



**CONSTRUCTING HONEYPOTS TO DEFEND
BUILDING AUTOMATION SYSTEMS**

THESIS

Caleb E. Mays, Captain, USAF
AFIT-ENG-MS-17-M-049

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-17-M-049

CONSTRUCTING HONEYPOTS TO DEFEND
BUILDING AUTOMATION SYSTEMS

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Cyber Operations

Caleb E. Mays, B.S.C.S.

Captain, USAF

March 2017

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-17-M-049

CONSTRUCTING HONEYPOTS TO DEFEND
BUILDING AUTOMATION SYSTEMS

THESIS

Caleb E. Mays, B.S.C.S.
Captain, USAF

Committee Membership:

Lieutenant Colonel Mason J. Rice, Ph.D.
Chair

Barry E. Mullins, Ph.D., P.E.
Member

Lieutenant Colonel John M. Pecarina, Ph.D.
Member

Abstract

The Internet of Things (IoT) and building automation technology are growing fields. Many automation networks use proprietary protocols though few works evaluate the security of these protocols. This lack of security research leaves consumers' homes and buildings vulnerable to intruders. One such protocol is INSTEON, which maintains general acceptance within the IoT and building automation market.

Concurrently, Software-Defined Radios (SDRs) are experiencing significant growth in part due to low-cost and open-source platforms. Despite the growth of SDRs, research evaluating the security of building automation devices and networks is lacking.

Separately, recent research details the development of ICS honeypots, as using honeypots is a proven method in traditional Information Technology (IT) environment to alert network defenders of a possible attack. Given the current state of cybersecurity, research topics for home and building automation security and ICS security are converging.

This research improves on previous research using an SDR to receive and transmit INSTEON commands. The improved reception tool brings the packet capture capability to industry standards by outputting INSTEON network traffic to Wireshark. Additionally, the wireless Radio Frequency (RF) settings of the SDR are modified and tested in order to increase the packet capture rate of the original receiver. An experiment is conducted to verify the capture rate improves from approximately 40% to almost 75% for the INSTEON protocol. Furthermore, this work incorporates ideas from ICS honeypot research to drive the development of an authentic and targetable INSTEON honeypot. An INSTEON enumeration technique is developed and used to verify the authenticity of the honeypot. Simulated network traffic mimics genuine

INSTEON network communications, making the honeypot targetable. The authentic and targetable honeypot helps protect a building automation network from potential attackers by increasing the amount of work the attacker must do to identify the genuine INSTEON network.

Acknowledgments

To my wife, thank you for your encouragement, sacrifices and for your editing skills. I could not have gotten through this program without you or your support.

To my kids, coming home to play with you was exactly what I needed after staring at computers all day. You two are so energetic, fun, loving and kind. You're the best kids anyone could ask for.

To LTC Rice and Steve Dunlap, thanks for your guidance, assistance, keeping me on track and telling it like it is. Your mentoring was very appreciated throughout this research.

Caleb E. Mays

Table of Contents

	Page
Abstract	iv
Acknowledgments	vi
List of Figures	x
List of Tables	xii
List of Code Listings	xiii
List of Acronyms and Abbreviations	xiv
I. Introduction	1
1.1 Background Scenario	1
1.2 Motivation	2
1.3 Research Goals and Hypothesis	2
1.4 Approach	3
1.4.1 Improving Previous Research	3
1.4.2 Honeypot Development	3
1.5 Assumptions	4
1.6 Research Contributions	4
1.7 Thesis Overview	5
II. Background and Literature Review	6
2.1 Overview	6
2.2 The Internet of Things	6
2.3 INSTEON Protocol	8
2.3.1 INSTEON Documentation	9
2.3.2 Related INSTEON Research	13
2.3.3 INSTEON Protocol Summary	15
2.4 Related Building Automation Security Research	16
2.4.1 Wi-Fi	16
2.4.2 Bluetooth	16
2.4.3 ZigBee and Z-Wave	17
2.5 Related Honeypot Research	17
2.6 Summary	18
III. System Design and Architecture	19
3.1 Overview	19
3.2 Software-Defined Radio	19

	Page
3.3 RfCat Firmware and API	20
3.4 Python Software	21
3.4.1 Receiver	21
3.4.2 Transmitter	23
3.5 Wireshark Integration	23
3.5.1 Generating pcap Data	24
3.5.2 Lua Dissector	25
3.5.3 Wireshark Configuration	27
3.6 Computer Configuration	28
3.7 INSTEON Equipment	29
3.8 Honeypot Design Considerations	29
3.8.1 Targetability	30
3.8.2 Authenticity	31
3.9 Summary	33
IV. Methodology	34
4.1 Overview	34
4.2 Problem Statement	34
4.2.1 Goals	34
4.2.2 Approach	34
4.3 Pilot Studies	35
4.3.1 Determining Packet Count	36
4.3.2 Enumerating Devices	36
4.4 Packet Capture Rate Test	38
4.4.1 Test Environment	38
4.4.2 Experiment Design	38
4.4.3 System Under Test and Component Under Test	39
4.4.4 Control Variables	39
4.4.5 Assumptions	41
4.5 Honeypot Functional Test	41
4.5.1 Test Environment	42
4.5.2 Experiment Design	42
4.6 Summary	44
V. Results and Analysis	45
5.1 Overview	45
5.2 Packet Capture Rate Test	45
5.2.1 Varying Center Frequency	45
5.2.2 Varying Frequency Shift	45
5.2.3 Varying Symbol Rate	46
5.2.4 All Trials	46
5.3 Honeypot Functional Test	49

	Page
5.4 Summary	51
VI. Conclusion	53
6.1 Overview	53
6.2 Research Conclusions	53
6.2.1 Packet Capture Rate Test	53
6.2.2 Honeypot Functional Test	54
6.3 Research Significance	54
6.3.1 Technical Contributions	54
6.3.2 Functional Contribution	54
6.4 Future Work	55
6.4.1 Develop Additional Devices	55
6.4.2 Performance Test	55
6.4.3 Reporting and Logging Capabilities	55
6.5 Summary	56
Appendix A. Code	57
Bibliography	115

List of Figures

Figure		Page
1.	Network of Things network diagram	8
2.	INSTEON network diagram.	9
3.	Packet structure described in INSTEON Whitepaper: The Details [20].	11
4.	Packet structure from INSTEON: Developer’s Guide and claimed by Shipley and Gooler [5, 19].	14
5.	System architecture.	20
6.	YARD Stick One.	21
7.	Wireshark folders menu.	26
8.	User DLTs edit menu.	28
9.	INSTEON equipment.	30
10.	Results of packet capture rate pilot study.	36
11.	Device identity request, acknowledgment and broadcast messages.	37
12.	Broadcast message packet structure [5].	37
13.	Packet capture rate test environment.	39
14.	System under test and component under test.	40
15.	Functional test environment.	42
16.	Status request and response messages.	43
17.	INSTEON network baseline enumeration.	44
18.	Packet capture results for varying center frequencies.	46
19.	Packet capture results for varying frequency shifts.	47
20.	Packet capture results for varying symbol rate.	47
21.	All trials results.	48

Figure		Page
22.	General Linear Model analysis.	48
23.	Honeypot baseline enumeration.	49
24.	Depiction of an INSTEON network and a honeypot network.	51
25.	Depiction of an INSTEON network and five honeypot networks.	52

List of Tables

Table	Page
1. RF specifications from INSTEON Whitepaper: The Details [20].	10
2. INSTEON flag description [20].	12
3. RF specifications from INSTEON: Developer's Guide [5].	13
4. INSTEON RF specifications from all sources [5, 20, 19].	15
5. Computer configuration.	28
6. Packet capture rate trials.	40
7. INSTEON network compared to honeypot network.	50

List of Code Listings

Listing	Page
1. Receiver code excerpt.....	22
2. Transmitter code excerpt.....	23
3. Generating pcap data excerpt.....	25
4. init.lua excerpt to include dissector.	26
5. INSTEON dissector excerpt.	27
6. Honeypot configuration file example.	31
7. Honeypot configuration file example.	33

List of Acronyms and Abbreviations

ACK Acknowledge.

API Application Programming Interface.

BLE Bluetooth Low Energy.

CCA Clear Channel Assessment.

CRC Cyclic Redundancy Check.

DLT Datalink Type.

DoS Denial of Service.

FCC Federal Communications Commission.

FSK Frequency Shift Keying.

GLM General Linear Model.

HVAC Heating, Ventilation and Air Conditioning.

ICS Industrial Control Systems.

IoT Internet of Things.

IT Information Technology.

LED Light Emitting Diode.

NAK Non-acknowledge.

NIST National Institute of Standards and Technology.

NoT Network of Things.

PoS Point-of-Sale.

RF Radio Frequency.

SDR Software-Defined Radio.

SP Special Publication.

USB Universal Serial Bus.

YARD Yet Another Radio Dongle.

CONSTRUCTING HONEYPOTS TO DEFEND BUILDING AUTOMATION SYSTEMS

I. Introduction

1.1 Background Scenario

The family was in the car ready for a great weekend getaway. They had planned a 3-day camping trip complete with fishing, hiking and s'mores. Dad was pulling away from the house and pushed the button on his phone application to close the garage door. Mom verified that the external doors were locked and the thermostat was set from her smartphone. Dad maneuvered the car past parked cars on the street and the family started what they all hoped would be a great weekend.

The weekend was great! The weather was perfect; not too hot, not too cold, no rain. As the family drove home and their car approached the driveway everything looked normal. The garage door was down, doors shut, no broken windows and no signs of a break-in. Inside, however, it was a different story. The house had been burgled. The TV and entertainment systems were gone. Jewelry was missing. Family heirlooms had been taken. Someone had gained access through the garage door. The garage door could only have been opened and closed through the application on the phone, they all thought. How did this happen?

Little did they know that when they were driving away, the intruder was in one of the cars on the street. The burglars noticed the full trunk and car-top carrier, inferring the family would be gone all weekend. They used a cheap radio and computer to see the wireless commands used to open and close the garage door. With a few simple

keystrokes, the hacker replayed the command to open the garage door and walked into the house. Could the family do anything more to secure their house?

1.2 Motivation

The Internet of Things (IoT) provides users with convenience, but the security of building automation networks is lacking. Manufacturers tend to prioritize convenience and usability over user security. Users are unknowingly often making themselves, their businesses and their personal property more vulnerable to intruders – not less – by using building automation systems. As home and building automation systems becomes more prevalent, defending these automation networks is of utmost importance for true security. The lack of security and research about building automation defense provides the motivation to research and create security mechanisms in defense of these networks. This research presents the development and evaluation of a honeypot to protect a building automation system.

1.3 Research Goals and Hypothesis

This research answers the following question: can building automation systems be protected by decoy honeypot networks? The initial hypothesis is that decoy honeypot networks can provide sufficient authenticity and targetability to attract would-be attackers. Such a decoy will effectively protect building automation systems. In order to achieve this, two prerequisites and two primary goals must be met to prove or disprove this research.

Prerequisites:

1. Gain fundamental knowledge of the INSTEON wireless protocol.
2. Gain understanding of applicable honeypot ideas, techniques or technologies.

Goals:

1. Develop or improve methods for transmitting and receiving INSTEON network communication using a Software-Defined Radio (SDR).
2. Develop a virtual honeypot to emulate INSTEON devices and evaluate the authenticity and targetability of the honeypot to protect building automation systems.

1.4 Approach

To accomplish these goals, a set of SDR-based tools for INSTEON packet reception and transmission is improved from previous research (see [19]). The modified SDR tools are then extended to create an authentic and targetable INSTEON honeypot. The honeypot is used to distract would-be attackers, thereby defending the genuine INSTEON network and devices.

1.4.1 Improving Previous Research

Previous research (see [19]) provides a baseline for transmitting and receiving INSTEON traffic. The original receiver is limited thus it can be significantly improved. An experiment is designed to test the original receiver's packet capture rate and finds the original receiver captured approximately 40% of INSTEON packets. Based on the INSTEON documentation (see [5, 20]) and the previous research, select variables in the receiver's Radio Frequency (RF) settings are adjusted to better tune the radio. The modifications are tested, and the packet capture rate is improved to nearly 75%.

1.4.2 Honeypot Development

The SDR tools are extended to create a honeypot. The honeypot is developed using ideas and techniques from Industrial Control Systems (ICS) honeypot research

[10, 28]. The primary design considerations for the honeypot are targetability and authenticity. Network traffic simulation makes the honeypot a targetable decoy while INSTEON device identification information is used as the basis to evaluate authenticity. The honeypot provides a defensive layer to the INSTEON protocol.

1.5 Assumptions

This research assumes the following:

- ICS honeypot ideas and techniques transfer easily to IoT protocols.
- Computing resources for IoT components are designed to serve a specific purpose and are generally focused on device availability and user convenience, lacking basic security mechanisms.
- IoT components do not have the extensive processing power and memory resources of general purpose computers that are designed to run multiple applications at once.

1.6 Research Contributions

This research contributes to the overall understanding of the INSTEON communications protocol and the defense of building automation. The first contribution is the improvement of the packet capture rate for the protocol. The second contribution is to integrate the receiver output with Wireshark to view INSTEON packets. This brings INSTEON network traffic analysis to industry standard. These two contributions were presented at The Wireless Village at DEFCON 24 (see [14]). The third contribution is the development of authentic, targetable decoy honeypot networks which hide a genuine INSTEON network, providing defense to the system.

1.7 Thesis Overview

Chapter II describes the background and establishes context for this research including specific information for the INSTEON protocol and related INSTEON research. Chapter III describes the system design and architecture used for this research. Chapter IV explains the methods used to improve the previous INSTEON receiver and evaluate the honeypot. Chapter V presents the results of the experiments. Finally, Chapter VI offers conclusions and opportunities for further research.

II. Background and Literature Review

2.1 Overview

This chapter provides a basic understanding of the background knowledge necessary for the research presented in this work. Section 2.2 discusses the growth of the Internet of Things (IoT) and summarizes IoT definitions. Section 2.3 provides necessary details for the INSTEON protocol by reviewing published INSTEON documentation and the previous work accomplished to reverse-engineering the protocol. Section 2.4 summarizes other IoT and building automation security research. Finally, Section 2.5 discusses honeypot techniques and ideas in both Information Technology (IT) and Industrial Control Systems (ICS) environments.

2.2 The Internet of Things

The IoT and building automation technology are growing fields. These technologies connect to appliances in buildings and homes from anywhere in the world. Homeowners can turn on lights, lock their doors and change their thermostat settings all on their mobile devices. Building maintainers control access and industrial heating, ventilation and air conditioning (HVAC) settings with a few mouse-clicks. While IoT is a relatively new field, a recent report by Transparency Market Research projects it to be a \$21.6 billion global industry by 2020 [22]. Furthermore, the Gartner research firm predicts that IoT devices – for both business and in-home use – would rise from six billion to 20 billion devices by 2020 [9].

The IoT is comprised of individual Networks of Things (NoTs) connected together. The National Institute of Standards and Technology (NIST) recently published guidelines and definitions for a NoT. NIST Special Publication (SP) 800-183 defines terms and outlines generic security principles for a NoT. Figure 1 portrays a NoT diagram

using the NIST definitions.

Sensors: The primary function of a sensor is to measure some physical property (e.g., temperature, weight or sound) and output data based on that property. Examples of sensors are cameras, microphones, motion, leak or open/close sensors. Sensors are physical devices and can typically be associated with the ‘thing’ to which they are attached (e.g., door sensor).

Aggregators: An aggregator, sometimes known as a network hub, collects the data provided from sensors. It then starts to make sense of that data. For example, a leak sensor may detect water on the ground and will output that information to the aggregator. The aggregator will collect that information and take the appropriate measures to alert the user.

Communication Channel: The communication channel is the physical connection between networked devices. There can be multiple channels within a given network and the communication channel may be wired or wireless (e.g., Power Line Control, Wi-Fi or Ethernet).

eUtility: eUtilities are the cloud computing platforms and services they provide including web servers and databases (e.g., Amazon Web Service and Microsoft Cloud). A user will typically connect via a mobile device application to the eUtility, not directly to an aggregator, to receive status updates for the devices.

Decision Triggers: Finally, decision triggers are summarized as events or moments in time when an action is performed. The action may change the physical aspects of something else on the network (e.g., a thermostat senses the temperature falls below 70 degrees Fahrenheit and turns on the heating system).

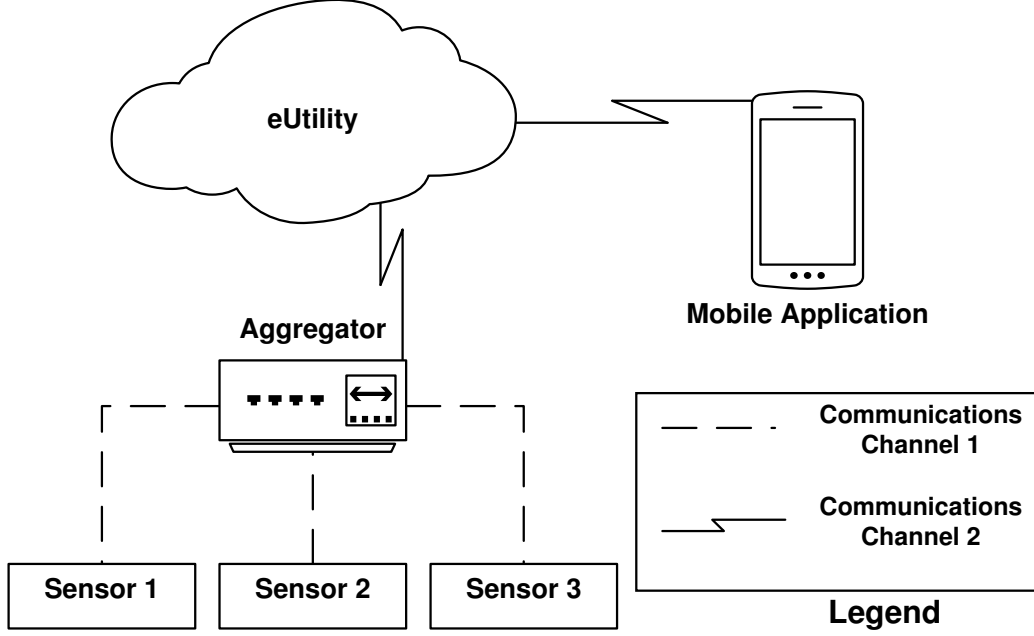


Figure 1. Network of Things network diagram

2.3 INSTEON Protocol

INSTEON is one of the earliest manufacturers of home automation technology. The company has been producing home automation devices for over ten years. Revention, a Point-of-Sale (PoS) system manufacturer, recently announced plans to integrate with INSTEON [16]. The two companies intend to integrate INSTEON's automation technology with Revention's PoS system to give restaurants easy building automation control on one user interface. A wide range of INSTEON devices including LED bulbs, dimmable light modules and switches, open/close sensors, security cameras and other automation devices are available. INSTEON enjoys general acceptance within the IoT community through interoperability with the Amazon Echo and Sonos home surround sound system. Applications on Apple iOS, Google Android and Microsoft Windows 8/10 provide the interface for monitoring and modifying the devices.

Figure 2 diagrams a basic INSTEON network. INSTEON devices communicate

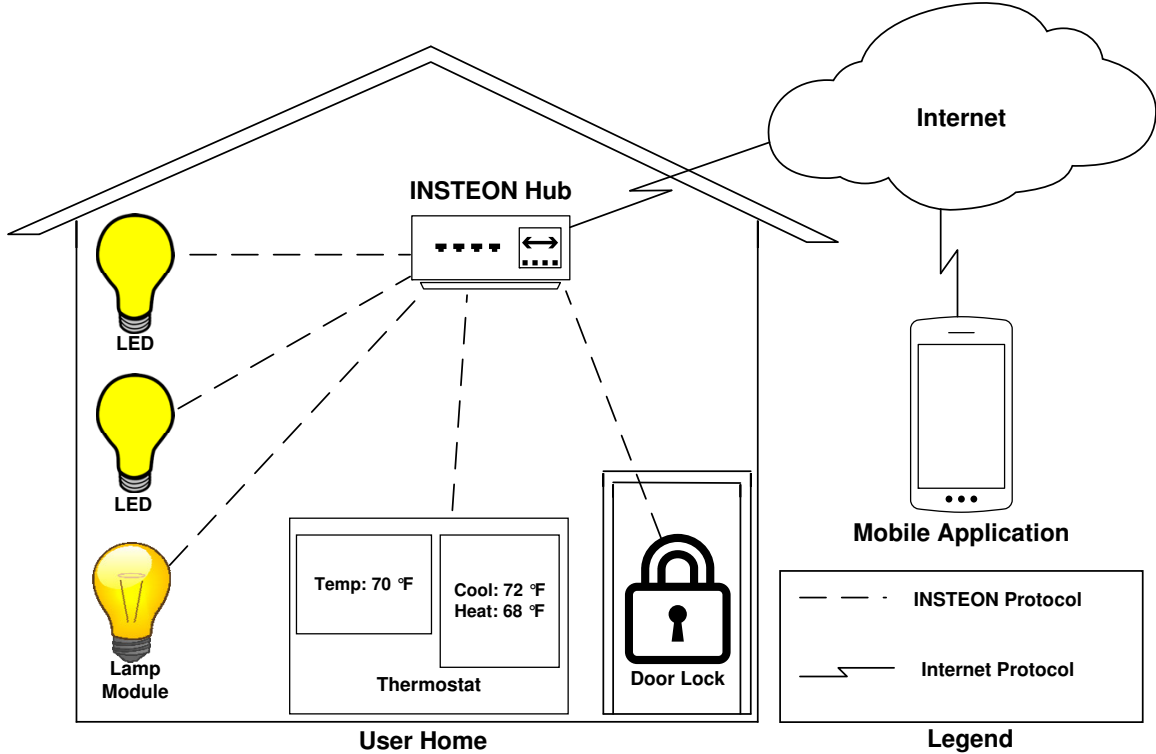


Figure 2. INSTEON network diagram.

by either wireless Radio Frequency (RF) or wired Power Line Control communications, though this research focuses on INSTEON’s wireless RF protocol. The devices are statically networked during the setup process. The user performs a series of button-pushes or uses INSTEON’s smartphone application to network the devices. One device is the controller (or master) device while the other device becomes the responder (slave) device. INSTEON devices can communicate in a mesh network to limit a malfunctioning device from disrupting service. The INSTEON Hub connects to the user’s router for Internet access. The following sections detail the prerequisite information about the INSTEON protocol necessary for this research effort.

2.3.1 INSTEON Documentation

INSTEON created their own communications protocol and published many of its specifications. Two primary documents: (i) INSTEON Whitepaper: The Details [20]

and (ii) INSTEON: Developer’s Guide [5] are intended to inform developers, security professionals and users about the protocol. Two additional INSTEON documents describe commands to communicate with the devices and INSTEON device category information [6, 7]. These four INSTEON documents are described in detail in the following sections.

2.3.1.1 INSTEON Whitepaper: The Details

INSTEON Whitepaper: The Details (see [20]) describes the specifications for INSTEON’s RF and Power Line Control protocol settings, packet structure, hopping mechanism, timing scheme, network diagram and many other details. INSTEON claims the center frequency for their radios is 915 MHz and that the frequency modulates using Frequency Shift Keying (FSK) at 64 KHz shift. The documentation also claims that the symbols rate is 76.8 KBaud. Finally, the symbols are encoded using the Manchester encoding method. Table 1 summarizes the wireless RF specifications from this document.

Table 1. RF specifications from INSTEON Whitepaper: The Details [20].

RF Specification	Value
Center Frequency*	915 MHz
Modulation Method	FSK
FSK Shift	64 KHz
FSK Symbol Rate	76.8 KBaud
Encoding Method	Manchester
<i>*United States center frequency only</i>	

INSTEON also defines their own packet structure as illustrated in Figure 3. The INSTEON packet contains five mandatory fields and one optional field. INSTEON: The Details claims that the structure starts with the source device address, followed by the destination device address. These device addresses are three bytes long. They

are set during manufacturing and remain static. The packet structure continues with one byte for flag data, two bytes for commands and finishes with a one byte Cyclic Redundancy Check (CRC).

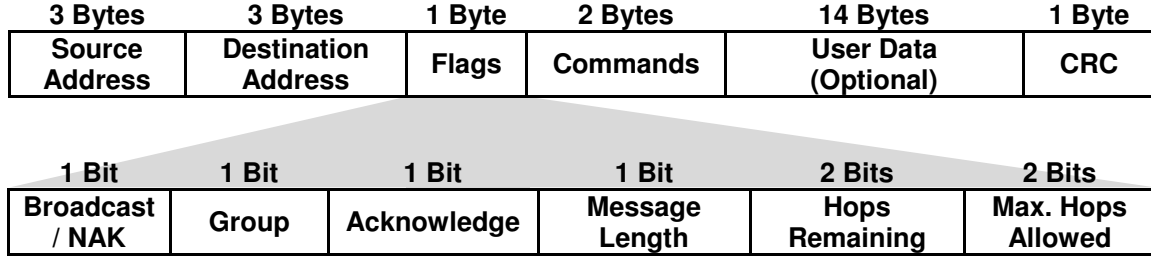


Figure 3. Packet structure described in INSTEON Whitepaper: The Details [20].

Table 2 provides the meaning for each bit in the flags field. INSTEON flags describe the message type, message length, number of retransmissions (hops) remaining and maximum retransmissions allowed for the packet. The first three flag bits specifically describe which kind of the message the packet contains (e.g., a direct message, acknowledgment message, etc). The message length bit defines whether the packet is standard length or extended length. Standard length packets are ten bytes long while extended length packets include the optional 14 bytes of user-defined data to make these packets 24 bytes long. The final 4 bits maintain the number of hops remaining and maximum hops allowed for the packet. INSTEON messages are not routed like typical Internet Protocol messages. Instead, packets traverse the network by hopping across devices. Each packet may hop a maximum of three times as it traverses the network to the destination device.

2.3.1.2 INSTEON: Developer's Guide

Other INSTEON protocol information is found in INSTEON: Developer's Guide (see [5]). While most of the specifications in this document match the previous document, a few details about the wireless RF settings differ. Specifically FSK deviation

Table 2. INSTEON flag description [20].

Bit Position	Flag	Meaning
Bit 7–Broadcast or Non-acknowledge (NAK) Bit 6–Group Bit 5–Acknowledge (ACK)	Message Type	111 = NAK of Group Cleanup Direct Message
		110 = Group Broadcast Message
		101 = NAK of Direct Message
		100 = Broadcast Message
		011 = ACK of Group Cleanup Direct Message
		010 = Group Cleanup Direct Message
		001 = ACK of Direct Message
		000 = Direct Message
Bit 4	Extended	1 = Extended Message 0 = Standard Message
Bit 3 Bit 2	Hops Left	11 = 3 hops remaining 10 = 2 hops remaining 01 = 1 hop remaining 00 = 0 hops remaining
Bit 1 Bit 0	Max Hops	11 = 3 hops allowed 10 = 2 hops allowed 01 = 1 hop allowed 00 = 0 hops allowed

and symbol rate are listed as 200 KHz and 9.124 KBaud, respectively. Table 3 summarizes the RF specifications from this document.

Additionally, this document describes the INSTEON packet structure differently. It places the flags field as the first field, followed by the source address, destination address, commands, optional user data and ends with the CRC. This structure is illustrated in Figure 4.

Table 3. RF specifications from INSTEON: Developer’s Guide [5].

RF Specification	Value
Center Frequency*	915 MHz
Modulation Method	FSK
FSK Shift	200 KHz
FSK Symbol Rate	9.124 KBaud
Encoding Method	Manchester
<i>*United States center frequency only</i>	

2.3.1.3 INSTEON: Command Tables

This INSTEON document (see [6]) provides useful information for interacting with INSTEON devices. The list of commands, updated as of 2007, is likely outdated and not entirely accurate, but includes information about commands not seen in normal INSTEON network traffic (e.g., ping and ID Request).

2.3.1.4 INSTEON: Device Categories and Product Keys

The final piece of useful INSTEON documentation found during this research is a table of information documenting INSTEON device categories (see [7]). As with the command tables document, this document is likely outdated but the device categories are still applicable and used in current INSTEON devices. These device categories become useful for enumerating the INSTEON devices and become the basis for producing an authentic INSTEON honeypot.

2.3.2 Related INSTEON Research

Security researchers Peter Shipley and Ryan Gooler reverse-engineered the INSTEON protocol and developed a basic transmitter and receiver using the YARD Stick One [12, 19]. They presented their efforts at DEFCON 23 declaring that INSTEON’s documentation was not correct. They claimed that INSTEON:

- Did not use center frequency 915 MHz, but rather 914.975 MHz (914.95 MHz coded in SDR receiver).
- Did not use traditional FSK but rather inverted FSK.
- Did not use the FSK shift value detailed in the documentation.
- Did not use the symbol rate detailed in the documentation.
- Did not use true Manchester encoding.
- Did not use the CRC algorithm as detailed in the documentation.
- Did not use the packet structure as detailed in the documentation. Figure 4 illustrates Shipley and Gooler’s claimed INSTEON packet structure.

1 Byte	3 Bytes	3 Bytes	2 Bytes	14 Bytes	1 Byte
Flags	Destination Address	Source Address	Commands	User Data (Optional)	CRC

Figure 4. Packet structure from INSTEON: Developer’s Guide and claimed by Shipley and Gooler [5, 19].

While Shipley and Gooler’s transmitter and receiver tool works, it is limited and could be improved. The limitations of their tool are described in the following paragraphs.

First, the receiver program output the packets directly to the terminal window. This solution is not ideal as Wireshark is the industry standard network traffic packet analyzer. During this research their receiver is modified to view the INSTEON packets in Wireshark. This modification is described in Section 3.5.

Second, the receiver seemingly drops packets. Sections 4.3.1 and 4.4 describe the pilot study and the experiment used to test and improve the packet capture rate.

Third, Shipley and Gooler did not present any method to enumerate or otherwise identify INSTEON devices. This limitation provided the opportunity to find

commands to query, identify and enumerate INSTEON devices beyond the device address.

Finally, Shipley and Gooler did not outline any methods to secure or defend the protocol. This research begins the development of an INSTEON honeypot to secure building and home automation systems.

2.3.3 INSTEON Protocol Summary

The INSTEON documentation and previous research are inconsistent regarding the exact wireless specifications and packet structure for the INSTEON protocol. One document states the center frequency is 915 MHz while Shipley and Gooler state the center frequency is 914.975 MHz. Further examination of Shipley and Gooler’s code reveals a third potential center frequency value of 914.95 MHz. Neither the documentation nor previous research agree on the FSK shift value. Finally, the documentation and previous research disagree about the FSK symbol rate, listing 76.8 KBaud in one document, 9.124 KBaud in another document and 9.125 KBaud from Shipley and Gooler’s research. Table 4 provides a summary for how each piece of documentation varies. These conflicts lead a researcher to ask: which is correct?

Table 4. INSTEON RF specifications from all sources [5, 20, 19].

Specification	The Details	Developer’s Guide	Shipley & Gooler
Center Frequency	915.000 MHz	915.000 MHz	914.975 MHz 914.950 MHz (in code)
Modulation Method	FSK	FSK	Inverted FSK
FSK Shift	64 KHz	200 KHz	150 KHz
FSK Symbol Rate	76.800 KBaud	9.124 KBaud	9.125 KBaud
Encoding Method	Manchester	Manchester	“Tokenized” Manchester

2.4 Related Building Automation Security Research

The home and building automation market has many technologies and protocols available to consumers (e.g., Wi-Fi, Bluetooth, ZigBee, Z-Wave and INSTEON). Multiple studies have compared the technical specifications and capabilities for these automation technologies [1, 11, 30], but few published works evaluate the security of proprietary IoT protocols. The lack of security research leaves corporate offices and individual residences vulnerable and dangerously accessible to intruders.

2.4.1 Wi-Fi

Wi-Fi is a popular technology that is already broadly incorporated into homes and businesses. The Nest line of products and Amazon’s Echo use this protocol. Wi-Fi attacks such as the deauthentication or rogue access point attacks are well-known and tools are incorporated into the Kali version of Linux specifically for evaluating Wi-Fi security. Wi-Fi can be secured through strong encryption techniques and as Wi-Fi security increases, Wi-Fi enabled IoT devices will benefit.

2.4.2 Bluetooth

Bluetooth Low Energy (BLE) researchers Anthony Rose and Benjamin Ramsey [17] showed that many companies do not implement optional BLE security features (e.g., encryption through properly pairing devices). Implementing Bluetooth pairing improperly leads to confidentiality problems. Rose and Ramsey passively sniffed passwords for several BLE-enabled locks. They documented several BLE security flaws and presented tools to open many of the locks. Further, when presented with improper commands (e.g., unrecognized characters), many locks failed into a non-recoverable state causing a Denial of Service (DoS) to the user.

2.4.3 ZigBee and Z-Wave

Joseph Hall and Benjamin Ramsey researched the ZigBee and Z-Wave protocols and developed tools for securing these automation networks [13, 15]. The Philips Hue brand of lights is one product that uses ZigBee. Hall and Ramsey were able to capitalize on Z-Wave broadcast commands to enumerate Z-Wave networks. Their research demonstrates over-availability issues and general lack of confidentiality in the Z-Wave protocol. Additionally, they found similar issues to the Bluetooth research where manufacturers did not properly implement encryption.

2.5 Related Honeypot Research

This section provides the prerequisite understanding of honeypot ideas, techniques and technologies. The idea of a honeypot – a separate decoy network designed to lure potential attackers away from a genuine operational network – is not new. Research for traditional Information Technology (IT) honeypots prior to 2006 was constrained by costly hardware. Virtual technologies have since made traditional honeypots more realistic and prevalent, while large research IT honeynets have been actively used to capture malicious software [3, 25].

Industrial Control Systems are being scrutinized for their general lack of security mechanisms and the important physical systems they control. In 2014, the Director of the National Security Agency, Admiral Mike Rogers, stated that multiple nation-state actors have the capability to cause physical destruction, personal injury and even death by using cyber-attacks against utility systems in the United States [4].

Due to the serious threat to the national critical infrastructure, research is being directed toward investigating the feasibility of ICS honeypots. The Honeynet Project produced significant development toward an open source ICS honeypot in May 2013 [26]. In 2015 Winn et al. [28] researched an ICS honeypot and extended the “honeyd”

program to “honeyd+”, making it more authentic. The researchers discussed two levels of interaction a honeypot can display: (i) high-level interaction and (ii) low-level interaction. Winn et al. also made the argument that the targetability of the honeypot needs to match the intended network, and the authenticity of the decoy needs to be adequately convincing for an attacker to target the fake devices instead of the true devices. Girtz et al. [10] improved the performance of “honeyd+” by developing an application layer emulator.

2.6 Summary

In summary, security writer Bruce Schneier argues that IoT has given the Internet “hands and feet” with the ability to change a physical system through cyber means [18]. He describes IoT and cyber-physical systems (i.e., Industrial Control Systems) interchangeably. Given the current state of cyber-security, research topics for ICS honeypots and IoT security are converging. The research presented in this work applies ICS honeypot research ideas and techniques to develop and evaluate a honeypot with the intent to mimic, hide and thereby defend an INSTEON automation network.

III. System Design and Architecture

3.1 Overview

This chapter presents the design and architecture of the system, as seen in Figure 5, by discussing the implementation of each component. The system discussed in this work is an extension of Shipley and Gooler’s previous work. Section 3.2 discusses the role and use of Software-Defined Radios (SDRs) in this system. Section 3.3 briefly discusses the RfCat firmware and the RfCat Application Programming Interface (API). Section 3.4 discusses the Python software used to program the SDR while Section 3.5 describes the process to integrate the output to Wireshark. Section 3.6 briefly talks about the computer configuration used for this research, and Section 3.7 lists the INSTEON equipment used in this research. Finally Section 3.8 describes the honeypot implemented during this research.

3.2 Software-Defined Radio

The Federal Communications Commission (FCC) defines an SDR as a radio in which the frequency, modulation, output power or other settings used to transmit and receive are modified by making changes in software [8]. The system presented here uses an SDR to mimic the functionality of proprietary INSTEON transceivers. Specifically, this system uses the Yet Another Radio Dongle (YARD) Stick One SDR as seen in Figure 6 [12].

The YARD Stick One is an open-source hardware radio. It is a half-duplex transceiver that operates between 300-348 MHz, 391-464 MHz and 782-928 MHz [12]. The settings for the YARD Stick One are programmed in software discussed in Sections 3.3 and 3.4. The YARD Stick One connects to a computer via Universal Serial Bus (USB). To accomplish full-duplex communication, two YARD Stick One

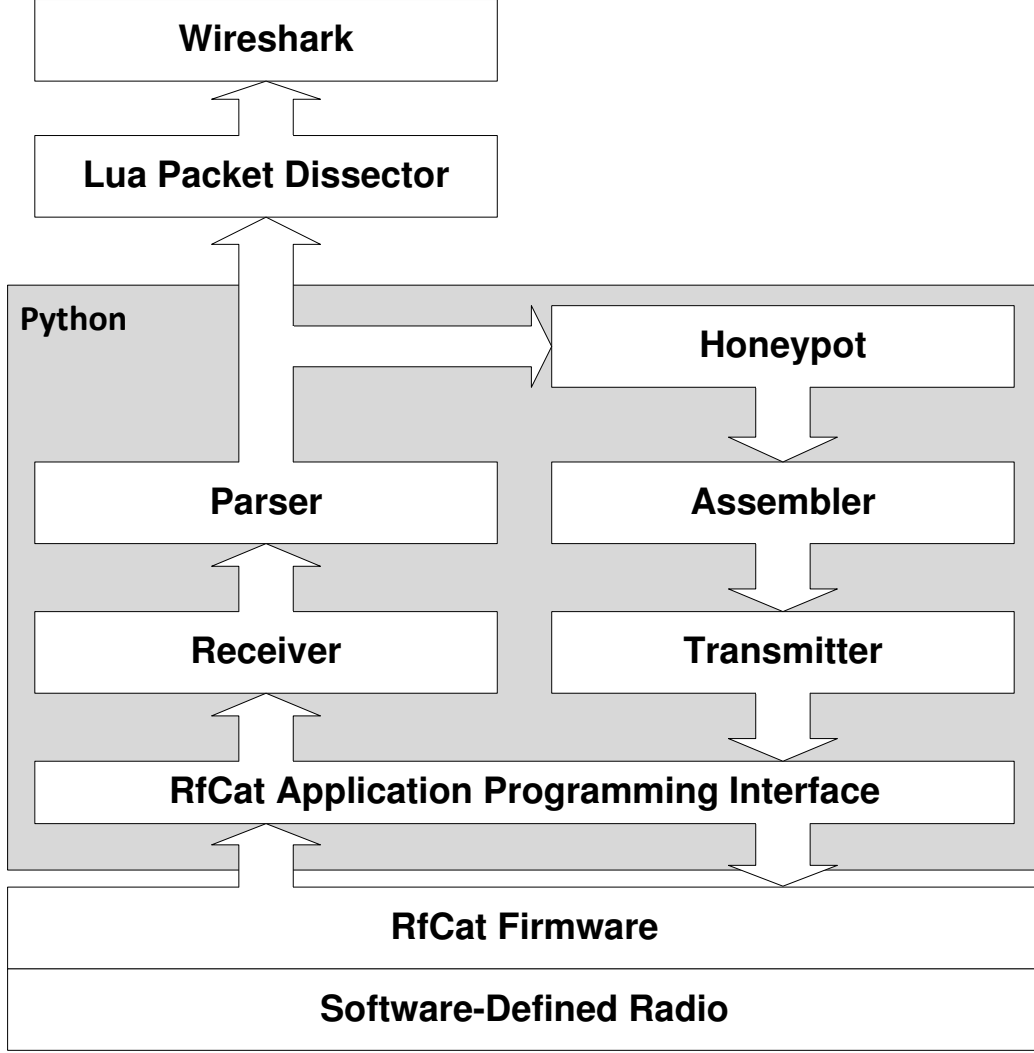


Figure 5. System architecture.

radios must be employed. This design introduces a limitation: the inability to conduct a Clear Channel Assessment (CCA). A CCA is important to optimize wireless network performance, and one should minimize transmission time to prevent other devices from communicating.

3.3 RfCat Firmware and API

The YARD Stick One comes with the RfCat firmware installed. The API that interfaces with the RfCat firmware must be downloaded to the host computer and

installed (see [2, 23]). The RfCat firmware is written in the C programming language. The RfCat API provides Python functions for the developer to define the radio settings for the YARD Stick One.

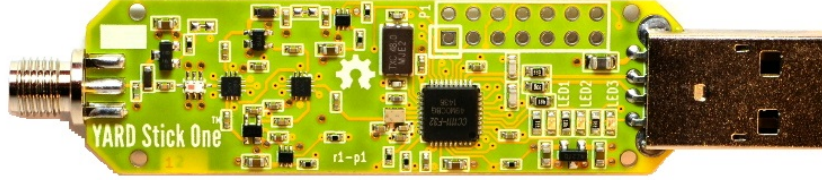


Figure 6. YARD Stick One.

3.4 Python Software

Python is a widely used programming language. It comes pre-installed with the Ubuntu Linux distribution. This research relies heavily on the Python functions provided by the RfCat API and other Python packages to receive and transmit INSTEON messages. The software for the reception and transmission programs are described in the following sections.

3.4.1 Receiver

Listing 1 is an excerpt of the receiver program software. The reception program first initializes a YARD Stick One by calling the `configure_rfcats()` function on line 21. This function defines variables such as center frequency, modulate method, signal rate and more, as seen in lines 1-14. Once the YARD Stick One is configured the software tells it to enter a listening state. This listening state consists of a waiting loop and the `Rfrecv()` function (Listing 1 lines 24-28). The `Rfrecv()` function provides timestamp information and the signal data of the radio waves detected as based on the YARD Stick One's settings.

The receiver program continues by analyzing the resulting signal data. The analysis process includes parsing the data (see Listing 1 line 37) to look for INSTEON packets. The parsed packets are filtered based on if the calculated Cyclic Redundancy Check (CRC) matches the received CRC (line 44). Only valid packets are output to Wireshark (not shown). More details about the Wireshark integration process are found in Section 3.5.

```

1 def configure_rfcats():
2     # local variables.
3     frequency = 915000000 # center freq = 915 MHz
4     baudrate = 9125 # symbol rate = 9.125 KBaud
5     FSKdeviation = 32000 # FSK shift = 2*FSK deviation
6
7     # make the device.
8     device = RfCat(interface)
9
10    # configure the device.
11    device.setFreq(frequency) # sets device frequency.
12    device.setMdmModulation(MOD_2FSK) # sets device modulation method.
13    device.setMdmDeviatn(FSKdeviation) # sets frequency deviation = 1/2 shift.
14    device.setMdmDRate(baudrate) # sets baud (symbol) rate.
15
16    # give radio back to listen function.
17    return device
18
19 def listen():
20     # configures the YARD Stick One.
21     yardstick = configure_rfcats()
22
23     #waiting for <enter> key to stop program
24     while not keystop():
25         try:
26             # RFrecv function returns raw 1s and 0s representing radio waves
27             # based on the RF settings and timestamp information
28             rawdata, timestamp = yardstick.RFrecv()
29
30             # converts timestamp data into the format DD MMM YY HH:MM:SS
31             timestampstring = str(asctime(localtime(timestamp)))
32
33             # join the individual 1s and 0s into bytes for easier reading
34             bytes = ''.join("{0:08b}".format(x) for x in bytearray(rawdata))
35
36             # call parser to begin determining packet contents
37             subpackets = parse_insteon_pkt(bytes, timestampstring)
38
39             # if there are any packets that made it through the parser
40             if subpackets is not None:
41                 # evaluate each subpacket
42                 for packet in subpackets:
43                     # I really care about valid packets here, based on the CRC.
44                     if packet.validcrc == 1:
45                         timestamp = packet.timestamp
46                         readablepacket = packet.hex_str

```

Listing 1. Receiver code excerpt.

3.4.2 Transmitter

Listing 2 is an excerpt of the transmission program software. The software first receives the INSTEON packet to send from the user as input arguments (Listing 2 lines 3 and 6). The user input includes the flags field, source and destination device addresses, the two bytes of command data and the optional user data. The input does not include the CRC as the program calculates it for the user and appends it to the end of the packet (Listing 2 lines 13 and 16). The program then assembles the packet data. Assembling the packet includes encoding it to Manchester encoding and prepending the INSTEON preamble (not shown). Finally, the transmitter configures the YARD Stick One and sends the command (Listing 2 lines 23 and 26).

```
1 def main():
2     # parse all arguments
3     arguments = init()
4
5     # get the packet from the arguments.
6     pktlist = read_raw_pkt(arguments)
7
8     # make sure the packet is the correct length before the CRC is added.
9     # either standard (9 bytes) or extended (23 bytes)
10    if len(pktlist) == 9 or len(pktlist) == 23:
11
12        # calculate the CRC
13        pktcrc = pkt_crc(pktlist)
14
15        # append the CRC to the packetlist.
16        pktlist.append(pktcrc)
17
18        # assemble_packet makes packet Manchester (INSTEON version)
19        # encoded, includes preamble, etc.
20        pkttosend = assemble_packet(pktlist)
21
22        # configures the YARD Stick One.
23        device = configure_rfcats()
24
25        # sends the packet.
26        send_pkt(device, pkttosend)
```

Listing 2. Transmitter code excerpt.

3.5 Wireshark Integration

Wireshark is the industry standard network traffic analyzer. There are currently two ways for dissecting packets in Wireshark. The first method involves creating or

modifying C and C++ code and re-compiling Wireshark. The second, arguably more simple method, involves three steps: (i) program the receiver to generate pcap data; (ii) create a Lua script to dissect the pcap data; and (iii) configure Wireshark to use the Lua dissector (see [29]). The following sections describe each portion of this process in detail.

3.5.1 Generating pcap Data

The first step in this process is to program the receiver to generate pcap data and write the data to a named pipe. Named pipes in Linux are created by using the `mkfifo [pipename]` command. This command is executed before starting the receiver program. Listing 3 is another excerpt from the receiver software, this time with pcap generation functions included (adapted from [24]). Line 2 opens the named pipe for writing in byte-mode. Line 6 creates information that is required at the beginning of pcap data. This information includes four bytes that make up the pcap header, followed by data indicating which pcap file version is following (in this case version 2.4). The following three lines include information about timezone, accuracy of the timestamps, and length of the captured data. The 21st byte (first byte on line 12) is critical to ensuring the protocol data is parsed correctly into Wireshark. This byte tells Wireshark which user-defined Datalink Type (DLT) (i.e., protocol) will follow. Using hexadecimal value 0x93 (decimal value 147) tells Wireshark to parse and display the data using the dissector information for user DLT 147. The Wireshark user-defined DLT will be configured later in Section 3.5.3. These bytes are sent first to the named pipe (Listing 3 line 16) so Wireshark can start parsing the data that is to follow.

Lines 19-31 of Listing 3 are described previously in Section 3.4.1. The next important piece of code for generating pcap data is the `rawtopcap()` function on line

34. This function converts the packet's timestamp to pcap bytes for Wireshark. It also converts the readable packet into byte-data (details not shown). Finally, these bytes are sent to the pipe as input to Wireshark.

```

1  # open the pipe for writing in byte-mode ('wb').
2  mypipe = open(pipename, 'wb')
3
4  # magic numbers, etc for pcap format header.
5  # Wireshark needs this at the beginning of the pipe.
6  bytestring = ('D4 C3 B2 A1' # magic numbers
7               '02 00'       # File format major revision (i.e. pcap <2>.4)
8               '04 00'       # File format minor revision (i.e. pcap 2.<4>)
9               '00 00 00 00'  # timezone info. leaving blank.
10              '00 00 00 00'  # accuracy of timestamps
11              'FF FF 00 00'  # max length of captured packet
12              '93 00 00 00') # Datalink type 93 = DLT_USR 147
13                          #-- the WireShark non-defined protocol.
14
15 # send bytes to pipe.
16 writebytestringtofile(bytestring, mypipe)
17
18 # this is the same loop as in Listing 1.
19 while not keystop():
20     try:
21         rawdata, timestamp = yardstick.RFrecv()
22         timestampstring = str(asctime(localtime(timestamp)))
23         bytes = ''.join("{0:08b}".format(x) for x in bytearray(rawdata))
24
25         subpackets = parse_insteon_pkt(bytes, timestampstring)
26         if subpackets is not None:
27             for packet in subpackets:
28                 # I really care about valid packets here.
29                 if packet.validcrc == 1:
30                     timestamp = packet.timestamp
31                     readablepacket = packet.hex_str
32
33                     # get ready to send this to a pipe.
34                     bytestring = rawtopcap(timestamp, readablepacket)
35                     # send to the pipe.
36                     writebytestringtofile(bytestring, mypipe)

```

Listing 3. Generating pcap data excerpt.

3.5.2 Lua Dissector

The next step is to create a Lua script (see [29]) defining how Wireshark will parse and display the pcap data. Lua is a lightweight programming language. Wireshark uses Lua scripts during the initialization process. In a standard Wireshark installation the configuration files are found at `/etc/wireshark/` and user-created Lua scripts are included in `/usr/share/wireshark`. Figure 7 displays important Wireshark folders.

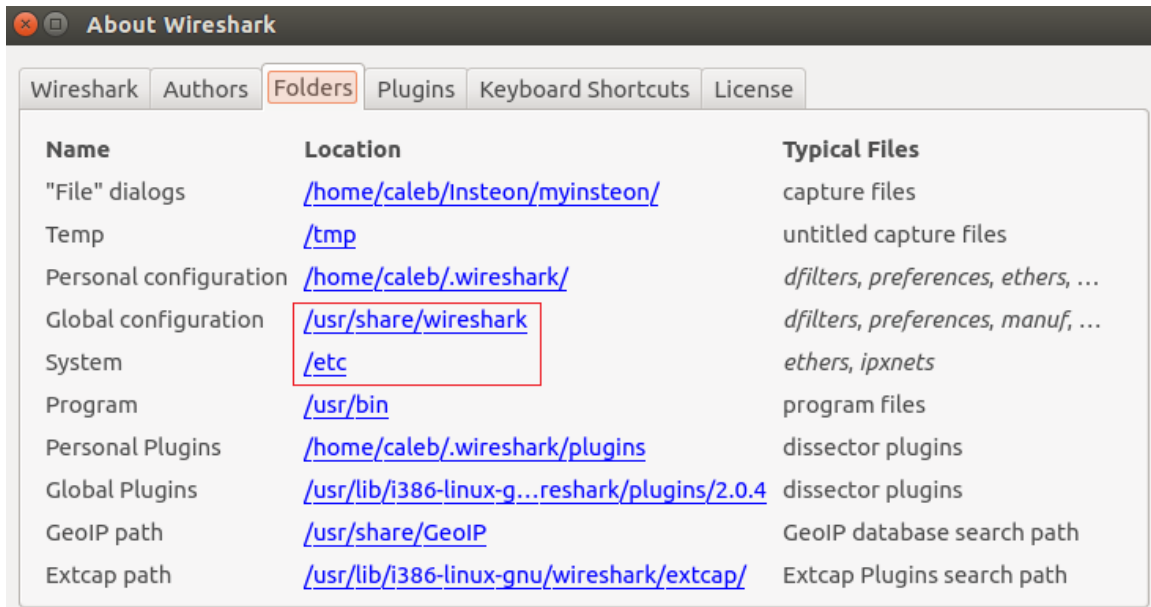


Figure 7. Wireshark folders menu.

Wireshark's main initialization file is `/etc/wireshark/init.lua`. This file creates an environment to define user protocols. The end of this initialization file contains calls to other Lua scripts via the `dofile()` function. Listing 4 portrays the function to include the INSTEON protocol dissector.

```

653 dofile(DATA_DIR.. "console.lua")
654 dofile(DATA_DIR.. "dtd_gen.lua")
655 dofile(DATA_DIR.. "Insteon-dissector.lua") #-- include dissector here.

```

Listing 4. `init.lua` excerpt to include dissector.

The Lua INSTEON dissector builds on the environment created in the `init.lua` file. It defines the protocol so Wireshark can display the packet data. Listing 5 is an excerpt of the INSTEON dissector. Protocols are best described using a tree structure. The `Proto` object is the root of the tree and is created in Listing 5 line 2. The code in lines 5-27 define the INSTEON flags as `ProtoField` objects which attach to the protocol tree, making them branches and leaves. Lua also provides basic programming structures including if-then statements and loops. The code following Listing 5 (not shown) creates fields for the rest of the INSTEON packet. Decisions

based on the packet's flag field determine the information displayed in Wireshark. The full INSTEON dissector is found in Appendix A.

```

1  # -- creates a Proto object, but doesn't register it yet
2  local insteon=Proto("Insteon","Insteon")
3
4  # -- start declaring some fields. Starting with the flags.
5  local flags=ProtoField.new("Flags", "insteon.flags", ftypes.UINT8, nil, base.HEX)
6
7  # -- Each individual flag is attached to the flags tree.
8  local flag_broadcast=ProtoField.new("Broadcast/NAK", "insteon.flags.broadcast",
9      ftypes.BOOLEAN, {"this is a broadcast", "this is not a broadcast"},
10     8, 0x80, "is this a broadcast?")
11
12  local flag_group=ProtoField.new("Group/Not-Group", "insteon.flags.group",
13      ftypes.BOOLEAN, {"this is a group message", "this is not a group message"},
14      8, 0x40, "is this a group msg?")
15
16  local flag_acknowledge=ProtoField.new("Acknowledge", "insteon.flags.acknowledge",
17      ftypes.BOOLEAN, {"acknowledgement", "not an acknowledgement"},
18      8, 0x20, "is this an acknowledgement?")
19
20  local flag_extended=ProtoField.new("Message Type", "insteon.flags.extended",
21      ftypes.BOOLEAN, {"Extended Message", "Standard Message"},
22      8, 0x10, "is this a standard message or extended messages?")
23
24  local flag_hopsleft=ProtoField.new("Hops Left", "insteon.flags.hopsleft",
25      ftypes.UINT8, nil, base.DEC, 0x0C, "hops left")
26
27  local flag_maxhops=ProtoField.new("Max Hops", "insteon.flags.hopsleft",
28      ftypes.UINT8, nil, base.DEC, 0x03, "max hops")

```

Listing 5. INSTEON dissector excerpt.

3.5.3 Wireshark Configuration

The final step to view user-defined protocols in Wireshark is to configure it to use the Lua dissector described above. It is important to start Wireshark from a terminal window with appropriate flags. Specifically, the command `wireshark -i pipename` where `pipename` is the named pipe discussed in Section 3.5.1. Once Wireshark starts, the configuration menu is found at Edit Menu - Preferences - Protocols - DLT_USER - Edit. User 0 (DLT=147) is configured by typing the name of the protocol into the “Payload protocol” field. This tells Wireshark to parse the following bytes using the INSTEON dissector when the DLT byte (21st byte in the pcap data discussed in Section 3.5.1) is decimal value 147. Figure 8 illustrates the menu when Wireshark is configured properly.

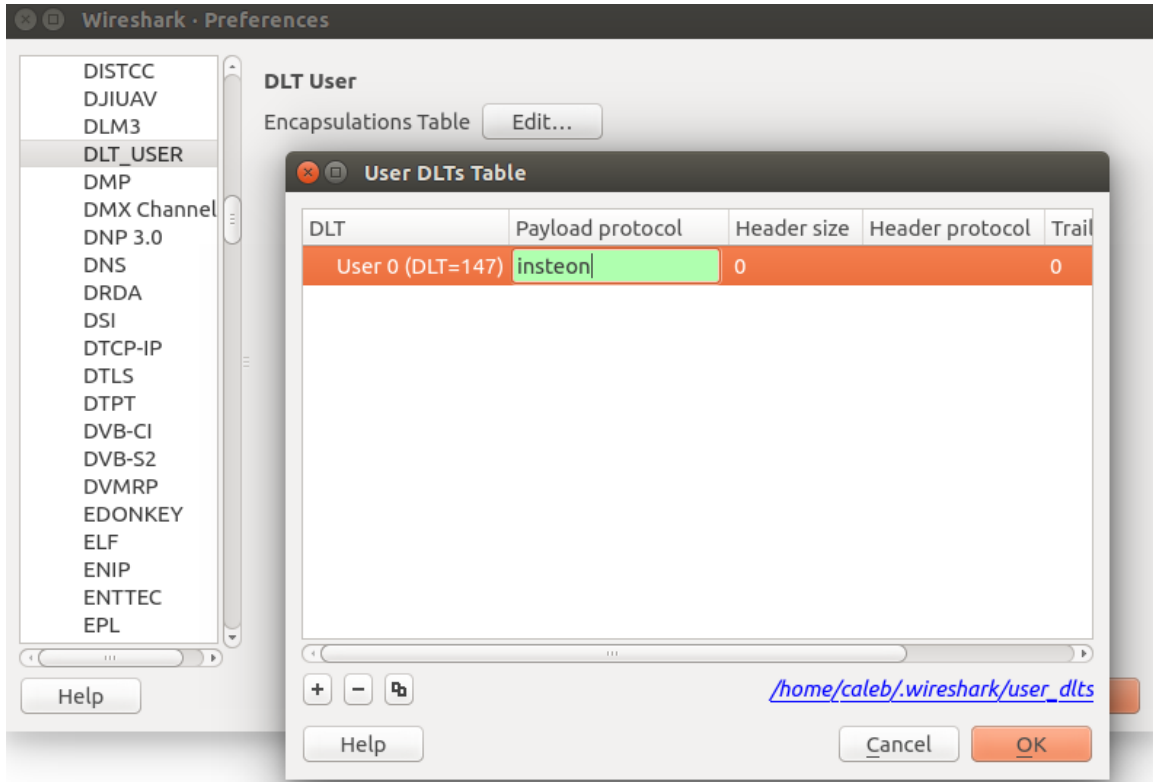


Figure 8. User DLTs edit menu.

3.6 Computer Configuration

All of the previously described components are attached to or installed on a high-performance computer. Table 5 outlines the hardware and software details of the computer used in this research.

Table 5. Computer configuration.

Dell Precision M4500 running Ubuntu 14.04 LTS		
4 processor cores	Linux kernel 3.19	bash 4.3.11
8 GB RAM	Python 2.7.3	Wireshark 2.0.4 with Lua 5.2
225 GB HD space	gcc 4.8	

3.7 INSTEON Equipment

Figure 9 illustrates INSTEON equipment purchased and used in this research. The equipment includes:

- 1 INSTEON Hub (model #: 2245-222)
- 2 INSTEON LEDs (model #: 2672-222)
- 1 INSTEON Thermostat (model #: 2441TH)
- 1 INSTEON Lamp Dimmer Module (model #: 2457D2) connected to a standard light bulb
- 1 INSTEON Lock Controller (model #: 2862-222) connected to a MiLocks door lock (model #: 51890N)

The INSTEON devices are networked and configured in a Pelican travel case for ease in portability and demonstration. Also visible in the Pelican case are an INSTEON Leak Sensor and INSTEON Open/Close Sensor. Both sensors are unused in this research. Further details about this equipment are found in Sections 4.4.1 and 4.5.1.

3.8 Honeypot Design Considerations

One goal of this research is to develop an IoT honeypot, a targetable and authentic decoy, to hide building automation systems. Shipley and Gooler’s previous software provides the basis for the honeypot due to their software’s demonstrated ability to receive and transmit INSTEON packets. Their software is extended to generate INSTEON commands and appropriately respond to incoming INSTEON commands, making the honeypot both targetable and authentic.

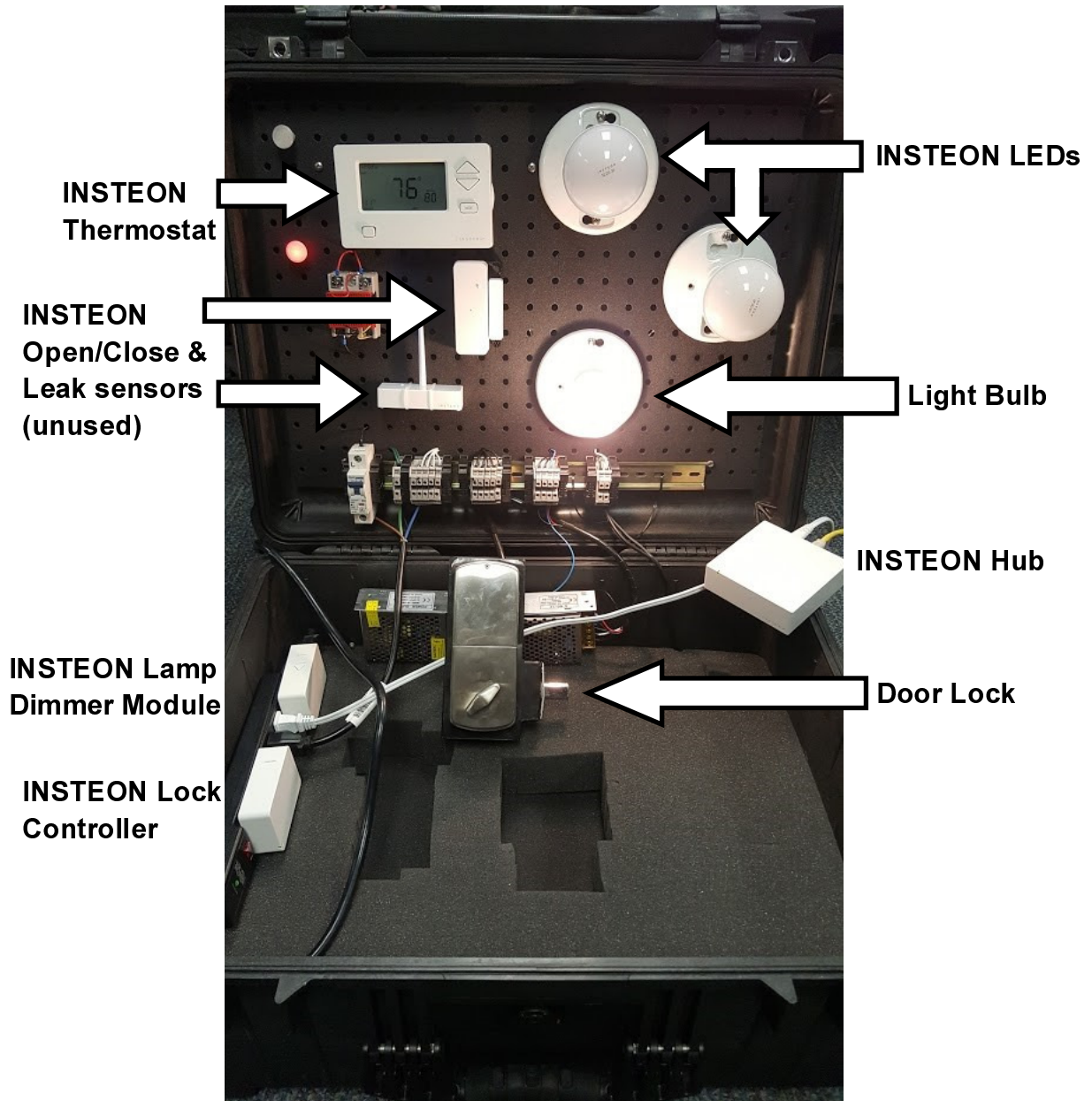


Figure 9. INSTEON equipment.

3.8.1 Targetability

A targetable honeypot should consist of multiple decoys to attract an attacker's attention. The decoys should also appear in relevant surroundings. A lone IoT light bulb is of little significance to an attacker when it is not connected to a larger building automation network. However, several connected IoT devices could convince

the attacker that the decoy is a legitimate building automation network. In the case of an INSTEON network, an INSTEON Hub, Thermostat, multiple lights and a Lock Controller present a network similar to an INSTEON network in a house thus posing as an attractive target to an attacker. The proposed honeypot implements the equipment discussed in Section 3.7.

Simulated network traffic is generated using a Python program that makes use of the previously described transmitter. This network traffic generation program receives a configuration file as input. The input file contains the packets to send, each one followed by a wait time (in seconds). Listing 6 provides an example input file to simulate network traffic.

```

1 OF : A1 A1 A1 : B1 B1 B1 : 19 02 # command to send
2 1 # wait 1 second
3 OF : C1 C1 C1 : B1 B1 B1 : 19 02 # this continues.
4 1
5 OF : D1 D1 D1 : B1 B1 B1 : 19 02
6 1
7 OF : E1 E1 E1 : B1 B1 B1 : 19 02
8 1

```

Listing 6. Honeypot configuration file example.

The honeypot is programmed to receive commands and autonomously respond. Using this method, the honeypot will simulate network traffic that is reasonably similar to network traffic generated by the real INSTEON network. The experiment testing this design can be found in Section 4.5.2.

3.8.2 Authenticity

Not only should the decoy be targetable, but the decoy should also be authentic enough that an attacker can interact with the virtual devices. INSTEON: Command Tables (see [6]) describes commands useful for interacting with the INSTEON devices, including device identification and commands used to modify the state of a device (e.g., turn a light on/off and change thermostat setting). The device identification request command is described further in Section 4.3.2. Successful authenticity consists

of the virtual devices responding to commands in a manner identical to how the genuine devices respond.

As discussed previously, honeypots typically portray one of two levels of interaction: (i) low interaction; or (ii) high interaction. The two levels of interaction describe the authenticity of the honeypot. When a honeypot emulates limited behavior, uses simple scripts and responds with static information, it is a low interaction honeypot. High interaction honeypots emulate the full functionality of the intended operating system or application, including logging and reporting capabilities.

The honeypot implemented in this research is a low interaction honeypot as it will respond using scripts and does not incorporate logging capabilities. However, the honeypot maintains device state. This allows the attacker to interact with honeypot devices that appear dynamic. For example, a light can either be on (to a given power-level) or off. When an attacker requests the status of a light, the device responds with its current state. The attacker may transmit commands to turn the lights on or off. The honeypot is designed to maintain virtual devices that act in a similar fashion. Specifically, this honeypot can mimic the INSTEON Hub, Lock Controller, Thermostat and lights, keeping track of each device's settings. Therefore, this honeypot is a low interaction honeypot, though its authenticity is increased because the honeypot maintains device state.

The honeypot is set up using a configuration file in the honeypot initialization. This configuration file defines multiple devices, each defined by device address, state information, device category, subcategory and firmware. Additionally, each device maintains a list of controlling addresses and a list of responding addresses. In this manner, each device maintains a piece of the overall network architecture.

Listing 7 is an example a honeypot input file. If this file were used, a basic INSTEON network would be created. The network would consist of two devices: (i)

a light and (ii) a hub and would be networked such that the hub would control the light.

```

1 Device
2 A1 A1 A1          # device address
3 01                # device category (Dimmable Light Control)
4 3A                # device subcategory
5 48                # device firmware version
6 00                # device state
7 isresponderyes    # this device is a responder
8 1                 # this device responds to 1 device
9 B1 B1 B1          # this device responds to device address B1 B1 B1
10 iscontrollerno   # this device is not a controller
11 Device
12 B1 B1 B1          # device address
13 03                # device category (Network Bridge)
14 33                # device subcategory
15 25                # device firmware version
16 00                # device state
17 isresponderno    # this device is not a responder
18 iscontrolleryes  # this device is a controller
19 1                 # this device controllers 1 device
20 A1 A1 A1          # this device controlls device address A1 A1 A1

```

Listing 7. Honeypot configuration file example.

3.9 Summary

This chapter presents the full system design and architecture used for this research. By using the aforementioned system, Python-based tools can be quickly developed for testing INSTEON networks and devices for vulnerabilities using the YARD Stick One. The experiments following rely on the system described in this chapter to accomplish the goals presented in Section 1.3.

IV. Methodology

4.1 Overview

This research focuses on improving the previous security tools and defending building automation systems by creating a honeypot. Section 4.2 discusses the problem statement, including the goals and the approaches taken to accomplish them. Section 4.3 describes two studies used to determine the metrics by which the goals will be measured. Sections 4.4 and 4.5 describe the details of the two experiments carried out in this research.

4.2 Problem Statement

4.2.1 Goals

This research conducts a total of two experiments to address the two primary research goals from Section 1.3.

1. Develop or improve methods for transmitting and receiving INSTEON network communication using a Software-Defined Radio (SDR).
2. Develop a virtual honeypot to emulate INSTEON devices and evaluate the targetability and authenticity of the honeypot to protect building automation systems.

4.2.2 Approach

The aforementioned goals require separate approaches which are discussed in the following sections.

4.2.2.1 Goal 1

The current body of INSTEON research is limited to a transmitter and a receiver (see [19]) as described in Section 2.3. This goal represents discovering the packet capture rate of the previous work and modifying the YARD Stick One settings to achieve a better capture rate. The baseline metric for this goal is established in a pilot study described in Section 4.3.1. The test environment and experiment design are discussed in detail in Section 4.4.

4.2.2.2 Goal 2

This goal represents evaluating the targetability and authenticity of the INSTEON building automation honeypot described in Section 3.8. As INSTEON is a relatively un-researched protocol, an enumeration technique must be established. This technique will be used to specifically evaluate the authenticity of the honeypot. The enumeration technique is discussed in Section 4.3.2. The test environment and experiment design used to evaluate this goal is described in Section 4.5.

4.3 Pilot Studies

Two pilot studies are accomplished for this research. The first pilot study establishes the baseline for the packet capture rate by capturing packets between the INSTEON Hub and an INSTEON LED. The second pilot study verifies and describes a command necessary for querying a device for its identity information, thereby enumerating the device. A device identity is necessary to build a honeypot that mimics the original device.

4.3.1 Determining Packet Count

In conjunction with research by Chris Talbot et al. (see [21]), a pilot study is conducted to determine the packet count between the INSTEON Hub and an INSTEON LED. The pilot study uses a significantly more robust reception method, but a similar test environment as described in Section 4.4. It is determined that for every “on” command the INSTEON Hub and LED exchange eight packets. The same is true for every “off” command. Turning the LED on and off 20 times therefore produces a total of 320 packets. This establishes the baseline number for the packet capture success rate. The results of this pilot study are illustrated in Figure 10.

No.	Flags	Destination	Source	Cmd1	Cmd2	CRC	Info
1	05	33 D3 32	DE C2 33	11	FF	D6	light on
2	01	33 D3 32	DE C2 33	11	FF	12	light on (repeat)
3	25	DE C2 33	33 D3 32	11	FF	86	ACK light on
4	21	DE C2 33	33 D3 32	11	FF	42	ACK light on (repeat)
5	05	33 D3 32	DE C2 33	19	02	53	status request
6	01	33 D3 32	DE C2 33	19	02	97	status request (repeat)
7	25	DE C2 33	33 D3 32	19	FF	8E	ACK status response
8	21	DE C2 33	33 D3 32	19	FF	4A	ACK status response (repeat)
9	05	33 D3 32	DE C2 33	13	00	3B	Light off
10	01	33 D3 32	DE C2 33	13	00	FF	Light off (repeat)
11	25	DE C2 33	33 D3 32	13	00	6B	ACK light off
12	21	DE C2 33	33 D3 32	13	00	AF	ACK light off (repeat)
13	05	33 D3 32	DE C2 33	19	02	53	status request
14	01	33 D3 32	DE C2 33	19	02	97	status request (repeat)
15	25	DE C2 33	33 D3 32	19	00	61	ACK status response
16	21	DE C2 33	33 D3 32	19	00	A5	ACK status response (repeat)

Figure 10. Results of packet capture rate pilot study.

4.3.2 Enumerating Devices

The INSTEON command tables document (see [6]) describes a command to request the identity of a device. There are three messages transmitted to issue and respond to a device identity request. The three commands are illustrated in Figure 11. First the control device issues a direct command to request the identity of the responding device. The responder transmits acknowledgment of the command and follows this by transmitting a broadcast message.

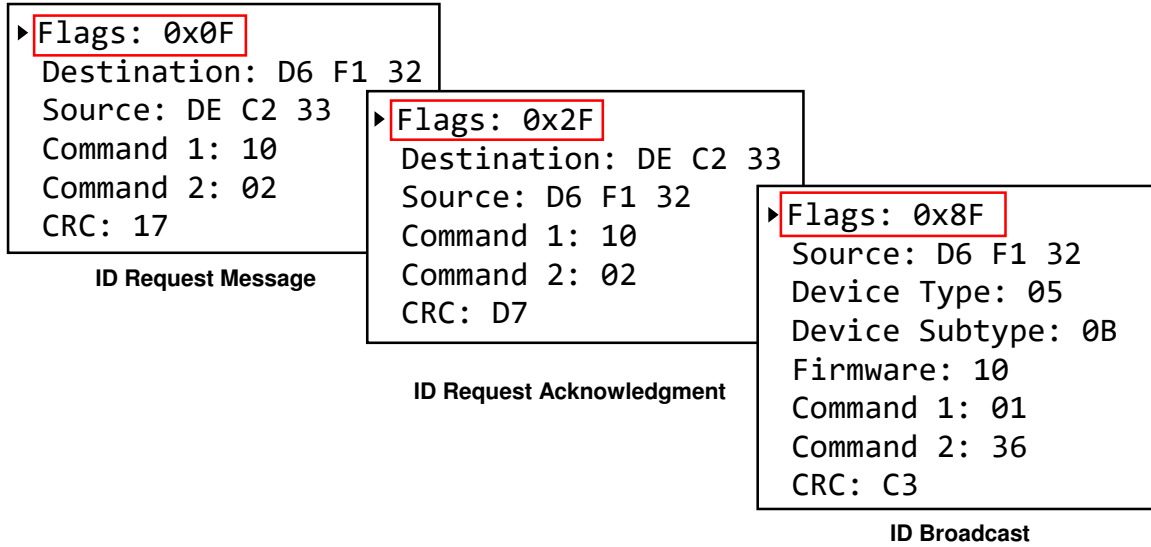


Figure 11. Device identity request, acknowledgment and broadcast messages.

In the INSTEON protocol, broadcast messages are typically used in the device pairing process, but not during normal communication. These messages contain the device category, subcategory and firmware version of the source device. The broadcast message packet structure is illustrated in Figure 12.

1 Byte	3 Bytes	1 Byte	1 Byte	1 Byte	2 Bytes	1 Byte
Flags	Source Address	Device Category	Device Subcategory	Firmware Version	Commands	CRC

Figure 12. Broadcast message packet structure [5].

The pilot study confirms the INSTEON devices used for this research and their respective categories match the device category documentation (see [7]). The device category therefore becomes the basis for device enumeration. The device category is enough information to tell an attacker about what the device physically controls (e.g., lighting control, climate control and access control). This information is used to create honeypot devices with matching device identities which provides the basis for an authentic INSTEON honeypot.

4.4 Packet Capture Rate Test

This test has two objectives. The primary objective is to determine the best settings for achieving the maximum packet capture success rate using the YARD Stick One. The secondary objective is to validate (or refute) Shipley and Gooler’s claims on the INSTEON wireless Radio Frequency (RF) protocol specifications, specifically the center frequency, FSK shift value and symbol rate. The following sections describe the test environment, experiment design, system under test (SUT), control variables and assumptions for the packet capture success rate test.

4.4.1 Test Environment

Figure 13 illustrates the network configuration for this test. Two INSTEON devices are used in this experiment: (i) the INSTEON Hub and (ii) an INSTEON LED. The INSTEON for Hub application is operated on a Windows Surface 3 and is connected via the Internet to the INSTEON Hub. The Hub and the INSTEON LED are also networked so the Hub controls the LED. A YARD Stick One is connected to a laptop running Ubuntu and the software defining the configurations for the SDR.

4.4.2 Experiment Design

This experiment uses the findings from the pilot study in Section 4.3.1 as the baseline for the packet capture success rate. The test procedure is to turn the LED on and off 20 times to generate 320 packet transmissions while the receiver is listening for packets. Each received or dropped packet is viewed as a separate, binary trial (either successful or failed reception). This experiment produces data that is a binary distribution. The 320 individual results are pooled and viewed as one test. The number of received packets divided by the number of total packets transmitted determines the mean packet capture success rate.

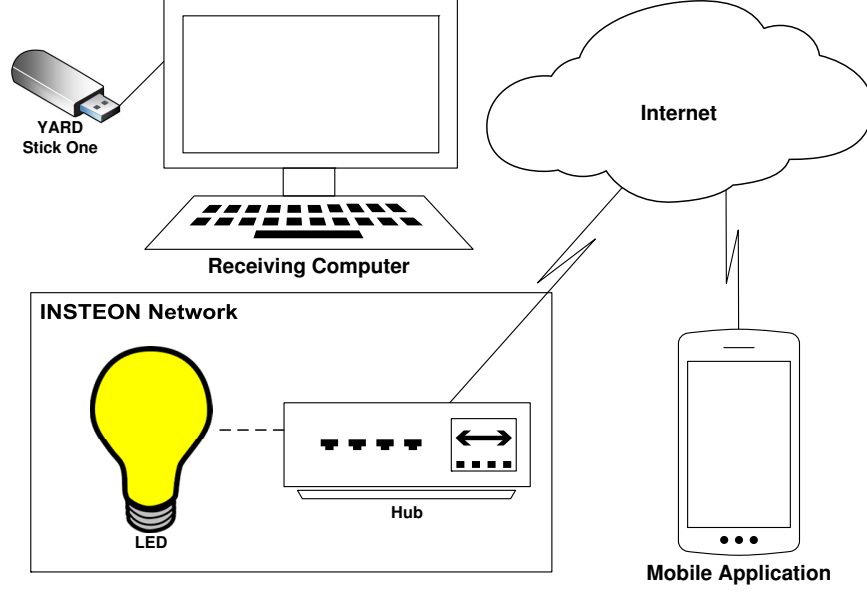


Figure 13. Packet capture rate test environment.

4.4.3 System Under Test and Component Under Test

As shown in Figure 14, the system under test (SUT) is the reception program. The primary component under test (CUT) is the YARD Stick One. The metric for evaluating the settings is the packet capture rate. The workload into the system is the signal data produced from turning the INSTEON LED on and off.

4.4.4 Control Variables

There are three control variables for this experiment: (i) center frequency; (ii) frequency shift; and (iii) symbol rate. The experiment starts by varying center frequency between the three values in Table 4 (914.95 MHz, 914.975 MHz and 915 MHz). Frequency shift and symbol rate are held constant at the start to Shipley and Gooler's specifications. The best center frequency is used in the next trial and so forth. If there is no statistical difference between a given trial, Shipley and Gooler's settings are used for the next trial. It should be noted that the symbol rate of 76.8 KBaud is assumed incorrect in this experiment because it is vastly different than both the

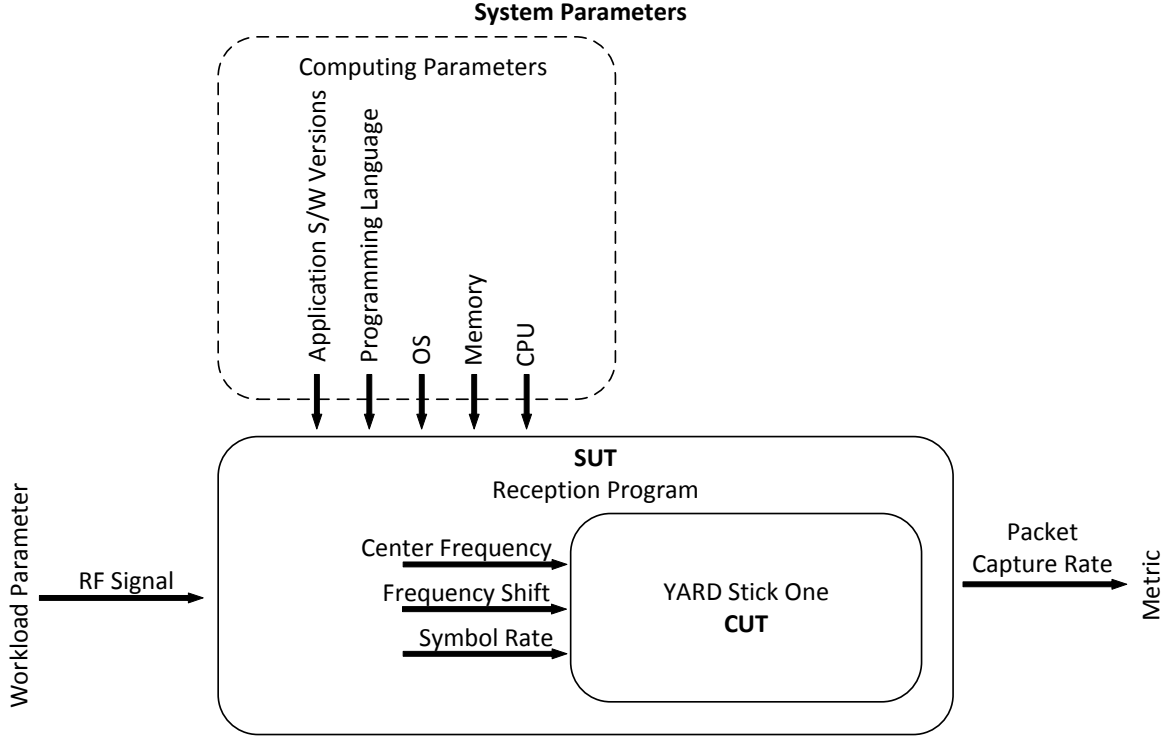


Figure 14. System under test and component under test.

specification in [5] and [19]. As such, the decision is made to vary symbol rate between 9 KBaud, 9.125 KBaud and 9.250 KBaud. This process results in seven radio configurations for this experiment. Table 6 shows the control variable settings for each trial.

Table 6. Packet capture rate trials.

Trial #	Center Frequency	Shift	Symbol Rate
1	914.950 MHz	150 KHz	9.125 KBaud
2	914.975 MHz	150 KHz	9.125 KBaud
3	915.000 MHz	150 KHz	9.125 KBaud
4	915.000 MHz	64 KHz	9.125 KBaud
5	915.000 MHz	200 KHz	9.125 KBaud
6	915.000 MHz	150 KHz	9.000 KBaud
7	915.000 MHz	150 KHz	9.250 KBaud

4.4.5 Assumptions

There are three assumptions made for this experiment which are listed below.

1. It is assumed that the rest of Shipley and Gooler's software, including the Manchester decoding capability, parser, Cyclic Redundancy Check (CRC) calculation method, packet structure and other aspects apart from the radio configuration, are correct.
2. It is assumed that the YARD Stick One is not a perfect reception tool. Previous experience with the YARD Stick One suggests that an approximate maximum 80% packet capture rate is typical. As one objective of this experiment is to find the best configuration for the YARD Stick One, 80% packet capture success would likely indicate the best possible configuration.
3. It is assumed that the distance between the INSTEON equipment and the receiver is within standard transmission range. The equipment is kept less than five feet apart and remains constant throughout this experiment. This experiment is strictly concerned with the radio configuration settings and will not be used to determine any distance measuring or packet loss based on distance.

4.5 Honeypot Functional Test

The evaluation approach of the honeypot consists of a functional test to evaluate authenticity and targetability. Simulated network traffic and devices are based on the results from the pilot study in Section 4.3.2 and other observations made over the course of this research. The following sections describe the test environment and experiment design.

4.5.1 Test Environment

The test environment is illustrated in Figure 15. The INSTEON equipment used in this test is the INSTEON Hub, Thermostat, two LEDs, a Lamp Dimmer Module with traditional light bulb and one Lock Controller with door lock. The honeypot is configured to mimic these devices and is hosted on the high-performance computer (see Table 5). A single honeypot host controls the honeypot networks.

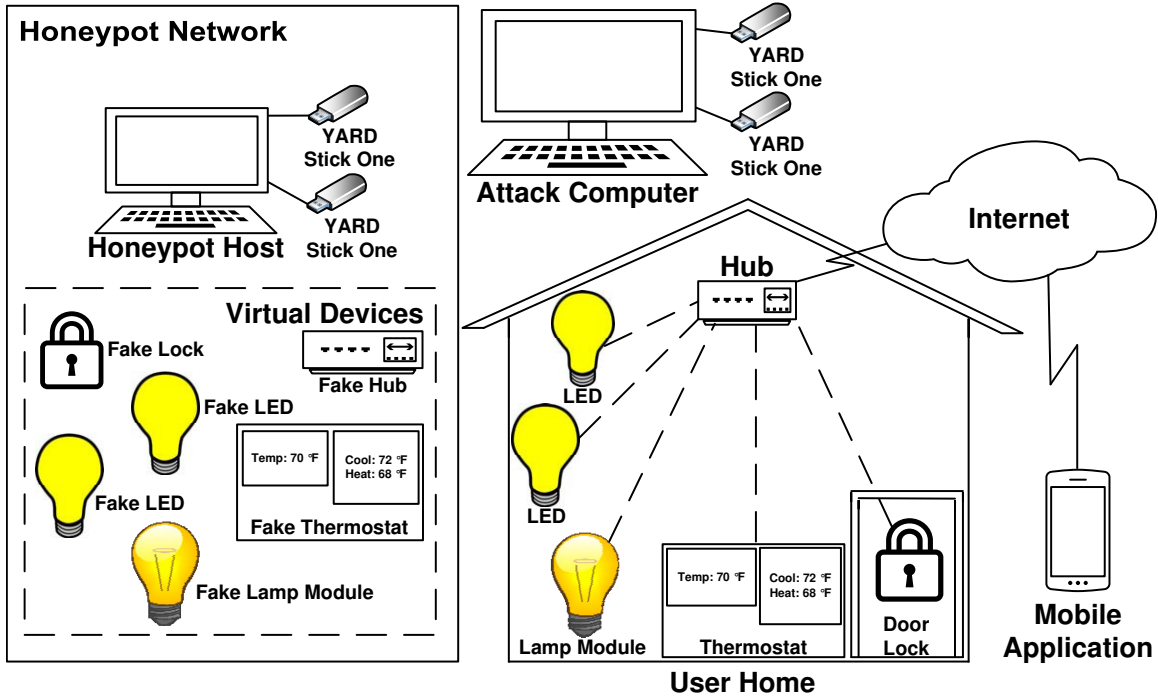


Figure 15. Functional test environment.

4.5.2 Experiment Design

The test procedure consists of the user checking the status of the devices in the INSTEON for Hub application, thereby generating INSTEON network traffic. Figure 16 shows Wireshark output of traffic generated when the INSTEON Hub requests the status of a device. Command 2 of the status response message in Figure 16 indicates

that device E7 5C 2F is at power-level 00 (the device is turned off). When the user logs into the INSTEON for Hub application, the Hub requests the status of all non-battery-powered devices on the INSTEON network.

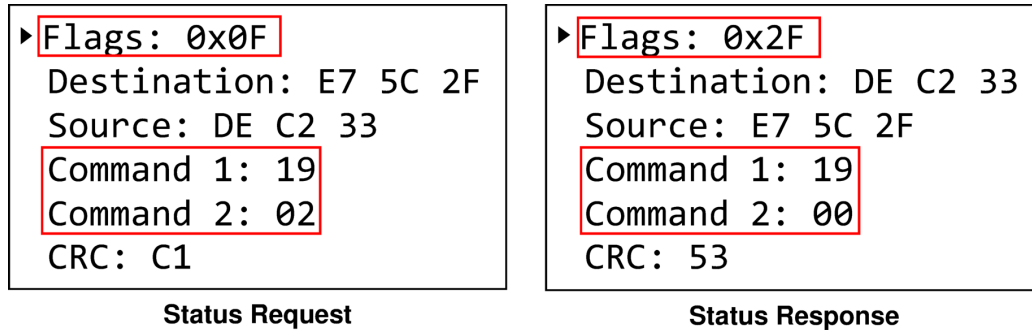


Figure 16. Status request and response messages.

Based on this network traffic, the attacker will query the devices for their identity using the command described in Figure 11. The attacker will create a full network map by spoofing commands between all known devices.

In order to evaluate the honeypot, the genuine INSTEON network is enumerated using the method described above. The results of the INSTEON network enumeration scan are shown in Figure 17.

An interesting point in this figure is that the thermostat (ID: D6 F1 32) responds to every other INSTEON device. This not typical for INSTEON devices and is a flaw in the thermostat's logic. Based on this information the decision was made to design the honeypot thermostat to have the same flaw.

The test includes creating a honeypot with one and five distinct virtual networks that each mimic the true INSTEON network. If all of the honeypot networks mimic the genuine network, the honeypot networks will be targetable and authentic decoys which will hide the genuine network from a would-be attacker.

```

D6 F1 32 is a: Climate Control
DE C2 33 is a: Network Bridge
33 D3 32 is a: Dimmable Lighting Control
56 E2 3E is a: Access Control
95 A3 2E is a: Dimmable Lighting Control
E7 5C 2F is a: Dimmable Lighting Control

Controllers are:
DE C2 33 controls: ['D6 F1 32', '33 D3 32', '56 E2 3E', '95 A3 2E', 'E7 5C 2F']
33 D3 32 controls: ['D6 F1 32']
56 E2 3E controls: ['DE C2 33', 'D6 F1 32']
95 A3 2E controls: ['DE C2 33', 'D6 F1 32']
E7 5C 2F controls: ['D6 F1 32']

Responders are:
D6 F1 32 responds to: ['DE C2 33', '33 D3 32', '95 A3 2E', '56 E2 3E', 'E7 5C 2F']
DE C2 33 responds to: ['95 A3 2E', '56 E2 3E']
33 D3 32 responds to: ['DE C2 33']
56 E2 3E responds to: ['DE C2 33']
95 A3 2E responds to: ['DE C2 33']
E7 5C 2F responds to: ['DE C2 33']

```

Figure 17. INSTEON network baseline enumeration.

4.6 Summary

This chapter presents the methodologies used to test the packet capture rate and develop an INSTEON honeypot. The Python software described in Section 3.4 is used to transmit and receive INSTEON commands to accomplish the aforementioned research goals.

V. Results and Analysis

5.1 Overview

This chapter details the results and provides analysis of the experiments conducted in Chapter IV. Both experiments were completed successfully without complications or discernible errors. Section 5.2 discusses the results of the packet capture rate test, and Section 5.3 discusses the results of the honeypot functional test.

5.2 Packet Capture Rate Test

A 99% confidence interval plot is created for each pooled trial with comparison plots to portray the results. A General Linear Model (GLM) for the binary logistical distribution is also produced. The results of varying each control setting are discussed in the following subsections.

5.2.1 Varying Center Frequency

Figure 18 portrays the impact of varying the center frequency while keeping Frequency Shift Keying (FSK) shift and symbol rate constant at 150 KHz and 9.125 KBaud, respectively. 915 MHz produces the best packet capture rate of approximately 65%. 914.95 MHz, which is the center frequency value used in Shipley and Gooler's YARD Stick One settings, definitively produces an inferior packet capture rate near 40% when compared to the other center frequencies.

5.2.2 Varying Frequency Shift

Figure 19 shows the confidence intervals when center frequency is held constant at 915 MHz and symbol rate is constant at 9.125 KBaud, while FSK shift is varied between 200 KHz, 150 KHz and 64 KHz. There is no statistical difference on packet

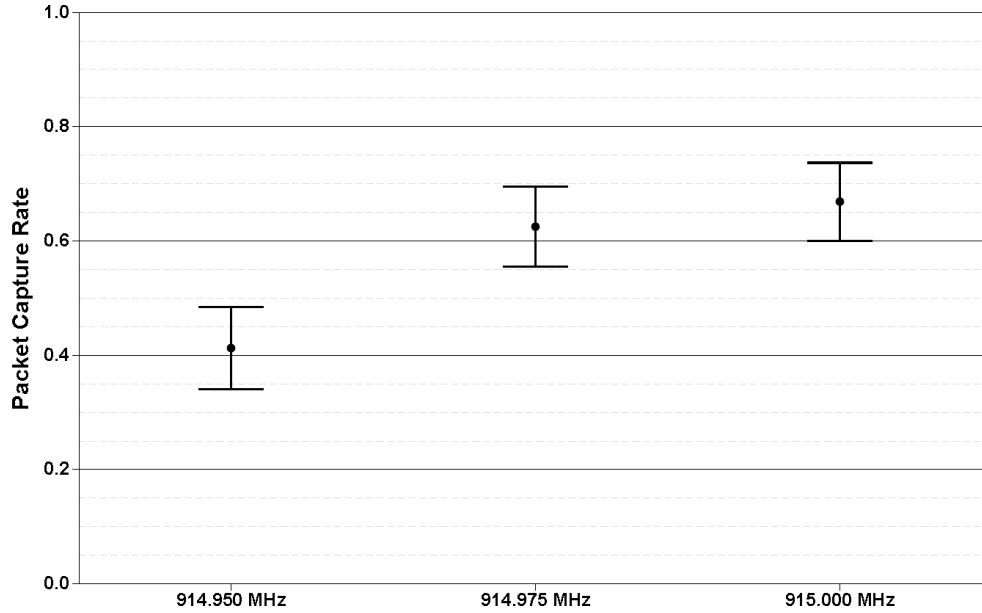


Figure 18. Packet capture results for varying center frequencies.

capture rate when varying the FSK shift. Using FSK shift of 64 KHz produces the best capture rate at almost 75% however this experiment does not produce a statistically significant result. The experiment continues with FSK shift set to 150 KHz (the claimed FSK shift from Shipley and Gooler).

5.2.3 Varying Symbol Rate

The final test varies symbol rates between 9 KBaud, 9.125 KBaud and 9.25 KBaud while maintaining center frequency at 915 MHz and FSK shift at 150 KHz. The confidence intervals in Figure 20 show no distinguishable difference between the three symbol rates.

5.2.4 All Trials

To verify the findings, a graph containing the confidence intervals with each trial is made and a GLM for the binary linear regression is created. Figure 21 shows the confidence interval plots for all trials. The first trial (center frequency 914.95 MHz,

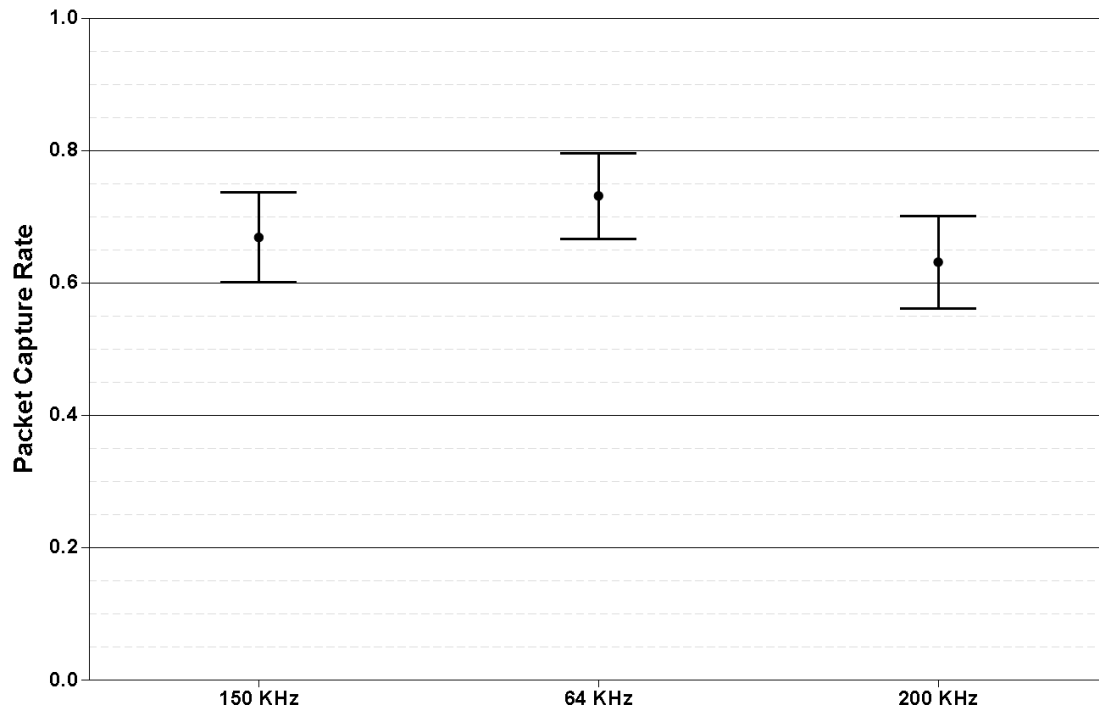


Figure 19. Packet capture results for varying frequency shifts.

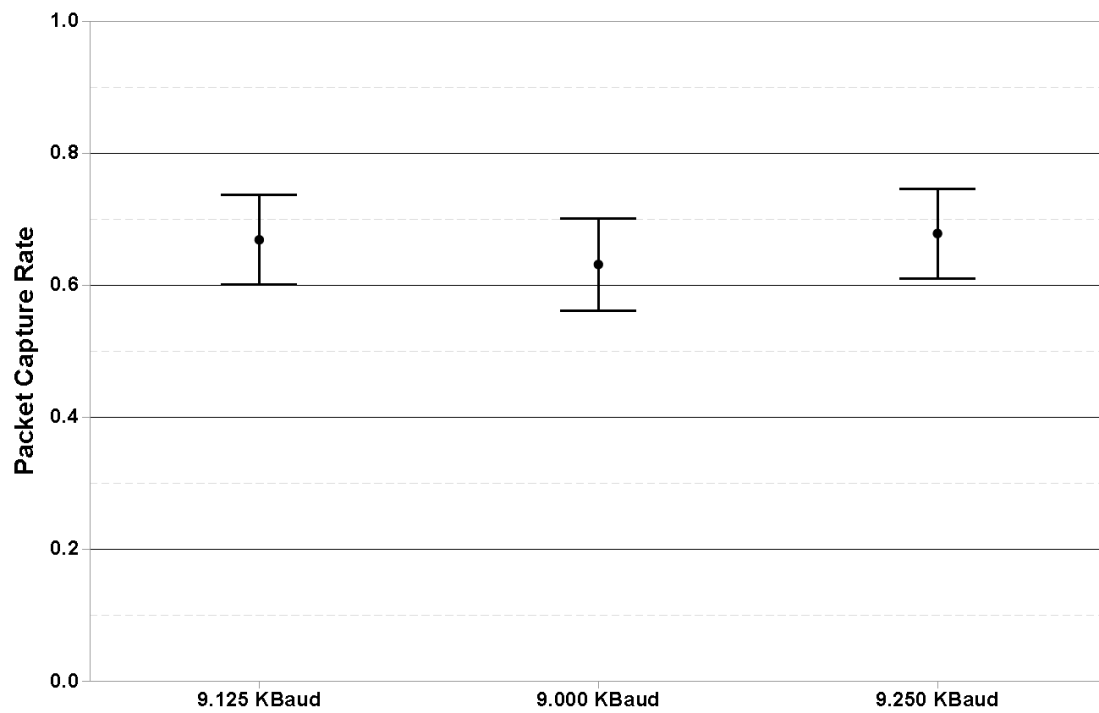


Figure 20. Packet capture results for varying symbol rate.

150 KHz FSK shift and 9.125 KBaud symbol rate) produces an inferior packet capture rate. The best capture rate is nearly 75% and occurs when the YARD Stick One is set to 915 MHz center frequency, 64 KHz FSK shift and 9.125 KBaud. The results are unable to confirm or refute Shipley and Gooler's claims on INSTEON's RF settings.

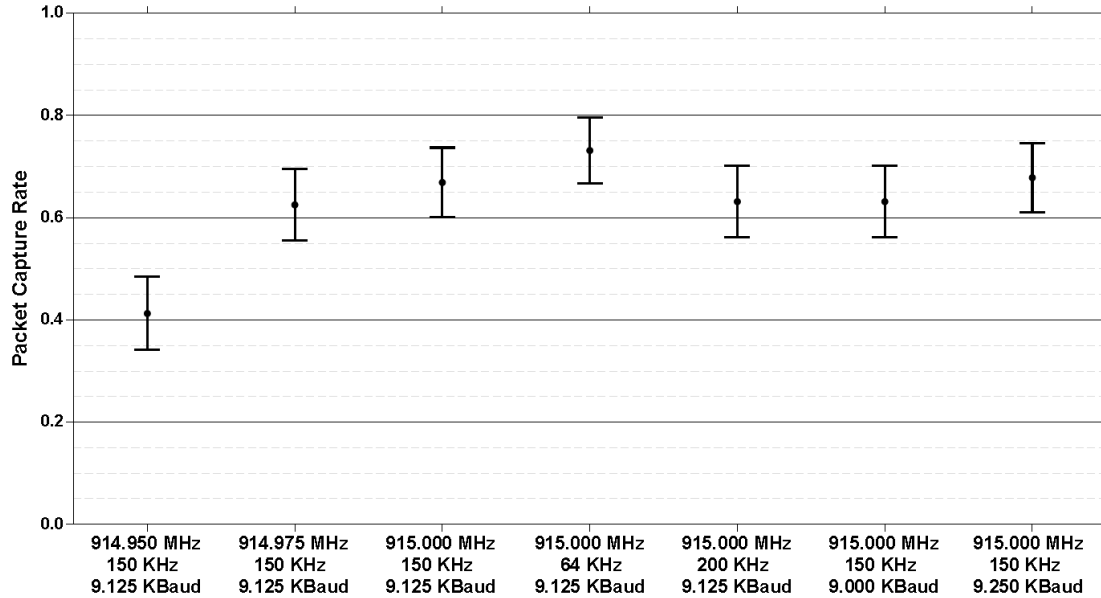


Figure 21. All trials results.

The GLM confirms these results and are shown in Figure 22. This analysis verifies statistically that frequency makes a significant impact on packet capture rate. No other variables are statistically significant as all other factors produce z-values with probability of < 0.05 .

Coefficients:

	Estimate	Std. Error	z value	Pr(> z)	
(Intercept)	-0.43267	0.31192	-1.387	0.165	
Frequency 914.975 MHz	0.86447	0.16195	5.338	9.41e-08	***
Frequency 915 MHz	1.05618	0.16432	6.427	1.30e-10	***
Shift 150 KHz	-0.08592	0.16927	-1.508	0.612	
Shift 200 KHz	-0.25087	0.16724	-1.500	0.134	
Shift 64 KHz	0.21252	0.17449	1.218	0.223	
Symbol.Rate 9.125 KBaud	0.16494	0.16593	0.994	0.320	
Symbol.Rate 9.250 KBaud	0.20759	0.16656	1.246	0.213	

Figure 22. General Linear Model analysis.

5.3 Honeypot Functional Test

There are two portions to this functional test. First, authenticity is determined by mapping and investigating the honeypot network individually and comparing it to the genuine INSTEON network. For the second test, targetability is measured by determining if the honeypot devices present themselves as genuine devices.

The authenticity is found by enumerating one honeypot network and comparing it to the genuine INSTEON network. The honeypot network enumeration is found in Figure 23. Table 7 summarizes the true INSTEON network enumeration compared to the honeypot enumeration. The honeypot devices matched the true INSTEON devices in device category information and when compared against the full network map in Figure 17. Therefore, the honeypot network accurately mimics the genuine INSTEON network causing an attacker to think the virtual devices are actually authentic.

```
C1 C1 C1 is a: Climate Control
B1 B1 B1 is a: Network Bridge
E1 E1 E1 is a: Dimmable Lighting Control
F1 F1 F1 is a: Access Control
A1 A1 A1 is a: Dimmable Lighting Control
D1 D1 D1 is a: Dimmable Lighting Control

Controllers are:
B1 B1 B1 controls: ['C1 C1 C1', 'E1 E1 E1', 'F1 F1 F1', 'A1 A1 A1', 'D1 D1 D1']
E1 E1 E1 controls: ['C1 C1 C1']
F1 F1 F1 controls: ['B1 B1 B1', 'C1 C1 C1']
A1 A1 A1 controls: ['B1 B1 B1', 'C1 C1 C1']
D1 D1 D1 controls: ['C1 C1 C1']

Responders are:
C1 C1 C1 responds to: ['B1 B1 B1', 'E1 E1 E1', 'A1 A1 A1', 'F1 F1 F1', 'D1 D1 D1']
B1 B1 B1 responds to: ['A1 A1 A1', 'F1 F1 F1']
E1 E1 E1 responds to: ['B1 B1 B1']
F1 F1 F1 responds to: ['B1 B1 B1']
A1 A1 A1 responds to: ['B1 B1 B1']
D1 D1 D1 responds to: ['B1 B1 B1']
```

Figure 23. Honeypot baseline enumeration.

Table 7. INSTEON network compared to honeypot network.

INSTEON ID	Honeypot ID	Device Category	Category Matched
DE C2 33	B1 B1 B1	Network Bridge	✓
D6 F1 32	C1 C1 C1	Climate Control	✓
E7 5C 2F	D1 D1 D1	Dimmable Light Control	✓
33 D3 32	E1 E1 E1	Dimmable Light Control	✓
95 A3 2E	A1 A1 A1	Dimmable Light Control	✓
56 E2 3E	F1 F1 F1	Access Control	✓

The targetability of the honeypot is determined by presenting distinct honeypot networks together with the genuine network. For a honeypot to be effective, not only does the attacker need to believe that the honeypot devices are authentic, but the attacker also needs to believe the virtual devices exist and are available to attack. Figure 24 shows the representation of two networks (one true INSTEON network and one honeypot network) running simultaneously. The honeypot generates network traffic that closely mimics a user checking the status of the INSTEON Thermostat, LEDs, Lamp Dimmer Module and the Lock Controller. This simulated traffic therefore presents a targetable, distinct honeypot network to an attacker. The honeypot and the actual INSTEON network appear to be identical thus deceiving the attacker to believe multiple distinct INSTEON networks exist.

Figure 25 shows the representation of six networks (one true INSTEON network and five honeypot networks) running simultaneously. In each case the simulated network traffic presented the honeypot devices as functionally identical to the INSTEON network devices. This helps to “hide” the true network with the honeypot networks. Instead of the lone genuine network being targetable, there are now six distinct, targetable networks. This protects the genuine network by decreasing the probability of attack to 1/6 without the attacker performing deeper analysis.

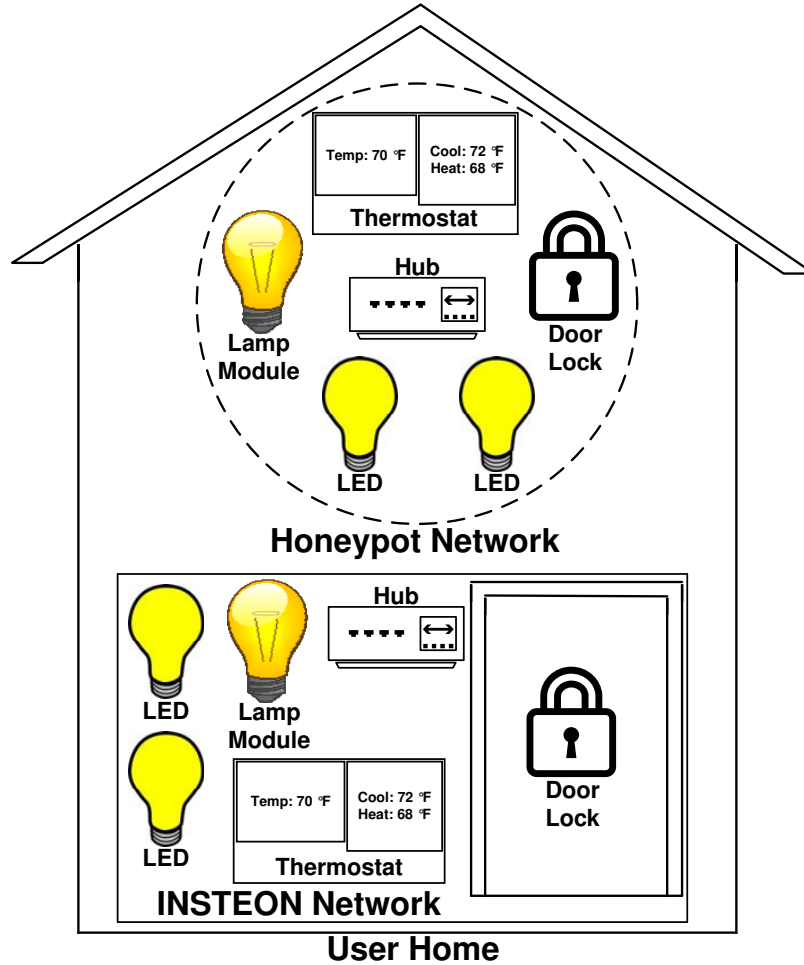


Figure 24. Depiction of an INSTEON network and a honeypot network.

5.4 Summary

This chapter presents and analyzes the results of two separate experiments. Section 5.2 describes the results of testing the packet capture rate to determine the best RF settings for the YARD Stick One. Section 5.3 describe the evaluation of an authentic and targetable INSTEON honeypot which distracts would-be attackers and protect the genuine INSTEON network.

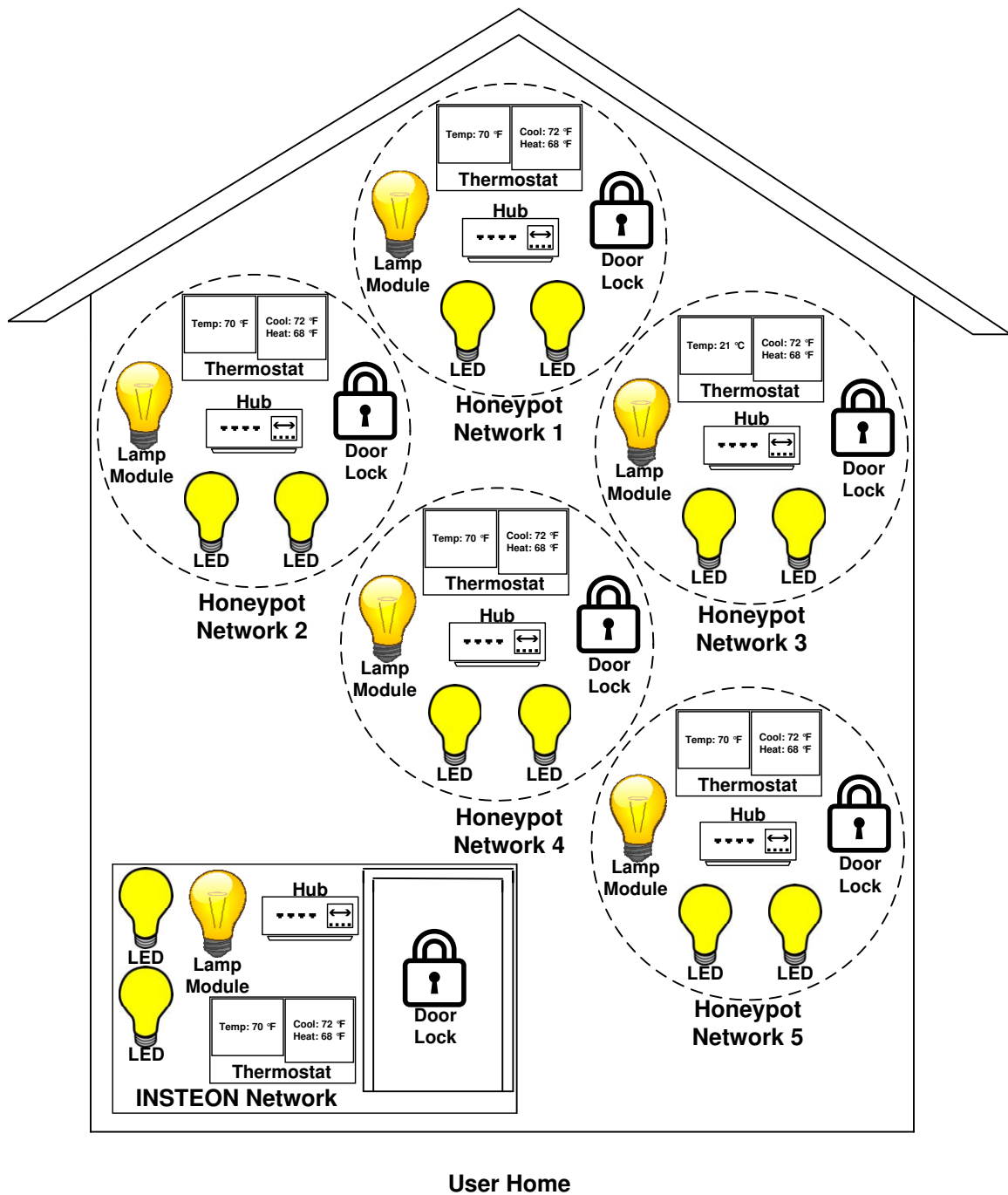


Figure 25. Depiction of an INSTEON network and five honeypot networks.

VI. Conclusion

6.1 Overview

This chapter summarizes the overall conclusions of the research. Section 6.2 presents conclusions based on the results and analysis from Chapter V. Section 6.3 discusses the significance of this research, and finally Section 6.4 presents ideas for future research.

6.2 Research Conclusions

The results of the experiments conducted in this research are discussed in the following sections.

6.2.1 Packet Capture Rate Test

This test confirmed the limitations of the previous research. Shipley and Gooler's tool captured packets with approximately 40% success. The modifications made to their software improved the packet capture rate to nearly 75%. The primary objective of this test was to find the best RF settings for the YARD Stick One. Based on previous experience, a packet capture rate of 75% likely indicates the optimal RF settings for INSTEON packet capture success. The secondary objective of this test was to validate (or refute) Shipley and Gooler's previous work. The RF settings programmed in their software produced a statistically-inferior packet capture rate. Tuning the center frequency to be 915 MHz proved to be the contributing factor to increasing the packet capture rate when using the YARD Stick One.

6.2.2 Honeypot Functional Test

The functional test demonstrated that the honeypot authentically mimicked INSTEON devices when the devices were enumerated. The test used the INSTEON device category information as the basis for authenticity. Further, the test demonstrated that the honeypot produced targetable decoys by autonomously generating network traffic that imitated genuine INSTEON network traffic. The resulting decoys are therefore authentic and targetable, decreasing the probability that an attacker will find and target the true INSTEON network.

6.3 Research Significance

6.3.1 Technical Contributions

The research discussed here increases the packet capture rate for the INSTEON protocol using the YARD Stick One SDR. Additionally, the research developed a honeypot for the INSTEON protocol to help protect home and building automation. This research contributes to the overall understanding of the INSTEON communications protocol and the defense of building automation systems.

6.3.2 Functional Contribution

Security and safety will become tantamount as building automation technologies and more IoT devices connect to the Internet. The fictional family in the introduction should not have to worry about their house and the security of their belongings since they have invested in a home automation system. Device developers and manufacturers need to incorporate security principles throughout the design and production of building automation devices. This research contributes to consumer safety and security by protecting building automation systems through the use of targetable and

authentic INSTEON honeypot decoys.

6.4 Future Work

6.4.1 Develop Additional Devices

The honeypot presented is limited in the type of devices it can host. Currently it can mimic dimmable lights, access controllers, hubs and thermostats. A typical user will likely have these basic devices, but may have others. Some other device types include switched lighting control (e.g., in-wall light switches and dimmers) and security/safety sensors (e.g., door open/close, motion and leak sensors). These additional devices need to be investigated and programmed into the honeypot.

6.4.2 Performance Test

Additionally, the performance of this honeypot was not measured in this experiment. An experiment measuring packet response times could be performed to discover how many honeypot devices could be instantiated before the honeypot experiences noticeable or measurable delay.

6.4.3 Reporting and Logging Capabilities

Finally, in its current state, this honeypot does not log anomalies or report any malicious network traffic to the user. Additional reporting and logging capabilities need to be designed and programmed to make the honeypot a more valuable defensive tool. A simple reporting capability would be an email server installed on the honeypot host that emails the user when a trigger event occurs.

6.5 Summary

This chapter presents the conclusions and impact of the INSTEON honeypot research. To conclude, several recommendations for future work are provided which build upon and extend the honeypot developed in this research.

Appendix A. Code

A.1 readme.txt

Insteon Tools

These tools are used to 1) listen for Insteon devices and 2) identify and label any devices based on their device category (if possible) and run the Insteon honeypot.

Installation instructions:

- 1) Ensure a working RfCat / Yard Stick One.
See instructions here: <http://forum.universal-devices.com/topic/17474-how-to-receive-send-log-spoof-all-insteon-rf-traffic/>
Or here: <https://bitbucket.org/atlas0fd00m/rfcat/overview>
- 2) Ensure Wireshark is latest version (2.0.4.) See Wireshark -> Help -> About Wireshark. Must be compiled with Lua as well. If using an not-quite-updated Ubuntu distro (like 14.04 LTS), follow these instructions: <http://ubuntuhandbook.org/index.php/2015/11/install-wireshark-2-0-in-ubuntu-15-10-wily/>
- 3) Now's a good time to download these files. I recommend putting the Insteon-dissector.lua in the /usr/share/wireshark directory.
- 4) Find Wireshark's original init.lua file (link in /usr/share/wireshark/ -> /etc/wireshark/init.lua)
Either: a) replace it with this init.lua file here or b) add a line at the very end to dofile(<path to Insteon-dissector.lua>).
- 5) Restart Wireshark to ensure no issue with the init.lua and Insteon-dissector configs. Then in Wireshark go to Edit->Preferences->Protocols->DLT_USER->Edit. Click +, Select User 0 (DLT=147), type "Insteon" in the payload protocol column. If everything has gone well, this box should be green. Click Ok. Ok again.

Running Instructions:

- 1) Receiving Packets
 - a) For off-line mode, run:
 - ./instelisten.py -o filename
 - The end product is a filename.pcap with any Insteon traffic that was collected.
 - 2 Additional files are created, filename.raw and filename.insteondump.
 - filename.insteondump can be put into ./insteonanalyzer -i filename.insteondump for network "map" analysis on what was collected.
 - b) For live mode:
 - mkfifo pipename
 - wireshark -i pipename and start listening on the pipe once wireshark starts.
 - run ./instelisten.py -l -p pipename (run using either sudo or root)
 - Wait for some traffic to ensure the insteon.devices file gets populated.
- 2) Enumerating Devices
 - In a second terminal, run: ./insteonscanner.py -i 1 (again, run using either sudo or root. This requires a 2nd Yard Stick to be plugged in). This program reads device IDs saved in insteon.devices file, then spoofs ID request from each devices to every other device. ID request responses contain the device category from the responder.
- 3) Insteon Honeypot
 - The honeypot uses 2 Yard Sticks.
 - run ./insteonhoney.py -iL X -iS Y -f filename -- where X is the interface for the listening YS1, Y is the interface for the sending YS1 and filename is the configuration file which details the devices and network architecture for the honeypot.

```

Configuration Files
- This file defines the devices for the network
The file is like this...
Device
Dev ID
Device Type (01, 03, 05; light, hub, thermostat, respectively)
Device SubType      }
Device firmware      }
Device status        }
isresponder[yes/no]  } For the light, hub and thermostat.
Responder information (if applicable)    } The infor under this is only
[yeiscontrollers/no]    } for the thermostat
Controller infomation (if applicable)    }
Current Temp
Current Humidity
FanMode (True = on / False = off)
HeatMode (True = on / False = off)      }
CoolMode (True = on / False = off)      } if both hot and cool True -> Auto mode
Cool Setpoint
Heat Setpoint
----- as seen below -----
Device
A1 A1 A1
01
3A
48
00
isresponderyes
1
B1 B1 B1
iscontrollerno
Device
B1 B1 B1
03
01
01
00
isresponderyes
2
F1 F1 F1
A1 A1 A1
iscontrolleryes
5
A1 A1 A1
C1 C1 C1
D1 D1 D1
E1 E1 E1
F1 F1 F1

4) Simulate Traffic
- This program uses yet another YARD Stick.
run ./insteonsimtraffic.py -i X -f filename where X is the interface for the sending
    YS1 and filename is the input file with the list of commands to send.
The honeypot responds accordingly.

1F : C1 C1 C1 : B1 B1 B1 : 2E 02 00 00 00 00 00 00 00 00 00 00 00 00 00 92 96
2
OF : A1 A1 A1 : B1 B1 B1 : 19 02
1
OF : D1 D1 D1 : B1 B1 B1 : 19 02
1
OF : E1 E1 E1 : B1 B1 B1 : 19 02
1
OF : F1 F1 F1 : B1 B1 B1 : 19 02
1

```

A.2 instelisten.py

```
1 #!/usr/bin/env python
2 """
3 @Description: This is the main listener program. Users can either:
4 1) Listen "off-line" -- save all the traffic to a .raw file, then convert that
5 traffic into readable Insteon and finally converts it to .pcap file.
6 OR
7 2) Listen "live" -- covert raw data live, and using named pipe, send data to
8 Wireshark live.
9
10 """
11
12 from argparse import *
13 from pcapcreator import *
14 from pktcreator import *
15 from rfcatmod import *
16 from insteonanalyzer import *
17 import re
18 import os
19
20 def parseargs():
21     """
22     :return: the arguments.
23     """
24     parser = ArgumentParser(
25         description = 'Read Insteon commands from RfCat and dump to pcap file or
26         named pipe for WireShark. '
27         'Examples:\t./insteondump.py -o capture -- for off-line mode. '
28         '\t\t ./insteondump.py -l -p mypipe -- where mypipe is a pre-named pipe.
29         mkfifo mypipe')
30
31     # keep it simple.
32     parser.add_argument('-i', '--interface', help = 'USB interface to use.
33         Default is 0', default = 0)
34     parser.add_argument('-l', '--livemode', help = 'Do capture with live mode.
35         Must provide -p <pipename>'
36         'for this mode.', action = 'store_true')
37     parser.add_argument('-p', '--pipename', help = 'named pipe for wireshark capture.
38         mkfifo <pipename> first,'
39         'then use -p <pipename> here.')
40     parser.add_argument('-o', '--outputFileName', help = 'Output file name
41         (will be pcap format). '
42         'Assumed to be used in \'offline\' mode')
43
44     return parser.parse_args()
45
46 def updatescreen(devicelist):
47     """
48     might be a simple function, but could be helpful in the future.
49     :param devicelist: the global devicelist.
50     :return: n/a
51     """
52     # found this line online somewhere. it works great!
53     os.system('cls' if os.name == 'nt' else 'clear')
54     devicelistprint(devicelist)
55
56 def livecapture(devicelist, yardstick, pipename):
57     """
58     the function that does the live listening / screen updating.
59     :param devicelist: the devicelist to be maintained throughout the live capture
60     :param yardstick: the yardstick object used.
61     :param pipename: a named pipe to send the data.
```

```

62 :return: n/a
63 """
64
65 # I would normally do this with the an open(filename, "a") function,
66 # but I don't care to save duplicate device IDs.
67 # So I have to read the file first, saving each address,
68 # then re-open it in 'w' mode to write over everything.
69 devicestosave = list()
70 try:
71     # if the file exists, open it, read devices, save them, close file.
72     # reopen file in 'w' mode and write devices.
73     devicesfile = open("insteon.devices", "r")
74
75     for x in devicesfile:
76         devicestosave.append(x.strip("\n"))
77     devicesfile.close()
78
79     # reopening file.
80     devicesfile = open("insteon.devices", "w")
81     for device in devicestosave:
82         devicesfile.write(device + "\n")
83 except IOError:
84     # means that the file didn't exist, so just open it in write mode.
85     devicesfile = open("insteon.devices", "w")
86 finally:
87     # if there's an error here, well...woops?
88     pass
89
90 # need the write in byte mode.
91 mypipe = open(pipename, 'wb')
92
93 # magic numbers, etc for pcap format header.
94 # Wireshark needs this at the beginning of the pipe.
95 bytestring = ("d4 c3 b2 a1" +
96               "02 00" +
97               "04 00" +
98               "00 00 00 00" +
99               "00 00 00 00" +
100              "ff ff 00 00" +
101              "93 00 00 00")
102
103 # send bytes to pipe.
104 writebytestringtofile(bytestring, mypipe)
105
106 while not keystop():
107     try:
108         rawdata, timestamp = yardstick.RFrecv(timeout=2000)
109         timestampstring = str(asctime(localtime(timestamp)))
110         _bytes_ = ''.join("{0:08b}".format(x) for x in bytearray(rawdata))
111
112         subpackets = parse_insteon_pkt(_bytes_, timestampstring)
113         if subpackets is not None:
114             for packet in subpackets:
115                 # I really care about valid packets here.
116                 if packet.validcrc == 1:
117                     timestamp = packet.timestamp
118                     readablepacket = packet.hex_str
119
120                     # analyze the packet
121                     analyzepacket(devicelist, readablepacket)
122
123                     # update the screen
124                     updatescreen(devicelist)
125
126                     # get ready to send this to a pipe.
127                     bytestring = rawtopcap(timestamp, readablepacket)

```

```

128         # send to the pipe.
129         writebytestringtofile(bytestring, mypipe)
130
131         # save any IDs for later spoofing.
132         # A is NAK, which typically means the destination address was a
133         # wrong source address.
134         # Don't want wrong sources.
135         if packet.hex_str[3:11] not in devicestosave and
136            packet.hex_str[0] is not "A":
137
138             devicesfile.write(packet.hex_str[3:11] + "\n")
139             devicesfile.flush()
140             devicestosave.append(packet.hex_str[3:11])
141
142         # however, for some messages, (group, broadcast, etc),
143         # there is not a device ID in the second device ID set.
144         # For the purposes of keeping track of IDs, I don't care.
145         # C is group broadcast standard
146         # 8 is broadcast standard
147         # D is group broadcast extended
148         # 9 is broadcast extended
149         # A is a NAK standard.
150         # B is NAK extended (just to be sure).
151         # Because NAKs typically mean it was from a wrong address,
152         # don't want wrong addresses.
153         if packet.hex_str[0] is not "C" and
154            not "8" and not "D" and not "9":
155
156             if packet.hex_str[12:20] not in devicestosave:
157                 devicesfile.write(packet.hex_str[12:20] + "\n")
158                 devicesfile.flush()
159                 devicestosave.append(packet.hex_str[12:20])
160         devicesfile.flush()
161
162     except KeyboardInterrupt:
163         print "Please press <enter> to stop"
164
165     except IndexError:
166         print "Index fail"
167
168     except ChipconUsbTimeoutException:
169         print "ChipconUsbTimeoutException"
170
171     finally:
172         pass
173
174     # ensure the yardstick goes into idle mode
175     yardstick.setModeIDLE()
176
177
178 def offlinecapture(yardstick, rawout):
179     print "press <enter> when done, then view output file in WireShark...."
180     while not keystop():
181         try:
182             rawdata, timestamp = yardstick.RFrecv(timeout=2000)
183
184             # converts the timeStamp info returned from RFrecv and
185             # turns it into a readable string.
186             # important for creating / sorting / viewing in WireShark.
187             timestampstring = str(asctime(localtime(timestamp)))
188
189             # makes certain that the raw bytearray data turns into a string of 1s and 0s
190
191             _bytes_ = ''.join("{0:08b}".format(x) for x in bytearray(rawdata))
192
193             # start building the raw data file.

```

```

193         # This minimizes the amount of time spent not in the RRecv function.
194         rawout.write(timestampstring + "\n")
195         rawout.write(_bytes_ + "\n")
196
197     except KeyboardInterrupt:
198         print "Please press <enter> to stop"
199
200     except IndexError:
201         print "Index fail"
202
203     except ChipconUsbTimeoutException:
204         print "ChipconUsbTimeoutException"
205
206     finally:
207         pass
208
209     # ensure the yardstick goes into idle mode
210     yardstick.setModeIDLE()
211
212 def main():
213     """
214     :return: Main doesn't return anything, but it can exit if the output file
215             name is not given as an argument.
216     """
217     # just get the args right away.
218     args = parseargs()
219
220
221
222     # see if the user wants live mode or non-live mode.
223     livemode = args.livemode
224     if livemode:
225         # devicelist keeps track of devices I've heard & starts building relationships.
226         # needed for ntwk map.
227         devicelist = list()
228
229         # better exist.
230         pipename = args.pipename
231         if pipename is None and not re.search("\W", pipename):
232             print "Requires '-p <pipename> to direct the live capture to a named pipe.
233                   Also, only letters and numbers."
234             sys.exit(0)
235
236         print "view output file in WireShark....press <enter> when done."
237
238         # start actually capturing stuff.
239         # note: should probably do some kind of validation for the interface...
240
241         interface = int(args.interface)
242         yardstick = configure_rfcats(interface)
243         livecapture(devicelist, yardstick, pipename)
244     else:
245         # get the output file name the user wants from args.
246         # if no output file name included, quit.
247         pcapoutputfilename = args.outputFileName
248
249         if pcapoutputfilename is None:
250             print "Requires '-o <output_file_name>' to store a pcap file."
251             sys.exit(0)
252
253         if re.search("\W", pcapoutputfilename):
254             print "pcap file name must only use letters and numbers.
255                   no special characters, please. "
256             print "The \'pcap\' tag will be added later."
257             sys.exit(0)
258

```

```

259     # adding the .pcap
260     pcapoutputfilename += ".pcap"
261
262     # i'm going to create 2 other temp files in addition to the pcap.
263     # call them .raw and .insteondump.
264     # will be used for conversions later.
265     rawoutputfilename = pcapoutputfilename.replace(".pcap", ".raw")
266     insteonoutputfilename = pcapoutputfilename.replace(".pcap", ".insteondump")
267
268     # get the rawoutput handle to give to the offlinecapture function.
269     rawouthandle = open(rawoutputfilename, "w")
270
271     # create the yardstick to give the capture function.
272     yardstick = configure_rfcats(int(args.interface))
273
274     # capture!
275     offlinecapture(yardstick, rawouthandle)
276
277     # 2 conversions.
278     createreadablefile(rawoutputfilename, insteonoutputfilename)
279     createpcap(insteonoutputfilename, pcapoutputfilename)
280
281     # analyze the file for display.
282     analyzefile(insteonoutputfilename)
283
284 # run main.
285 if __name__ == "__main__":
286     main()
287     sys.exit(1)

```

A.3 rfcatsmod.py

```
1 #!/usr/bin/env python
2 """
3 @Description: This is the config function for the yardstick.
4 I did this so I could ensure the listener and sender were configured
5 exactly the same for maximum spam-listening.
6 Use this package to create the yardsticks
7
8 """
9
10 from rflib import *
11
12
13 def configure_rfcats(interface=0):
14     """
15     :param interface: YardStick USB to use (first one plugged in is 0, next is 1, etc)
16     :return: The configured YardStick (RfCat) device.
17     """
18     # configs.
19     frequency = 915000000
20     brate = 9125
21     bandwidth = 200000
22     mdmdeviation = 75000
23     # syncword = 0x662A
24     # syncword = 0x6666
25
26     # make the device.
27     device = RfCat(interface)
28
29     # configure the device.
30     device.setFreq(frequency)
31     device.setMdmDRate(brate)
32     device.setMdmChanBW(bandwidth)
33     device.setMdmDeviatn(mdmdeviation)
34     device.setMdmModulation(MOD_2FSK)
35     device.setMdmSyncMode(SYNCM_CARRIER)
36     # device.setMdmSyncMode(SYNCM_CARRIER_15_of_16)
37     # device.setMdmSyncWord(syncword)
38     device.setBSLimit(BSCFG_BS_LIMIT_6)
39     device.setPktPQT(0)
40     device.setMaxPower()
41
42     return device
```

A.4 insteonanalyzer.py

```
1  #!/usr/bin/env python
2  """
3  @Description: This module reads an Insteon packet (or a .insteondump file) from the
4  off-line mode) and analyzes what happened. This builds the network "map".
5  The Device class contains the necessary attributes per device ID.
6  """
7
8  import argparse
9  import sys
10
11
12  class Device(object):
13      # obj for keeping track of the devices in the observed network traffic.
14      def __init__(self, address, devcatstring=None, isresponder=False,
15                  respondstoaddress=None, iscontroller=False, controlsaddress=None,
16                  isgroupcontroller=False, controllgroupaddress=None, isgroupresponder=False,
17                  respondstogroupaddress=None):
18          # the address is important
19          self.address = address
20
21          # info about the device, if available.
22          self.devcatstring = devcatstring
23          self.devcatdescription = "unknown device type"
24          if self.devcatstring is not None:
25              self.devcatlookup(devcatstring)
26
27          # on init, this device could be a responder
28          self.isresponder = isresponder
29          self.respondstolist = list()
30          if self.isresponder:
31              self.udpaterespondsdirecttolist(respondstoaddress)
32
33          # on init, this device could be a controller
34          self.iscontroller = iscontroller
35          self.controlslist = list()
36          if self.iscontroller:
37              self.updatecontrolsdirectlist(controlsaddress)
38
39          # contoller group info
40          self.isgroupcontroller = isgroupcontroller
41          self.controlgroupaddresses = list()
42          if self.isgroupcontroller:
43              self.updatecontrolgrouplist(controllgroupaddress)
44
45          # responder group info
46          self.isgroupresponder = isgroupresponder
47          self.respondgroupaddresses = list()
48          if self.isgroupresponder:
49              self.updaterepondtougrouplist(respondstogroupaddress)
50
51      def devcatlookup(self, devcatstring):
52          if self.devcatstring is None:
53              return "unknown device type"
54          else:
55              # make sure to not get an index out of bounds.
56              if int(self.devcatstring, 16) > len(self.devcatlookuptable):
57                  return "unknown device type"
58              else:
59                  return self.devcatlookuptable[int(self.devcatstring, 16)]
60
61      def udpaterespondsdirecttolist(self, respondstoaddress):
```

```

62         if len(self.respondstolist) == 0:
63             self.respondstolist.append(respondstoaddress)
64         else:
65             if respondstoaddress not in self.respondstolist:
66                 self.respondstolist.append(respondstoaddress)
67
68     def updatecontrollsdirectlist(self, controlsaddress):
69         if len(self.controlslist) == 0:
70             self.controlslist.append(controlsaddress)
71         else:
72             if controlsaddress not in self.controlslist:
73                 self.controlslist.append(controlsaddress)
74
75     def updatecontrolgrouplist(self, groupaddress):
76         if len(self.controlgroupaddresses) == 0:
77             self.controlgroupaddresses.append(groupaddress)
78         else:
79             if groupaddress not in self.controlgroupaddresses:
80                 self.controlgroupaddresses.append(groupaddress)
81
82     def updaterepondtogrouplist(self, groupaddress):
83         if len(self.respondgroupaddresses) == 0:
84             self.respondgroupaddresses.append(groupaddress)
85         else:
86             if groupaddress not in self.respondgroupaddresses:
87                 self.respondgroupaddresses.append(groupaddress)
88
89     devcatlookuptable = ["General Controller", "Dimmable Lighting Control",
90                          "Switched Lighting Control", "Network Bridge",
91                          "Irrigation Control", "Climate Control", "Pool and Spa control",
92                          "Sensors and Actuators", "Home Entertainment",
93                          "Energy Management", "Built-in Appliance",
94                          "Plumbing", "Communication", "Computer Control",
95                          "Window Coverings", "Access Control",
96                          "Security, Health, Safety", "Surveillance",
97                          "Automotive", "other"]
98
99 # ----- END OF CLASS -----#
100
101 def parseargs():
102     """
103     :return: the arguments.
104     """
105     parser = argparse.ArgumentParser(
106         description='Read Insteon log file and create network "map"
107         and categorize devices (if ID request command available)')
108
109
110     parser.add_argument('-i', '--inputfilename', help='Input file name.
111         File must be of the .insteondump format')
112     return parser.parse_args()
113
114 def devicelistprint(devicelist):
115     if len(devicelist) == 0:
116         print "no devices relationships found in capture"
117     else:
118         print "-----"
119         for device in devicelist:
120             if device.devcatstring is not None:
121                 print device.address, "is a: ", device.devcatdescription
122             else:
123                 print device.address, "is unknown"
124
125     print
126     # tell user about the controllers.
127     print "Controllers are:"

```

```

128     for device in devicelist:
129         if device.iscontroller:
130             print device.address, " controls: ", device.controlslist
131
132     print
133     # tell user about responders.
134     print "Responders are:"
135     for device in devicelist:
136         if device.isresponder:
137             print device.address, " responds to: ", device.respondstolist
138
139     print
140     # tell user about any group controllers
141     print "Group controllers are:"
142     for device in devicelist:
143         if device.isgroupcontroller:
144             print device.address, " controls group(s): ",
145                 device.controlgroupaddresses
146
147     print
148     # tell user about any group responders
149     print "Group responders are:"
150     for device in devicelist:
151         if device.isgroupresponder:
152             print device.address, " responds to group(s): ",
153                 device.respondgroupaddresses
154     print "-----"

```

A.5 pktcreator.py

```
1  #!/usr/bin/env python
2  """
3  @Description: This file contains the Pkt class, which is used for parsing raw bits
4  into lists of Pkts. Additionally, pkt_crc function is used in the sending process.
5  """
6
7  from myexceptions import InvalidManchesterEncoding
8
9
10 starthead_i = "0011000101010101"
11 starthead = "1100111010101010"
12 startheadmark = 5
13
14 lsfr_table = [0x00, 0x30, 0x60, 0x50, # 0 1 2 3
15               0xC0, 0xF0, 0xA0, 0x90, # 4 5 6 7
16               0x80, 0xB0, 0xE0, 0xD0, # 8 9 A B
17               0x40, 0x70, 0x20, 0x10] # C D E F
18
19 class Pkt(object):
20     def __init__(self, dat, timestamp=None, offset=0) :
21         """
22         reads a raw packet in the form of a string of '1' & '0'
23         """
24         self.raw_dat = dat
25         self.offset = dat
26         self.validcrc = 0
27         self.calc_crc = None
28         self.timestamp = timestamp
29         self.dat = extract_pkt_data(self.raw_dat)
30
31         if len(self.dat) >= 10:
32             self.calc_crc = pkt_crc(self.dat)
33             self.hex_str = self.pkt_simple()
34
35     def pkt_simple(self):
36         try:
37             if len(self.dat) < 4 :
38                 return ""
39             h = ["{:02X}".format(x) for x in self.dat]
40             # a = [h[0], " ".join(h[1:4]), " ".join(h[4:7]), " ".join(h[7:23])]
41             a = [h[0], " ".join(h[1:4]), " ".join(h[4:7]), " ".join(h[7:23])]
42
43             if len(h) > 22:
44                 # a.append(":")
45                 a.append(" ".join(h[23:]))
46
47         except Exception, err:
48             # pass
49             print "err", err
50             if self.dat:
51                 print "dat", self.dat
52             # print "a", a
53             # if h: print "h", h
54             # raise
55
56             return " ".join(a)
57 # ----END OF CLASS-----### #
58
59 def pkt_crc(dat):
60     """
61     :param dat: takes an instion packet in form of a list of ints
```

```

62 :return: returns the CRC for RF packet
63
64 This uses a table lookup to effectively doing:
65 r ^= dat[i] ;
66 r ^= (( r ^ ( r << 1 )) & 0x0F) << 4 ;
67 """
68
69 # check if this is an short packet or extended packet
70 if dat[0] & 0b00010000:
71     crc_len = 23
72 else:
73     crc_len = 9
74
75 r = 0
76 for i in dat[:crc_len]:
77     r ^= i
78     r ^= lsfr_table[r & 0x0F]
79
80 return r
81
82 def parse_insteon_pkt(data, timestamp):
83     """
84     :param data: string of raw bytes
85     :param timestamp: timestamp info for the bytes
86     :return: a list of insteon pkt in the form of a list of hex strings.
87     """
88
89     pktlist = get_pkt(data, timestamp)
90     if pktlist is None:
91         return None
92     for p in pktlist:
93         if p.calc_crc is not None:
94             if p.dat[0] & 0b00010000:
95                 x = 23
96                 p.hex_str = p.hex_str[0:x * 3 + 2]
97             else:
98                 x = 9
99                 p.hex_str = p.hex_str[0:x * 3 + 2]
100         if x <= len(p.dat):
101             if p.calc_crc == p.dat[x]:
102                 p.validcrc = 1
103     return pktlist
104
105 def demanchester(s):
106     """
107     :param s: byte in string "10011001
108     :return: nibble in string "1010"
109     """
110     i = 1
111     b = list()
112     slen = len(s)
113
114     # Note that since 1 starts as "1"
115     # we are testing the 2nd bit for '1' or '0'
116     while i < slen:
117         if s[i] == s[i - 1]:
118             raise InvalidManchesterEncoding(s[i], s[i - 1])
119         if s[i] == "1":
120             b.append("1")
121         else:
122             b.append("0")
123         i += 2
124     return "".join(b)
125
126 def extract_pkt_data(dat):
127     """

```

```

128 reads a raw packet in the form of a string of "1" & "0"
129 returns a string parsed data in the form of an array of ints
130 """
131
132 i = 0
133 if dat[startheademark:startheademark+2] == "11" :
134     i = startheademark
135
136 results = list()
137
138 # j = 0
139 dm = ''
140 while dat[i:i + 2] == "11":
141     i += 2
142     try:
143         dm = demanchester(dat[i:i + 26])
144     except Exception:
145         if len(results) >= 10 :
146             break
147
148     if len(dm) < 13:
149         break
150     i += 26
151     # count_field = int(dm[4::-1], 2)
152     dat_field = int(dm[:4:-1], 2)
153
154     results.append(dat_field)
155
156 return results
157
158 def get_pkt(dat, ts):
159     """ returns list packet object """
160     try:
161         si = startindex = dat.find(starthead, 0)
162
163         if startindex == -1:
164             si = startindex = dat.find(starthead_i, 0)
165             if si > -1:
166                 dat = dat.replace('0','A')
167                 dat = dat.replace('1','0')
168                 dat = dat.replace('A','1')
169
170             if startindex == -1:
171                 # print "No starthead"
172                 return None
173
174             header_len = len(starthead)
175             data_len=len(dat)
176             subpkt = list()
177             pkt_list = list()
178
179             prev_header = si
180             si = dat.find(starthead, prev_header + header_len)
181             while si > 0:
182                 subpkt.append((prev_header, si))
183                 prev_header = si
184                 si = dat.find(starthead, prev_header + header_len)
185             else:
186                 subpkt.append((prev_header, data_len))
187
188         except IndexError:
189             print "Index Fail"
190             return
191
192     for sb in subpkt:
193         try:

```

```
194         pkt = Pkt(dat[sb[0]:sb[1]], ts, sb[0])
195     except ValueError:
196         print ["{:02X}".format(x) for x in pkt.dat]
197         continue
198     else:
199         pkt_list.append(pkt)
200
201     return pkt_list
```

A.6 pcapcreator.py

```
1 #!/usr/bin/env python
2 """
3 @Description: This reads in a rawfile and outputs a correctly formatted pcap file.
4     The first line of the input file is a asctime(localtime()) string.
5     The second line is a string of bytes representing an Insteon packet.
6     Repeat to end of file.
7
8 -- Input file format example:
9 Tue Jun 21 13:10:03 2016
10 01 E7 5C 2F DE C2 33 11 FF 44
11
12 """
13 import binascii
14 from time import *
15 import time
16 from pktcreator import *
17
18
19 # writes the string of bytes to the file in the arguments.
20 def writebytestringtofile(bytestring, filehandle):
21     bytelist = bytestring.split()
22     bytes = binascii.a2b_hex(''.join(bytelist))
23     filehandle.write(bytes)
24     filehandle.flush()
25
26 # a helper function to get the length of the bytes.
27 # necessary for pcap file information.
28 def getbytlength(str1):
29     return len(''.join(str1.split())) / 2
30
31 # conversion function. Takes timestamp information and a message and
32 # gives back a pcap formatted sub-packet.
33 def rawtopcap(timestamp, message):
34
35     # pcap packet header that must preface every packet
36     pcap_packet_header = ('00 00 00 00' # time info.
37         'XX XX XX XX' # Frame Size (little endian)
38         'YY YY YY YY') # Frame Size (little endian)
39     pcap_len = getbytlength(message)
40     hex_str = "%08x"%pcap_len
41     # reverse the hex so that it's the correct format (little endian)
42     reverse_hex_str = hex_str[6:] + hex_str[4:6] + hex_str[2:4] + hex_str[:2]
43     pcaph = pcap_packet_header.replace('XX XX XX XX',reverse_hex_str)
44     pcaph = pcaph.replace('YY YY YY YY',reverse_hex_str)
45
46     # get the time. going to have to change this to be reading from
47     # either the screen or a file.
48     # the below conversion takes a string input of time and spits out hex.
49     # beautiful. pay attention to the
50     # format stuff at the end. This page was helpful
51     # https://docs.python.org/3/library/time.html
52     timehex = hex(int(time.mktime((time.strptime(timestamp,'%a %b %d %H:%M:%S %Y')))))
53
54     # but you gotta strip the "0x" from the first part of the hex and make it a string
55     timehexstr_rvrs = str(timehex)[2:]
56
57     # now time to re-order that hex to be reversed because little endian.
58     # use same method as above.
59     timehexstr = timehexstr_rvrs[6:] + timehexstr_rvrs[4:6] +
60         timehexstr_rvrs[2:4] + timehexstr_rvrs[:2]
61
```

```

62     # put it all together in a string of bytes.
63     bytestring = timehexstr + pcap + message
64
65     # return those bytes to the overall bytestring
66     return bytestring
67
68 def createreadablefile(inputfilename, outputfilename):
69     """
70     :param inputfilename: file name to read from
71     :param outputfilename: file name to write to
72     :return: None
73     """
74     # i know this file exists because I just used it. No need to check if it exists.
75     fin = open(inputfilename, "r")
76     readout = open(outputfilename, "w")
77
78     devicestosave = list()
79     # if the file exists, read the devicestosave listed into devicestosave list.
80     # reopen file and write devices again.
81     try:
82         devicesfile = open("insteon.devices", "r")
83
84         for x in devicesfile:
85             devicestosave.append(x.strip("\n"))
86         devicesfile.close()
87         devicesfile = open("insteon.devices", "w")
88         for device in devicestosave:
89             devicesfile.write(device+"\n")
90     except IOError:
91         devicesfile = open("insteon.devices", "w")
92
93     finally:
94         pass
95
96     # start reading from the readable file.
97     for _timestamp_ in fin:
98         _rawData_ = next(fin)
99         _rawData_ = _rawData_.strip("\n")
100         subpackets = parse_insteon_pkt(_rawData_, _timestamp_)
101         if subpackets is not None:
102             for packet in subpackets:
103                 # validcrc is set to one at the end of parse_insteon_pkt
104                 # if the packet is full, etc etc.
105                 if packet.validcrc == 1:
106                     # generate the readable lines necessary.
107                     readablebytes = _timestamp_
108                     readablebytes += packet.hex_str + "\n"
109                     # send readable bytes to file.
110                     readout.write(readablebytes)
111                     # there is always a device ID in a message (first device ID set)
112                     if packet.hex_str[3:11] not in devicestosave:
113                         devicesfile.write(packet.hex_str[3:11]+"\n")
114                         devicestosave.append(packet.hex_str[3:11])
115
116                     # however, for some messages, (group, broadcast, etc),
117                     # there is not a device ID
118                     # in the second device ID set.
119                     # For the purposes of keeping track of IDs, I don't care.
120                     # C is group broadcast standard
121                     # 8 is broadcast standard
122                     # D is group broadcast extended
123                     # 9 is broadcast extended
124                     if packet.hex_str[0] is not "C" and not "8" and not "D" and not "9":
125                         if packet.hex_str[12:20] not in devicestosave:
126                             devicesfile.write(packet.hex_str[12:20]+"\n")
127                             devicestosave.append(packet.hex_str[12:20])

```

```

128     print "known device IDs (could be from previous captures) : ", devicestosave
129
130 def createpcap(inputfilename, outputfilename):
131     # bytestring is initially set to the global pcap header.
132     # look here for more info:
133     # http://www.codeproject.com/Tips/612847/
134     # Generate-a-quick-and-easy-custom-pcap-file-using-P
135     bytestring = ('D4 C3 B2 A1' # magic numbers
136                  '02 00'        # File format major revision (i.e. pcap <2>.4)
137                  '04 00'        # File format minor revision (i.e. pcap 2.<4>)
138                  '00 00 00 00'  # timezone info. leaving blank.
139                  '00 00 00 00'  # accuracy of timestamps
140                  'FF FF 00 00'  # max length of captured packet
141                  '93 00 00 00') # Datalink type 93 = DLT_USR 147
142                                # the WireShark non-defined protocol.
143     outfilehandle = open(outputfilename, "wb")
144     writebytestringtofile(bytestring, outfilehandle)
145
146
147     # open the readable file
148     raw_file = open(inputfilename, "r")
149     # going to tell user how many packets we got...
150     counter = 0
151     # start by reading the first thing (a timestamp)
152     for __timestamp__ in raw_file:
153         __timestamp__ = __timestamp__.strip("\n") # gotta strip the newline
154         __rawdata__ = next(raw_file)
155         __rawdata__ = __rawdata__.strip("\n") # gotta strip the newline
156
157         # send the timestamp and rawdata to the converter function,
158         # then add that returning info to bytestring.
159         bytestring = rawtopcap(__timestamp__, __rawdata__)
160         writebytestringtofile(bytestring, outfilehandle)
161         counter += 1
162
163     # write the entire bytestring to the outputfile.
164     # hopefully that bytestring doesn't get too big....
165
166     print "received: " + str(counter) + " packets"
167
168     # for when I want to run this file individually.
169     # createpcap("rawfile.txt","output.pcap")

```

A.7 myexceptions.py

```
1 #!/usr/bin/env python
2
3
4 class InvalidManchesterEncoding(Exception):
5     def __init__(self, s1, s2):
6         self.value = "%s == %s" % (s1, s2)
7         self.s1 = s1
8         self.s2 = s2
9
10    def __str__(self):
11        err = "Invalid Manchester Encoding: %s == %s"
12        err %= (self.s1, self.s2)
13        return err
```

A.8 insteonsend.py

```

1 #!/usr/bin/env python
2 """
3 @Description: Most of this code was written by Evil Pete (Peter Shipley).
4 I modified portions, but mainly in an effort to "pretty-up" the code.
5 It works great, though! Of note, I decided to take out all of the argument options
6 that Shipley had, allowing only the sending of the raw data.
7 Figure out what the packet flags and order for yourself.
8
9 A good number of these functions are called in insteonscanner.py
10 """
11
12 import argparse
13 from pktcreator import pkt_crc # needed for calculating crc before it's sent.
14 from rfcatmod import *
15 import threading
16
17 rawpkt = False
18 verbose = False
19 interface = 0
20
21 preamble = "0101010101"
22 postamble = "0101010101010101010101010101010101010101010101010101010101010101"
23
24
25 def read_raw_pkt(plist):
26     a = list()
27     for i in plist:
28         if i == ":":
29             continue
30         a.append(int(i, 16))
31
32     return a
33
34 def init():
35     parser = argparse.ArgumentParser(add_help=True,
36     epilog="example:\n\t# ./insteonsend.py -v -r 05 : 33 d3 32 : de c2 33 : 11 ff",
37     description='generate an Insteon packet')
38
39     parser.add_argument('-i', '--interface', help='USB interface', default=0)
40
41     parser.add_argument("-v", "--verbose", action="store_true", help="verbose mode")
42
43     parser.add_argument("-r", "--raw", dest="raw",
44     action='store_true',
45     help="Raw Packet")
46
47     args, unknown_args = parser.parse_known_args()
48
49     if args.verbose:
50         # go get the verbose variable
51         global verbose
52         verbose = args.verbose
53
54     if args.raw:
55         # go get the rawpkt variable.
56         global rawpkt
57         rawpkt = args.raw
58
59     if args.interface:
60         global interface
61         interface = int(args.interface)

```

```

62
63     return unknown_args
64
65 def manchester_encode(*args):
66     """
67     :param args: input    string of "1011"
68     :return: output    string of "01100101"
69     """
70     return_bits = list()
71
72     # nest of for loops since args may be a list of strings
73     for x in args:
74         for y in x:
75             for z in y:
76                 if z == "0":
77                     return_bits.append("10")
78                 elif z == "1":
79                     return_bits.append("01")
80                 elif z == "-":
81                     return_bits.append(z)
82                 else:
83                     print "char=",z
84                     raise ValueError("invalid bit '{:!s}'".format(z))
85
86     return "". join(return_bits)
87
88 def assemble_packet(dat, firstmarker=1):
89     ret = list()
90     if isinstance(dat[0], int):
91         dat_int = dat[:]
92     else:
93         print "dat is not of an int list...quiting"
94         exit(0)
95
96     # preamble defined already, make it manchester and append
97     m = manchester_encode(preamble)
98     ret.append(m)
99
100     # now...time to encode the rest of the bytes.
101     dat_len = len(dat_int)
102     if firstmarker == 1:
103         ret.append('11')
104
105         firstb = dat_int.pop(0)
106         x = "{:08b}".format(firstb)[::-1]
107         m = manchester_encode("11111", x)
108         ret.append(m)
109
110         if firstb & 0b00010000:
111             c = 31
112         else:
113             c = 11
114     else:
115         c = dat_len - 1
116
117     for x in xrange(len(dat_int)):
118         rc = "{:05b}".format(c)[::-1]
119         c -= 1
120         rx = "{:08b}".format(dat_int[x])[::-1]
121         ret.append('11')
122         m = manchester_encode(rc, rx)
123         ret.append(m)
124
125     m = manchester_encode(postamble)
126     ret.append(m)
127

```

```

128     ret_str = "".join(ret)
129     # print ret_str
130     return ret_str
131
132 def invert_pkt(p):
133     p = p.replace('0', 'A')
134     p = p.replace('1', '0')
135     p = p.replace('A', '1')
136     return p
137
138 def b_to_binstr(dat):
139     if verbose:
140         print "bits sent:", dat
141     r = bytearray(int(dat[x:x + 8], 2) for x in range(0, len(dat), 8))
142     return r
143
144 def send_pkt(device, dat):
145     bs = b_to_binstr(dat)
146     try:
147         device.RFxmit(bs)
148         # time.sleep(60 / 1000) # is there a reason for sleeping after sending??
149         # device.strobeModeIDLE()
150     finally:
151         device.setModeIDLE()
152
153 def main():
154     av = init()
155     global rawpkt
156     if rawpkt:
157         if len(av) == 0:
158             print "must have either 9 or 23 args in hex"
159             exit(1)
160         pktlistdecimal = read_raw_pkt(av)
161         if len(pktlistdecimal) == 9 or len(pktlistdecimal) == 23:
162             pktcrc = pkt_crc(pktlistdecimal)
163             pktlistdecimal.append(pktcrc)
164
165             if verbose:
166                 print "pkt crc: ", pktcrc
167                 print "full pkt in decimal: ", pktlistdecimal
168             else:
169                 print "must have either 9 or 23 args in hex"
170                 exit(1)
171     if verbose:
172         print ">> sys.stderr, \"full pkt list in hex: \", [{:02X}].format(j) for j in
            pktlistdecimal]
173
174     pkttosend = assemble_packet(pktlistdecimal)
175     # print "{:s}".format(pkttosend)
176
177     pkttosendinverted = invert_pkt(pkttosend)
178     # print "{:s}".format(pkttosendinverted)
179
180     # AT THIS POINT, pkttosendinverted HAS THE CORRECT INFORMATION TO MAKE IT
181     # THROUGH THE PARSER ./print_pkt.py (Shipley's old printer).
182     # SO...SHOULD BE GOOD SEND IT...WILL HAVE TO MAKE SURE THAT
183     # MY OWN RECEIVER WILL BE ABLE TO HEAR IT.
184     device = configure_rfcats(interface)
185     send_pkt(device, pkttosendinverted)
186
187 if __name__ == "__main__":
188     main()
189     exit(0)

```

A.9 insteonscanner.py

```
1 #!/usr/bin/env python
2 """
3 @Description: Read from the known devices file.
4     Ping and do an ID request from each device to every other device.
5 It takes some time, but this is the main enumeration process.
6 """
7
8
9 from insteonsend import *
10
11
12 def parseargs():
13     """
14     :return: the arguments.
15     """
16     parser = argparse.ArgumentParser(
17         description='Spoof ping and ID request commands from the insteon.devices list.
18         This file must already exist and device IDs must be in this list to work.')
19
20     parser.add_argument('-i', '--interface', help='USB interface', default=0)
21     return parser.parse_args()
22
23
24
25 def getdevicelist():
26     """
27     :return: list of devices from the insteon.devices file, if it exists.
28     """
29     devices = list()
30     try:
31         devicesfile = open("insteel.devices", "r")
32         for x in devicesfile:
33             devices.append(x.strip("\n"))
34         devicesfile.close()
35
36     except IOError:
37         # if the file doesn't exist, tell user to run
38         # insteonlisten.py -d -o output.pcap to save devices.
39         print "program needs insteon.devices file to exist. That file is created by
40         running ./insteelonlisten.py and observing for an Insteel action."
41         sys.exit(0)
42     finally:
43         pass
44
45     return devices
46
47 def buildpackets():
48     """
49     :return: list of commands to ping and perform ID request
50     """
51     # get the list of devices.
52     devicelist = getdevicelist()
53
54     # start setting up the list of packets.
55     packetslist = list()
56
57     # individual statements will be held here.
58     statement = list()
59
60     # for simplifying this to a for-loop.
61     command1 = list()
```

```

62 # command1.append("0F") # ping command
63 command1.append("10") # ID request command
64 # command1.append("03")
65
66 # i'm going to scan through each device for each device.
67 for device1 in devicelist:
68     for device2 in devicelist:
69         # don't care about having the same device ID for both addresses...
70         if device1 is not device2:
71             for command in command1:
72                 # build each command that I want to sent.
73                 statement.append("0F") # for now, standard direct messages.
74                 statement.append(device1[0:2])
75                 statement.append(device1[3:5])
76                 statement.append(device1[6:])
77                 statement.append(device2[0:2])
78                 statement.append(device2[3:5])
79                 statement.append(device2[6:])
80                 statement.append(command)
81                 statement.append("02")
82                 packetslist.append(statement[:])
83                 statement = list()
84
85 # packetlist contains full list of commands (without CRC).
86 # now that I have all the commands in the packetslist.
87 # I need to convert those lists to integers and calc the crc.
88 finalpackets = list()
89 for packet in packetslist:
90     pktlistdecimal = read_raw_pkt(packet)
91     if len(pktlistdecimal) == 9 or len(pktlistdecimal) == 23:
92         pktcrc = pkt_crc(pktlistdecimal)
93         pktlistdecimal.append(pktcrc)
94         finalpackets.append(pktlistdecimal)
95         # have to reset the pktlistdecimal for some reason....
96         pktlistdecimal = list()
97
98 # will return both the final packets in decimal form
99 # and packets list for nice printing later.
100 return finalpackets, packetslist
101
102 def main():
103     # create pre-gened packets from insteon.devices list and command table
104     # I really just care about ping (0F 00) and ID Request (10 00)
105     pktlistdecimal, pktlisthex = buildpackets()
106
107     # give me a yardstick. I'm going to lock it up for the entire time I'm scanning.
108     args = parseargs()
109     yardstick = configure_rfcats(int(args.interface))
110
111     # for every packet in the packet list, assemble the binary and send.
112     # Will wait 1/2 second between sends for
113     # listener see what comes back.
114     for pkt in pktlistdecimal:
115         pkttosend = assemble_packet(pkt)
116         pkttosendinverted = invert_pkt(pkttosend)
117
118         for x in range(0, 3):
119             send_pkt(yardstick, pkttosendinverted)
120             # send, wait for responses. 1/2 second should be enough time.
121             # hopefully the live capture tool will keep up!
122             time.sleep(.5)
123             index = pktlistdecimal.index(pkt)
124             print "sent: ", pktlisthex[index]
125
126     yardstick.setModeIDLE()
127

```

```
128
129 # run main.
130 if __name__ == "__main__":
131     main()
132     sys.exit(1)
```

A.10 insteonhoneydevices.py

```
1 #!/usr/bin/env python
2 """
3 @Description: This program will be the class defining a simple for an Insteon Honey
4 Pot.
5 There are 2 goals:
6 1) Create simulate fake traffic to confuse a would-be attacker and
7 obfuscate my real devices.
8 2) Allow the attacker to interact with my fake devices.
9 The attacker should think that he's turning the lights on and off, etc.
10 """
11
12
13 class FakeSimpleDevice(object):
14     def __init__(self, devid):
15         self.devid = devid
16         self.devcats = None
17         self.devsubcat = None
18         self.firmware = None
19         self.status = None
20         self.isresponder = None
21         self.respondsto = list()
22         self.iscontroller = None
23         self.controls = list()
24         self.devcatsdescription = None
25
26     def devcatslookup(self):
27         if self.devcats is None:
28             self.devcatsdescription = "unknown"
29         else:
30             # make sure to not get an index out of bounds.
31             if int(self.devcats, 16) > len(self.devcatslookuptable):
32                 self.devcatsdescription = "unknown"
33             else:
34                 self.devcatsdescription = self.devcatslookuptable[int(self.devcats, 16)]
35
36     def getdevcatsdescription(self):
37         return self.devcatsdescription
38
39     def printdevice(self):
40         print "DevID: " + self.getdevid()
41         print "\t category: " + self.getdevcategory() + " -> " +
42             str(self.getdevcatsdescription())
43         print "\t subcategory: " + self.getdevsubcategory()
44         print "\t firmware: " + self.getfirmware()
45         print "\t status: " + self.getstatus()
46         if self.getisresponder():
47             print "\t respondsto: " + str(self.getrespondstolist())
48
49         if self.getiscontroller():
50             print "\t controls: " + str(self.getcontrolslist())
51
52     def setfirmware(self, firmware):
53         self.firmware = firmware
54
55     def setdevcategory(self, devcat):
56         self.devcats = devcat
57
58     def setdevsubcategory(self, devsubcat):
59         self.devsubcat = devsubcat
60
```

```

61     def setisresponder(self, isresponder):
62         self.isresponder = isresponder
63
64     def setiscontroller(self, iscontroller):
65         self.iscontroller = iscontroller
66
67     def getdevcategory(self):
68         return self.devcat
69
70     def getdevsubcategory(self):
71         return self.devsubcat
72
73     def getfirmware(self):
74         return self.firmware
75
76     def getdevid(self):
77         return self.devid
78
79     def setstatus(self, status):
80         self.status = status
81
82     def getstatus(self):
83         return self.status
84
85     def getisresponder(self):
86         return self.isresponder
87
88     def getiscontroller(self):
89         return self.iscontroller
90
91     def addresponder(self, address):
92         self.respondsto.append(address)
93
94     def getrespondstolist(self):
95         return self.respondsto
96
97     def addcontroller(self, address):
98         self.controls.append(address)
99
100    def getcontrolslist(self):
101        return self.controls
102
103    devcatlookuptable = ["General Controller", "Dimmable Lighting Control",
104                        "Switched Lighting Control", "Network Bridge",
105                        "Irrigation Control", "Climate Control", "Pool and Spa control",
106                        "Sensors and Actuators", "Home Entertainment",
107                        "Energy Management", "Built-in Appliance",
108                        "Plumbing", "Communication", "Computer Control",
109                        "Window Coverings", "Access Control",
110                        "Security, Health, Safety", "Surveillance",
111                        "Automotive", "other"]

```

A.11 insteonthermostat.py

```
1 #!/usr/bin/env python
2 """
3 @Description: This program will be the class defining a Thermostat
4 for an Insteon Honey Pot.
5 """
6
7 from math import *
8 from time import *
9 import numpy as np
10
11
12 class FakeThermostat(object):
13     def __init__(self, devid):
14         self.devid = devid
15         self.devcats = None
16         self.devsubcats = None
17         self.firmware = None
18         self.status = None
19         self.isresponder = None
20         self.respondsto = list()
21         self.iscontroller = None
22         self.controls = list()
23         self.currenttemp = None
24         self.humidity = None
25         self.currentcoolsetpoint = None
26         self.currentheatsetpoint = None
27         # self.flags = None
28         self.devcatsdescription = None
29         self.lasttimestamp = time() # on init this is easiest
30         self.fanstatus = None
31         self.heatonoff = None
32         self.coolonoff = None
33         self.leaveonoff = False
34         self.coolactive = False
35         self.heatactive = False
36         self.fanactive = False
37
38     def devcatslookup(self):
39         if self.devcats is None:
40             self.devcatsdescription = "unknown"
41         else:
42             # make sure to not get an index out of bounds.
43             if int(self.devcats, 16) > len(self.devcatslookuptable):
44                 self.devcatsdescription = "unknown"
45             else:
46                 self.devcatsdescription = self.devcatslookuptable[int(self.devcats, 16)]
47
48     def printdevice(self):
49         print "DevID: " + self.getdevid()
50         print "\t category: " + self.getdevcategory() + " -> " +
51             str(self.getdevcatsdescription())
52         print "\t subcategory: " + self.getdevsubcategory()
53         print "\t firmware: " + self.getfirmware()
54         print "\t status: " + self.getstatus()
55         if self.getisresponder():
56             print "\t respondsto: " + str(self.getrespondstolist())
57
58         if self.getiscontroller():
59             print "\t controls: " + str(self.getcontrolslist())
60
61         print "\t temp: detected " + str(hex(self.getcurrenttemp())) + " -> " +
```

```

62         hex(2*int(self.getdisplaytemp())) + " -> " + self.getdisplaytemp()
63     print "\t humidity: " + self.gethumidity()
64     print "\t mode: " + str(self.getextendedmode())
65     print "\t fan status: " + str(self.getfanstatus())
66     print "\t heat mode: " + str(self.getheatonoff())
67     print "\t cool mode: " + str(self.getcoolonoff())
68     print "\t cool set point: " + str(hex(int(str(self.getcoolsetpoint())))) +
69         " -> " + str(self.getcoolsetpoint())
70     print "\t heat set point: " + str(hex(int(str(self.getheatsetpoint())))) +
71         " -> " + str(self.getheatsetpoint())
72     print "\t flags: " + str(self.updateflags())
73     print "\t fan active: " + str(self.getfanactive())
74     print "\t cool active: " + str(self.getcoolactive())
75     print "\t heat active: " + str(self.getheatactive())
76     print "\t last time: " + str(asctime(localtime(self.getlasttime())))
77     print "\t curr time: " + str(asctime(localtime(time())))
78
79     def getcoolonoff(self):
80         return self.coolonoff
81
82     def getfanactive(self):
83         return self.fanactive
84
85     def getcoolactive(self):
86         return self.coolactive
87
88     def getheatactive(self):
89         return self.heatactive
90
91     def setfanactive(self, active):
92         if active == "True":
93             self.fanactive = True
94         else:
95             self.fanactive = False
96
97     def setcoolactive(self, active):
98         if active == "True":
99             self.coolactive = True
100         else:
101             self.coolactive = False
102
103     def setheatactive(self, active):
104         if active == "True":
105             self.heatactive = True
106         else:
107             self.heatactive = False
108
109     def updatethermostatdetails(self):
110         # use this function to update info.
111         # things like modifying temp.
112         # turning the fan / active cooling / active heating on off
113         # based on temps.
114
115         displaytemp = int(self.getdisplaytemp())
116
117         coolsetpoint = self.getcoolsetpoint() # the decimal version of coolsetpoint
118         heatsetpoint = self.getheatsetpoint() # the decimal version of heatsetpoint
119         time2 = time()
120         time1 = self.getlasttime()
121         coolonoff = self.getcoolonoff()
122         heatonoff = self.getheatonoff()
123         fanonauto = self.getfanstatus()
124
125         heattime = 30 # seconds. used to simulate the room heating/cooling
126         cooltime = 30
127

```

```

128     if coolonoff and displaytemp > coolsetpoint:
129         # cool mode is on but the temp is higher then the coolsetpoint.
130         # turn on the fan, and turn on the cool.
131         self.setcoolactive("True")
132         self.setfanactive("True")
133
134         timediff = int(time2 - time1)
135
136         # now, if the cooler has been going for a while...
137         # figure out how long it's been going for.
138         # say 30 seconds changes the temp 1 hex degree.
139         if timediff > cooltime:
140             multiple = int(timediff / cooltime)
141             if displaytemp - multiple >= coolsetpoint:
142                 self.setcurrenttemp(self.getcurrenttemp() - multiple)
143                 self.setlasttime(time())
144             else:
145                 self.setcurrenttemp(self.getcurrenttemp() -
146                                     (displaytemp - coolsetpoint))
147                 self.setlasttime(time())
148
149         elif coolonoff and displaytemp <= coolsetpoint:
150             # self.setcurrenttemp(self.getcurrenttemp())
151             if self.getcoolactive():
152                 self.setcoolactive("False")
153                 self.setlasttime(time())
154
155             if not self.getfanstatus() and self.getfanactive():
156                 # this means if fan auto mode but the fan is still running.
157                 self.setfanactive("False")
158                 self.setlasttime(time())
159
160     if heatonoff and displaytemp < heatsetpoint:
161         self.setheatactive("True")
162         # self.setfanactive("True")
163
164         timediff = int(time2 - time1)
165
166         if timediff > heattime:
167             multiple = int(timediff / heattime)
168             if displaytemp + multiple <= heatsetpoint:
169                 self.setcurrenttemp(self.getcurrenttemp() + multiple)
170                 self.setlasttime(time())
171             else:
172                 self.setcurrenttemp(self.getcurrenttemp() +
173                                     (heatsetpoint - displaytemp))
174                 self.setlasttime(time())
175
176         elif heatonoff and displaytemp >= heatsetpoint:
177             # this means we need to turn off the stuff.
178             if self.getheatactive():
179                 self.setheatactive("False")
180                 self.setlasttime(time())
181
182             if not self.getfanstatus() and self.getfanactive():
183                 pass
184             # self.setfanactive("False")
185             # self.setlasttime(time())
186
187     if not heatonoff:
188         self.setheatactive("False")
189
190     if not coolonoff:
191         self.setcoolactive("False")
192
193     self.updateflags()

```

```

194
195 def setcoolonoff(self, cool):
196     if cool == "True":
197         self.coolonoff = True
198     else:
199         self.coolonoff = False
200
201 def getheatonoff(self):
202     return self.heatonoff
203
204 def setheatonoff(self, heat):
205     if heat == "True":
206         self.heatonoff = True
207     else:
208         self.heatonoff = False
209
210 def setfanstatus(self, status):
211     if status == "True":
212         self.fanstatus = True
213     else:
214         self.fanstatus = False
215
216 def getfanstatus(self):
217     return self.fanstatus
218
219 def setlasttime(self, t):
220     self.lasttimestamp = t
221
222 def getlasttime(self):
223     return self.lasttimestamp
224
225 def getdevcatdescription(self):
226     return self.devcatdescription
227
228 def getdisplaytemp(self):
229     temp = str(hex(self.getcurrenttemp()))
230     firstnib = int(temp[2], 16)
231     secondnib = int(temp[3], 16)
232     # newtemp = int(temp, 16)
233     if 10 <= secondnib <= 15:
234         if firstnib <= 14:
235             firstnib += 1
236             secondnib = 0
237
238     newtempstr = str(firstnib) + str(secondnib)
239     return str(int(floor(int(newtempstr) / 2)))
240
241 def updateflags(self):
242     flags = 00
243     if self.getfanactive():
244         flags += 64
245
246     if self.getcoolactive():
247         flags += 1
248
249     if self.getheatactive():
250         flags += 72
251
252     if not self.getfanstatus() and not self.getcoolactive()
253     and not self.getheatactive():
254
255         flags += 128
256
257     return hex(flags)
258
259 def setcurrenttemp(self, temp):

```

```

260         self.currenttemp = temp
261
262     def getcurrenttemp(self):
263         return self.currenttemp
264
265     def sethumidity(self, humidity):
266         self.humidity = humidity
267
268     def gethumidity(self):
269         return self.humidity
270
271     def getleaveonoff(self):
272         return self.leaveonoff
273
274     def getextendedmode(self):
275         mode = 00
276         fanyesno = self.getfanstatus()
277         heatyesno = self.getheatonoff()
278         coolyesno = self.getcoolonoff()
279         leaveyesno = self.getleaveonoff()
280
281         if fanyesno:
282             mode += 01
283
284         if heatyesno and not coolyesno:
285             mode += 32
286
287         if coolyesno and not heatyesno:
288             mode += 48
289
290         if heatyesno and coolyesno:
291             if leaveyesno:
292                 mode += 64
293             else:
294                 mode += 16
295
296         return hex(mode)
297
298     def getshortstatus(self):
299         heatyesno = self.getheatonoff()
300         coolyesno = self.getcoolonoff()
301
302         if heatyesno and not coolyesno:
303             return "01"
304         elif coolyesno and not heatyesno:
305             return "01"
306         elif heatyesno and coolyesno:
307             return "03"
308         else:
309             return "00"
310
311     def setcoolsetpoint(self, coolsetpoint):
312         self.currentcoolsetpoint = coolsetpoint
313
314     def getcoolsetpoint(self):
315         return self.currentcoolsetpoint
316
317     def setheatsetpoint(self, heatsetpoint):
318         self.currentheatsetpoint = heatsetpoint
319
320     def getheatsetpoint(self):
321         return self.currentheatsetpoint
322
323     def setfirmware(self, firmware):
324         self.firmware = firmware
325

```

```

326     def setdevcategory(self, devcat):
327         self.devcat = devcat
328
329     def setdevsubcategory(self, devsubcat):
330         self.devsubcat = devsubcat
331
332     def setisresponder(self, isresponder):
333         self.isresponder = isresponder
334
335     def setiscontroller(self, iscontroller):
336         self.iscontroller = iscontroller
337
338     def getdevcategory(self):
339         return self.devcat
340
341     def getdevsubcategory(self):
342         return self.devsubcat
343
344     def getfirmware(self):
345         return self.firmware
346
347     def getdevid(self):
348         return self.devid
349
350     def setstatus(self, status):
351         self.status = status
352
353     def getstatus(self):
354         return self.status
355
356     def getisresponder(self):
357         return self.isresponder
358
359     def getiscontroller(self):
360         return self.iscontroller
361
362     def addresponder(self, address):
363         self.respondsto.append(address)
364
365     def getrespondstolist(self):
366         return self.respondsto
367
368     def addcontroller(self, address):
369         self.controls.append(address)
370
371     def getcontrolslist(self):
372         return self.controls
373     devcatlookuptable = ["General Controller", "Dimmable Lighting Control",
374     "Switched Lighting Control", "Network Bridge",
375     "Irrigation Control", "Climate Control", "Pool and Spa control",
376     "Sensors and Actuators", "Home Entertainment",
377     "Energy Management", "Built-in Appliance",
378     "Plumbing", "Communication", "Computer Control",
379     "Window Coverings", "Access Control",
380     "Security, Health, Safety", "Surveillance",
381     "Automotive", "other"]

```

A.12 insteonhoney.py

```
1 #!/usr/bin/env python
2 """
3 This program will be the controller for an Insteon Honey Pot. There are 2 goals:
4
5 1) Create simulate fake traffic to confuse a would-be attacker
6    and obfuscate my real devices. (completed by insteonsimtraffic.py)
7 2) Allow the attacker to interact with my fake devices.
8    The attacker should think that he's turning the lights on and off, etc.
9
10 """
11
12 from sys import *
13 from insteonhoneydevices import *
14 from insteonthermostat import *
15 from insteonlisten import *
16 from rfcattmod import *
17 import time
18 import random
19 import os
20 import argparse
21
22
23 def parseargs():
24     """
25     :return: the arguments.
26     """
27     parser = argparse.ArgumentParser(description='Create simulated Insteon devices.')
28     parser.add_argument('-iL', '--interface1',
29         help='USB interface to listen', default=0)
30     parser.add_argument('-iS', '--interface2',
31         help='USB interface to listen', default=1)
32     parser.add_argument('-f', '--filename', help='the filename that
33         contains the simulated devices.',
34         default="insteonsimdevices.txt")
35     return parser.parse_args()
36
37
38 def parsedevicesfile(inputfilename):
39     # print "parsing..."
40     try:
41         inputfile = open(inputfilename, "r")
42     except IOError:
43         print "insteonsimdevices.txt file does not exist.
44             Either create it or specify another filename to use."
45         sys.exit(0)
46
47     devicelist = list()
48
49     for line in inputfile:
50
51         line = line.strip("\n")
52         if line == "Device":
53
54             device = None
55             deviceid = next(inputfile).strip("\n")
56             category = next(inputfile).strip("\n")
57             if category == "01" or category == "03" or category == "0F":
58                 device = FakeSimpleDevice(deviceid)
59                 device.setdevcategory(category)
60                 # print device.getdevid()
61                 # print device.getdevcategory()
```

```

62
63     subcategory = next(inputfile).strip("\n")
64     device.setdevsubcategory(subcategory)
65     # print device.getdevsubcategory()
66
67     firmware = next(inputfile).strip("\n")
68     device.setfirmware(firmware)
69     # print device.getfirmware()
70
71     status = next(inputfile).strip("\n")
72     device.setstatus(status)
73
74     line = next(inputfile).strip("\n")
75     if line == "isresponderyes":
76         device.setisresponder(True)
77         y = int(next(inputfile).strip("\n"))
78         for x in range(0, y):
79             responderaddress = next(inputfile).strip("\n")
80             device.addresponder(responderaddress)
81     elif line == "isresponderno":
82         device.setisresponder(False)
83
84     line = next(inputfile).strip("\n")
85     if line == "iscontrolleryes":
86         device.setiscontroller(True)
87         y = int(next(inputfile).strip("\n"))
88         for x in range(0, y):
89             controlleraddress = next(inputfile).strip("\n")
90             device.addcontroller(controlleraddress)
91
92     elif line == "iscontrollerno":
93         device.setiscontroller(False)
94
95 elif category == "05": # time to parse the thermostat
96     # print "here"
97     # initialize and set basic information
98     device = FakeThermostat(deviceid)
99     device.setdevcategory(category)
100    subcategory = next(inputfile).strip("\n")
101    device.setdevsubcategory(subcategory)
102    # print device.getdevsubcategory()
103
104    firmware = next(inputfile).strip("\n")
105    device.setfirmware(firmware)
106    # print device.getfirmware()
107
108    status = next(inputfile).strip("\n")
109    device.setstatus(status)
110
111    # get network relationship information
112    line = next(inputfile).strip("\n")
113    if line == "isresponderyes":
114        device.setisresponder(True)
115        y = int(next(inputfile).strip("\n"))
116        for x in range(0, y):
117            responderaddress = next(inputfile).strip("\n")
118            device.addresponder(responderaddress)
119
120    elif line == "isresponderno":
121        device.setisresponder(False)
122
123    line = next(inputfile).strip("\n")
124    if line == "iscontrolleryes":
125        device.setiscontroller(True)
126        y = int(next(inputfile).strip("\n"))
127        for x in range(0, y):

```

```

128         controlleraddress = next(inputfile).strip("\n")
129         device.addcontroller(controlleraddress)
130
131     elif line == "iscontrollerno":
132         device.setiscontroller(False)
133
134     currenttemp = next(inputfile).strip("\n")
135     device.setcurrenttemp(int(currenttemp, 16))
136
137     humidity = next(inputfile).strip("\n")
138     device.sethumidity(humidity)
139
140     fanstate = next(inputfile).strip("\n")
141     device.setfanstatus(bool(fanstate))
142
143     heatyesno = next(inputfile).strip("\n")
144     device.setheatonoff(heatyesno)
145
146     coolyesno = next(inputfile).strip("\n")
147     device.setcoolonoff(coolyesno)
148
149     # device.initmode()
150     # device.updatecurrentmode()
151
152     coolsetpoint = next(inputfile).strip("\n")
153     device.setcoolsetpoint(int(coolsetpoint, 16))
154
155     heatsetpoint = next(inputfile).strip("\n")
156     device.setheatsetpoint(int(heatsetpoint, 16))
157
158     device.updatethermostatdetails()
159
160     device.devcatlookup()
161     devicelist.append(device)
162     return devicelist
163
164 def printdevices(devices): # a good functions for debugging the simdevices file.
165     print "-----"
166     for device in devices:
167         device.printdevice()
168     print "-----"
169
170 def updatescreen(devicelist):
171     """
172     might be a simple function, but could be helpful in the future.
173     :param devicelist: the global devicelist.
174     :return: n/a
175     """
176     # found this line online somewhere. it works great!
177     os.system('cls' if os.name == 'nt' else 'clear')
178     printdevices(devicelist)
179
180 def getstringofcurrenttime():
181     currtime = str(asctime(localtime(time.time())))
182     strday = None
183     if currtime[0:3] == "Sun":
184         strday = "00"
185     elif currtime[0:3] == "Mon":
186         strday = "01"
187     elif currtime[0:3] == "Tue":
188         strday = "02"
189     elif currtime[0:3] == "Wed":
190         strday = "03"
191     elif currtime[0:3] == "Thu":
192         strday = "04"
193     elif currtime[0:3] == "Fri":

```

```

194     strday = "05"
195 elif currrtime[0:3] == "Sat":
196     strday = "06"
197
198 strhour = currrtime[11:13]
199 inthour = int(strhour)
200 if 0 <= inthour <= 15:
201     strhour = "0" + str(hex(inthour))[2:3]
202 elif 15 < inthour <= 24:
203     strhour = str(hex(inthour))[2:4]
204
205 strmin = currrtime[14:16]
206 intmin = int(strmin)
207 if 0 <= intmin <= 15:
208     strmin = "0" + str(hex(intmin))[2:3]
209 elif 15 < intmin <= 60:
210     strmin = str(hex(intmin))[2:4]
211
212 strsec = currrtime[17:19]
213 intsec = int(strsec)
214 if 0 <= intsec <= 15:
215     strsec = "0" + str(hex(intsec))[2:3]
216 elif 15 < intsec <= 60:
217     strsec = str(hex(intsec))[2:4]
218
219 return strday + " " + strhour + " " + strmin + " " + strsec
220
221 def honeypot(devices, packet, senderinterface):
222     # make sure line is set every time.
223     # this is going to be the start of my response.
224     line = "./insteonsend.py -i " + str(senderinterface) + " -r "
225
226     destination = packet[3:11]
227     destinationindex = searchdevicelist(devices, destination)
228     source = packet[12:20]
229     sourceindex = searchdevicelist(devices, source)
230     command1 = packet[21:23]
231     command2 = packet[24:26]
232     # the crc is assumed correct at this point.
233
234     category = None
235     device = None
236     if destinationindex:
237         device = devices[destinationindex]
238         category = device.getdevcategory()
239
240     if packet[0] == "0": # a standard direct command.
241         if category == "01": # these are the commands the "dimmable light control"
242             # but the light doesn't care if it's standard or extended,
243             # it treats the commands the same.
244             if source in device.getrespondstolist():
245                 if command1 == "0F": # ping command1
246                     line += "2F : " + source + " : " + destination +
247                         " : " + "0F " + command2
248                 # print line
249                 os.system(line)
250
251         elif command1 == "10": # ID request commands
252             # an ID request.
253             # need to respond with an acknowledgement, then provide the device info.
254             line += "2F : " + source + " : " + destination +
255                 " : " + command1 + " " + command2
256             # print line
257             os.system(line)
258             line = "./insteonsend.py -i " + str(senderinterface) + " -r "
259             line += "8F : " + destination + " : " + device.getfirmware() + " " +

```

```

260         device.getdevsubcategory() + " " + device.getdevcategory() +
261         " : " + "01 00"
262     # print line
263     os.system(line)
264
265 elif command1 == "11": # slow on command. use command 2 for status.
266     # turn device on and reply
267     device.setstatus(command2)
268     line += "2F : " + source + " : " + destination +
269     " : " + command1 + " " + command2
270     # print line
271     os.system(line)
272
273 elif command1 == "12": # fast on. don't worry about command2
274     device.setstatus("FF")
275     line += "2F : " + source + " : " + destination +
276     " : " + command1 + " " + device.getstatus()
277     # print line
278     os.system(line)
279
280 elif command1 == "13":
281     device.setstatus("00")
282     line += "2F : " + source + " : " + destination +
283     " : " + command1 + " " + device.getstatus()
284     os.system(line)
285
286 elif command1 == "14": # turn off
287     # turn device off and reply
288     device.setstatus("00")
289     line += "2F : " + source + " : " + destination +
290     " : " + command1 + " " + device.getstatus()
291     # print line
292     os.system(line)
293
294 elif command1 == "15":
295     # step the dimmer up by 8.
296     status = int(device.getstatus(), 16)
297     if status <= 247:
298         status += 8
299     else:
300         status = 255
301         device.setstatus(str(hex(status)))
302     line += "2F : " + source + " : " + destination +
303     " : " + command1 + " " + command2
304     os.system(line)
305
306 elif command1 == "16":
307     # step the dimmer down by 8.
308     status = int(device.getstatus(), 16)
309     if status >= 8:
310         status -= 8
311     else:
312         status = 0
313         device.setstatus(str(hex(status)))
314     line += "2F : " + source + " : " + destination +
315     " : " + command1 + " " + command2
316     os.system(line)
317
318 elif command1 == "17":
319     if (int(command2, 16) % 2) == 0: # if command2 is even, light goes off.
320         device.setstatus("00")
321     else:
322         # if command2 is odd, light goes on.
323         device.setstatus("FF")
324
325     # simple response.

```

```

326         line += "2F : " + source + " : " + destination +
327             " : " + command1 + " " + command2
328         os.system(line)
329
330     # status request responds different if it's standard or extended.
331     elif command1 == "19":
332         # time to reply.
333         if command2 == "02": # a specific value for command2.
334             line += "2F : " + source + " : " + destination +
335                 " : " + command1 + " " + device.getstatus()
336             # print line
337             os.system(line)
338         else:
339             line += "2F : " + source + " : " + destination +
340                 " : " + "00" + " " + device.getstatus()
341             os.system(line)
342
343     elif command1 == "1F":
344         if command2 == "00":
345             # this is a get operating flags request per p. 5 of command tables.
346             # looks like a good response is 0x60.
347             line += "2F : " + source + " : " + destination +
348                 " : " + command1 + " " + "60"
349             os.system(line)
350         elif command2 == "01":
351             # this is asking if the destination is a controller.
352             # 0x01 = iscontrolleryes. 0x00 = iscontrollerno.
353             if device.getiscontroller():
354                 line += "2F : " + source + " : " + destination +
355                     " : " + command1 + " " + "01"
356             else:
357                 line += "2F : " + source + " : " + destination +
358                     " : " + command1 + " " + "00"
359             os.system(line)
360
361     else:
362         # if the light doesn't recognize the command,
363         # the command is basically like ping.
364         line += "2F : " + source + " : " + destination +
365             " : " + command1 + " " + command2
366         os.system(line)
367
368     else:
369         # this is for when the destination is not a responder to the source.
370         # It sends a NAK.
371         line += "AF : " + source + " : " + destination +
372             " : " + command1 + " FF"
373         os.system(line)
374
375     elif category == "03": # the network bridge
376         if source in device.getrespondstolist():
377             if command1 == "0F": # ping command1
378                 line += "2F : " + source + " : " + destination +
379                     " : " + "0F " + command2
380             # print line
381             os.system(line)
382
383     elif command1 == "10": # ID request commands
384         # an ID request.
385         # need to respond with an acknowledgement, then provide the device info.
386         line += "2F : " + source + " : " + destination +
387             " : " + command1 + " " + command2
388         # print line
389         os.system(line)
390         line = "./insteonsend.py -i " + str(senderinterface) + " -r "
391         line += "8F : " + destination + " : " + device.getfirmware() +

```

```

392         " " + device.getdevsubcategory() + " " +
393         device.getdevcategory() + " : " + "01 00"
394     # print line
395     os.system(line)
396
397 elif command1 == "19": # status request
398     # time to reply.
399     line += "2F : " + source + " : " + destination +
400     " : " + "19 " + device.getstatus()
401     # print line
402     os.system(line)
403
404 else: # unknown command, but valid source.
405     line += "2F : " + source + " : " + destination +
406     " : " + command1 + " " + command2
407     # print line
408     os.system(line)
409
410 elif category == "05": # climate control
411     # the thermostat responds to everything....
412     # if source in device.getrespondstolist():
413
414     # make sure thermostat "updates" to make time-sensitive decisions.
415     device.updatethermostatdetails()
416
417 if command1 == "0F": # ping command1
418     line += "2F : " + source + " : " + destination +
419     " : " + "0F " + command2
420     # print line
421     os.system(line)
422
423 elif command1 == "19": # status request
424     # time to reply.
425     line += "2F : " + source + " : " + destination +
426     " : " + "19 00"
427     # print line
428     os.system(line)
429
430 elif command1 == "10": # ID request commands
431     # an ID request.
432     # need to respond with an acknowledgement, then provide the device info.
433     line += "2F : " + source + " : " + destination + " : " +
434     command1 + " " + command2
435     # print line
436     os.system(line)
437     line = "./insteonsend.py -i " + str(senderinterface) + " -r "
438     line += "8F : " + destination + " : " + device.getfirmware() +
439     " " + device.getdevsubcategory() + " " + device.getdevcategory() +
440     " : " + "01 00"
441     # print line
442     os.system(line)
443
444 elif command1 == "6B":
445     if command2 == "00" or command2 == "01":
446         line += "2F : " + source + " : " + destination +
447         " : " + command1 + " " + command2
448         # print line
449         os.system(line)
450
451 elif command2 == "02":
452     line += "2F : " + source + " : " + destination +
453     " : " + command1 + " " + device.getshortstatus()
454     # print line
455     os.system(line)
456
457 elif command2 == "03":

```

```

458     line += "2F : " + source + " : " + destination + " : " +
459         command1 + " " + str(hex(2 * (int(device.getdisplaytemp()))))
460     # print line
461     os.system(line)
462     # time.sleep(2)
463
464     else:
465         line += "AF : " + source + " : " + destination + " : " +
466             command1 + " " + "fd"
467         # print line
468         os.system(line)
469
470     elif category == "0F": # an access controller
471         if source in device.getrespondstolist():
472             if command1 == "0F": # ping command1
473                 line += "2F : " + source + " : " + destination +
474                     " : " + "0F " + command2
475                 # print line
476                 os.system(line)
477
478             elif command1 == "10": # ID request commands
479                 # an ID request need to respond with an acknowledgement,
480                 # then provide the device info.
481                 line += "2F : " + source + " : " + destination +
482                     " : " + command1 + " " + command2
483                 # print line
484                 os.system(line)
485                 line = "./insteonsend.py -i " + str(senderinterface) + " -r "
486                 line += "8F : " + destination + " : " + device.getfirmware() +
487                     " " + device.getdevsubcategory() + " " +
488                     device.getdevcategory() + " : " + "01 00"
489                 # print line
490                 os.system(line)
491
492             elif command1 == "13" or command1 == "14": # turn off
493                 # turn device off and reply
494                 devices[destindex].setstatus("00")
495                 line += "2F : " + source + " : " + destination +
496                     " : " + command1 + " " + command2
497                 # print line
498                 os.system(line)
499
500             elif command1 == "11": # slow on command. use command 2 for status.
501                 # turn device on and reply
502                 device.setstatus(command2)
503                 line += "2F : " + source + " : " + destination +
504                     " : " + command1 + " " + command2
505                 # print line
506                 os.system(line)
507
508             elif command1 == "12": # fast on. don't worry about command2
509                 device.setstatus("FF")
510                 line += "2F : " + source + " : " + destination +
511                     " : " + command1 + " " + command2
512                 # print line
513                 os.system(line)
514
515         # status request responds different if it's standard or extended.
516         elif command1 == "19":
517             # time to reply.
518             # a standard message get's a different response then an extended
519             if packet[0] == "0":
520                 if command2 == "02": # a specific value for command2.
521                     line += "2F : " + source + " : " + destination +
522                         " : " + command1 + " " + device.getstatus()
523                     # print line

```

```

524         os.system(line)
525     else:
526         line += "2F : " + source + " : " + destination +
527             " : " + "00" + " " + device.getstatus()
528         os.system(line)
529     elif packet[0] == "1":
530         line += "AF : " + source + " : " + destination +
531             " : " + command1 + " FD"
532         os.system(line)
533     elif command1 == "1F": # Get operating flags request.
534         if command2 == "00":
535             # this is a get operating flags request per p. 5 of command tables.
536             # looks like a good response is 0x60.
537             line += "2F : " + source + " : " + destination +
538                 " : " + command1 + " " + "60"
539             os.system(line)
540         elif command2 == "01":
541             # pass
542             # this is asking if the destination is a controller. 0x01 =
iscontrolleryes. 0x00 = iscontrollerno.
543             if device.getiscontroller():
544                 line += "2F : " + source + " : " + destination +
545                     " : " + command1 + " " + "01"
546             else:
547                 line += "2F : " + source + " : " + destination +
548                     " : " + command1 + " " + "00"
549         else: # unrecognized command.
550             # if the lock controller doesn't recognize the command,
551             # the command is basically like ping.
552             line += "2F : " + source + " : " + destination +
553                 " : " + command1 + " " + command2
554             os.system(line)
555             os.system(line)
556         else: # when the source address does not control the lock.
557             # NAK
558             line += "AF : " + source + " : " + destination +
559                 " : " + command1 + " FF"
560             os.system(line)
561     else: # TODO other device categories
562         pass
563
564     elif packet[0] == "1": # extended direct messages.
565         if category == "01": # these are the commands the "dimmmable light control"
566             # but the light doesn't care if it's standard or extended,
567             # it treats most commands the same.
568             if source in device.getrespondstolist():
569                 if command1 == "0F": # ping command1
570                     line += "2F : " + source + " : " + destination +
571                         " : " + "0F " + command2
572                     # print line
573                     os.system(line)
574
575                 elif command1 == "11": # slow on command. use command 2 for status.
576                     # turn device on and reply
577                     device.setstatus(command2)
578                     line += "2F : " + source + " : " + destination +
579                         " : " + command1 + " " + command2
580                     # print line
581                     os.system(line)
582
583                 elif command1 == "12": # fast on. don't worry about command2
584                     device.setstatus("FF")
585                     line += "2F : " + source + " : " + destination +
586                         " : " + command1 + " " + device.getstatus()
587                     # print line
588                     os.system(line)

```

```

589
590 elif command1 == "13":
591     device.setstatus("00")
592     line += "2F : " + source + " : " + destination +
593           " : " + command1 + " " + device.getstatus()
594     os.system(line)
595
596 elif command1 == "14": # turn off
597     # turn device off and reply
598     device.setstatus("00")
599     line += "2F : " + source + " : " + destination +
600           " : " + command1 + " " + device.getstatus()
601     # print line
602     os.system(line)
603
604 elif command1 == "15":
605     # step the dimmer up by 8.
606     status = int(device.getstatus(), 16)
607     if status <= 247:
608         status += 8
609     else:
610         status = 255
611         device.setstatus(str(hex(status)))
612     line += "2F : " + source + " : " + destination +
613           " : " + command1 + " " + command2
614     os.system(line)
615
616 elif command1 == "16":
617     # step the dimmer down by 8.
618     status = int(device.getstatus(), 16)
619     if status >= 8:
620         status -= 8
621     else:
622         status = 0
623         device.setstatus(str(hex(status)))
624     line += "2F : " + source + " : " + destination +
625           " : " + command1 + " " + command2
626     os.system(line)
627
628 elif command1 == "17":
629     if (int(command2, 16) % 2) == 0: # if command2 is even,
630     # light goes off.
631         device.setstatus("00")
632     else:
633         # if command2 is odd, light goes on.
634         device.setstatus("FF")
635
636     # simple response.
637     line += "2F : " + source + " : " + destination +
638           " : " + command1 + " " + command2
639     os.system(line)
640
641 # status request responds different if it's standard or extended.
642 elif command1 == "19":
643     # time to reply.
644     line += "AF : " + source + " : " + destination +
645           " : " + command1 + " FD"
646     os.system(line)
647
648 else: # unknown extended command.
649     # NAK
650     line += "AF : " + source + " : " + destination +
651           " : " + command1 + " FD"
652     os.system(line)
653 else: # un-linked device:
654     line += "AF : " + source + " : " + destination +

```

```

655         " : " + command1 + " FF"
656         os.system(line)
657
658     elif category == "03": # the network bridge
659         if source in device.getrespondstolist():
660             if command1 == "10":
661                 # an ID request.
662                 # need to respond with an acknowledgement,
663                 # then provide the device info.
664                 line += "2F : " + source + " : " + destination +
665                     " : " + command1 + " " + command2
666                 # print line
667                 os.system(line)
668                 line = "./insteonsend.py -i " + str(senderinterface) + " -r "
669                 line += "8F : " + destination + " : " + device.getfirmware() +
670                     " " + device.getdevsubcategory() + " " +
671                     device.getdevcategory() + " : " + "01 00"
672                 # print line
673                 os.system(line)
674             elif command1 == "19":
675                 line += "2F : " + source + " : " + destination +
676                     " : " + "00" + " " + "01"
677                 os.system(line)
678             else:
679                 # device seems to respond to pretty much every command with an ACK.
680                 line += "2F : " + source + " : " + destination +
681                     " : " + command1 + " " + command2
682                 os.system(line)
683         else: # when the source address does not control the network bridge.
684             line += "AF : " + source + " : " + destination +
685                 " : " + command1 + " FF"
686             os.system(line)
687
688     elif category == "05": # climate control
689         user1 = packet[66:68]
690         user2 = packet[63:65]
691         # if source in device.getrespondstolist():
692         # thermostat responds to everything...
693         if command1 == "2E" and command2 == "02"
694             and user1 == "96" and user2 == "92":
695
696             # long status request. send back the initial reply.
697             line += "2F : " + source + " : " + destination +
698                 " : " + command1 + " " + command2
699             # print line
700             os.system(line)
701
702             strtime = getstringofcurrenttime()
703
704             # now reply with the full status.
705             line = "./insteonsend.py -i " + str(senderinterface) + " -r "
706             line += "1F " + " : " + source + " : " + destination +
707                 " : " + command1 + " " + command2
708             line += " 01" + " " + strtime + " "
709
710             line += str(device.getextendedmode()) + " " +
711                 str(hex(device.getcoolsetpoint())) + " " +
712                 str(device.gethumidity()) + " "
713
714             line += "00 " + str(hex(device.getcurrenttemp())) + " " +
715                 str(device.updateflags()) + " " +
716                 str(hex(device.getheatsetpoint())) + " "
717             line += str(hex(random.randint(0, 255))) + " " +
718                 str(hex(random.randint(0, 255)))
719
720             # str(hex(random.randint(0, 255)))

```

```

721         os.system(line)
722
723     elif command1 == "6B" and command2 == "09"
724         and user1 == "8C": # this command shuts off the thermostat.
725
726         device.setcoolonoff("False")
727         device.setheatonoff("False")
728         device.setfanstatus("False")
729
730         device.setlasttime(time.time())
731         line += "2F : " + source + " : " + destination +
732             " : " + command1 + " " + command2
733         # print line
734         os.system(line)
735
736     elif command1 == "6B" and command2 == "08" and
737         user1 == "8D": # this message as to turn the fan off.
738
739         device.setfanstatus("False")
740         device.setlasttime(time.time())
741         line += "2F : " + source + " : " + destination +
742             " : " + command1 + " " + command2
743         # print line
744         os.system(line)
745
746     elif command1 == "6B" and command2 == "07"
747         and user1 == "8E": # this message is to turn the fan on.
748
749         device.setfanstatus("True")
750         device.setlasttime(time.time())
751         line += "2F : " + source + " : " + destination +
752             " : " + command1 + " " + command2
753         # print line
754         os.system(line)
755
756     elif command1 == "6B" and command2 == "05"
757         and user1 == "90": # this message turns the cold on.
758
759         device.setcoolonoff("True")
760         device.setlasttime(time.time())
761
762         if device.getheatonoff():
763             # if heat was on (either auto mode or heat only mode)
764             device.setheatonoff("False")
765
766         line += "2F : " + source + " : " + destination +
767             " : " + command1 + " " + command2
768         # print line
769         os.system(line)
770
771     elif command1 == "6B" and command2 == "04"
772         and user1 == "91": # this message turns heat mode on.
773
774         device.setheatonoff("True")
775         device.setlasttime(time.time())
776
777         if device.getcoolonoff():
778             # if cold mode was on (either auto or cold only mode)
779             device.setcoolonoff("False")
780         line += "2F : " + source + " : " + destination +
781             " : " + command1 + " " + command2
782         # print line
783         os.system(line)
784
785     elif command1 == "6B" and command2 == "06"
786         and user1 == "8F": # this message turns auto mode on.

```

```

787     # I first need to figure out what to do with the setpoints temps.
788
789     if device.getcoolonoff() and not device.getheatonoff():
790         # cold mode only.
791         # if going from cold to auto, need to make sure that
792         # hot is set 2 below cold.
793         if device.getheatsetpoint() >= device.getcoolsetpoint() - 2:
794             device.setheatsetpoint(device.getcoolsetpoint() - 2)
795
796     if device.getheatonoff() and not device.getcoolonoff():
797         # heat mode only.
798         # if going from hot to auto, make sure that cold is set 2 above heat.
799         if device.getcoolsetpoint() <= device.getheatsetpoint() + 2:
800             device.setcoolsetpoint(device.getheatsetpoint() + 2)
801
802     device.setheatonoff("True")
803     device.setcoolonoff("True")
804     device.setlasttime(time.time())
805     line += "2F : " + source + " : " + destination +
806           " : " + command1 + " " + command2
807     # print line
808     os.system(line)
809
810 elif command1 == "6C":
811     # set the coolsetpoint for the thermostat
812     value1 = int(command2, 16)
813     value2 = int(user1, 16)
814     value3 = 0 # initialize so the IDE doesn't freak out.
815     # not sure why they chose this, but...whatevs.
816     if value1 <= 148:
817         value3 = value1 + value2
818     elif value1 > 148:
819         value2 = 256 - int(user1, 16)
820         value3 = value1 - value2
821
822
823     # if device.getcoolonoff():
824     # doesn't matter if the device is in cold mode or not.
825     # The coolsetpoint can be set.
826     if value3 == 148: # 0x94
827         if 74 <= value1 <= 194:
828             # the thermostat coolsetpoint can be between. 37-97
829             newint = int(value1 / 2)
830             device.setcoolsetpoint(newint)
831             device.setlasttime(time.time())
832
833         if device.getheatonoff():
834             # means auto, so make sure heat lower then cool.
835             if device.getheatsetpoint() >= device.getcoolsetpoint() - 2:
836                 # something like this...
837                 device.setheatsetpoint(device.getcoolsetpoint() - 2)
838
839     # as long as value3 is correct, send a good response.
840     line += "2F : " + source + " : " + destination +
841           " : " + command1 + " " + command2
842     # print line
843     os.system(line)
844     else: # values weren't valid, so NAK.
845         line += "AF : " + source + " : " + destination +
846               " : " + command1 + " " + "FD"
847         # print line
848         os.system(line)
849
850 elif command1 == "6D":
851     # set the heatsetpoint for the thermostat
852     value1 = int(command2, 16)

```

```

853     value2 = int(user1, 16)
854     value3 = 0 # initialize so the IDE doesn't freak out.
855     if value1 <= 147:
856         value3 = value1 + value2
857     elif value1 > 147:
858         value2 = 256 - int(user1, 16)
859         value3 = value1 - value2
860
861     if value3 == 147: # 0x93 # no idea why they picked this. So weird.
862         if 70 <= value1 <= 190:
863             newint = int(floor(value1 / 2))
864             device.setheatsetpoint(newint)
865             device.setlasttime(time.time())
866
867             if device.getcoolonoff(): # means auto mode.
868                 # so make sure the cool is set higher then heat.
869                 if device.getcoolsetpoint() <= device.getheatsetpoint() + 2:
870                     device.setcoolsetpoint(device.getheatsetpoint() + 2)
871
872             # respond as long as the values are valid.
873             # But only set the temp if it's within the range.
874             line += "2F : " + source + " : " + destination +
875                   " : " + command1 + " " + command2
876             # print line
877             os.system(line)
878
879         else: # the values were not valid, so it's a NAK.
880             line += "AF : " + source + " : " + destination +
881                   " : " + command1 + " " + "FD"
882             # print line
883             os.system(line)
884
885         else: # unrecognized extended command.
886             pass # TODO: Fill in what the thermostat does
887             # with an unrecognized extended command.
888     else:
889         pass # TODO: Can Add more categories of devices.
890 else:
891     pass # TODO: Add more message types. I.E. does anything
892     # happen when a door broadcasts that it's opened/closed.
893
894 def searchdevicelist(devicelist, address):
895     """
896     # a simple search function to find an address in the saved device list.
897     :param devicelist: the devicelist to be searched
898     :param address: the address i'm looking for.
899     :return: if the device exists, return True and the device index in the list.
900             otherwise, False and None.
901     """
902     deviceindex = 0
903     for device in devicelist:
904         if device.getdevid() == address:
905             return True, deviceindex
906         else:
907             pass
908         deviceindex += 1
909     return False, None
910
911 def listen(devices, yardsticklistener, senderinterface):
912     while not keystop():
913         try:
914             rawdata, timestamp = yardsticklistener.RFrecv(timeout=2000)
915             timestampstring = str(asctime(localtime(timestamp)))
916             _bytes_ = ''.join("{0:08b}".format(x) for x in bytearray(rawdata))
917
918             subpackets = parse_insteon_pkt(_bytes_, timestampstring)

```

```

919
920     # this next portion is needed to make the honeypot more realistic.
921     # My device should only respond to
922     # the first correct message received in a string of commands.
923     # Thanks to the protocol hopping,
924     # I may hear multiple versions of the same command hopping across
925     # the devices, so I need to only take the first version of the command.
926     # If the next command is not the same as the previous, then
927     # respond to that one as well.
928     if subpackets is not None:
929         if subpackets[0].validcrc == 1:
930             firstpacket = subpackets[0].hex_str
931             # no matter what, respond to the first packet.
932             honeypot(devices, firstpacket, senderinterface)
933             # if there is only 1 packet, then this for loop won't execute.
934             for x in range(1, len(subpackets)):
935                 if subpackets[x].validcrc == 1:
936                     nextpacket = subpackets[x].hex_str
937                     if nextpacket is not None:
938                         dst1 = firstpacket[3:11]
939                         src1 = firstpacket[12:20]
940                         cmd1 = firstpacket[21:23]
941                         dst2 = nextpacket[3:11]
942                         src2 = nextpacket[12:20]
943                         cmd2 = nextpacket[21:23]
944                         # if the details of nextpacket is different then
945                         # firstpacket, then respond as well.
946                         if dst1 != dst2 or src1 != src2 or cmd1 != cmd2:
947                             honeypot(devices, nextpacket, senderinterface)
948
949         # updatescreen(devices)
950         print "InsteHoney running."
951     except KeyboardInterrupt:
952         print "Please press <enter> to stop"
953
954     except IndexError:
955         print "Index fail"
956
957     except ChipconUsbTimeoutException:
958         print "ChipconUsbTimeoutException"
959
960     finally:
961         pass
962
963     yardsticklistener.setModeIDLE()
964
965 def main():
966     args = parseargs()
967     inputfilename = args.filename
968     listenerinterface = int(args.interface1)
969     senderinterface = int(args.interface2)
970     # print inputfilename
971     # print interface
972     if inputfilename is None:
973         print "must specify an input file name"
974         sys.exit(0)
975
976     devices = parsedevicesfile(inputfilename)
977
978     printdevices(devices)
979
980     # once i setup some devices, I need to listen for when those devices are queried
981     yardsticklistener = configure_rfcats(listenerinterface)
982     # yardsticksender = configure_rfcats(senderinterface)
983     listen(devices, yardsticklistener, senderinterface)
984

```

```
985
986 if __name__ == "__main__":
987     main()
988     exit(1)
```

A.13 simtraffic.py

```
1 #!/usr/bin/env python
2 """
3 @Description: This program will be the controller for an Insteon Honey Pot.
4     There are 2 goals:
5
6     1) Create simulate fake traffic to confuse a would-be attacker and
7         obfuscate my real devices.
8     2) Allow the attacker to interact with my fake devices.
9         The attacker should think that he's turning the lights
10        on and off, etc.
11 """
12 """
13
14 import argparse
15 import os
16 from rfcatmod import *
17
18
19 def parseargs():
20     """
21     :return: the arguments.
22     """
23     parser = argparse.ArgumentParser(description='Create simulated Insteon network
24         traffic.')
25     #
26     parser.add_argument('-i', '--interface', help='USB interface', default=0)
27     parser.add_argument('-f', '--filename', help='the filename that contains the
28         simulated commands.', default="simtraffic.txt")
29     return parser.parse_args()
30
31 def keystop(delay=0):
32     """
33     the <enter> pressed function.
34     :param delay: time to wait before registering the command?
35     :return: the length of the command. Not sure why...
36     """
37     return len(select.select([sys.stdin], [], [], delay)[0])
38
39
40 def genhoneytraffic(inputfilename, interface):
41     packets = list()
42     waittimes = list()
43     try:
44         inputfile = open(inputfilename, "r")
45     except IOError:
46         print "Input file does not exist.
47             Create simtraffic.txt file and fill with fake traffic."
48         sys.exit(0)
49
50     for line in inputfile:
51         # the line is the fake packet.
52         # Strip the end line so the next line is the packet I care about...
53         packets.append(line.strip("\n"))
54
55         # the next line is the amount of time to wait to
56         # send the next bit of traffic.
57         waittimes.append(int(next(inputfile).strip("\n")))
58
59     print "Send simulated traffic. Check WireShark for output. Press [enter] to stop."
```

```

60     #while not keystop():
61     try:
62         for x in range(0, len(packets)):
63             command = "./insteonsend.py -i " + str(interface) + " -r " + packets[x]
64             print command
65             os.system(command)
66             time.sleep(waittimes[x])
67
68     except KeyboardInterrupt:
69         print "press [enter] to stop."
70
71     except ChipconUsbTimeoutException:
72         print "ChipconUsbTimeoutException"
73
74 def main():
75     args = parseargs()
76     inputfilename = args.filename
77     interface = int(args.interface)
78     # print inputfilename
79     # print interface
80     if inputfilename is None:
81         print "must specify an input file name"
82         sys.exit(0)
83
84     genhoneytraffic(inputfilename, interface)
85
86
87 if __name__ == "__main__":
88     main()
89     exit(0)

```

A.14 1honeypotntwk.txt

```
1 Device
2 A1 A1 A1
3 01
4 3A
5 48
6 00
7 isresponderyes
8 1
9 B1 B1 B1
10 iscontrollerno
11 Device
12 B1 B1 B1
13 03
14 01
15 01
16 00
17 isresponderyes
18 2
19 F1 F1 F1
20 A1 A1 A1
21 iscontrolleryes
22 5
23 A1 A1 A1
24 C1 C1 C1
25 D1 D1 D1
26 E1 E1 E1
27 F1 F1 F1
28 Device
29 C1 C1 C1
30 05
31 0B
32 0A
33 00
34 isresponderyes
35 1
36 B1 B1 B1
37 iscontrollerno
38 e0
39 2A
40 False
41 False
42 False
43 48
44 46
45 Device
46 D1 D1 D1
47 01
48 3A
49 48
50 00
51 isresponderyes
52 1
53 B1 B1 B1
54 iscontrollerno
55 Device
56 E1 E1 E1
57 01
58 3A
59 48
60 00
61 isresponderyes
```

62 1
63 B1 B1 B1
64 iscontrollerno
65 Device
66 F1 F1 F1
67 OF
68 3A
69 48
70 00
71 isresponderyes
72 1
73 B1 B1 B1
74 iscontrollerno

A.15 1simtraffic.txt

```
1 1F : C1 C1 C1 : B1 B1 B1 : 2E 02 00 00 00 00 00 00 00 00 00 00 00 00 00 92 96
2 2
3 OF : A1 A1 A1 : B1 B1 B1 : 19 02
4 1
5 OF : D1 D1 D1 : B1 B1 B1 : 19 02
6 1
7 OF : E1 E1 E1 : B1 B1 B1 : 19 02
8 1
9 OF : F1 F1 F1 : B1 B1 B1 : 19 02
10 1
```

A.16 Insteon-dissector.lua

```
1 --script-name: Insteon-dissctor.lua
2 -- author: Caleb Mays
3 -----
4 -- creates a Proto object, but doesn't register it yet
5 local insteon = Proto("Insteon","Insteon")
6
7 -- start declaring some fields.
8 -- first, the flags.
9 local pf_flags = ProtoField.new ("Flags", "insteon.flags", ftypes.UINT8,
10     nil, base.HEX)
11
12 local pf_flag_broadcast = ProtoField.new ("Broadcast/NAK",
13     "insteon.flags.broadcast", ftypes.BOOLEAN, {"this is a broadcast",
14     "this is not a broadcast"}, 8, 0x80, "is this a broadcast?")
15
16 local pf_flag_group = ProtoField.new ("Group/Not-Group", "insteon.flags.group",
17     ftypes.BOOLEAN, {"this is a group message", "this is not a group message"},
18     8, 0x40, "is this a group msg?")
19
20 local pf_flag_acknowledge = ProtoField.new ("Acknowledge",
21     "insteon.flags.acknowledge", ftypes.BOOLEAN,
22     {"acknowledgement", "not an acknowledgement"}, 8, 0x20,
23     "is this an acknowledgement?")
24
25 local pf_flag_extended = ProtoField.new ("Message Type", "insteon.flags.extended",
26     ftypes.BOOLEAN, {"Extended Message", "Standard Message"}, 8, 0x10,
27     "is this a standard message or extended messages?")
28
29 local pf_flag_hopsleft = ProtoField.new ("Hops Left", "insteon.flags.hopsleft",
30     ftypes.UINT8, nil, base.DEC, 0x0C, "hops left")
31
32 local pf_flag_maxhops = ProtoField.new ("Max Hops", "insteon.flags.hopsleft",
33     ftypes.UINT8, nil, base.DEC, 0x03, "max hops")
34
35 -- now the other parts of the message.
36 local pf_src_addr = ProtoField.new ("Source", "insteon.source", ftypes.BYTES, nil,
37     base.NONE) -- consider making these bytes into strings
38
39 local pf_dst_addr = ProtoField.new ("Destination", "insteon.destination",
40     ftypes.BYTES, nil, base.NONE) -- consider making these bytes into strings
41
42 local pf_cmd1 = ProtoField.new ("Command 1", "insteon.cmd1", ftypes.BYTES,
43     nil, base.NONE)
44 local pf_cmd2 = ProtoField.new ("Command 2", "insteon.cmd2", ftypes.BYTES,
45     nil, base.NONE)
46 local pf_crc = ProtoField.new ("CRC", "insteon.crc", ftypes.BYTES,
47     nil, base.NONE)
48
49 -- this one is only used for extended messages.
50 local pf_usr_data = ProtoField.new ("Data", "insteon.data", ftypes.BYTES,
51     nil, base.NONE)
52
53 -- a few others for special things:
54 local pf_firmware = ProtoField.new ("Firmware", "insteon.firmware",
55     ftypes.BYTES, nil, base.NONE)
56 local pf_devType = ProtoField.new ("Device Type", "insteon.device_type",
57     ftypes.BYTES, nil, base.NONE)
58 local pf_devSubType = ProtoField.new ("Device Subtype", "insteon.device_subtype",
59     ftypes.BYTES, nil, base.NONE)
60
61 -- must explicitly tell the insteon.fields what all of its fields are.
```

```

62 insteon.fields = {pf_flags, pf_flag_broadcast, pf_flag_group,
63 pf_flag_acknowledge, pf_flag_extended,
64 pf_flag_hopsleft, pf_flag_maxhops, pf_src_addr,
65 pf_dst_addr, pf_cmd1, pf_cmd2, pf_crc, pf_usr_data, pf_firmware, pf_devType,
66 pf_devSubType}
67
68
69 -- time to get some "fields". This allows me to do different
70 -- things based on the flags, etc.
71 local broadcast_field = Field.new("insteon.flags.broadcast")
72 local group_field = Field.new("insteon.flags.group")
73 local acknowledgement_field = Field.new("insteon.flags.acknowledge")
74 local extended_field = Field.new("insteon.flags.extended")
75
76 -- some helper functions to get those fields.
77 local function isBroadcast() return broadcast_field()() end
78 local function isGroupMsg() return group_field()() end
79 local function isAcknowledgement() return acknowledgement_field()() end
80 local function isExtended() return extended_field()() end
81
82
83 -- this is basically like the main() of the program.
84 function insteon.dissector(tvbuf,pktinfo,root)
85     pktinfo.cols.protocol:set("Insteon")
86
87     local pktlen = tvbuf:reported_length_remaining()
88     local tree = root:add(insteon, tvbuf:range(0, pktlen))
89
90     -- flag information tree.
91     local flagrange = tvbuf:range(0,1)
92     local flag_tree = tree:add(pf_flags, flagrange)
93
94     -- adding the individual bits to the tree.
95     local firstbit_flag_tree = flag_tree:add(pf_flag_broadcast, flagrange)
96     local secondbit_flag_tree = flag_tree:add(pf_flag_group, flagrange)
97     local thirdbit_flag_tree = flag_tree:add(pf_flag_acknowledge, flagrange)
98     local fourth_flag_tree = flag_tree:add(pf_flag_extended, flagrange)
99     local fiftsixth_flag_tree = flag_tree:add(pf_flag_hopsleft, flagrange)
100    local seveneight_flag_tree = flag_tree:add(pf_flag_maxhops, flagrange)
101
102    -- need to do some different things if it's a broadcast_field, a group_field, etc.
103
104    -- 000 -- Direct Message
105    if not isBroadcast() and not isGroupMsg() and not isAcknowledgement() then
106        tree:add(pf_dst_addr, tvbuf:range(1,3))
107        local destination_address = tvbuf:range(1, 3)
108        tree:add(pf_src_addr, tvbuf:range(4,3))
109        local source_address = tvbuf:range(4, 3)
110        pktinfo.cols.src:set(tostring(source_address)) -- sets the src column
111        pktinfo.cols.dst:set(tostring(destination_address)) -- sets the dst column
112        pktinfo.cols.info:set("Direct Message") -- sets the information column.
113
114    -- 010
115    elseif not isBroadcast() and isGroupMsg() and not isAcknowledgement() then
116        tree:add(pf_dst_addr, tvbuf:range(1,3))
117        local destination_address = tvbuf:range(1, 3)
118        tree:add(pf_src_addr, tvbuf:range(4,3))
119        local source_address = tvbuf:range(4, 3)
120        pktinfo.cols.src:set(tostring(source_address)) -- sets the src column
121        pktinfo.cols.dst:set(tostring(destination_address)) -- sets the dst column
122        pktinfo.cols.info:set("Group Cleanup Direct Message")
123
124    -- 011
125    elseif not isBroadcast() and isGroupMsg() and isAcknowledgement() then
126        tree:add(pf_dst_addr, tvbuf:range(1,3))
127        local destination_address = tvbuf:range(1, 3)

```

```

128     tree:add(pf_src_addr, tvbbuf:range(4,3))
129     local source_address = tvbbuf:range(4, 3)
130     pktinfo.cols.src:set(tostring(source_address)) -- sets the src column
131     pktinfo.cols.dst:set(tostring(destination_address)) -- sets the dst column
132     pktinfo.cols.info:set("ACK of Group Cleanup Direct Message")
133
134 -- 100 -- Broadcasts include device category, subcategory and firmware
135 elseif isBroadcast() and not isGroupMsg() and not isAcknowledgement() then
136     pktinfo.cols.info:set("Broadcast Message")
137     tree:add(pf_src_addr, tvbbuf:range(1,3))
138     local source_address = tvbbuf:range(1, 3)
139     pktinfo.cols.src:set(tostring(source_address)) -- sets the src column
140     -- get firmware
141     tree:add(pf_firmware, tvbbuf:range(4,1))
142     local firmware = tvbbuf:range(4, 1)
143
144     -- get device sub type
145     tree:add(pf_devSubType, tvbbuf:range(5,1))
146     local subType = tvbbuf:range(5, 1)
147
148     -- get device type
149     tree:add(pf_devType, tvbbuf:range(6,1))
150     local subType = tvbbuf:range(6, 1)
151
152     -- too weird to put this info in the dest column.
153     -- tell user to look at the info pane.
154     pktinfo.cols.dst:set("Firmware & Dev Type") -- sets the dst column
155
156 -- 101
157 elseif isBroadcast() and not isGroupMsg() and isAcknowledgement() then
158     tree:add(pf_dst_addr, tvbbuf:range(1,3))
159     local destination_address = tvbbuf:range(1, 3)
160     tree:add(pf_src_addr, tvbbuf:range(4,3))
161     local source_address = tvbbuf:range(4, 3)
162     pktinfo.cols.src:set(tostring(source_address)) -- sets the src column
163     pktinfo.cols.dst:set(tostring(destination_address)) -- sets the dst column
164     pktinfo.cols.info:set("NAK of Direct Message")
165
166 -- 110
167 elseif isBroadcast and isGroupMsg and not isAcknowledgement() then
168     tree:add(pf_src_addr, tvbbuf:range(1,3))
169     local source_address = tvbbuf:range(1, 3)
170     --For Group Broadcast messages Destination Address is:
171     --Group Number [0 - 255]
172     tree:add(pf_dst_addr, tvbbuf:range(4,1))
173     local destination_address = tvbbuf:range(4, 1)
174     pktinfo.cols.src:set(tostring(source_address)) -- sets the src column
175     pktinfo.cols.dst:set(tostring("Group: "..destination_address))
176     -- sets the dst column
177     pktinfo.cols.info:set("Group Broadcast Message")
178
179 -- 111
180 elseif isBroadcast() and isGroupMsg() and isAcknowledgement() then
181     -- a "NAK of Group Cleanup Direct Message"
182     --tree:add(pf_src_addr, tvbbuf:range(1,3))
183     --local source_address = tvbbuf:range(1, 3)
184     --tree:add(pf_dst_addr, tvbbuf:range(4,3))
185     --local destination_address = tvbbuf:range(4, 3)
186     --pktinfo.cols.info:set("NAK of Group Cleanup Direct Message")
187 end
188
189 -- command 1 is 1 byte
190 tree:add(pf_cmd1, tvbbuf:range(7,1))
191 local command1 = tvbbuf:range(7, 1)
192
193 -- command 2 is 1 byte

```

```

194     tree:add(pf_cmd2, tvbbuf:range(8,1))
195     local destination_address = tvbbuf:range(8,1)
196
197     -- if not extended then 1 byte CRC
198     if not isExtended() then
199         tree:add(pf_crc, tvbbuf:range(9,1))
200         local CRC = tvbbuf:range(9,1)
201         -- TODO: consider adding into the info portion the fact
202         -- that it is an standard message.
203         -- otherwise, extended data contains 14 bytes user data.
204     else
205         tree:add(pf_usr_data, tvbbuf:range(9,14))
206         local user_data = tvbbuf:range(9,14)
207         -- meaning the CRC is the 23rd (24th) byte.
208         tree:add(pf_crc, tvbbuf:range(23,1))
209         local CRC = tvbbuf:range(23,1)
210     end
211 end

```

Bibliography

1. A. Baviskar, J. Baviskar, S. Wagh, A. Mulla and P. Dave, Comparative study of communication technologies for power optimized automation systems: A review and implementation, *Proceedings of the Fifth International Conference on Communication Systems and Network Technologies*, pp. 375–380, 2015.
2. Bitbucket, RfCat (bitbucket.org/atlas0fd00m/rfcat), 2016.
3. M. Bowden, *Worm: The First Digital World War*, Atlantic Monthly Press, New York, 2011.
4. C-SPAN, Cybersecurity Threats, Washington, DC (c-span.org/video/?322853-1/hearing-cybersecurity-threats), 2014.
5. P. Darbee, INSTEON Developer’s Guide (2nd Edition), SmartLabs Technology, Irvine, California (cache.insteon.com/pdf/INSTEON_Developers_Guide_20070816a.pdf), 2007.
6. P. Darbee, INSTEON Command Tables, SmartLabs Technology, Irvine, California (cache.insteon.com/pdf/INSTEON_Command_Tables_20070925a.pdf), 2007.
7. P. Darbee, INSTEON Device Categories and Product Keys, SmartLabs Technology, Irvine, California (cache.insteon.com/pdf/INSTEON_DevCats_and_Product_Keys_20081008.pdf), 2008.
8. Federal Communications Commission. Software Defined Radio, Washington D.C. (fcc.gov/general/software-defined-radio), 2016.
9. Gartner, Gartner Says 6.4 Billion Connected “Things” Will Be In Use In 2016, Up 30 Percent From 2015, Stamford, Connecticut (gartner.com/newsroom/id/3165317), 2015.
10. K. Girtz, B. Mullins, M. Rice and J. Lopez, Practical application layer emulation in ICS honeypots, in *International Journal of Critical Infrastructure Protection X*, M. Rice and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 83–98, 2016.
11. C. Gomez and J. Paradells, Wireless home automation networks: A survey or architectures and technologies, *IEEE Communications Magazine*, vol. 48(6), pp. 92–101, 2010.
12. Great Scott Gadgets, YARD Stick One, Evergreen, Colorado, (greatscottgadgets.com/yardstickone).

13. J. Hall, A Practical Wireless Exploitation Framework for Z-Wave Networks, M.S. Thesis, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2016.
14. C. Mays and B. Ramsey, INSTEON, inste-off, inste-open? presented at *The Wireless Village at DEFCON 24*, 2016.
15. B. Ramsey, Improved Wireless Security through Physical Layer Protocol Manipulation and Radio Frequency Fingerprinting, Ph.D. Dissertation, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2014.
16. Revention, Revention integrates with Insteon to provide automated light fixtures, Houston, Texas (revention.com/news/articles/Revention-Integrates-with-Insteon-to-Provide-Automated-Light-Fixtures), 2016.
17. A. Rose and B. Ramsey, Picking Bluetooth locks from a quarter mile away, presented at *DEFCON 24*, 2016.
18. B. Schneier, Real-world security and the Internet of things, *Schneier On Security* (schneier.com/blog/archives/2016/07/real-world_secu.html), July 28, 2016.
19. P. Shipley and R. Gooler, Insteon: False security and deceptive documentation, presented at *DEFCON 23*, 2015.
20. SmartLabs Technology, INSTEON Whitepaper: The Details, Irvine, California (cache.insteon.com/documentation/insteon_details.pdf), 2013.
21. C. Talbot, T. Carbino and M. Temple, Securing INSTEON home automation systems using radio frequency distinct native attribute (RF-DNA) fingerprints, *Proceedings of the 12th International Conference on Cyber Warfare and Security (ICCWS)*, Dayton, Ohio, 2017 [to appear].
22. Transparency Market Research, Global Home Automation Market Will Be Worth \$21.6 Billion By 2020, Albany, New York (transparencymarketresearch.com/pressrelease/home-automation-market.htm), 2015.
23. Universal Devices Forum, How to receiver, send, log, spoof all Insteon RF traffic, Encino, California (forum.universal-devices.com/topic/17474-how-to-receive-send-log-spoof-all-insteon-rf-traffic/), 2015.
24. “RPGillespie”, Generate a quick and easy custom pcap file using Python, Code Project: For those who code, (codeproject.com/Tips/612847/Generate-a-quick-and-easy-custom-pcap-file-using-P), 2013.

25. N. Provos and T. Hotz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*, Addison-Wesley, Boston, Massachusetts, 2007.
26. L. Rist, D. Haslinger, J. Smith, J. Vestergaard and A. Pasquale, CONPOT ICS/SCADA Honeypot, Honeynet Project (conpot.org), 2013.
27. J. Voas, Network of ‘Things’, NIST Special Publication 800-183, National Institute of Standards and Technology, Gaithersburg, Maryland, 2016.
28. M. Winn, M. Rice, S. Dunlap, J. Lopez and B. Mullins, Constructing cost-effective and targetable industrial control system honeypots for production networks, *International Journal of Critical Infrastructure Protection*, vol. 10, pp. 47–58, 2015.
29. Wireshark, Lua, (wiki.wireshark.org/Lua), 2015.
30. C. Withanage, R. Ashok, C. Yuen and K. Otto, A comparison of the popular home automation technologies, *Proceedings of the 2014 IEEE Innovative Smart Grid Technologies - Asia*, pp. 600–605, 2014.

REPORT DOCUMENTATION PAGE					<i>Form Approved</i> OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 23-03-2017		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From — To) Oct 2015 — Mar 2017		
4. TITLE AND SUBTITLE Constructing Honeypots to Defend Building Automation Systems				5a. CONTRACT NUMBER		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Mays, Caleb E., Capt, USAF				5d. PROJECT NUMBER 17G310		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-MS-17-M-049		
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of Homeland Security ICS-CERT POC: Neil Hershfeld, DHS ICS-CERT Technical Lead ATTN: NPPD/CSC/NCSD/US-CERT Mailstop:0635 245 Murray Lane, SW, Bldg 410, Washington, DC 20528 Email: ics-cert@dhs.gov phone:1-877-776-7585				10. SPONSOR/MONITOR'S ACRONYM(S) DHS ICS-CERT		
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.						
13. SUPPLEMENTARY NOTES This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.						
14. ABSTRACT The Internet of Things and building automation technology are growing fields. Many automation networks use proprietary protocols though few works evaluate the security of these protocols. This lack of security research leaves consumers' homes and buildings vulnerable to intruders. Concurrently, Software-Defined Radios (SDRs) are experiencing significant growth. Despite this growth, research evaluating the security of building automation networks is lacking. The nation's critical infrastructure is at risk and Industrial Control Systems (ICS) are getting focused research attention due to their general lack of network security and the important systems these networks control. Recent research details the development of ICS honeypots, as using honeypots is a proven method in the Information Technology (IT) environment to assist network defenders. Given the current state of cyber-security, research topics for building automation security and ICS security are converging. This research improves on previous research to improve the packet capture rate of the INSTEON protocol from almost 40% to nearly 75% and creates a honeypot to defend building automation systems.						
15. SUBJECT TERMS Internet of Things, Honeypot, Building Automation Systems, Industrial Control Systems, Software-Defined Radio, INSTEON						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES 134	19a. NAME OF RESPONSIBLE PERSON Dr. Barry Mullins, AFIT/ENG	
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (include area code) (937) 255-3636, x7979; barry.mullins@afit.edu	