SWIFT Wireless Fire Alarm System Analysis

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

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

Darshan Singh
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
dsingh93@gatech.edu

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

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

Wolf Helm
University of North Georgia
Dahlonega, Georgia, United States
wchelm6495@ung.edu

Bryan Casalini
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
bcasalini3@gatech.edu

***Abstract*—SWIFT (Smart Wireless Integrated Fire Technology) is a line of fire alarm devices produced by Honeywell. The product line consists of many different devices like wireless smoke and heat detectors, pull stations, alarms, as well as monitoring software and control panels. SWIFT devices communicate wirelessly, reducing the need for extensive cabling and conduits. This is marketed to simplify installation and allows for flexible placement of devices. Unfortunately wireless communication can be vulnerable to hacking or tampering, especially if proper security measures are not implemented. Unauthorized access to the system could potentially compromise its functionality or lead to false alarms. This study aims to investigate the vulnerabilities of the Honeywell SWIFT system by reverse engineering the systems firmware, and targeting it with precise attacks.**

I. System Introduction

Fire alarm systems play a critical role in safeguarding lives and property by promptly alerting building occupants and local fire authorities to the immediate danger of a fire. Traditionally, these systems have relied on physical wiring to connect detection nodes, notification devices, and the fire alarm control panel (FACP). However, wired systems present various limitations, including reduced detector sensitivity over time and complexities in system modification and expansion.

To address these challenges, companies like Honeywell have begun developing wireless fire alarm systems that have significantly improved fire safety technology. Honeywell's specific system is known as the SWIFT (Smart Wireless Integrated Fire Technology) System and it offers a blend of wired and wireless capabilities to enhance flexibility and scalability while maintaining reliability. The SWIFT system uses a gateway device to control the the wireless mesh network and integrate it with the original wired system. Due to its critical nature, gateway is the primary focus of this project [1].

The gateway contains two processors: the SLC (Signaling Line Circuit) processor and the RF (Radio Frequency) processor. Both processors communicate with each other through a UART (Universal asynchronous receiver-transmitter) channel. The SLC processor is responsible for interfacing with the wired devices and managing communication with the physical SLC line. The RF processor manages the wireless mesh network, relays wireless device information to the SLC processor, and handles wireless communication. Additionally the gateway has three firmware files that are essential for the operation of the gateway and its components, the bootloader firmware, RF firmware, and SLC firmware [2]. Reverse engineering these files could provide us with valuable information on the gateway and will help us achieve our goal of gaining control over the system.

II. Previous Research

The previous teams were able to obtain information on various Node Type variables and their corresponding values in memory from the SWIFT Tools compatibility software. Additionally, the teams proposed and designed a surrogate device to send spoofed SLC messages using an Adurino unit and various supporting hardware modules to handle the difference in voltages between the two devices. The teams also wrote code to copy the voltage behavior of the SLC device output they observed while testing. Finally, the teams found and confirmed that the SLC sent messages in 17-bit intervals, which corresponded to the FlashScan patent [3] describing the 17-bit address word and command word combination that makes up SLC messages.

III. AES Encryption

A. Background

Encryption is used to conceal information; it hides information in plain sight. A key, which is a group of values, is used to transform plaintext (the text humans can understand)

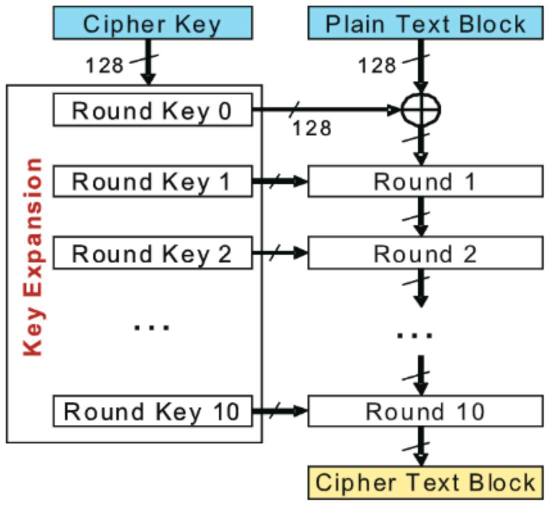


Fig. 1. AES Key Expansion to encrypt plaintext
[6]

into ciphertext. Ciphertext looks random on purpose. It cannot be understood unless the decryption key and the encryption method is known. There are many different encryption algorithms, but the one that the MSP430 microcontroller uses is the widely accepted Advanced Encryption Standard (AES) also known as Rijndael [4].

There are different AES modes depending on how blocks are manipulated. Blocks are fixed sized segments of ciphertext or plaintext. There are different ways these blocks can be manipulated during the encryption and decryption process: chaining, flipping, extending, boolean operations, etc... Hence, the many AES variations: Electronic Codebook, Cipher Block Chaining, Cipher Feedback mode, Output Feedback Mode, and Counter Mode [5]. The algorithm found in the RF Gateway Binary used the Electronic Codebook version with a 128-bit key. 128 bits is the shortest conventional key length that can be used for AES. The other options are 192 and 256 bits. Reasoning for using the shortest key may have been to save on power consumption and speed to optimize the microprocessor's performance.

B. AES Encryption and Decryption Function

Without getting into too much detail, notice that the 128-bit key is expanded to the size of the plaintext block, so that each part of the plaintext is encrypted with a different round key. This highlights the security of the algorithm, but is also noteworthy because this was a key feature in deciphering that the function being reversed is AES.

C. AES Key

Fortunately, AES encryption uses the same key to encrypt and decrypt data, so finding the encryption key is the same as finding the decryption key. The AES function can encrypt or decrypt data depending on whether the 3rd argument is set or cleared respectively. The second argument is the AES key, which was pulled from non volatile memory.

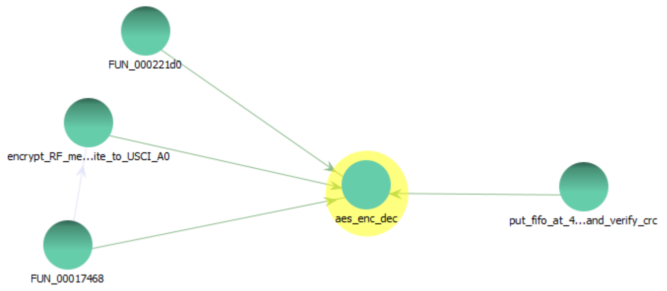

FUN_000221d0
encrypt_RF_me...ite_to_USCI_A0
FUN_00017468
aes_enc_dec
put_fifo_at_4...and_verify_crc

Fig. 2. Ghidra Call Graph: Incoming Function calls to AES encrypt decrypt

AES Key: "b8 dc 59 86 19 79 9d 79 41 de 21 97 1b b9 54 69"

To further verify that this is the encryption key, this key was searched for and found in a similar context in the pull station binary. It makes sense that key was also found in the other device's firmware because they need the key to understand any encrypted messages sent from the gateway.

D. Encryption and Decryption in Context

More is being discovered about the context under which encryption or decryption happens. Referring to the call graph in figure 2, the three functions on the left call the AES function to encrypt data, while the function on the right calls AES to decrypt data.

Currently is seems that encryption is called based on a loop in FUN_0000bd06 that checks that the transceiver's FIFO queue is not empty, implying that packets relating to an RF message are what is being encrypted. The team is still actively searching for what messages are encrypted and which are left unencrypted.

E. Debugging Results

At the beginning of the gateway's normal execution FUN_00221d0 encrypts data first each time. The message is 16 bytes long: "17 04 10 19 01 02 7f 01 08 0d 74 0d 00 00 96 35". When the debugger is fixed and the stack trace can be analyzed, it will be crucial to figure out when this specific message is decrypted and parsed out for use. Although this endpoint was fruitless, the debugging results do confirm that the AES key is copied to a location in ram and is manipulated in place to become a round key the same length as the data being encrypted or decrypted.

IV. Flash Memory

The flash peripheral is pervasive in its influence on the MSP430's operating capabilities. It is used to write variables to memory and efficiently erase parts of memory at different granularities. Flash memory is organized into banks and subsections. Depending on what value is written to the flash control register (FCTL), segments, a single memory bank, or all memory banks can be erased.

Because the MSP430 has interrupts that can influence control flow, safety measures are in place such that the flash

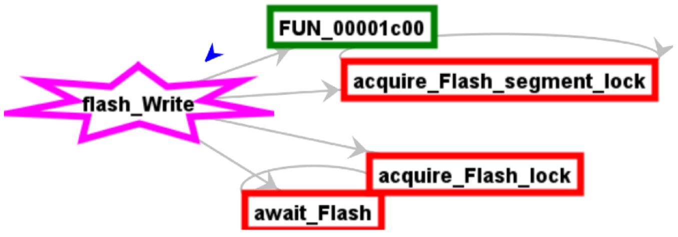


Fig. 3. Ghidra Call Graph: Outgoing methods for flash_Write

memory is treated as a shared resource and locked so that only one flash operation can happen at a time. Hence, why the await_Flash function or similar code is identified wherever the FCTL register is used.

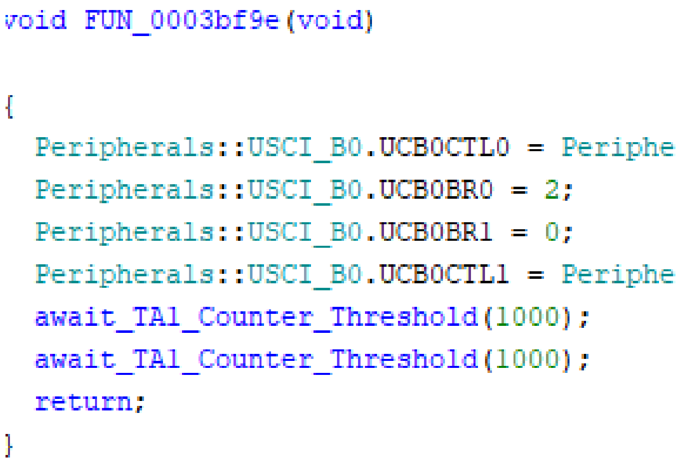
There are four different functions whose sole function is to write to memory using the flash peripheral. Not all of them use the await_Flash function, which polls the third flash control register; however, if they don't call this function they still use the same polling logic inline. Functions flash_Write and flash_Write2 are indirectly called by FUN_0000d506. FUN_0000d506 is a function that is a directly called by firmware's entry point. One possible reason these functions are used could be to optimize startup time. Both operate on smaller regions of memory and flash_Write has the potential to use a "Smart Write" feature. This feature speeds up writing to flash such that "program time is shortened" [7] at the potential expense of accuracy, which must be checked separately.

The other flash write functions, flash_Write3 and flash_Write4, are called much more frequently by a variety of different functions. The most important incoming function at this point is Config_Transceiver. In Config_Transceiver, the dominating code block that influences what areas of memory are erased and then written to prints "COMMAND_3R" to USCI_A0. From this code block one of four code blocks, whom each call flash_Erase and flash_Write3 and or flash_Write4, is traversed based on what could be an argument to "COMMAND_3R".

Current and past research has not focused on the fire alarm control panel; however, that location or a external debugger a developer attached to the device could be the source of these mysterious commands. The device can accept these commands when it is in Factory Mode. Factory Mode has not been documented, but it is a string found nearby where commands can be input via USCI_A0 (see function read_str_from_USCI_A0).

V. Timer Peripheral

This device has two timer modules whose purpose is to generate interrupts. In this case, the module configured is Timer A3, also known as Timer 1 to the device. The timer has a control register that sets its mode. The mode it is referencing tells whether a counter register is incremented or decremented with a clock relative to a maximum or minimum value set in a capture control register. The counter register changes based on clock cycles; the rate at which this happens can be changed by


```
void FUN_0003bf9e(void)

{
  Peripherals::USCI_B0.UCB0CTL0 = Periphe
  Peripherals::USCI_B0.UCB0BR0 = 2;
  Peripherals::USCI_B0.UCB0BR1 = 0;
  Peripherals::USCI_B0.UCB0CTL1 = Periphe
  await_TA1_Counter_Threshold(1000);
  await_TA1_Counter_Threshold(1000);
  return;
}
```

Fig. 4. Decompiled view of FUN_0003bf9e

altering bits in the timer's control register to divide the clock by some power of two.

There are two functions aptly named await_TA1_Counter_Threshold that take a maximum value as an argument and poll the capture control register until an interrupt flag is set. After recognizing that the interrupt has been asserted the function returns. This essentially means that these functions behave similarly to how the sleep system call would behave in various high-level languages.

They are normally called after various peripherals ports have be modified like in FUN_0003bf9e in figure 4 where USCI_B0 is being altered. This makes sense because it takes time to write those values so a certain time buffer is necessary to prevent errors even at the expense of slowing down the operating speed.

VI. Watchdog Timer

This module restarts the system if a "software problem" occurs or it can be used to generate interrupts at regular time intervals. It can also be temporarily disabled to conserve power. In this context, the Watchdog controller can only be changed to one of seven different values. One value stops the timer, the next four change the rate intervals interrupts happen at, and the last two values take the device out of active mode.

Given a clock source operating at 32.768 kHz, the timer intervals represented will generate interrupts every 1.95ms, 1s, 16s, or 4m:16s.

VII. DMA

Direct memory access is a way to move memory around without using the CPU. Because it doesn't need the CPU, less power is used and the CPU can spend its time accomplishing other tasks like manipulating the other peripherals. The MSP430 has eight different transfer channels or different paths along which this memory movement can happen [7].

There are two separate functions that setup DMA0 and DMA1 channels. The DMA is setup the exact same way for both of them; however, the only difference are some memory locations where the data is moving from one spot in RAM to another spot in RAM. Both functions setup the DMA such that 200 bytes are moved; however, the 200 byte arrays overlap in RAM implying that these functions are called at different times and under different circumstances. There is a path from the entry point to both functions. Future efforts in dynamic analysis would be useful to see if either or both of these functions are used, when they are used, and what data they are moving.

VIII. Configuring Port Mapping

While doing general analysis of the firmware, the team was able to identify a function responsible for configuring the port mapping for some of the ports on the gateway. Called near the firmware's entry point the function configure_port_mapping is responsible for setting up Port 1,2,3 and 4. The port pin is switched from a general purpose I/O to the selected peripheral/secondary function by setting the corresponding PxSEL.y bit to a 1 from a 0. If the input or the output function of the port is used it is typically determined by the setting the PxDIR.y bit. If PxDIR.y = 0, the pin is an input, if PxDIR.y = 1, the pin is an output. The function then makes three outputs to port 4 spaced apart by three timer calls. This is likely a test/setup output for when the device is first starting to run.

IX. Control Flow Fix

Ghidra had difficulty analyzing non-returning function leading to decompilation issues. This resulted in the binary being sparse in terms of defined functions. Statistics were gathered by running Tarjan's strongly connected components algorithm on the binary before and after the fix. The original binary had 1,652 functions, 9,548 code blocks, and 7,496 strongly connected components where the largest component had 81 code blocks in it. The new binary has 2,305 functions, 14,804 code blocks, and 10,813 strongly connected components where the largest component has 202 code blocks in it. The difference can be visually seen in figure 5.

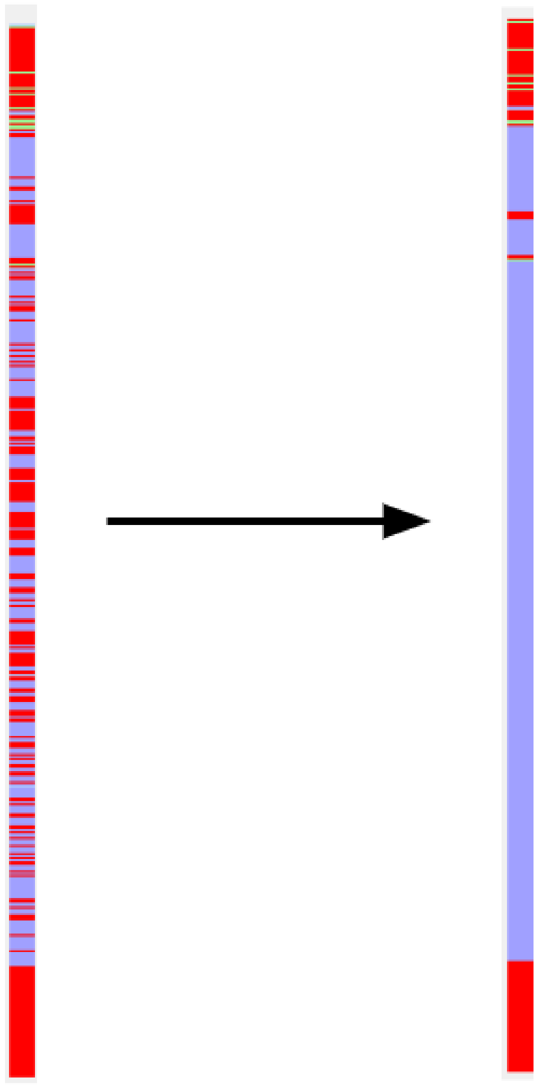


Fig. 5. Ghidra Defined Functions Heatmap Before and After Analysis Change

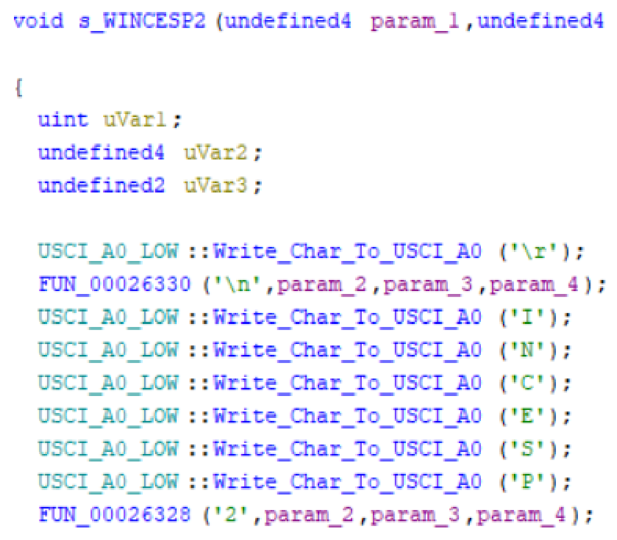
A. Hidden Strings

Fixing the control flow analysis revealed almost 1,000 more functions, many of which are obfuscated calls to Write_Char_to_USCI_A0. Tracing the program and concatenating the characters reveals strings. For example, observe the string ”WINCESP2” in figure 6. Windows CE is a Windows operating system meant for embedded devices. SP2 means service pack two, which is usually how Windows labels major updates to their OS's. The MSP430 definitely does not have a Windows OS, but it may indicate the presence of an external tool connected to the device during the development of it's firmware. Future work will be done to log the strings being printed as the gateway is running.

X. RF Backdoor Attack

A. Background

One attack that the team has been working towards is implementing a backdoor attack into the gateway. A backdoor would allow malicious individuals direct access the gateway while bypassing its authentication methods. Past work has been dedicated to investigating the RF chips message reception mechanism. Having a full understanding of how RF message reception worked on the gateway would be a significant goal for the team, as it would allow them to edit the range of messages the gateway can parse. Currently the team understands that the chip primarily uses the Do_stuff_and_wait_payload_ready function for message reception. Its job is to constantly loop while checking to see if a payload has been received and is ready to be read. Once the payload is ready to be read, The do_stuff_and_wait_payload_ready calls the function put_fifo_at_461d_and_verify_crc to store bytes about the message at 0x461d and 0x471c and also verify the calculating the payloads CRC and comparing it to the expected CRC value. If the CRC is valid the payload-waiting loop is broken, if not, the



```
void s_WINCESP2(undefined4 param_1,undefined4

{
  uint uVar1;
  undefined4 uVar2;
  undefined2 uVar3;

  USCI_A0_LOW::Write_Char_To_USCI_A0('\r');
  FUN_00026330('\n',param_2,param_3,param_4);
  USCI_A0_LOW::Write_Char_To_USCI_A0('I');
  USCI_A0_LOW::Write_Char_To_USCI_A0('N');
  USCI_A0_LOW::Write_Char_To_USCI_A0('C');
  USCI_A0_LOW::Write_Char_To_USCI_A0('E');
  USCI_A0_LOW::Write_Char_To_USCI_A0('S');
  USCI_A0_LOW::Write_Char_To_USCI_A0('P');
  FUN_00026328('2',param_2,param_3,param_4);
```

Fig. 6. Function with obfuscated calls that prints Win CE SP2

function attempts to improve its signal strength by restarting the transceiver.

While the team has a good understanding of the payload reception mechanism of the RF firmware, there is little understanding of how the gateway retrieves the message type and commands from received package for further use. Currently we understand that the functions that that parse the payloads are likely highly ingrained with the functions that have already been analyzed; However, most attempts to reverse engineer them have been unsuccessful [8].

B. RF Payload Parsing

Upon investigation, there doesn't seem to be a function called within the Do_stuff_and_wait payload_ready function that parses the messages, so there is a high chance that the message parsing function may be integrated elsewhere. Currently the team has identified a function that could potentially be related to the parsing of the messages. The function interesting_2nd_payload_reception_func seems to be similar to the do_stuff_and_wait_payload_ready function in that it seems to receive a message from the USCI_B0 and uses the writes_data_out_SPIB0_to_RF_Chip_at_Reg function to interact with the transceiver registers. The function has three features that one would expect to see in a function that interacts with a packet. It first, calls the Read_from_transceiver_via_UCB0 to retrieve the packet, then it checks the packet length to ensure that it is not to large, and finally it seems to store each byte of the packet to the RX_Packet_Arr Array. There a couple of differences between the 2nd_payload_reception_func and do_stuff_and_wait_payload_ready. One such difference being that there is no verify CRC check in this function. This could be a significant vulnerability as the data received by this function gets stored in the exact same location as the packets received in Do_stuff_and_wait_payload_ready.

There is a possibility that this function could be used find a function that is responsible for parsing packets in the firmware. Since this function as well as do_stuff_and_wait_payload_ready store the received RX packets at the same exact location. If we can find a connection between this function and Do_Stuff_and_wait then we may be able to identify where the payloads received by these two functions get parsed. The interesting_2nd_payload_reception_func function should definitely be heavily looked at in the future.

C. Data Integrity Checks

It is also of the utmost importance to ensure data integrity by searching for any possible security issues or vulnerabilities that might leave a device susceptible to attackers. One such time this might occur is in the bootloader, or startup, mode. The purpose of this mode is to initialize the hardware and load the main firmware into the device's RAM. Given that the bootloader mode is executed first, it represents a prime target for attackers looking to manipulate the early onset of the firmware loading process, which could then cascade into detrimental effects on the entire system's functionality and security.

Due to the fact that our investigation primarily targets vulnerabilities within the device, we carefully scrutinized the entry point of the bootloader located in the address range 0x00001000-0x000017ff in the RAM. A significant finding from our investigation was the absence of data integrity checks in this mode. Data integrity checks, such as checksums, are especially vital to verify that the firmware being loaded has not been tampered with, and that the firmware remains unaltered from its intended state. Checksums are values derived from a set of a data. To calculate this value, an algorithm is chosen that fits the needs and scope of the data being verified. Essentially, this value helps in detecting whether the data has been corrupted or modified either intentionally or even accidentally. Some checksum functions implemented in previous firmware versions include checksum, checksum_calc_then_dma, checksum_result, and checksum_related?. Despite the recognized importance of such functions, our current team goals do not align with developing or enhancing checksum capabilities. Instead, our focus is on identifying and exploiting these found vulnerabilities within the firmware. In turn, this will help us better understand any potential security weaknesses that need addressing in future developments.

D. Cyclic Redundancy Checks

Although these checksum functions may not exist in this version any longer, there are Cyclic Redundancy Checks (CRC) periodically scattered amongst the code. For instance, a function entitled Perform_CRC is performed twice when verifying regions of memory in the firmware. According to the user guide, a CRC check will provide a signature for any given data sequence. There are four registers at play in the operation of these checks: CRC Data In (CRCDI), CRC

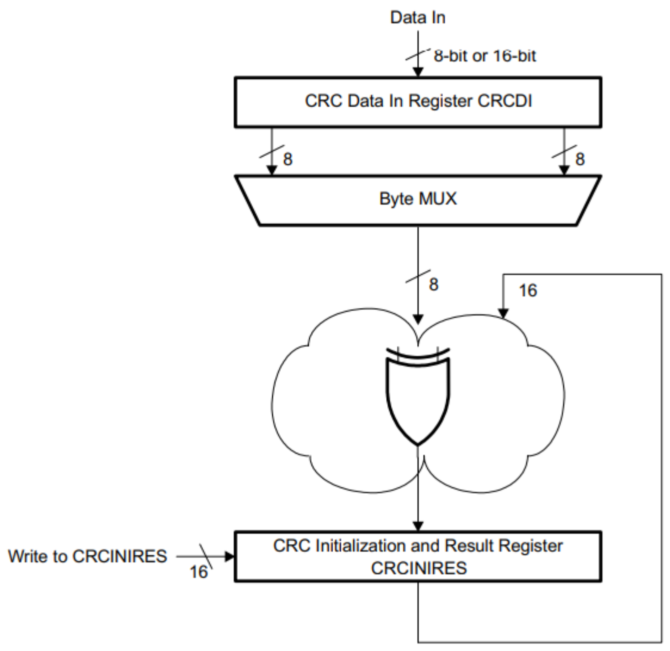

Data In
8-bit or 16-bit
CRC Data In Register CRCDI
8
8
Byte MUX
8
16
Write to CRCINIRES
16
CRC Initialization and Result Register
CRCINIRES

Fig. 7. The active registers when performing a CRC check

Data In Reverse Byte (CRCDIRB), CRC Initialization and Result (CRCINIRES), and CRC Result Reverse (CRCRESR). As seen in Figure 7, the CRC generator is first initialized by writing a 16-bit word to the CRCINIRES register, and any data that should be including in the calculation must be written to the CRCDI or CRCDIRB registers in the same order that the original CRC signature was calculated. The guide states, "The actual signature can be read from the CRCINIRES register to compare the computed checksum with the expected checksum." The checksum is stored in memory and used to check the correctness of the result found from the CRC calculation.

XI. Entry Point

A. Background

In this section, we focus on the examination of the firmware entry points in the device, which are particularly critical during its startup or bootloader mode. Our investigation reveals not only the functions associated with the standard initial execution, but also an intriguing aspect of the firmware's operational logic regarding procedures for both successful and unsuccessful verification of data.

B. Firmware Verification

Our analysis began by identifying the actual entry point of the firmware, which is critical as it is the first code executed upon boot and orchestrates the initial setup and environment configuration for the device. The function labeled as entry_point in the Ghidra decompiler is a significant focus since it sets the initial execution context and begins the firmware operations. This function effectively sets up the environment by configuring the watchdog timer, clearing portions of RAM,

```
if (((short)uVar22 == 0) && (DAT_00001c7c != 0)) {
  *(undefined4 *)(pbVar4 + -4) = 0xd708;
  uVar14 = Verify_RF_Firmware();
  if ((char)uVar14 == '\0') {
    looks_like_entry_point();
    return;
  }
}
```

Fig. 8. FUN_0000d506 makes a call to looks_like_entry_point

and performing initial firmware integrity checks. It then calls FUN_0000d506, which can be considered an extension of the entry point functionality due to its integral role in setting up the device's operational parameters, including peripheral setups and preliminary security checks.

FUN_0000d506 plays an interesting role when it comes to initializing the device. Within FUN_0000d506, there is a call to Verify_RF_Firmware(), which is essentially a function devoted to verifying different regions of memory with various calls to Verify_Memory, Call_Verify_Memory_#1, Call_Verify_Memory_#2, and Call_Verify_Memory_#3. Once Verify_RF_Firmware calls these functions and completes the memory verification process, it checks the result of the verification against certain expected values. If the value returned is equal to 0, the secondary entry point FUN_0000d506 makes a call to looks_like_entry_point and returns, indicating that we will continue the initialization process inside that function. This procedure can be seen in Figure 8.

Our analysis initially led us to believe that looks_like_entry point functioned as a recovery mechanism when firmware verification failed. However, upon closer inspection, we discovered that looks_like_entry_point is actually invoked when the verification of the RF firmware succeeds, not fails.

The discovery that looks_like_entry_point is actually linked to successful verification as opposed to failure as we previously thought introduces an entirely different perspective on the device's security measures. For one, we realized that when the RF firmware is verified successfully, looks_like_entry_point is called in order to proceed with the regular operational set-up that is typically found after a successful data integrity check. This will likely prepare the system to transition from a secure boot to normal operational mode. This discovery also highlights the device's reliance on successful start-up procedures to ensure security and integrity. In the case where RF firmware verification fails, the code would carry on with the rest of the FUN_0000d506 function, which performs different operations in the case of a failure as opposed to the operations carried out in looks_like_entry_point following a success. This functionality (i.e. comprising multiple different functions for different verification scenarios) can be viewed as a robust design choice, promoting resilience by ensuring that only verified and intact firmware can enact the device's full capabilities.

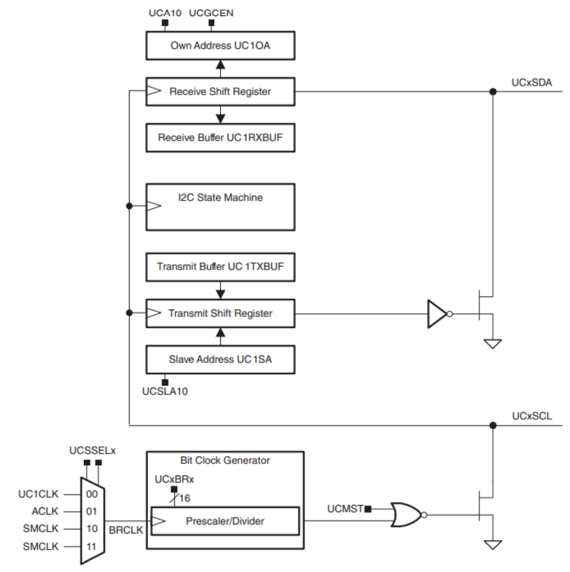


Fig. 9. I^2C mode - USCI Operation

XII. Data Transmission

A. Background

Data transmission is a crucial step for the system to operate; it allows the fire alarm system to control its various components, including detectors, control panels, and monitoring software. Understanding how the fire alarm system handles the data transmission is a crucial step in reverse engineering this system.

B. Peripherals

Our analysis began by checking which function would be in-charge of the peripherals for this device. A function called FUN_0000ee08 stood out as it began by initializing and configuring peripherals, such as ports and control registers (Peripherals::PORT_3_4, Peripherals::USCI_B0). The universal serial communication interface (USCI) handles multiple serial communication modes, this function in specific utilizes USCI_B which supports I^2C mode. I^2C mode allows for 7-bit and 10-bit device addressing modes, START/RESTART/STOP, multi-master transmitter/receiver mode, and slave receiver/-transmitter mode. External components attached to the I^2C bus serially transmit and/or receive serial data to/from the USCI module through the 2-wire I^2C interface, as shown in 10. So the analysis of this function is a big step toward understanding how information is transmitted within this system.

C. Data Transfer

The main function, FUN_0000ee08, utilizes five other functions to perform the data transmission. The five functions it utilizes can be broken down to what they are responsible for: FUN_0000f09c controls the peripherals based on certain inputs, FUN_0000f95a sends default values to FUN_0000f09c, FUN_0000e6b8 prepares and sends data to the USCI_B0 peripheral, FUN_0000f6c6 waits for the transmission to be

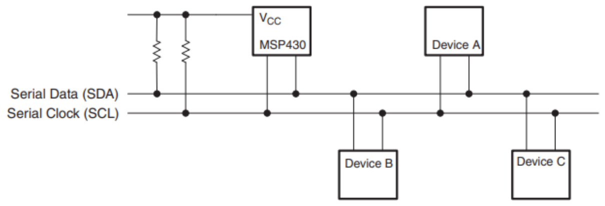


Fig. 10. I^2C Bus Connection Diagram

completed, and finally FUN_0000e814 actually performs the data transmission. This function provides valuable information as to how the fire alarm system communicates and transmits data to its peripherals, which by fully understanding this behavior we would be able to take advantage of any vulnerabilities presented by this data transmission.

XIII. Surrogate Device

A. Background

Another attack that the team has been working on is using an Arduino and other additional hardware to send 'spoofed' packets to the gateway. This would theoretically allow an attacker to remotely compromise the gateway via the remote bootloader attack previously discovered [9]. Currently the team understands the packet structure and what signals can be sent to trigger the alarm. The team has also created a design for the surrogate device, but has yet to build it.

XIV. Conclusion

A. Next Steps

Now that the peripherals are identified within key functions more data needs to be collected dynamically. There are many future directions research could go: tracing decrypted data messages, logging hidden debug strings, tracing data transferred with DMA. The main priority next semester will be to write scripts to collect the necessary information and synthesize that with current results to be able to craft a message that manipulates the gateway.

The next steps for signal spoofing would be to build and test the surrogate device designed in the previous paper, as well as modify and add onto the device so that it can read and respond to SLC messages sent by the gateway. This could provide further insight into reverse engineering the remainder of the device code, as it would provide both this team and further teams with potential insights as to how the device processed and sent out SLC messages, as well as with various insights into the device's source code.

One possible step this team can take in terms of the device's firmware verification is investigating the device's hardware and software capabilities to support real-time monitoring of firmware integrity. This can be critical to the security or insecurity of a device, as it could allow continuous, automatic checks that the firmware has been tampered with, and not just one check at the entry point. This investigation could include a deep dive into the existing architecture of the devices to identify and manipulate any already built-in security features, such as the CRC module and any other defenses.

The next step for analyzing data transmission would be to delve deeper into the peripheral's architecture and explore the physical hardware. By analyzing the peripheral and port used by the data transmission function the team would possibly be able to find new exploits that could bypass current limitations. The physical hardware that this port connects to possibly holds more information that can be extracted and better understand the transmission function, potentially allowing for the team to achieve a breakthrough.

When it comes to identifying the packet parsing methods for the RF gateway, there are a couple of directions that be pursued in the future. A better understanding of the functions that reference interesting_2nd_payload_reception_func and Do_stuff_and_wait_payload_ready would give us a far better understanding of the context in which both of these functions are called. Since they both seem to perform the same job, It would be valuable to know when one is used instead of the other. Another area of interest would be a dynamic analysis of what the firmware does when the gateway receives a message. This would likely reveal a significant amount of information about what the firmware does with the packet data it receives.

References

[1] Detectors. [Online]. Available: https://www.securityandfire.honeywell.com/notifier/en-us/browseallcategories/wireless/swift/detectors

[2] Texas Instruments. SWIFT™ Smart Wireless Integrated Fire Technology Manual. [Online]. Available: https://prod-edam.honeywell.com/content/dam/honeywell-edam/hbt/en-us/documents/manuals-and-guides/user-manuals/LS10036-000FL.pdf?download=false

[3] E. Bystrak and A. Berezowski, "Enhanced group addressing system," U.S. Patent 5 539 389, Jul. 23, 1996.

[4] Wikipedia contributors, "Advanced encryption standard — Wikipedia, the free encyclopedia," 2024, [Online; accessed 2-February-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=1209329215

[5] M. Dworkin. (2001) Recommendation for block cipher modes of operation. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[6] T. Uli Kretzschmar. Aes128 - a c implementation for encryption and decryption. [Online]. Available: https://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-files/156/slaa397a.pdf

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

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

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