Wyze Camera Fall 2024

Michael Edoigiawerie, Varsha Jacob, Joshua Muehring, Deniz Timurturkan,
Andrew Graffeo, Ben Cordova, Tanay Rajkumar, Blake Wood

December 2024

Embedded Systems Cyber Security
Georgia Institute of Technology
Atlanta, Georgia

***Abstract* — This paper covers details about the Wyze Camera and the Georgia Tech Embedded System Cyber Security's research into the camera as of Fall 2024. The goal of the research project is to exploit the Wyze IP Camera. This paper demonstrates our work to reverse engineer the Wyze Camera's radio frequency (RF) protocols. The team will then try to replicate these results through a process called fuzzing. The results of these two processes will then be compared to see if fuzzing is a viable alternative for manual reverse engineering RF devices.**

I. Introduction

Georgia Tech's Embedded System Cyber Security Team's goal is to analyze embedded systems through various means such as reverse engineering, vulnerability discovery, and forensics analysis. The team works on various devices such as radios, modems, routers, and embedded controllers. The main focus on this paper is the team's work on the Wyze Camera.

The Wyze IP camera is a popular branded camera that people use for both their homes as well as their businesses. The Wyze camera is controlled by using the Wyze mobile application. Users must first download the Wyze app, and then place the cameras at which ever location they desire. After completing the setup on the app, users are able to utilize the Wyze camera to its fullest. Some of Wyze camera's capabilities include two-way audio, integration with other smart home devices like Amazon Alexa, motion and sound detection, as well being able to view the camera feed using the mobile application.

While the Wyze camera is very popular, numerous vulnerabilities within the device have cause some consumers to distrust Wyze's systems. Some of the discovered vulnerabilities include authentication bypass, remote control execution flaw caused by a stack-based buffer overflow, and access to the Camera's SD card without authentication [3]. Breaking the cameras system will allow for the discovery on new vulnerabilities.

In the previous semester, the team used the memory capture in order to reverse engineer the camera's process of data manipulation. This semester, the team plans to further understand the over-the-air protocol between the sensors and camera. Furthermore, the team plans to develop a testbed for RF fuzzing, or sending malformed data to the program. The end goal of the team's research is to exploit these systems through a process called fuzzing to generate the same results in order to see if fuzzing is a feasible alternative for manually reverse engineering RF devices.

II. Device Description

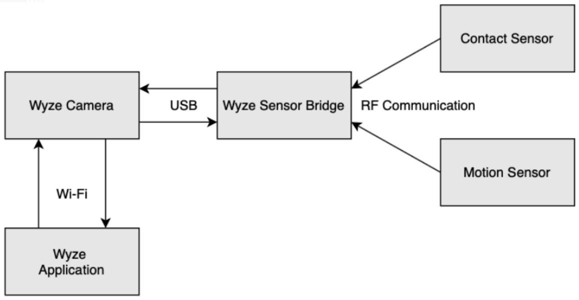


Fig. 1. Diagram of high-level components of the Wyze Cam V2, annotated with means of internal dataflow.

The team is currently working on a Wyze Cam V2. It is made up of several devices: a sensor bridge, contact sensor, motion sensor, and the camera itself. The motion and contact sensors communicate with the sensor bridge through RF signals. The sensor bridge and camera are connected through USB, allowing for the transmission of information to and from the sensors. The camera and Wyze cloud servers are internet connected, which allows for data to be transferred to the Wyze application from the servers and vice versa.

Each one of these high-level components represents a distinct board within the Wyze camera system. The subsequent sections will enumerate every components with a photo and functional description.

A. *Contact Sensor and Motion Sensor*

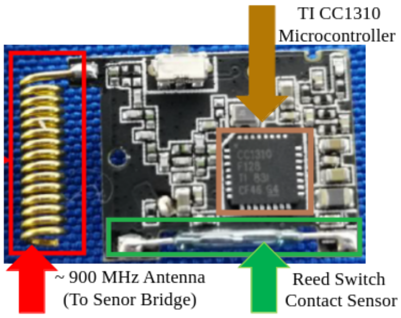


Fig. 2. Labeled photo of contact sensor board.

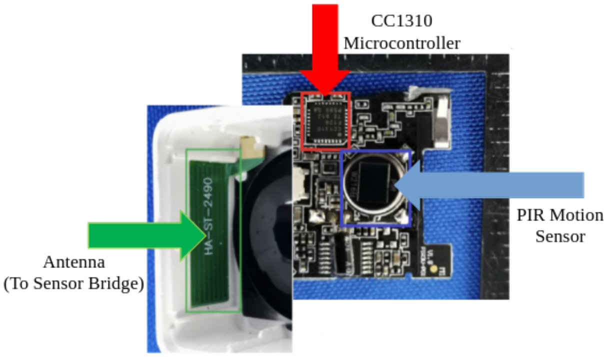


Fig. 3. Labeled photo of motion sensor board.

The contact sensor and motion sensor communicate wirelessly with the sensor bridge. The printed circuit boards (PCBs) for each sensor are similar, with both having an antenna and CC1310 microcontroller. However, the contact sensor has a magnetic switch and the motion sensor has a passive infrared (PIR) motion sensor. The magnetic switch on the contact sensor aids in the transmission of data from the sensor to the sensor bridge. When the state of the switch changes (whether or not there is a magnet pressed against the switch), a packet is created and sent to the camera. Similarly, the motion sensor's PIR sensor helps with the wireless transmission of messages to the sensor bridge.

B. *Sensor Bridge*

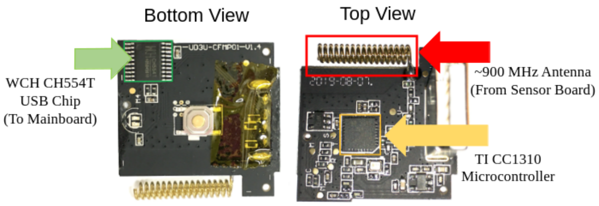


Fig. 4. Labeled photo of sensor bridge board.

The sensor bridge board acts as the intermediary between the camera and the sensors. The antenna receives the RF signals from the two sensor boards and then processes the input using its ARM TI CC1310 Microcontroller Unit (MCU). Unlike the similar MCUs on the sensors, the firmware of that on the sensor bridge board is proprietary to Wyze and contains the bulk of the firmware of interest for our reverse engineering team.

The MCU has 128 KB of flash memory and 20 KB of SRAM. It also supports several protocols, including IEEE 802.15.4g, 6LoWPAN, and proprietary RF protocols [1], such as the proprietary protocol that Wyze uses. The team's research is mostly focused on its RF protocol. By knowing exactly how the board converts input from its antenna to control signals to the mainboard, this opens up the possibility for a replay attack in which carefully curated signals can be sent to exploit the system, a process detailed further in later sections.

C. *Mainboard*

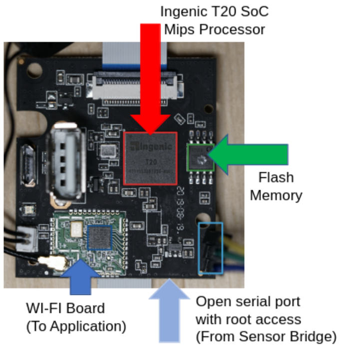


Fig. 5. Labeled photo of the camera mainboard.

The mainboard contains an Ingenic T20 MIPS processor that runs proprietary Wyze firmware responsible for receiving the control packets from the sensor board and relaying it over Wi-Fi to the main

Wyze Applicaiton. This board also runs an embedded Linux for which root access can be obtained using the open serial port that connects to the sensor bridge. The team was able to interface with the board directly by soldering headers to this port and cracking the root password.

D. Technical Documents

Lastly, there are several technical documents that the team is referencing. These include the TI CC13X0, CC26X0 SimpleLink™ Wireless MCU Technical Reference Manual and the TI CC13X0, CC26X0 Software Development Kit (SDK).

The technical manual contains information about the CC1310 microcontroller in use in the sensor bridge, contact sensor, and motion sensor of the Wyze Cam V2 [1]. Specifically, the team is referencing the manual for information regarding the radio: the RF core, data queue usage, radio registers, and proprietary radio information. The technical manual has helped the team discover more about the proprietary radio commands, the packet format, and the different command structures.

The SDK provides an application programming interface (API) for the microcontroller. However, most importantly, it includes sample code for the API, giving several examples to cross-reference with the camera's disassembled firmware. By referencing the SDK, the team has familiarized itself with the uses and meanings of structures in the firmware's memory. This is especially useful when examining and reverse-engineering the firmware. Finally, there are several examples of RF protocols in the SDK, showing possible implementations of packet transmission and reception [2].

III. Existing Vulnerabilities

Several vulnerabilities are discovered in the Wyze camera system that could be exploited to leak users' sensitive information. For instance, Bitdefender's vulnerability assessment of Wyze Camera revealed that this device may be vulnerable to remote connection authentication bypass (CVE-2019-9564) which occurs when client sends IOCtl command and NULL value in place of ID 0x2710 to the device where authentication code is already NULL, thus comparing NULL to NULL would result in successful authentication [3]. Other vulnerabilities found by Bitdefender include unauthenticated access to contents of the SD card and remote code execution flaw caused by stack-based buffer overflow (CVE-2019-12266) caused by camera's lack of checking mechanism for destination buffer's size when it comes to processing IOCtl with 0x2776 [3]. Other weakness found in Wyze camera device is the fact that power consumption dramatically increased in both idle and active state when the device is experiencing distributed denial of service (DDoS) attack compared to control and Man in the Middle (MitM) attack group [4].

Although these vulnerabilities already exists within the Wyze Camera system, the research team discovered that Wyze camera system is susceptible to a replay attack.

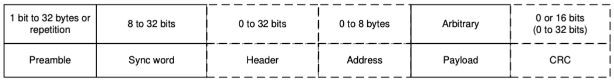
A Replay attack is one of the vulnerabilities in the Wyze camera system. Replay attacks occur when malicious actor is able to capture the messages and retransmit them to the recipients [5]. To demonstrate this vulnerability, the team first captured packets by recording signals with GNU radio using the contact and motion sensors. These captured packets were then replayed back to the sensor bridge, using a Ettus USRP N210. The bridge would then send back an ACK packet that indicates successful reception of the replayed message. The first packet associated with alert contains 4-hexadecimal character value representing increment counter. This counter is maintained in the sensor and incremented each time an event alert (open/close, motion/no motion) occurs. If the sensor is switched off, the 4-hexadecimal character value would reset to 0. When captured packets were replayed, the 4-character field would return to the value that was captured. To send arbitrary packets to the dongle, Universal Radio Hacker (URH)'s generate functionality was used to modulate the signals.

Unsigned Wyze camera's firmware is another factor contributing to the Wyze camera system's susceptibility. In embedded systems, firmware authors can choose to enhance security posture by implementing firmware signing. This can potentially prevent their firmware from being modified or corrupted. The process of signing a firmware involves the creation of cryptographic hash of the target firmware which is signed with a private key resulting in the signature attached to the firmware image [6]. The team modified the stock firmware and were able to install it onto the Wyze camera, indicating that it is not signed by the vendor [7]. This is another vulnerability that could be used to bypass security controls.

IV. Analysis

A. TX

The CC1310 allows for multiple modes of radio transmission, however the Wyze developers chose to operate using it's proprietary mode. In order to prepare for packet transmission using cc1310's proprietary mode, the radio must be set up with the CMD_PROP_RADIO_DIV_SETUP command. To transmit a packet the TI CC13x0 utilizes the CMD_PROP_TX_ADV command (Advanced Transmit Command) to allow for more flexible packet setup outlined in [1].

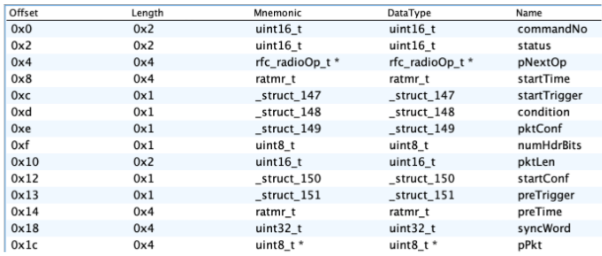


1 bit to 32 bytes or repetition	8 to 32 bits	0 to 32 bits	0 to 8 bytes	Arbitrary	0 or 16 bits (0 to 32 bits)
Preamble	Sync word	Header	Address	Payload	CRC

Advanced Packet Format [1]

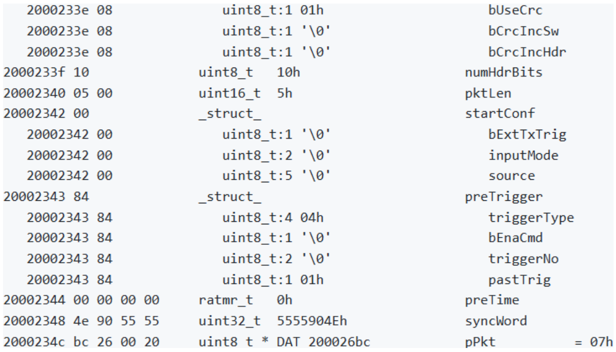
The Wyze developers chose to have the preamble consists of a fixed 2-byte sequence, with the first bit set to 0, ensuring proper synchronization between the transmitter and receiver. The sync word is a 32-bit value set to `0x5555904E`, which serves as a unique identifier that allows the packet engine to detect the start of the packet. Following the sync word, a 16-bit header is included to carry essential metadata required for processing the packet. The length information is specified as 5 bytes, defining the size of the payload. The payload itself is set to the sequence `0x07, 0x08, 0x40, 0x00, 0x21`, representing the actual data being transmitted. The cyclic redundancy check (CRC) is appended to the end of the packet to verify its integrity during transmission and excludes sync word and header from the calculation.

Building off the previous semester's work, references to the rfc_CMD_PROP_TX_ADV_s command structure was used to identify functions that corresponded to RF Core operations.



Offset	Length	Mnemonic	DataType	Name
0x0	0x2	uint16_t	uint16_t	commandNo
0x2	0x2	uint16_t	uint16_t	status
0x4	0x4	rfc_radioOp_t *	rfc_radioOp_t *	pNextOp
0x8	0x4	ratmr_t	ratmr_t	startTime
0xc	0x1	_struct_147	_struct_147	startTrigger
0xd	0x1	_struct_148	_struct_148	condition
0xe	0x1	_struct_149	_struct_149	pktConf
0xf	0x1	uint8_t	uint8_t	numHdrBits
0x10	0x2	uint16_t	uint16_t	pktLen
0x12	0x1	_struct_150	_struct_150	startConf
0x13	0x1	_struct_151	_struct_151	preTrigger
0x14	0x4	ratmr_t	ratmr_t	preTime
0x18	0x4	uint32_t	uint32_t	syncWord
0x1c	0x4	uint8_t *	uint8_t *	pPkt

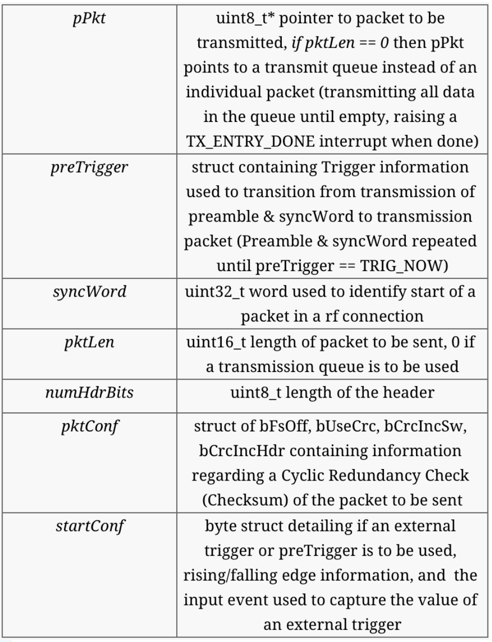
Figure 7a: rfc_CMD_PROP_TX_ADV_s Command Fields:



```
   2000233e 08            uint8_t:1 01h                      bUseCrc
   2000233e 08            uint8_t:1 '\0'                     bCrcIncSw
   2000233e 08            uint8_t:1 '\0'                     bCrcIncHdr
2000233f 10            uint8_t   10h                      numHdrBits
20002340 05 00         uint16_t  5h                       pktLen
20002342 00            _struct_                           startConf
   20002342 00            uint8_t:1 '\0'                     bExtTxTrig
   20002342 00            uint8_t:2 '\0'                     inputMode
   20002342 00            uint8_t:5 '\0'                     source
20002343 84            _struct_                           preTrigger
   20002343 84            uint8_t:4 04h                      triggerType
   20002343 84            uint8_t:1 '\0'                     bEnaCmd
   20002343 84            uint8_t:2 '\0'                     triggerNo
   20002343 84            uint8_t:1 01h                      pastTrig
20002344 00 00 00 00   ratmr_t   0h                       preTime
20002348 4e 90 55 55   uint32_t  5555904Eh                syncWord
2000234c bc 26 00 20   uint8 t * DAT 200026bc             pPkt         = 07h
```

Figure 7b: rfc_CMD_PROP_TX_ADV_s Struct Values Sample

Certain fields of this command structure were important to identify as they corresponded to important packet processing functionality. Using pPkt we can idenfy the data being sent between modules, preTrigger is helpful in determining the length of preamble bytes and where the syncword starts in packet transmission, pktLen / numHdrBits we can use to identify the length of payload / header data in a capture of RF traffic, and pktConf & startConf are useful in determining how the specific packet was manufactured and what the redundancy checks encompass.



pPkt	uint8_t* pointer to packet to be transmitted, *if pktLen == 0* then pPkt points to a transmit queue instead of an individual packet (transmitting all data in the queue until empty, raising a TX_ENTRY_DONE interrupt when done)
preTrigger	struct containing Trigger information used to transition from transmission of preamble & syncWord to transmission packet (Preamble & syncWord repeated until preTrigger == TRIG_NOW)
syncWord	uint32_t word used to identify start of a packet in a rf connection
pktLen	uint16_t length of packet to be sent, 0 if a transmission queue is to be used
numHdrBits	uint8_t length of the header
pktConf	struct of bFsOff, bUseCrc, bCrcIncSw, bCrcIncHdr containing information regarding a Cyclic Redundancy Check (Checksum) of the packet to be sent
startConf	byte struct detailing if an external trigger or preTrigger is to be used, rising/falling edge information, and the input event used to capture the value of an external trigger

Description of packet fields [2]

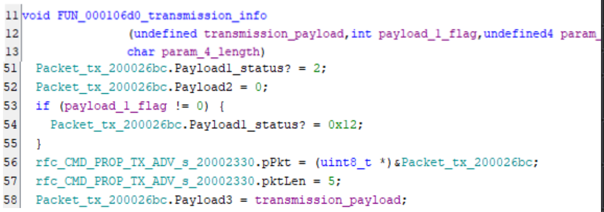
As shown in the figure above, pPkt is a pointer to the packet to be transmitted. This packet structure is focused as part of the team's effort to understand the TX structure.

As shown in Figure 7b, pktLen's value is 5h which defines packet's length as 5 bytes. numHdrBits' value is 10h, indicating that Header is comprised of 2 bytes. According to the manual, first bytes of the buffer pointed to by the pPkt are header bytes [1]. Thus, first 2 bytes of the buffer pointed by pPkt are header bytes. Since the packet consists of 5 bytes, next 3 bytes consist of payload bytes.

Analysis of Function FUN_0000c7b8

This function appears in the external references of the status field when **PROP_DONE_OK** signals successful operation, indicating normal completion. The main function initializes, validates, and schedules tasks or packets in a real-time system, ensuring integrity and responsiveness. It calculates timing using `FUN_00014c3c` and `FUN_000145d0`, stores results in the task structure, and disables interrupts during critical operations. Tasks are queued via `FUN_00014200`, which updates timing and calls `FUN_00011684_CMP` to insert tasks into a sorted queue. `FUN_000112a8_check0_TX` validates the next task, prioritizing the one with the earliest timing. Future research into `FUN_000112a8_check0_TX` is necessary to fully understand its functionality.

Analysis of Function FUN_000106d0_transmission_info



```
11 void FUN_000106d0_transmission_info
12                (undefined transmission_payload,int payload_1_flag,undefined4 param_
13                char param_4_length)
51   Packet_tx_200026bc.Payload1_status? = 2;
52   Packet_tx_200026bc.Payload2 = 0;
53   if (payload_1_flag != 0) {
54     Packet_tx_200026bc.Payload1_status? = 0x12;
55   }
56   rfc_CMD_PROP_TX_ADV_s_20002330.pPkt = (uint8_t *)&Packet_tx_200026bc;
57   rfc_CMD_PROP_TX_ADV_s_20002330.pktLen = 5;
58   Packet_tx_200026bc.Payload3 = transmission_payload;
```

Figure 8: Sample of FUN_000106d0_transmission_info function

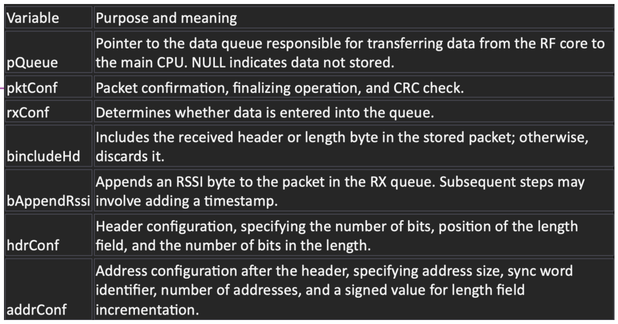
This function referenced the buffer memory locations pointed by the pPkt. The ultimate goal in this function was to understand the payloads. In Figure 8, lines 11-13 revealed that the function takes in 4 parameters. Investigating payload_1_flag (parameter 2) in line 53 as well as other functions that are referencing FUN_000106d0_transmission_info function revealed that the parameter 2 acted as flag that would determine the value of payload 1. Payload 2 is determined to be 0 in line 52.

Despite these findings, due to the limitations caused by the need to further investigate certain parts of this function to gain complete understanding, the future goal for this function includes further investigation of payload 3.

B. RX

One of the many functionalities the sensors in the device have called micro-controllers include creating messages. In the receiving part (RX), various elements, like managing addresses and handling the reception queue, make sure the communication between the device and the system is smooth and trustworthy. The data queue acts like a conveyor belt, moving packages (messages) between the radio frequency core (RF CC1310) and the main brain of your device, the CPU.

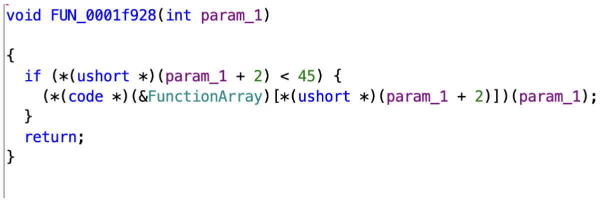
As the message travels through the RX chain, which is like a series of stops, certain parts of the message are removed. This ”stripping” process isn't limited to just between the CPU and RF; it happens at different points within the device. It's akin to opening a package, extracting what's necessary, and passing along only the vital information. For the semester, our goals is to understand how we found data structures and be able to point to the first entry of queue, code, etc, but also extend this knowledge. By looking at different functions and understanding more and more about the device and how it works, we can we can have a better understanding for our future goal, which is using Scapy to spoof the system.



Variable	Purpose and meaning
pQueue	Pointer to the data queue responsible for transferring data from the RF core to the main CPU. NULL indicates data not stored.
pktConf	Packet confirmation, finalizing operation, and CRC check.
rxConf	Determines whether data is entered into the queue.
bincludeHd	Includes the received header or length byte in the stored packet; otherwise, discards it.
bAppendRssi	Appends an RSSI byte to the packet in the RX queue. Subsequent steps may involve adding a timestamp.
hdrConf	Header configuration, specifying the number of bits, position of the length field, and the number of bits in the length.
addrConf	Address configuration after the header, specifying address size, sync word identifier, number of addresses, and a signed value for length field incrementation.

Figure 9: rfc_CMD_PROP_RX_ADV_s Command Fields

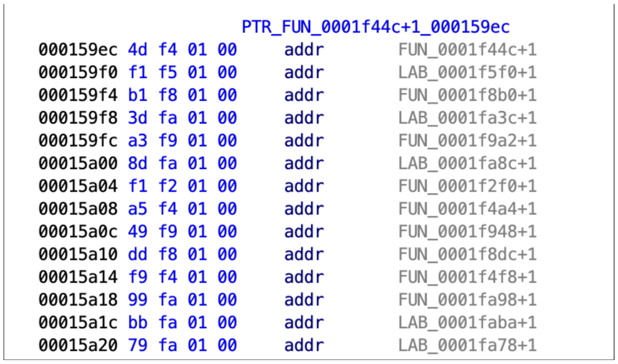
Analysis of FunctionArray



```
void FUN_0001f928(int param_1)

{
  if (*(ushort *)(param_1 + 2) < 45) {
    (*(code *)(&FunctionArray)[*(ushort *)(param_1 + 2)])(param_1);
  }
  return;
}
```

Fig. 10: FunctionArray array being called under FUN_0001f928

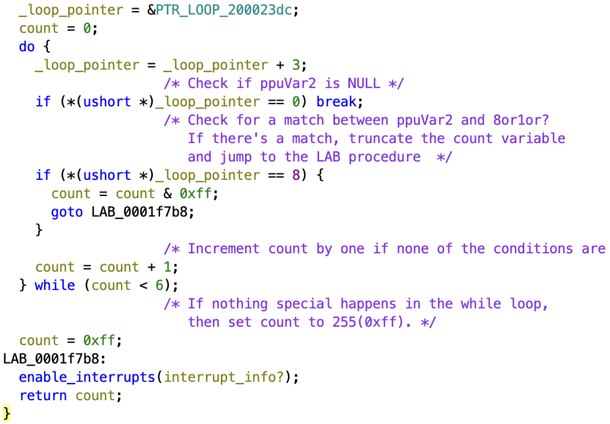
The **FunctionArray** is a collection of function pointers, and its behavior is determined by a specific control function, **FUN_0001f928**. This function first ensures that a calculated condition is met before proceeding to execute one of the functions from the array. The selection process relies on an index derived from another function, **AddressCount**, which computes the index by iterating through a controlled loop. This index determines one of six possible outcomes, each associated with specific entries in the **FunctionArray**. By structuring this logic, the system dynamically selects and executes operations based on parameters passed during runtime. The function AddressCount calculates the index by iterating through a loop that increments local variable *count* up to 6 times, in which it terminates at the end. This way index becomes an integer that has a maximum value of 6. After this, it gets passed into FUN_0001f928 by multiplying it's index value by 3. This gives us 6 possible arrays inside of **FunctionArray** array: 5, 8, 11, 14, 17, 20.



```
                    PTR_FUN_0001f44c+1_000159ec
000159ec 4d f4 01 00     addr       FUN_0001f44c+1
000159f0 f1 f5 01 00     addr       LAB_0001f5f0+1
000159f4 b1 f8 01 00     addr       FUN_0001f8b0+1
000159f8 3d fa 01 00     addr       LAB_0001fa3c+1
000159fc a3 f9 01 00     addr       FUN_0001f9a2+1
00015a00 8d fa 01 00     addr       LAB_0001fa8c+1
00015a04 f1 f2 01 00     addr       FUN_0001f2f0+1
00015a08 a5 f4 01 00     addr       FUN_0001f4a4+1
00015a0c 49 f9 01 00     addr       FUN_0001f948+1
00015a10 dd f8 01 00     addr       FUN_0001f8dc+1
00015a14 f9 f4 01 00     addr       FUN_0001f4f8+1
00015a18 99 fa 01 00     addr       FUN_0001fa98+1
00015a1c bb fa 01 00     addr       LAB_0001faba+1
00015a20 79 fa 01 00     addr       LAB_0001fa78+1
```

Fig. 11: Examples from FunctionArray

So far the expected *index* value is 0 because the **AddressCount** function's goal is to count how many times a while loop is iterated and terminates in under one specific condition. Ultimately this leads to count becoming 0 and the loop ending; applying the integer into our local variable *index*.

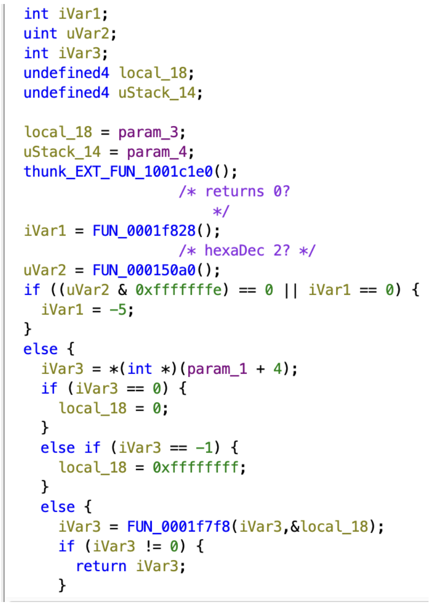


```
  _loop_pointer = &PTR_LOOP_200023dc;
  count = 0;
  do {
    _loop_pointer = _loop_pointer + 3;
                    /* Check if ppuVar2 is NULL */
    if (*(ushort *)_loop_pointer == 0) break;
                    /* Check for a match between ppuVar2 and 8or1or?
                       If there's a match, truncate the count variable
                       and jump to the LAB procedure  */
    if (*(ushort *)_loop_pointer == 8) {
      count = count & 0xff;
      goto LAB_0001f7b8;
    }
                    /* Increment count by one if none of the conditions are
    count = count + 1;
  } while (count < 6);
                    /* If nothing special happens in the while loop,
                       then set count to 255(0xff). */
  count = 0xff;
LAB_0001f7b8:
  enable_interrupts(interrupt_info?);
  return count;
}
```

Fig. 11.5: A majority of the function *AddressCount* which shows how the index was found

As seen from here, the index value is found by iterating through the function *AddressCount*. The value *_loop_pointer* is first assigned the memory address DAT_200023dc. The goal memory address DAT_200023e8(integer value 8) is successfully assigned after moving 4 bytes by the line where loop pointer variable is added integer 3.

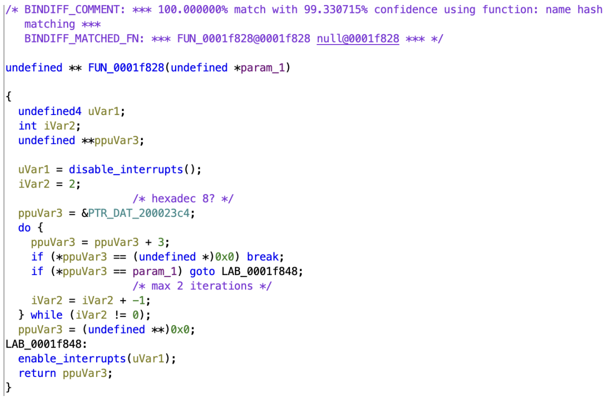
Overall, the method *fun_Array* will be dived in on array 10 with the parameter 8.



```
int iVar1;
uint uVar2;
int iVar3;
undefined4 local_18;
undefined4 uStack_14;

local_18 = param_3;
uStack_14 = param_4;
thunk_EXT_FUN_1001c1e0();
                  /* returns 0?
                     */
iVar1 = FUN_0001f828();
                  /* hexaDec 2? */
uVar2 = FUN_000150a0();
if ((uVar2 & 0xfffffffe) == 0 || iVar1 == 0) {
  iVar1 = -5;
}
else {
  iVar3 = *(int *)(param_1 + 4);
  if (iVar3 == 0) {
    local_18 = 0;
  }
  else if (iVar3 == -1) {
    local_18 = 0xffffffff;
  }
  else {
    iVar3 = FUN_0001f7f8(iVar3,&local_18);
    if (iVar3 != 0) {
      return iVar3;
    }
```

Figure 12: Sample of FUN_0001f4f8 function

Function FUN_0001f4f8 is the expected method being produced from *FunArray* (index 10). So far variable 1 and 2 have been reversed engineered. Var3 is 8 (found from the address). This function, FUN_0001f828, searches through a predefined list of pointers (PTR_DAT_200023c4) to find a match for the input parameter param_1. It temporarily disables interrupts during the search for thread safety and re-enables them before returning. If a match is found within a maximum of two iterations, a pointer to the matched entry is returned. Otherwise, it returns NULL, indicating that the input parameter was not found in the list. Var3 is hexadecimal 8 by finding the value located at memory address DAT_20003aec.

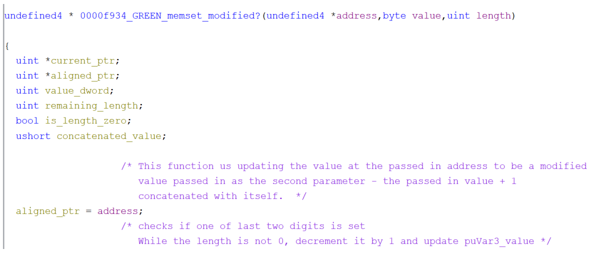


```
/* BINDIFF_COMMENT: *** 100.000000% match with 99.330715% confidence using function: name hash
   matching ***
   BINDIFF_MATCHED_FN: *** FUN_0001f828@0001f828 null@0001f828 *** */

undefined ** FUN_0001f828(undefined *param_1)

{
  undefined4 uVar1;
  int iVar2;
  undefined **ppuVar3;

  uVar1 = disable_interrupts();
  iVar2 = 2;
                    /* hexadec 8? */
  ppuVar3 = &PTR_DAT_200023c4;
  do {
    ppuVar3 = ppuVar3 + 3;
    if (*ppuVar3 == (undefined *)0x0) break;
    if (*ppuVar3 == param_1) goto LAB_0001f848;
                    /* max 2 iterations */
    iVar2 = iVar2 + -1;
  } while (iVar2 != 0);
  ppuVar3 = (undefined **)0x0;
LAB_0001f848:
  enable_interrupts(uVar1);
  return ppuVar3;
}
```

Fig. 13: Function FUN_0001f828 used for finding Var1

This function, FUN_0001f828, searches through a predefined list of pointers (PTR_DAT_200023c4) to find a match for the input parameter param_1. It temporarily disables interrupts during the search for thread safety and re-enables them before returning. If a match is found within a maximum of two iterations, a pointer to the matched entry is returned. Otherwise, it returns NULL, indicating that the input parameter was not found in the list.

Analysis of GREEN_memset_modified



```
undefined4 * 0000f934_GREEN_memset_modified?(undefined4 *address,byte value,uint length)

{
  uint *current_ptr;
  uint *aligned_ptr;
  uint value_dword;
  uint remaining_length;
  bool is_length_zero;
  ushort concatenated_value;

                    /* This function us updating the value at the passed in address to be a modified
                       value passed in as the second parameter - the passed in value + 1
                       concatenated with itself.  */
  aligned_ptr = address;
                    /* checks if one of last two digits is set
                       While the length is not 0, decrement it by 1 and update puVar3_value */
```

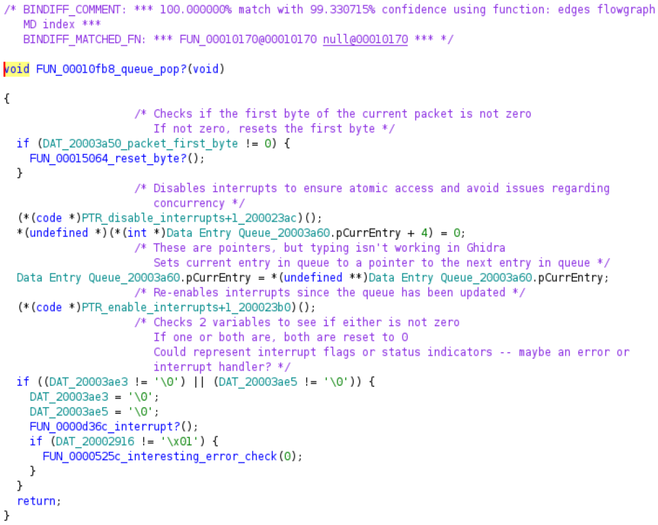
Fig. 12: The definition of the GREEN_memset_modified function.

The function **GREEN_memset_modified** takes a starting memory address, a value, and a length. The function fills the memory space with a modified version of the provided **value** that was passed in. The **value** argument ends up being concatenated and expanded. It also aligns memory writes for larger blocks of data efficiently.

The function is like a custom implementation of **memset** with modifications. It sets the memory at a specified address to a "modified" version of the value provided. Specifically, the value is transformed into a "concatenated" form (e.g., value + 1 concatenated with itself), then repeatedly written to the memory in chunks of different sizes for performance optimization.

The function is optimized for performance in multiple ways. First, it aligns the address to a 4-byte boundary that minimizes misaligned memory access, which is slower. It also writes in chunks (16, 8, 4 bytes) so it ends up minimizing the number of memory write operations. Finally, the function efficiently determines how many bytes are left to be written and then processes them in descending order of chunk size.

Analysis of the Queue Pop



```
/* BINDIFF_COMMENT: *** 100.000000% match with 99.330715% confidence using function: edges flowgraph
   MD index ***
   BINDIFF_MATCHED_FN: *** FUN_00010170@00010170 null@00010170 *** */

void FUN_00010fb8_queue_pop?(void)

{
                    /* Checks if the first byte of the current packet is not zero
                       If not zero, resets the first byte */
  if (DAT_20003a50_packet_first_byte != 0) {
    FUN_00015064_reset_byte?();
  }
                    /* Disables interrupts to ensure atomic access and avoid issues regarding
                       concurrency */
  (*(code *)PTR_disable_interrupts+1_200023ac)();
  *(undefined *)(*(int *)Data Entry Queue_20003a60.pCurrEntry + 4) = 0;
                    /* These are pointers, but typing isn't working in Ghidra
                       Sets current entry in queue to a pointer to the next entry in queue */
  Data Entry Queue_20003a60.pCurrEntry = *(undefined **)Data Entry Queue_20003a60.pCurrEntry;
                    /* Re-enables interrupts since the queue has been updated */
  (*(code *)PTR_enable_interrupts+1_200023b0)();
                    /* Checks 2 variables to see if either is not zero
                       If one or both are, both are reset to 0
                       Could represent interrupt flags or status indicators -- maybe an error or
                       interrupt handler? */
  if ((DAT_20003ae3 != '\0') || (DAT_20003ae5 != '\0')) {
    DAT_20003ae3 = '\0';
    DAT_20003ae5 = '\0';
    FUN_0000d36c_interrupt?();
    if (DAT_20002916 != '\x01') {
      FUN_0000525c_interesting_error_check(0);
    }
  }
  return;
}
```

Fig. 13: A screenshot of the queue pop function in Ghidra with comments.

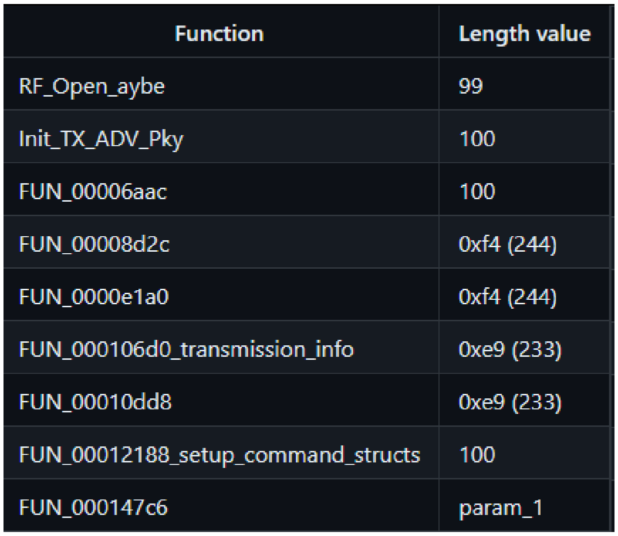
The above function, known as **FUN00010fb8_queue_pop?**, works to "pop" or advance a queue. After checking the first byte in the packet, it then disables interrupts to avoid concurrency issues and ensure atomic access. The queue is advanced and interrupts are re-enabled. Lastly, it performs some sort of error or interrupt handling with the use of two status indicators or flags.

Analysis of the RF Setup Function

One function, **FUN_00004f58_setup_rf_system**, works to initialize the RF system. It sets up most of this system, including global pointers, data queue entries, and certain function calls, including the frequency, chip type, and sync words.

After performing several calls to error-checking and interrupt handling functions, this setup function initializes data fields and pointers. It then calls several other functions related to setup: **FUN_00012188_setup_command_structs**, **FUN_00014180_set_global_chip_type_and_power**, and **FUN_00012140_set_synthesizer_frequency**. These functions set up the command structs (in-

cluding the pointers to the RX queue and output structure and the sync words), declare the chip type and power, and calculate the synthesizer frequency, respectively. Specifically, the frequency must be as close to possible to:
$(frequency + fractFreq/65536)$ MHz
where frequency is the synthesizer frequency and fractFreq is the fractional part of the frequency.



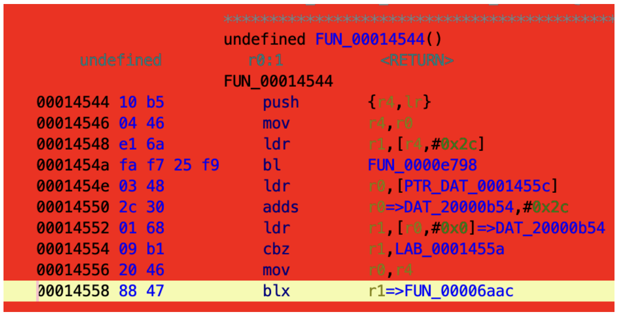
Function	Length value
RF_Open_aybe	99
Init_TX_ADV_Pky	100
FUN_00006aac	100
FUN_00008d2c	0xf4 (244)
FUN_0000e1a0	0xf4 (244)
FUN_000106d0_transmission_info	0xe9 (233)
FUN_00010dd8	0xe9 (233)
FUN_00012188_setup_command_structs	100
FUN_000147c6	param_1

Fig. 14: A table containing the functions that call the memcpy function and each value for length.

Additionally, the RF setup function contains a call to **FUN_000121d0_memcpy_something**. This function seems to do something related to a memory copy and length comparison. Its first parameter is the length, which could vary depending on which function calls **FUN_000121d0_memcpy_something**. The table above shows each function that calls it and the value that is passed into the length parameter. **FUN_000147c6** passes in its first parameter to the length value, which had three separate occurrences: 234, 235, and 240.

Analysis of FUN_00006aac

The function **FUN_00006aac** appears to be one that affects both the RX and TX systems in different ways. While a complete analysis of this function has not been completed, an sufficient amount of work has been done so that a hypothesis can be formed.



```
                         ****************************************************
                         undefined FUN_00014544()
       undefined            r0:1              <RETURN>
                         FUN_00014544
00014544 10 b5              push           {r4,lr}
00014546 04 46              mov            r4,r0
00014548 e1 6a              ldr            r1,[r4,#0x2c]
0001454a fa f7 25 f9        bl             FUN_0000e798
0001454e 03 48              ldr            r0,[PTR_DAT_0001455c]
00014550 2c 30              adds           r0=>DAT_20000b54,#0x2c
00014552 01 68              ldr            r1,[r0,#0x0]=>DAT_20000b54
00014554 09 b1              cbz            r1,LAB_0001455a
00014556 20 46              mov            r0,r4
00014558 88 47              blx            r1=>FUN_00006aac
```

Fig 15.1: FUN_00014544 calls on FUN_00006aac

As shown in the above image, the value of **param_1** is the immediate value getting passed into **FUN_00006aac**. This made it critical to understand both the contents and purpose of the parameter. Parameter 1 ended up being a value that indirectly modified other values within other functions. Once inside the function, the original value of **param_1** is not used in any meaningful capacity.

```
if (uVar3 == 0) {
  sVar1 = *(short *)(unaff_r6 + 8);
  PTR_Pkt_Data = *(Packet_tx **)(unaff_r6 + 4);
  DAT_200026b8 = unaff_r6;
  FUN_000121d0_memcpy_something(100,4);
  rfc_CMD_PROP_TX_ADV_s_20002330.pktLen = sVar1 + 2;
  uVar6 = (uint)rfc_CMD_PROP_TX_ADV_s_20002330.pktLen;
  FUN_000121d0_memcpy_something(0xe9,5);
  if (in_stack_00000005 == '\x01') {
    iVar5 = 2;
    bVar2 = (byte)((uVar6 << 0x15) >> 0x1d) | 0x18;
  }
  else {
    iVar5 = 4;
    uVar6 = (uint)(ushort)(sVar1 + 4);
    bVar2 = (byte)((uVar6 << 0x15) >> 0x1d) | 8;
  }
  PTR_Pkt_Data[-1].Payload3 = bVar2;
  rfc_CMD_PROP_TX_ADV_s_20002330.pPkt = &PTR_Pkt_Data[-1].Payload2;
  *rfc_CMD_PROP_TX_ADV_s_20002330.pPkt = (char)uVar6;
  FUN_0000f21c_big_to_little_endian(PTR_Pkt_Data,uVar6 - iVar5 & 0xffff);
  if (in_stack_00000004 == '\0') {
    rfc_CMD_PROP_TX_ADV_s_20002330.syncWord = 0x5555904e;
    rfc_CMD_PROP_RX_ADV_s_20002378.syncWord0 = rfc_CMD_PROP_TX_ADV_s_20002330.syncWord;
  }
  else {
    rfc_CMD_PROP_TX_ADV_s_20002330.syncWord = 0x55557a0e;
    rfc_CMD_PROP_RX_ADV_s_20002378.syncWord0 = rfc_CMD_PROP_TX_ADV_s_20002330.syncWord;
  }
}
```

Fig 15.2: Shows where in FUN_00006aac the RX and TX packets are modified

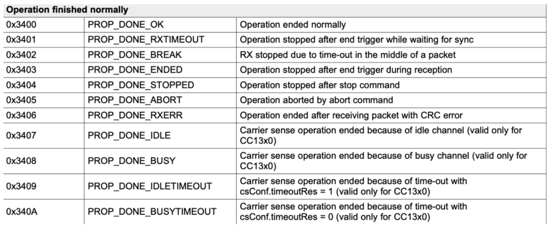
After traversing through the code, the system reach's the above chunk of code. The value of uVar3, which is set by the function **FUN_000065f4** not shown above, is checked to see if its 0. The assumption made is that if the value is not 0, an error has occurred during the process. If the value is 0, then the information used is correct. This will allow the RX and TX systems to properly receive the information. It is currently unknown what exactly happens within the RX and TX systems. The team hopes to find out more about the contents of the function in future semesters.

Packet Processing

The **00002520_packet_processing** function works on the current packet that is at the top of the **RX_queue**. It was determined to be a function of interest since it modifies values in and calculates values from a packet in the **RX_queue**. The packet processing section is divided into four main sections, which were determined based on the locations in the function that have labels. At certain labels within the function, the local variables are reset to the values stored in corresponding global variables. These labels, each associated with a distinct phase of the function's behavior (error checking, setup, processing, error handling,), divide the function into its four primary processes.

The team initially used Ghidra to reverse engineer the function code, before then transitioning to stepping through the code to reveal further information about its behavior. This was achieved by first manually recording values in an Excel spreadsheet, then by using Ghidra's debugging tool to step through each line of code and track the values stored in registers. The team then transitioned to using JLink in order to step through the code as it was actively running on the camera using the remote test bed feature. This method allowed the team to confirm the behavior of the camera in real conditions. It also allowed for easier memory access and management using the Write feature of JLink, which allowed the team to directly write values into the camera's memory, and the WReg feature, which allowed the team to manipulate register values. Using a combination of these methods, the team was able to determine the following information about the **00002520_packet_processing** function. The team was able to analyze the overall structure of the function, which is as follows:

1. Error Checking
   The first section of the function ensures that the radio is operating as expected before any further work on the packet is completed. In order to do this, it verifies based on expected status codes and halt codes. If the radio status is set to OK, the function will continue to operate as normal. This is checked by verifying that the **status** attribute of the **rfc_CMD_PROP_RX_ADV_s_2000237** is set to **0x3400**, which corresponds to the **PROP_DONE_OK** status. However, if the status is set to an error or failure mode, it will give control to the last section of the function, which handles errors. The following image shows all possible error codes and their corresponding status:



Operation finished normally		
0x3400	PROP_DONE_OK	Operation ended normally
0x3401	PROP_DONE_RXTIMEOUT	Operation stopped after end trigger while waiting for sync
0x3402	PROP_DONE_BREAK	RX stopped due to time-out in the middle of a packet
0x3403	PROP_DONE_ENDED	Operation stopped after end trigger during reception
0x3404	PROP_DONE_STOPPED	Operation stopped after stop command
0x3405	PROP_DONE_ABORT	Operation aborted by abort command
0x3406	PROP_DONE_RXERR	Operation ended after receiving packet with CRC error
0x3407	PROP_DONE_IDLE	Carrier sense operation ended because of idle channel (valid only for CC13x0)
0x3408	PROP_DONE_BUSY	Carrier sense operation ended because of busy channel (valid only for CC13x0)
0x3409	PROP_DONE_IDLETIMEOUT	Carrier sense operation ended because of time-out with csConf.timeoutRes = 1 (valid only for CC13x0)
0x340A	PROP_DONE_BUSYTIMEOUT	Carrier sense operation ended because of time-out with csConf.timeoutRes = 0 (valid only for CC13x0)

Fig. 15: A complete list of possible radio statuses.

2. Set Up
   This next section of the function uses information from the packet's header to initialize variables that will be used to perform calculations later in the function. The packet's header contains information regarding the structure of the packet. This information includes the length of the packet, whether or not whitening is enabled, and the length of the CRC. The packet header is two bytes, with bits 0-10 specifying the length of the packet, bit 11 corresponding to whether or not whitening is applied, and bit 12 corresponding to the length of the CRC. Currently, the values contained in these fields for the packet in memory are as follows:
   Length: 0b101101, corresponding to a decimal value of 0d45
   Whitening: 1, meaning that whitening is applied
   CRC: 0, meaning that the CRC length is 4 bytes

3. Processing
   The next section of the function uses the values and variables that were set up and calculated in the earlier section of the function to actually perform operations on the packet payload. A few major functions that are called in this section of the packet include the **FUN00010fb8_queue_pop?** and the **0000f934_GREEN_edit_value** functions, both of which were also called in the RX section and have been discussed in detail earlier in the paper. In the **00002520_packet_processing** function, the **FUN00010fb8_queue_pop?** and the **0000f934_GREEN_edit_value** function perform the same roles that they do in the **RX** section: popping, or advancing, the first element of the queue, and writing a value to memory, respectively. Another major function in this section is **FUN_00011684_CMP**, which compares two values, which are passed

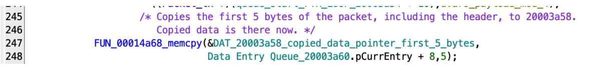
in as parameters to the function. This function modifies the values passed in based on the results of the comparison before returning them.

4. Error Handling
   The final section of this function serves to catch any errors that may occur during error checking, set up, or processing. It contains various handlers that may be called earlier in the function, and if they are called, control is diverted to this section of the packet processing function.

The team was also able to gain insight into how some values in the payload are used throughout the function. This provides information into how the values in the payload are determined, which may help the team craft payloads for fuzzing in future work.

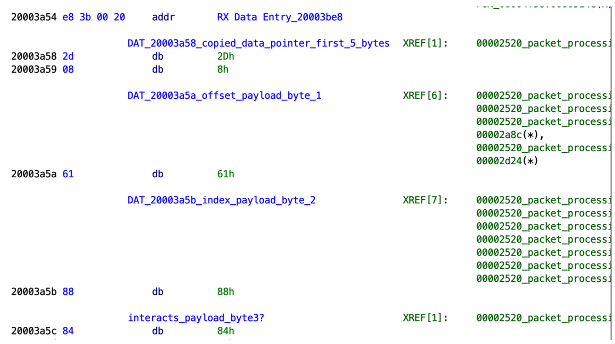
It has already been determined that the first two bytes of the packet, which make up the packet header, are bit-packed fields that contain information about the packet payload. This information is then used in the set up and processing sections of the function. Similarly, the first three bytes of the payload also appear to be bit-packed fields that are used to set values and determine the program's control flow throughout the function.

The first three bytes of the payload are copied from the packet to a separate global variable, as seen in the following two images:



```
                /* Copies the first 5 bytes of the packet, including the header, to 20003a58.
                   Copied data is there now. */
        FUN_00014a68_memcpy(&DAT_20003a58_copied_data_pointer_first_5_bytes,
                            Data Entry Queue_20003a60.pCurrEntry + 8,5);
```

Fig. 15: First five bytes of the packet copied to global memory.

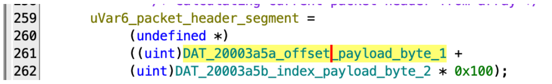


```
20003a54 e8 3b 00 20     addr       RX Data Entry_20003be8

                  DAT_20003a58_copied_data_pointer_first_5_bytes  XREF[1]:   00002520_packet_processi
20003a58 2d              db         2Dh
20003a59 08              db         8h

                  DAT_20003a5a_offset_payload_byte_1              XREF[6]:   00002520_packet_processi
                                                                             00002520_packet_processi
                                                                             00002520_packet_processi
                                                                             00002a8c(*),
                                                                             00002520_packet_processi
                                                                             00002d24(*)
20003a5a 61              db         61h

                  DAT_20003a5b_index_payload_byte_2               XREF[7]:   00002520_packet_processi
                                                                             00002520_packet_processi
                                                                             00002520_packet_processi
                                                                             00002520_packet_processi
                                                                             00002520_packet_processi
                                                                             00002520_packet_processi
                                                                             00002520_packet_processi
20003a5b 88              db         88h

                  interacts_payload_byte3?                        XREF[1]:   00002520_packet_processi
20003a5c 84              db         84h
```

Fig. 16: Location of first five bytes of packet in memory.

As seen in the image, the first two of these five bytes correspond to the header and have values of 0x2d and 0x08. Both these bytes have been discussed in the section of this paper that covers the set up portion of the packet processing function. The following three bytes are the first three bytes of the payload, and have values of 0x61, 0x88, and 0x84.

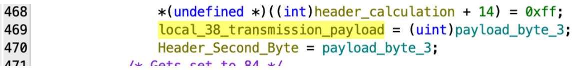
These three bytes have various uses throughout the function. The first two bytes of the payload are used as an index and offset for a relative memory address calculation, as seen in the following image:



```
        uVar6_packet_header_segment =
             (undefined *)
             ((uint)DAT_20003a5a_offset_payload_byte_1 +
             (uint)DAT_20003a5b_index_payload_byte_2 * 0x100);
```

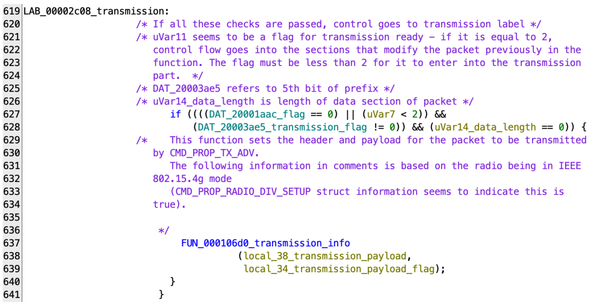
Fig. 17: First two bytes of payload used as index and offset for relative address calculation.

The third payload byte is also copied to the local variable local_38_transmission_payload, as seen in Fig 18. This local variable is passed in as the payload parameter to the FUN_000106d0_transmission_info function, which sets the header and payload for the packet to be transmitted by CMD_PROP_TX_ADV. This can be seen in Fig. 19.



```
              *(undefined *)((int)header_calculation + 14) = 0xff;
              local_38_transmission_payload = (uint)payload_byte_3;
              Header_Second_Byte = payload_byte_3;
```

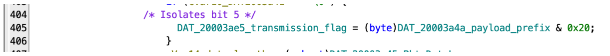
Fig. 18: local_38_transmission_payload variable is set to the third byte of the payload.



```
619 LAB_00002c08_transmission:
620                 /* If all these checks are passed, control goes to transmission label */
621                 /* uVar11 seems to be a flag for transmission ready - if it is equal to 2,
622                    control flow goes into the sections that modify the packet previously in the
623                    function. The flag must be less than 2 for it to enter into the transmission
624                    part.  */
625                 /* DAT_20003ae5 refers to 5th bit of prefix */
626                 /* uVar14_data_length is length of data section of packet */
627                      if ((((DAT_20001aac_flag == 0) || (uVar7 < 2)) &&
628                          (DAT_20003ae5_transmission_flag != 0)) && (uVar14_data_length == 0)) {
629                 /*    This function sets the header and payload for the packet to be transmitted
630                    by CMD_PROP_TX_ADV.
631                       The following information in comments is based on the radio being in IEEE
632                    802.15.4g mode
633                       (CMD_PROP_RADIO_DIV_SETUP struct information seems to indicate this is
634                    true).
635 
636                     */
637                        FUN_000106d0_transmission_info
638                                  (local_38_transmission_payload,
639                                   local_34_transmission_payload_flag);
640                      }
641                    }
```

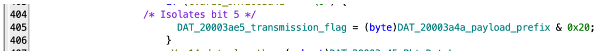
Fig. 19: Third byte of the payload is passed into the FUN_000106d0_transmission_info function as the payload parameter (line 637).

In addition to these uses, the first three payload bytes are also used as bit-packed fields, with individual bits in these bytes corresponding to different flags that determine the control flow in the program. For example the fifth bit of the first payload byte is isolated, as seen in Fig. 20, and then used as a flag that determines whether or not the FUN_000106d0_transmission_info function is called, as seen in Fig. 21.



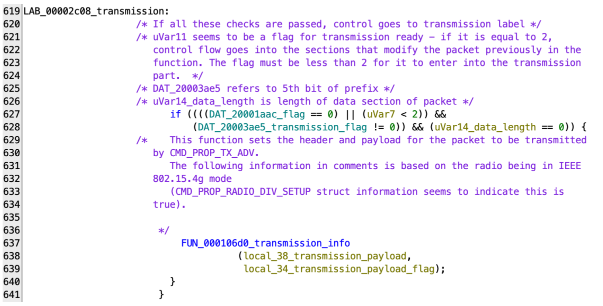
```
                /* Isolates bit 5 */
                   DAT_20003ae5_transmission_flag = (byte)DAT_20003a4a_payload_prefix & 0x20;
                 }
```

Fig. 20: Fifth bit of first payload byte being isolated and set to DAT_20003ae5_transmission_flag local variable.



```
                /* Isolates bit 5 */
                   DAT_20003ae5_transmission_flag = (byte)DAT_20003a4a_payload_prefix & 0x20;
                 }
```

Fig. 20: Fifth bit of first payload byte being isolated and set to DAT_20003ae5_transmission_flag local variable.



```
619 LAB_00002c08_transmission:
620                     /* If all these checks are passed, control goes to transmission label */
621                     /* uVar11 seems to be a flag for transmission ready - if it is equal to 2,
622                        control flow goes into the sections that modify the packet previously in the
623                        function. The flag must be less than 2 for it to enter into the transmission
624                        part.  */
625                     /* DAT_20003ae5 refers to 5th bit of prefix */
626                     /* uVar14_data_length is length of data section of packet */
627                           if ((((DAT_20001aac_flag == 0) || (uVar7 < 2)) &&
628                               (DAT_20003ae5_transmission_flag != 0)) && (uVar14_data_length == 0)) {
629                     /*    This function sets the header and payload for the packet to be transmitted
630                        by CMD_PROP_TX_ADV.
631                           The following information in comments is based on the radio being in IEEE
632                        802.15.4g mode
633                           (CMD_PROP_RADIO_DIV_SETUP struct information seems to indicate this is
634                        true).
635 
636                         */
637                             FUN_000106d0_transmission_info
638                                       (local_38_transmission_payload,
639                                        local_34_transmission_payload_flag);
640                           }
641                         }
```

Fig. 19: Fifth bit of first payload byte stored in the-DAT_20003ae5_transmission_flag local variable being used as transmission flag (line 628).

Future work on this function includes further exploring the ways that the first three bytes of the payload are used, particularly the individual bits in these three bytes. By crafting a clear idea of how these bytes are used, it will be possible to create payloads that interact with the camera in specified, controlled ways.

V. Conclusion

The purpose of this research is to explore how the Wyze camera receives, transmits, and processes packets. This information will allow the team to better understand the structure and semantic meaning of the information contained in the packet, which will then allow the team to spoof packets to send to the camera. By purposefully sending malformed packets, the team can analyze how the camera behaves on receiving unexpected inputs, which may provide insights into potential vulnerabilities in the camera's code and behavior.

A long term goal of this project is to eventually make the process of sending malformed packets and analyzing the camera's behavior automatic. This is a process called fuzzing. This process will allow the team to send packets and analyze the camera's behavior after receiving the packets faster and require less manual effort. Furthermore, since fuzzing is an automatic process, packets can continuously be sent to the camera, which means that a larger range of packets can be tested.

The final goal of this project is to compare the results of fuzzing the packets automatically and manually spoofing them. Since fuzzing is an automatic process, it has the potential to save time and effort from manually spoofing the packets. However, there is a lack of in-depth research that considers using fuzzing on RF data. If the results of fuzzing packets are comparable to manually spoofing them, it means that fuzzing may be a viable alternative to manual spoofing for RF data, which would save time and effort on reverse engineering.

VI. References

[1] Texas Instruments Inc., Dallas, TX, USA. *CC13x0, CC26x0 SimpleLink™ Wireless MCU Technical Reference Manual* (2015). Accessed: Oct. 6, 2024. [Online]. Available: `https://www.ti.com/lit/ug/swcu117i/swcu117i.pdf?ts=1597428862158`

[2] Texas Instruments Inc., Dallas, TX, USA. *SimpleLink™ Sub-1 GHz CC13x0 Software Development Kit*. Accessed: Oct. 6, 2024. Available: `https://www.ti.com/tool/SIMPLELINK-CC13X0-SDK`

[3] "Vulnerabilities Identified in Wyze Cam IoT Device," Bitdefender, `https://www.bitdefender.com/files/News/CaseStudies/study/413/Bitdefender-PR-Whitepaper-WCam-creat5991-en-EN.pdf`. Mar 2022. [accessed Oct. 7, 2024].

[4] A. J. Majumder, J. D. Miller, C. B. Veilleux and A. A. Asif, "Smart-Power: A Smart Cyber-Physical System to Detect IoT Security Threat through Behavioral Power Profiling," 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid, Spain, 2020, pp. 1041-1049, doi: 10.1109/COMPSAC48688.2020.0-135. `https://www.researchgate.net/profile/Akm-Jahangir-Majumder/publication/341323163_Smart-Power_A_Smart_Cyber-Physical_System_to_Detect_IoT_Security_Threat_through_Behavioral_Power_Profiling/links/5ec46d3a299bf1c09acbe3fc/Smart-Power-A-Smart-Cyber-Physical-System-to-Detect-IoT-Security-Threat-through-Behavioral-Power-Profiling.pdf`

[5] "Replay attack," Computer Security Resource Center, `https://csrc.nist.gov/glossary/term/replay_attack` [accessed Oct. 6, 2024].

[6] "What is signed firmware," Chipkin, `https://store.chipkin.com/articles/what-is-signed-firmware#:~:text=Signed%20firmware%20is%20a%20security,accepting%20and%20installing%20the%20firmware`. [accessed Oct. 6, 2024].

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