

Wyze Camera Report

Antonia Nunley, Chris Reid, David Wolfson, Joseph Lucas, Makiyah Dee, Kyser Montalvo
Embedded System Cyber Security VIP
Georgia Institute of Technology
Atlanta, GA

Abstract—This is the final paper for the Wyze research team at the Georgia Institute of Technology’s vertically integrated project on embedded systems and cyber security. This document gives a brief introduction of the camera and its characteristics, followed by information about known vulnerabilities, and the firmware of the camera, dongle, and contact sensor. The goal of the research this semester is to look into the RF protocol used by the Wyze camera and determine characteristics about the transfer of packets.

I. INTRODUCTION

The Wyze Camera V2 is an Internet of Things (IoT) Device. One of its main functions is serving as a security camera, providing surveillance of the locations the camera and contact sensors are placed. Mobile applications allow users to maintain the status of the surroundings their devices are in. Their smartphone application (Wyze) is available on and compatible with Android, iOS, and Google Assistant devices.

Recently, there has been a major increase in the use of wireless cameras, cloud access, and mobile applications [1]. This has resulted in a need for the security of these devices to be enhanced. As the number of these devices available to the public increases, so does the number of individuals wanting to conduct malicious attacks [2]. Enhancing security on IoT devices is very expensive for companies that produce them, and some decide to continue producing the original product without making the necessary changes to the devices’ security[3].

II. FUNCTIONAL DESCRIPTIONS

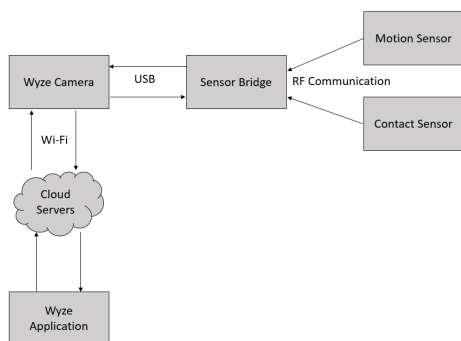


Fig. 1. Overview of System Communications

The Wyze IP Camera V2 setup, as shown in Figure 1, consists of mainly four parts: the motion and contacts sen-

sors, the sensor bridge, the camera and the Wyze application. The sensors communicate with the sensor bridge through radio frequency communication (RF). The sensor bridge sends that data to the camera via a USB connection. The camera communicates through Wi-Fi with the Wyze cloud servers, which sends data to the Wyze application. The sensor bridge can communicate with up to 100 sensors[4].

A. Mainboard

The main Wyze Camera has 3 printed circuit boards (PCB) sandwiched together. The main PCB with the SoC (System on a Chip) (Figure 2), the microSD board (Figure 3), and a sensor board (Figure 4) are all contained inside the main camera.

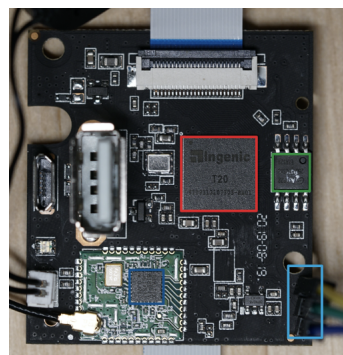


Fig. 2. PCB of the Main Board

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

TABLE I

COMPONENTS FOR MAIN BOARD PCB

The PCB shown in Figure 2 is one of the circuit boards inside the Wyze camera. The board runs on a T20 processor [5] that uses the MIPS (Million Instructions per Second) Instruction Set Architecture. This is the main board and it also contains flash memory, Wi-Fi, and serial access as shown in the figure.

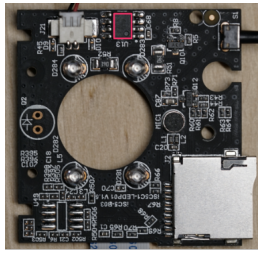


Fig. 3. microSD PCB for the Main Board with motor driver in red

The main camera has another PCB that provides an SD card slot as well as a motor driver to move the camera. This is shown in Figure 3.

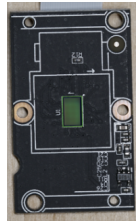


Fig. 4. sensor board with no lens

B. Sensor Bridge

The sensor bridge connects the camera’s sensors (motion and contact sensor) to the main camera. The sensor runs on the CC1310 micro controller and has an antenna for communicating wirelessly with the contact and motion sensors. The sensor bridge connects to the main camera via USB-A. The board also has a WCH CH554T chip on the back to control the USB communication.[6]

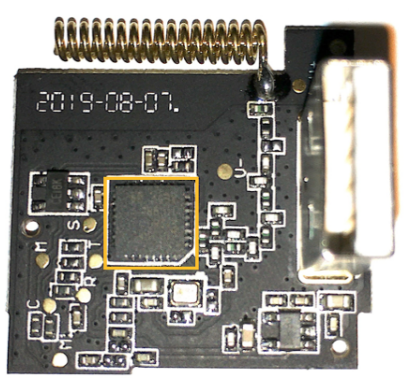


Fig. 5. Front of Sensor Bridge PCB with CC1310 highlighted in Orange

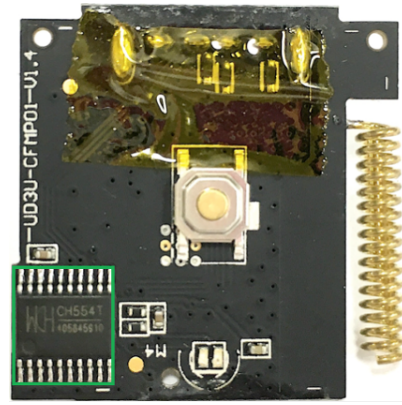


Fig. 6. Back of Sensor Bridge PCB with WCH CH554T chip highlighted in green

C. Motion Sensor and Contact Sensor

The contact and motion sensors communicate with the sensor bridge wirelessly. Both sensors are controlled by a CC1310 micro controller. The contact sensor has a magnetic switch, which tells the micro controller when contact is made and the motion sensor uses a PIR (passive infrared) motion sensor.

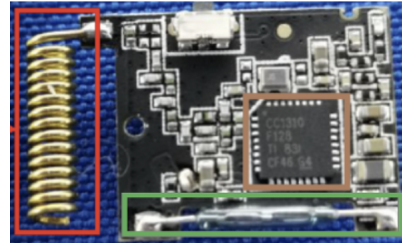


Fig. 7. Contact Sensor PCB

Label Color	Red	Brown	Green
Component	Antenna	CC1310	Magnetic Switch

TABLE II
COMPONENTS FOR THE CONTACT SENSOR

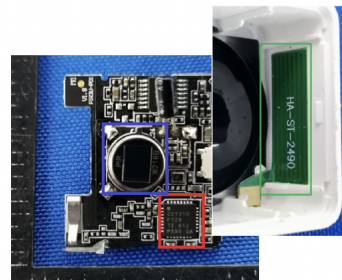


Fig. 8. Motion Sensor PCB

Label Color	Blue	Red	Green
Component	PIR Motion Sensor	CC1310	Antenna

TABLE III
COMPONENTS FOR THE MOTION SENSOR

D. CC1310 Micro controller

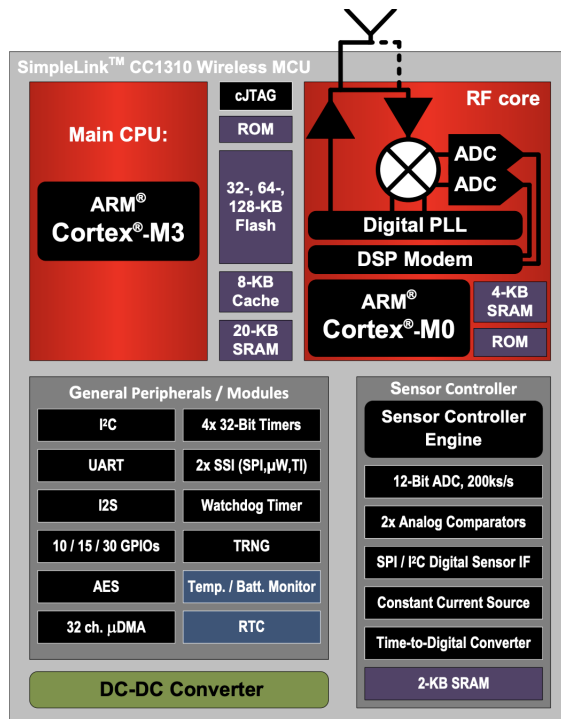


Fig. 9. CC1310 functional block diagram

The Wyze Camera uses a CC1310 TI Simplelink Wireless MCU as the main chip on the sensor bridge as well as the contact and motion sensors. The focus of our work this semester was to focus on this processor, specifically the RF core and its use. This is the radio peripheral of the CC1310 which can be programmed to handle multiple protocol standards.[7] We focused on the wireless communication protocol between the sensor bridge along with the contact and motion sensors. This processor uses a 20KB single-cycle on-chip SRAM with full retention in all power modes, except shutdown [8]. This type of SRAM is useful to share and store information between multiple parts of the chip.

Data from the SRAM can be transferred with direct memory access (DMA) [8]. Looking into a capture of the SRAM it is expected that there will be pieces of information such as packets and other volatile structures. These structures could be encrypted or encoded by the CC1310. Being able to understand the encryption protocols of these packets is one the goals of this semester. Other goals of the semester in reference to the CC1310 involve reverse engineering the firmware in Ghidra, a national security agency provided software, to define structures and variables useful to understand this protocol.

Not much is know about the RF Core of the CC1310; however, sections V and VI go further in depth about what was discovered about the RF Core from the team’s research.

III. EXISTING VULNERABILITIES

To date, there are several very significant vulnerabilities of the Wyze Camera system that can be exploited to leak sensitive information. These vulnerabilities are subject to several different types of attacks, but the most significant attack to the Wyze Camera is a replay attack. Additionally, the Wyze camera’s firmware is not signed, allowing the firmware to be modified.

A. Replay Attacks

Replay attacks are a very significant vulnerability that compromises the security of the Wyze Camera. Replay attacks are defined to be network-based attacks where the attacker delays, replays, or repeats data transmission between the user and the application [9]. Replay attacks explain that messages are not being authenticated well. A common technique for preventing replay attacks is to use a nonce (number used once) [10]. Since we can do replay attacks, we know that the Wyze Over-the-Air (OTA) protocol does not implement nonces.

Using captured packets from a contact and motion sensor, an USRP N210 was used to replay the recorded packets back to the dongle. The dongle successfully received the packets, as verified by the chosen input being displayed in the Wyze app. The first packet associated with an alert contains a 4-hex character value, that increments upwards 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 their incrementing when an event happens it is possible, they are some kind of sequence counter. Recorded packets are able to be sent in any order, as long as the event types alternate, and will be processed successfully by the dongle.

Expanding on the replay attack, URH was used to modulate packets so that arbitrary changes could be made. The settings used to successfully modulate packets can be seen in Figure 10 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

Fig. 10. Setting used to modulate packets

Using URH, arbitrary packets were able to be sent to the dongle. Using a recorded contact sensor open alert, nibbles and then bytes were zero’d out sequentially and then transmitted with a contact sensor close alert in between.

B. Unsigned Firmware

In embedded systems, authors of the firmware may sign that firmware. This is a cybersecurity technique used to prevent malicious or corrupted firmware to be flashed onto the

device. Signing a firmware entails generating a cryptographic hash value, signing (encrypting) it with the private key of a private/public key pair, and attaching it to the firmware[11]. The firmware on the devices of the Wyze system are not signed, allowing unauthorized firmware to be flashed onto the devices [12].

IV. SENSOR FIRMWARE

The sensors and sensor bridge communicate through radio frequency (RF) with OTA packets. The packets received from the sensors are decoded in the binary of the sensor bridge. Depending on the command or type of packet being sent, they can also be serial packets, which contain information about the camera and state of the sensors and other devices in the Wyze network. We were able to obtain log files for different communication scenarios between the devices. Looking through the RTSP (real time streaming protocol) log files, there are certain parts of the communication that directly matches with the packet information.

A. Packet Decomposition

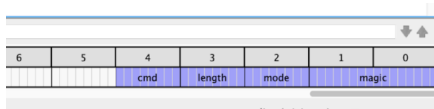


Fig. 11. Ghidra Serial Packet Struct Header

The serial packets which are sent between the sensors and the sensor bridge differ in length dependent on the contents and command type of the packet being sent. They will range between a length of 7 bytes to a length of 64 bytes. Packet interpretation from the sensor bridge begins by checking the two magic bytes and the command byte: The magic bytes are the first two indices of the package, the "mode" byte is the third byte of the packet. The length byte is the fourth, and following this is the command byte. The remainder of the packet length is the payload. The Wyze camera system uses variable length packets to communicate between the devices, however the first 5 sections of the payload are consistent and present in all packages. [13] Finding any information on the serial packet protocol, could help us in determining the RF protocol used, since this communication is between the camera and dongle.

- 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 length byte of the packet is interpreted as a decimal value (see Figure 12 below). And is located 3 bytes from the first magic byte (the 4th byte). The full length of the packet is this value + 4. It can be inferred that this + 4 accounts for the 4 bytes previous to the length index of the packet. Meaning, this byte actually

designates the number of remaining bytes in the packet length.

Knowing the length of each packet is necessary to differentiate between two consecutive packets. The team confirmed that the Wyze system communicates using packets with variable length payloads. Therefore, without the packet length, it would be very difficult to tell where one packet ended and another packet began, since the amount of bytes in each packet can vary.

```

/* 00 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. 12. Ghidra interpretation of packet length in decimal

- 2) Finding the sub_mac value of the camera
 - If the packet is not an ACK packet, the sub_mac value of the camera can be found in bytes [16:23] of the packet. Like the length of the packet, the value of these bytes is interpreted as decimal, and their ASCII values are the sub_mac value.
- 3) Simply acknowledging that the previous packet is received
 - This simple ACK packet is 7 bytes long.
 - The type byte (3rd byte) has only two options: 0x43 or 0x53. This difference is between synchronous (0x53) and asynchronous (0x43). When the type is 0x43 the response packet from the other device will immediately be sent, with any payload information the current command requests. If the type is 0x53, along with an ACK packet, the responding device will send packets corresponding to the command byte. [13]

B. JLinkExe

The JLinkExe is a program to interface with the segger J-link, which is a device that can connect to the JTAG of a processor. Once connected to the JTAG, it is possible to step through the instructions on the firmware of the processor. By doing this, we will be able to look through the Assembly Language of the processor and see which registers are being used, how they are being used (Branching, writing, and writing on registers), and which hexadecimal memory addresses are being occupied. Then we can take the hexadecimal memory address (R0 = 0x2000051C) and see what is stored at that address. In the case of this project, we are focusing on connecting the target device (CC1310F128)

via cJTAG. We first SSH into the system, then run JLinkExe command. While it is running, you will connect to the CC1310F128 and type (t) for the cJTAG. Once the device is selected and connected to, we will need to slow down the interface speed from it's default 4000 kHz to 100 kHz. We do this to slow down the processor so we can read the memory addresses.

```

J-Link>step
20000408: 70 47 BX LR
J-Link>step
2000032E: 01 BD POP (R0,PC)
J-Link>step
20000346: F0 6A LDR R0, [R6, #+0x2C]
J-Link>step
20000348: 00 28 CMP R0, #0
J-Link>step
2000034A: F9 D0 BEQ #-0x0E
J-Link>step
20000340: 20 00 MOVS R0, R4
J-Link>step
20000342: FF F7 F1 FF BL #-0x1E
J-Link>step
20000328: 80 B5 PUSH (R7,LR)
J-Link>step
2000032A: 00 F0 6D F8 BL #+0xDA
J-Link>step
20000408: 70 47 BX LR

```

Fig. 13. Stepping through the processor

```

J-Link>regs
PC = 2000032E, CycleCnt = 78D4E0F3
R0 = 2000051C, R1 = 2000059C, R2 = 20000574, R3 = FFFFFFFF
R4 = 2000051C, R5 = 2000053C, R6 = 2000053C, R7 = 50001000
R8 = 40051000, R9 = 40050000, R10 = 00000005, R11 = 40031000
R12 = 00000000
SP (R13) = 20000510, MSP = 20000510, PSP = 00000000, R14 (LR) = 2000032F
XPSR = 21000000: AFSR = nzCvq, EFSR = 01000000, IFSR = 000 (NoException)
CFBP = 00000000, CONTROL = 00, FAULTMASK = 00, BASEPRI = 00, PRIMASK = 00
FPU regs: FPU not enabled / not implemented on connected CPU.

```

Fig. 14. Registers being used in the processor

V. SIGNAL ANALYSIS OF THE RF PROTOCOL

This semester the focus is on reverse engineering Wyze's proprietary RF protocol between the contact sensors, motion sensors and the sensor bridge. The teams must first determine whether the OTA packets are encrypted and/or whitened. This can be done by analyzing Over-The-Air (OTA) captures of recordings between the sensor bridge and the sensors.

A. Packet Captures

A packet capture intercepts a data packet crossing a point in a data network, and once captured, the packet can then be stored and later analyzed. To capture packets going between the dongle and sensors, an Ettus N210 SDR with a standard vertical antenna was used. GNU Radio was used as the software to record the packets. The GNU Radio flowgraph used to record and the block diagram for data flow can be seen below:

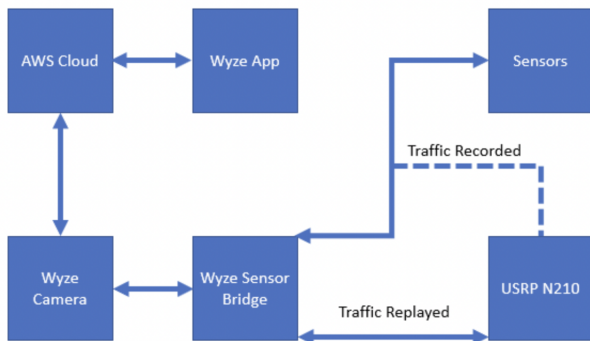


Fig. 15. Block Diagram showing where Packets and Logs were captured

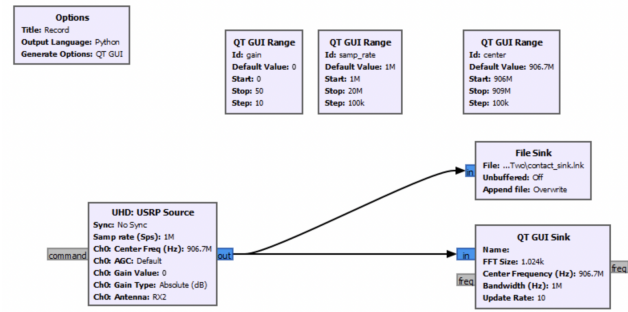


Fig. 16. GNU Radio flow graph used to record packets for playback

Serial logs were captured using a direct connection to the Wyze Camera, which outputs received packets as part of its debug information. Alternatively, serial logs can be directly captured using WyzeSensePy [14], a raspberry pi and a USB connection to the dongle, which implements the communication protocol between the dongle and camera.

B. RF Overview

Information from the contact and motion sensor is generated by their microcontrollers and modulated through Frequency Shift Keying (FSK). Universal Radio Hacker (URH) [15] an aid in analyzing OTA packets to analyze the protocol between the camera and the dongle.

C. Sensor Over-The-Air Packet

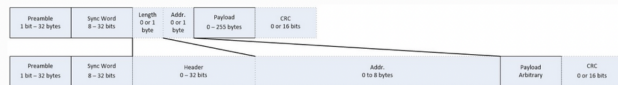
URH can be used to get a clearer look at the OTA packets of recordings. It depicts what is actually happening during the time between the motion and contact sensors of the camera before the packet is received. This will allow the team to gain a better understanding of the communication protocol.

D. Packet Contents

Not all data transmitted via over-the-air (OTA) packets is currently known. The OTA packet is composed of a proprietary protocol packet with a payload that contains the application level which may consist of the: MAC, Battery, Counter, and Event Type. We suspect OTA packets from sensor-to-dongle contain the MAC of the sensor which is used for identification. The sensors transmit how much battery they have left to the dongle via the sensor bridge. Additionally, there is a 16-bit sequence counter that increments each time there is an event (open/close, motion/no motion). This sequence counter resets to 0 every time the sensor is powered down. Finally, the type of event is transmitted (open/close, motion/no motion). Given that we know that the packets are largely the same, with variation for battery/MAC/counter/event type, it is very likely that the packets are encrypted, whitened, or both. The TI SDK has been a good source for understanding the physical structure of the packets. We have an idea that Wyze is using advanced TX/RX packets from TI. The diagram below provides more information:

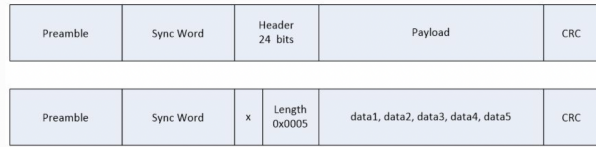
cmd_prop_rx vs cmd_prop_rx_adv

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. 17. RX information

E. Communication Protocol

When an event is detected, the sensor transmits a packet to the dongle. After the dongle has received the packet, the dongle replies with an “ACK” communication that indicates it received the event alert from the sensor. The dongle keeps state of sensors in memory. For a contact sensor, sending two “open” event alerts consecutively, will result in the second message being considered an error, and not another event. This state keeping seems 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. Recordings

The packets were transmitted through a frequency of 906.7MHz with a modulation type of FSK. Using URH the packets could 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, delete.

G. Motion Sensor

An analysis of the motion sensor captures was done using the URH tools along with an analysis of the log files using a parser. The goal of this was to gain more insight about the data being sent in the OTA packets.

H. OTA Protocol

The focus of our research was to answer whether the OTA packets were whitened and/or encrypted. With this information, one could be able to manipulate and spoof messages to the Wyze Camera. To begin uncovering the mystery of the OTA packets, the team reviewed the past teams’ work, which included an outline on how to reach the answer:

Future work

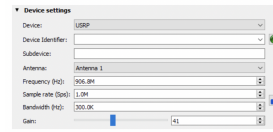
This replay attack can be expanded on to send arbitrary packets to the dongle. Some work has already been done in Universal Radio Hacker to support this.

Using URH’s “Generate” functionality, signals can be modulated and successfully received by the dongle.

The generation parameters are below:

```
Encoding: -
Carrier Frequency: 906.7MHz
Carrier Phase: 0°
Symbol Length: 300
Sample Rate: 3M
Modulation Type: FSK
Bits per Symbol: 1
Frequencies in Hz: -13K/12K
```

The transmission parameters are below:



URH supports fuzzing profiles. On the “Generator” tab, click the load button in the top left. This will load a fuzzing profile including the messages to modulate, modulation parameters, and pauses in between messages. This can be combined with the serial logs from the dongle to correlate parts of the over-the-air (OTA) packet with the payload. The included fuzzing profiles focused on trying to zero out entire nibbles/bytes and iterating through the entire OTA packet, expected to see a change in the serial output. However, no change was observed. Note that a fuzzed OTA packet will need to alternate with a legitimate packet of the opposite event type (open/close or motion/no motion) to change the dongle state so it will accept the fuzzed packet.

Fig. 18. Outline of Continuing Research in the OTA Protocol

Using URH, the team would take an old open replay attack that was successful and edit some of the bits; the edited replay attack was then uploaded to the diagram in GNUradio companion to have it tested to see if a valid open replay attack still occurred.

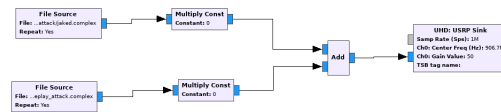


Fig. 19. Edited GNU Radio Diagram

Once the flow graph was executed and WyzeSensePy was running, the edited open and the successful close replay attacks were tested. The attacks were tested to see if they were valid by alternating between the open and close sliders by changing their value from 0 to 1. We expected to see results like the ones in the Figure 20 below:

```
0x5319 | b'000 timestamp: 178efcad855 unknown: a20 mac: 0000000000000001 signal: 1a batter
[2021-04-20 10:59:11][StateEvent: sensor_typeswitch, state=open, battery=95, signal=54]
0x5319 | b'000 timestamp: 178efcdae6 unknown: a20 mac: 0000000000000001 signal: 1a batter
[2021-04-20 10:59:12][StateEvent: sensor_typeswitch, state=close, battery=95, signal=54]
0x5319 | b'000 timestamp: 178efcadd1 unknown: a20 mac: 0000000000000001 signal: 1a batter
[2021-04-20 10:59:12][StateEvent: sensor_typeswitch, state=open, battery=95, signal=51]
0x5319 | b'000 timestamp: 178efca8ac unknown: a20 mac: 0000000000000001 signal: 1a batter
[2021-04-20 10:59:13][StateEvent: sensor_typeswitch, state=close, battery=95, signal=54]
0x5319 | b'000 timestamp: 178efca351 unknown: a20 mac: 0000000000000001 signal: 1a batter
[2021-04-20 10:59:14][StateEvent: sensor_typeswitch, state=open, battery=95, signal=51]
0x5319 | b'000 timestamp: 178efcae6ad unknown: a20 mac: 0000000000000001 signal: 1a batter
[2021-04-20 10:59:15][StateEvent: sensor_typeswitch, state=close, battery=95, signal=53]
```

Fig. 20. Results of Successful Open and Closed Replay Attacks

If the experiment was a success, meaning the edited open replay attack worked, then we would gain insight on whether or not the packets were whitened and/or encrypted. However, the results of the experiment were inconclusive because the edited open replay attack was not properly executed. There are multiple reasons why the edited message did not go through. The cyclic redundancy check (CRC) [16] could have been incorrect, or possibly the edited piece of the packet was indistinguishable and couldn’t be interpreted (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.

VI. REVERSE ENGINEERING THE RF PROTOCOL

A. Ghidra

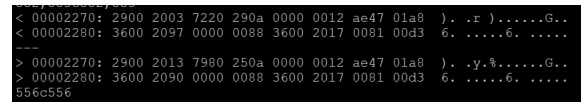
The primary tool used to reverse engineer the RF protocol is Ghidra. Ghidra is a software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission[17]. The firmware of the Wyze camera is loaded into Ghidra, and Ghidra disassembles the program allowing users to reverse engineer the program.

A helpful tool in the reverse engineering is the SVD loader. SVD stands for System View Description and is a file that contains information like the memory map and names of memory addresses. This file can be loaded into Ghidra with the SVD Loader and Ghidra will automatically create names in the memory map to make reverse engineering easier.

As described above in section IV, the packets used for communication contain a command number which describe to the receiving device what action is required of it. This command number was reverse engineered to determine what type of packets the Wyze system is using. A simple scalar search was conducted in the binary for the command number for transmitting and receiving standard packets (command number 0x3801 and 0x3802) and for transmitting and receiving advanced packets (0x3803 and 0x3804). These command numbers were taken from the SDK for the CC1310 MCU and also in the technical manual for the MCU. This technique was used because it was a simple and quick way to attempt to determine what kind of packets the Wyze system uses. The results of the search were that only the command numbers for the advanced packets were found in the binary file. This lead the team to believe that the Wyze system is using advanced packets for wireless communication. Advanced packets have the option to repeat the preamble and have an arbitrary amount of memory allocated for the payload.

Additionally, while conducting this search, the team found what seems to be an important SRAM pointer. The pointer was found in a function for receiving advanced packets. The pointer points to the location 0x200022c0 in the SRAM. In the function, the value stored in this location is compared to several hex values which represent 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, which could be a clue to the team for where to start looking for the packet structures in the SRAM.

This location has also shown up in another area of the team's research. A snapshot of the contents of the SRAM was taken with the contact sensor pushed on and off, and a "diff" was performed of those two files. This exact memory location showed up in the output of the "diff", as shown in Figure 21 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 2033 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

Another focus of this semester was to continue reverse engineering the decompiled C code to find functions relating to the RF protocol, labeling any new data structures, scalars, and other relevant methods to RF and the data queue. To do this, the TI CC1310 technical manual and SDK are used.

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

In order to begin reverse engineering the disassembled code in Ghidra, the team studied TI's CC13x0, CC26x0 SDK. The SDK allows the team to become familiar with the possible methods, structures, and constants that can be found in memory and their uses. Some methods available in the SDK have already been discovered in the Ghidra disassembly. The SDK also comes with example implementations of different protocols and procedures that can be implemented through the API provided with the SDK. Specifically, there are example implementations of RF protocol including packet transmission and receiving. These examples include a main thread, and setup functions to show exactly how the SDK could be used in different situations.

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

Another file of focus is RFCC26X2multiMode.c driver. Although this driver file would seem to only apply to CC26x2 MCU's, the team believes that the same code would also be used for the CC1310 MCU. This file contains many important functions pertaining to the RF core, but the function the team focused on is RF_init(). This function handles initializing the RF driver. The team believed this function could help understand where the data for the packets is coming from in the SRAM. There is a function labeled RF_init_maybe() in Ghidra, which is labeled by students of previous semesters.

While exploring these two functions, the team realized that the RF_init_maybe() function cannot be the RF_init() function from the SDK. One of the reasons for this conclusion is that the function in Ghidra has four parameters in the function header, while the function in the SDK has no parameters in the header. Additionally, the RF_init() function in the SDK is called by several other functions to initialize the driver, while the function in Ghidra is not called anywhere. To confirm this, a search of the memory location of the function is conducted over the entire binary file, but nothing is found. Therefore, the team believes that the RF_init_maybe() in Ghidra is mislabeled and is in fact not the RF_init() function. Finding the actual RF_init() function in the binary file should be a focus of future research.

C. RF Data Queue

The CC1310 uses a data queue to maintain packets which are transferred over the air. Data queues are used for transferring packets from the RF core to the main CPU and vice versa [18]. Since this data queue is volatile it can be inferred the packets present in it will be in the 20kB shared S-RAM on the CC1310 chip. According to [18] there are four types of data queue entries:

- 1) Single Packet Entry: the entry only contains one packet, and the payload(data) is after the header
- 2) Multi Packet Entry: the entry can contain more than one packet, with payload (data) after the header
- 3) Pointer Entry: the entry contains only one packet, with payload at another location in memory
- 4) Partial Entry: stores packets with unknown or unlimited length with payload after the header

Currently it is unknown how the Wyze camera uses the data queue. Of the methods we have discovered in Ghidra, none seem to reference any use of a data queue. In the future, compiling the example code from the SDK and dumping this into Ghidra will allow us to compare the Wyze decompilation and the example code using the Linux executable function "diff". The "diff" function compares two different files line by line and displays the differences. This will be difficult to setup, as the decompiled code in Ghidra may not line up perfectly with the SDK code, but once the files are setup correctly, it could be extremely useful. Comparing the two files will make it easier to see similarities, and possibly discover locations of packet protocols.

The main Thread in rfPacketRx.c can be made into a block diagram or flow chart, to see the order of successive setup and method calls. Creating this block diagram will provide insight into the ordering and flow of the code we find in Ghidra.

Figure 22 shows the block diagram of rfPacketRx.c. The calls to external methods are mostly to RF command handler functions. Most follow the RF_*Cmd naming convention and revolve around maintaining the state of the RF queue, and completing any commands that are sent. Comments in the SDK around these functions refer to a "pool" of commands. It can be assumed this pool is referring to the data queue. Being able to find these basic command handlers in ghidra will point us in the correct direction for finding implementations of data queue usage in the Wyze camera.

D. The Callback Function

In this example implementation RfPacketRx.c, the only other method to the main thread is a method called callback. This method seems to do all of the handling for the data portion of the packet. The method uses a memcpy() to copy the payload of the packet into memory. In this simple example, much isn't done with the packet, but it can be assumed that if the Wyze camera implements something similar, this is where decrypting and handling of packets will occur.

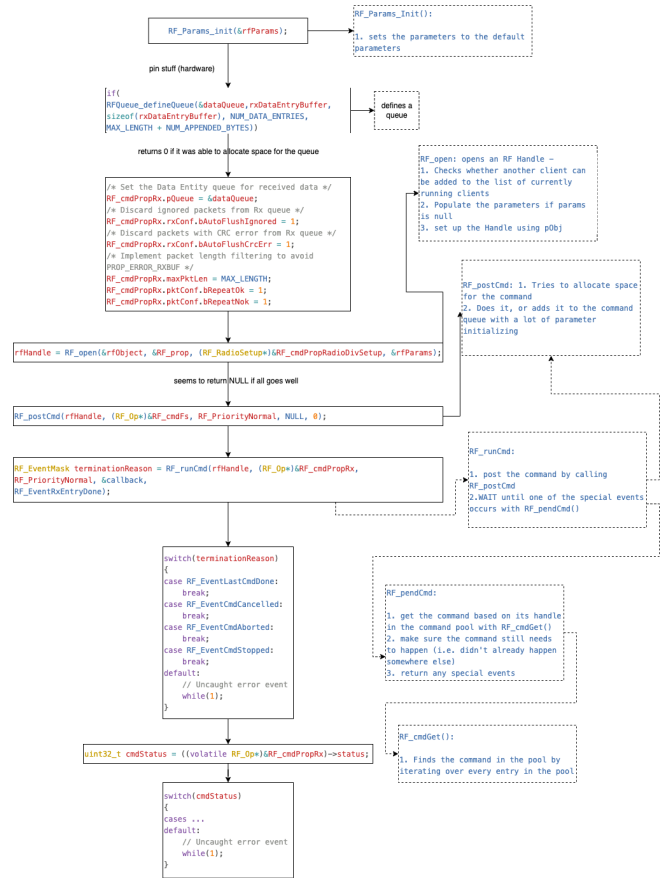


Fig. 22. rfPacketRx.c block diagram

E. The Sync Word

A pivotal part of reverse engineering the OTA protocol depends on finding the sync word that is used. The sync word indicates where actual data in OTA packets begin. It is indicated in the Proprietary RF User's Guide [19] that the default sync word is 32 bits and has the value 0x930B51DE. Using TI's SmartRF studio along with a CC1310 Launchpad, we can configure the launchpad to receive packets from other CC1310 devices, such as the dongle and sensors. However, when the sync word was configured to be the default sync word, no packets were received. Also note that the default sync word value could not be found in any DAT structures in the binary file in Ghidra (0.0.0.33.bin), further supporting the notion that the default sync word is not being used by Wyze. An example window for Smart RF Studio 7 can be seen in Figure 23. Due to the fact that the sync word has a variable size (8-32 bits), it was very hard to try to determine the exact sync word after finding that the default value was not being used.

One way we tried to determine the sync word was by using auto correlation. This method is mentioned in the cc13x0 Proprietary RF User's Guide [19] when describing the format of packets. The guide states that the sync word should be (1) long enough to be unique and (2) the auto-correlation of the sync word must have one high peak with flat side lobes. This can be seen in Figure 24 for the default sync word.

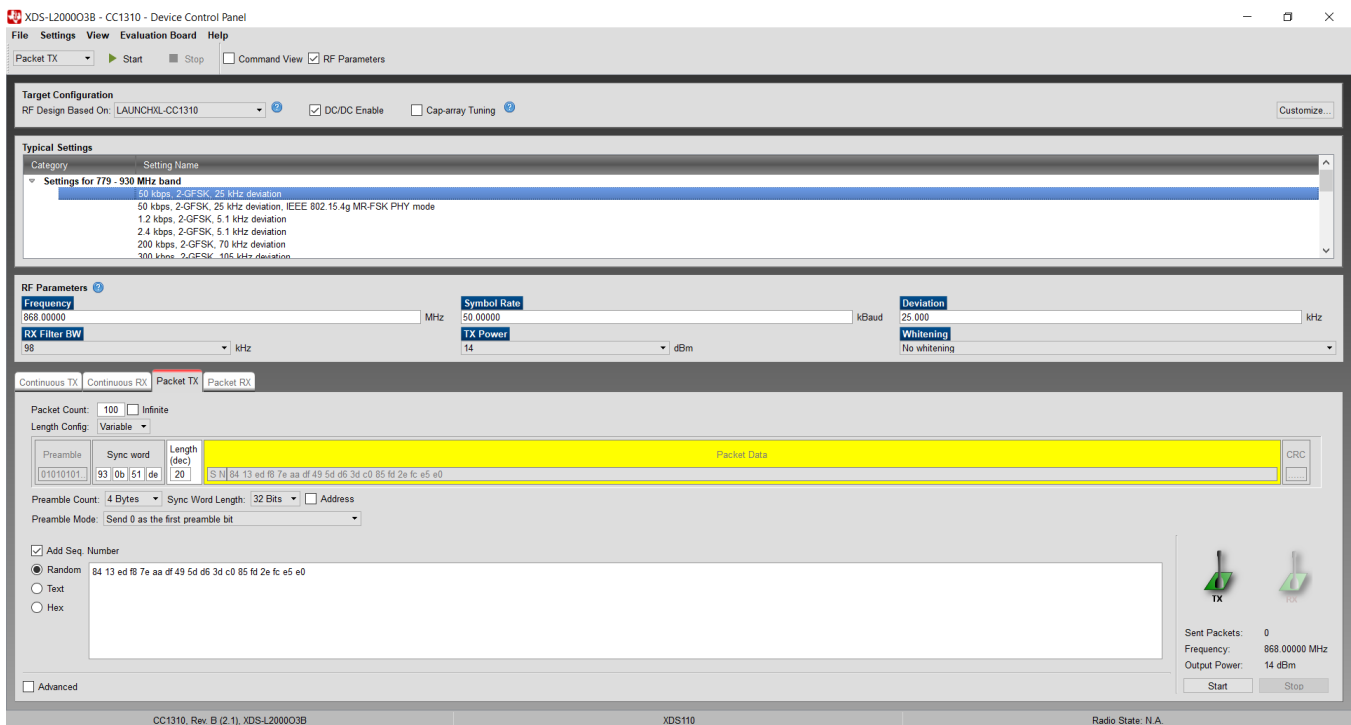


Fig. 23. Window showing RF configuration using Smart RF studio 7

Using the packets captured in Universal Radio Hacker, we tried to use auto correlation over every window of 32 bits in order to find possible sync words. This was repeated for possible sync words of size 24 and 16 bits. Unfortunately, no auto correlation resulted in a single high peak with flat side lobes.

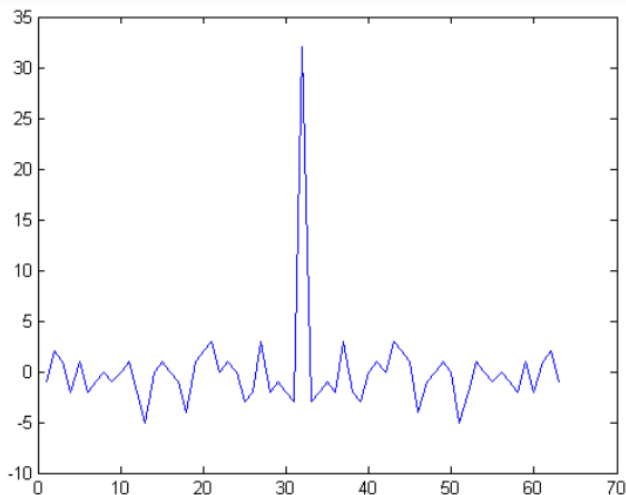


Fig. 24. Auto correlation of default sync word 0x930B51DE

However, it is fair to assume that the developers used Smart RF studio – along with other TI Software such as Code Composer Studio (CCStudio) and Flash Programmer – to configure their CC1310 chips. We can assume that the

Wyze developers used Smart RF Studio to configure the RF protocol of their devices because a key feature of Smart RF Studio is that it allows a user to export a settings file after configuring the necessary RF parameters (preamble, sync word, frequency, etc.). This settings file can be used in C/C++ code to establish the RF settings in code. This takes the burden off the developer to manually set each variable in order to make the RF protocol work the way they intend for it to work. It is also important to determine the sync word that is being used because the sync word is needed in order to unwhiten packets in Universal Radio Hacker.

The Universal Radio Hacker software was also used when trying to determine the sync word of the OTA packets. Note that the preamble in Smart RF studio is either set to repeating 0xAA or repeating 0x55. And then the sync word should arrive after the preamble, and after the sync word is either the length of the packet or the address of where the packet is going (1 byte/8 bits). This is the pattern that we were looking for in the captured packets being used in the replay attack. However, we could not concretely identify any packets that followed this format of preamble, sync word (8-32 bits), address (optional), and length. This could be for a multitude of reasons. It is possible that everything after the sync word is whitened, therefore it would be hard to identify the address bytes and the length bytes. It is also possible that the parameters used to view the bytes in Universal Radio Hacker are wrong even though the captured packet itself is valid. Taking into consideration all these possibilities, the road to finding the sync word is going to be difficult if we can not find an easier way to deduce it.

F. Memory Map

A memory map gives a detailed look at the memory structure of a chip. This semester's focus was the CC1310 chip on the camera dongle. The memory map of this chip includes RAM, SRAM, RFCRAM, CCFG, FCFG1, and many peripherals, some of which is shown in Figure 25.[8]

Base Address	Module	Module Name
0x0000 0000	FLASHMEM	On-Chip Flash
0x2000 0000	SRAM	Low-Leakage RAM
0x2100 0000	RFC_RAM	RF Core RAM
0x4000 0000	SSI0	Synchronous Serial Interface 0
0x4000 1000	UART0	Universal Asynchronous Receiver/Transmitter 0
0x4000 2000	I2C0	I2C Master/Slave Serial Controller 0
0x4000 8000	SSI1	Synchronous Serial Interface 1

Fig. 25. Cortex M3 Memory Map

Because the system uses memory mapped I/O, which means peripherals and instructions are accessed directly through memory without an intervening operating system, these memory addresses are heavily referenced in binary and it is important to have them easily accessible for easy understanding of the code. Ghidra is smart enough to use a memory map in its decompiled code, but it cannot load in the names of regions by itself. In order to bring this memory map into Ghidra, the Ghidra scripting functionality must be used. An easy way to do this is by using a System View Description (SVD) file. An SVD file contains the description of Arm Cortex-M based microcontrollers, including the memory maps[20]. Luckily, leveldown security have already created a well documented script to load these SVD files into Ghidra[21]. The SVD Loader parses the .svd file and creates a memory map in Ghidra. Ghidra is smart enough to take the memory map and translate it to the decompiled code to give memory locations names. The SVD Loader can be downloaded from GitHub and the corresponding cc1310.svd file can be used with it in Ghidra[22]. Some of the loaded memory map can be seen in Figure 26 and this corresponds to the provided memory map from the technical manual[8].

Name	Start	End	Length	R	W	X	Volatile
ram	00000...	0001ffff	0x20000	✓	✓	✓	☐
SRAM	20000...	20004fff	0x5000	✓	✓	✓	☐
RFCRAM	21000...	21000fff	0x1000	✓	✓	✓	☐
I2C0_U...	40000...	40002fff	0x3000	✓	✓	☐	✓
SSI1	40008...	40008fff	0x1000	✓	✓	☐	✓

Fig. 26. Memory Map from Ghidra

G. SVD Loader

This semester's research went on to improve this current SVD Loader. The original loader would create a memory map, but the memory regions themselves would be uninitialized, meaning there was no data for Ghidra to interpret. Specifically the CCFG and FCFG1 regions would be cleared. To create initialized memory in those regions, the original python script needed to be changed. The old script had a loop that went through each region/peripheral, and loaded them into Ghidra with a function that created uninitialized memory. The java API that is used in python scripting also provides functions to create initialized memory regions. To

create initialized memory, you need to provide Ghidra with FileBytes which can be created with a create FileBytes function. FileBytes are created with memory pulled from the chip. To pull memory from the CC1310 chip, wires were soldered onto the test pads of the PCB for an antenna to read the device with a utility called Uniflash. Uniflash allows a user to quickly browse a target devices memory and export the binary of a selected memory region[23]. The JTAG connection was also utilized to export the SRAM. Memory files for the CCFG and FCFG1 regions were pulled and FileBytes were created for these regions. CCFG is the customer configuration of the device and FCFG1 is the factory configuration of the device. These regions may contain important data values that give a better picture of how the device is used. The updated script initializes the CCFG and FCFG1 regions based on their names.

```

if r.name == "FCFG1":
    t = currentProgram.memory.createInitializedBlock(r)
elif r.name == "CCFG":
    t = currentProgram.memory.createInitializedBlock(r)
else:
    t = currentProgram.memory.createUninitializedBlock(

```

Fig. 27. Updated Script Loop Condition

When the loop comes to one of these names it will use the provided FileBytes to initialize the region. To use the current script, you must provide it with the same .svd file, but now the .bin files needed to initialize memory. The initialized memory is seen in Figure 28.

500010dc	3f d0 9e f7	uint	F79E03Fh	CONFIG_SYNTH... Internal. Only to ...
500010e0	3f d0 df f7	uint	F7DF03Fh	CONFIG_SYNTH... Internal. Only to ...
500010e4	3f d0 f0 f7	uint	F7F043Fh	CONFIG_SYNTH... Internal. Only to ...
500010e8	3f 70 f0 f7	uint	F7F073Fh	CONFIG_SYNTH... Internal. Only to ...
500010ec	3f 60 cf f7	uint	F7CF63Fh	CONFIG_SYNTH... Internal. Only to ...
500010f0	3f d0 f0 f7	uint	F7F043Fh	CONFIG_SYNTH... Internal. Only to ...
500010f4	4d 01 da ff	uint	FFDA014Dh	CONFIG_MISC... Internal. Only to ...
500010f8	4d 01 d2 ff	uint	FFD2014Dh	CONFIG_MISC... Internal. Only to ... XREF[1]:
500010fc	4d 01 da ff	uint	FFDA014Dh	CONFIG_MISC... Internal. Only to ... XREF[1]:
50001100	4d 01 d2 ff	uint	FFD2014Dh	CONFIG_MISC... Internal. Only to ... XREF[1]:
50001104	4d 01 da ff	uint	FFDA014Dh	CONFIG_MISC... Internal. Only to ... XREF[1]:
50001108	4d 01 da ff	uint	FFDA014Dh	CONFIG_MISC... Internal. Only to ...

Fig. 28. Initialized Memory in Ghidra from Script

This initialized memory shows the names given from the .svd file as well as the actual data values associated with them. These data fields are defined in the CC13x0 technical manual.[8] One of these values is the USER ID value as seen in Figure 29.

Figure 9-144. USER_ID Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PG_REV	VER	RESERVED				SEQUENCE				PKG					
R-X	R-X	R-X				R-X				R-X					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PROTOCOL								RESERVED							
R-X								R-X							

Fig. 29. USER ID field from Technical Manual

The technical manual defines this data structure to have bits relating to the supported protocols. Specifically bits 12-15. As seen in Figure 29, these bits will show what protocol the Wyze camera uses.

15-12	PROTOCOL	R	X	Protocols supported. 0x1: BLE 0x2: RF4CE 0x4: Zigbee/lowpan 0x8: Proprietary More than one protocol can be supported on same device - values above are then combined. Default value differs depending on partnumber.
-------	----------	---	---	--

Fig. 30. Protocol Bits

When examining this data value in Ghidra, we can see that the bit desired is 0x8, as seen in Figure 30. This means that the protocol used by the Wyze camera is proprietary, as shown above in Figure 30. Therefore, more research will be required to discover how the protocol works. This knowledge can be applied to more data in the CCFG and FCFG1 regions and more information can be discovered on how the camera works internally.

```

00001294 ff          ??      FFh          field_0x294
50001293 ff          ??      FFh          field_0x293
50001294 00 80 01 20    uint    20018000h    USER_ID

50001298 00          ??      00h          field_0x298

```

Fig. 31. USER ID data value in Ghidra

VII. CONCLUSIONS

A. Future Goals

More research into the RF protocol used is necessary. In the future, continuing to look for and locate methods and constants related to the RF data queue in the Ghidra disassembly will help gain a much better understanding of the exact protocol used. Additionally, further investigation into the SRAM pointer described in section VI may be able to help locate RF packets in the SRAM.

The continuation of research regarding the OTA protocol should focus on recalculating the CRC and finding the sync word for the packets. Once the CRC and sync word are correct, the edit message can be retested to see if a valid replay attack occurs. If the message goes through successfully, information about whether or not the packets are whitened and/or encrypted should become clearer to the team. The team will then be one step closer to being able to arbitrarily spoof messages to the Wyze Camera.

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] "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/>.
- [5] B. Dipert, "Teardown: High-quality and inexpensive security camera." <https://www.edn.com/teardown-high-quality-and-inexpensive-security-camera/2/>.
- [6] "8-bit cost-effective enhanced usb microcontroller ch554." <http://wch-ic.com/products/CH554.html>.
- [7] "Rf core — simplelink™ cc13x2 / cc26x2 sdk proprietary rf user's guide 2.80.0 documentation."
- [8] T. Instruments, "Cc13x0, cc26x0 simplelink™ wireless mcu technical reference manual," *Texas Instruments*, Feb 2015.

- [9] A. Ohri, "How to prevent a replay attack in 2021." <https://www.jigsawacademy.com/blogs/cyber-security/replay-attack/>, Mar 2021.
- [10] I. Docs, "Nonce, a randomly generated token." <https://www.ibm.com/docs/en/was-nd/8.5.5?topic=services-nonce-randomly-generated-token>, Nov 2021.
- [11] A. Communications, "Signed firmware, secure boot, and security of private keys," July 2020.
- [12] stacksmashing, "Iot security: Backdooring a smart camera by creating a malicious firmware upgrade." <https://www.youtube.com/watch?v=hV8W4o-Mu2ot=612s>.
- [13] hclxing, "Reverse engineering wyzesense bridge protocol (part ii)," May 2019.
- [14] HclX, "Hclx/wyzesensepy." <https://github.com/HclX/WyzeSensePyreadme>.
- [15] 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.
- [16] "Cyclic redundancy check." https://en.wikipedia.org/wiki/Cyclic_redundancy_check.
- [17] "Ghidra software." <https://ghidra-sre.org/>.
- [18] T. Instruments, "Working with data queues," 2017.
- [19] T. Instruments, "Proprietary rf user's guide 2.60.0," 2017.
- [20] A. Ltd., "Cmsis system view description." [https://www.keil.com/pack/doc/CMSIS/SVD/html/index.html#:text=The CMSIS System View Description,data in device reference manuals](https://www.keil.com/pack/doc/CMSIS/SVD/html/index.html#:text=The%20CMSIS%20System%20View%20Description,data%20in%20device%20reference%20manuals), Jun 2021.
- [21] leveldown security, "Svd loader for ghidra 2019." <https://leveldown.de/blog/svd-loader/>, Sep 2019.
- [22] leveldown security, "Svd loader ghidra repository." <https://github.com/leveldown-security/SVD-Loader-Ghidra>.
- [23] T. Instruments, "Uniflash-quick-start-guide." https://software-dl.ti.com/ccs/esd/uniflash/docs/v70/uni_flash_quick_start_guide.html.