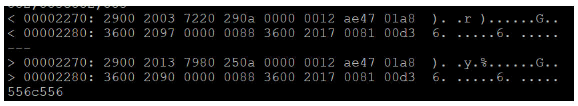
A set of data structures was discovered that starts at memory location 20003168. Each data structure starts at an offset with a multiple of 0x30 from 20003168. In the current snapshot of the SRAM, there is a data structure in the memory locations 20003168 and 20003198. The team believes these data structures are important as they are referenced frequently throughout the firmware, including in the Radio1 and Radio2 functions discussed in the RFC_DBELL (VI.C) section. Moreover, in the two data structures present in the current snapshot of the SRAM, each contains a reference to either rfc_CMD_PROP_TX_ADV_s_20002330 or rfc_CMD_PROP_RX_ADV_s_20002378, which are important structures in the memory that are covered more in RF Command Structures (VI.G).

F. Advanced Packets

It was determined that the Wyze system uses advanced packets for communication. This was initially found using the command number in packets, which, as described above in section IV, informs the receiving device of what action to take. A simple scalar search was conducted in the binary for the command numbers for transmitting and receiving standard packets(0x3801 and 0x3802), and for transmitting and receiving advanced packets(0x3803 and 0x3804). These command numbers were taken from the CC1310 MCU SDK and are also in the technical manual for the MCU. The search results were that only the command numbers for advanced packets were found in the binary file. This led the team to believe that the Wyze system uses advanced packets for wireless communication. This was further verified when the radio setup structures were found in the binary, and only structures corresponding to the advanced packets were found. Advanced packets have the option to repeat the preamble and can have an arbitrary amount of memory allocated for the payload.

Additionally, while conducting this search, the team found what seems to be an important pointer in a function for receiving advanced packets. The pointer points to the memory location 0x200022c0 in the SRAM. In the function, the value stored in this location is compared to several values representing different command numbers. This comparison is used to determine the subsequent actions of the microcontroller. This could mean that the command number is being stored at this location in the SRAM.

This location has also shown up in another area of the team's research. Using J-link's memory dump feature, snapshots were taken of the contents of the SRAM with the contact sensor pushed on and off, and a "diff" was performed of those two snapshots. This exact memory location showed up in the output of the "diff", as shown in Figure 20 below. The lines of interest are the lines beginning with 2270. This represents the memory location 0x200022c0 with an offset of 0x20000050. The contents of what is being stored in this memory are not yet understood but should be a focus of future research.



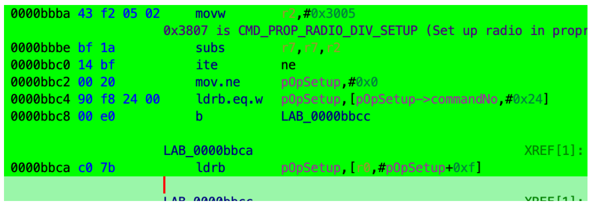
```
< 00002270: 2900 2003 7220 290a 0000 0012 ae47 01a8  ). .r )......G..
< 00002280: 3600 2097 0000 0088 3600 2017 0081 00d3  6. .....6. .....
---
> 00002270: 2900 2013 7980 250a 0000 0012 ae47 01a8  ). .y.%......G..
> 00002280: 3600 2090 0000 0088 3600 2017 0081 00d3  6. .....6. .....
556c556
```

Fig. 21. Contents in location 0x200022c0

G. RF Command Structures

As mentioned in the Packet Contents (V.A.) subsection as well as the RFC_DBELL (VI.C) subsection, there are important RF structures in the code that are referenced by the radio to initialize, transmit, and receive packets. More specifically, rfc_CMD_PROP_RADIO_DIV_SETUP_t, rfc_CMD_PROP_RX_ADV_t, and rfc_CMD_PROP_TX_ADV_t are radio structures referenced in the code and contain important information relating to the OTA protocol. [10] This information will be used for demodulating and interpreting the packets received from the system. All the data in these structs is defined in the appendix.

These structs, and a few other radio initialization structs, were found in multiple binary files in a contiguous memory region in the SRAM. rfc_CMD_PROP_RADIO_DIV_SETUP_t was found using a combination of static and dynamic analysis on the contact sensor's firmware. As mentioned in the RFC_DBELL (VI.C) subsection, it is referenced in the RFCRTrimRead() function. Dynamic analysis allows one to confirm which branch a system takes by executing code and monitoring register values, for example. JLink is the tool we use to do this. Using the JLink, we set a breakpoint on this function and stepped through to find the address of the reference. Stepping through this function confirmed the use of proprietary radio commands, and also it provided a possible pointer to the rfc_CMD_PROP_RADIO_DIV_SETUP_t struct.

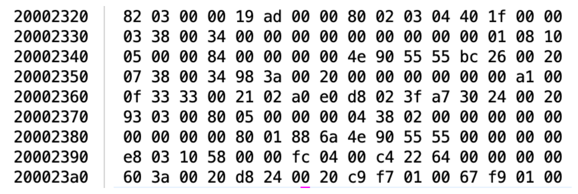


```
0000bbba 43 f2 05 02     movw        r2,#0x3005
                 0x3807 is CMD_PROP_RADIO_DIV_SETUP (Set up radio in propr
0000bbbe bf 1a           subs        r7,r7,r2
0000bbc0 14 bf           ite         ne
0000bbc2 00 20           mov.ne      pOpSetup,#0x0
0000bbc4 90 f8 24 00     ldrb.eq.w   pOpSetup,[pOpSetup->commandNo,#0x24]
0000bbc8 00 e0           b           LAB_0000bbcc

                 LAB_0000bbca                                    XREF[1]:
0000bbca c0 7b           ldrb        pOpSetup,[r0,#pOpSetup+0xf]
```

Fig. 22. Assembly of RFCRTrimRead() showing R0 referencing the struct

At address 0x0000bbca in figure 21 we can see register 0 is being used as the base address to reference rfc_CMD_PROP_RADIO_DIV_SETUP_t. The value of R0 is what we were looking for when dynamically executing the function. We pulled the value after executing and went to that memory location.



```
20002320 | 82 03 00 00 19 ad 00 00 80 02 03 04 40 1f 00 00
20002330 | 03 38 00 34 00 00 00 00 00 00 00 00 00 01 08 10
20002340 | 05 00 00 84 00 00 00 00 4e 90 55 55 bc 26 00 20
20002350 | 07 38 00 34 98 3a 00 20 00 00 00 00 00 00 a1 00
20002360 | 0f 33 33 00 21 02 a0 e0 d8 02 3f a7 30 24 00 20
20002370 | 93 03 00 80 05 00 00 00 04 38 02 00 00 00 00 00
20002380 | 00 00 00 00 80 01 88 6a 4e 90 55 55 00 00 00 00
20002390 | e8 03 10 58 00 00 fc 04 00 c4 22 64 00 00 00 00
200023a0 | 60 3a 00 20 d8 24 00 20 c9 f7 01 00 67 f9 01 00
```

Fig. 23. Raw bytes at SRAM location of structs

Address 20002350 was the address pulled from R0. As shown in figure 22, the first bytes here show the value 0x3807. This is the command number of rfc_CMD_PROP_RADIO_DIV_SETUP_t. When we converted the bytes to the correct data type, all of the fields aligned with the information about the struct definition provided in the TI SDK.

The RX_ADV and TX_ADV structs were found using a scalar search of their command numbers in the dongle binary. They were referenced directly in the code, so Ghidra could lead us back to the structure in the SRAM, as compared to the rfc_CMD_PROP_RADIO_DIV_SETUP_t structure, which we needed more help with finding via dynamic analysis.

When we refer back to the bytes in figure 22, we can see at address 20002330 there seems to be another command number. This prompted up to see if we could locate any additional command numbers. In this one contiguous region, the RF command structs of PROP_RADIO_DIV_SETUP, RX_ADV, TX_ADV, CS, NOP, and FS were found, and we were able to set these data types to their correct values. We confirmed that these data types were in the dongle and the contact sensor binary with matching values.

These structures solve many of the issues we had relating to the OTA protocol. They provide a great deal of information regarding how packets are created, transmitted, and received with important values like the sync word, the preamble, the baud rate, header information, the whitening mode, and more. Please refer to the appendix for the struct definitions and initialization values as found in the SRAM of the dongle. When referring to the values, if there is a mode defined, you can find the mode definition in the struct definition in rf_prop_cmd.h in the TI SDK. The SDK provides proprietary radio struct definitions in this file and defines the use for many of the values. Using this information will now be a primary focus of future work as we continue to demodulate and interpret data received from the system.

In the CC1310 Technical Reference Manual, there are tables that define the command structures by specifying their byte and bit fields while providing meaning for the values the fields can possess. Cross-referencing the struct values from a memory dump from the system with the manual has revealed the following information specifically regarding the TX_ADV struct. The struct header for TX_ADV is 16 bits in length, and consists of the first bytes of the buffer pointed to be pPkt. The remaining bytes in the buffer (presumably the payload) are transmitted byte-by-byte after the header is transmitted [10]. Going to the memory address pointed to by pPkt revealed the header to be 0x0807 (starting from the least significant bit). Work from this point will attempt to determine the plain-language meaning of the header (i.e., what does assigning this value to indicate about the system) and be conducted by tracing the references in Ghidra.

A similar process was used to deduce information about the RX_ADV structs. A conclusion was made that the RX_ADV also included a 16-bit header. Additionally, in the header, 11 bits were found to belong to the length field starting at the least significant bit of the header. Furthermore, it was found that no address is used in the RX_ADV command, which is to be expected. This is supported by the fact that pAddr, the pointer to the location of the address is null and ignored. However, it was noted that another pointer pQueue references many data structs that appear promising in unraveling more about the workings of the camera. Work from this point will focus on using the data structs referenced by pQueue and will be conducted through tracing in Ghidra.

H. RF Data Queue

The CC1310 uses a data queue to maintain packets that are transferred over the air. Data queues are used during the transfer of packets from the RF core to the main CPU, and vice versa [23]. In the radio setup structures covered in RF Command Structures (VI.G), there is a pointer to the data queue labeled pQueue. This pointer leads to a Data Entry Queue structure, which contains a pointer to the current entry on the data queue that is being processed and a pointer to the last entry added to the data queue [10]. The last entry pointer is NULL, but the current entry pointer does point at a data entry, meaning no entries can be added to the data queue [10]. The current entry in the data queue is a General Data Entry Structure judging by the value of its type field [10]. The pointer in the data entry for the next entry in the queue points at itself, implying that the data queue is circular and has only one data entry in it. Also important is that the total available storage space for packets is (temp) as indicated by the length field for RX entries, and the received packet starts from byte eight of the entry [10].

XREFs to the packet led the team to the 00002520_radio_something function. Through analysis of how the function interacted with different parts of the packet, the team suspected that the Wyze camera is set in the IEEE 802.15.4g mode of the Proprietary Radio mode. This was then confirmed by checking that various fields in the radio command structures were set to the necessary values for the radio to be in IEEE 802.15.4g mode [10]

I. Advanced Packet Format

By cross-referencing the technical reference manual with information gained from the RF command structures in Ghidra, we were able to make some important conclusions about the format of packets transmitted and received by radio.

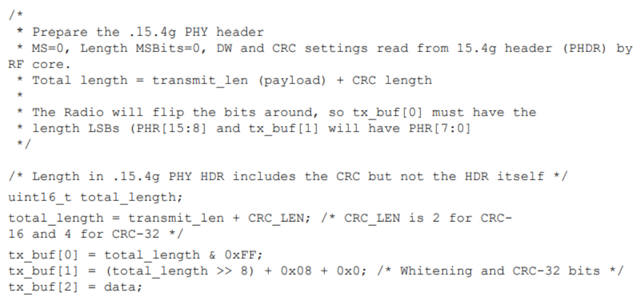
We determined that the radio was in IEEE 802.15.4g mode by checking various fields in the radio structs. Among those was the formatConf.whitenMode field in the rfc_CMD_PROP_RADIO_DIV_SETUP_t struct, which had a value of 7. The technical manual shows that the formatConf.whitenMode field having a value of 7 indicates that many of the header bits have very specific meanings [10].

As mentioned at the end of RFC Command Structures (VI.G), the pointer pPkt from the rfc_CMD_PROP_TX_ADV_t struct led us to a region in memory where the packet to be transmitted was located. Also, in RF Data Queue (VI.H), we found a packet that had been received. The first two bytes of the packets comprised the packet header. When the IEEE 802.15.4g format is being followed, the value of the first 11 bits of the header indicates the total length of the packet (consisting of payload length plus CRC length) [10]. Then, the formatConf.whitenMode field having a value of 7, indicates the meaning of many of the rest of the bits in the header [10]. Here are the known meanings of header bits, with the bits counted from the least significant bit being 0:

- When the 11th bit of the header is 1, whitening is enabled (otherwise no whitening is assumed) [10].

- When the 12th bit of the header is 1, a 16-bit CRC is assumed (otherwise a 32-bit CRC is assumed) [10].
- When the 15th bit of the header is 1, it is assumed that the frame contains a header and no payload or CRC [10].

An example of setting the header according to these specifications can be seen in the figure below:

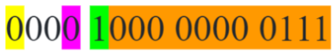


```
/*
 * Prepare the .15.4g PHY header
 * MS=0, Length MSBits=0, DW and CRC settings read from 15.4g header (PHDR) by
RF core.
 * Total length = transmit_len (payload) + CRC length
 *
 * The Radio will flip the bits around, so tx_buf[0] must have the
 * length LSBs (PHR[15:8] and tx_buf[1] will have PHR[7:0]
 */

/* Length in .15.4g PHY HDR includes the CRC but not the HDR itself */
uint16_t total_length;
total_length = transmit_len + CRC_LEN; /* CRC_LEN is 2 for CRC-
16 and 4 for CRC-32 */
tx_buf[0] = total_length & 0xFF;
tx_buf[1] = (total_length >> 8) + 0x08 + 0x0; /* Whitening and CRC-32 bits */
tx_buf[2] = data;
```

Fig. 24. Setting transmit packet header for 32-bit CRC

We were able to confirm that this format was being followed by analyzing the header found in Ghidra through the transmit command, which has a value of 0x0807.



0000 1000 0000 0111

Fig. 25. Binary representation of transmit packet header found in Ghidra with important bits highlighted

As seen in the figure above, the 12th bit (highlighted in purple) is 0, indicating a 32-bit CRC. The 11th bit (highlighted in green) is 1, indicating whitening is enabled. This is expected based on the value of formatConf.whitenMode explained at the beginning of this section. The 11 bits of the header (highlighted in orange), represent a value of 7 in decimal. This aligns with its meaning of payload length plus CRC length. 32-bit CRC indicates that it contributes 4 bytes to the total length. We know the payload is 3 bytes long using values from the rfc_CMD_PROP_TX_ADV_t struct: Subtracting numHrdbits (16 in decimal, which equates to 2 bytes) from pktLen (5 bytes) to get 3 bytes for the payload. Note that the field pktLen in the struct carries a different meaning from the "total length" information found in the header. Finally, the 15th bit (highlighted in yellow) is 0, indicating the frame includes a payload and CRC as we expect.

VII. CONCLUSIONS

In working towards our overall goal of decoding the Wyze OTA protocol to enable the setup of a fuzzing testbed, we made significant progress this semester in fleshing out fields from the RF command structures and determining the fields in the radio packet headers. Moving forward, we need to be able to extract the application data contained in the payload of the RF packets we have been analyzing in Ghidra. This step would aid in the creation of a spoofer/message encoder, which would create artificial packets. We would then be able to use these artificial packets to probe the system for vulnerabilities as we develop the fuzzing testbed.

References

[1] S. Sinha, "State of iot 2021: Number of connected iot devices growing 9% to 12.3 billion globally, cellular iot now surpassing 2 billion." https://iot-analytics.com/number-connected-iot-devices/.

[2] "Iot security needed now more than ever." https://cisomag.eccouncil.org/iot-security-needed-now-more-than-ever/.

[3] "why-do-iot-companies-keep-building-devices-with-huge-security-flaws." https://hbr.org/2017/04/why-do-iot-companies-keep-building-devices-with-huge-security-flaws.

[4]

[5] B. Lovejoy, "Wyze camera security breach: 2.4m users have personal data exposed." https://9to5mac.com/2019/12/30/wyze-camera-security/, Dec 2019.

[6] "Wyze sensor limit." https://support.wyze.com/hc/en-us/articles/360030677072-Is-there-a-limit-to-the-number-of-sensors-I-can-connect-to-a-Bridge/.

[7] B. Dipert, "Teardown: High-quality and inexpensive security camera." https://www.edn.com/teardown-high-quality-and-inexpensive-security-camera/2/.

[8] "8-bit cost-effective enhanced usb microcontroller ch554." http://wch-ic.com/products/CH554.html.

[9] "Rf core — simplelinktm cc13x2 / cc26x2 sdk proprietary rf user's guide 2.80.0 documentation."

[10] T. Instruments, "Cc13x0, cc26x0 simplelink™ wireless mcu technical reference manual," *Texas Instruments*, Feb 2015.

[11] "Ghidra software." https://ghidra-sre.org/.

[12] "How Does NI USRP Hardware Work?."

[13] A. Communications, "Signed firmware, secure boot, and security of private keys," July 2020.

[14] stacksmashing, "Iot security: Backdooring a smart camera by creating a malicious firmware upgrade." https://www.youtube.com/watch?v=hV8W4o-Mu2ot=612s.

[15] hclxing, "Reverse engineering wyzesense bridge protocol (part ii)," May 2019.

[16] "Ghidra."

[17] HclX, "Hclx/wyzesensepy." https://github.com/HclX/WyzeSensePyreadme.

[18] "Gaussian Frequency Shift Keying - an overview | ScienceDirect Topics."

[19] J. Pohl and A. Noack, "Universal radio hacker: A suite for analyzing and attacking stateful wireless protocols," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, (Baltimore, MD), USENIX Association, 2018.

[20] "Cyclic redundancy check." https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[21] "What is anti-replay protocol and how does it work?."

[22] leveldown security, "Svd loader for ghidra 2019." https://leveldown.de/blog/svd-loader/, Sep 2019.

[23] T. Instruments, "Working with data queues," 2017.

APPENDIX

```
rfc_CMD_PROP_RADIO_DIV_SETUP_t
RF_cmdPropRadioDivSetup =
{
    .commandNo = 0x3807,
    .status = 0x0000,
    .pNextOp = 0, // INSERT APPLICABLE POINTER:
        (uint8_t*)&xxx
    .startTime = 0x00000000,
    .startTrigger.triggerType = 0x0,
    .startTrigger.bEnaCmd = 0x0,
    .startTrigger.triggerNo = 0x0,
    .startTrigger.pastTrig = 0x0,
    .condition.rule = 0x0,
    .condition.nSkip = 0x0,
    .modulation.modType = 0x1,
    .modulation.deviation = 0x14,
    .symbolRate.preScale = 0xF,
    .symbolRate.rateWord = 0x3333,
    .symbolRate.decimMode = 0x0,
    .rxBw = 0x21,
    .preamConf.nPreamBytes = 0x2,
    .preamConf.preamMode = 0x0,
    .formatConf.nSwBits = 0x20,
    .formatConf.bBitReversal = 0x0,
    .formatConf.bMsbFirst = 0x1,
    .formatConf.fecMode = 0x0,
    .formatConf.whitenMode = 0x7,
    .config.frontEndMode = 0x0,
    .config.biasMode = 0x1,
    .config.analogCfgMode = 0x2D,
    .config.bNoFsPowerUp = 0x0,
    .txPower = 0xA73F,
    .pRegOverride = pOverrides,
    .centerFreq = 0x0393,
    .intFreq = 0x8000,
    .loDivider = 0x05
};

rfc_CMD_PROP_RX_ADV_t
RF_cmdPropRxAdv =
{
    .commandNo = 0x3804,
    .status = 0x0000,
    .pNextOp = 0, // INSERT APPLICABLE
    POINTER: (uint8_t*)&xxx
    .startTime = 0x00000000,
    .startTrigger.triggerType = 0x0,
    .startTrigger.bEnaCmd = 0x0,
    .startTrigger.triggerNo = 0x0,
    .startTrigger.pastTrig = 0x1,
    .condition.rule = 0x1,
    .condition.nSkip = 0x0,
    .pktConf.bFsOff = 0x0,
    .pktConf.bRepeatOk = 0x0,
    .pktConf.bRepeatNok = 0x0,
    .pktConf.bUseCrc = 0x1,
    .pktConf.bCrcIncSw = 0x0,
    .pktConf.bCrcIncHdr = 0x0,
    .pktConf.endType = 0x0,
    .pktConf.filterOp = 0x1,
    .rxConf.bAutoFlushIgnored = 0x0,
    .rxConf.bAutoFlushCrcErr = 0x1,
    .rxConf.bIncludeHdr = 0x1,
    .rxConf.bIncludeCrc = 0x0,
    .rxConf.bAppendRssi = 0x1,
    .rxConf.bAppendTimestamp = 0x1,
    .rxConf.bAppendStatus = 0x0,
    .syncWord0 = 0x5555904e,
    .syncWord1 = 0x00000000,
    .maxPktLen = 0x03E8,
    .hdrConf.numHdrBits = 0x10,
    .hdrConf.lenPos = 0x0,
    .hdrConf.numLenBits = 0xB,
    .addrConf.addrType = 0x0,
    .addrConf.addrSize = 0x0,
    .addrConf.addrPos = 0x0,
    .addrConf.numAddr = 0x0,
    .lenOffset = 0xFC,
    .endTrigger.triggerType = 0x4,
    .endTrigger.bEnaCmd = 0x0,
    .endTrigger.triggerNo = 0x0,
    .endTrigger.pastTrig = 0x0,
    .endTime = 1680000000,   //7 minutes
    .pAddr = 0,
    .pQueue = 0, // INSERT APPLICABLE
    POINTER: (dataQueue_t*)&xxx
    .pOutput = 0, // INSERT APPLICABLE
    POINTER: (uint8_t*)&xxx
};

rfc_CMD_PROP_TX_ADV_t RF_cmdPropTxAdv =
{
    .commandNo = 0x3803,
    .status = 0x0000,
    .pNextOp = 0, // INSERT APPLICABLE
    POINTER: (uint8_t*)&xxx
    .startTime = 0x00000000,
    .startTrigger.triggerType = 0x0,
    .startTrigger.bEnaCmd = 0x0,
    .startTrigger.triggerNo = 0x0,
    .startTrigger.pastTrig = 0x0,
    .condition.rule = 0x1,
    .condition.nSkip = 0x0,
    .pktConf.bFsOff = 0x0,
    .pktConf.bUseCrc = 0x1,
    .pktConf.bCrcIncSw = 0x0,
    .pktConf.bCrcIncHdr = 0x0,
    .numHdrBits = 0x10,
    .pktLen = 0x5,
    .startConf.bExtTxTrig = 0x0,
    .startConf.inputMode = 0x0,
    .startConf.source = 0x0,
    .preTrigger.triggerType = 0x4,
    .preTrigger.bEnaCmd = 0x0,
    .preTrigger.triggerNo = 0x0,
    .preTrigger.pastTrig = 0x1,
    .preTime = 0,
    .syncWord = 0x5555904e,
    .pPkt = 0, // INSERT APPLICABLE
    POINTER: (uint8_t*)&xxx
};
```