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

Daniel Chou
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
dchou33@gatech.edu

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

Sidney Wright
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
swright92@gatech.edu

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

***Abstract*—The market for fire protection systems has been transitioning from traditional wired networks to modern wireless mesh networks that offer superior convenience and management tools for building administrators. However, in addition to convenience, wireless networks expose new opportunities for attack that were not possible on wired systems. Despite the danger that these vulnerabilities can pose for users of a wireless fire system, relatively little research has been conducted in this area. This study investigates and uncovers the vulnerabilities in the well-known building management manufacturer Honeywell's SWIFT system, specifically with regards to its SLC connection to previously existing wired devices, including pull stations, smoke detectors, and the fire alarm control panel.**

I. Background

As important parts of building safety, fire alarm systems protect against fire emergency situations in which it is necessary to notify building occupants and begin evacuation protocols immediately. Gaining control of a building's fire protection wirelessly would allow attackers to conduct malicious activities such as deploying ransomware, triggering false alarms, delivering false information to the building's monitoring system, or impairing fire detection and mitigation. With these attacks available, attackers gain tools to exploit encrypted material for elevated access to the system. Removing the vulnerabilities in such a system improves its ability to provide the protection it was designed to deliver and defends against inevitable cyber attacks.

A. The SWIFT System

The SWIFT system is a departure from the methods used by Honeywell's other building management tools. Rather than being based on a traditional wired setup, SWIFT components communicate wirelessly, which allows for use cases that a traditional fire alarm system would not be able to fill. Additionally, the SWIFT system is designed to integrate into existing wired setups, supplementing the existing devices with wireless ones. This strategy allows customers to take advantage of new technologies without a full system replacement. While Honeywell is offering many of the same wired devices in a wireless format, such as smoke detectors, audio/visual alarm systems, and pull stations, there is a device specific to the SWIFT system that allows for its integration with existing wired systems, known as the wireless gateway. Although the SWIFT system is designed around a mesh network to provide redundancy, all wireless connections must be routed through the gateway in order to reach the rest of the wired system. Thus, the gateway serves as a single point of failure, making it a prime target for exploitation.

Although the gateway is a singular device, it actually contains two separate co-processors based on the TI MSP403X architecture. One processor, dubbed RF, handles communication with the wireless devices on the mesh network, and also provides management functionality such as firmware updates and mesh configuration. The other processor, dubbed SLC, translates the messages sent by the wireless devices into a format that the existing wired FACP (fire alarm control panel) can understand. These wired devices communicate via SLC, a communication method that allows for addressable control of wired devices.

B. Prior Research

Previous research by the team [1] on the SWIFT system examined the OTA (over-the-air) and serial (W-USB) protocols used by the system. In this study, a HackRF One was utilized to capture the RF signals being emitted from SWIFT's fire alarm pull station. Thorough analysis of the OTA signal inside Universal Radio Hacker led to the initial partial decoding of the OTA protocol. The reverse engineering of SWIFT Tools (a companion application used to manage the wireless network) aided in decoding the serial protocol, and a systematic comparison between the two protocols resulted in the full decoding of both the OTA and serial protocols. This research has proven useful in determining how firmware updates are transmitted to the various devices on the mesh network.

Another prior study [2] on the SWIFT system analyzed the wireless gateway component of the SWIFT system. The gateway serves as the heart of the SWIFT system because all wireless devices on the mesh network must communicate with the gateway in order to reach the Fire Alarm Control Panel (FACP), which does not have the ability to communicate with the wireless devices directly. In this study, the team examined the gateway's firmware binaries by utilizing the Ghidra reverse engineering suite. Additionally, this research launched efforts towards uploading a custom firmware binary to the gateway. An authentication bypass was found that allows for any actor with a wireless dongle to gain control of the gateway's management functions. This research has been critical to the progress made during this semester, as the team aims to build upon this existing work by continuing analysis of the gateway's firmware, while also making great strides toward a full takeover of the fire alarm system.

This paper will focus on 1.) an analysis of the gateway's SLC firmware, 2.) the decoding of the SLC protocol, and 3.) uploading a custom firmware to the gateway.

II. SLC Binary Analysis

A. Preparing The Binary

To jump-start analysis of the SLC binary, the team has made attempts to use BinDiff [3], which was previously utilized for the RF binary, to see if relevant analysis on previous binaries could be transferred. After building BinExport [4], the Ghidra extension for exporting analyzed binaries in the format BinDiff requires, for the current version of Ghidra (10.1.2), diffs between the SLC binary and both the RF and wireless pull station firmware can be performed, as those are the binaries the team had analyzed in previous semesters. These diffs did not yield many common code blocks, as seen in 1, which did not come as a surprise, considering that the SLC and wireless sides of the gateway differ at a structural level, and also use different protocols for communication.

To be able to communicate with the physical SLC lines, the SLC firmware needs to make liberal use of the peripherals provided by the MSP430 processor. As documented in the MSP430 family user's guide [5], as well as the datasheet for the specific model used in the gateway [6], these peripherals are mapped to a certain address range in the processor's memory. The team utilized a symbol mapping developed in a previous semester and the "ImportSymbolsScript.py" script from within Ghidra to automatically import these symbol labels.

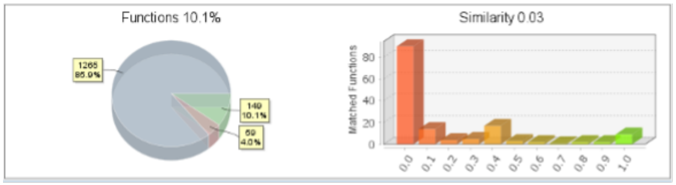
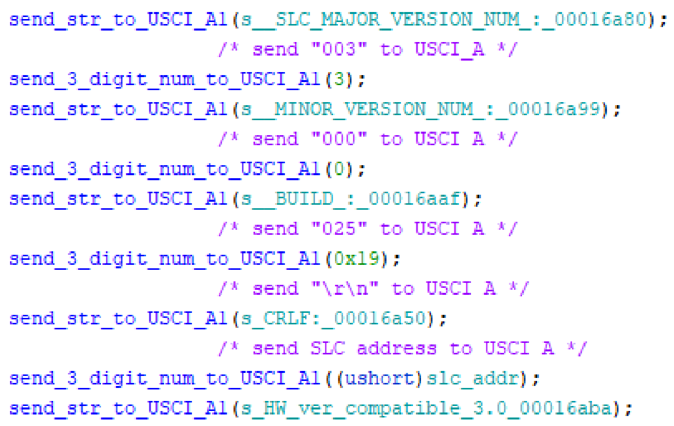


Fig. 1. BinDiff results on SLC firmware vs RF firmware. The green area of the pie chart represents the number of functions which show some similarity between the two binaries. However, most of these functions end up being 10 instructions or less.



```
send_str_to_USCI_A1(s__SLC_MAJOR_VERSION_NUM_:_00016a80);
                    /* send "003" to USCI_A */
send_3_digit_num_to_USCI_A1(3);
send_str_to_USCI_A1(s__MINOR_VERSION_NUM_:_00016a99);
                    /* send "000" to USCI A */
send_3_digit_num_to_USCI_A1(0);
send_str_to_USCI_A1(s__BUILD_:_00016aaf);
                    /* send "025" to USCI A */
send_3_digit_num_to_USCI_A1(0x19);
                    /* send "\r\n" to USCI A */
send_str_to_USCI_A1(s_CRLF:_00016a50);
                    /* send SLC address to USCI A */
send_3_digit_num_to_USCI_A1((ushort)slc_addr);
send_str_to_USCI_A1(s_HW_ver_compatible_3.0_00016aba);
```

Fig. 2. Example of debugging strings sent to USCI A in the function "main_loop".

The last task necessary to complete before analyzing the SLC firmware was to correctly set up the memory map and entry point for Ghidra. Again, as documented in the user's guide, the program code starts with a function called `c_int00`, located at address 0x5c00 in memory. After separating the memory addresses into blocks in Ghidra's "memory map" window, and marking the address 0x5c00 as an entry point, Ghidra's auto-analysis could start the reverse engineering process.

B. USCI A Debugging Output

When looking at the decompiler's output for the SLC firmware, one function immediately stands out, as its only parameter is strings such as "`SLC MAJOR_VERSION_NUM :`" and \`HW ver compatible 3.0`" (2). This function can be quickly identified as sending the string one character at a time to a memory-mapped peripheral register labeled "`UCA1TXBUF`". Cross-referencing this with the MSP430 user's guide, it seems that these strings are being transmitted to a Universal Serial Communication Interface (USCI) port, likely for debugging or logging purposes. A major benefit of having this function being used is that it gives reliable information on the purpose of functions that would otherwise take a lot more effort to discern.

C. Identifying The Main Loop

One of the functions called in `c_int00` ends with a "`do-while true`" loop, which likely contains the logic for relaying messages from the wireless mesh network to the fire alarm control panel (FACP) via the SLC. This would imply that the code before this loop initialized processor peripherals, global data structures in memory, and the like.

D. *Initialization Functions*

Building on this idea, the team decided to start by looking at these initialization functions, so that the purposes of various peripherals and global data structures used in message processing would be clearer. Most of these functions are ordinary housekeeping, such as initializing the power management module, clearing memory, and logging firmware version numbers to the USCI port mentioned earlier. However, one interesting function sets up the chip's direct memory access (DMA) module to copy bytes received via another USCI port, USCI B1, into a 200-byte long buffer in memory, and to copy bytes in a different buffer to that port's transmit register. Although the team initially conjectured that this port may be interfacing with the SLC output, further investigation, detailed below, casts doubt on this hypothesis.

E. *USCI B Messages*

Looking at the messages received and processed by the gateway, it seems that these consist of a small 6-byte header, followed by a number of command packets, identified by their first byte. So far, there appear to be 8 distinct commands, one of which is a sync command, and another is a boot command that appears to perform a soft reset of the gateway. Two other commands seem to handle adding and removing devices to the SLC, but that hypothesis has not been confirmed. The purposes of the other commands has not yet been determined.

F. *Ports*

Another goal of the firmware analysis is to determine how the SLC chip communicates with the physical SLC wire, passing information from the wireless devices to the FACP. This would obviously be done through a port, of which the MSP430X architecture provides two types - standardized serial ports, such as USCI A1 and B1, discussed previously, and general-purpose IO ports, which are 8 bits wide with each bit being individually configurable as input or output. Of these, the firmware appears to regularly use only ports USCI A1, USCI B1, general-purpose port 1 for both input and output, and general-purpose ports 2 and 8 strictly for output. However, during initialization, general-purpose ports 5, 6, and 7 are also used as output. While it is clear that port USCI A1 is used to assist with debugging, the purpose of the other ports was not as clear. Therefore, the team attempted to perform continuity tests on the gateway hardware to better understand what each port's pins were connected to.

G. *Continuity*

While these tests were limited due to time constraints, the preliminary results were interesting. The team prioritized two tasks: determining what the USCI B1 port on the SLC chip is connected to, and determining which ports, if any, could be used to control the SLC output. For the former, the team was unable to determine a conclusive connection between the USCI B1 port and any other wire on the gateway. However, the multimeter used to perform the tests did show a non-zero reading when testing connectivity between the USCI B1 port and several pins on the RF processor. Since there was no beep indicating a clear connection, this result warrants further investigation. As for the ports controlling SLC output, the first observation the team made was that since the SLC line carries a 24 V voltage, it is unlikely that any GPIO pin on the processor is directly connected to it, as these are rated for a maximum voltage of 5 V or less. However, there were multiple definitive connections between pins on both processors and the SLC output wire. Again, this will likely be a high priority goal for the team's future work.

H. *Firmware Analysis*

From observing decompiled outputs from Ghidra, it was found that the SLC firmware uses two USCI channels to communicate with the external devices. One is the USCI_A channel and the other one is the USCI_B channel. According to the MSP processor manual, it provides 3 different USCI modes which are UART, I2C, and SPI. By further investigating program texts and string values in the decompiled program, it was discovered that the firmware was using SPI mode (Synchronous Peripheral Interface). The specific related portions revealing the mode can be found in the Ghidra by searching the key words "synchronous" and "SPI." A string "Reinitialize SPI DMA" was passed onto the function send_str_to_USCI_A1(param_1) which then passes the value as byte stream into the function send_byte_to_USCI_A1(param_1). The final destination of the corresponding function is USCI_transmit_buffer_UCA1TXBUF. There was a corresponding buffer named USCI_receive_buffer_UCA1RXBUF. Thus, there were two types of buffers found which one (transmit buffer) handles outgoing data and the other (receive buffer) handles incoming data. According to the naming conventions listed in the MSP manual, buffers starting with UCA[buffer_number]TXBUF stands for USCI_A transmit buffer and UCB[buffer_number]TXBUF stands for USCI_B transmit buffer (UCA[buffer_number]RXBUF and UCB[buffer_number]RXBUF are receive buffers for each corresponding types). Two types of functions were related to USCI_A and USCI_B channels for receiving and sending data which were send_byte_to_USCI_A1(param_1) and USCIB1_receive_buffer0(param_1). Functions for receiving data from USCI A channel and sending data to USCI B channel were not found or decoded.

III. Signaling Line Circuit Protocol

A. *Background*

Signaling Line Circuit (SLC) is a data and power bus that transmits vital information and power between the devices that make up a complete fire alarm system [7]. The National Fire Protection Association (NFPA) classified SLC in such a way that it can be implemented on fiber optics and wireless transmitting devices as well as electricity as it is a protocol. This differs from traditional fire alarm circuitry. The traditional or ”conventional” circuit is binary in nature, meaning it is either off or on, while SLC is more akin to a network cable which has data bursts and several types of signals. Once a device is done carrying a signal then another may go through the line.

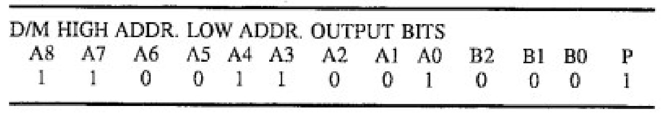
The SLC protocol itself contains many diverse types of messages. These messages range from simple polling from the Fire Alarm Control Panel (FACP) to seeing which devices are on the network and receiving the polling message as well as alarm signals incoming to the FACP from a pull station or smoke detector. Each addressable device is programmed into the control panel. The panel polls a specific device address. Once that device replies the next device is polled. This means that each SLC message gets broadcasted to all other devices on the network. However, this is not the only way the FACP populates the network. SLC can support conventional devices as well by means of only supplying power to them without the need for polling.

SLC provides both power and signals. The information that can be seen on the line is data bursts of SLC messages being sent to devices and in between these data bursts there is a length of time for non-data which is when the addressable modules charge their internal SLC power circuit capacitors. Most of the time SLC has full voltage. This voltage is interrupted occasionally for transfer of data. The way data is transferred is rapidly turning off power and turning it on (shorting the circuit) which can be interpreted as binary bits. This binary data itself is proprietary to the manufacturer.

Each SLC data packet can be expressed in two sections. The first section, located at the beginning of the message, contains the address information relating to the specific device the message is being sent to as well as information regarding the address or name of the sender. The way the SLC protocol can populate this field is proprietary. The second section is monitoring information and commands being sent to devices. The first data packet of information is an address of the device the message is being sent to then additional digital information or analog information as well. The information within these packets is all proprietary to the manufacturer. [8]

B. FlashScan Protocol

FlashScan is the branding of a patented advanced SLC protocol created by Honeywell that enables communication with a plurality of devices over traditional bidirectional serial wiring. Compared to the older Classic Loop Interface Protocol (CLIP), which polls devices in the loop sequentially, FlashScan is able to address groups at a faster rate with the same detector and control module hardware [9]. While CLIP is backward compatible with FACPs that do not support FlashScan, Honeywell’s SWIFT gateway does not support CLIP as a means to connect to wired components through an FACP [10]. As a result, the FlashScan protocol is the key to understanding the SLC component of the gateway, and the patent itself contains an extensive explanation, including bit sequences, timings, and example commands. [11]



D/M	HIGH ADDR.				LOW ADDR.				OUTPUT BITS			
A8	A7	A6	A5	A4	A3	A2	A1	A0	B2	B1	B0	P
1	1	0	0	1	1	0	0	1	0	0	0	1

Fig. 3. Example of a single device bit pattern sent to poll control unit 99. [11]

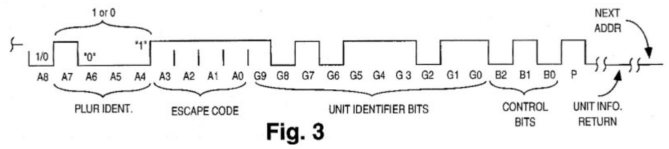


Fig. 4. Breakdown of address word and command word sent using FlashScan SLC [11]

Utilizing the patent, a foundation for the principles of the protocol can be made although the exact implementation of the gateway could be a slightly modified version. Single unit addressing is achieved by reading bits A0-7 as an address, two digit decimal for example. No unit identifiers are provided in this case.

To accomplish group addressing, bit streams are assembled with a address word along with a command word. The address word is then broken down into words signifying the plurality identifier, escape code, unit identifiers, and control bits, as displayed in Fig. 4. When identifying groups, the protocol creates two types of devices denoted by 1 or 0 in bit A8: controls units, such as pull stations, and detectors, such as smoke and temperature detectors. Bits A0-3 signify an escape sequence for group addressing mode, varying per system but hexadecimal F for example. Each group 0-9 corresponds with bits A4-7, which represents the group number to be addressed. Within the group, bits G0-9 serve as unit identifiers, deciding which devices in the group should read the bit stream according to a preset unit address. A group of control units or detectors can be sent an address and command word, checking if their respective bits are set, executing the command, and returning signal for a command confirmation or a sensor reading, respectively. In order to return a sensor reading, detectors can simultaneously draw current from the SLC line, allowing the FACP to compare it to a preset threshold to detect an alarm condition. Additionally, bits B0-2 determine the command to be executed for control/command units. Finally, the last bit P is a parity bit used as a checksum to detect errors in reading signals. After the checksum, there is an information interval of variable time, depending on the command selected, that allows for characters to be transferred between the FACP and a device. The interval is composed of sub-intervals where, in this instance, each character is 305 µs. The protocol is not limited by the implementation of the information transmission, but examples of varying speeds are provided. It is currently unclear whether the gateway uses one of the communication protocols outlined in the patent.

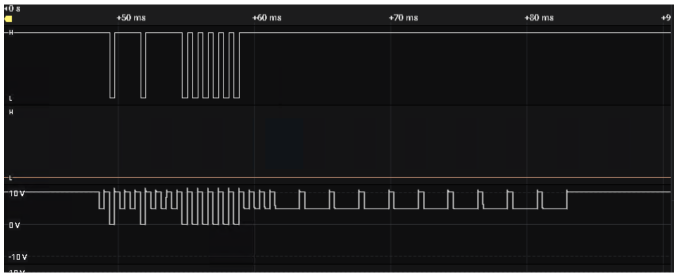


Fig. 5. Example SLC message capture in Logic Analyzer. The message above is digital and the message below is its analog counterpart.

C. Captured SLC Signals and Analysis

SLC signals were captured using Salae's Logic Analyzer tool and analyzed using the companion software Logic Analyzer. The Logic Analyzer was left running while the system was idle and only two different messages were captured at this time. The team, at this time has not identified the specifics of the messages, however according to SLC documentation they are presumed to be polling messages between devices. This has not been confirmed at the time of writing.

The length in time of one of the messages was measured to be 11.769ms long. This message repeats about every 98.75ms. The other message was measured to be 9.5ms long which repeats in the same length of time as the previous message.

Computing the time differences between value changes as exported utilizing Logic Analyzer's export as CSV function, a time array of each time difference can be created. That is for each value in the array time[], the value of `time[i + 1] - time[i]` is calculated. Calculating this over the SLC values and finding the minimum yields a time of 0.3748ms between value changes (i.e., a 1 to a 0 and vice versa). Dividing by the total time of message one and dividing by 8 bits in one byte a value of around 4 bytes in each message can be arrived to. The team at the time of writing has not confirmed this value to be correct and was arrived at with the data captured.

The team built a script in python to convert these signals into bytes. Utilizing the threshold value, it is possible to subtract it from each time value until the current time value being subtracted is less than the threshold value itself. From here a bit is placed corresponding to the value captured at this time. In summary, for each threshold value amount of time we take the corresponding value at that time and create a bit stream extrapolating the assumptions made earlier. The exact bytes being created are not confirmed to be correct, however, the script can be extended to include different timing constraints and values. More research is needed to modify the script to generate a definitively correct byte stream.

IV. Firmware Integrity Checks

A. Background

When examining potential attack vectors for the SWIFT system, the SWIFT's wireless gateway is liekly near the top of the list in terms of targets for cyber-criminals. The gateway's bridge-like characteristics, connecting the wireless devices of SWIFT's mesh network to the FACP, scream of a single point of failure for compromising the system. One way for a bad actor to obtain control of the gateway is through uploading a custom firmware to the device. SWIFT's companion application, SWIFT Tools, makes this process easy enough, and although it's not entirely likely, it's the job of security professionals to consider every possibility. The team's previous research [CITE 2] found taking control of the gateway to be a difficult task. The following research is conducted with an intermediate goal in mind: bypass the gateway's MSP430 CRC check. In other words, upload any useless custom firmware as a proof of concept. Future efforts will look at the possibility of demonstrating control of the system.

An issue faced by the team has been the lack of insight into how the CRC check is being performed by the gateway. The gateway is utilizing a rather strange setup of overlaying two separate firmware binaries in the same address space. This made it difficult to develop an overarching understanding of the exact process of firmware verification. Relatively early on, it was found that the built in CRC functionality of the MSP430X processor family was not being utilized. Rather, the CRC check is being done in software. A lookup table for a CRC16-CCITT algorithm was found embedded in the gateway's BU firmware, and this became the starting point for analysis within Ghidra, by examining which processor instructions reference this block of memory.

B. Gateway Bootup Process

At powerup, code execution begins in the BU firmware, a small binary that contains logic for checking integrity of the rest of the device's flash memory, along with a reduced set of commands receivable over RF that allow for recovery of the device in the event of a failed update. The wireless gateway utilizes a simple CRC16 algorithm to ensure data integrity, which can be bypassed by determining the location of the reference checksum value in each firmware binary. The RF and SLC processors each contain their own bootup firmware, but the team is focusing on the RF side, as it handles wireless firmware update functionality, which is of particular interest.

6 illustrates the general layout of the RF processor's flash memory, showing the overlaying of the RF and BU firmwares as well. The BU firmware performs a separate check of each block of memory shown, entering an infinite loop if the BU portion fails the checksum, and entering a special recovery mode if the RF portion fails. By utilizing the debug features provided by the TI Code Composer software, the team can find the exact ranges of memory being accessed, allowing for a tool to be developed that will calculate the CRC for an arbitrary firmware binary.

C. CRC Calculation Tool

Upon deciphering the memory ranges for which the CRC calculation is performed in the `WSG_BU3_RF_3_0.bin` and `WSG_RF_3_0.bin` binaries, the team created a tool that reads these two SWIFT-provided firmware files and performs

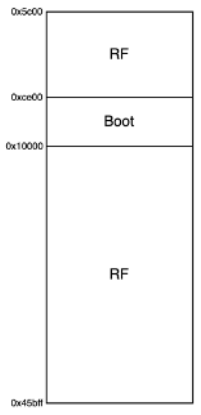


Fig. 6. A high-level diagram of the RF processor's memory layout.

a correct CRC calculation. Further, the tool is capable of replacing the hard-coded CRC values in these two binaries with the newly calculated CRC value – in essence patching the firmware (ensuring it will pass the gateway's MSP430 CRC verification). The tool described is a python script that accepts the gateway's firmware files as input. Currently, the script is limited to performing accurate CRC calculations and patching for the two aforementioned binaries; however, the script can easily be extended to account for any version of SWIFT's provided binaries. Running the script in its most basic form simply prints out the calculated CRC value for the firmware. In this form, the script is most useful for when paired with an Honeywell-provided binary as it allows for verification of what the hard-coded CRC value is. Running the script with the -p flag causes the firmware to be patched with calculated CRC value. In this form, the script is most useful when paired with a custom firmware that needs to have its CRC adjusted so it can be uploaded to the Gateway as a valid firmware.

V. Firmware Upload Protocol

A. Background

SWIFT Tools, the companion application that manages Honeywell's SWIFT suite, offers a firmware upgrade/downgrade interface in its user interface. However, in the world of a bad actor, spending valuable time operating a GUI is likely not realistic. On top on this, SWIFT Tools isn't natively available on Linux systems. In order to bypass this restriction, a bad actor could attempt to re-implement SWIFT's firmware upload protocol for the Wirelesss Gateway (WSG) in their own scripts. To test the feasibility of this, this section explores what is SWIFT's firmware upload protocol for the WSG and how can it be re-implemented.

B. Wireless Gateway Upload Protocol

Previous research by the team [1] [2] has explored the reverse engineering the SWIFT Tools application in an open-source .NET decompiler called ILSpy. Since SWIFT Tools is an unobfuscated .NET application with several supporting DLLs (dynamic link libraries), ILSpy allows for the parsing of all of SWIFT Tools' symbols including interfaces, classes, functions, and global variables. One DLL labeled WirelessPlugin.dll contains most of if not all functions related to the firmware upload protocol. More specifically, it contains a function named firmwareUpdateScanObject() that handles the firmware upload process for the Gateway. Through analysis of this function, the firmware upload protocol for the Wireless Gateway can be deciphered and broken into six steps.

The first step in the firmware upload protocol is to place the Gateway in bootloader mode by sending the Gateway a packet with the BootloaderIn message type (2f in hex). Placing the Gateway in bootloader mode largely reduces its functionality and disbands the mesh network. This mode is used by the Gateway for receiving firmware updates. If SWIFT Tools detects that the Gateway is already in bootloader mode through a previous background scan, then this step will be skipped. This message type is sent once per processor which amounts to two times (one for SLC and one for RF). This step is implemented by the putInBootLoaderMode() function which is called inside the firmwareUpdateScanObject() function.

The second step in the firmware upload protocol is to select which of the Gateway's processors (RF or SLC) will have its firmware upgraded/downgraded by sending the Gateway a packet with the SelectionRequest message type (61 in hex). Since the Gateway has two MSP430 (specifically MSP430F5437A) processors on its circuit board, it's necessary to select which processor to upgrade/downgrade before uploading firmware to the Gateway This step is implemented by the selectionRequest() function which is called inside the firmwareUpdateScanObject() function.

The third step in the firmware upload protocol is to prepare the processor selected in the previous step for a firmware update by sending the Gateway a packet with the AppEraseDownload message type (5b in hex). As of now, the exact purpose of the AppEraseDownload message type hasn't been strictly defined, but there's two working ideas: 1.) erases the selected processor's non-BU (Boot-Up) portion of firmware before a firmware update or 2.) notifies the selected processor to allow for its firmware to be overwritten by the incoming firmware update. This step is implemented by the eraseAppCode() function which is called inside the firmwareUpdateScanObject() function.

The fourth step in the firmware upload protocol is to transfer all the firmware bytes (from the selected firmware binary) to the Gateway by sending the Gateway a packet with the AppDownloadRequest message type (5d in hex). The AppDownloadRequest packets contain two important components in its payload: 1.) the memory address at which the firmware bytes in the remainder of the payload will be stored in the memory of the selected processor, and 2.) a byte-for-byte copy of the firmware hex stored in whichever binary was selected for use in the firmware upload process. As expected, packets with this message type take up almost the entire time in the upload process since they contain the actual firmware due to it taking 6000 plus AppDownloadRequest packets to send all the firmware bytes to the Gateway. This step is implemented by the downloadAppCode() function which is called inside the firmwareUpdateScanObject() function.

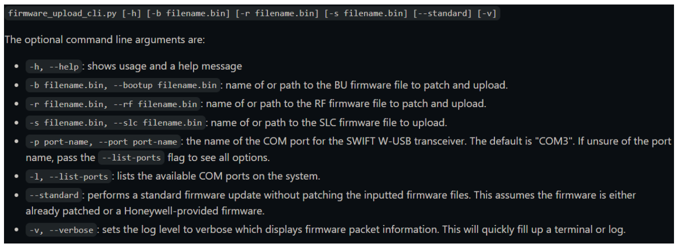
The fifth step in the firmware upload protocol is to boot up the Gateway after all of its firmware has been uploaded by sending the Gateway a packet with the LaunchAppRequest message type (5f in hex). In step 1, the Gateway was placed in bootloader mode in order to complete a firmware update; however, following the update the Gateway needs to boot back into normal mode. In order to do so, SWIFT Tools sends this message type to tell the Gateway to wake up (into normal mode). This step is implemented by the launchAppCode() function which is called inside the firmwareUpdateScanObject() function.

The sixth and final step in the firmware upload protocol is to start a live background scan on the devices on SWIFT's mesh network by sending the Gateway a packet with the StartLiveEventsWithBackgroundScan message type (49 in hex). The StartLiveEventsWithBackgroundScan tells the Gateway to send status updates on all the devices on SWIFT's mesh network including itself. This step isn't essential to the firmware upload protocol (in terms of re-implementation) and can almost be considered not a part of it, but it does allow SWIFT Tools to make sure that the firmware update didn't corrupt the Gateway or the mesh network.

C. Upload Protocol Re-implementation

The team developed a module to streamline the firmware upload process for Honeywell's SWIFT Gateway by re-implementing the protocol used by SWIFT Tools. The module is made up of two useful tools for facilitating the upload process: 1.) a command line interface (CLI) and 2.) a graphical user interface (GUI). Both tools achieve the same end-result and are interchangeable in terms of functionality. The tools carry out a successful firmware update by implementing the six steps discussed in the previous section in python. This means that the tools engage in real meaningful communication with the Gateway rather than attempting an easier form of communication such as a replay attack.

The CLI tool is called firmware_upload_cli.py and its usage can be seen in Fig. 7. This python script accepts up to three firmware binaries (i.e., SLC, BU, and RF), patches the BU and RF firmware (using the previously discussed CRC tool), and uploads the inputted firmware to the Wireless Gateway. The GUI tool named firmware_upload_ui.py is the optional graphical user interface for the firmware upload process as seen in Fig. 8. The GUI tool essentially uses the firmware_upload_cli.py as its back-end and performs the same upload process in a graphical manner. The only physical requirement of utilizing these tools is that the SWIFT W-USB must be plugged into the system on which the scripts are run. The tool has been tested not only on Windows 10, but also on a Raspberry Pi to confirm compatibility with Linux systems and to demonstrate how easy it would be for a bad actor to use a similar device.



```
firmware_upload_cli.py [-h] [-b filename.bin] [-r filename.bin] [-s filename.bin] [--standard] [-v]
```

The optional command line arguments are:

- `-h, --help`: shows usage and a help message
- `-b filename.bin, --bootup filename.bin`: name of or path to the BU firmware file to patch and upload.
- `-r filename.bin, --rf filename.bin`: name of or path to the RF firmware file to patch and upload.
- `-s filename.bin, --slc filename.bin`: name of or path to the SLC firmware file to upload.
- `-p port-name, --port port-name`: the name of the COM port for the SWIFT W-USB transceiver. The default is "COM3". If unsure of the port name, pass the `--list-ports` flag to see all options.
- `-l, --list-ports`: lists the available COM ports on the system.
- `--standard`: performs a standard firmware update without patching the inputted firmware files. This assumes the firmware is either already patched or a Honeywell-provided firmware.
- `-v, --verbose`: sets the log level to verbose which displays firmware packet information. This will quickly fill up a terminal or log.

Fig. 7. Command line interface usage

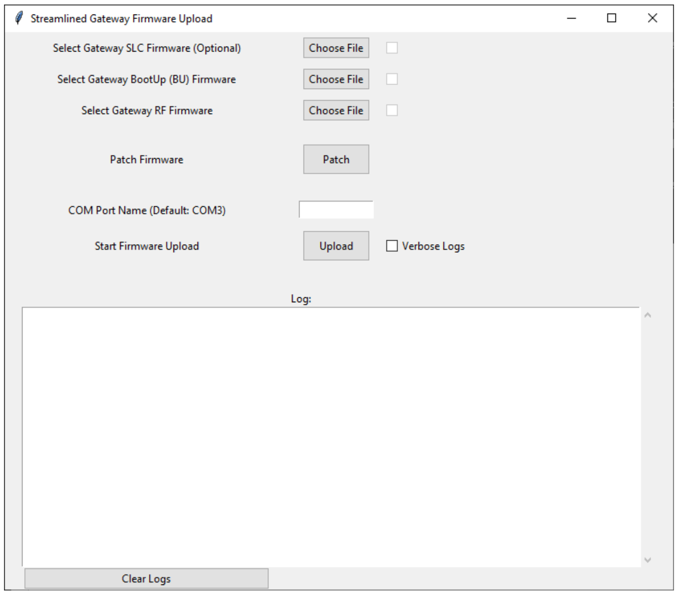
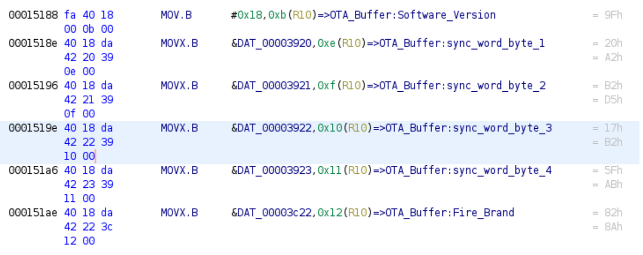

Streamlined Gateway Firmware Upload
Select Gateway SLC Firmware (Optional)
Choose File
Select Gateway BootUp (BU) Firmware
Choose File
Select Gateway RF Firmware
Choose File
Patch Firmware
Patch
COM Port Name (Default: COM3)
Start Firmware Upload
Upload
Verbose Logs
Log:
Clear Logs

Fig. 8. Graphical user interface for uploading firmware

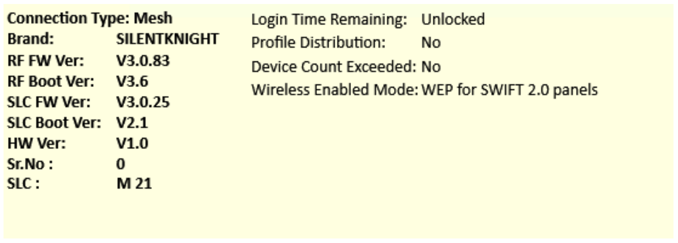
VI. Firmware Modification

With the firmware update CRC verification bypass, a substantial modification to the firmware can be created. A malicious actor can modify the firmware in such a way as to create malware to upload to the device. The beginnings of this can be demonstrated by modifying the RF signals that the gateway sends to the W-USB to be processed by Honeywell's companion application, SWIFT Tools. By modifying the RF signals one can demonstrate control of the device and easily verify that the modifications came to fruition by referencing the data displayed in SWIFT Tools.



```
00015188 fa 40 18    MOV.B     #0x18,0xb(R10)=>OTA_Buffer:Software_Version          = 9Fh
         00 0b 00
0001518e 40 18 da    MOVX.B    &DAT_00003920,0xe(R10)=>OTA_Buffer:sync_word_byte_1  = 20h
         42 20 39                                                                   = A2h
         0e 00
00015196 40 18 da    MOVX.B    &DAT_00003921,0xf(R10)=>OTA_Buffer:sync_word_byte_2  = B2h
         42 21 39                                                                   = D5h
         0f 00
0001519e 40 18 da    MOVX.B    &DAT_00003922,0x10(R10)=>OTA_Buffer:sync_word_byte_3 = 17h
         42 22 39                                                                   = B2h
         10 00
000151a6 40 18 da    MOVX.B    &DAT_00003923,0x11(R10)=>OTA_Buffer:sync_word_byte_4 = 5Fh
         42 23 39                                                                   = ABh
         11 00
000151ae 40 18 da    MOVX.B    &DAT_00003c22,0x12(R10)=>OTA_Buffer:Fire_Brand       = 82h
         42 22 3c                                                                   = 8Ah
         12 00
```

Fig. 9. Shown is the software version, sync word, and brand being placed into the OTA buffer in the `builds_RF_message` function.



Connection Type: Mesh		Login Time Remaining:	Unlocked
Brand:	**SILENTKNIGHT**	Profile Distribution:	No
RF FW Ver:	**V3.0.83**	Device Count Exceeded:	No
RF Boot Ver:	**V3.6**	Wireless Enabled Mode:	WEP for SWIFT 2.0 panels
SLC FW Ver:	**V3.0.25**		
SLC Boot Ver:	**V2.1**		
HW Ver:	**V1.0**		
Sr.No :	**0**		
SLC :	**M 21**		

Fig. 10. Shown is the information panel of the gateway in SWIFT Tools and the results of modification of `builds_RF_message`. Note the serial number and brand fields.

The function that handles the creation of the RF buffer in memory is present at location 0x16842 hereinafter referred to as `builds_RF_message`. The first if condition within the function contains a check for a memory address 0x3C24 if the value is not 6F which during normal execution 0x3C24 = 6F so the if condition fails and most of the code within the function is skipped. After this if condition, the payload for the `BackgroundScanResponse` message starts being created beginning at memory address 0x4A0B with the value 0x6F being written to that address. The exact description of the value is unknown currently. Then, the rest of the OTA (over the air) buffer is built utilizing several memory addresses for a total size of 238 bytes at memory address ranges 0x4A0B through 0x4AF8. The next byte is hardcoded to be 0xEE which is 238 in decimal. This is the length of the buffer. Next the serial number is the next 4 bytes. The serial number is pulled from a different location in memory. This location, 0x3C21 through 0x3C27, provides the serial number as well as node type, brand, and hardware version. This location in memory is written to during bootup i.e., when the processor is reset back to address 0xCE00. During an initialization function, the flash memory info location D (0x1800 through 0x187F) is read and placed into the addresses 0x3c21 through 0x3c27. Other fields are read directly from the flash memory as in the Application build number fields which read from the info A area (memory locations 0x1980 through 0x19FF) of flash memory [5]. The sync word is similar in that it is read from info B area (0x1900 to 0x197F) of flash memory in an initialization function and placed into RAM at location 0x3920 through 0x3923 9. The constant fields that are specific to the device are found in flash memory, the variable fields that depend on the current state of the device are found in RAM and is read periodically by the `builds_RF_message` function, and other fields are built into the code as constants within the function. An example of a dynamic state of the device being read in RAM is the magnet lock status which is read from memory address 0x3743. This value is changed depending on the current state at bootup during an initialization function and written to at that address. Some locations are hardcoded like the `RFApplicationBuildNumber` written to the buffer utilizing a constant value found in the `builds_RF_message` function.

Utilizing Texas Instruments,' Code Composer, one could change the values, utilizing the debugging functionality, at the addresses of where the `builds_RF_message` reads from in the process of building the OTA buffer 10. Modifying these values yields the expected values within SWIFT Tools, however these modifications are not persistent during resets. This is because during a reset, the static values of the device are pulled from flash memory which cannot be arbitrarily written to. The other constant values within the `builds_RF_message` function are trivial to change. The bytes themselves that make up the constants can be modified in the instruction and sent as a modified firmware update. This can be done for more substantial code modifications as well to get around the requirements to write to flash memory and modify the code in the `build_RF_message` function to produce any desired values that the device transmits. Further research into firmware code modifications is still needed.

VII. Conclusions

By continuing to reverse engineer the Gateway's SLC firmware in Ghidra, more information has been discovered about the gateway's SLC protocol. In Logic Analyzer, capturing SLC signals from the FACP has enabled the team to start decoding the SLC protocol. Further, the combination of reverse engineering the Gateway's BU and RF firmware in Ghidra paired with dynamic analysis of the gateway's live memory with a physical debugger had lead to the bypassing of the Gateway's MSP430 CRC verification. In turn, uploading a custom firmware to SWIFT's Gateway is now possible.

Future goals for research on the SWIFT system are: continued reverse engineering efforts in the gateway's SLC binary in order to discover more about the SLC protocol, use of the Logic Analyzer tool to collect and decode portions of the SLC protocol, and determining how to alter a custom Gateway firmware to demonstrate control of the SWIFT system. With this information, the team will be able to execute a full-fledged attack by blocking/injecting messages in the SLC network (e.g., prevent the a fire alarm message from reaching the FACP) or by controlling the entire SWIFT system through altering the gateway's firmware (e.g., have the gateway send out false messages to wireless devices).

References

[1] D. Lawrence, G. Kokinda, G. Brown, J. Jeung, and C. Roberts, "Swift wireless fire alarm pull station analysis," May 2021.

[2] D. Lawrence, G. Kokinda, G. Brown, A. Lukman, Y. Kim, J. Smalligan, and C. Roberts, "Swift wireless fire alarm pull station analysis," Nov. 2021.

[3] Zynamics. BinDiff. [Online]. Available: https://www.zynamics.com/bindiff.html

[4] BinExport. [Online]. Available: https://github.com/google/binexport/tree/main/java

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

[6] ——. MSP430F543xA, MSP430F541xA Mixed-Signal Microcontrollers. [Online]. Available: https://www.ti.com/lit/ds/symlink/msp430f5419a.pdf?ts=1645687841030

[7] S. Mahoney. A guide to fire alarm basics – initiation. [Online]. Available: https://www.nfpa.org/News-and-Research/Publications-and-media/Blogs-Landing-Page/NFPA-Today/Blog-Posts/2021/04/14/A-Guide-to-Fire-Alarm-Basics-Initiation?icid=W483

[8] D. Krantz, *Make It Work - Addressable Signaling Line Circuits*. Douglas Krantz, 2021.
[9] *Notifier SLC Wiring Manual*, Honeywell, 2011.
[10] *SWIFT® Smart Wireless Integrated Fire Technology Manual*, Honeywell, 2020.
[11] E. Bystrak and A. Berezowski, "Enhanced group addressing system," U.S. Patent 5 539 389, Jul. 23, 1996.