SWIFT Wireless Fire Alarm System Analysis

Donald Lawrence
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
dl@gatech.edu

George Kokinda
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
gkokinda3@gatech.edu

Garrett Brown
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
gbrown94@gatech.edu

Andrew Lukman
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
alukman3@gatech.edu

Yeonhak Kim
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
ykim713@gatech.edu

Jack Smalligan
College of Engineering
Georgia Institute of Technology
Atlanta, Georgia, United States
jack.smalligan@gatech.edu

Chris Roberts (Advisor)
Principal Research Engineer
Georgia Tech Research Institute
Atlanta, Georgia, United States
chris.roberts@gtri.gatech.edu

***Abstract*—Fire protection systems play a crucial role in the realm of building management. Despite its importance, there's relatively little literature surrounding this topic in the field of cybersecurity. Over the past decade, building management systems have shifted away from age-old wired systems to more modern wireless systems that offer a higher-level convenience over their older counterparts. Though the benefits are undeniable, the ability to go wireless has opened up building management systems to a new domain of attacks. A malicious actor could assault a wireless fire protection system by triggering false alarms, disabling the system entirely, or partaking in other hostile activities to provoke havoc within a building. This study investigates a product suite from Honeywell (a manufacturer of building management systems) known as the SWIFT system which includes common fire protection components such as an alarm pull station, a smoke detector, a gateway, and more. The purpose of this study is to uncover any vulnerabilities that could lead to attacks on the SWIFT system through RF analysis, serial monitoring, and software/firmware reverse engineering.**

I. Background

Fire alarm systems are vital components to a well-protected environment. These systems are responsible for alerting and protecting against fire emergency situations. Important systems incentivize individuals to capture, manipulate, and pollute the protections put in place. Malicious activities can include all the following: setting off a false alarm to breed distrust, flooding the system to interfere with an alarm, and decrypting any critical encrypted material to gain elevated access to the system. By compromising a fire protection system, malicious actors introduce a powerful tool into their arsenal. Discovering and eliminating weaknesses in the system are important defense strategies that need to be implemented to have a robust protected environment. Eliminating vulnerabilities in the fire alarm system will deter potential malicious actors.

A. The SWIFT System

As technology continues to develop exponentially, the market demand for wireless fire systems will continue to increase as well. Buildings that house large and complex technologies will need to implement a fire safety solution. Despite wireless systems having a significantly higher price than wired solutions, corporations are discovering that the increased price of a wireless fire system outweighs the inconvenience of a wired fire system. Honeywell, a prominent technology company, carries and supports a product known as "SWIFT" (Smart Wireless Integrated Fire Technology) [1]. In addition to being designed to integrate with existing wired Honeywell products, the SWIFT system's product line includes wireless components. The wireless components include a pull station, A/V bases, gateway, modules, and smoke detectors [2]. The SWIFT system utilizes a "mesh network" that allows the various wireless components to interface and interact with one another. Each component in the system services as a repeater for signals as they attempt to travel to the gateway [3]. This means that each wireless device in the mesh network has redundancy, meaning that if a component malfunctions, the system will still function normally as each component has multiple paths to the control panel.

B. Existing Literature

A previous study on the fire alarm pull station almost entirely identified both the OTA (over-the-air) and USB protocols used by the SWIFT system. This analysis will be focusing on the wireless gateway component of the system, which serves as the heart of the SWIFT system. All wireless devices on SWIFT's mesh network must communicate with the gateway in order to reach the fire alarm control panel (FACP). In other

words, the gateway is essentially a bridge between the wireless devices on the mesh network and the FACP. If a bad actor can compromise the gateway, then the entire SWIFT system is compromised, and further access can possibly be gained into the wired side of the system.

Additionally, although examining the OTA messages emitted from the gateway is valuable, the team is pursing analyzing the communication between the gateway and SWIFT Tools (SWIFT's companion application) as its main means for deciphering its messages. This allows for quicker results as it's easier to monitor the interaction and makes the operation of XOR'ing the data by 0xAA unnecessary. However, it's important to note that communication in this fashion would not be possible without SWIFT's W-USB transceiver which will be covered in the following section.

II. W-USB Protocol

The companion application that manages the SWIFT suite, SWIFT Tools, uses a wireless USB transceiver, referred to as the "W-USB," [1] to communicate to the devices on the mesh network. This device is used to send and receive data within 20 feet of where the W-USB is connected to a PC with the SWIFT Tools software installed. The data that the W-USB receives is populated into the companion application and stored. This allows an administrator to overview data about the network and perform administrative actions regarding the devices. This data can be analyzed with the tool Serial Port Monitor [2] which allows the analysis of COM 3 bus traffic between the W-USB and the SWIFT Tools companion application. This USB protocol is found to be like the OTA protocol but with slightly different headers for SWIFT Tools to parse the data.

A. Reverse Engineering SWIFT Tools

Once data is captured using Serial Port Monitor, the headers and payloads need to be analyzed to understand how and why messages are being created and how that data became readable within the SWIFT Tools application. Reverse engineering the companion application which parses and creates these messages is a vital to understanding these messages in their entirety. The team reverse engineered the application using an open-source .NET de-compiler, ILSpy [3]. SWIFT Tools is an unobfuscated .NET application with several supporting Dynamic-Link Libraries (DLLs). The application being unobfuscated means all symbols including interfaces, classes, functions, and global variable names are present. This makes it possible to understand exactly how SWIFT Tools parses the messages coming from the W-USB knowing the exact names the original developers created for each portion of the protocol.

Within the DLLs WirelessComm.dll, WirelessInterfaces.dll, and WirelessPlugin.dll, all parsing and generation of W-USB messages can be found. Each message can be one of two types of general frames. The two frames are the adapter frame, which is used for the W-USB itself, and the node frame which is used for information transfer between each device and the adapter. Both frames contain an open delimiter (7B = ""), message type (1 byte), payload length (1 byte), payload (variable length depending on payload length), CRC (Cyclic Redundancy Check) that is an XOR of previous bytes (1 byte), and a closing delimiter (7D = ""). These fields make up the header of each USB packet. The node frame has two extra fields when compared to the adapter frame. One is a serial number (4 bytes) and the device type (1 byte). This is because the node frame is used to transfer information to and from each device on the network and these fields help direct the message to the intended device. Several of these message payloads have been analyzed and decoded.

B. Device State Messages

Periodically, the W-USB receives messages of type, "BackGroundScanResponse." This message includes information about the device's current state. Inside the de-compilation of the DLL file WirelessPlugin.dll, a class named ScanForm can be found. This class contains two methods, "fillScanDataforDevices" and "fillDevcieScanData." Both methods are responsible for parsing the incoming "BackGroundScanResponse" message types. Following the de-compiled C# code, each field in this message type has been identified knowing the exact location of each in the payload.

The device model is identified by a 1-byte "NodeType" field and the specific identifier for that device is present as a 4-byte serial number. The device model field is responsible in determining the structure of the rest of the message as different models can have different status fields depending on the type of device. For example, in the Honeywell SWIFT suite, the pull station has four battery slots. Within the background scan response message, information such as how many batteries are currently inserted and the time remaining on them are present. Despite this, some fields are general to all devices. These fields include firmware version numbers, application build numbers, SLC addresses, and brand. All fields show as information in which an administrator can view in the SWIFT Tools software see Fig. 1.

One different message structure of the "BackgroundScanResponse" the team identified is the gateway's state message. This message contains all the above fields but with added gateway specific fields. Instead of using the method "fillScanDataforDevices," the method "fillScanDataforgateway" is used because of its NodeType field being equal to the value for a gateway device. These gateway only attributes include configuration values related to gateway options, magnet lock status, mesh network states, and a locked attribute. Some fields of note are the magnet lock status and gateway locked attribute. The magnet lock status it at the time of analysis "MagnetVerfied." This means to change gateway related settings within SWIFT Tools an administrator must place a physical magnet onto the gateway. The locked attribute refers to the option of an administrator to enable a password to modify gateway options. At the time of capture a password was enabled.

These state messages and understanding them are vital to analyzing attack vectors the team implements. The "BackgroundScanResponse" provides valuable information into the state of each device on the mesh network. It is because of this,

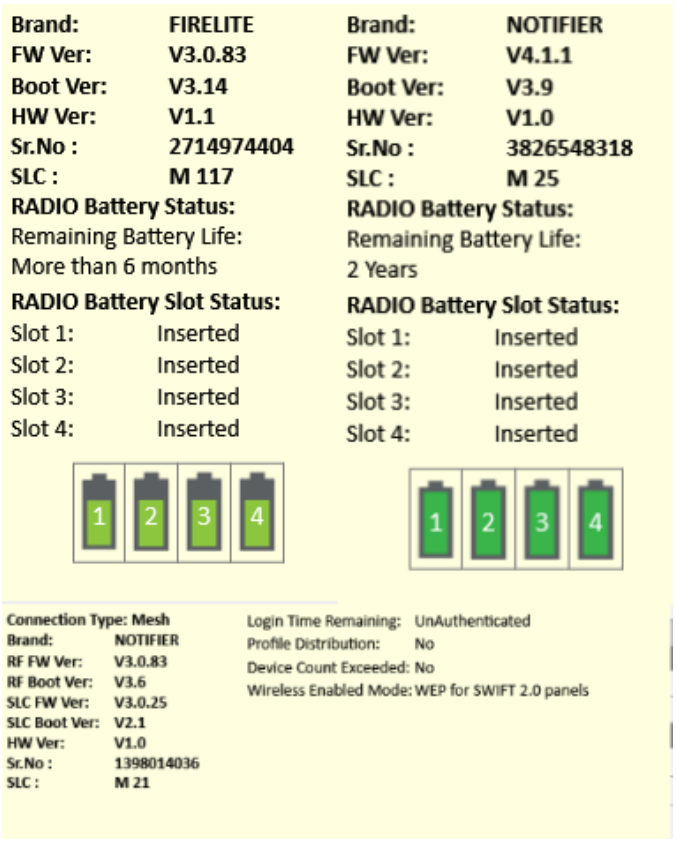

Fig. 1. SWIFT Tools information hover in Diagnostics for the Pull Station (left), Relay Station (right), and gateway (bottom).

understanding each device's attributes help the team understand exactly what an attack vector modified or accomplished pertaining to a device.

C. Other Messages

Other message types the team analyzed are of the node frame structure. These messages typically have small payload sizes if they have a payload at all. The main other messages the team captured were related to gateway functionality and control.

The messages that relate to the gateway password functionality are "VarifyPasswordRequest," "VarifyPasswordReply," "ChangePasswordRequest," and "ChangePasswordResponse." The "VarifyPasswordRequest" message contains a node type and a serial number which usually is equivalent to the gateway's node type and serial number. Then it contains a payload length variable which is the length of the password, and the payload is the password inputted into SWIFT Tools in clear text see Fig. 2. The "ChangePasswordRequest" message type follows an identical format to the "VarifyPasswordRequest" apart from the message type field. Their responses "VarifyPasswordReply" and "ChangePasswordResponse." have instead a 1-byte payload which usually is a "01" if the message was received and the password was successful with all other fields being identical.

```
[22/09/2021 22:03:44] Written data (COM3)
    7b 19 46 54 00 54 53 08 41 6e 64 72 65 77 36 21   {.FT.TS.Andrew6!
    38 7d                                             8}
[22/09/2021 22:03:45] Read data (COM3)
    7b 01 00 01 7d                                    {...}
[22/09/2021 22:03:59] Read data (COM3)
    7b 1a 46 54 00 54 53 01 01 0f 7d ed ed            {.FT.TS...}ii
```

Fig. 2. A ChangePasswordRequest and ChangePasswordReply in Serial Port Monitor with a hex to ASCII translation on the right. The password set at the moment of capture is "Andrew6!".

Other messages relate to firmware updates in SWIFT Tools to a specific device. These messages were captured during a firmware update on the gateway. Before the update, a "BootloaderIn" message is sent to place the gateway into a bootloader mode. The exact specifics of this bootloader mode are unknown to the team currently. This message contains a serial number and node type to identify which device this message is being sent to. It also contains no payload. During the update, AppDownloadResponse and AppDownloadRequest messages are sent. The payload of the AppDownloadRequest message contains a byte-for-byte copy of the hex from the firmware binary. The payload of this message is of size 52 bytes. The first four bytes of the payload are currently unknown, but the remaining 48 bytes are copied directly from the firmware binary. Although it hasn't been proven, the AppDownloadResponse message seems to be a response verifying that the firmware the gateway received was valid or invalid. These messages contain the aforementioned data in addition to the other node frame fields. After the update on the gateway, the gateway becomes automatically unlocked. An administrator will have to relock it if they wish to keep the gateway locked. The messages to relock come after "ChangePassword" message types. The messages are "LockgatewayRequest" and "LockgatewayResponse." These messages contain only node frame fields.

III. Gateway Authentication

Being the controller of the entire mesh, the gateway also offers an authentication system to prevent unauthorized users from making configuration changes. By default, the gateway is in an unlocked state, but once the mesh network is formed, the building maintainer places the gateway into a locked state. Each time the gateway is locked, a new password must be created that will later be used to unlock it. In addition, a Hall sensor is used to ensure the physical presence of the user attempting to unlock the gateway. The user passes this check by simply swiping a magnet over the gateway.

While in a locked state, a user can still access certain characteristics of the gateway, such as battery level or firmware version. Changing device configurations, along with viewing the mesh layout and updating the firmware, can only be done when the gateway is unlocked. This presents a challenge to the team when attempting to find an exploit.

A. Fuzzer

To find a way to bypass the authentication, the team decided to create a fuzzer in an attempt to find a vulnerability in the gateway's software. A fuzzer is a piece of software that manipulates inputs to a system to try to cause some sort of unintended behavior. This unintended behavior could then lead to a security vulnerability. The decision was made to continue the previous team's development, but this time by utilizing the BooFuzz [4] fuzzing framework to speed up the development process. During the development of the fuzzer, the team discovered a vulnerability in the authentication system's logic just by testing different variations of message types.

B. Authentication Bypass

By sniffing the connection between SWIFT Tools and the gateway, the general authentication flow can be discovered (see Fig. 3). Notably, authentication occurs through the use of two different message types: "VarifyPasswordRequest" and "LockgatewayRequest". By omitting the "VarifyPasswordRequest" message and simply utilizing the "LockgatewayRequest" portion of the authentication process, it is possible to unlock the gateway without the use of a password. This means that, although the password check is enforced from inside SWIFT Tools, it is not actually needed on the gateway side to complete the unlock process. However, the magnet step is still required when using this method.

While most sensitive message types are protected by authentication, the message to enter bootloader (update) mode is not. By placing the gateway into bootloader mode, then rebooting, the gateway is placed into an unlocked state, allowing access to the configuration and the ability to upgrade/downgrade firmware. See Fig. 4 for the exact process of fully unlocking the gateway. Although the first method of bypassing authentication still requires the use of the Hall sensor, it does not disrupt the devices on the mesh network, which may be important depending on the situation. Placing the device into bootloader mode causes the mesh to be disbanded and all protection to be disabled, which may arouse suspicion.

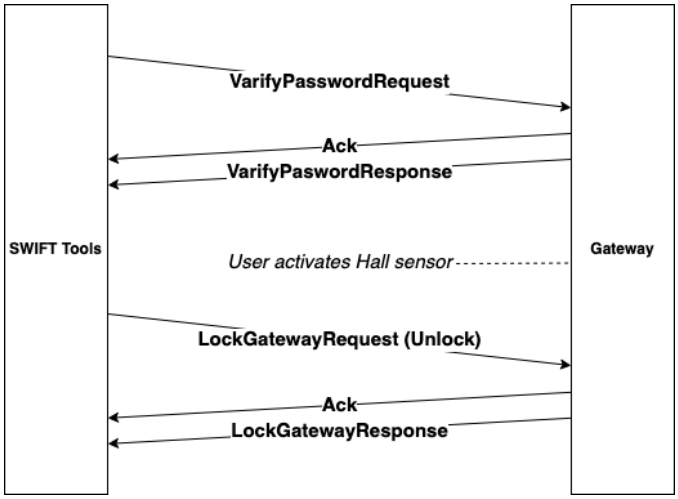

Fig. 3. The communication sent during the usual authentication process

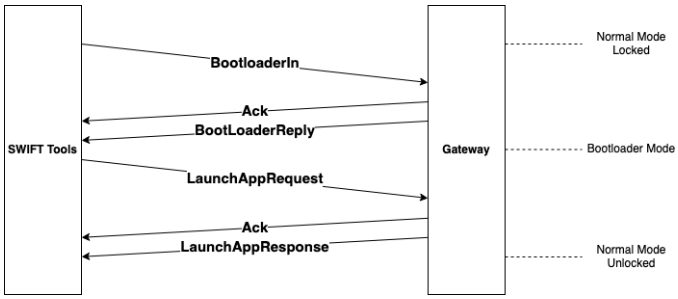

Fig. 4. The process for bypassing the gateway's authentication

IV. Firmware Analysis

The gateway for the SWIFT system utilizes the Texas Instruments (TI) MSP430X family [5] of processors, which is responsible for the gateway's operation at the firmware level. The gateway runs three separate firmware binaries– one for RF communication, another for SLC interfacing, and a third that handles the gateway's bootup procedure. The RF binary will be the main focus of the team's investigation, as it handles all wireless communication and will likely be where the most valuable vulnerabilites lie. Ghidra [6], an open-source tool for the static analysis of software binaries, will be used to better understand the inner workings of the firmware binaries.

A. Firmware Disassembly

Each release of SWIFT Tools includes several versions of the gateway firmware as raw binary files meant to be copied directly to the gateway's internal memory. When one of these publicly available binaries (North American version 3) is loaded into Ghidra, the machine code is partially disassembled by the tool's built-in disassembler. When loaded directly from the publicly available binary, the results of the disassembly are inaccurate, because of address mismatches, which cause sections of data to be disassembled as code and pieces of code to be interpreted as stored data. This problem occurs because the first section of the binary program on the microcontroller is not delivered during the the firmware update process; only the contents after address 0x5C00 is delivered. Shifting the binary forward such that the first address was placed at address 0x5C00 allows a more accurate disassembly to be performed.

In order for Ghidra to perform its automated disassembly analysis, it must be provided with an entry point of the software's operation, from which it will begin disassembly. In programs for the MSP430 family of microprocessors, the entry point is identified with the function label `c_int00()` [7]. The documentation provided by TI for the MSP430 compiler documents the actions taken by the `c_int00()` function, which allowed it to be identified with relative ease. The function, as seen in the tool Ghidra, may be seen below in Fig. 5.

With the disassembly process beginning at `c_int00()`, Ghidra is able to flow through a significant portion of the total machine code, disassembling in the process. However,

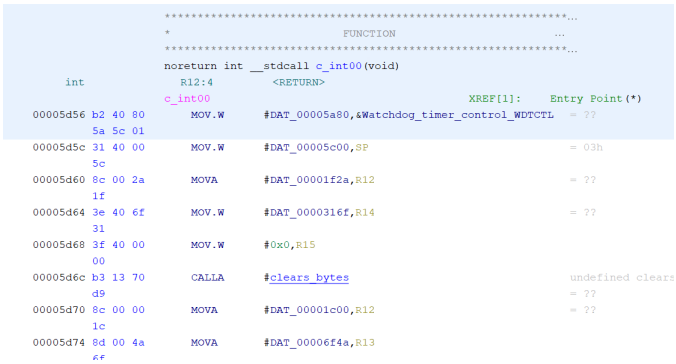

Fig. 5. The function `c_int00()`, as seen in Ghidra. One of the actions it performs that assists with identifying it is the setting of the stack pointer to the address 0x5C00.

Ghidra does not continue disassembly after an instruction that modifies the program counter (e.g., branches, jumps, and returns) if it does not determine that control flow returns to the following instruction. With the use of a script that integrates with Ghidra's scripting API [8], it is possible to proceed past some of these instructions (namely, return statements), and thus disassemble and view much of the remainder of the gateway firmware.

B. Binary Comparison

BinDiff [9] is a comparison tool for disassembled code binaries. In prior research, the live firmware was extracted from the pull station in the fire alarm system as well; to leverage this research in the current study, it is useful to create a comparison between the gateway firmware and the pull station firmware, because it was hypothesized that there would be significant shared code between the two (as they are developed by the same company, for closely related purposes as part of the same system, for the same target ISA, and because some common functions have been identified manually). Owing to the challenges with disassembling the gateway firmware in its entirety (the solutions to which are described above), an initial effort to generate the binary difference showed that more than 70% of the functions have no match between the binaries, and that those that do match have very low similarity ratings as can be seen below in Fig. 6.

The true figures for the percentage of shared functions and their similarity ratings, which were identified after implementing the address rebase, disassembly from the entry point function, and the scripting to proceed past return instructions, are significantly higher. Subsequent analysis revealed that approximately 65% of functions were in common, with many having similarity ratings above 0.9, indicating a very close match. The results of this analysis may be seen below in Fig. 7.

C. BinDiffHelper

BinDiffHelper [10] is a Ghidra extension which makes it possible to copy function labels, variable names, and other data from one binary under analysis to another after a binary

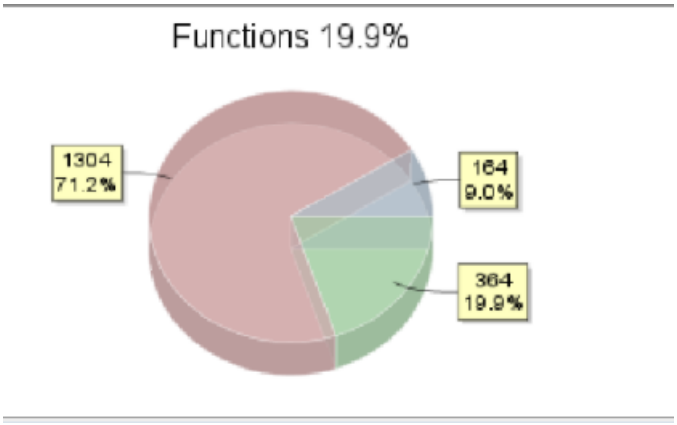

Fig. 6. The output from the tool BinDiff, showing the percentage of common functions between the gateway firmware and the pull station firmware. On the first iteration, slightly fewer than 30% of functions are identified as being in common.

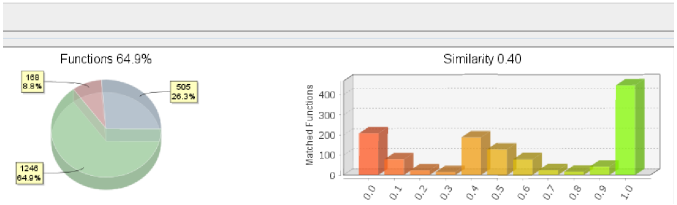

Fig. 7. The output from BinDiff, showing the percentage of common functions and their similarity ratings between the gateway firmware and the pull station firmware. After various solutions were implemented to improve results, more than 60% of functions were identified in common.

difference has been created using BinDiff. By copying this data over and leveraging the tool's ability to create call graphs and identify common function blocks, many functions that were previously studied in research on the pull station could also be identified in the gateway. The tool's interface, which allows the functions to be filtered for copying based on their similarity, can be seen below in Fig. 8.

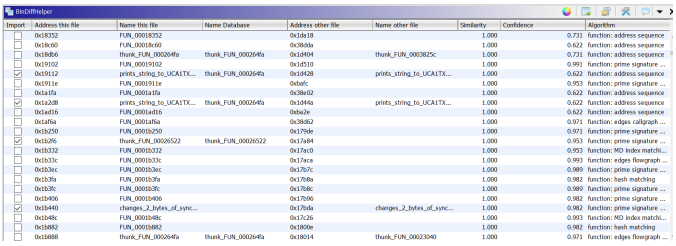

Fig. 8. The interface of the tool BinDiffHelper, showing the options for copying functions between binaries; note also the column listing the function's similarity ratings.

D. OTA Encryption

Honeywell claims that the SWIFT system uses encryption on each of its messages to prevent miscommunication and for security purposes, with each device having its own unique key (which they identify with a "profile") [11]. For this reason, the team hypothesized the presence of an encryption algorithm

in the gateway firmware. The Advanced Encryption Standard (AES) [12] is a widely-used encryption method, and a segment of the data memory of the gateway firmware (as well as a similar segment in the pull station firmware) contains a series of lookup tables that are used in most implementations of AES. One of these tables, called "S-box," can be seen below, both as shown in a reference work on the implementation of AES (Fig.9) and in the disassembled gateway firmware (Fig. 10). With these tables identified, the functions responsible for performing AES encryption were also identified, and may be a subject for future work.

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Fig. 9. The S-box lookup table, as used in standard implementations of AES. These values may be matched with those seen in Fig. 10.

```
                        AES_sbox
00000cbc 63 7c 77       db[256]
         7b f2 6b
         6f c5 30 ...

                        AES_Rsbox
00000dbc 52 09 6a       db[256]
         d5 30 36
         a5 38 bf ...

                        AES_Rcon_missing_first_byte_0x8d
00000ebc 01 02 04       db[10]
         08 10 20
         40 80 1b ...
```

Fig. 10. The AES lookup tables in the gateway firmware, as seen in Ghidra.

E. String Analysis

Many strings are found in the gateway firmware; this section details the team's discoveries concerning their presence and purpose throughout the code. The strings may be separated into two categories: human-readable strings and unreadable strings. In order to distinguish the two categories, the list may be filtered in Ghidra by the "Is Word" attribute; by filtering with attribute as true and false, lists of readable and unreadable strings may be obtained, respectively. The list of human-readable and unreadable strings are shown below in Fig. 11 and Fig. 12, respectively.

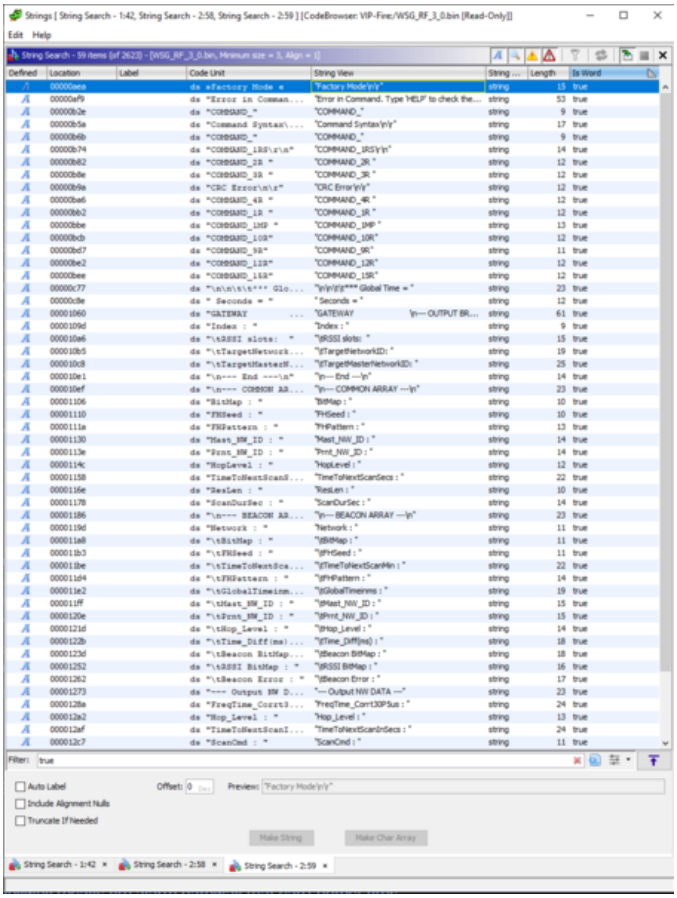

Fig. 11. The list of human-readable strings in the gateway firmware.

From the given lists, readable strings are identified by locating their specific addresses in the assembly code. A number of strings are candidates for further study, including, among others, "GATEWAY", "Index:", "HopLevel", and "Network:". These strings are located in the first section of the assembly code, the data segment. Among these list of strings some have no references throughout the firmware because the relevant sections have not been disassembled (or partially there are instructions incorrectly disassembled).

By filtering with the "Is Word" attribute set to "false," Ghidra produces the list of unreadable strings; this set constitutes the majority of the identified strings, most of which are of unidentified function. Possible reasons for their classification include errant behavior by Ghidra's automatic tool (i.e., they are, in fact, not strings at all) or a type mismatch between the string data type in the tool and the firmware implementations. As a result, it is likely that Ghidra has not correctly interpreted these fields in the disassembly process.

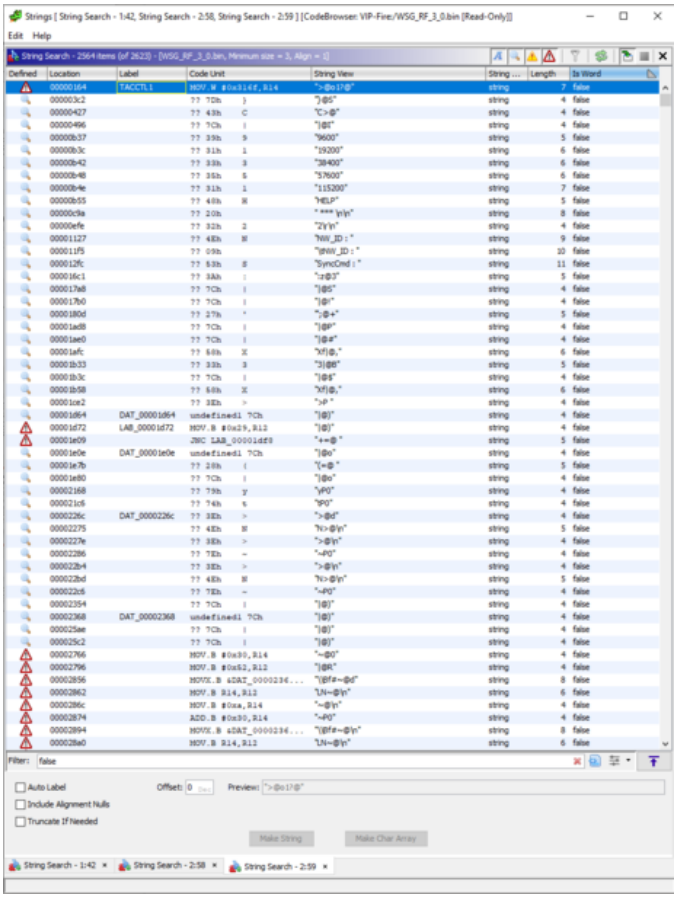

Fig. 12. The list of unreadable strings in the gateway firmware.

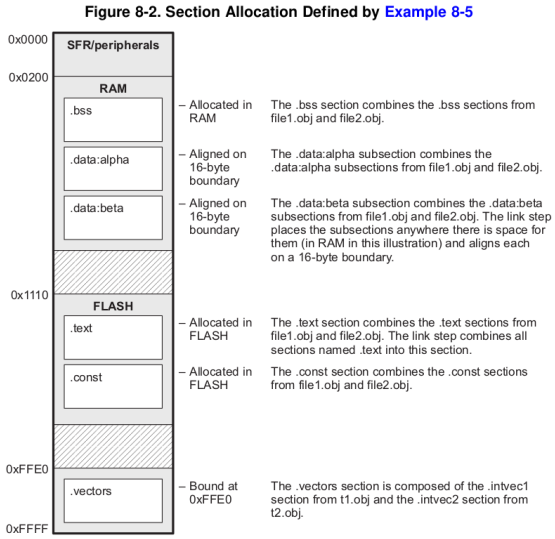

Figure 8-2. Section Allocation Defined by Example 8-5

Fig. 13. Section Allocation Architecture.

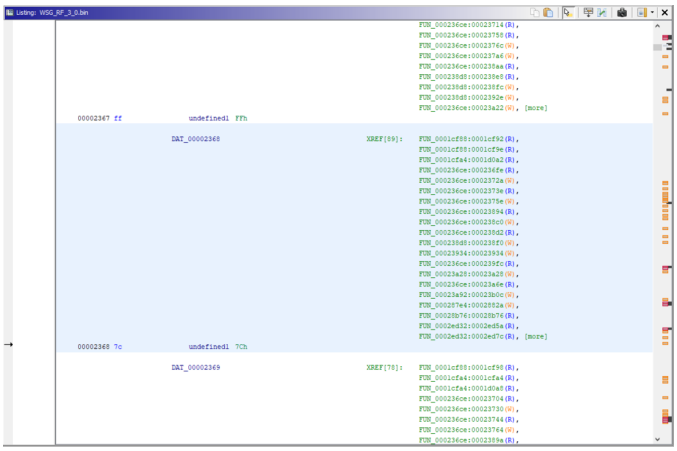

Fig. 14. The list of cross-references to selected data locations in the assembly code.

Some of the unreadable ”strings,” as identified by Ghidra, are miscellaneous data fields, not strings at all. Some of these are referenced frequently throughout the firmware, both in reading and writing operations, depending on the data and the referencing function. Such cross-referencing details may be seen below in Fig. 14.

Some of the strings which were investigated are ”19200”, ”38400”, ”57600”, and ”115200”. These selected strings were referenced by same functions in the firmware; `FUN_0002443c` and `FUN_0003c6c4`. `FUN_0002443c` takes the input command and compares its value with one of these fixed values of strings. It specifically compares two values by passing them to the `FUN_0003c6c4`; the function loops through the string and compares each character and returns 0 if the strings match. After further investigation of these string values and functions they were also found in the pull station firmware which were defined as baud-rate related functionality. As to how the functions are defined, the baud-rate for the device can be configured among the provided values; ”19200”, ”38400”, ”57600”, and ”115200”. Such details can be found in Fig. 15 and Fig. 16.

F. Memory Allocation

After fully disassembling the gateway firmware, it is clear that there is more than one instruction code block in the firmware. According to the MSP430 Assembly Language Tools User’s Guide [13], the processor provides several ways to perform memory allocation. The default memory allocation scheme is defined in a sequential manner. One of the other memory allocation schemes, which corresponds to the gateway firmware, is the section allocation scheme. This scheme consists of multiple code blocks, distributed across different locations. The gateway firmware consists of two or three distinguishable instruction code blocks. According to TI’s documentation, the first block of instruction memory may begin at the memory location of 0x00007274 and the second block at 0x00FFAF, although this does not fully comport with manually identified functions that are located at an address as high as 0x5D56. A diagram representing the section memory allocation scheme is shown below in Fig. 13.

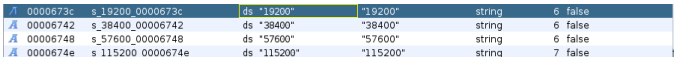

Fig. 15. Fixed values of strings investigated.

Fig. 16. Function referencing strings ”19200”, ”38400”, ”57600”, and ”115200”.

V. Custom Firmware

A. Background

One possible approach for cyber-criminals to attack the SWIFT system is for them to upload their own custom firmware which would allow them to control various devices on SWIFT’s mesh network. While not entirely likely, it’s important to consider every attack vector when securing any piece of hardware. As such, research has been conducted in an attempt to demonstrate control of the Gateway via custom firmware.

B. Custom Service Pack

In order to put a user-created firmware on the Gateway, SWIFT Tools (SWIFT’s companion application) firmware upgrade/downgrade interface requires that the firmware is part of a service pack. In terms of the SWIFT system, a service pack is a collection of firmware upgrade/downgrade binaries that can be used to update the various devices on SWIFT’s mesh network. Each service pack contains a few binaries along with a ”servicepack.config” XML file (Fig. 17.) which contains the general properties of the entire service pack (e.g., Name, Region, Notes, Version, etc.) and the specific properties of each individual firmware file (e.g., FileName, DeviceType, Version, IsBootloader, etc.). SWIFT Tools is shipped with a few service packs (e.g., SP_NA_2.2, SP_NA_3.0, etc.) which correspond to different versions of the firmware. A picture of the contents of the Honeywell provided service pack SP_NA_2.2 can be seen in Fig. 18.

Because Honeywell already provides service packs (SPs), creating a custom SP is as simple as copying and pasting one of Honeywell’s and editing the desired binary. As long as the SP exists within the ”ServicePacks” directory in the SWIFT Tools installation location and it contains a ”servicepack.config” file, then SWIFT Tools will recognize it as a valid service pack. The custom service pack (seen in Fig. 19) created for this procedure has the name SP_NA_9.5 (i.e., Service Pack, North America, version number) which follows SWIFT’s service pack naming convention of SP _continent _version . This SP is a duplicate of SP_NA_2.2 (Fig. 18), except that the version numbers have been changed in order to make it easily distinguishable from

```xml
<?xml version="1.0" encoding="utf-8"?>
<ServicePack>
  <ServicePackVersion>9.5.1</ServicePackVersion>
  <ServicePackName>SP_NA_9.5</ServicePackName>
  <Region>NorthAmerica</Region>
  <Notes>SP_NA_9.5</Notes>
  <FirmwareDetails>
    <Firmware>
      <DeviceType>Gateway</DeviceType>
      <FileName>WSG_RF_9_0.bin</FileName>
      <Type>RF</Type>
      <Version>9.5.20</Version>
      <IsBootloader>false</IsBootloader>
    </Firmware>
    <Firmware>
      <DeviceType>Gateway</DeviceType>
      <FileName>WSG_SLC_9_0.bin</FileName>
      <Type>SLC</Type>
      <Version>9.5.40</Version>
      <IsBootloader>false</IsBootloader>
    </Firmware>
    <Firmware>
      <DeviceType>Gateway</DeviceType>
      <FileName>WSG_BU2_RF_9_0.bin</FileName>
      <Type>RF</Type>
      <Version>9.5</Version>
      <IsBootloader>true</IsBootloader>
    </Firmware>
  </FirmwareDetails>
</ServicePack>
```

Fig. 17. The servicepack.config XML file used with the custom serice pack

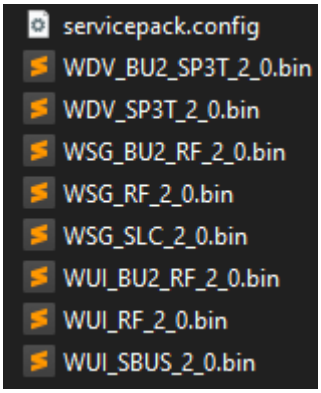

Fig. 18. Honeywell provided SP_NA_2.2 service pack

the other SPs. Since the focus is the gateway, all non-gateway binaries were stripped out of the service pack. With this custom service pack loaded, SWIFT Tools firmware upgrade/-downgrade interface recognized the service pack as seen in Fig. 20.

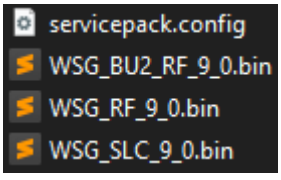

Fig. 19. The custom service pack

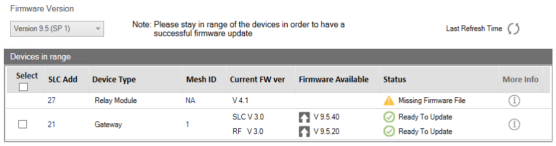

Fig. 20. SWIFT Tools upgrade/downgrade interface with custom service pack present

C. Custom Firmware Binary

Within the created custom SP, there are three gateway binaries: WSG_BU2_RF_9_0.bin, WSG_RF_9_0.bin, and WSG_SLC_9_0.bin. Each corresponds to a different component of the Gateway (i.e., Bootloader, RF, and SLC respectively). In order to prove that the custom firmware is living on the Gateway, it is critical that any bytes changed in one of the binaries are significant to the OTA (over-the-air) communication to or from the Gateway. As such, the gateway's RF binary WSG_RF_9_0.bin was the only firmware that was changed in this SP. The other two binaries are identical to those provided by Honeywell (WSG_BU2_RF_2_0.bin and WSG_SLC_2_0.bin) in SP_NA_2.2.

From examining the SWIFT Tools C# decompilation as described previously in the paper, it was determined that the eighth-to-last byte of every firmware binary is used to define a variable called `HardwareVersionSupported` which is a property of each firmware file. This variable is actually a list, so it can contain multiple values, but regardless it only ever contains 1.0 or 1.1. This value appears to correlate with the hardware version (HW_Ver) field which is seen on various devices in SWIFT Tools. The gateway specifically has a value of 1.0 for this field inside SWIFT Tools as seen in Fig. 21.

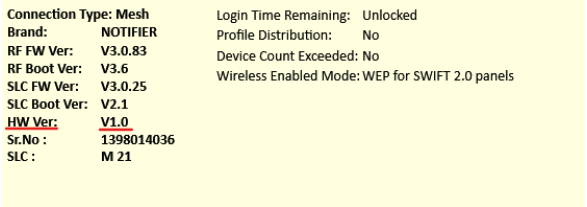

Fig. 21. Gateway HW Ver number in SWIFT Tools

Since SWIFT Tools is distributed as an unobfuscated .NET application and due to poor programming by Honeywell, the decompiled C# code tends to be accurate in terms of variable naming. This means it's safe to assume that the HardwareVersionSupported variable means one of two things: 1.) this variable is directly linked with the HW Ver field seen inside SWIFT Tools, or 2.) this variable is used to check if the real HW Ver bytes yield a version that is supported. Either way, by changing this byte it should provide the significant change needed to confirm that the custom firmware is on the Gateway because the HW Ver number has already been found in the serial communication with the Gateway as described in the W-USB Protocol section.

By running SWIFT Tools through a .NET debugger called dnSpy (similar to ILSpy), it was determined that changing the 8th byte back from 08 to 17 which will change the HardwareVersionSupported variable from "1.0" to "2.7". This change to the firmware binary should be visible not only in SWIFT Tools, but also in the OTA/serial communication with the Gateway. With this small change to the RF binary, the team initiated the firmware upgrade/downgrade process within SWIFT Tools.

D. Upgrade Results

Upon selecting to upgrade the firmware of the Gateway within SWIFT Tools, the entire communication between the SWIFT W-USB and the Gateway was recorded using the application Serial Port Monitor (SPM). Without examining these captures, it was clear that the custom firmware upgrade was unsuccessful due to the "Corrupted Firmware" status seen within the SWIFT Tools firmware upgrade/downgrade interface (Fig. 22). Fig. 22 also shows that the unchanged SLC firmware was uploaded successfully, but the RF binary was not.

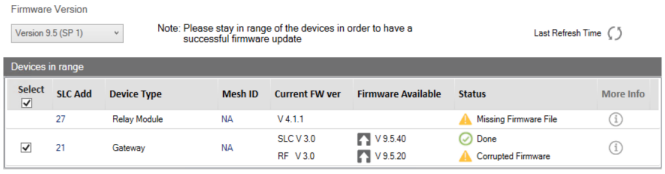

Fig. 22. "Corrupted Firmware" message inside SWIFT Tools

E. Upgrade Rejection Inside SWIFT Tools

By analyzing the SPM capture, the team found that the firmware upgrade failed during a LaunchAppRequest message. Since the firmware update failed during a LaunchAppRequest message (returning a 0 value within its payload), looking at the SWIFT Tools decompilation using ILSpy yields insight into the behavior of the firmware failure within the application. Within the "Honeywell.WirelessTool.WirelessPlugin.ScanForm" class, the team found a method named LaunchAppCode. This method deals with the sending of the LaunchAppCode message type. Depending on the result of the response from the device, the LaunchAppCode returns a value in accordance with the success of the operation. It checks the payload of the LaunchAppResponse message the device generates and obtains the value of the one-byte payload within the message type. If this value is a 0 it will return a value that is not zero meaning it was unsuccessful. This value is processed by the parent function and the value returned corresponds to a device status enum. The DeviceStatus enum class is shown in 23. The value returned from this parent method results in an enum value of 3 which corresponds to the value "Corrupted." This enum also corresponds to the message "Corrupted firmware" that is present in the GUI.

```
public enum DeviceStatus
{
    BootloaderVersionMismatch,
    Cancelled,
    CommunicationLost,
    Corrupted,
    DeviceNotSupported,
    Done,
    Failed,
    FileNotPresent,
    HardwareVersionMismatch,
    InProgress,
    LowBattery,
    PasswordRequired,
    Pending,
    ProvidingFireProtection,
    Ready,
    UnKnown,
    ClassA,
    UptoDate,
    Ineligible,
    Aborted
}
```

Fig. 23. Enum class showcasing the device status values.

Completely understanding the behavior in SWIFT Tools is crucial to ruling out any validation checks of the firmware update process present in the client. The values recovered from SWIFT Tools included in the DeviceStatus enum class can be helpful for future analysis of the binary as the device generates these messages during the firmware update process. This allows the team to narrow their focus in understanding the firmware update process device side.

F. Tools for Serial Port Monitor Analysis

Analysis of serial port monitor text dumps can be difficult and time consuming. This is because the ability to filter out irrelevant information and use automated tooling for deeper analysis is hindered by the information that Serial Port Monitor adds to the text dumps. A script was created to remove timestamps and hex to ASCII generated by Serial Port Monitor, remove extraneous bytes added after the end message delimiter, and place each message separately on its respective line number. The tool also allows for the ability to extract specific message types and filter out any message types that the user does not want to extract. Multiple message filtering is also supported. This is helpful for filtering out re-transmissions and NACK messages that can crop up during listening.

With this tool, other automated tools can be used to process the data such as diff, a command line difference tool that can compare files line by line. The firmware update process yields over 6000 messages communicated over USB between SWIFT Tools and the device. Parsing through all information and determining where, if any, differences between the valid firmware and custom firmware uploads can be quickly determined using diff. The results of this difference between the firmware update from version 2.2 to version 3.0 using the custom firmware compared to the valid update yielded only a difference mentioned previously in the LaunchAppResponse message. The payload was a 00 instead of a 01 in the valid update. Even though the results were not fruitful, the tool can be of use in the future for rapid analysis of Serial Port Monitor message dumps.

VI. Gateway Internals

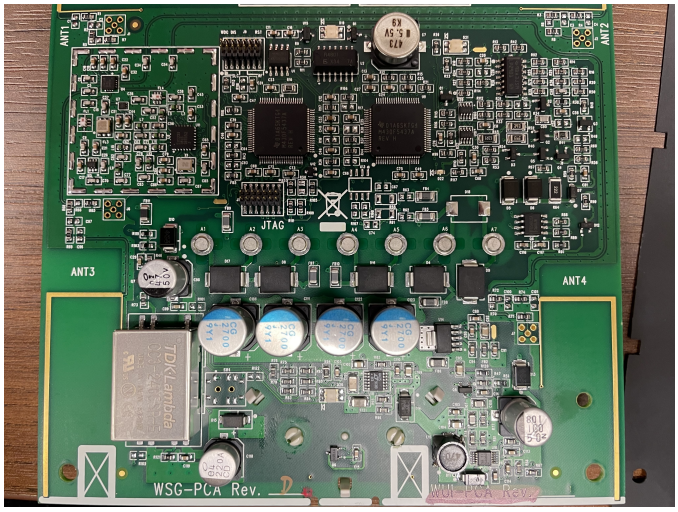

Fig. 24. The wireless gateway removed from its shell

A. Background

As mentioned previously, the SWIFT gateway is meant to interface the wireless capabilities of the SWIFT system with an existing SLC-based FACP. Rather than use a single processor with a complicated multi-tasking setup, Honeywell opted to include two discrete MSP430X processors on the gateway board that communicate via a serial channel. These two processors can be found in the upper portion of the board as seen in Fig. 24. Additionally, a JTAG header is available to directly connect to either of the processors for debug purposes.

B. Firmware Update Mechanism

The board's JTAG port can be utilized to dump the RF processor's memory during different parts of the firmware update process, which allows several details to be seen that

could not be found during the static analysis of the update files. Firstly, the RF chip orchestrates the update process for all three of the firmware binaries. During the first portion of the update, it receives the SLC update and passes it directly to the SLC processor via UART. Afterwards, the bootloader firmware is updated, with the RF update being received and applied last. Rather than using the included BSL mode of the MSP430 family of processors, the requisite functions for the update included in the bootloader firmware are moved to RAM with execution resuming from there, allowing for the internal flash to be overwritten without issue.

Secondly, as deduced from the failed custom firmware package, there is some sort of verification process being performed on each binary, presumably through a checksum embedded somewhere in the binary's data. Unfortunately, a method for bypassing this check has not been determined yet, which will likely be the next order of business for a future team.

VII. CRC Verification

During a firmware update. The MSP430 chip within the gateway provides a CRC-CCITT standard module. This CRC implementation is widely used, so it is well-documented and the exact code for the implementation found in the firmware can be found [14]. As it turns out, a one-to-one implementation of this algorithm is found to exist inside the firmware update binary for the gateway as analyzed in Ghidra. Analyzing the last several bytes of the firmware binary in Ghidra yielded a cross reference to a function that passes memory addresses and lengths to another function. This other function, with the arguments passed, performs the CRC-CCITT algorithm on the specified memory addresses. The exact memory blocks this function creates the CRC for within the binary are currently unknown, as more research is needed.

An analysis of the parent functions relating to the CRC check yields an execution path to the `c_int00` function. The function, `c_int00` calls a function that starts a firmware update process. This is referenced in one of the parent functions relating to the CRC check. A function that prints the string `MU_START` is called in one of these parent functions. The string `MU_START` can be referenced in the SWIFT Tools decompilation standing for Mesh Update. This coincides with the subsequent functions that call the CRC-CCITT algorithm and check if these values return 0 successfully. The decompilation of this code is seen in Fig. 25 which determine if the CRC is valid.

Turning to SWIFT Tools for information regarding the key memory addresses used during the firmware update may provide useful information. In the class `Honeywell.WirelessTool.WirelessPlugin.ScanForm` the `downloadAppCode` method is analyzed again as it uses memory address values related to the firmware update process. These methods are named `LoadAddressesBeforeVersion3` and `LoadAddressesAsPerFileFooter`. Within code and by their respective naming convention SWIFT tools utilizes these

```
if (DAT_00002c94 == '\x02') {
  uVar2 = related_to_CRC_Check();
  if ((char)uVar2 == '\0') {
    DAT_00002028 = 3;
    DAT_00002c94 = 1;
    return 3;
  }
  cVar1 = FUN_00037948();
  if (cVar1 == '\0') {
    DAT_00002028 = 4;
    DAT_00002c94 = 1;
    return 4;
  }
  cVar1 = FUN_00038d4e(&DAT_00001903,0x44);
  if (cVar1 != '\0') {
    DAT_00002028 = 5;
    DAT_00002c94 = 1;
    return 5;
  }
  cVar1 = FUN_00038a86(0x1800,0x7f);
  if (cVar1 != '\0') {
    DAT_00002028 = 6;
    DAT_00002c94 = 1;
    return 6;
  }
```

Fig. 25. CRC parent function calling CCITT algorithm.

functions to determine what memory addresses in the device correspond to the data found in the firmware binary for update. With version 3 and up, SWIFT Tools changed the method of determining this from static values in the application to values appended at the end of the firmware file. The static values within the `LoadAddressesBeforeVersion3` are shown in 26 with their values in hexadecimal shown to the right. However, the function to analyze is the `LoadAddressesAsPerFileFooter` as the custom firmware upload is utilizing version 3. The function takes the last 30 bytes and uses them along with some mathematical manipulation to determine the values shown in the static version. A re-implementation of the function in another programming language (Python) determined that version 3 contains the same values except for `IPacketFirst` which is a value lower than the hard coded version. This value deals with the number of packets to send and does not have to do with memory addresses. The link between these values and CRC is to be determined and further research is needed.

VIII. Conclusions

By continuing to analyze the decompiled code generated by reverse engineering Honeywell's companion application called "SWIFT Tools", more information has been found on the firmware upgrade/downgrade process that devices like

```
// Honeywell.WirelessTool.WirelessPlugin.ScanForm
private void LoadAddressesBeforeVersion3()
{
        ISTARTFIRMWARE = 23504; //= 0x5BD0
        IStARTSECONDBLOCK = 65488; //= 0xFFD0
        ISTARTUSBFIRSTBLOCK = 17360;
        ISTARTUSBSECONDBLOCK = 65488;
        IPacketFirst = 651;
        ISecondPacket = 4587;
        IPacketFirstUSB = 619;
        IPacketSecondUSB = 1728;
}
```

Fig. 26. LoadAddressesBeforeVersion3 shown in the ScanForm class.

the gateway undertake. Further, string analysis in Ghidra is moving the team closer to finding the encryption keys used to encrypt the OTA messages emitted from the Gateway. Through the use of Serial Port Monitor and the fuzzer, the team has deciphered the meaning of more message types and found a notable vulnerability in the SWIFT system. If this vulnerability in the gateway were exploited, a bad actor could bypass two levels of authentication and disable SWIFT's mesh network which would render all wireless devices in the mesh network incapable of communicating with the fire alarm control panel (FACP). Research and effort has been put towards uploading a custom firmware to the Gateway which will be possible only after determining the specifics of the CRC validation.

Additionally, the team has performed extensive analysis on the gateway firmware, including the development of scripts that allow for improved disassembly. While the exact method of verifying the firmware has not been deduced yet, much progress has been made in determining exactly how the update process works and how the RF processor writes to flash.

Future goals for research on the SWIFT system and in particular the gateway include: converting a larger portion of the firmware's assembly into a readable C code in Ghidra, finding the encryption keys used to encode the OTA messages emitted from the gateway, learning more about the encryption being used, bypassing the CRC validation used when upgrading firmware, and sending a valid team-created firmware binary to the gateway. With this information, the team will be able to execute a successful cyber-attack and demonstrate full control of the Gateway and thus the entire SWIFT system.

References

[1] Honeywell. W-usb swift transceiver. [Online]. Available: https://www.firelite.com/en-US/Pages/Product.aspx?category=Wireless\%20Fire\%20Alarm\20Solution&cat=HLS-FIRELITE\&pid=W-USB

[2] Serial port monitor track and analyze the activity of your system com ports. [Online]. Available: https://www.eltima.com/products/serial-port-monitor/

[3] Ilspy is the open-source .net assembly browser and decompiler. [Online]. Available: https://github.com/icsharpcode/ILSpy

[4] boofuzz: Network protocol fuzzing for humans. [Online]. Available: https://github.com/jtpereyda/boofuzz

[5] Texas Instruments. MSP430x5xx and MSP430x6xx Family User's Guide. [Online]. Available: https://www.ti.com/lit/ug/slau208q/slau208q.pdf

[6] National Security Agency. Ghidra. [Online]. Available: https://ghidra-sre.org/

[7] Msp430 optimizing c/c++ compiler v21.6.0.lts. [Online]. Available: https://www.ti.com/lit/ug/slau132y/slau132y.pdf

[8] National Security Agency. Ghidra script. [Online]. Available: https://ghidra.re/ghidra_docs/api/ghidra/app/script/GhidraScript.html

[9] Zynamics. Bindiff. [Online]. Available: https://www.zynamics.com/bindiff.html

[10] Bindiffhelper. [Online]. Available: https://github.com/ubfx/BinDiffHelper

[11] Honeywell. Frequently-asked questions about swift. [Online]. Available: https://www.securityandfire.honeywell.com/notifier/en-us/latesttopics/frequently-asked-questions-about-swift

[12] National Institute of Standards and Technology (NIST). Announcing the advanced encryption standard. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[13] Texas Instruments. MSP430 Assembly Language Tools User's Guide. [Online]. Available: https://www.ti.com/lit/pdf/SLAU131Y

[14] Crc-16 routine. [Online]. Available: https://cs.fit.edu/code/svn/cse2410f13team7/wireshark/wsutil/crc16.c