Wyze Camera Report

Antonia Nunley, Joseph Lucas, Makiyah Dee, Marshall Yuan, Sheel Shah, Siddharth Suman
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 goes over known and discovered information about the camera, including its capabilities, found vulnerabilities, and aspects of its firmware. The goal of the research is to discover the Over the Air protocol used by the Wyze camera, so that RF Fuzzing can be achieved.**

I. INTRODUCTION

The Wyze Camera V2 is an Internet of Things (IoT) Device. It allows for wireless connection to multiple devices in a location in order to provide surveillance of the locations in which the camera(s), motion sensor(s) and contact sensor(s) are placed. Contact sensors provide enhanced security in giving the user access to physical interruptions in the environment of the camera. These are most commonly used on windows and doors. Motion sensors not only allow for a fast and reliable way to notify users of disturbances in the area they have placed their system, but also give an extra layer of security allowing users to place them outside of the views of the camera to serve as a precursor that something may come into view soon. This expands the space that can be monitored with this system. The surveillance of the locations and status of their devices are provided for users through the "Wyze - Make your Home Smarter" mobile application.

Recently, there has been a 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 devices using wireless and cloud based technology increases, the risk of it being attacked by individuals with malicious intent has too [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].

An example would be the CloudPets toys released by the company Spiral Toys, which acted as IoT devices by allowing parents to communicate with their children through the toys via the internet. In late 2016 to early 2017, it was found that the CloudPets toys had some massive security vulnerabilities, which lead 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].

Another example involving the Wyze camera this team is working on occurred in 2019. Wyze confirmed that from December 4th to December 26th of 2019, a large amount of personal data was leaked for more than 2.4 million users [5]. These examples should abundantly show that enhanced security on IoT devices is increasingly necessary.

The purpose of this research team is to achieve RF fuzzing for the Wyze camera. Fuzzing is a technique to automatically input structured malformed data into a computer programming. Monitoring the output of the program while doing this helps find crashes, memory leaks, and other issues, which would present a way to more easily discover security flaws in the Wyze camera. To achieve RF Fuzzing, the main goal of the team is to decode the Over the Air (OTA) protocol of the Wyze camera. Sections II and III will detail known information about the Wyze camera, then following sections will document the progress the team has made regarding the OTA protocol.

II. FUNCTIONAL DESCRIPTIONS

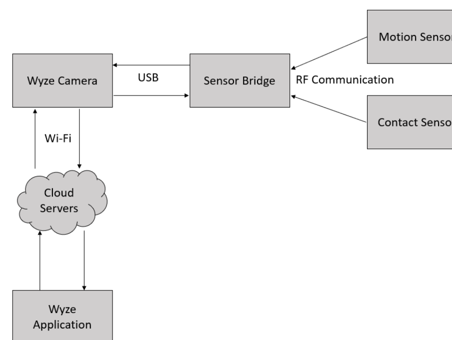


Fig. 1. Overview of System Communications

The Wyze IP Camera V2 setup, as shown in Figure 1, consists of mainly four parts: the motion and contacts sensors, the sensor bridge, the camera and the Wyze application. The sensors communicate with the sensor bridge through radio frequency comunication (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 [6]. We have JTAG

access to the sensor bridge and sensors which allows for dynamic analysis as well as the ability to grab snapshots of memory.

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 system.

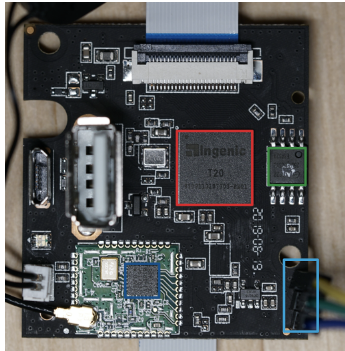


Fig. 2. PCB of the Main Board

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

TABLE I
COMPONENTS FOR MAIN BOARD PCB

The PCB shown in Figure 2 is one of the circuit boards inside the Wyze camera. The board runs on a T20 processor [7] that uses the MIPS (Million Instructions per Second) ISA (Instruction Set Architecture). This is the main board and it also contains flash memory, Wi-Fi, and serial access as shown in the figure. The open serial access provides a root shell to interact with the provided Linux OS.

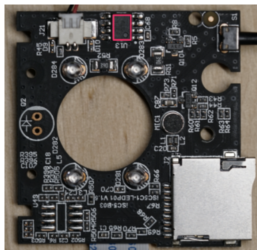


Fig. 3. microSD PCB for the Main Board with motor driver in red

Figure 3 shows the PCB that has a motor driver to move the camera as well as an SD card slot. There is another PCB shown in figure 4 that is a part of the main camera system.

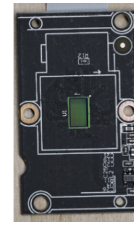


Fig. 4. sensor board with no lens

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, which deals with the transmitting and receiving of RF packets and converting packets into data. There is also an antenna for this wireless communication.

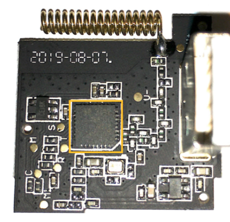


Fig. 5. Front of Sensor Bridge PCB with CC1310 highlighted in Orange

The sensor bridge connects to the main camera via USB-A, which is controlled with a WCH CH554T chip on the back of the board [8].

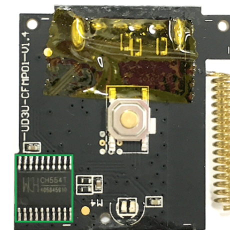


Fig. 6. Back of Sensor Bridge PCB with WCH CH554T chip highlighted in green

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.

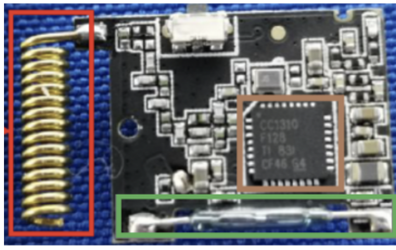


Fig. 7. Contact Sensor PCB

Label Color	Red	Brown	Green
Component	Antenna	CC1310	Magnetic Switch

TABLE II
COMPONENTS FOR THE CONTACT SENSOR

The contact sensor has a magnetic switch that provides feedback to the CC1310 GPIO. The CC1310 handles RF communication and will send packets to the sensor bridge based on this feedback.

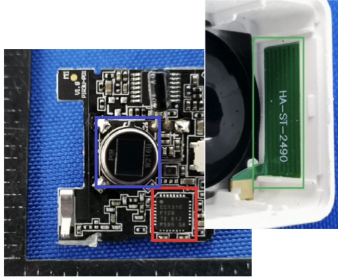


Fig. 8. Motion Sensor PCB

Label Color	Blue	Red	Green
Component	PIR Motion Sensor	CC1310	Antenna

TABLE III
COMPONENTS FOR THE MOTION SENSOR

The motion sensor has a PIR (passive infared) sensor that provides feedback to the micro-controller. The CC1310 handles the RF communication with the sensor bridge.

D. CC1310 Micro controller

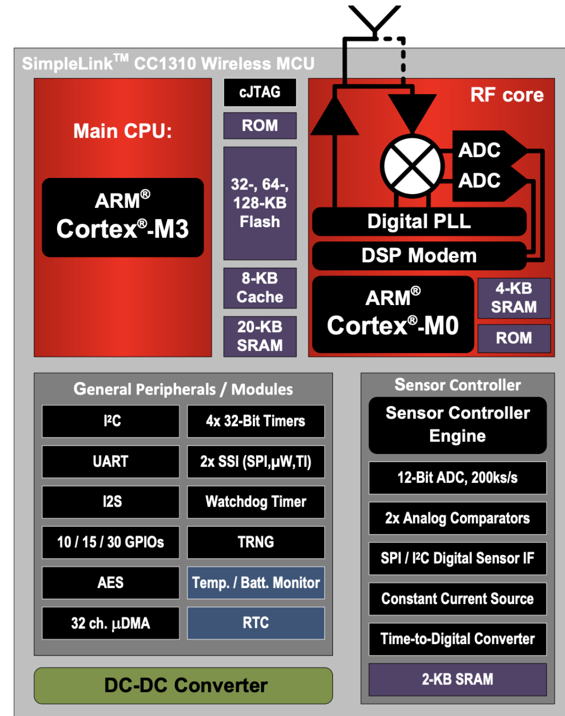


Fig. 9. CC1310 functional block diagram

Figure 9 shows the CC1310 TI Simplelink Wireless MCU that the Wyze Camera uses as the main chip on the sensor bridge as well as the contact and motion sensors. The goal of our work was to focus on this processor, specifically the RF core, its use, and how it interacts with the main CPU. This is the radio peripheral of the CC1310 which can be programmed to handle multiple protocol standards [9]. The radio protocols include 802.15.4 RF4CE and ZigBee®, Bluetooth® low energy, etc [10].

The system CPU is the main CPU in figure 9, also labeled ARM® Cortex®-M3. It is the main system processor that handles the application layer, running the user's application, and the high-level protocol stack [10]. It runs code from the boot ROM and the system flash. The ROM is loaded with a boot sequence, low-level protocol stack, device driver functions, and a serial bootloader [10]. The system flash is nonvolatile memory that saves certain data when the device is turned off so that it is available again after restarting the device [10].

The RF core contains a ARM® Cortex®-M0 as shown in figure 9. The M0 interfaces the analog RF and baseband circuitries, handles data to and from the system side, and assembles information bits in a given packet structure [10]. The RF core has a dedicated 4-KB SRAM block and runs almost entirely from a separate ROM [10]. This means we might not have the code on the ROM, which contains useful information, such as the RF processing code [10]. Currently,

we are most interested in how the M0 and the RF core as a whole communicates with the main CPU, M3.

The Direct Memory Access (DMA) moves information from the radio RAM to the system RAM and vice versa, if direct CPU access is not used [10]. However, the primary way for the system CPU and radio CPU is the radio doorbell module (RFC_DBELL), also known as the command and packet engine (CPE) [10]. the RFC_DBELL contains a set of dedicated registers, parameters in any of the SRAMs of the device, and a set of interrupts to both the radio CPU, M0, and the system CPU, M3 [10]. This means that the RFC_DBELL can be used to send information and instructions between the system CPU and radio CPU by changing parameters and interrupts in the RFC_DBELL.

The M3 processor uses a 20KB single-cycle on-chip SRAM with full retention in all power modes, except shutdown [10]. This SRAM contains packet information (TX and RX payloads) and the different parameters or configuration options for a given transaction [10]. This type of SRAM is useful to share and store information between multiple parts of the chip.

Using a TI debugger tool (XDS110) that is hooked to the Wyze camera, we are able to dump the memory of the device while it is in operation. This memory will contain a capture of the SRAM at the moment that the dump occurred. 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 as part of the Over the Air(OTA) radio protocol used by the RF core before they are transmitted [10]. Discovering if these packets are encrypted, and understanding the encryption protocols of these packets if they are, was one the goals of this semester.

Not much is known about the RF Protocol being used by the RF Core of the CC1310 however, sections V and VI go further in depth about what was discovered about the RF Protocol from the team's research.

E. Technical Documents

The TI CC13x0, CC26x0 SimpleLink™ Wireless MCU Technical Reference Manual Technical manual is the technical manual for the above mentioned CC1310 MCU. The manual contains important information about how the chip works, which can help with understanding how the Wyze camera is using the chip.

The Texas Instruments CC13x0, CC26x0 Software Development Kit (SDK) is useful for understanding the Wyze firmware. An SDK somewhat abstracts away the hardware for developers wanting to create applications using the chip. Furthermore, it allows developers to make application programming interface (API) calls to tell the chip to do something. Importantly for this team, the CC13x0 SDK also comes with example code of the API being used. 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 applications 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.

III. Existing Vulnerabilities

The Wyze Camera system contains vulnerabilities that can be exploited to leak sensitive information. These vulnerabilities are susceptible to several attacks, but the most significant attack that we have found to the Wyze Camera is a replay attack. Another concern is that the Wyze Camera's firmware is not signed, which can allow a user to modify it. Firmware signing will be explained in detail in the succeeding sections of this paper.

A. Replay Attacks

Replay attacks are detrimental to the security of the Wyze Camera. In regards to the Wyze camera, the team initiated replau attacks by replaying valid data transmissions. Replay attacks indicate that messages do not have proper authentication. A countermeasure in response to replay attacks is to use a nonce, which is a number used once [11]. Nonces allow the authentication of messages to ensure that the messages cannot be replayed. Since replay attacks can be played to the Wyze Camera system, we know that the Wyze Over-the-Air (OTA) protocol does not implement nonces.

Previously captured packets from a contact and motion sensor were replayed to the dongle by an USRP N210. The packets were successfully received by the dongle as it was verified by the chosen input being displayed in the Wyze application. 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 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 are able to 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, 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 modified 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 [12]. The firmware on the devices of the Wyze system are not signed, allowing unauthorized firmware to be flashed onto the devices [13].

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 [14]:

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

We were able to obtain log files for different communication scenarios between the devices by monitoring communication at startup, and movement in front of the camera and sensors. This allowed us to see differenc scenarios of communication between the devices. 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.

A. *Packet Decomposition*

Communication between the camera and dongle happens through over the air packets (OTA). Starting here, we are able to see the communication methods, and logs which provide information into the metadata of the camera, sensors, and other devices which are connected to the Wyze system. In Ghidra, we have created a serial packet register (struct) which breaks down the important sections of the packets which are sent OTA. The figure below shows the header of those packets. The header is uniform over each packet, only differing in contents, but the size is consistently 4 bytes.

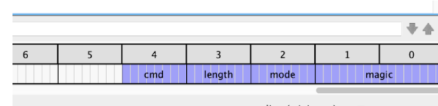


Fig. 11. Ghidra Serial Packet Struct Header

The serial packets which are sent between the camera and the sensor bridge differ in length after the header 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. [14] 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.

```
                /* 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. 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. [14]

V. SIGNAL ANALYSIS OF THE RF PROTOCOL

The focus of the team's current research is on reverse engineering Wyze's proprietary RF protocol between the contact sensors, motion sensors and the sensor bridge. However, the team must know whether or not the Over-The-Air packets are whitened and/or encrypted before proceeding. The team can gain insight into the problem by analyzing 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 the packet can be stored and analyzed. An Ettus N210 SDR was used to capture packets going between the dongle and sensors. GNU Radio was used to record the packets. Depicted below is the GNU Radio flowgraph used to record the packets and the block diagram for data flow:

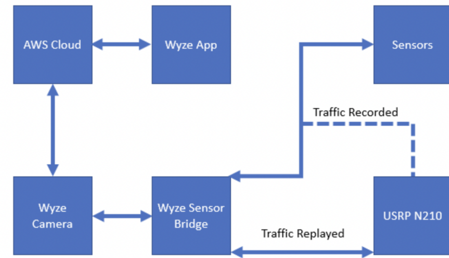


Fig. 13. Block Diagram showing where Packets and Logs were captured

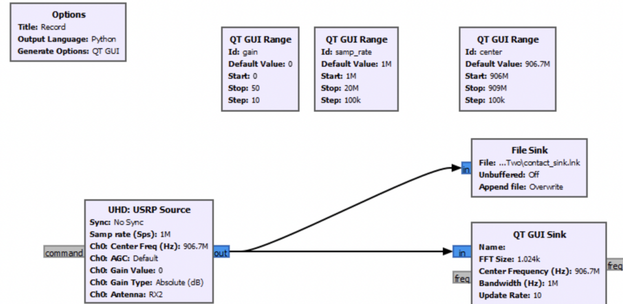


Fig. 14. GNU Radio flow graph used to record packets for playback

Serial logs can be directly captured by using WyzeSensePy [15], a raspberry pi and a USB connection to the dongle, which implements the communication protocol between the dongle and camera. The dongle was acting as an interface to send/receive messages to/from the contact sensor, and the WyzeSensePy printed out the serial logs that contain the serial data after the RF data (data sent over-the-air) reached its destination.

B. RF Overview

Information from the contact and motion sensor is read by their microcontrollers and modulated through Gaussian Frequency Shift Keying (GFSK). URH [16] assists in analyzing OTA packets to understand the protocol between the camera and the dongle.

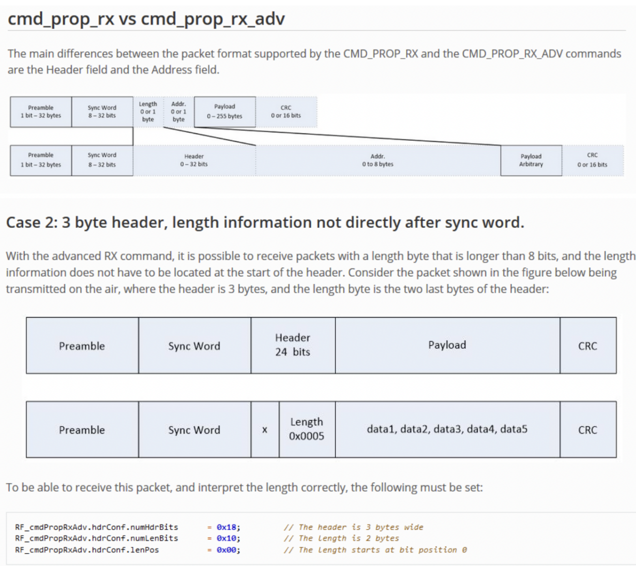
C. Sensor Over-The-Air Packet

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 is known. OTA packets are composed of what we suspect is

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 would look for its MAC address to see first if it is compatible to accept the message. Sensors transmit how much battery they have left to the dongle via the sensor bridge as it was seen in the WyzeSensePy debug information with a marker that read "battery =" some number. Finally, the type of event is transmitted (open/close, motion/no motion). Given that we know that the packets are largely the same, after reviewing the different open and closed packets that only differed 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 key resource for understanding the physical structure of the packets. We have an idea that Wyze is using advanced TX/RX packets from TI since the packets may be modified during the time it is sent over-the-air. 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. 15. RX information

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 "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 events of the same type of alerts consecutively, will result in the second message being considered an error, and not 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 through a frequency of 906.8MHz with a modulation type of FSK. 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, delete.

G. Motion Sensor

URH was used to conduct an analysis of the motion sensor captures as well as analyzing the log files by parsing the contents to find commonalities. The goal of this was to to gain more insight about 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 begin uncovering the details of the OTA packets, the team reviewed work done in previous semesters and continued along the outline depicted in Figure 16.

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:	-	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

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

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. 16. Outline of Continuing Research in the OTA Protocol

Using URH, the team would take a previously successful replay attack and edit some of the bits. We would download the edited replay attack (converted back to complex data from the demodulated bits) from URH and uploaded it to the flowgraph in GNUradio companion (Figure 14) to have it tested to see if a valid open replay attack still occurred; 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 a multitude of reasons as to why the edited message did not go through: the cyclic redundancy check (CRC) [17] 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. The sync word, as mentioned later, will allow the team to see where the data begins in the packet, which would allow us to edit the data and see if the message can be replayed and captured by the Wyze Camera as a valid message.

I. 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 [18] that the default sync word is 32 bits and has the value 0x930B51DE. To find the sync word the team used TI's SmartRF studio along with a CC1310 Launchpad (it is part of the development kit for the cc1310 chip — the same chip that is used in the Wyze Camera [19]), we can configure the launchpad to receive packets from other CC1310 devices, such as the dongle and sensors. The team did not proceed with this approach as they discovered a data structure that revealed the sync word to be: 0x5555904E. However, in the following paragraphs the team explains how they used TI's SmartRF studio to gain further insight about 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:

Fig. 17. Packet Data is Revealed with the incorrect sync word

Fig. 18. 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:

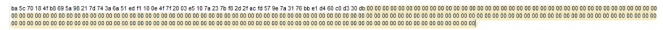


Fig. 19. Packet Data is Revealed with the correct sync word

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 [20].

VI. REVERSE ENGINEERING THE RF PROTOCOL

A. Ghidra

The primary tool used to reverse engineer the binary files is Ghidra. Ghidra is a software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission[21]. The firmware of the Wyze camera, that was obtained by dumping the memory using a TI debugger, 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 [22] 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 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 using Jlinks memory dump feature 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 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. 20. 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 function called $radio_three$ was reversed engineered, which we assumed could be potentially related to RF protocol. This function mainly dealt with updating memory and interrupt flags through other sub functions. One main sub function that indirectly interacted with the transmission of the RF protocol was called RFCDoorBellTo which takes an passed in input from the $radio_three$ and sends a command stating if a message is set to be transmitted. This indicates that the function does not directly affect communication of the OTA protocol but sends some kind of information commanding the state of the radio.

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, introduced in Section II.

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, the radio peripheral mentioned in Section II that would handle the Over The Air protocol of the radio. The function the team focused on is RF_init(), the function that 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 the team during previous semesters.

While exploring these two functions, the team realized that the RF_init_maybe() function is likely not 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. It is not ruled out that Ghidra is misinterpreting the binary or we are wrong in our assumption. However, given the current evidence, the team believes that the RF_init_maybe() in Ghidra is mislabeled and is in fact not the RF_init() function. The team will be operating under this assumption until new evidence is found or the assumption is confirmed.

C. RFC_DBELL

The RFC_DBELL located at memory location 0x40041000 is crucial for narrowing down the relevant functions for finding 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 to both the radio and system CPU [10]. Thus the RFC_DBELL allows the system and radio to send information to each other through the registers, as well as send commands to each other through the interrupts. As such, all functions that reference the memory location of the RFC_DBELL are likely to be related to the RF protocol and are a high-priority for reverse engineering. We can see these referenced functions thanks to the work of the SVD loader which labeled this memory region while still keeping the Ghidra created cross references [22].

One of these functions is 00008bc0_radio_one, hereafter referred to as Radio1, which is one of the functions that interact with the interrupts stored in the RFC_DBELL section of memory. Specifically, it modifies the values of the RFCPEIEN/RFCPEIFG and RFHWIEN/RFHWIFG pairs of interrupt registers in the RFC_DBELL. All interrupt registers are 32-bits, and for the pairs that Radio1 interacts with, each bit represents a different interrupt that is only active when the same bit is set in both registers. The RFCPEIEN/RFCPEIFG pair corresponds to Command and Packet Engine Generated Interrupts, and the RFHWIEN/RFHWIFG pair corresponds to RF Hardware Modules [10]. Radio1 overall appears to be more of a setup function, as it just modifies the values in certain memory locations and interrupt registers, and it always calls the 0000cfdc_radio_two and

0000ee60_radio_three functions before the function concludes.

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

Another function found through the RFC_DBELL is the RFCRTrimRead() function. This function is found in the TI SDK in the rfc.c file. Within this function, an important struct, called rfc_CMD_PROP_RADIO_DIV_SETUP_t, is referenced. This is an important struct because it contains information relating to the whitening protocol and the number of preamble bytes. Both of these pieces of information are very helpful for understanding the OTA protocol. As referenced in RF Command Structures (VI.G), this struct was later found using a combination of static and dynamic analysis and this would not have been possible without the RFC_DBELL command registers.

D. dongle_app

The dongle app contains multiple areas of importance in the RE area. Specifically, functions were found that were crucial for the error handling that occurs. One of the functions was msgsnd. This was found to be important as it is responsible for returning several things, such as message_queue_id, message_pointer, message_size and message_flag, all of which are utilized in various ways across the dongle_app.

Furthermore, msg_success_checker plays an important role in determining whether a message being sent is successful or not. This is shown by the return value of the function, where 0 is successful and less than 0 is unsuccessful. There is a loop that makes 3 attempts to get a successful message, and if not, it is unsuccessful. This function is used in many places where error messages occur, such as when verifying camera info, setting camera to play an audio file or upgrading dongle information. The 0 or less than 0 allows for easy programming of error messages. This function is crucial for being able to design error message protocols and for programming error messages, as seen in the add_to_msg_queue function.

Finally, the add_to_msg_queue function adds messages to a queue and is utilizes msg_success_checker to be certain that a message added to the queue is successful or not. In the instance that it is not successful, it will not add to the queue. It also sends an error message containing the error number. This process is a depiction of how the msg_success_checker is used across multiple functions.

E. Important Data Structures

An important set of data structure was discovered that starts at memory location 20003168. Each data structure starts at an offset of a multiple of 0x30 from 20003168. In the current snapshot of the SRAM there is a data structure in 20003168 and 20003198. These data structures are believed to be important as they are referenced extensively throughout the firmware, including in the Radio1 and Radio2 functions discussed in the RFC_DBELL section. Moreover, in the two data structures present in the current snapshot of the SRAM, they each contain a reference to either rfc_CMD_PROP_TX_ADV_s_20002330 or rfc_CMD_PROP_RX_ADV_s_20002378. These are locations in memory that store important information relating to the RF.

F. 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 [23]. Currently it is unknown if or how the Wyze camera uses the data queue. Documentation in the CC1310 manual does not indicate that it is optional, and therefore we are assuming it is being used.

Plans for confirming this include compiling the example code rfPacketRx.c. In order to do this, cross compilation is necessary. This is the process of compiling code on a machine with a certain instruction set architecture (ISA) (eg. x86) for a machine with a different ISA (eg. ARM). This directly corresponds to the ISA differences in common computers that run on an x86 architecture and the camera dongle chip. A binary specifically compiled for an ARM system is necessary to compare to the Wyze camera and CC1310 since this is the ISA of the CC1310. There are many cross compilers that exist that allow ARM cross compilation. The one used for this project, which allowed us to compile for our ARM processor was the buildroot toolkit [24], which is a Linux kernel toolkit which focuses on allowing cross compilation for embedded systems. Using buildroot, it is possible to cross compile c code meant for an ARM processor on a x86 64 bit Linux machine.

Once the example code is compiled, we can then load the binary file into Ghidra. This will allow a diff to be run between the example code, and the Wyze Camera binary. Focusing on similarities instead of differences between the two files, it will be possible to see any sections that directly relate to use of the data queue in the CC1310 Wyze binary. This allows the group to locate useful address spaces that may possibly hold packets in memory, or functions that directly interact with recieved packets.

This was done using a Ghidra plugin: UBF Bindiff Helper [25]. Which emphasized differences over similarities. This made it much harder to find sections of code which were the

same in the binaries. However, immediately after loading the files, the actual structs were found and we pivoted our focus.

G. RF Command Structures

As mentioned in the RFC_DBELL (VI.C) subsection as well as the Packet Contents (V.A) subsection, there are important RF structures that are referenced in code to initialize, send, and receive packets with the radio. More specifically, rfc_CMD_PROP_RADIO_DIV_SETUP_t, rfc_CMD_PROP_RX_ADV_t, and rfc_CMD_PROP_TX_ADV_t are referenced in the code and contain important information relating to the OTA protocol. [10] This information will help in demodulating and interpreting the packets received from the system. All the data from 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 sensors firmware. As mentioned in the RFC_DBELL (VI.C) subsection, it is referenced in the RFCRTrimRead() function. Dynamic analysis involves us actually executing code on the system and analyzing things like register values and this allows us to confirm which branch the system takes. JLink is the tool we use to do this. Using the JLink we set a break point 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. 21. Assembly of RFCRTrimRead() showing R0 referencing the struct

At address 0000bbca 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. 22. Raw bytes at SRAM location of structs

Address 20002350 was the address pulled from R0. As we can see 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 made sense based on 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 was interesting and led us to look for more 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 both 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 contain lots of information describing how the RF portion of the packets are set up and sent including the sync word, the preamble, the baud rate, header information, the whitening mode, and many more important values. Please refer to the appendix for the struct definitions and initialization values as found in the SRAM of the dongle. When refering 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 propritary 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 can now work to demodulate and interpret data received from the system.

VII. CONCLUSIONS

A. Future Goals

As mentioned in RF Command Structures (VI.G), information relating to how the Wyze system initializes its proprietary radio has been found in SRAM. Now that we have this OTA protocol information we need to work on demodulating and interpreting packets from the system. With the whitening protocol we can demodulate the IQ data received and work on removing RF information from the packets so we just have application data. We can take this application data from different states of the system and see what changes. Once we have a complete understanding of the packets, we can work on creating a device to spoof the system.

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] 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.

[12] A. Communications, “Signed firmware, secure boot, and security of private keys,” July 2020.

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

[14] hclxing, “Reverse engineering wyzesense bridge protocol (part ii),” May 2019.

[15] HclX, “Hclx/wyzesensepy.” https://github.com/HclX/WyzeSensePyreadme.

[16] 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.

[17] “Cyclic redundancy check.” https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[18] T. Instruments, “Proprietary rf user’s guide 2.60.0,” 2017.

[19] “Meet the cc1310 launchpad.” https://dev.ti.com/tirex/explore/node?devtools=LAUNCHXL-CC1310node=AB1pdyKg1Uipdgbfl4wJbQ$_{FUz-xrs_{LATEST}}$.

[20] “What is anti-replay protocol and how does it work?.”

[21] “Ghidra software.” https://ghidra-sre.org/.

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

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

[24] B. Association, “Bibroot toolkit,” Mar 2016.

[25] Ubfx, “Ubfx/bindiffhelper: Ghidra extension to integrate bindiff for function matching.”

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
};
```