SWIFT Wireless Fire Alarm System Analysis

Drew Petry (Advisor)
Research Engineer II
Georgia Tech Research Institute
Atlanta, Georgia, United States
drew.petry@gtri.gatech.edu

Madelyn Novelli
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
mnovelli3@gatech.edu

Ky Tran
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
ktran323@gatech.edu

Garrett Brown (Advisor)
Research Scientist I
Georgia Tech Research Institute
Atlanta, Georgia, United States
garrett.brown@gtri.gatech.edu

Alexander Schoolcraft
Mike Cottrell College of Business and Technology
University of North Georgia
Dahlonega, Georgia, United States
adscho3199@ung.edu

Spencer Redelman
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
sredelman3@gatech.edu

Trey Durden
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
tdurden8@gatech.edu

James Ragazino
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
jragazino3@gatech.edu

Tony Tanory
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
ttanory3@gatech.edu

Liam Smith
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
lsmith398@gatech.edu

***Abstract*—The SWIFT (Smart Wireless Integrated Fire Technology) system, developed by Honeywell, includes various fire alarm devices such as wireless smoke and heat detectors, pull stations, alarms, monitoring software, and control panels. By utilizing wireless communication, SWIFT eliminates the need for extensive cabling, which simplifies installation and allows for more flexible device placement. However, wireless systems can be vulnerable to hacking or tampering if not properly secured. Unauthorized access could disrupt the system's functionality or trigger false alarms. This study seeks to examine the security weaknesses of the Honeywell SWIFT system through firmware reverse engineering and targeted attacks.**

I. System Introduction

The Honeywell SWIFT system represents a significant advancement in fire detection technology by incorporating wireless communication into a traditionally wired infrastructure. This system integrates various fire safety devices through its wireless mesh network, such as smoke detectors, heat sensors, and pull stations, offering flexibility in installation while maintaining high reliability in system performance. This wireless capability reduces the need for extensive cabling and allows for easier deployment in environments where running physical wiring is challenging or cost-prohibitive.

At the core of the system is the Signaling Line Circuit (SLC) processor, which acts as a critical intermediary between wireless devices and the wired Fire Alarm Control Panel (FACP) through the Wireless Gateway. The SLC processor manages communication across the system, translating signals from the wireless mesh network into formats the FACP can interpret and control. This ensures that wireless devices, such as smoke and heat detectors, seamlessly function alongside traditional wired components.

Complementing the SLC is the Radio Frequency (RF) processor, a key interface that bridges the gap between wireless devices and the system's wired infrastructure. The RF Gateway oversees the operation of the wireless mesh network, which uses wireless communication to ensure efficient data transmission between the devices and the FACP. This gateway plays a pivotal role in maintaining system integrity, managing wireless device activity, and relaying information to the SLC for processing.

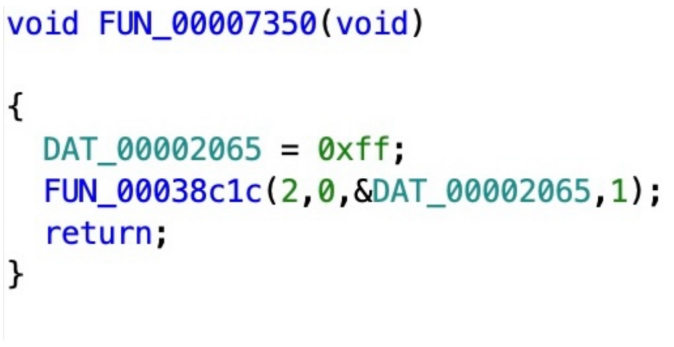
Despite these technological advancements, the integration of wireless communication introduces potential security risks. Reverse engineering both the SLC and RF Gateway is crucial for understanding the underlying firmware and communication protocols used by SWIFT. By analyzing how data is exchanged and processed, we can identify potential weaknesses. These investigations aim to highlight potential risks to the the SWIFT system's security.

II. Previous Research

The previous team delved into the AES encryption and decryption functions, as well as payload parsing and data integrity checks functions.

III. RF Gateway Reverse Engineering

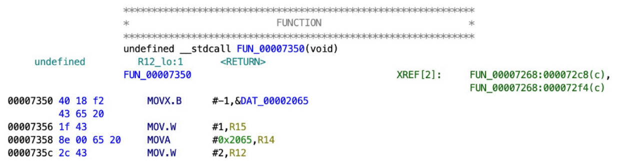
By examining WSG_RF_MCU_FW_Dump_v4_1_0x00000-0x45BFF.bin in Ghidra, insight was gained into how the compiler optimizes the space taken up by functions, and the functions that revealed this drew attention to another function that may be useful when implementing the backdoor attack on the RF gateway. This section will begin by examining how FUN_00007350 and FUN_0000735e relate to each other in a way that saves space in memory, and it will then elaborate on how those two functions relate to the potentially important function FUN_00038c1c.



```
void FUN_00007350(void)

{
  DAT_00002065 = 0xff;
  FUN_00038c1c(2,0,&DAT_00002065,1);
  return;
}
```

Fig. 1. Decompiled View of FUN_00007350

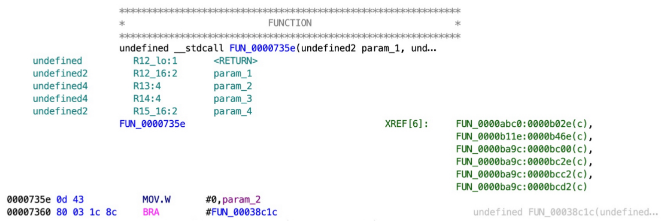
The decompiler view of FUN_00007350 shows that the function sets the byte at address 0x00002065 to be 0xff. Before calling FUN_00038c1c with the parameters shown. On the surface, this seems like a very simple function. However, more is revealed when looking at the assembly code for the function.



```
                    ****************************************************************
                    *                          FUNCTION                            *
                    ****************************************************************
                    undefined __stdcall FUN_00007350(void)
     undefined        R12_lo:1      <RETURN>
                    FUN_00007350                                XREF[2]:     FUN_00007268:000072c8(c),
                                                                             FUN_00007268:000072f4(c)
00007350 40 18 f2       MOVX.B      #-1,&DAT_00002065
         43 65 20
00007356 1f 43          MOV.W       #1,R15
00007358 8e 00 65 20    MOVA        #0x2065,R14
0000735c 2c 43          MOV.W       #2,R12
```

Fig. 2. Assembly Code for FUN_00007350

Note that FUN_00038c1c has four parameters, and these parameters are stored in the registers R12, R13, R14, and R15. Looking at the assembly code for FUN_00007350, it initially seems to be incompatible with the decompiler view. Setting the byte at address 0x00002065 to be 0xff occurs first, and the same parameters shown in the decompiler view were passed into R12, R14, and R15. However, passing 0 into R13 and calling FUN_00038c1c does not seem to happen, but there is also no statement returning back to where the function is called from. Due to this, a call to FUN_00007350 will result in the code right after, which is in FUN_0000735e, also being executed.



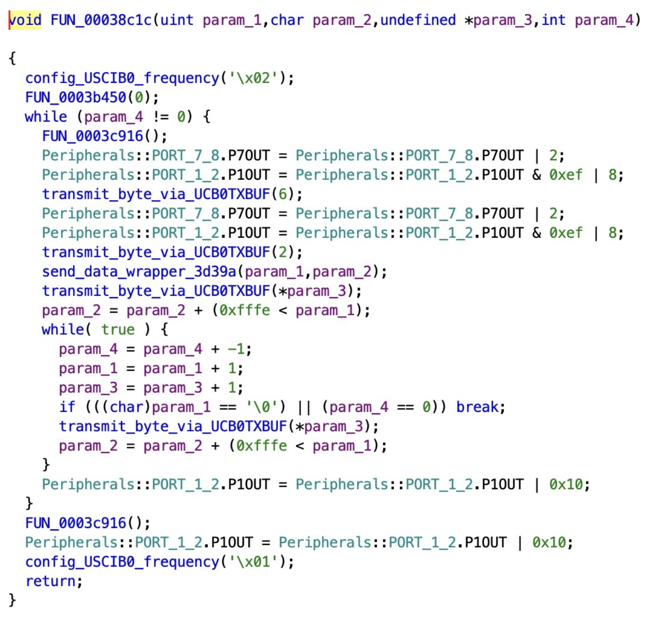
```
                    ****************************************************************
                    *                          FUNCTION                            *
                    ****************************************************************
                    undefined __stdcall FUN_0000735e(undefined2 param_1, und…
     undefined        R12_lo:1      <RETURN>
     undefined2       R12_16:2      param_1
     undefined4       R13:4         param_2
     undefined4       R14:4         param_3
     undefined2       R15_16:2      param_4
                    FUN_0000735e                                XREF[6]:     FUN_0000abc0:0000b02e(c),
                                                                             FUN_0000b11e:0000b46e(c),
                                                                             FUN_0000ba9c:0000bc00(c),
                                                                             FUN_0000ba9c:0000bc2e(c),
                                                                             FUN_0000ba9c:0000bcc2(c),
                                                                             FUN_0000ba9c:0000bcd2(c)
0000735e 0d 43          MOV.W       #0,param_2
00007360 80 03 1c 8c    BRA         #FUN_00038c1c                            undefined FUN_00038c1c(undefined…
```

Fig. 3. Assembly Code for FUN_0000735e

This code completes the operations shown by the decompiler view of FUN_00007350. 0 is placed into R13, which completes the filling of the parameters for FUN_00038c1c, and FUN_00038c1c is what is branched to and executed immediately after that. Based on this analysis, it appears that both FUN_00007350 and FUN_0000735e serve only to set the parameters for FUN_00038c1c. This can be seen by the fact that FUN_0000735e is called from six other functions, not including how it is, for all intents and purposes, called by FUN_00007350 not returning to any return address. While the first, third, and fourth parameters of FUN_00038c1c may be set in other functions, such as FUN_00007350, those other functions all most likely need the second parameter to be 0, which is likely why FUN_0000735e exists only to set that parameter and then call FUN_00038c1c. The compiler likely did this in order to save space in memory by having a single function to do this instead of writing it into every function that needs to use FUN_00038c1c. To conserve even more space, FUN_00007350 can exist without returning to some return address because it is right before FUN_0000735e, meaning that it can immediately proceed to executing the code in FUN_0000735e. Examining this portion of code has shown how efficient the compiler can be in terms of converting code into as few assembly instructions as possible. However, it also has drawn attention to FUN_00038c1c.

Based on its decompiler view, FUN_00038c1c appears to be an incredibly important function in the RF Gateway. This is because it is referenced 48 times by other functions, and there are also other functions throughout the gateway, such as FUN_00007350, that are entirely dedicated to setting the parameters of calls to FUN_00038c1c. Note that FUN_00038c1c has not been fully reverse engineered yet, but there are a few ideas that can be taken away from a cursory look at the decompiler view. The frequency of USCIB0 is changed to 0x02, data is transmitted, and the frequency of USCIB0 is changed to 0x01. The importance of the data being transmitted, the destination of that data, and the functions called along the way will be determined through further reverse engineering. What the team will find in FUN_00038c1c will hopefully be helpful in implementing the backdoor attack on the RF Gateway.

Upon further research and investigation, it appears that FUN_00038c1c appears to be a function that is indicating a message being passed between devices. We believe that the ports involved are the indicators that messages are being sent



```
void FUN_00038c1c(uint param_1,char param_2,undefined *param_3,int param_4)

{
  config_USCIB0_frequency('\x02');
  FUN_0003b450(0);
  while (param_4 != 0) {
    FUN_0003c916();
    Peripherals::PORT_7_8.P7OUT = Peripherals::PORT_7_8.P7OUT | 2;
    Peripherals::PORT_1_2.P1OUT = Peripherals::PORT_1_2.P1OUT & 0xef | 8;
    transmit_byte_via_UCB0TXBUF(6);
    Peripherals::PORT_7_8.P7OUT = Peripherals::PORT_7_8.P7OUT | 2;
    Peripherals::PORT_1_2.P1OUT = Peripherals::PORT_1_2.P1OUT & 0xef | 8;
    transmit_byte_via_UCB0TXBUF(2);
    send_data_wrapper_3d39a(param_1,param_2);
    transmit_byte_via_UCB0TXBUF(*param_3);
    param_2 = param_2 + (0xfffe < param_1);
    while( true ) {
      param_4 = param_4 + -1;
      param_1 = param_1 + 1;
      param_3 = param_3 + 1;
      if (((char)param_1 == '\0') || (param_4 == 0)) break;
      transmit_byte_via_UCB0TXBUF(*param_3);
      param_2 = param_2 + (0xfffe < param_1);
    }
    Peripherals::PORT_1_2.P1OUT = Peripherals::PORT_1_2.P1OUT | 0x10;
  }
  FUN_0003c916();
  Peripherals::PORT_1_2.P1OUT = Peripherals::PORT_1_2.P1OUT | 0x10;
  config_USCIB0_frequency('\x01');
  return;
}
```

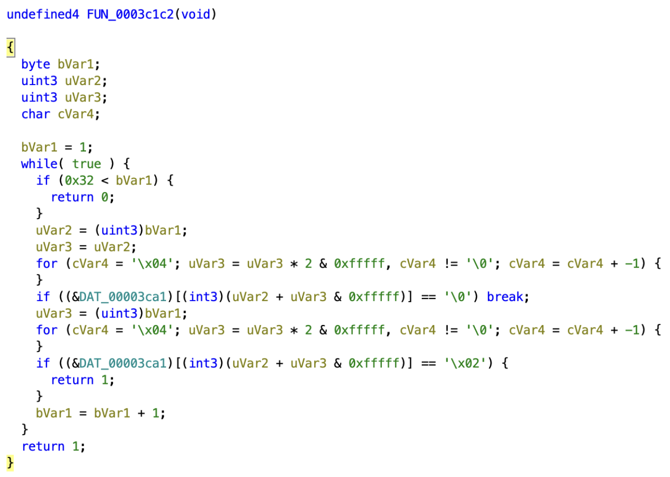
Fig. 4. Decompiled View of FUN_00038c1c

between devices.

We have also discovered FUN_0003c1c2. Upon reverse engineering, it appears that this function is performing some sort of check on a data table or buffer. Note that the function performs an iterative loop over &DAT_00003ca1. This data is the subject of this verification or check process. Furthermore, we found that this function is used as a boolean in FUN_00021884. When FUN_00003c1c2 returns 1, FUN_00021884 appears to print "FULL" to the console. This leads us to believe that FUN_0003c1c2 is indeed checking if &DAT_00003ca1 is full. The next step is to further investigate &DAT_00003ca1 and FUN_00021884 to gather a more complete understanding of what is being represented as "FULL" and why it is relevant.

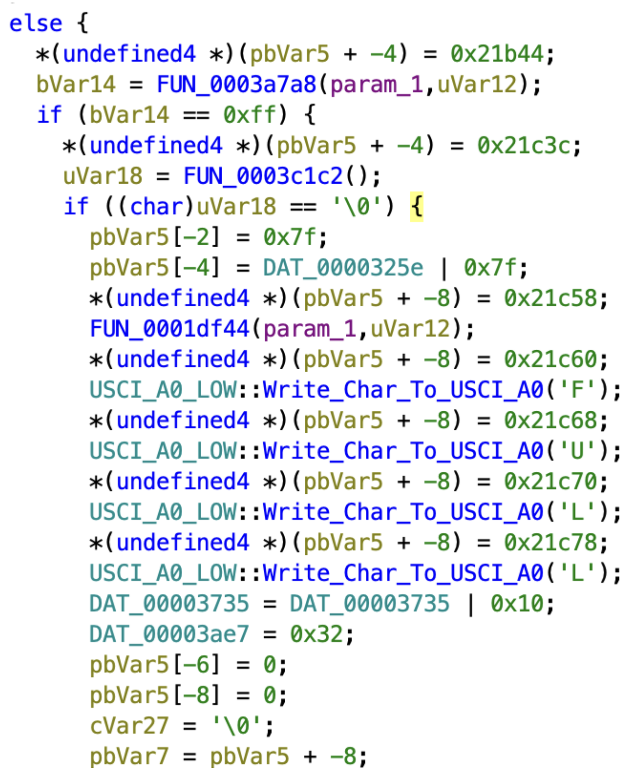
Another key area of focus is reverse engineering functions that interact with the SX 1231 Transceiver. For example, change_RF_mode. Each of these functions have an address that corresponds with the SX 1231 data sheet. This address shows us what instruction is taking place on the transceiver. In this case, x01 corresponds to changing the operating mode of the transceiver. Within this section of the data sheet, we are able to see that bits 4-2 of the parameter passed into this function decide which operating mode to change the transceiver to. There are 5 different modes: Sleep, Standby, Frequency Synthesizer, Transmitter, and Receiver Mode. All of the other combinations are reserved. When this function is called, it sets the operating mode of the transceiver to one of these modes before returning.

Another critical function we identified is FUN_0000e83a. This function is pivotal in managing firmware updates, verifying message integrity, and ensuring proper buffer handling via



```
undefined4 FUN_0003c1c2(void)

{
  byte bVar1;
  uint3 uVar2;
  uint3 uVar3;
  char cVar4;

  bVar1 = 1;
  while( true ) {
    if (0x32 < bVar1) {
      return 0;
    }
    uVar2 = (uint3)bVar1;
    uVar3 = uVar2;
    for (cVar4 = '\x04'; uVar3 = uVar3 * 2 & 0xfffff, cVar4 != '\0'; cVar4 = cVar4 + -1) {
    }
    if ((&DAT_00003ca1)[(int3)(uVar2 + uVar3 & 0xfffff)] == '\0') break;
    uVar3 = (uint3)bVar1;
    for (cVar4 = '\x04'; uVar3 = uVar3 * 2 & 0xfffff, cVar4 != '\0'; cVar4 = cVar4 + -1) {
    }
    if ((&DAT_00003ca1)[(int3)(uVar2 + uVar3 & 0xfffff)] == '\x02') {
      return 1;
    }
    bVar1 = bVar1 + 1;
  }
  return 1;
}
```

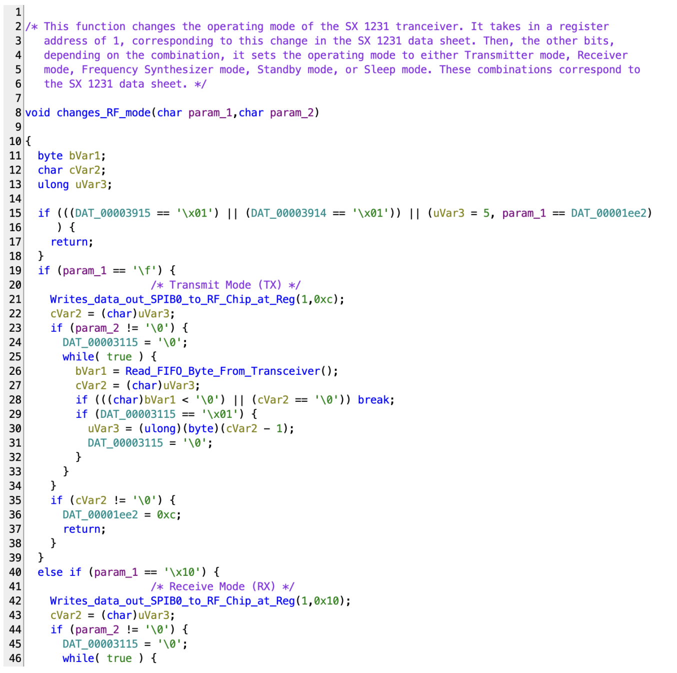
Fig. 5. Decompiled View of FUN_0003c1c2



```
else {
  *(undefined4 *)(pbVar5 + -4) = 0x21b44;
  bVar14 = FUN_0003a7a8(param_1,uVar12);
  if (bVar14 == 0xff) {
    *(undefined4 *)(pbVar5 + -4) = 0x21c3c;
    uVar18 = FUN_0003c1c2();
    if ((char)uVar18 == '\0') {
      pbVar5[-2] = 0x7f;
      pbVar5[-4] = DAT_0000325e | 0x7f;
      *(undefined4 *)(pbVar5 + -8) = 0x21c58;
      FUN_0001df44(param_1,uVar12);
      *(undefined4 *)(pbVar5 + -8) = 0x21c60;
      USCI_A0_LOW::Write_Char_To_USCI_A0('F');
      *(undefined4 *)(pbVar5 + -8) = 0x21c68;
      USCI_A0_LOW::Write_Char_To_USCI_A0('U');
      *(undefined4 *)(pbVar5 + -8) = 0x21c70;
      USCI_A0_LOW::Write_Char_To_USCI_A0('L');
      *(undefined4 *)(pbVar5 + -8) = 0x21c78;
      USCI_A0_LOW::Write_Char_To_USCI_A0('L');
      DAT_00003735 = DAT_00003735 | 0x10;
      DAT_00003ae7 = 0x32;
      pbVar5[-6] = 0;
      pbVar5[-8] = 0;
      cVar27 = '\0';
      pbVar7 = pbVar5 + -8;
```

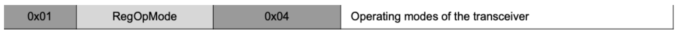
Fig. 6. Decompiled View of FUN_00021884

the ipc_tx_buff. It validates incoming data using a checksum (e.g., ipc_tx_buff[0x9f]), outputs -V to A0, the debug port, and prepares verified data for writing into flash memory. Specifically, it erases a flash segment at 0x1980 and writes new firmware or configuration data stored in ipc_tx_buff[0x9b] through 0x9e. This suggests that 0x1980 to 0x1983 may hold version-related information, though further investigation into how this function is utilized—particularly the relationship between 0x1980 to 0x1983 and bytes 155 to 159 (x9b to x9e) in the ipc_tx_buf—is needed to confirm this. Reversing this



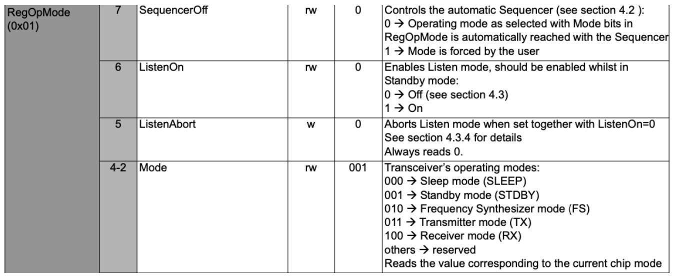
```
/* This function changes the operating mode of the SX 1231 tranceiver. It takes in a register
   address of 1, corresponding to this change in the SX 1231 data sheet. Then, the other bits,
   depending on the combination, it sets the operating mode to either Transmitter mode, Receiver
   mode, Frequency Synthesizer mode, Standby mode, or Sleep mode. These combinations correspond to
   the SX 1231 data sheet. */

void changes_RF_mode(char param_1,char param_2)

{
  byte bVar1;
  char cVar2;
  ulong uVar3;

  if (((DAT_00003915 == '\x01') || (DAT_00003914 == '\x01')) || (uVar3 = 5, param_1 == DAT_00001ee2)
     ) {
    return;
  }
  if (param_1 == '\f') {
                    /* Transmit Mode (TX) */
    Writes_data_out_SPIB0_to_RF_Chip_at_Reg(1,0xc);
    cVar2 = (char)uVar3;
    if (param_2 != '\0') {
      DAT_00003115 = '\0';
      while( true ) {
        bVar1 = Read_FIFO_Byte_From_Transceiver();
        cVar2 = (char)uVar3;
        if (((char)bVar1 < '\0') || (cVar2 == '\0')) break;
        if (DAT_00003115 == '\x01') {
          uVar3 = (ulong)(byte)(cVar2 - 1);
          DAT_00003115 = '\0';
        }
      }
    }
    if (cVar2 != '\0') {
      DAT_00001ee2 = 0xc;
      return;
    }
  }
  else if (param_1 == '\x10') {
                    /* Receive Mode (RX) */
    Writes_data_out_SPIB0_to_RF_Chip_at_Reg(1,0x10);
    cVar2 = (char)uVar3;
    if (param_2 != '\0') {
      DAT_00003115 = '\0';
      while( true ) {
```

Fig. 7. Decompiled view of changes_RF_mode



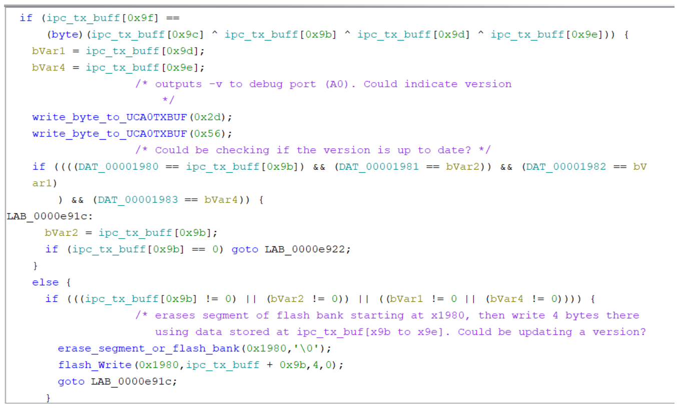
0x01	RegOpMode	0x04	Operating modes of the transceiver

Fig. 8. SX 1231 Data Sheet



RegOpMode (0x01)	7	SequencerOff	rw	0	Controls the automatic Sequencer (see section 4.2): 0 → Operating mode as selected with Mode bits in RegOpMode is automatically reached with the Sequencer 1 → Mode is forced by the user
	6	ListenOn	rw	0	Enables Listen mode, should be enabled whilst in Standby mode: 0 → Off (see section 4.3) 1 → On
	5	ListenAbort	w	0	Aborts Listen mode when set together with ListenOn=0 See section 4.3.4 for details Always reads 0.
	4-2	Mode	rw	001	Transceiver's operating modes: 000 → Sleep mode (SLEEP) 001 → Standby mode (STDBY) 010 → Frequency Synthesizer mode (FS) 011 → Transmitter mode (TX) 100 → Receiver mode (RX) others → reserved Reads the value corresponding to the current chip mode

Fig. 9. SX 1231 Data Sheet



```
  if (ipc_tx_buff[0x9f] ==
      (byte)(ipc_tx_buff[0x9c] ^ ipc_tx_buff[0x9b] ^ ipc_tx_buff[0x9d] ^ ipc_tx_buff[0x9e])) {
    bVar1 = ipc_tx_buff[0x9d];
    bVar4 = ipc_tx_buff[0x9e];
                    /* outputs -v to debug port (A0). Could indicate version
                        */
    write_byte_to_UCA0TXBUF(0x2d);
    write_byte_to_UCA0TXBUF(0x56);
                    /* Could be checking if the version is up to date? */
    if ((((DAT_00001980 == ipc_tx_buff[0x9b]) && (DAT_00001981 == bVar2)) && (DAT_00001982 == bV
    ar1)
        ) && (DAT_00001983 == bVar4)) {
LAB_0000e91c:
      bVar2 = ipc_tx_buff[0x9b];
      if (ipc_tx_buff[0x9b] == 0) goto LAB_0000e922;
    }
    else {
      if (((ipc_tx_buff[0x9b] != 0) || (bVar2 != 0)) || ((bVar1 != 0 || (bVar4 != 0)))) {
                    /* erases segment of flash bank starting at x1980, then write 4 bytes there
                       using data stored at ipc_tx_buf[x9b to x9e]. Could be updating a version?
        erase_segment_or_flash_bank(0x1980,'\0');
        flash_Write(0x1980,ipc_tx_buff + 0x9b,4,0);
        goto LAB_0000e91c;
      }
```

Fig. 10. Decompiled View of FUN_0000e83a

function not only provided insight into the firmware update mechanism, but also revealed how incoming messages are parsed, validated, and utilized by the gateway. This knowledge is particularly useful, as understanding the data flow and memory mapping allows attackers to inject malicious payloads disguised as valid firmware updates. Furthermore, the use of fallback values and retry loops in the function provides potential entry points for crafting persistent exploits. Further analysis of functions that call FUN_0000e83a could reveal the specific triggers and conditions under which firmware updates are initiated. This understanding would provide insight into the operational timing and decision-making process for updates, potentially identifying other opportunities for exploitation.

Lots of work has been done reversing smaller mathematical functions and utility functions. For example, the divide and divide byte functions as well as the function at 0x3d5a2 that converts numbers into string characters and transmits them via the USCI_A0 peripheral. The current reversing focus on FUN_000221d0 because of its call location in a small loop where the RF chip is configured to be in receive mode each iteration. More work needs to be done figure out what this functions purpose is. Currently, it is only clear that it transmit data and uses CRC functions.

IV. Categorizing Functions

This semester built on the work started in the Spring 2023 paper where the USCB peripheral connected to the transceiver was used to identify Writes_data_out_SPIB0_to_RF_Chip_at_Reg(loc: 0x3a188). Every function that interacted with this peripheral was labeled and categorized this semester. Categories are denoted via bookmarks in the Ghidra repository for the RF firmware. Note in the following subsections that generic names using the memory locations of the functions will be given instead of names for brevity and clarity for future teams in case the function names change.

A. *SX1231_REG_CONFIG*

FUN_ee08: It loops and writes configuration byte data to SX1231 registers 0x18 to 0x4f using values from RAM 0xd483 to 0xd4ba. It wasn't obvious that these values were being read from these memory locations at first because Ghidra can't see the results of a loop being incremented given this is static analysis. By the same reasoning, it is not clear when these values were written to RAM. The current hypothesis is that this function restores register configurations from a previous time, so this function is used to reload a configuration in case the configuration had to switch for some reason, which happens often given the configuration of these registers determines how packets are received and sent.

B. *BUILD_PACKET_SYNC_WORD*

FUN_1fa06: SX1231 Configuration register is used to set the sync word length to 4. The synce work is then set to 0x31 0x54 0x77 0x9a. This contrast with the only sync word that has been seen OTA (0x21 0x43 0x65 0x87). Either the branch where this sync word is set is never taken or is taken rarely for very special interactions.

FUN_34a90: Based on the first parameter this function chooses different 5 byte sync words. One branch sets the sync word to 0x2a 0x86 0xdf 0xca 0x34 while another branch sets it to 0xaa 0x55 0xaa 0x55 0xaa. The other branch sets it using memory 0x4c08 to 0x4c0b and 0xaa for the fifth byte. Using this memory location is interesting because it is the same location builds_rf_message reads from. More work should be done to look at FUN_38b82 and other functions that write to this memory location.

C. AES

Previously there was confusion regarding encryption and decryption of packets. OTA the payload is encrypted and then the sync word and crc bytes are added to either side. This was corroborated by the encryption function found in Swift Tools application when it was reversed with ILSpy and the Pull station firmware having the same AES encryption key.

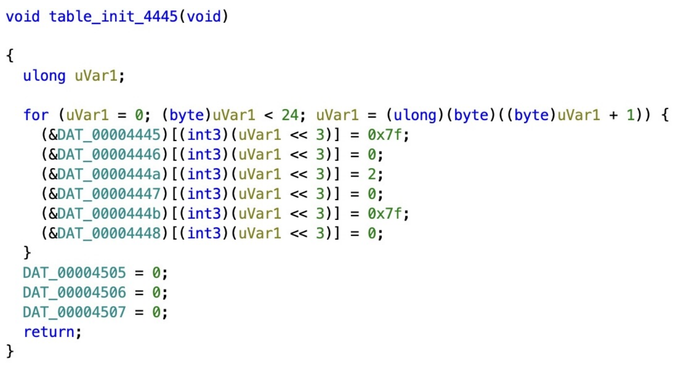
Interestingly, based on the documentation for the SX1231 transceiver the maximum message length that can be encrypted is 64 or sometimes 65 bytes long. This is because the buffer size is limited and the full message is needed to encrypt. This is due to the fact that AES is a block cipher; if the developers used a stream cipher they wouldn't have made the same choices that these developers had to make to encrypt and decrypt longer messages. The decryption process had to be implemented in software (0x2c60e and 0x17544). That's not to say that it isn't using hardware encryption and decryption for shorter messages (see 0xd64c and 0xd68a where the registers are configured).

D. FIFO

FUN_33ff8: This function reads from the SX1231 FIFO register (register 0x00) to get the packet into memory. Note the software doesn't need to remove the sync word or crc bytes when reading from the register, because that is automatically handled by the transceiver hardware. The AES module is configured, so the maximum message length is 64 bytes long. This function returns 0xfe error code if the message is greater than or equal to 65 bytes long. Memory location 0x1f58 is set to 100 by default or the value of param_4. It decrements everytime an interrupt occurs for Timer A0 peripheral. This functions as a timeout with the packet is being read in. If the entire packet isn't read in time the function will return error code 0xfd. 0 is returned when the message is successfully read into memory.

V. Table in RF Binary

Through examining the RF binary, several functions were found in which memory addresses were exclusively offset by multiples of eight, which was accomplished by bit shifting integers to the left by three. The trend seemed strange at first, but the function in Figure 11 was discovered and made it obvious what all of these references are for.



```
void table_init_4445(void)

{
  ulong uVar1;

  for (uVar1 = 0; (byte)uVar1 < 24; uVar1 = (ulong)(byte)((byte)uVar1 + 1)) {
    (&DAT_00004445)[(int3)(uVar1 << 3)] = 0x7f;
    (&DAT_00004446)[(int3)(uVar1 << 3)] = 0;
    (&DAT_0000444a)[(int3)(uVar1 << 3)] = 2;
    (&DAT_00004447)[(int3)(uVar1 << 3)] = 0;
    (&DAT_0000444b)[(int3)(uVar1 << 3)] = 0x7f;
    (&DAT_00004448)[(int3)(uVar1 << 3)] = 0;
  }
  DAT_00004505 = 0;
  DAT_00004506 = 0;
  DAT_00004507 = 0;
  return;
}
```

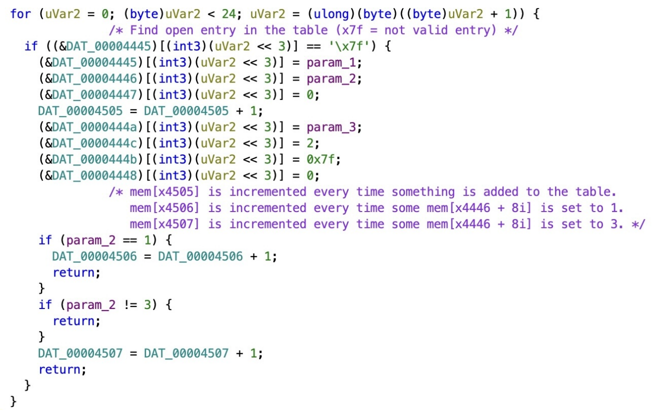
Fig. 11. Decompiled Table Initialization Function

This function initializes the state of a table within memory. When altering this table, functions reference addresses $0x4445$ through $0x444c$ and offset them by multiples of eight, and the integers multiplied by eight cannot exceed 24. Therefore, this table is an array of 24 entries, with each entry consisting of 8 bytes of data, making the table a 24 by 8 array. For the sake of explaining the key functions throughout this portion of the paper, $TABLE[i][j]$ will denote $mem[0x4445 + 8i + j]$.

Before looking at other functions, it is important to examine the initial state of the table. For each valid index k, $TABLE[k][0] = 0x7f$. This is done to mark that entry as being invalid. $TABLE[k][1] = 0$ because, for valid table entries, $TABLE[k][1]$ tends to be in the range $[1, 3]$. $TABLE[k][2] = 0$ because $TABLE[k][2]$ acts as a counter for attempts to write $TABLE[k][0]$ to memory. There has not been as much insight into the initial settings of $TABLE[k][3]$, $TABLE[k][5]$, and $TABLE[k][6]$. The function also clears addresses $0x4505$ through $0x4507$. Each of these addresses hold statistics about the table, with $mem[0x4505]$ holding the number of valid entries, $mem[0x4506]$ holding the number of valid entries $TABLE[k]$ such that $TABLE[k][1] = 1$, and $mem[0x4507]$ holding the number of valid entries $TABLE[k]$ such that $TABLE[k][1] = 3$.

The first function to examine is **byte index_in_table_4445(char param_1)**. This function returns an index k such that $TABLE[k][0] = param\_1$. If $param\_1 = 0x7f$ or no k exists that satisfies this criteria, the function simply returns $0x7f$ to denote that this was an invalid call to the function. This is a key helper function that several other functions use. One function that uses this prominently is void **set_table_4445_entry(byte param_1,byte param_2,byte param_3)**. This function begins by checking that the table is not full ($mem[0x4505] < 24$) and that param_1 is in the range $[1, 50]$. If one of these properties does not hold, no entry will be added to the table and the function returns. If both properties hold, the function then calls **index_in_table_4445(param_1)** to check if there is already an index k such that $TABLE[k][0] = param\_1$. If there is a k that satisfies that criteria, the function sets $TABLE[k][5] = param\_3$ if param_2 is either 1 or 3 and sets $TABLE[k][1] = param\_2$ only if

$param\_2 = 1$, with $mem[0x4506]$ being updated accordingly. If **index_in_table_4445(param_1)** returns $0x7f$, the function searches for space in the table, and, once that space is found, the new table entry is created in the manner shown in Figure 12.



```
for (uVar2 = 0; (byte)uVar2 < 24; uVar2 = (ulong)(byte)((byte)uVar2 + 1)) {
              /* Find open entry in the table (x7f = not valid entry) */
  if ((&DAT_00004445)[(int3)(uVar2 << 3)] == '\x7f') {
    (&DAT_00004445)[(int3)(uVar2 << 3)] = param_1;
    (&DAT_00004446)[(int3)(uVar2 << 3)] = param_2;
    (&DAT_00004447)[(int3)(uVar2 << 3)] = 0;
    DAT_00004505 = DAT_00004505 + 1;
    (&DAT_0000444a)[(int3)(uVar2 << 3)] = param_3;
    (&DAT_0000444c)[(int3)(uVar2 << 3)] = 2;
    (&DAT_0000444b)[(int3)(uVar2 << 3)] = 0x7f;
    (&DAT_00004448)[(int3)(uVar2 << 3)] = 0;
              /* mem[x4505] is incremented every time something is added to the table.
                 mem[x4506] is incremented every time some mem[x4446 + 8i] is set to 1.
                 mem[x4507] is incremented every time some mem[x4446 + 8i] is set to 3. */
    if (param_2 == 1) {
      DAT_00004506 = DAT_00004506 + 1;
      return;
    }
    if (param_2 != 3) {
      return;
    }
    DAT_00004507 = DAT_00004507 + 1;
    return;
  }
}
```

Fig. 12. Method of Adding a New Table Entry

This function showed that TABLE functions closer to a dictionary than a standard two-dimensional array. This is because, for some c in range $[1, 50]$, there can be only by one k such that $TABLE[k][0] = c$. Therefore, $TABLE[k][0]$ acts like a key for the data held in the rest of $TABLE[k]$.

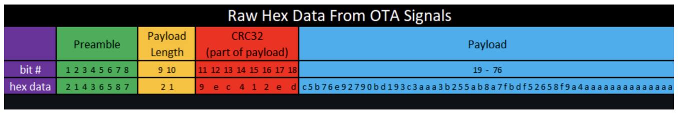
Another function that calls **byte index_in_table_4445(char param_1)** is void **delete_4445_table_entry(char param_1, char param_2)**. If there exists an index k such that $TABLE[k][0] = param\_1$ and $TABLE[k][1] = param\_2$, then that $TABLE[k]$ is deleted by resetting its fields to their initial states from **table_init_4445()**. Notably, if some entry $TABLE[k]$ is deleted such that $TABLE[k][1] = 1$ or $TABLE[k][1] = 3$, then the function sets $TABLE[k][4] = 0$. Future analysis shows that $TABLE[k][4]$ is a boolean value that denotes whether $TABLE[k][0]$ has had the opportunity to be written to memory.

The next functions found were simpler and less notable functions that take in no parameters and simply search for a valid entry that satisfies certain properties, and those aren't very interesting. There are functions like **byte largest_table_key_lt(byte param_1,byte *param_2)**, which returns the largest $TABLE[k][0]$ such that $TABLE[k][0] < param\_1$, and it then sets $*param\_2 = k$. Similarly, **byte smallest_table_key_gt(byte param_1,byte *param_2)** returns the smallest valid $TABLE[k][0]$ such that $param\_1 < TABLE[k][0]$ and then sets $*param\_2 = k$. There are also other functions to edit the table like **void swap_key_in_table(byte *param_1)**. For all indices k such that $TABLE[k][0] < 50$, the function sets $TABLE[k][0] = param\_1[TABLE[k][0] - 1]$, fully deleting the entry if $TABLE[k][0]$ becomes $0x7f$. These functions are interesting in how they traverse and alter the table, but they do not answer the question of what the table is for.

The main function that answers this is **void transmit_and_delete_from_table()**. Before explaining what the function does for each index k, let a boolean $cond = mem[0x3c08] \vee (mem[0x3743] \wedge mem[0x3c09])$. For each index k, the function begins by continuing to the next index if $TABLE[k][0] = 0x7f$ or $TABLE[k][1]$ is not 1 or 3. $TABLE[k][2]$ is then incremented, and the next index is checked if $TABLE[k][2] < 11$. Afterwards, if $cond$, then the function sets $TABLE[k][2] = TABLE[k][4] = 0$. Otherwise, the function sets $TABLE[k][4] = 1$, denoting that the entry has had the opportunity to be written to memory. Finally, if $TABLE[k][1] = 3$, then the function writes S and R to USCIA0, which could denote "Signal Received", and $TABLE[k][0]$ is written into an array starting at $0x322b$. After that loop is over, the table entries that satisfy the condition $TABLE[k][4] = 1$ are deleted. From this function, it is apparent that $TABLE[k][2]$ is a counter denoting the number of times $TABLE[k][0]$ has attempted to be written into memory and that $TABLE[k][4]$ is a boolean value denoting whether $TABLE[k][0]$ has had a real chance to be written to memory. Overall, $TABLE$ is a table of values to be written to memory, with $TABLE[k][0]$ holding the value to be written and the rest of each entry holding various statistics about whether it is ready to be stored. The next steps for analyzing this table will be to analyze the functions that call the functions discussed and examine what they do with the results of these operations.

VI. RF Protocol Analysis

Honeywell's SWIFT system utilizes an SX1231 Transceiver to handle the RF communication (195Mhz, FSK) between the gateway and peripherals. In order to reverse engineer the protocol we utilized a HackRF in combination with the SWIFT Tools application to trigger different scenarios and record the RF communication. From previous semesters we know the general format of the messages sent.



Raw Hex Data From OTA Signals

	Preamble	Payload Length	CRC32 (part of payload)	Payload
bit #	1 2 3 4 5 6 7 8	9 10	11 12 13 14 15 16 17 18	19 - 76
hex data	2 1 4 3 6 5 8 7	2 1	9 e c 4 1 2 e d	c5b76e92790bd193c3aaa3b255ab8a7fbdf52658f9a4aaaaaaaaaaaaaa

Fig. 13. Diagram of OTA message format

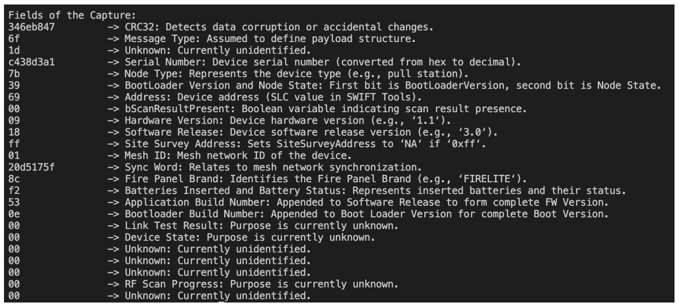
Previous semesters identified a constant preamble/syncword "**0x21436587**" and a packet length of "**0x21**" specifying that the encoded payload is 33 bytes long. In order to decode this we must XOR every byte with the hex value "**0xaa**".

Using ILSPY, an open-source .NET decompiler, we can utilize the fact that Honeywell used common dlls (dynamically linked libraries) to identify the fields of the payload. Some notable fields are: Bytes (7:10) is the serial number of the pull station and Byte (13) is the SLC address of the pull station. Armed with this information we were able to create a python script to automatically take data from a HackRF capture and translate it to the respective fields.

Overview of RF Packet disassembly process

```
1. Example Gateway RF Capture:
```

```
001000010100001101100101100001110010000
011001111011000100000100101110110111100
010110110111011011101001001001111001000
001011110100011001001111000011101010100
101000111011001001010101101010111001...

2. Translate to Hex (if not already):
21436587219ec412edc5b76e92790bd193c3aa
a3b255ab8a7fbdf52658f9a4aaaaaaaaaaaaaa

-> Syncword: 0x214365872
-> Packet Lengh: 0x21

3. XOR non syncword + pck len bytes:
2143658721346eb8476f1dc438d3a17b396900
0918ff0120d5175f8cf2530e00000000000000

4. Match bytes with IL Spy description
```



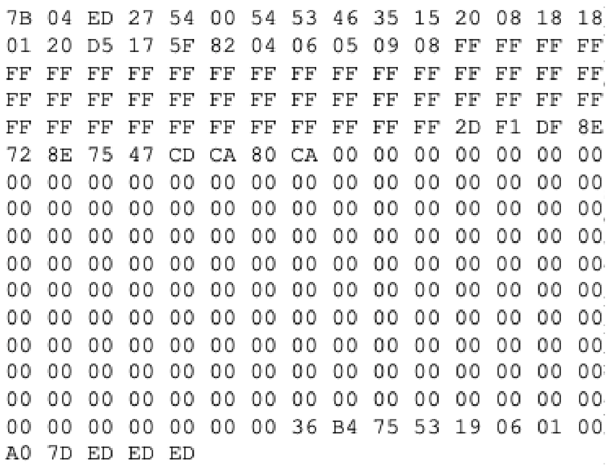
```
Fields of the Capture:
346eb847         -> CRC32: Detects data corruption or accidental changes.
6f               -> Message Type: Assumed to define payload structure.
1d               -> Unknown: Currently unidentified.
c438d3a1         -> Serial Number: Device serial number (converted from hex to decimal).
7b               -> Node Type: Represents the device type (e.g., pull station).
39               -> BootLoader Version and Node State: First bit is BootLoaderVersion, second bit is Node State.
69               -> Address: Device address (SLC value in SWIFT Tools).
00               -> bScanResultPresent: Boolean variable indicating scan result presence.
09               -> Hardware Version: Device hardware version (e.g., '1.1').
18               -> Software Release: Device software release version (e.g., '3.0').
ff               -> Site Survey Address: Sets SiteSurveyAddress to 'NA' if '0xff'.
01               -> Mesh ID: Mesh network ID of the device.
20d5175f         -> Sync Word: Relates to mesh network synchronization.
8c               -> Fire Panel Brand: Identifies the Fire Panel Brand (e.g., 'FIRELITE').
f2               -> Batteries Inserted and Battery Status: Represents inserted batteries and their status.
53               -> Application Build Number: Appended to Software Release to form complete FW Version.
0e               -> Bootloader Build Number: Appended to Boot Loader Version for complete Boot Version.
00               -> Link Test Result: Purpose is currently unknown.
00               -> Device State: Purpose is currently unknown.
00               -> Unknown: Currently unidentified.
00               -> Unknown: Currently unidentified.
00               -> Unknown: Currently unidentified.
00               -> RF Scan Progress: Purpose is currently unknown.
00               -> Unknown: Currently unidentified.
```

Fig. 14. OTA Packet Decoded Fields

VII. Serial Protocol Analysis - Gateway Communication

The SWIFT Tools Application utilizes a serial protocol to send messages to the SWIFT W-USB Transceiver, allowing communication with devices on the mesh network. The application, distributed as an unobfuscated.NET executable and DLL, allows for straightforward reverse engineering using tools like ILSpy. Each message frame contains a WirelessComm.WirelessPacket, encoded based on frame type, with WirelessComm.ProtocolManager managing frame type selection and CRC calculations. To decode captured serial traffic (USB Protocol), the team reverse-engineered SWIFT Tools, decompiling its DLLs to understand its operation. Key analysis focused on three DLLs (WirelessComm.dll, WirelessInterfaces.dll, and WirelessPlugin.dll) responsible for parsing and creating serial messages to be sent to and from the W-USB device and the Gateway / Pull Stations.

Through the use of Serial Port Monitor we were able to grab an example of such communication between SWIFT Tools and the Gateway. At first glance, the purpose of this message is unknown as it contains a long string of hex bytes. Therefore, in order to deconstruct the meaning behind the bytes, a Python script was created to automatically pull out fields and relate them to enums found in IL Spy.



```
7B 04 ED 27 54 00 54 53 46 35 15 20 08 18 18
01 20 D5 17 5F 82 04 06 05 09 08 FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF 2D F1 DF 8E
72 8E 75 47 CD CA 80 CA 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 36 B4 75 53 19 06 01 00
A0 7D ED ED ED
```

Fig. 15. General Serial Gateway Message

The main source of information when deconstructing this packet was the WirelessNode Class located in the *Honeywell.WirelessTool.WirelessInterfaces.WirelessNode* dll (figure 13).

```
public bool bIsSelected;
public bool isDeviceSelected;
public bool bIsToolsLocked;
public static GridElements gElement;
public string SerialNo;
public NodeState State;
public string strDeviceLabel;
public bool isSiteSurveyDevice;
public FireBrand FirePanelBrand;
private string softwareVersion;
private string slcFirmwareVersion;
public string strHardwareVersion;
public string strSiteSurveyAddress;
public string strSiteSurveyLocation;
public bool bScanResultPresent;
public bool bIsFirmwareSelected;
public bool bIsCompatiable;
public BatteryStatus BatteyStatus;
public GatewayMeshAttribute gateOnlyAttributes;
public int iDeviceLevelRSSI;
public BatteryLocations bLocations;
public int LinkTestResult;
public string DeviceState;
public int RFScanProgress;
public string SyncExternalPowerStatus;
public AVBaseAttributes.AVBaseConfig1 AVBaseCnfg1;
public AVBaseAttributes.AVBaseConfig2 AVBaseCnfg2;
public AVBaseAttributes.AVBaseBatteryInfo AVBaseBattery;
public string HWRevision;
public string MUBootloader;
public string SyncWord;
```

Fig. 16. Wireless Node local variables

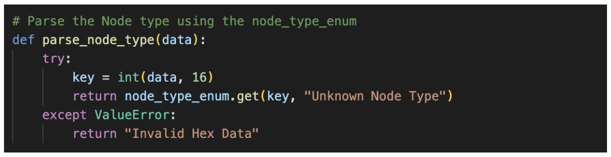
Accompanied by previous semester's research we were able to determine the bytes in the Serial Packet that corresponded with these variables:

```
Bytes -> Field
```

```
-------------------------------------
1:4 -> Serial Number (int)
5 -> Node Type (enum)
6 -> BootLoader Version and Node State
                 (bitfield / enum)
7 -> SLC Address (int)
8 -> SLC BootLoader Version
9 -> Hardware Version
10 -> Software Version
11 -> SLC Firmware Version
12 -> Mesh ID (int)
13:16 -> Sync Word (int)
17 -> Fire Brand (enum)
18 -> Gateway Mesh Attribute List Capacity
19:22 -> Mesh SLC Address (int)
68:80 -> Serial Numbers (int)
229 -> Magnet Lock Status (bitfield)
230 -> Various booleans (bitfield)
231 -> Lock Time Remaining (enum)
232 -> RF Application Build Number
233 -> SLC Application Build Number
234 -> RF Bootloader Build Number
235 -> SLC Bootloader Build Number
236 -> Unknown Field
237 -> XOR Checksum
```

As many of the field present in the packet represented enums, bit fields, and integers requiring manipulation, specific parsing functions were required to deconstruct each value:

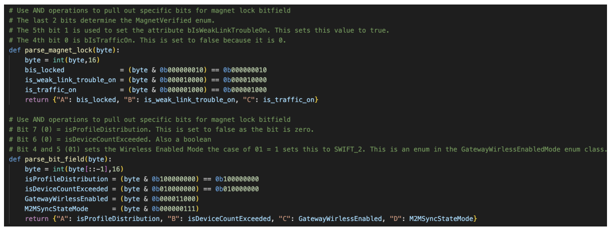
- **Enum:** Enums were stored into python dictionaries and keyed with the base 10 integer corresponding to the ILSpy disassembly



```python
# Parse the Node type using the node_type_enum
def parse_node_type(data):
    try:
        key = int(data, 16)
        return node_type_enum.get(key, "Unknown Node Type")
    except ValueError:
        return "Invalid Hex Data"
```

Fig. 17. Enum Parsing

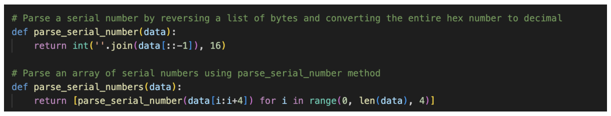
- **Bitfields:** Initially represented as hex bytes, simple AND operations were used to specify bits and coorelate them to booleans



```python
# Use AND operations to pull out specific bits for magnet lock bitfield
# The last 2 bits determine the MagnetVerified enum.
# The 5th bit 1 is used to set the attribute bIsWeakLinkTroubleOn. This sets this value to true.
# The 4th bit 0 is bIsTrafficOn. This is set to false because it is 0.
def parse_magnet_lock(byte):
    byte = int(byte,16)
    bis_locked              = (byte & 0b000000010) == 0b000000010
    is_weak_link_trouble_on = (byte & 0b000010000) == 0b000010000
    is_traffic_on           = (byte & 0b000001000) == 0b000001000
    return {"A": bis_locked, "B": is_weak_link_trouble_on, "C": is_traffic_on}

# Use AND operations to pull out specific bits for magnet lock bitfield
# Bit 7 (0) = isProfileDistribution. This is set to false as the bit is zero.
# Bit 6 (0) = isDeviceCountExceeded. Also a boolean
# Bit 4 and 5 (01) sets the Wireless Enabled Mode the case of 01 = 1 sets this to SWIFT_2. This is an enum in the GatewayWirlessEnabledMode enum class.
def parse_bit_field(byte):
    byte = int(byte[::-1],16)
    isProfileDistribution = (byte & 0b100000000) == 0b100000000
    isDeviceCountExceeded = (byte & 0b010000000) == 0b010000000
    GatewayWirlessEnabled = (byte & 0b000011000)
    M2MSyncStateMode      = (byte & 0b000000111)
    return {"A": isProfileDistribution, "B": isDeviceCountExceeded, "C": GatewayWirlessEnabled, "D": M2MSyncStateMode}
```

Fig. 18. Bit Field Parsing

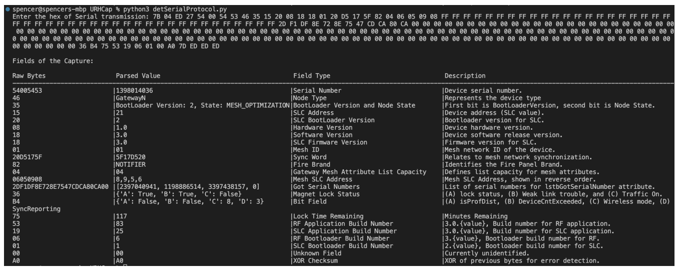
- **Int:** Serial Numbers and Mesh ID's were given in the packet as reversed hex bytes, so in order to relate them to thier base 10 represenations, they were reversed and converted into base 10



```python
# Parse a serial number by reversing a list of bytes and converting the entire hex number to decimal
def parse_serial_number(data):
    return int(''.join(data[::-1]), 16)

# Parse an array of serial numbers using parse_serial_number method
def parse_serial_numbers(data):
    return [parse_serial_number(data[i:i+4]) for i in range(0, len(data), 4)]
```

Fig. 19. Integer Parsing

Iterating through all fields discovered in the Serial Packet, we were able to automatically deconstruct the information hidden within.



```
spencer@spencers-mbp URHCap % python3 detSerialProtocol.py
Enter the hex of Serial transmission: 7B 04 ED 27 54 00 54 53 46 35 15 20 08 18 18 01 20 D5 17 5F 82 04 06 05 09 08 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 2D F1 DF 8E 72 8E 75 47 CD CA 80 CA 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 36 B4 75 53 19 06 01 00 A0 7D ED ED ED

Fields of the Capture:

Raw Bytes                 Parsed Value                                 Field Type                          Description
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
54005453                  |1398014036                                  |Serial Number                      |Device serial number.
46                        |GatewayN                                    |Node Type                          |Represents the device type
35                        |BootLoader Version: 2, State: MESH_OPTIMIZATION|BootLoader Version and Node State  |First bit is BootLoaderVersion, second bit is Node State.
15                        |21                                          |SLC Address                        |Device address (SLC value).
20                        |2                                           |SLC BootLoader Version             |Bootloader version for SLC.
08                        |1.0                                         |Hardware Version                   |Device hardware version.
18                        |3.0                                         |Software Version                   |Device software release version.
18                        |3.0                                         |SLC Firmware Version               |Firmware version for SLC.
01                        |01                                          |Mesh ID                            |Mesh network ID of the device.
20D5175F                  |5F17D520                                    |Sync Word                          |Relates to mesh network synchronization.
82                        |NOTIFIER                                    |Fire Brand                         |Identifies the Fire Panel Brand.
04                        |04                                          |Gateway Mesh Attribute List Capacity |Defines list capacity for mesh attributes.
06050908                  |8,9,5,6                                     |Mesh SLC Address                   |Mesh SLC Address, shown in reverse order.
2DF1DF8E728E7547CDCA80CA00 |[2397040941, 1198886514, 3397438157, 0]     |Got Serial Numbers                 |List of serial numbers for lstbGotSerialNumber attribute.
36                        |{'A': True, 'B': True, 'C': False}          |Magnet Lock Status                 |(A) lock status, (B) Weak link trouble, and (C) Traffic On.
B4                        |{'A': False, 'B': False, 'C': 8, 'D': 3}    |Bit Field                          |(A) isProfDist, (B) DeviceCntExceeded, (C) Wireless mode, (D)
SyncReporting
75                        |117                                         |Lock Time Remaining                |Minutes Remaining
53                        |83                                          |RF Application Build Number        |3.0.{value}, Build number for RF application.
19                        |25                                          |SLC Application Build Number       |3.0.{value}, Build number for SLC application.
06                        |6                                           |RF Bootloader Build Number         |3.{value}, Bootloader build number for RF.
01                        |1                                           |SLC Bootloader Build Number        |2.{value}, Bootloader build number for SLC.
00                        |00                                          |Unknown Field                      |Currently unidentified.
A0                        |A0                                          |XOR Checksum                       |XOR of previous bytes for error detection.
```

Fig. 20. Serial Packet Deconstruction

VIII. Ghidra's Support for MSP430

Ghidra's support for the MSP430 language is incomplete and causes several difficulties when decompiling and viewing functions in the listing view. The main reason for the problem is the odd way this language handles addresses. The addressability of memory is only 20-bits; however, the operands can be 20-bits and be extended to take up the full space of a 32-bit register. When Ghidra sees the 32-bit register it interprets it as such and this nuance contributes to some edge cases that make the decompiler show unnecessary (although not necessarily incorrect) code lines. For example, before most function calls there will be a line of code showing the return address being placed on the stack; this behavior is normally implied, so showing it leads to a messy decompiler view. Further complicating the decompiler view, this addressability leads Ghidra to add extensive masking to numerous variables (E.G. "& 0xFFFFF"), which increases the clutter in the code, making it that much harder to read.

Other problems have stemmed from the "Decompiler Parameter ID" option of the analyzer. This option can be used on a binary to automatically create parameters and local variables. After exploring the functions, the team realized that a significant number of functions have incorrect arguments and incorrect return types. Registers that are immediately overwritten in the first few lines of assembly instructions have been consistently defined as inputs, which doesn't make sense. Also, void return values weren't committed, so functions that returned nothing looked like they would fill variables in the decompiler view.

In addition to choosing the wrong registers for input, the wrong granularity was also chosen. MSP430 uses three different granularities for arguments: 8-bit, 16-bit, and 20-bit. This is denoted by instructions that end in B, W, and A respectively. The automated analysis only chose the correct register (R12_lo in Ghidra notation) when the type B instruction was used, not type W or type A. Therefore, the team has had to continually manually change register arguments for many functions. The combination of the functions with the wrong number of arguments, the wrong return values, and the wrong argument granularities has made the decompiler unusable and even useless in some scenarios.

IX. Spoofing a Device on the SLC

The Gateway discussed so far is a singular device with two separate co-processors: the RF and SLC. While the RF handles communication with the wireless devices on the mesh network, the Signaling Line Circuit (SLC) handles communication between the control panel and intelligent and addressable initiating, monitor, and control devices. Essentially, the SLC is a data and power bus that transmits both information and power between everything that makes up the fire alarm system. While other circuits may have an 'on' or 'off,' the SLC has several types of signals. It also handles many different types of messages, such as simple polling from the fire alarm control panel (FACP) to see which devices are on the network, receiving the polling message, and alarm signals that arise from a pull station or smoke detector. Each SLC message is broadcasted to all other devices on the network. In order to pass information from the wireless devices to the FACP, the SLC firmware regularly uses ports USCI A1, USCI B1, and General-Purpose I/O (GPIO) ports 1, 2, and 8 [1].

The goal of our team is to spoof a device on the SLC network, which entails replicating any of the legitimate devices that would be on the SLC network. The development of this device allows us to send 'spoofed' packets to the gateway, which will allow an attacker to remotely compromise the gateway. This device can be some kind of microcontroller-based device that can arbitrarily change General Purpose I/O (GPIO) lines to toggle the lines of the SLC network. Some top candidates the team discussed were Arduino and Raspberry Pi. The Raspberry Pi has all the features of a standard PC, such as a dedicated processor, graphics driver, memory, and its own operating system known as the Raspberry Pi OS. It can perform numerous tasks, such as plugging a monitor, mouse, and keyboard to it, as well as connecting to the Internet and adding a camera, among many other things. Due to this, the Raspberry Pi is seen as much more complex; while the Arduino is an electronic board with a simple microcontroller, the Raspberry Pi is a full-fledged computer [2].

Therefore, the team decided to use the Arduino as our surrogate device. The Arduino provides a programmable circuit board and can read data from sensors and buttons and turn it into outputs. The Arduino is best suited for repetitive tasks and for projects that need a simple output based on sensory input, and the code ran on the surrogate device will go through less levels of complexity than with the Raspberry Pi, thus it fits our design goals well [3].

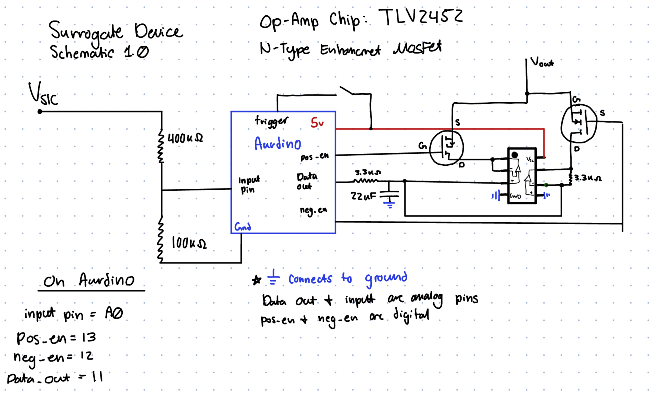


Fig. 21. Previously proposed surrogate device design

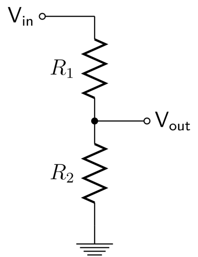


Fig. 22. A simple resistive voltage divider

$$V_{\text{out}} = \frac{R_2}{R_1 + R_2} \cdot V_{\text{in}}$$

Fig. 23. Formula for a voltage divider

In order to spoof a fire alarm on the wired SLC network, the team realized that we can't actually replace a device on the SLC line. Instead, the surrogate device must be added to the line in a way such that it appears to be part of the system and can replicate the appropriate addresses and SLC messages. Otherwise, the system would enter a "System Trouble" state from not recognizing the surrogate device.

A current design proposed by the Fall 2023 team consists of a very similar design to the wired pull station set up inside the lab. Our team is currently examining it to see what may need to be edited or adjusted, if anything. The first design choice we noticed is the addition of a voltage divider. Because messages going into the pull station can fluctuate from 10V to 24V, and the Arduino can only support up to 5V, the device would need a resistive voltage divider to lower the input voltage going into the microcontroller. Thus, two resistors of 400 microohms (R1) and 100 microohms (R2) are used in the design, though these numbers are not obligatory and can be slightly altered (i.e. a max input of 24V with R1 = 800 and R2 = 200 produces an output of 4.8V and is still in the Arduino's safe zone).

Throughout the semester, our team worked diligently to design a new version of the surrogate device and code. One implementation we decided to incorporate was debouncing logic. Debouncing is a technique that removes unwanted input

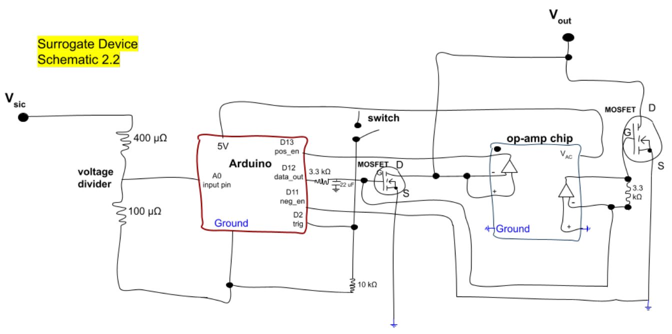


Fig. 24. Surrogate Device Schematic 2.2

noise from buttons or switches. This could be important for the trigger that lets us know when there is an alarm or not. There are 2 ways to implement debouncing: through hardware or software. With hardware, we can use a resistor-capacitor (RC) filter with a Schmitt trigger diode. However, this might be too complicated for our purposes—our team instead opted to write the debouncing in our software code. The hardware included for the debounce logic include: Arduino Board, momentary button or switch, 10k ohm resistor, hook-up wires, and breadboard.

Our team also created a design we aptly named the Surrogate Device Schematic 2.2, as seen in Figure 24. Two designs were created, both 2.1 and 2.2, with Schematic 2.2 more closely modeling the design of the previous semester. One important feature of these designs is the two MOSFETs. MOSFETs are Metal-Oxide-Semiconductor Field-Effect Transistors that are used to amplify or switch electronic signals in circuits. They can efficiently control high-power devices with minimal energy loss–they consume little power and can control high-current signals. There are two types: N-channel and P-channel. N-channel turns on when the voltage applied to the gate is positive relative to the source, and P-channel is the opposite (negative). In our design we are using an N-type Enhancement MOSFET, which has three visible terminals: Source (S), Drain (D), and Gate (G). Source is where the current enters the MOSFET, and Drain is where the current exits. Gate is the control terminal that can either allow or block voltage flow between the other two terminals. For example, if there is a sufficient amount of positive voltage at the Gate, it will allow the current to flow from Drain to Source. If the voltage at the Gate is too low, the MOSFET will remain off and no current will flow. In our design, the MOSFET likely controls the voltage output (Vout) based on the input signal from the Arduino. This essentially means the Arduino can tell the MOSFET to turn on (through a control signal to its Gate) and thus allow the current to pass, or vice versa.

X. SLC Gateway Reverse Engineering

One of the challenges with spoofing a device on the SLC network is understanding what the SLC firmware does. To that end, the team worked towards reverse engineering the firmware in Ghidra. While conducting the analysis of the code, there were a few sections of code that raised some red flags, prompting a more in-depth analysis. Within the initialization function, the system performs a series of Cyclic Redundancy Checks, or CRC, on a few key sections of code, as well as stack variables.

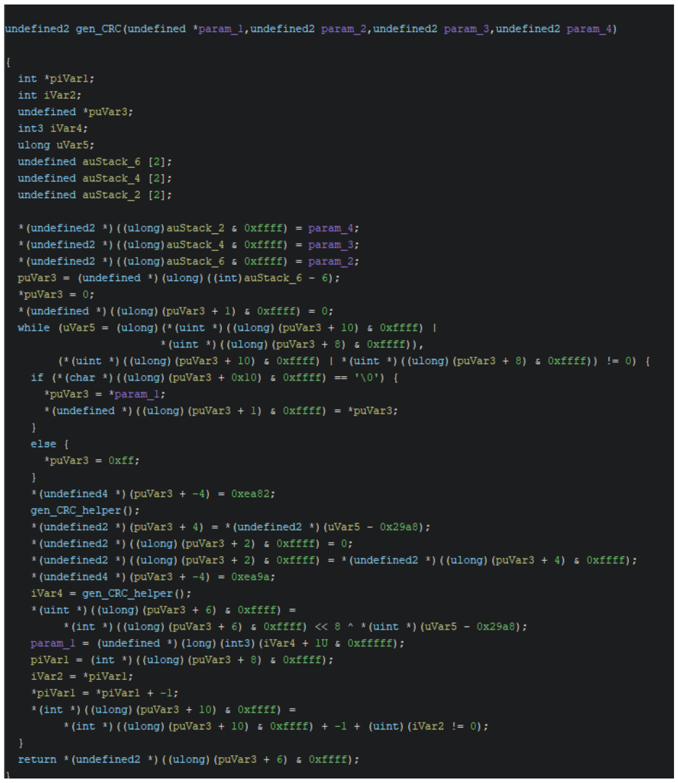

```
undefined2 gen_CRC(undefined *param_1,undefined2 param_2,undefined2 param_3,undefined2 param_4)

{
  int *piVar1;
  int iVar2;
  undefined *puVar3;
  int3 iVar4;
  ulong uVar5;
  undefined auStack_6 [2];
  undefined auStack_4 [2];
  undefined auStack_2 [2];

  *(undefined2 *)((ulong)auStack_2 & 0xffff) = param_4;
  *(undefined2 *)((ulong)auStack_4 & 0xffff) = param_3;
  *(undefined2 *)((ulong)auStack_6 & 0xffff) = param_2;
  puVar3 = (undefined *)(ulong)((int)auStack_6 - 6);
  *puVar3 = 0;
  *(undefined *)((ulong)(puVar3 + 1) & 0xffff) = 0;
  while (uVar5 = (ulong)(*(uint *)((ulong)(puVar3 + 10) & 0xffff) |
                        *(uint *)((ulong)(puVar3 + 8) & 0xffff)),
        (*(uint *)((ulong)(puVar3 + 10) & 0xffff) | *(uint *)((ulong)(puVar3 + 8) & 0xffff)) != 0) {
    if (*(char *)((ulong)(puVar3 + 0x10) & 0xffff) == '\0') {
      *puVar3 = *param_1;
      *(undefined *)((ulong)(puVar3 + 1) & 0xffff) = *puVar3;
    }
    else {
      *puVar3 = 0xff;
    }
    *(undefined4 *)(puVar3 + -4) = 0xea82;
    gen_CRC_helper();
    *(undefined2 *)(puVar3 + 4) = *(undefined2 *)(uVar5 - 0x29a8);
    *(undefined2 *)((ulong)(puVar3 + 2) & 0xffff) = 0;
    *(undefined2 *)((ulong)(puVar3 + 2) & 0xffff) = *(undefined2 *)((ulong)(puVar3 + 4) & 0xffff);
    *(undefined4 *)(puVar3 + -4) = 0xea9a;
    iVar4 = gen_CRC_helper();
    *(uint *)((ulong)(puVar3 + 6) & 0xffff) =
         *(int *)((ulong)(puVar3 + 6) & 0xffff) << 8 ^ *(uint *)(uVar5 - 0x29a8);
    param_1 = (undefined *)(long)(int3)(iVar4 + 1U & 0xfffff);
    piVar1 = (int *)((ulong)(puVar3 + 8) & 0xffff);
    iVar2 = *piVar1;
    *piVar1 = *piVar1 + -1;
    *(int *)((ulong)(puVar3 + 10) & 0xffff) =
         *(int *)((ulong)(puVar3 + 10) & 0xffff) + -1 + (uint)(iVar2 != 0);
  }
  return *(undefined2 *)((ulong)(puVar3 + 6) & 0xffff);
}
```

Fig. 25. Decompiled View of gen_crc function

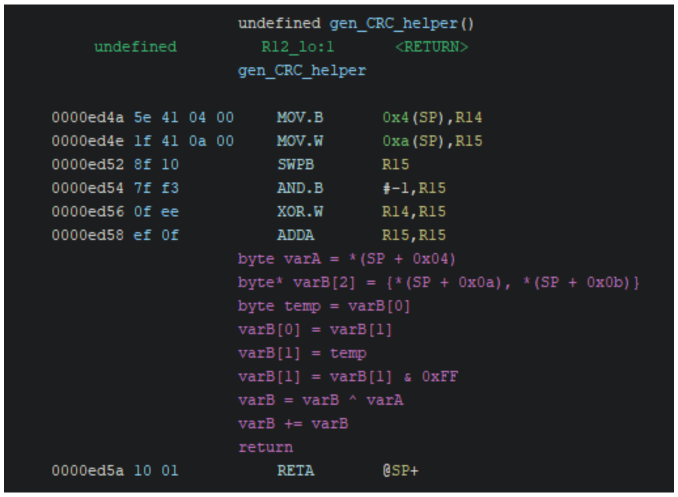
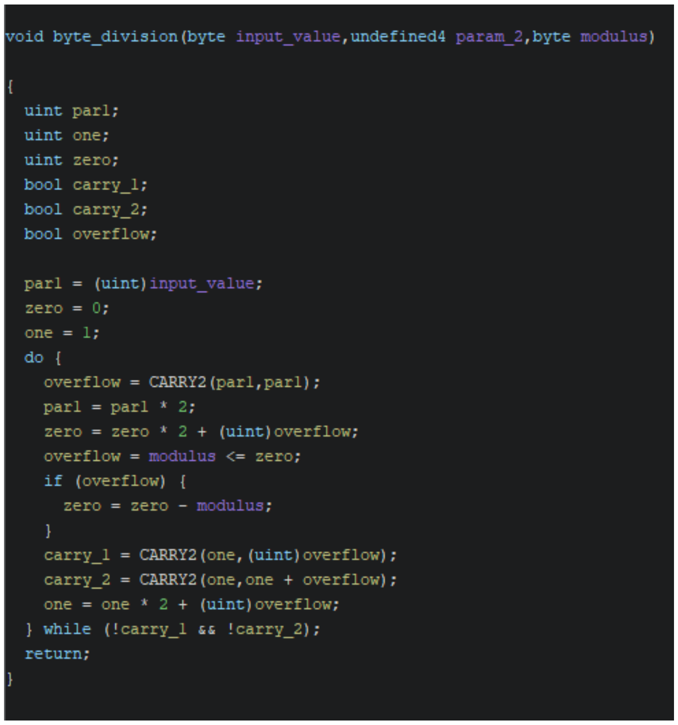

```
                       undefined gen_CRC_helper()
     undefined           R12_lo:1        <RETURN>
                       gen_CRC_helper

   0000ed4a 5e 41 04 00     MOV.B       0x4(SP),R14
   0000ed4e 1f 41 0a 00     MOV.W       0xa(SP),R15
   0000ed52 8f 10           SWPB        R15
   0000ed54 7f f3           AND.B       #-1,R15
   0000ed56 0f ee           XOR.W       R14,R15
   0000ed58 ef 0f           ADDA        R15,R15
                       byte varA = *(SP + 0x04)
                       byte* varB[2] = {*(SP + 0x0a), *(SP + 0x0b)}
                       byte temp = varB[0]
                       varB[0] = varB[1]
                       varB[1] = temp
                       varB[1] = varB[1] & 0xFF
                       varB = varB ^ varA
                       varB += varB
                       return
   0000ed5a 10 01           RETA        @SP+
```

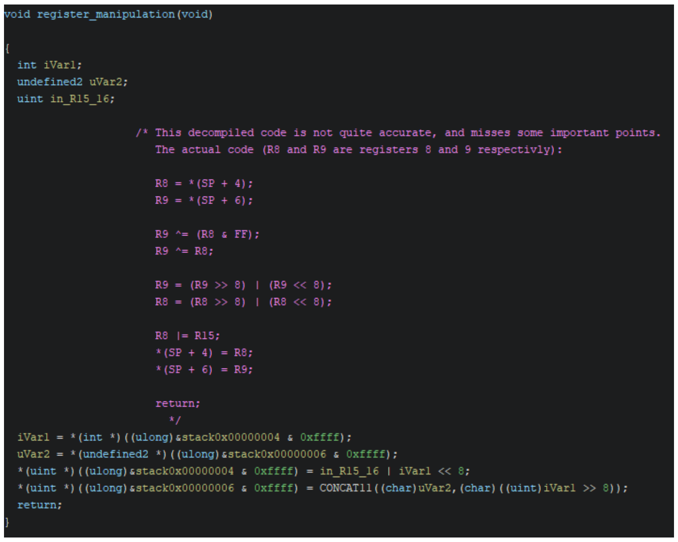
Fig. 26. Listing View of gen_crc_helper function

After completing these checks, it verifies them against hard-coded values to ensure the integrity of the specific locations. The potential issue comes after this. If the CRCs fail, the system enters an error correction state. While this state is far from fully analyzed currently, most of these corrections do not work to correct errors within the code or local variables – they fix the peripheral pin values and perform some byte manipulation. Most of the manipulation conducted on the bytes are bit shifts, long division, and byte swaps, and conducted on register values and stack values.



```
void byte_division(byte input_value,undefined4 param_2,byte modulus)

{
  uint par1;
  uint one;
  uint zero;
  bool carry_1;
  bool carry_2;
  bool overflow;

  par1 = (uint)input_value;
  zero = 0;
  one = 1;
  do {
    overflow = CARRY2(par1,par1);
    par1 = par1 * 2;
    zero = zero * 2 + (uint)overflow;
    overflow = modulus <= zero;
    if (overflow) {
      zero = zero - modulus;
    }
    carry_1 = CARRY2(one,(uint)overflow);
    carry_2 = CARRY2(one,one + overflow);
    one = one * 2 + (uint)overflow;
  } while (!carry_1 && !carry_2);
  return;
}
```

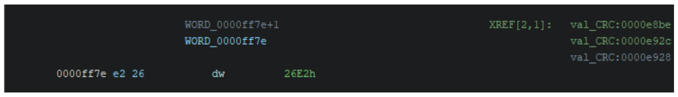
Fig. 27. Decompiled View of byte_division function



```
void register_manipulation(void)

{
  int iVar1;
  undefined2 uVar2;
  uint in_R15_16;

                  /* This decompiled code is not quite accurate, and misses some important points.
                     The actual code (R8 and R9 are registers 8 and 9 respectivly):

                     R8 = *(SP + 4);
                     R9 = *(SP + 6);

                     R9 ^= (R8 & FF);
                     R9 ^= R8;

                     R9 = (R9 >> 8) | (R9 << 8);
                     R8 = (R8 >> 8) | (R8 << 8);

                     R8 |= R15;
                     *(SP + 4) = R8;
                     *(SP + 6) = R9;

                     return;
                       */
  iVar1 = *(int *)((ulong)&stack0x00000004 & 0xffff);
  uVar2 = *(undefined2 *)((ulong)&stack0x00000006 & 0xffff);
  *(uint *)((ulong)&stack0x00000004 & 0xffff) = in_R15_16 | iVar1 << 8;
  *(uint *)((ulong)&stack0x00000006 & 0xffff) = CONCAT11((char)uVar2,(char)((uint)iVar1 >> 8));
  return;
}
```

Fig. 28. Decompiled View of register manipulation function

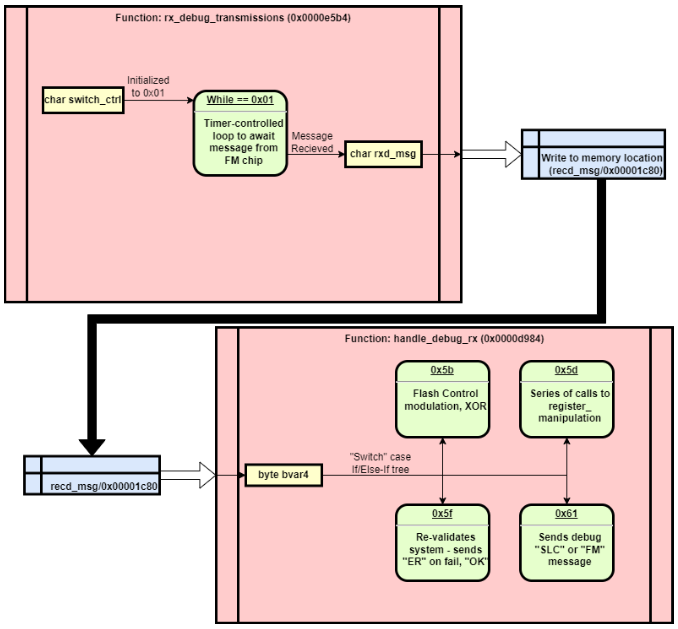
Further investigation into this section of the code shows two significant security concerns. First, the debugging/error-correction loop is controlled by a ”do/while(true)” loop that uses the system validation to escape the loop by booting into the main system loop. With physical access to the gateway device, an attacker could potentially leverage this structure by inserting or manipulating a single byte to cause the system to continually fail to validate, thus never booting into the main loop, effectively causing a denial of service for the system as a whole. If an attacker could change the hard-coded, unencrypted CRC value located at 0x0000ff7e (0x26e2), this would be enough to cause this failure.



```
                     WORD_0000ff7e+1                          XREF[2,1]:   val_CRC:0000e8be
                     WORD_0000ff7e                                         val_CRC:0000e92c
                                                                           val_CRC:0000e928
    0000ff7e e2 26         dw          26E2h
```

Fig. 29. Hard-coded CRC value

The second potential issue is a little more complicated, however, potentially easier to exploit. Within this debugging loop, there is a function, ”rec_debug_transmissions”. This function receives debugging information from the FM chip, using the timer control peripheral to ensure that the system does not end up hung up waiting for the data. It then stores this byte in memory, which is then passed to another function, ”handle_debug_rx”, where it is run through a pseudo-switch structure, where there are four options based on what the received data is. It will either run through some flash-control adjustments, a series of calls to the ”register_manipulation” function that adjusts the various registers in distinct ways, system re-validation that transmits ”ER” if it fails or ”OK” if it passes to the debug serial port, or uses the buffer section that is reset on boot to either transmit a message indicating SLC or FM failure to the debug serial port.



Function: rx_debug_transmissions (0x0000e5b4)
char switch_ctrl
Initialized to 0x01
While == 0x01
Timer-controlled loop to await message from FM chip
Message Recieved
char rxd_msg
Write to memory location (recd_msg/0x00001c80)
Function: handle_debug_rx (0x0000d984)
recd_msg/0x00001c80
byte bvar4
"Switch" case If/Else-If tree
0x5b
Flash Control modulation, XOR
0x5d
Series of calls to register_ manipulation
0x5f
Re-validates system - sends "ER" on fail, "OK"
0x61
Sends debug "SLC" or "FM" message

Fig. 30. Schema of how the system handles received debug data

This transmission could be intercepted by an attacker, and the attacker could instead send the byte that corresponds to what they would like the system to do, which could cause unintended functionality, including full system failure.

Further work is needed to verify if either, or even both, of these potential security concerns are indeed exploitable vulnerabilities within the system, and, if they are, how they could be exploited and how to patch them if they are.

XI. SLC Interprocessor Message

Identifying the interprocessor message (IPC) in the context of spoofing a device on the SLC (Signaling Line Circuit) network is crucial because these messages are the primary means by which devices communicate and exchange data on the network. Understanding the structure and content of these messages allows an attacker to replicate or manipulate the communication between devices, which is essential for successfully impersonating a legitimate device.

The IPC message is composed of several key components, including a header, a 190-byte buffer holding the data of the message, a footer, and a checksum. The message format is structured with specific bytes for various functions, such as byte 5, which represents the gateway's primary SLC address, and byte 3 of the header, which indicates the wireless mode (0 for off, 1 for on). Additionally, the data point DAT_00002c51, which corresponds to the 4th byte of the header, seems to correspond to a control signal for USCI A1, possibly used to trigger specific actions, such as turning on or off certain components. This signal appears to be influenced by the wireless mode setting, with DAT_00002c51 being set to 0 when the wireless mode is off, indicating a condition for further operations. The footer, consisting of bytes 195-199, plays a critical role in message validation, where byte 199 acts as a checksum or CRC, ensuring the integrity of the data during transmission. Our analysis suggests that byte 195 is initially null during initialization but is set to 1 in the reset_globals function during runtime, potentially indicating a state change or flag.

XII. Conclusion

A. Next Steps

Continuing the team's effort in deciphering what messages are encrypted and which are left unencrypted, creating an RF backdoor attack, and the building of a surrogate device are all important goals that will further the reverse engineering attempts. The next steps for the team entail continuing our investigation into the SLC firmware, as well as reverse engineering the RF Gateway. We also plan to continue the development of the surrogate device, to which the Surrogate Device Schematic 2.2 design has already been developed. Relating to the understanding of the RF protocols used, we plan to further support our understating of the OTA protocol as some fields are still yet to be determined. Similarly, we'd like to investigate the Serial Protocol used by the gateway as well as research and use ILSpy to determine fields of the Serial Protocol used by the SWIFT Tools application and the W-USB transceiver. Both of these goals support the notion of spoofing a packet by specifying known fields, potentially targeting a vulnerability in the system or triggering a certain state.

References

[1] A. Bussey, D. Chou, M. Fabregas, and S. Wright, "Swift wireless fire alarm system analysis," *Apr.*, 2023.

[2] Leo Rover, "Raspberry pi or arduino – when to choose which?" 2024, retrieved October 17, 2024. [Online]. Available: https://www.leorover.tech/post/raspberry-pi-or-arduino-when-to-choose-which

[3] WebbyLab, "Arduino vs raspberry pi: Comparison," 2024, retrieved October 17, 2024. [Online]. Available: https://webbylab.com/blog/arduino-vs-raspberry-pi-comparison/