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

Aniyah Bussey
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, United States
abussey6@gatech.edu

Nathan Dailey
School of Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, United States
ndailey6@gatech.edu

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

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

***Abstract*—Given the increase of large commercial buildings throughout America, building management has become a bigger concern than ever, as it becomes less and less possible for one person to manage a building thoroughly . One crucial system in these very large buildings is the fire alarm system. These systems are traditionally wired, however newer developments are migrating to wireless infrastructure to support expansion of these systems into larger and more complex places. However, moving to a wireless infrastructure means that these fire alarm systems are now being opened up to new methods of attack by malicious entities. With enough experience an individual could wirelessly gain control of a building's fire alarm, and remotely control the system potentially putting innocent people into dangerous situations. This ongoing study analyzes the vulnerabilities of Honeywell's Smart Wireless Integrated Fire Technology (SWIFT) system, which integrates wired and wireless communication by using a gateway communicating with smoke detectors, pull stations, and other addressable fire devices.**

I. System Introduction

The FACP, or Fire Alarm Control Panel, is a wired component of any fire alarm system that can receive information from all wired devices in the system.

The SWIFT system is a commercial wireless fire detection system that uses a robust mesh network created by the SWIFT Gateway to integrate wireless devices into an existing wired system [1]. The wireless gateway is the SWIFT device that bridges the gap between wired and wireless devices. Thus, it is the main target for this project, as it controls the mesh network, which consists of wireless devices, by managing its formation and configuration while also interfacing the wireless mesh network with the wired network. In traditional systems, all components communicate over a wire which connects the devices. This wire is known as the Signaling Line Circuit, or the SLC. The FACP the team analyzes is made by Honeywell and communication is provided to the control panel by its SWIFT devices.

The gateway has two processors which communicate with each other through a universal asynchronous receiver-transmitter, or UART channel; these are the SLC and RF (radio frequency) processors, whose main functions are to interface with the wired devices, and handle wireless communication and initialize the bootup process, respectively. The gateway has three firmware update files; they contain the bootloader firmware, the RF firmware, and the SLC firmware [2].

The SLC chip is one of the primary components of the gateway that the team is analyzing, and is outlined in blue in figure 1. This processor's main responsibility is interfacing with the physical SLC line, whose primary purposes are to carry signals between components and provide power for the addressable device modules in the fire alarm system. The signals sent on the SLC line both report device status and provide feedback to the periodic polls of the FACP.

The RF chip is outlined in orange as seen in figure 1, and it is responsible for managing the mesh network that contains the wireless devices, relaying wireless device information to the SLC chip for communication with the FACP, and updating both itself and the SLC processor when the gateway's firmware is updated.

II. Previous Research

The previous teams figured out the layout of bytes in the packets for the OTA (Over-the-Air) protocol; however, how the payload was specifically parsed remained undiscovered. [3] The team partially uncovered how messages were received and stored; they found the loop that waits for payloads and also analyzed how the CRC verification failure could undermine that process.

III. RF Backdoor Attack

A. Patching Preparations

The team uses Ghidra and other software to reverse engineer the RF firmware files to work towards the primary goal of identifying methods of gaining control of the system. Previous teams tried to write exploits in assembly. This

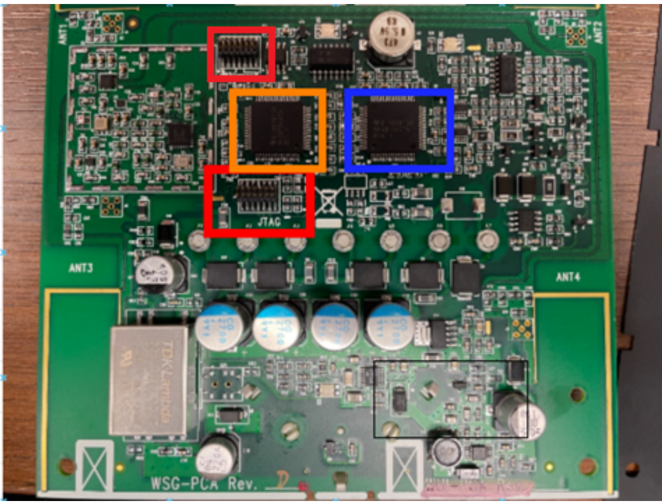


Fig. 1. The wireless gateway board, with various components outlined, such as the RF (orange) and SLC (blue) processors, as well as the two JTAG headers (red).

semester the team tested whether or not that was necessary. The relevant binaries were written for a msp430 processor, with the specific device being f5437a. After compiling a simple program written in C, opening the binary in Ghidra, and verifying the instructions the team concludes that whatever exploit code that is needed to be written in the future can now be written in a high-level language.

```
#include <msp430.h>
// #include <msp430f5437a.h>

int main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog timer
  P1DIR |= 0x01;                            // Set P1.0 to output direction

  while (1)                                 // Test P1.4
  {
      P1OUT ^= 0x01;                        // Toggle P1.0 using exclusive-OR
      __delay_cycles(100000);
  }
}
```

```
msp430-elf-gcc -I /root/ti/msp430-gcc/include -L
/root/ti/msp430-gcc/include -mmcu=msp430f5437a -O2
-g msp430x54xA_1.c -o msp430x54xA_1.o
```

Fig. 2. Command to compile and example program

B. RF Message Function Analysis

The previous teams were able to obtain source code for SWIFT Tools, the compatibility software that users can use to monitor the gateway and other aspects of the wireless fire alarm system. In this C# source code several enums were defined. The most obvious and easily distinguishable enum to find in Ghidra was NodeType.cs, which defined values with each one representing a specific component of the fire alarm with an independent purpose and function. Initially, by using the Ghidra search feature the team was able to find a small function that has a chain of comparisons to six values. Each of these values are located in the NodeType enum. FUN_000382d4 compares its second argument to the values that we assume are from the enum: RelayModule, GatewayN, Pull, AVBase, SyncModule, DisplayDriver. While knowing what this function does is not immediately relevant, the fact that it is comparing these values to a specific argument is crucial. The team traced this parameter back to glean some insight on where the value is being derived from. The parameter is a pointer to byte, given that the instructions that act on are all at a byte level granularity, and this pointer points to the memory location DAT_00003ccc. At this point the team established a direct connection to the builds_RF_message function, which read this memory location four different times. Additionally FUN_00021236 reads this location several more times. By simply defining the enums the team gained some insight into how the builds_RF_message function uses those values to modify its control flow and the main focus of FUN_00021236. Relative to the other functions in the binary, it is medium sized having 120 different vertices in its graph. Initial impressions of the graph shape and knowing that it reads DAT_00003ccc_NodeType indicates that the purpose of this function deals with a subset of the known enums.

```
namespace Honeywell.WirelessTool.WirelessInterfaces
{
    public enum NodeType
    {
        InvalidDevice = 0,
        Gateway = 1,
        Detector = 2,
        LCD_UI_Prev = 3,
        SounderStrobe = 4,
        MCP = 5,
        Strobe = 6,
        EsserDetector = 7,
        HeatDetector = 0x10,
        SiteSurvey = 20,
        PhotoDetector = 36,
        PhotoHeatDetector = 66,
        MonitorModule = 67,
        Pull = 123,
        SyncModule = 72,
        AVBase = 71,
        RelayModule = 69,
        GatewayN = 70,
        AcclimateDetector = 76,
        USBAdapter = 99,
        DisplayDriver = 101,
        Output = 102,
        Sounder = 103,
        Orphan = 104,
        NewDeviceRetrievedButNotPrecommissioned = 105,
        Unknown = 106,
        Repeater = 200
    }
}
```

Fig. 3. Node Type values

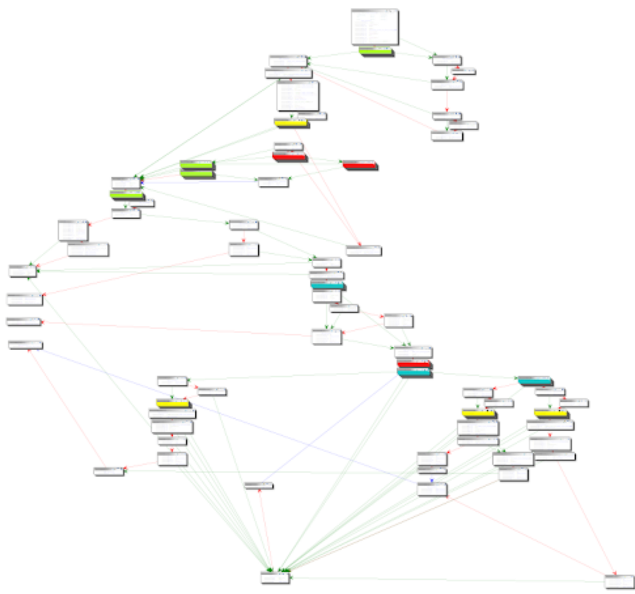


Fig. 4. FUN_00021236

One comparison block has the following node types: Photo Detector, Photo Heat Detector, Acclimate Detector, and Heat Detector. These nodes are referring to the various detection features that any fire alarm may or may not have. Photo refers to the photoelectric sensors used to detect smoke and fire, heat detector refers to the ionization feature that thermal alarms usually use to detect fires, and acclimate detector is a mix between photoelectric and thermal alarms. In the figure graph vertices are grouped and colored to visualize the clusters of node types. Green represents component type nodes: monitor, relay, “gatewayN”, and pull. Red represents the heat detector type nodes mentioned previously. Yellow represents a change in control flow depending upon whether the node type is LCD_UI or Gateway. Finally, blue represents checking a memory location to see whether it refers to SyncModule or AVBase node types. The flow of this function broadly follows the following logic: check what components you are dealing with, verify that you are dealing with either the LCD_UI or the gateway, check what fire detections are available to use, communicate with the sync module and AV Bases, then whatever information that was presumably synced between the bases informs the LCD_UI and gateway.

Knowing which functions deal with which enums and which enums are together during comparisons allows us to discover which Node types are closely related and how groups of them interact with other groups.

IV. Surrogate Device

A. Background

A goal that the team prioritized is spoofing a fire alarm on the SLC network. One specific approach that was taken was creating a surrogate device that can send spoofed SLC messages. At the beginning of the semester, the team decided to use the following components to make the surrogate, but since then the components have been updated, all of which which will be discussed later in this paper: raspberry pi, 2 5V relay boards, a breadboard, and wires. In order to achieve this goal, the team previously decided to insert the surrogate device into the SLC line, which is daisy chained through all of the devices, and have the surrogate replicate a an existing device on the SLC line that would be replaced by the surrogate. We have since discovered that the surrogate device must still be added to the SLC line, but instead of replace an existing device, it must ”appear” to be part of the system and replicate the appropriate address/SLC messages so that the system doesn’t enter a ”System Trouble” state and just not recognize the surrogate device. One thing that has remained the same between the beginning and ending of the semester is how the surrogate device will have to insert the spoofed messages we make in order to spoof the fire alarm. This semester the team focused on confirming the various components that will make up the surrogate device and crafting the spoofed SLC message.

B. Important functions

1) add_device_??: One function that the team looked at was `add_device_??`, as this function seems to play an integral role in determining what devices are recognized by the control panel. This would definitely aid in the surrogate device efforts since we must integrate the surrogate into the existing system. We came to this conclusion because there are places in the code where the logic indicates that this function specifically is involved in the group polling process. For example, as seen in ***, this snippet of the function is responsible for counting the number of devices in the system. Next semester, the team will continue to work towards further analyzing and configuring the logic in this function so that the surrogate device can be recognized and counted. This semester, we decided to gain a better understanding of this function as a whole, so we did some reverse engineering work on the `add_device_??` incoming function calls to better understand the various variables used since they are used throughout.

2) decode_recieved_ipc_message: Another one of the functions that the team looked into was `decode_recieved_ipc_message` as this is an important function that will be used within the surrogate device. This function is quite impactful, as it calls many other functions, so while we could not decode the entire function, multiple of its sub-functions have been decoded. For `decode_recieved_ipc_message` what we know is that the input is a pointer in memory to the message location, since the input is a pointer and not the actual message, the conclusion has been made that all of the messages are stored into memory and as such there may be a way to influence what each message does by changing what the pre-stored messages are. After some initial data allocation the function then calls `Fun_00029354` which has yet to be decoded, but then this function calls `check_trbl_and_tamper` and

`change_trblcnt`. These functions both deal with error correction and data stability.

```
/* BINDIFF_COMMENT: *** 100.000000% match with 99.330715% confidence using function: hash mat
   ***
   BINDIFF_MATCHED_FN: *** @00028780 WSG_SLC_3_0.bin@00028780 ***

   This function appears to check for tamper and for trouble for a SLC device. This function
   only return false when the tamper count or the trouble count for a device is greater than

bool check_trbl_and_tamper(bool param_1)

{
  byte bVar1;
  byte bVar2;

  bVar1 = get_tamper_slc_addr();
  bVar2 = get_trbl_slc_addr();
```

Fig. 5. Header and description of the check_trbl_and_tamper function

```
/* BINDIFF_COMMENT: *** 100.000000% match with 99.172096% confidence using function: hash mat
   ***
   BINDIFF_MATCHED_FN: *** @0002926a WSG_SLC_3_0.bin@0002926a ***

   It would appear that this function looks at the trouble count (global variable) and either
   increments or decriments it based upon the input. Returns False only when the trblcnt for
   specific device is above 0x65, else returns true */

bool change_trblcnt(bool param_1)

{
  bool bVar1;
  byte bVar2;

  bVar2 = get_trbl_slc_addr();
  if (param_1 == false) {
    if (trblcnt != 0) {
      trblcnt = trblcnt - 1;
    }
  }
```

Fig. 6. Header and description of the change_trblcnt function

`check_trbl_and_tamper` looks at both the trouble and tamper memory addresses and returns false if something has been tampered with or if there is too high of a trouble count. Since this function checks to see if any tampering has happened within the SLC message, we believe that this function could play a big role in the viability of the surrogate device. If we can figure out what is held at the tamper memory address of the message that would allows us to circumvent the tamper check and hopefully have the surrogate device infiltrate the SLC undetected. `change_trblcnt` looks at the trouble memory address and increments the trouble count accordingly based upon the message that was recieved and executed. This method only returns false if the trouble count is greater than `0x65`. Both of these functions play a large role in `decode_recieved_ipc_message` and the are some of the lowest level functions that the code is built off of. By understading these it will help us gain a greater grasp of the function as a whole, and help in the reverse engineering process.

C. Physical Hardware

One of the two main challenges for the surrogate device was coming up with the physical design, as we needed something that was easily programmable, could interact with the high voltage SLC network and would have all the functionality of the wired pull station that it was trying to emulate. We quickly decided that the most efficient tool to use would be a micro-controller as they are very easy to program, and due to the teams familiarity with them. After deliberation we decided to go with using an Arduino for the ”brains” of the surrogate device. The Arduino was chosen over a raspberry pi due to the an Arduino not having an OS which means that all the code that we will run on the surrogate device will have to go through less levels of complexity than if it was on a the raspberry pi. The Arduino also offers the benefits of drawing less power and having dedicated pins that we can easily use to prototype. The team is also aware that a high signal on port P1.0 will trigger the message reception handler, however there is a compatibility issue between the voltages as the voltage of the SLC is 24v while the Arduino operates at 5v. Due to this incompatibility, we have concluded that the Arduino would need supplemental hardware, to both read and send messages to the SLC.

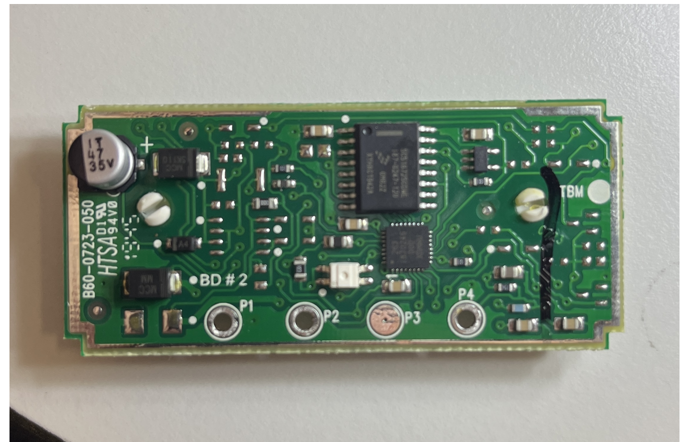


Fig. 7. Image of the PCB of a pull station

After we decided on the hardware we would use, the next task we were faced with was the implementation of the hardware. We were unsure of how the Arduino would need to interface with the SLC, and thus what kind of hardware would need to be designed for it. We concluded that it would be best to reverse engineer a design from the wired pull station that we had set up inside of the lab. When taking apart the pull station we discovered that the PCB had two ICs on board, which allowed us to realize that the pull station was a state machine, i.e. not just a simple switch-resistor device. By learning that the pull station has states that it can enter, the team created a state diagram for a device on the SLC network.

Discovering the possible states of a device was very beneficial because we were able to deduce that if a device was not set up inside the system, then it would be deactivated and not able to do anything. This meant that the surrogate device would have to replace an existing device in the network, as trying to act as a device that was currently not in the network would lead to an error in the system.

After visual analysis of the pull-station was done, it was reconstructed, and we used an oscilloscope to see how the messages were transmitted on the SLC. We discovered that the messages going into the pull station would fluctuate from

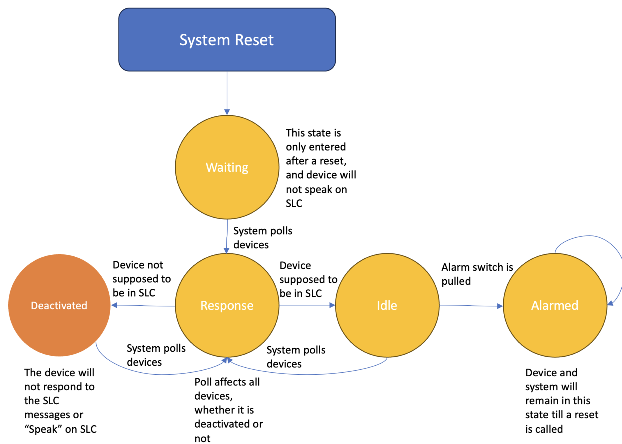


Fig. 8. Diagram showing the possible states of an SLC device

10v to 24v, solidifying the fact that the device would need a voltage dropper so the Arduino could properly read what was being sent. When the outbound messages were observed, they had a structure reminiscent of a sinusoidal wave. The output would appear to spike to opposite voltages at the same time (1v and -1v), this meant that the physical hardware would need to be able to recreate this activity.

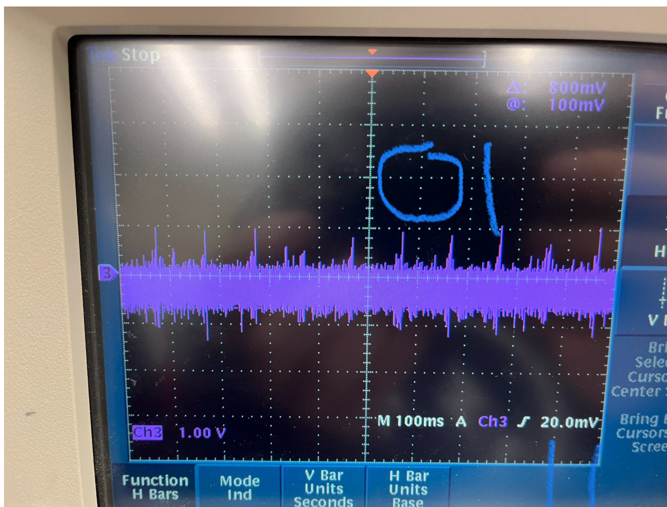


Fig. 9. The output signal of the pull-station, with spikes visible

To achieve this behavior a simple circuit was designed that uses a DAC to control voltage, giving us two outputs, one positive and one negative. Then two N-Enhancement MOSFETs were added to allow for a digital signal to control which signal, positive or negative, was being sent to the SLC. Once this output design was considered, a final schematic was able to constructed. Unfortunately due to time constraints and lack of materials, this design was never tested and will need to be implemented in the following semesters.

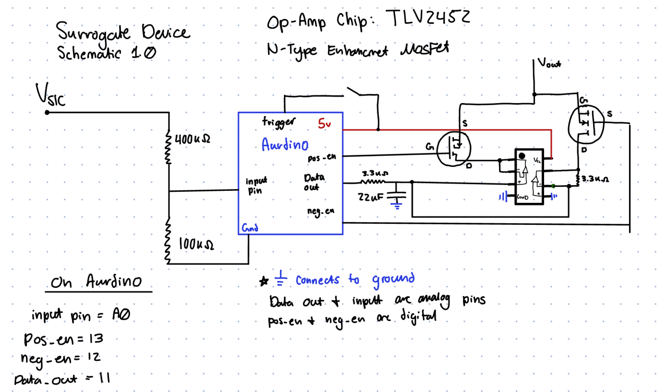


Fig. 10. Final Circuit design of the Surrogate Device

This circuit diagram allows of the surrogate device to read the inputs from the SLC (scaled to 1/5 voltage so that it will not break the Arduino) and then the output allows for a negative or positive voltage response. The voltage level is controlled by the data_out pin, the system acts as a DAC which converts a duty cycle into a constant voltage. This also allows for the voltage to be amped back up to 24 volts if needed, as we can connect the op amp in the circuit to a larger power supply.

D. SLC device output

When reverse engineering the wired pull-station, the output of the device on the SLC was observed and recorded. This allowed the team to understand how the SLC devices respond to the gateway and also how the gateway knows that a fire alarm has been signaled in a pull station. The base frequency of the negative end of the the SLC is the thick band of signals that lie between 0.2 and -0.8v. The idle frequency of the Pull station is 10 Hz with it spiking to 1V and -1V every 100 ms. The variations in the spike heights are due to the oscilloscope lacking the proper resolution to show the complete signal. The image below shows the triggered state of the pull station on the SLC, with the address of 001.

Once the pull station has been triggered the spikes maintain the same frequency but increase to 1.5 and -1.5V, and they maintain this until the system has been reset. This spike in voltage is what shows the system that the pull station has been pulled, as long as it is high, the system will be alerted. Once the system resets the SLC goes "dark", aka the pull station is no longer sending out pulses, until about 1 second after it goes dark where the pull station sends a bloop to the SLC. This is most likely the response to what devices are in the system that is sent out by the gateway. After the bloop the system waits for 1.5 seconds then starts spitting out the idle pull station frequency. This response is something the the future surrogate device team will need to focus on, as if the surrogate device cannot properly respond to it during a system reset then it will be shut of from the system. A portion of code was written to copy the behavior that was shown in this section, to be tested

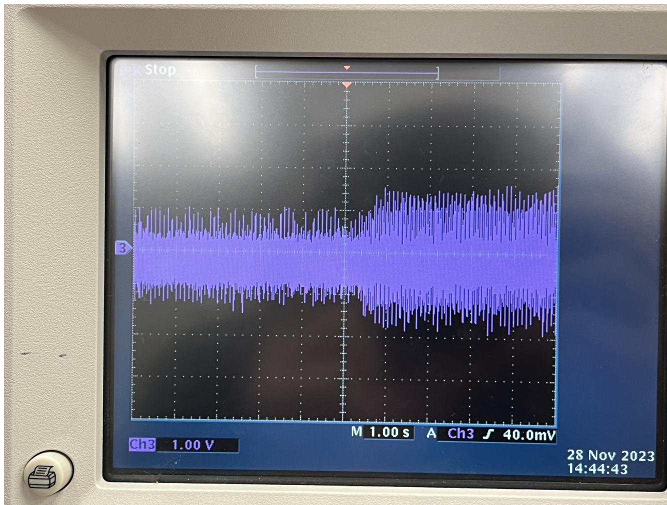


Fig. 11. Triggered state of SLC leaving the pull-station.

on the surrogate device when it has been prototyped, it can be seen in 12.

E. SLC Message Structure

In order to craft the spoofed SLC message(s) that would be put on the surrogate device, we had to gain a better understanding of the structure of the SLC regularly sent out by the pull station. To do so, an oscilloscope was used to take captures of the messages sent by the pull station during both idle and alarm states. As seen in 13, idle SLC messages sent came in 17-bit intervals. The SLC messages sent during the alarm state, meaning when the pull station was triggered, were also sent in 17-bit intervals like what can be seen in 14. Both are significant because this corresponds to the 17-bit address word and command word combination that is described in the FlashScan patent [4] and what can be seen in 15.

V. Conclusion

A. Next Steps

Continuing the team's efforts in creating an RF backdoor attack is an important goal for the future, and it will involve further analysis of the RF firmware. With payload storage and verification explored, the team will need to deduce what is done with the data after the aforementioned steps. Although efforts to reverse engineer them have not been successful, functions that perform payload parsing are likely tightly coupled with functions that have already been analyzed. Given the team is able to fully reverse-engineer the parsing logic, it will be possible to edit the firmware in preparation for the backdoor, and the W-USB or a software-defined radio will be able to trigger it.

The team will also continue to reverse engineer the functions in the binary files associated with the `slc_normalish_dump2.bin` file in to further understand the functions, so that we can alter them to serve the purposes of the surrogate device. Another eventual goal is to draft

```
void setup() {

int input_pin = A0; //This pin will control when we turn on the signal
int pos_en = 13;  //this pin enables the high part of the signal
int neg_en = 12;  //this pin enables the low part of the signal
int data_out = 11; //This pin will control our output voltage
int sensor_value = 0; //voltage level of input
int trig = 2; //pin for the switch to be connected to
int val = 0; //trigger value

pinMode(pos_en, OUTPUT);
pinMode(neg_en, OUTPUT);
pinMode(data_out, OUTPUT);
pinMode(input_pin, INPUT);
pinMode(trig, INPUT);
}

void loop() {
  val = digitalRead(trig);
  if (val == 0) {
    analogWrite(data_out,51); // output 1V
  } else {
    analogWrite(data_out,77); // output 1.5V AKA Triggered ALARM
  }

  sensor_value = analogRead(input_pin);
  //if the input voltage has surpassed 20V (SLC sends a message)
  // the surrogate device will sent back a single period of a 100HZ
  // digital wave, This is to mimic the behavior that was observed
  //on the oscilloscope
  if (sensor_value > 820) {
    digitalWrite(pos_en,HIGH);
    delay(50);
    digitalWrite(pos_en,LOW);
    digitalWrite(neg_en,HIGH);
    delay(50);
    digitalWrite(neg_en,LOW);
}
```

Fig. 12. Image depicting the Arduino code that uses the implemented surrogate device design.

and implement the spoofed SLC message that will be put on the surrogate device once it is assembled, which can be achieved by using the previous research concerning SLC message structure and continuing our current RE efforts.

The next steps for the surrogate device will be to physically build the device that was designed this semester, as well as test out the simple code that was created for it. The system is not terribly difficult to build the lab just lacks the parts needed and as the semester wrapped up time was short. The simple code for the device needs to be tested to see if it can perform its job, or if a new approach is needed. The physical design is stable and all that is needed is it is to be constructed.

After the surrogate device is constructed the next step that needs to be done is implementing the ability for it to read and respond to the SLC messages that are sent by the gateway. If the surrogate device is able to accurately read what the SLC is saying then it will be able to blend in better, thus reducing the risk of the surrogate device flagging an error in the system. Ultimately, the potential of the surrogate device is limitless once it is able to decode SLC messages, so giving it that capability should be the main priority of the surrogate device next semester.

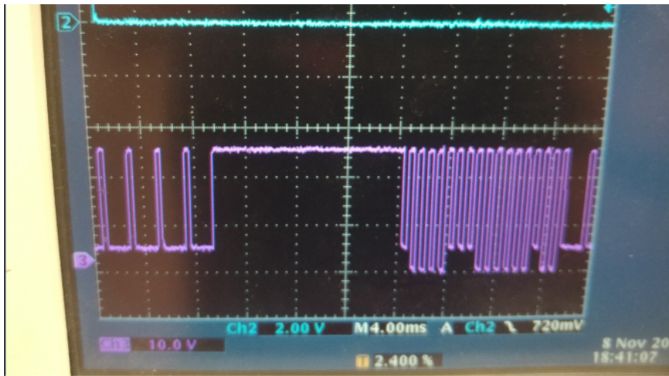


Fig. 13. This image depicts captures taken from the oscilloscope when the pull station was in an idle state.

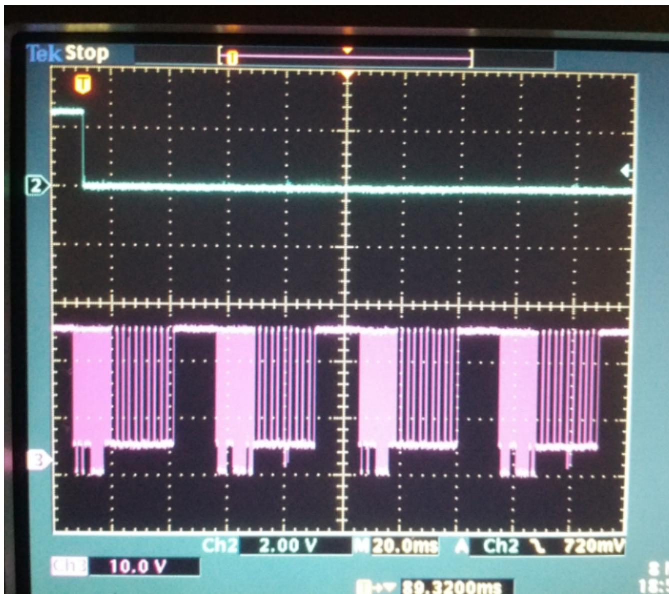
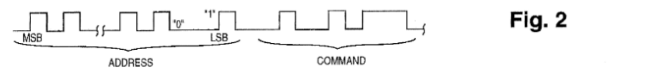


Fig. 14. This image depicts captures taken from the oscilloscope when the pull station is in an alarm state.



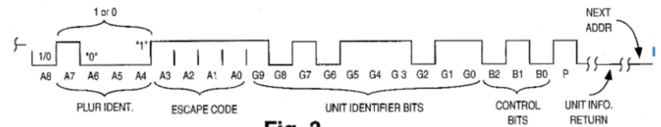


Fig. 15. This image depicts the part of the FlashScan patent that describes the 17-bit address word and command word combination that makes up SLC messages.

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] A. Bussey, D. Chou, J. Y. Kim, S. Suman, S. Wright, and D. Keskin, "Swift wireless fire alarm system analysis," Nov. 2022.

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